Project: Fast Fourier Transform (FFT)

BASELINE

Date: Wed Dec 1 11:22:32 2021
Version: 2021.1 (Build 3247384 on Thu Jun 10 19:36:33 MDT 2021)
Project: Project4_FFT_Stages
Solution: solution1 (Vivado IP Flow Target)
Product family: zynq
Target device: xc7z020-clg400-1

Timing Estimate

Target	Estimated	Uncertainty
10.00 ns	7.256 ns	2.70 ns

Performance & Resource Estimates

Modules Loops

Modules && Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
fft				-	142345	1.423E6	-	70914	-	dataflow	90	234	25624	38098	0
fft_stage_first				-	529	5.290E3	-	529	-	no	0	18	2302	3206	0
fft_stages4				-	555	5.550E3	-	555	-	no	2	24	2592	3852	0
fft_stages5				-	1109	1.109E4	-	1109	-	no	2	24	2594	3854	0
fft_stages6				-	2217	2.217E4	-	2217	-	no	2	24	2596	3856	0
fft_stages7				-	4433	4.433E4	-	4433	-	no	2	24	2598	3857	0
fft_stages8				-	8865	8.865E4	-	8865	-	no	2	24	2600	3858	0
fft_stages9				-	17729	1.770E5	-	17729	-	no	2	24	2602	3859	0
fft_stages10				-	35457	3.550E5	-	35457	-	no	2	24	2604	3860	0
fft_stages11				-	70913	7.090E5	-	70913	-	no	2	24	2606	3859	0
fft_stage_last				-	529	5.290E3	-	529	-	no	2	24	2512	3765	0

INFO: [SIM 2] ************* CSIM start ******************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
Compiling ../../../../../../../../Desktop/WES237C_Fall2021/Read_the_docs-master/project_files/project4/FFT/HLS/2_Skeleton_Restructured/fft_test.cpp in debug mode
Compiling ../../../../../../../../Desktop/WES237C_Fall2021/Read_the_docs-master/project_files/project4/FFT/HLS/2_Skeleton_Restructured/fft.cpp in debug mode
Generating csim.exe

RMSE(R) RMSE(I)
8742.694335937500000 9995.070312500000000

****
FAIL: Output DOES NOT match the golden output
****
ERR: [SIM 100] 'csim_design' failed: nonzero return value.
INFO: [SIM 3] ************ CSIM finish ******************

Question 1: What changes would this code require if you were to use a custom CORDIC similar to what you designed for Project: CORDIC? Compared to a baseline code with HLS math functions for cos() and sin(), would changing the accuracy of your CORDIC core make the DFT hardware resource usage change? How would it affect the performance? Note that you do not need to implement the CORDIC in your code, we are just asking you to discuss potential tradeoffs that would be possible if you used a CORDIC that you designed instead of the one from Xilinx.

ANS: What would be different to use a custom CORDIC would be having to specify the library search path, we would have more control. However, the main drawback of the conventional CORDIC is that the number of iterations is equal to the number of angle constants. Among a great deal of research to overcome this disadvantage , angle recording method is an effective method because it is capable of reducing maybe 50% of the number of iterations.

the CORDIC algorithm stops converging after a point. This is because the successive approximation step sizes is equal to the magnitude of angle, where j is the iteration number. Custom has a effective limit for the sine and cosine functions' input magnitude. If the value of t is constrained to be smaller than this limit, then the CORDIC is used directly. Otherwise, the magnitude of t must be tested and if it is above the limit, an alternative software algorithm must be used, such as the math.h library functions cos() and sin().

Question 2: Rewrite the code to eliminate these math function calls (i.e. cos() and sin()) by utilizing a table lookup. How does this change the throughput and resource utilization? What happens to the table lookup when you change the size of your DFT?
ANS: High precision arbitrary sin() and cos() in particular tend to be expensive operations. The coefficients of cos and sin have been calculated in advance and the constant table is easy to read directly, so the code only needs to be accessed through

W_realand W_imag. (W_real[k]=cos(-2*PI*k/SIZE);)

Simply getting a number from a list is much faster than calculating the number with an algorithm or using a trigonometric function. The primary disadvantage of lookup tables is their memory usage and that continues as the DFT size is increased.

Question 3: Modify the DFT function interface so that the input and outputs are stored in separate arrays. Modify the testbench to accommodate this change to DFT interface. How does this affect the optimizations that you can perform? How does it change the performance? And how does the resource usage change? You should use this modified interface for the remaining questions.
ANS: The FFT is calculated in two parts. The first one transforms the original data array into a bit-reverse order array by applying the bit-reversal method. This makes the mathematical calculations of the second easier. The second part processes the FFT in N*log2(N) operations.

Question 4: Loop Optimizations: Examine the effects of loop unrolling and array partitioning on the performance and resource utilization. What is the relationship between array partitioning and loop unrolling? Does it help to perform one without the other? Plot the performance in terms of number of DFT operations per second (throughput) versus the unroll and array partitioning factor. Plot the same trend for resources (showing LUTs, FFs, DSP blocks, BRAMs). What is the general trend in both cases? Which design would you select? Why?
ANS: A1024-point FFT of complex 32-bit floating point values, running at 250 MHz would require 1024 points*(8 bytes/point)*250*109Hz = 1Terabyte/second of data into the FPGA. The relationship between array partitioning and loop unroll is that it tries to achieve kernel computation optimization by expanding the processing code to match that data path. The logic is

to consume all the data as soon as it arrives at the kernel. I have to use both together for it to compile.

Question 5: Dataflow: Apply dataflow pragma to your design to improve throughput. You may need to change your code and make submodules so that it aligns with the task-level or function-level modularity that dataflow can exploit; Xilinx provides some examples of dataflow code. The HLS User Guide pages 145-154 and this summary provide more information. How much improvement does dataflow provide? How does dataflow affect resource usage? What about BRAM usage specifically? Did you modify the code to make it more amenable to dataflow? If so, how? Please describe your architecture(s) with figures on your report.

ANS: The dataflow directive attempts to best implement the memory used to pass data between the two functions. The second for loop labeled butterfly performs all of the butterfly operations for the current stage. This butterfly for loop has another nested for loop with the label DFTpts; each iteration of this for loop performs one butterfly operation. Remember that we are dealing with complex numbers, thus we must perform complex additions and multiplications. The first line in this DFTpts for loop determines the offset of the butterfly.

```
#pragma HLS dataflow
  //Call fft
  DTYPE Stage1_R[SIZE], Stage1_I[SIZE];
  DTYPE Stage2_R[SIZE], Stage2_I[SIZE];
  DTYPE Stage3_R[SIZE], Stage3_I[SIZE];
  DTYPE Stage4_R[SIZE], Stage4_I[SIZE];
  DTYPE Stage5_R[SIZE], Stage5_I[SIZE];
  DTYPE Stage6_R[SIZE], Stage6_I[SIZE];
  DTYPE Stage7_R[SIZE], Stage7_I[SIZE];
  DTYPE Stage8_R[SIZE], Stage8_I[SIZE];
  DTYPE Stage9_R[SIZE], Stage9_I[SIZE];
  DTYPE Stage10_R[SIZE], Stage10_I[SIZE];

  bit_reverse(X_R, X_I, Stage1_R, Stage1_I);
  fft_stage_first(Stage1_R, Stage1_I, Stage2_R, Stage2_I);
  fft_stages(Stage2_R, Stage2_I, 2, Stage3_R, Stage3_I);
  fft_stages(Stage3_R, Stage3_I, 3, Stage4_R, Stage4_I);
  fft_stages(Stage4_R, Stage4_I, 4, Stage5_R, Stage5_I);
  fft_stages(Stage5_R, Stage5_I, 5, Stage6_R, Stage6_I);
  fft_stages(Stage6_R, Stage6_I, 6, Stage7_R, Stage7_I);
  fft_stages(Stage7_R, Stage7_I, 7, Stage8_R, Stage8_I);
  fft_stages(Stage8_R, Stage8_I, 8, Stage9_R, Stage9_I);
  fft_stages(Stage9_R, Stage9_I, 9, Stage10_R, Stage10_I);
  fft_stage_last(Stage10_R, Stage10_I, OUT_R, OUT_I);

}
```

Date: Tue Dec 7 20:43:16 2021
Version: 2021.1 (Build 3247384 on Thu Jun 10 19:36:33 MDT
Project: Project4_FFT_Stages
Solution: solution1 (Vivado IP Flow Target)
Product family: zynq
Target device: xc7z020-clg400-1

Timing Estimate

Target	Estimated	Uncertainty
10.00 ns	9.591 ns	2.70 ns

Performance & Resource Estimates

Modules Loops

Modules && Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
fft				-2.29	4171	4.171E4	-	4172	-	no	380	961	288477	304654	0
fft_Pipeline_VITIS_LOOP_93_1	II Violation			-	2050	2.050E4	-	2050	-	no	0	0	108	1426	0
fft_stage_first				-	81	810.000	-	81	-	no	0	0	4210	594	0
fft_stages_2_s	II Violation			-	115	1.150E3	-	115	-	no	0	100	17448	21088	0
fft_stages_3_s	II Violation			-	82	820.000	-	82	-	no	0	156	26314	32264	0
fft_stages_4_s	II Violation			-	82	820.000	-	82	-	no	0	174	38784	41034	0
fft_stages_5_s	II Violation			-	101	1.010E3	-	101	-	no	0	186	65142	62445	0
fft_stages_6_s	II Violation			-0.91	179	1.790E3	-	179	-	no	0	192	66033	62864	0
fft_stages_7_s	II Violation			-2.29	147	1.470E3	-	147	-	no	0	0	24117	19963	0
fft_stages_8_s	II Violation			-1.27	140	1.400E3	-	140	-	no	0	0	17713	11646	0
fft_stages_9_s	II Violation			-1.27	131	1.310E3	-	131	-	no	0	0	15086	8510	0
fft_Pipeline_butterfly_loop	II Violation			-0.91	1042	1.042E4	-	1042	-	no	0	0	931	1108	0

Figure 1 wo #DATAFLOW

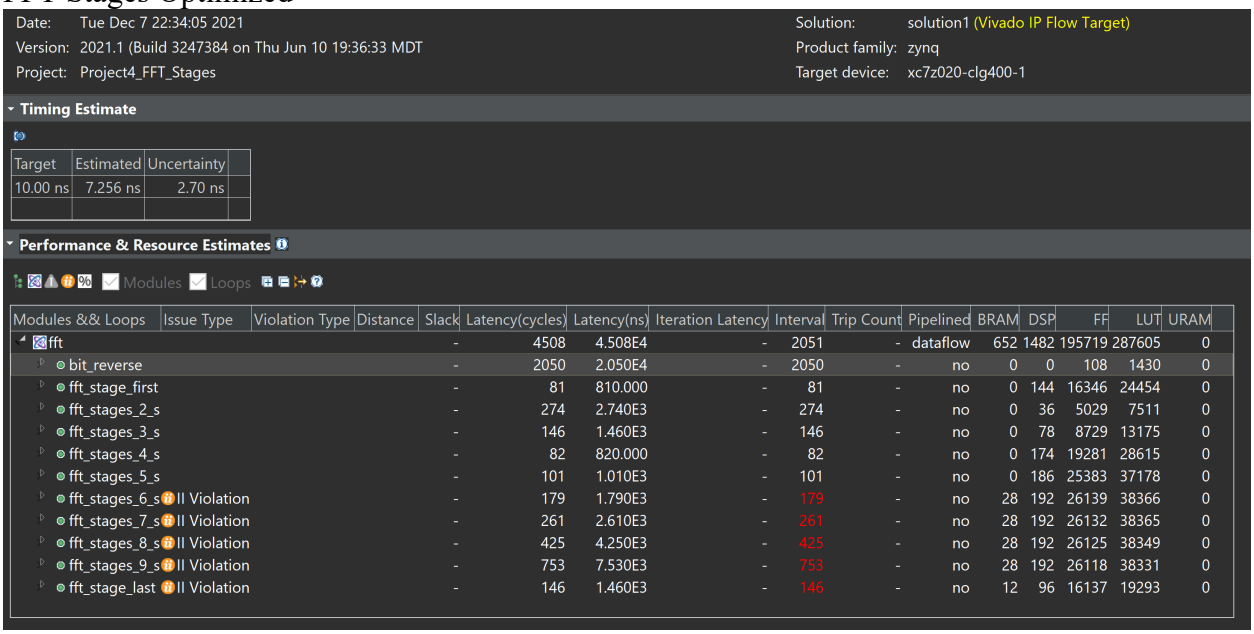
Most of FFT algorithms decomposes the overall N-point DFT into successively smaller and smaller transforms known as a butterfly. The butterfly is the heart of the FFT. It takes data words from memory and computes the FFT. Results are written back to the same memory locations since an in-place algorithm is used. We modify our code into stages. The FFT contains significant parallelism, because each butterfly is independent of one another in the same stage. The n/2 butterfly computations every clock cycle with a task interval of 1. When adding the #dataflowthe parallelism in the FFT can be exploited.

Question 6: Best architecture: Briefly describe your "best" architecture. In what way is it the best? What optimizations did you use to obtain this result? What are the tradeoffs that you considered in order to obtain this architecture?
ANS:
The pipeline is implemented, but the resources available on the hardware chip are not utilized to the greatest extent. Through loops, the `#Pragma unroll` parallelism can be further increased, the number of loops can be reduced , and the final latency can be reduced. Here are some of the optimizations have been made or tried:

1. Use template to implement fft_stages, so you can enter the stage number as a constant at compile time, which is convenient for pragma writing
2. bit_reverse Function I and reversed situation of address classifies discussed, to avoid the occurrence of read and write the same address, while explicitly informed that the write after write HLS (WAW) dependence does not exist, and thus the cycle can be done II = 1
3. Reduce the fft_stage_first sum fft_stage_last to a loop, because there is a layer of loops with a loop count of 1
4. Set dft_loop the start of the loop to 0~loop_times, the number of loops can be determined at compile time, so that HLS can give performance results
5. For the operation dft_loop of reading X_R and X_I summing, create a local variable for storage to avoid repeated reading in the following calculations

6. To dft_loop perform unroll operations, the corresponding input and output arrays are also cyclic divided

FFT Stages Optimized



Date: Tue Dec 7 22:34:05 2021
Version: 2021.1 (Build 3247384 on Thu Jun 10 19:36:33 MDT
Project: Project4_FFT_Stages
Solution: solution1 (Vivado IP Flow Target)
Product family: zynq
Target device: xc7z020-clg400-1

Timing Estimate

Target	Estimated	Uncertainty
10.00 ns	7.256 ns	2.70 ns

Performance & Resource Estimates

Modules Loops

Modules && Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
fft				-	4508	4.508E4	-	2051	-	dataflow	652	1482	195719	287605	0
bit_reverse				-	2050	2.050E4	-	2050	-	no	0	0	108	1430	0
fft_stage_first				-	81	810.000	-	81	-	no	0	144	16346	24454	0
fft_stages_2_s				-	274	2.740E3	-	274	-	no	0	36	5029	7511	0
fft_stages_3_s				-	146	1.460E3	-	146	-	no	0	78	8729	13175	0
fft_stages_4_s				-	82	820.000	-	82	-	no	0	174	19281	28615	0
fft_stages_5_s				-	101	1.010E3	-	101	-	no	0	186	25383	37178	0
fft_stages_6_s	II Violation			-	179	1.790E3	-	179	-	no	28	192	26139	38366	0
fft_stages_7_s	II Violation			-	261	2.610E3	-	261	-	no	28	192	26132	38365	0
fft_stages_8_s	II Violation			-	425	4.250E3	-	425	-	no	28	192	26125	38349	0
fft_stages_9_s	II Violation			-	753	7.530E3	-	753	-	no	28	192	26118	38331	0
fft_stage_last	II Violation			-	146	1.460E3	-	146	-	no	12	96	16137	19293	0

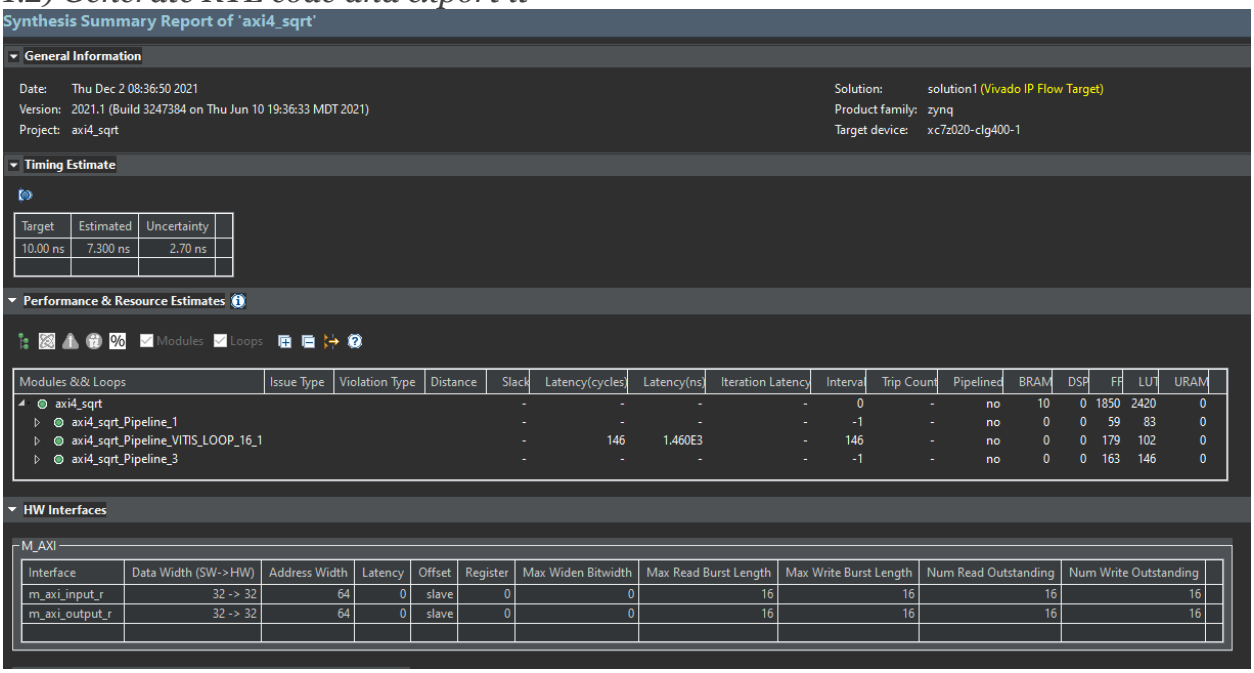
Generating csim.exe

RMSE(R) RMSE(I)
0.000450454419479 0.000541143584996

****

PASS: The output matches the golden output!

1.2) Generate RTL code and export it



Synthesis Summary Report of 'axi4_sqrt'

General Information

Date: Thu Dec 2 08:36:50 2021
Version: 2021.1 (Build 3247384 on Thu Jun 10 19:36:33 MDT 2021)
Project: axi4_sqrt
Solution: solution1 (Vivado IP Flow Target)
Product family: zynq
Target device: xc7z020-clg400-1

Timing Estimate

Target	Estimated	Uncertainty
10.00 ns	7.300 ns	2.70 ns

Performance & Resource Estimates

Modules Loops

Modules && Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interva	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
axi4_sqrt				-	-	-	-	0	-	no	10	0	1850	2420	0
axi4_sqrt_Pipeline_1				-	-	-	-	-1	-	no	0	0	59	83	0
axi4_sqrt_Pipeline_VITIS_LOOP_16_1				-	146	1.460E3	-	146	-	no	0	0	179	102	0
axi4_sqrt_Pipeline_3				-	-	-	-	-1	-	no	0	0	163	146	0

HW Interfaces

M_AXI

Interface	Data Width (SW->HW)	Address Width	Latency	Offset	Register	Max Widen Bitwidth	Max Read Burst Length	Max Write Burst Length	Num Read Outstanding	Num Write Outstanding
m_axi_input_r	32 -> 32	64	0	slave	0	0	16	16	16	16
m_axi_output_r	32 -> 32	64	0	slave	0	0	16	16	16	16

Question 7: Streaming Interface Synthesis: Modify your design to allow for streaming inputs and outputs using hls::stream. You must write your own testbench to account for the function interface change from DTYPE to hls::stream. NOTE: your design must pass Co-Simulation (not just C-Simulation). You can learn about hls::stream from the HLS Stream Library. An example of code with both hls::stream and dataflow is available (along with its testbench) here, and another example showing hls::stream between functions. Describe the major changes that you made to your code to implement the streaming interface. What benefits does the streaming interface provide? What are the drawbacks?