Cse 237c Project2

Yiming Ren

Y5ren@ucsd.edu

Question 1:

In my basic architecture, I use different loop condition with the book. As we discussed in the book and lecture, we are supposed to know the number of iterations in advance. However, for the implementation of converting Cartesian to Polar system, I think we don't need to know the number of iterations at the beginning. Instead, when we rotating the vector to the x-axis, we can stop it immediately when the absolute value of y is less than the least element in the 'angles' array. In other words, since $\sin(\theta) \rightarrow \theta$ when $\theta \rightarrow 0$, we can directly compare |y| (which is equal to $\sin(\theta)$) to θ when θ is trivial, and stop the rotation if |y| is even smaller than θ. This could save a lot of time without huge impact on the precision. So, in order to analyze influence of the number of rotations, I first find the number of rotations in my program by printing them on the simulation output.

```
15 rotations ran for this Test: x=0.8147, y=0.1269, golden theta=0.1545, golden r=0.8245, your theta=0.154
16 rotations ran for this Test: x=0.6323, y=-0.2785, golden theta=-0.4149, golden r=0.6909, your theta=-0.
12 rotations ran for this Test: x=-0.5469, y=-0.9575, golden theta=-2.0898, golden r=1.1027, your theta=-2
14 rotations ran for this Test: x=-0.4854, y=0.7003, golden theta=2.1769, golden r=0.8521, your theta=2.17
```

Now I can conclude that, these 4 numbers reach a limitation of ignorable tolerance, below which the precision would be worse, and exceeding the limitation would decrease the performance. To verify my conclusion, I first set the number of rotations to 16 for all tests. The modified result is at right.

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	2	73	3719	-
FIFO	-	-	-	-	-
Instance	-	110	9449	13301	-
Memory	8	-	277	868	-
Multiplexer	-	-	-	564	-
Register	-	-	2497	-	-
Total	8	112	12296	18452	0
Available	280	220	106400	53200	0
Utilization (%)	2	50	11	34	0

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	2	73	3734	-
FIFO	-	-	-	-	-
Instance	-	110	9449	13301	-
Memory	8	-	277	868	-
Multiplexer	-	-	-	619	-
Register	-	-	2552	-	-
Total	8	112	12351	18522	0
Available	280	220	106400	53200	0
Utilization (%)	2	50	11	34	0

From the contrast we can see, my original implementation slightly reduces the stress of hardware, using less FFs and LUT. Also, though the frequency shown in the synthesis solution are similar, running 16 full rotation could spend more time. The difference is just too small to be detected.

Then, I decrease the number by 1 each time until it is 14. At this time, the precision goes down. (See the difference between golden θ and your θ in test1, 2 and 3)

```
 golden theta=0.1545, golden r=0.8245, your theta=0.1544, your r=0.8245
, golden theta=-0.4149, golden r=0.6909, your theta=-0.4150, your r=0.6909
5, golden theta=-2.0898, golden r=1.1027, your theta=-2.0899, your r=1.1027
, golden theta=2.1769, golden r=0.8521, your theta=2.1769, your r=0.8521
```

Nevertheless, although the result generated by 15 rotation looks the same as that by 16 rotation, 16 rotation gives more precision if we show the result in more digits. If we keep increasing the number of rotations to infinity, we would finally get the exact value of polar coordinates.

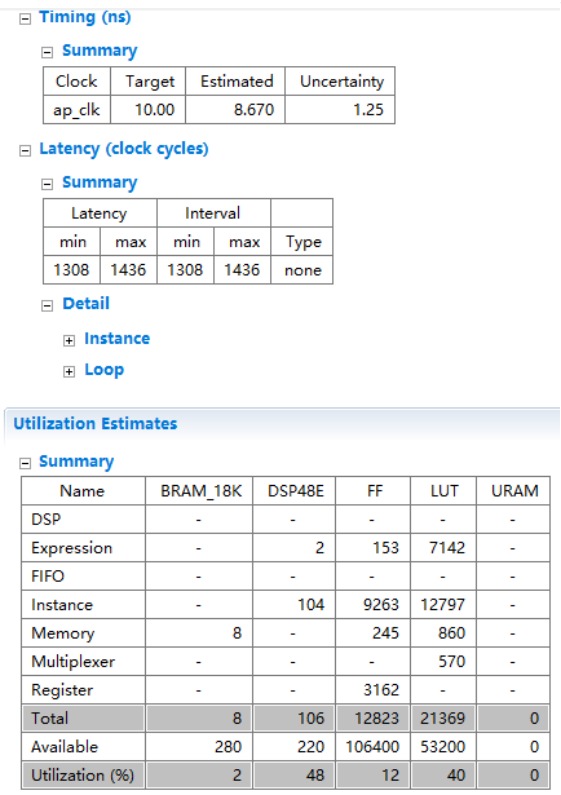
Question 2:

Below is the result of tests of using 32 bitwidth with 26 fraction bits fixed point type on input ‘x’ and ‘y’. As fixed-point data type applied, I can remove all the multiplications and use shifts in the rotation to optimize performance. In order to keep the input and output type for Pynq board interface, I only temporarily change the input data type, and then cast it back to ‘float’ after shifting. The accuracy is below. I also tested the fixed-point data type with different bitwidth. Here are some results needed for attention.

```
golden theta=0.1545, golden r=0.8245, your theta=0.1543, your r=0.8247
 golden theta=-0.4149, golden r=0.6909, your theta=-0.4151, your r=0.6919
, golden theta=-2.0898, golden r=1.1027, your theta=-2.0902, your r=1.1037
 golden theta=2.1769, golden r=0.8521, your theta=2.1766, your r=0.8531
```

18 bitwidth with 12 fraction bits fixed-point

As stated above, this image shows the accuracy of 18 bitwidth with 12 fraction bits fixed-point data type. The fixed-point data type with 18 bitwidth is a disaster for the accuracy, but at least it passes the tests, while 16 bitwidth fixed-point data type leads to failure on tests. By contrast, the 32 bitwidth fixed-point data type gives a nicely precise result. Therefore, when using fixed-point type for variables, we should keep the bitwidth as long as possible to pursue accuracy.



Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.670	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1308	1436	1308	1436	none

Detail

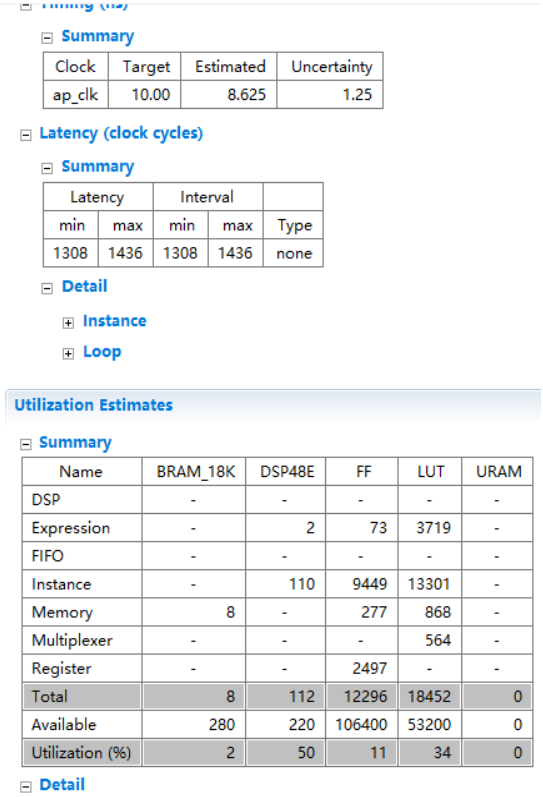
Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	2	153	7142	-
FIFO	-	-	-	-	-
Instance	-	104	9263	12797	-
Memory	8	-	245	860	-
Multiplexer	-	-	-	570	-
Register	-	-	3162	-	-
Total	8	106	12823	21369	0
Available	280	220	106400	53200	0
Utilization (%)	2	48	12	40	0



Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.625	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1308	1436	1308	1436	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	2	73	3719	-
FIFO	-	-	-	-	-
Instance	-	110	9449	13301	-
Memory	8	-	277	868	-
Multiplexer	-	-	-	564	-
Register	-	-	2497	-	-
Total	8	112	12296	18452	0
Available	280	220	106400	53200	0
Utilization (%)	2	50	11	34	0

Detail

32 bitwidth fixed-point data type
0.0803Mhz

Original float data type
0.0807Mhz

If we take a look at the performance difference between 2 data types, we can see an interesting fact. The 32 bitwidth fixed-point data type is less performant

and resource saving than the float data type. The explanation is that since I have to keep the data type of input and output, I have to use several type cast operators in the function. The cost of those operators is potentially more than the benefit of using fixed-point data type. If I can get rid of the Pynq board and only generate synthesis on HLS, I could directly change the definition of 'data_t' to fixed_point, therefore getting a better performance and resources usage.

Back to the question, we should better keep the variables all in a uniform type since we want to avoid costly type casts. Moreover, they'd better in a same type as the input variable. Based on these, the designer could try using a longer data type, as the growth of precision is huge compared to the decay of performance.

Question 3:

The optimization for this question is based on the optimization from last question. The 32 wide and 26 fraction bits ap_fixed type for inputs is inherited. I then modified the type of 'Kvalues' array in its declaration to the same type to make sure the operations between it and inputs are valid. Also, I changed the shifts operations to multiplications. The new program provides the same accuracy as simple operations. The only difference is captured:

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	10	219	7752	-
FIFO	-	-	-	-	-
Instance	-	104	9263	12797	-
Memory	8	-	272	867	-
Multiplexer	-	-	-	570	-
Register	-	-	3232	-	-
Total	8	114	12986	21986	0
Available	280	220	106400	53200	0
Utilization (%)	2	51	12	41	0

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	2	153	7142	-
FIFO	-	-	-	-	-
Instance	-	104	9263	12797	-
Memory	8	-	245	860	-
Multiplexer	-	-	-	570	-
Register	-	-	3162	-	-
Total	8	106	12823	21369	0
Available	280	220	106400	53200	0
Utilization (%)	2	48	12	40	0

Left image is the recourses usage of multiplication and right image is the recourses usage of simple operations. Although they have same performance regarding the clock cycle and period in the synthesis, theoretically, the simple operations should have a better performance. The superiority is just too small to be shown in the synthesis result. On the other hand, multiplications obviously use more resources. Specifically, they use more FFs and LUTs. The multiplications indeed need more resources than simple operations, but another reason is that 'Kvalues' array is accessed using multiplications. If I only use simple operations, I can shift the 'x' and 'y' by 'i' digits. 'i' is the index of the iteration currently running. I can avoid accessing 'Kvalues' doing like this. However, if multiplications are applied, I have to use the decimals in 'Kvalues' array because they are hard to generate. This requires more registers and LUTs.

Question 4:

Since my baseline code does not include the ternary operator. I have to modify my algorithm to insert it. Fortunately, my implementation still works after removing conditional operators and inserting ternary operator. Instead of using 2 formulas for 'right rotation' and 'left rotation', I can use the parameter, sigma, of ternary operator to determine if the coordinates are added to or taken from each other, by assign sigma -1 if right rotation and 1 if left rotation. This change has no impact on the precision and performance, but conditional operator wins in details.

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	2	73	3686	-
FIFO	-	-	-	-	-
Instance	-	113	9592	13622	-
Memory	8	-	277	868	-
Multiplexer	-	-	-	609	-
Register	-	-	2529	-	-
Total	8	115	12471	18785	0
Available	280	220	106400	53200	0
Utilization (%)	2	52	11	35	0

Ternary operator

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	2	73	3719	-
FIFO	-	-	-	-	-
Instance	-	110	9449	13301	-
Memory	8	-	277	868	-
Multiplexer	-	-	-	564	-
Register	-	-	2497	-	-
Total	8	112	12296	18452	0
Available	280	220	106400	53200	0
Utilization (%)	2	50	11	34	0

conditional operator

The reason of that ternary operator uses more resources is evident. The original algorithm clearly divides the calculation methods for right rotation and left rotation. For example, the formula of generating new 'x' is 'x = x + y * Kvalues[i]' for right rotation and 'x = x – y * Kvalues[i]' for left rotation. After simplifying the algorithm, we get 'x = x – sigma * y * Kvalues[i]' for both directions. This leads to one more multiplication operation in the formula. It turns out the high cost of multiplication.

Question 5:

I gradually increase the MAN_BITS from 2 up to 11 above which the simulation would end with compilation error. Meanwhile, as the increment of MAN_BITS, the accuracy is also increasing. When the size of LUT is 2^8, which means MAN_BITS is 2, the simulation only goes through few cases and ends with a low RMSE error value:

```
RMSE(R)           RMSE(Theta)
0.000100615427073 0.087266594171524
```

While MAN_BITS is 11, and size of LUT is 2^26, the simulation goes through far more cases than the previous:

```
RMSE(R)           RMSE(Theta)
0.474436134099960 0.987319469451904
```

However, this LUT size is so huge that the resources are over used. Specifically, the amount of BRAM does not support such overwhelming memory usage. This would have bad impact on the performance.

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	3164	-
FIFO	-	-	-	-	-
Instance	-	0	200	276	-
Memory	262144	-	128	0	0
Multiplexer	-	-	-	47	-
Register	-	-	943	-	-
Total	262144	0	1271	3487	0
Available	280	220	106400	53200	0
Utilization (%)	93622	0	1	6	0

So to obtain a healthy environment, I gradually decrease the MAN_BITS to 6.

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	3134	-
FIFO	-	-	-	-	-
Instance	-	0	200	276	-
Memory	256	-	0	0	0
Multiplexer	-	-	-	41	-
Register	-	-	931	-	-
Total	256	0	1131	3451	0
Available	280	220	106400	53200	0
Utilization (%)	91	0	1	6	0

Since we want the RMSE error value as high as possible without exceeding the limit, MAN_BITS = 6 is the best choice for best accuracy. Now that the best solution of LUT size is found, further changes are only based on the bitwidth.

The precision of MAN_BITS = 6 will be used for the best accuracy:

```
   RMSE(R)           RMSE(Theta)
0.000100088444015 0.031416587531567
```

First, I double the bitwidth of fixed representation to 64. The doubled bitwidth costs more and is less performant as expected, compared to 32 bitwidth.

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.720	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
5	5	5	5	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	2810	-
FIFO	-	-	-	-	-
Instance	-	0	200	276	-
Memory	256	-	0	0	0
Multiplexer	-	-	-	38	-
Register	-	-	742	-	-
Total	256	0	942	3124	0
Available	280	220	106400	53200	0
Utilization (%)	91	0	~0	5	0

32 bitwidth

Frequency = 22.9 MHz

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.554	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
6	6	6	6	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	3830	-
FIFO	-	-	-	-	-
Instance	-	0	200	276	-
Memory	256	-	0	0	0
Multiplexer	-	-	-	41	-
Register	-	-	1105	-	-
Total	256	0	1305	4147	0
Available	280	220	106400	53200	0
Utilization (%)	91	0	1	7	0

64 bitwidth

frequency = 19.5 MHz

This makes sense because the input type, float, contains 32 bits. 32 bits fixed-point variable is enough to represent the input. So 64 bits fixed-point variable not only spend extra spaces for nonsense, but also waste time to be handled.

Now if I decrease the bitwidth down to 8, the performance and the resources usage are both improved. From the below synthesis result, I believe that although decreasing bitwidth down to 8 does not contribute a lot, but it does give a better performance and resources usage without hurting the accuracy. As the decrease of bitwidth, we get a better synthesis result.

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.670	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
5	5	5	5	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	2686	-
FIFO	-	-	-	-	-
Instance	-	0	200	276	-
Memory	256	-	0	0	0
Multiplexer	-	-	-	38	-
Register	-	-	680	-	-
Total	256	0	880	3000	0
Available	280	220	106400	53200	0
Utilization (%)	91	0	~0	5	0

8 bitwidth

Frequency = 23.1 MHz

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.720	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
5	5	5	5	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	2810	-
FIFO	-	-	-	-	-
Instance	-	0	200	276	-
Memory	256	-	0	0	0
Multiplexer	-	-	-	38	-
Register	-	-	742	-	-
Total	256	0	942	3124	0
Available	280	220	106400	53200	0
Utilization (%)	91	0	~0	5	0

32 bitwidth

Frequency = 22.9 MHz

In all, the accuracy, resources usage, and performance do have a subtle relationship. If we pursue accuracy, we might need to sacrifice a little resources usage and performance. The better performance and resources usage may also need a higher tolerance on accuracy. We need to find a balance between the tradeoff of these elements.

Compared to regular CORDIC, CORDIC LUT uses more BRAM memory to store LUTs, but asks for less registers and clock cycles to do calculations. The CORDIC LUT is a better choice for a machine with a larger memory size.