

Report – Hardware Accelerator Project

Fast Fourier Transform

김현진

0. Table of Contents

1. Summary	03
2. Introduction	04
3. Specifications	07
4. Issues and Solutions	11
5. Conclusion and Future Plans	13
6. Appendix	16
A. Operating Principles of the FFT Module	16
B. AXI Interface and Block Design	19
C. SW level	21

1. Summary

Goal	FFT 하드웨어 가속기 설계
Duration	2023.09.27 ~ 11.22 (progressed irregularly) 2024.10.26 ~ 10.27
Device	FPGA development board(Zybo Z7)
Design field	HW, SW(test)
Language	(Design) Verilog, (Test) C, MATLAB
Tool	Xilinx Vivado, Vitis

2. Introduction

❖ FFT란?

FFT(Fast Fourier Transform)는 디지털 신호를 시간 영역에서 주파수 영역으로 변환하는 알고리즘이다. 기존의 이산 푸리에 변환(Discrete Fourier Transform)에 비해 계산 복잡도가 획기적으로 적다. Radix-2와 같은 다양한 FFT 알고리즘이 있으며, 이를 통해 복잡한 계산을 효율적으로 수행할 수 있다.

❖ 신호처리에서의 중요성

Fourier Transform은 시간 영역에서 복잡한 **주파수 성분을 분석**하는 데 필수적이다. 이를 통해 신호의 특성을 정확히 이해하고 분석할 수 있어, 필터링, 잡음 제거, 스펙트럼 분석 등에 널리 사용된다. 특히, **실시간 데이터 처리**가 필요한 경우에 FFT는 고속 변환으로 시스템 성능을 크게 향상시킨다.

❖ 활용 방안

- **통신 분야:** FFT는 OFDM(Orthogonal Frequency Division Multiplexing) 같은 다중화 기술에서 주파수 분할 및 신호 복원을 위해 사용된다. FFT를 통해 각 서브캐리어의 주파수 성분을 분리하여 효율적인 데이터 전송과 잡음 제거가 가능하다.
- **이미지 및 음성 처리:** 이미지나 음성 신호의 특성 분석에 활용되어, 압축, 잡음 제거, 필터링 등에 사용된다.
- **레이더:** 레이더 신호의 주파수 분석을 통해 물체 식별 및 거리 측정을 수행한다.

Project Goal

주 목표 FFT Module 설계

Optimization

- ✓ 파이프라이닝으로 연산 모듈을 공유하여 리소스 최적화
- ✓ 100MHz target frequency
 - (여유가 되면) BRAM으로 FF 사용량 감축
- ✓ **Vivado FFT IP와 유사한 수준의 리소스 사용**

Design

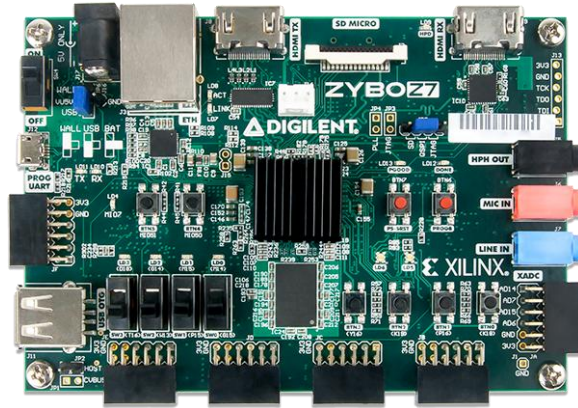
- ~~8-point Streaming I/O FFT 모듈 설계(취소, 이유 후술)~~
- ✓ **8-point Burst I/O FFT 모듈 설계**
 - 설계한 Burst I/O FFT 2개를 결합하여, 64-point streaming I/O FFT 모듈 설계

Testing

- ✓ Vitis로 SW Level에서 디지털 값 입력 및 출력 확인
- ✓ MATLAB으로 검증
- ✓ **동작의 신뢰성 확인**

✓ 달성 완료

Development Environment



Zybo Z7-20

FPGA Development Board
(Zynq-based)



80%

Usage
ratio



20%

3. Specifications

❖ Overview

Category	FFT IP
Point (Transform Length)	N = 8
Input/output Port Width	16-bit -32768~32767
Operating Frequency	100MHz
Processing Speed	13 clock cycle 130ns

❖ Details

Algorithm (Radix)	Radix-2
Algorithm (DIT or DIF)	DIF
I/O	Burst I/O
Twiddle Factor Width	16-bit
Number Format	Fixed Point
Output Ordering	Bit Reversed

❖ Reason for Selection

• 16-bit width

SW level에서 효율적으로 프로세싱할 수 있는 단위. 8-bit는 해상도가 너무 낮고, 12-bit 또한 고려하였으나, 사전 테스트 결과 16-bit와 DSP 사용량이 동일하여 16-bit를 최종 선택.

• Radix-2

설계 및 파이프라이닝이 비교적 단순함

• DIF

Appendix. A(p.18)참고




• Burst I/O

연산 모듈을 공유하기 때문에, 리소스 최적화에 유리함. 그러나 FFT 연산 도중에는 새로운 입력을 받을 수 없음

본래 실시간 신호 처리에는 적합하지 않으나, point가 작아 latency가 크기 않으므로 선택

3. Specifications

❖ Utilization

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	DSPs (220)	Bonded IOPADs (130)	BUFGCTRL (32)
▼  design_1_wrapper	714	1120	32	310	598	116	4	130	1
▼  design_1_i (design_1)	714	1120	32	310	598	116	4	0	1
>  fft_0 (design_1_fft_0_0)	365	689	32	175	309	56	4	0	0







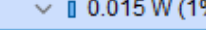







❖ Timing(including other IPs)

Setup	Hold
Worst Negative Slack (WNS): 1.182 ns	Worst Hold Slack (WHS): 0.065 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 2503	Total Number of Endpoints: 2503

All user specified timing constraints are met.

Pulse Width
Worst Pulse Width Slack (WPWS): 3.750 ns
Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 1295

❖ Power

Utilization	Name
▼  1.545 W (92% of total)	 design_1_wrapper
▼  1.545 W (92% of total)	 design_1_i (design_1)
>  1.527 W (91% of total)	 processing_system7_0 (design_1_proc7_0)
▼  0.015 W (1% of total)	 fft_0 (design_1_fft_0_0)
▼  0.015 W (1% of total)	 inst (design_1_fft_0_0_fft_v1_0)
▼  0.015 W (1% of total)	 fft_v1_0_S00_AXI_inst (design_1_fft_0_0_fft_v1_0_S00_AXI_inst)
>  0.014 W (1% of total)	 myip_inst (design_1_fft_0_0_fft_8p_to_0_0_myip_inst)

Register Map

Index	Address BASEADDR+ (hex)	Type of Register	Access Type	Description
0	0x00	Data Register	Write	Bit 31-16: real part Bit 15-0: imaginary part
1	0x04	Data Register	Read	Bit 31-16: real part Bit 15-0: imaginary part

- * Output is **bitwise inverted**. ex) $X[4] (0b100) \leftrightarrow X[1] (0b001)$
- * Write one value to the write register and then immediately read one value from the read register.
- * The output has a **1-cycle (8 counts) delay**. Thus, for the first 8 inputs, the results are not valid values.

vs. Vivado IP

Resource	Used	Total	Utilizatio...
LUT	415	53200	0.7801
LUTRAM	121	17400	0.6954
FF	586	106400	0.5508
BRAM	2	140	1.4286
DSP	3	220	1.3636
BUFG	1	32	3.125

<Resources of Vivado IP>

	Vivado IP	Custom IP
LUT	415	365
LUTRAM	121	56
FF	586	689
BRAM	2	0
DSP	3	4
Latency	356ns	130ns

<Comparative Analysis>

*음영은 Custom IP가 Vivado IP보다 우수한 부분에 표시

- Vivado의 FFT IP를 동조건(8-point, 2-Radix Burst I/O, 16-bit width etc.)으로 설정하고 값을 비교

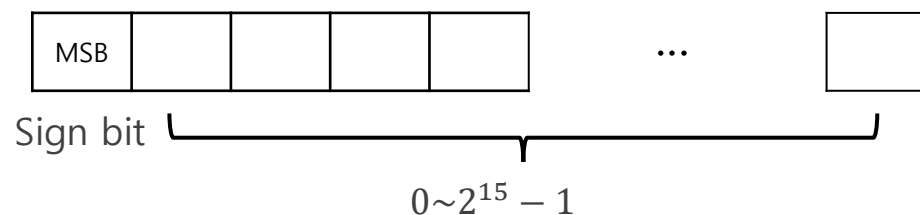
- Vivado IP의 경우, **FF** 사용 일부를 **BRAM**으로 대체한 것으로 보인다.
- LUT**는 Custom IP쪽의 사용량이 유의미하게 낮았다. 단, Vivado IP의 입출력 포트가 훨씬 많다는 것을 감안해야 한다. Custom IP는 기능 테스트 및 신뢰성 확인을 위해 필요최소한의 기능만 삽입하였다.
- Latency**면에서 Custom IP의 동작이 더 빨랐다. Vivado IP의 경우 point(transform length)가 낮을 경우 latency가 생각보다 긴 특징을 보였다. 잘 사용되지 않는 낮은 point의 FFT의 경우, 비교적 덜 최적화된 것으로 추정된다.
- DSP** 사용량은 Vivado IP가 더 낮았다. 이것은 Gauss' trick을 사용했기 때문으로 추정된다. (4장에서 후술)

4. Issues and Solutions

■ 연산(x,+, -)을 할수록 데이터의 크기가 커지는 문제

■ Solution 1. Normalization and Fixed Format

- 16-bit 데이터 중 **MSB**는 **Sign Bit**로, 하위 15-bit는 **소수부**로 사용(정수부 없음)
- 따라서 데이터의 범위를 아래와 같이 조정



실제 크기: -1~1

⇒ $-2^{15} \sim 2^{15} - 1$ 의 범위로 scaling

■ Solution 2. Truncation

- Normalization으로 곱셈 연산에서의 Overflow를 해결
- +,- 연산의 Overflow는 truncation 적용(정밀도 손실)

* 이 때, 최초에 Logical shift 연산(>>)을 적용하니, sign bit가 shift되어 값이 달라지는 이슈 발생. 아래의 연산으로 대체하여 해결

- 1. Arithmetic Shift(>>>) 사용
- 2. 직접 대입 ex) `wire [16:0] y_tmp = a + b;`
`wire [15:0] y = y_tmp[16:1];`

* 테스트 결과 두 방식은 리소스 차이가 전혀 없었음. 동일하게 합성되는 것으로 추정됨

4. Issues and Solutions

■ DSP 사용량 문제

- 최초에, 각각의 연산 블록을 Stage별로 할당하여 설계
- Streaming 입력이 가능해지나, 리소스(특히 DSP)가 과다하게 사용되는 문제 발생

■ Solution

- 보다 제약을 가해, 연산 블록 1개를 서로 다른 타이밍에 공유하는 설계로 변경
- **Streaming I/O → Burst I/O**

*Latency에는 차이가 없으나, 모든 stage의 계산을 마쳐야 다음 입력을 받을 수 있다. 단, point가 낮아서 실질적으로 영향은 없다.

- **DSP 소요 변경**

Before	After
12 (3 stage * 4 DSP for each)	4

- 추가적으로, **Gauss Trick**을 적용하여 ¾의 DSP를 사용하도록 시도

$$\begin{cases} p = (b + b_i)(w + w_i) \\ q = (b \times w) \\ r = (b_i \times w_i) \end{cases}$$

$$\begin{cases} real = q - r \\ imagine = p - q - r \end{cases}$$

- 단, 17-bit 수끼리의 곱에는 DSP가 3개 소요되어, 16-bit로 scaling 후 곱하였음. 그 결과 예상외로 큰 **정말도 손실** 유발
- 또한, Logic 연산으로 인해 LUT 사용량이 오히려 증가
- 이로 인해 DSP 사용량을 줄였음에도 Power면에서도 큰 차이가 없었음
- 따라서 해당 안은 **사용하지 않기로 결정**

5. Future Improvement

❖ Primary Objective

64-point FFT 추가 설계

- 2개의 Burst I/O 8-point FFT 모듈을 연결하여 64-point의 streaming I/O 모듈 설계
- 이 경우 3개의 stage를 하나의 연산 모듈이 처리한다. 따라서 3개의 stage에서 연산이 완료되면 다음 입력 신호를 받을 수 있다.
- Latency는 2.89us로 예상된다.*

* 연산 횟수: $\frac{N}{2} \log_2 N$

입력 delay: $\frac{N}{2}$

Twiddle factor delay: 1

Total delay: $256 + 32 + 1 = 289$

→ 2.89us

Streaming delay: $256/2 = 128$

→ 1.28us

❖ Secondary Objective

1. done 신호를 추가하여 PS로 전송 ————— Status Register 및 bus 1개 추가 사용
2. data buffer 및 twiddle factor table을
BRAM으로 변경하여 LUTRAM과 Flip-flop
절약

5. Conclusion

❖ 성과

- 목표 주파수(100MHz)에서의 동작을 달성하였다.
- 고정소수점 연산을 사용하면서도 적은 손실도를 달성하였다.
- 설계한 모듈을 확장하여, 보다 높은 Point의 모듈을 설계할 수 있는 기반을 마련했다.

❖ 학습한 내용

- 복잡한 계산 모듈을 설계하는데 있어, Tradeoff에 대한 개념을 숙지했다.

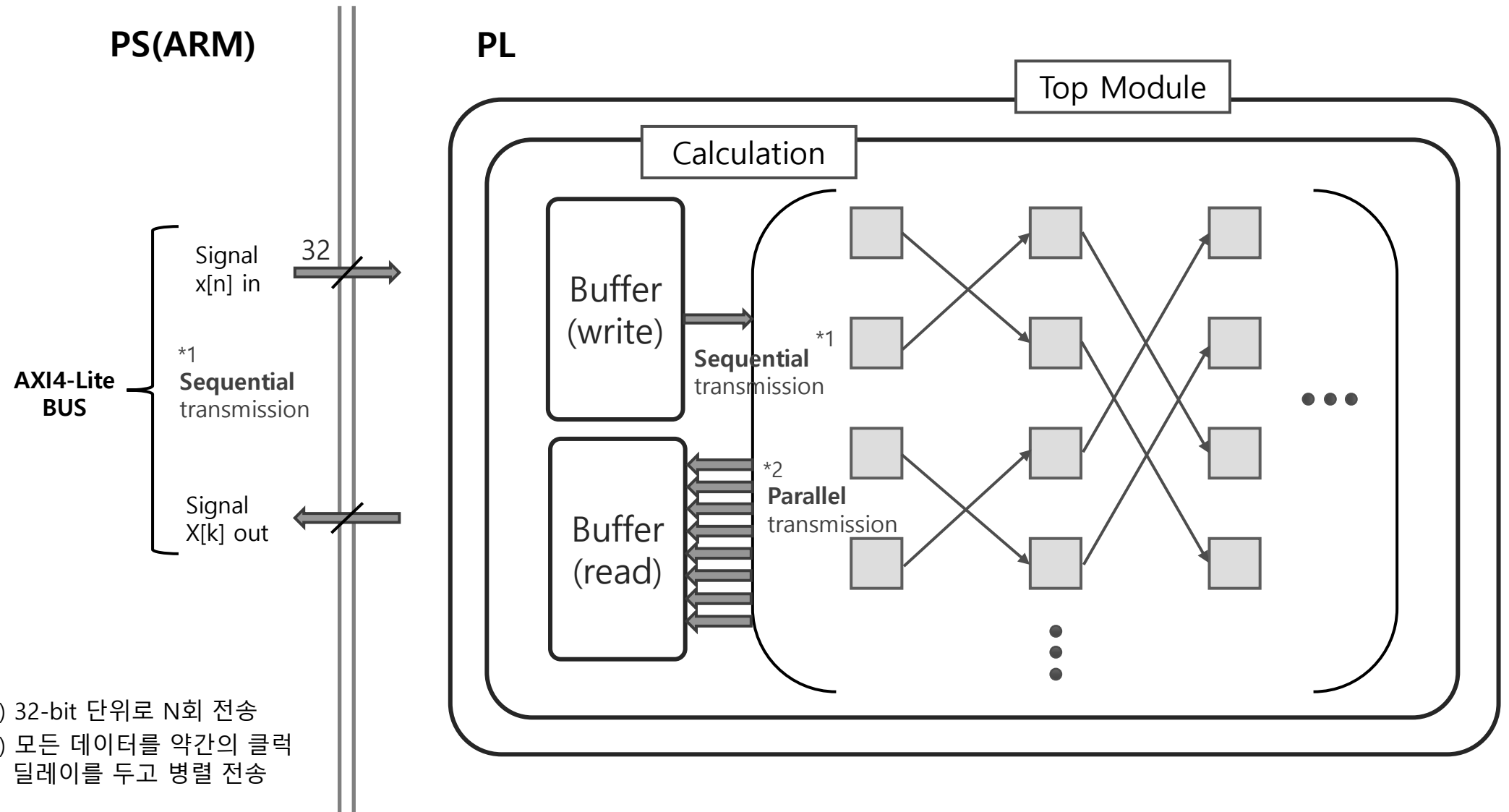
	Burst I/O	Gauss' Trick	Truncation
Area	P	P(DSP) / L(CLB)	P
Power	L*	-	P
Performance	L(latency)	L(latency)	L(precision)

*Burst I/O가 추가적인 LUT 사용으로 인해
Power 소모가 커질 것으로 예상됨

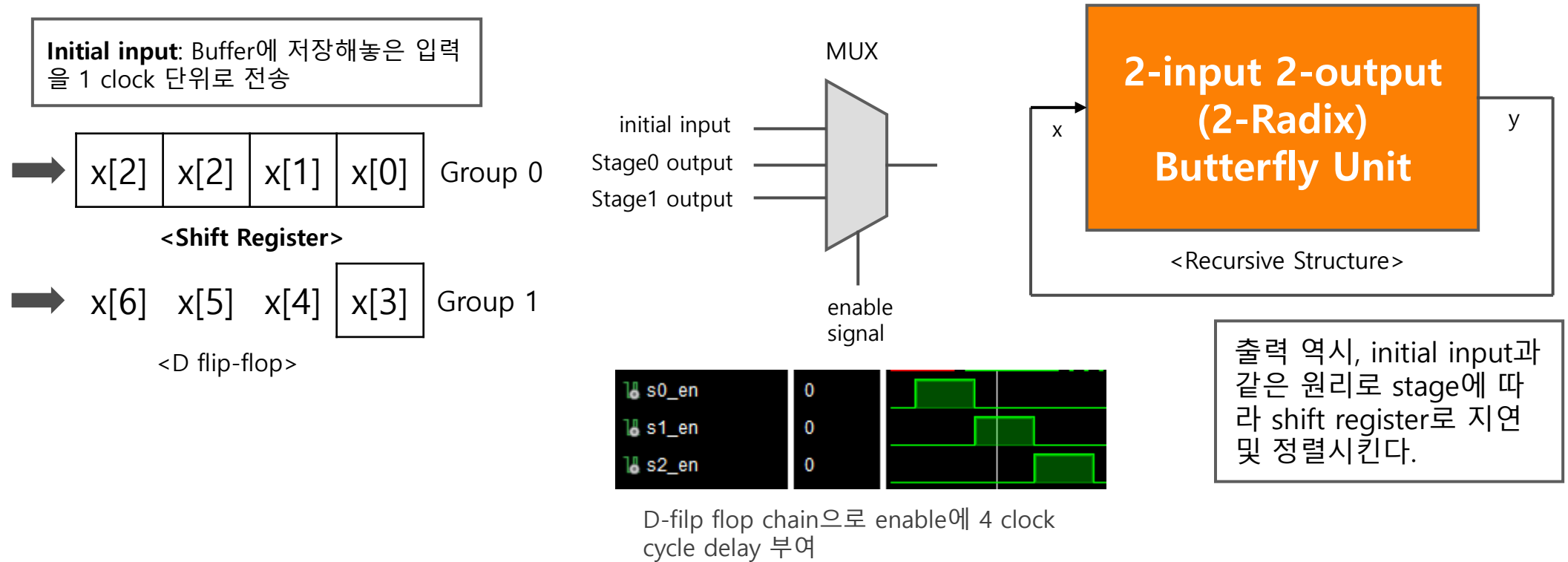
P: Profit L: loss -: 영향이 미미함

Thank you.

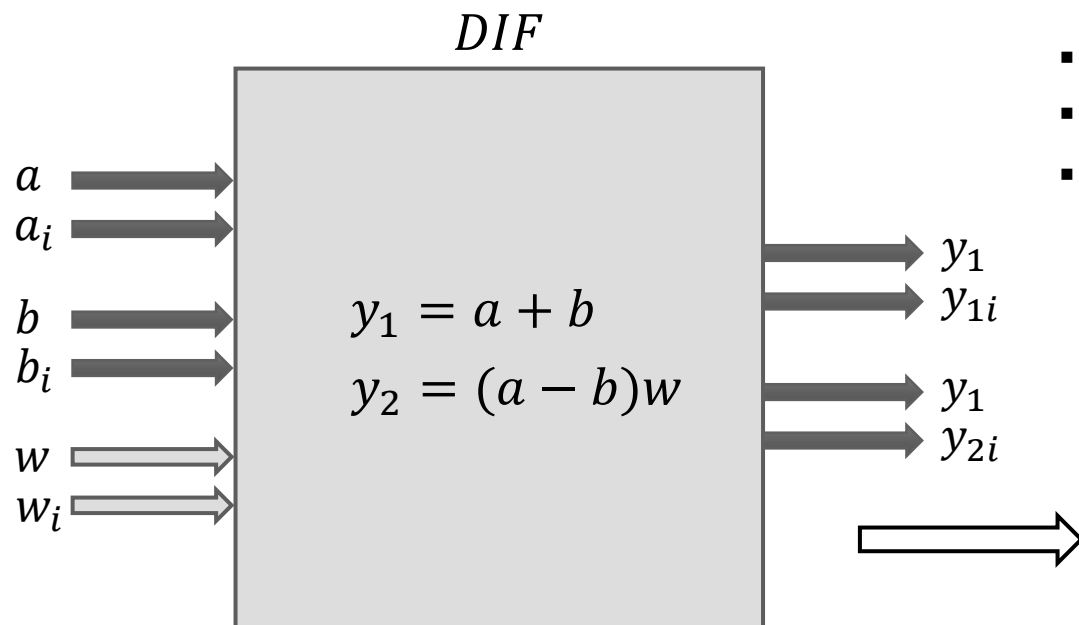
Appendix A. FFT Module



Appendix A. Shift Register



Appendix A. Calculation Unit

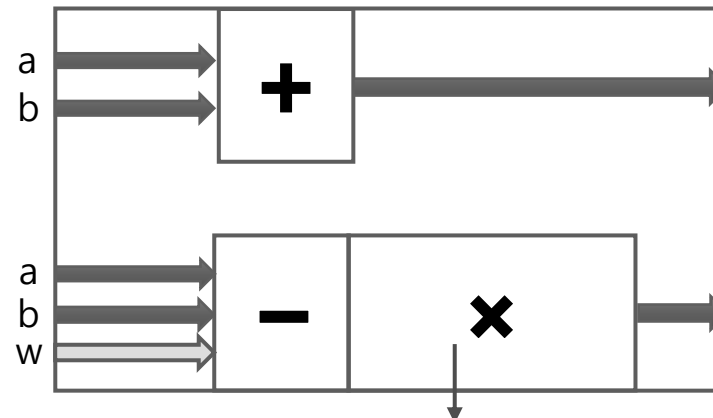


a, b: variable
w: constant

Twiddle factor(W)

$$W_N^k = e^{-\frac{j2\pi k}{N}} = \cos\left(\frac{2\pi k}{N}\right) - j\sin\left(\frac{2\pi k}{N}\right)$$

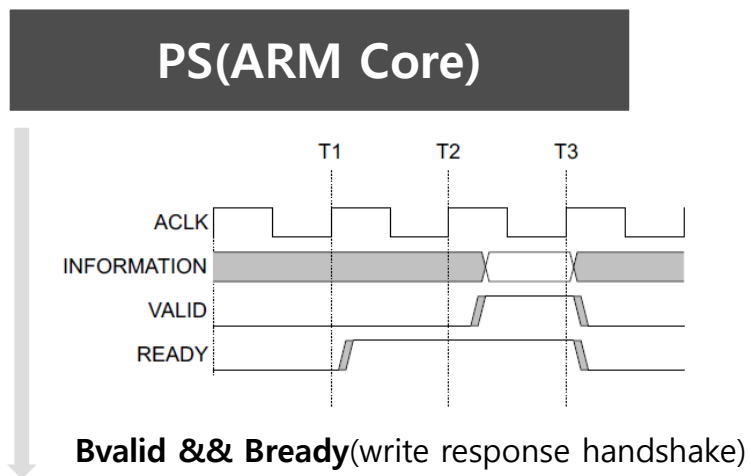
- 16-bit끼리 수의 곱셈에는 1개의 DSP 소요
- $(a + jb_i)(a + jb_i) = (ab - a_i b_i) + j(ab_i + a_i b)$
- 4번의 곱셈이 병렬로 실행되므로, 총 4개의 DSP 소요됨



1 clock cycle delay

- DSP의 높은 latency로 인해 값을 register에 1회 저장 후 출력
- 이것은 **DIT** 대신 **DIF 알고리즘**을 사용한 이유로, DIT의 경우 2개의 출력 모두 곱셈 연산의 필요로 인해 딜레이가 추가로 발생하여 파이프라인에 불리함
- 시스템 클럭 주파수를 높일 경우, register에 추가로 데이터를 중간 저장할 필요가 있음(플러스, 마이너스 연산 등에도 추가)

Appendix B. PS-PL Interface



Write Counter +1

- PS에서 레지스터에 write하면 AXI handshake 타이밍에 맞추어 **write 카운터**를 1 올린다.
- 카운터에 따라 **write buffer**의 index에 접근하여, PS로부터 입력받은 데이터를 차례대로 적재한다.
- 카운터가 N회(8회) 누적되면 시작 신호가 high가 되며, enable 신호가 켜진다. 입력 buffer의 데이터는 10ns 간격으로 차례대로 sub 모듈에 전송된다.

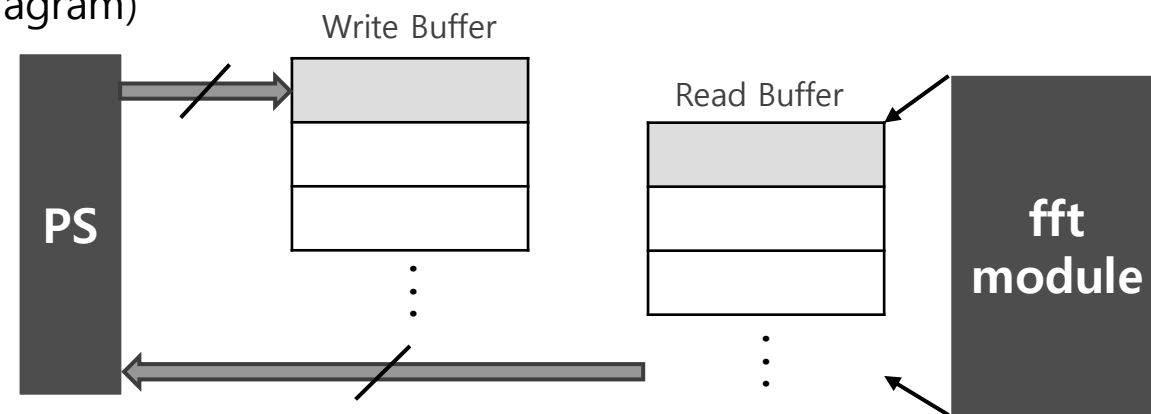
130ns delay

Read Counter +1

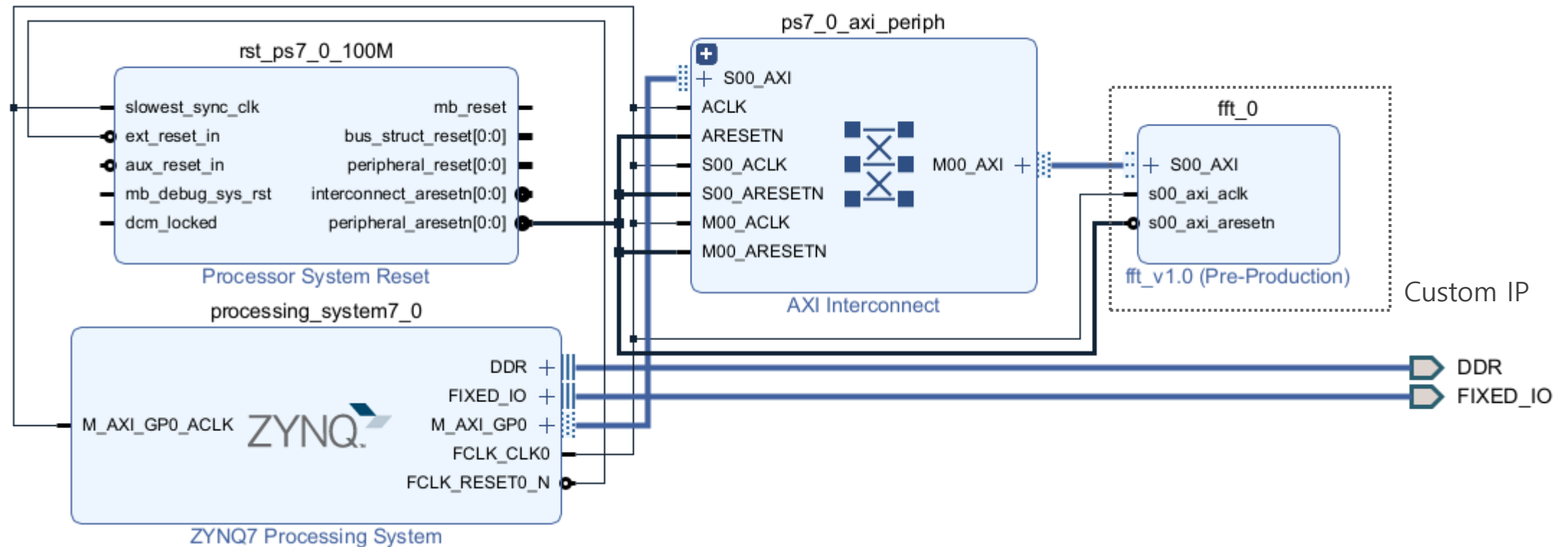
- FFT 계산에 소요되는 시간 만큼 딜레이를 준다(D flip-flops chain으로도 가능하나, 확장성을 고려해 counter delay 사용)
- Delay된 신호를 받을 때마다 **read 카운터**를 1 올린다. Read buffer의 index에 접근하여, **read register**에 데이터를 적재해둔다.
- ❖ PS에서 register write와 register read를 **1번씩 교차해서 입력**할 것을 상정하여 디자인됨

❖ 따라서 입출력에는 1cycle(8회)의 delay가 있다.

Diagram)



Appendix B. PS-PL Interface



<Block Design of IP Integration>

Appendix C. Verification

Code review

FFT 결과를 MATLAB으로 Inverse FFT하여 원래의 신호와 비교대조

```
/* random signal */
for(i=0; i<SAMPLES; i++){
    tmp[0][i] = 1000*i;           // real
    tmp[1][i] = 7000 - 1000*i;   // imagine
}
for(uint8_t i=0; i<SAMPLES; i++){
    x[0] = tmp[0][i]; x[1] = tmp[1][i];
    InSignal(x);
    OutSignal(X);
    printf("%2d: %d%di %d%di\n", i, x[0], x[1], X[0], X[1]);
}
```

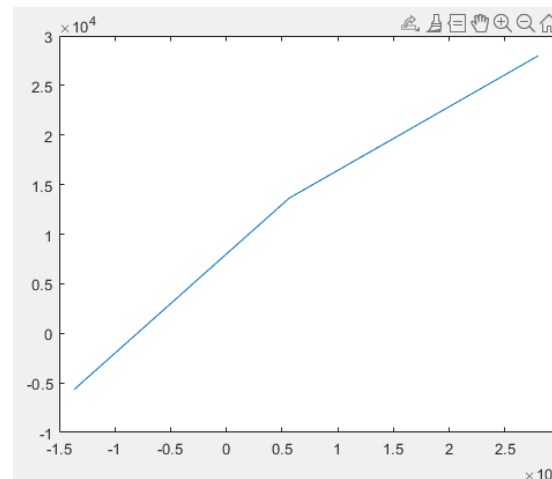
↓ Uart Transmission

0: +0+7000i +0+	0: +3500+3500i
1: +1000+6000i	1: -500+499i
2: +2000+5000i	2: +0+999i
3: +3000+4000i	3: -1000-1i
4: +4000+3000i	4: +707+1706i
5: +5000+2000i	5: -707+292i
6: +6000+1000i	6: -293+707i
7: +7000+0i +0+	7: -1707-708i

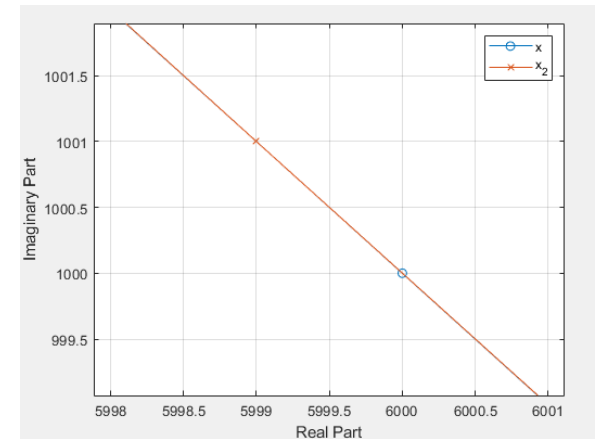
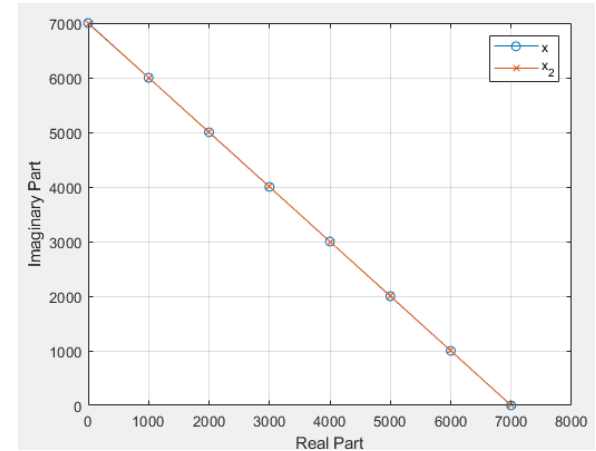
Input

Output

Bit Reverse



IFFT



Appendix C. 오차 원인 분석

※ 주 원인

1. 고정소수점 연산으로 인한 정밀도 제한
2. Signed format에서, 음수 최대의 절대값이 양수의 값보다 1 커서 발생하는 손실
- 3. Truncation으로 인한 손실**

*3.의 경우, 정밀도를 희생하지 않고 stage 1개당 출력 포트의 width를 1씩 증가시키면 손실이 없어진다. 단, 계산을 $\log(N)$ stage만큼 수행하므로, N 이 커질수록 출력 포트의 width가 감당하기 어려울 정도로 증가할 수 있다.

※ 기타 원인

- 16비트 width의 한계로 인한 정밀도 제한