Journal of the KIMST, Vol. 21, No. 4, pp. 437-446, 2018
ISSN 1598-9127
DOI http://dx.doi.org/10.9766/KIMST.2018.21.4.437

Research Paper 센서 · 신호처리 부문

압축센싱 디지털 수신기 신호처리 로직 구현

안우현*,1) · 송장훈1) · 강종진1) · 정 웅2)

1) 한화시스템㈜ 전자전연구센터
2) 자일링스 코리아 기술영업부

Signal Processing Logic Implementation for Compressive Sensing Digital Receiver

Woohyun Ahn*,1) · Janghoon Song1) · Jongjin Kang1) · Woong Jung2)

1) *EW R&D Center, Hanwha Systems, Korea*
2) *Technical Sales Department, Xilinx Korea, Korea*

(Received 22 December 2017 / Revised 10 May 2018 / Accepted 22 June 2018)

ABSTRACT

This paper describes the real-time logic implementation of orthogonal matching pursuit(OMP) algorithm for compressive sensing digital receiver. OMP contains various complex-valued linear algebra operations, such as matrix multiplication and matrix inversion, in an iterative manner. Xilinx Vivado high-level synthesis(HLS) is introduced to design the digital logic more efficiently. The real-time signal processing is realized by applying dataflow architecture allowing functions and loops to execute concurrently. Compared with the prior works, the proposed design requires 2.5 times more DSP resources, but 10 times less signal reconstruction time of 1.024 μs with a vector of length 48 with 2 non-zero elements.

Key Words : Compressive Sensing(압축센싱), Orthogonal Matching Pursuit, High-Level Synthesis(고 수준 합성), Complex Matrix Multiplication(복소 행렬곱), Inverse Matrix(역행렬)

1. 서 론

이스라엘 테크니온 공과대학의 Eldar 교수팀은 기존의 나이키스트 표본화 이론을 벗어난 혁신적인 sub-Nyquist 수신기[1]를 발표하였다. Xampling이라 명명된 이 수신기는 대상신호의 sparse한 특성을 바탕으로 다수의 대상신호를 나이퀴스트율 보다 낮은 표본화율을 갖는 ADC(Analog to Digital Converter)를 이용해 동시에 수신할 수 있다. Xampling 수신기는 크게 고주파 신호를 의사랜덤패턴(Pseudo Random Binary Sequence, PRBS)과 혼합하여 신호를 압축하는 아날로그 단과, 아날로그 신호를 디지털 신호로 변환하여 원신호로 복원하는 디지털 단으로 나눌 수 있다.

* Corresponding author, E-mail: woohyun.ahn@hanwha.com

이 논문에서는 압축센싱 수신기의 디지털 단 개발에 적용된 디지털 신호처리 알고리즘을 Xilinx FPGA (Field Programmable Gate Array)로 구현하는 과정 및 결과를 논의하였다. 압축신호 탐지 및 복원에는 Orthogonal Matching Pursuit(OMP) 알고리즘을 활용하였다. OMP 알고리즘은 다수의 행렬 연산이 포함되어 있어 HDL(Hardware Description Language)를 활용하여 로직을 직접 설계하기 보다는 로직의 수정과 디버깅이 용이한 Xilinx Vivado HLS(High-Level Synthesis)를 사용하였다. Vivado HLS는 C/C++ 언어를 Xilinx FPGA에 최적화된 HDL로 변환해 주기 때문에, 적절한 하드웨어 구조를 설정하면 알고리즘 로직 구현에 소모되는 시간과 비용을 절약할 수 있다. 제안한 디지털 로직은 고속으로 입력되는 압축신호를 실시간으로 처리할 수 있도록 설계하였다. 이를 위해 데이터의 흐름을 고려한 로직 구조(dataflow)를 적용하였으며, 기존의 연구보다 더 많은 FPGA 자원을 활용하여 선형대수 연산을 병렬로 처리하였다. 또한 압축신호를 주파수 영역에서 처리할 수 있도록 복소수 연산을 도입하였다.

2. 신호처리 알고리즘

신호처리 알고리즘은 압축신호를 전처리하는 부분과 OMP 알고리즘을 적용하는 두 부분으로 나눌 수 있다. 부분별 각 단계는 다음과 같다.

전처리 1 단계 : 4채널 16비트(bit) ADC 모듈에서 입력되는 표본을 이산 푸리에 변환하여 스펙트럼 데이터를 구한다. 이 때 이산 푸리에 변환 길이는 256으로 한다.

전처리 2 단계 : 스펙트럼 데이터에 0 부터 255 까지 순서대로 번호를 할당하고, Table 1의 번호에 따라 데이터를 정렬한 복소행렬 Y를 생성한다.

OMP 1 단계 (정합) : 측정행렬 $\boldsymbol{A}$를 정규화 하고 켤레전치(conjugate transpose)를 취한 행렬의 1~48행을 취한 $\boldsymbol{A_N^H}$과 $\boldsymbol{R_k}$행렬을 곱하여 상관 행렬 $\boldsymbol{Z}$를 구한다.

$$\boldsymbol{Z} = \boldsymbol{A_N^H R_k} \tag{1}$$

여기서 $\boldsymbol{R_k}$는 나머지(residue)를 뜻하는 복소행렬로 반복횟수가 $k=0$일 때 $\boldsymbol{R_0}=\boldsymbol{Y}$가 된다.

OMP 2 단계 (정합) : 상관 행렬 $\boldsymbol{Z}$의 각 원소의 크기를 구하고 열 방향으로 더한 뒤, 값이 가장 큰 행의 번호 $i_{\max}$를 구한다.

$$i_{\max} = \max_i \sum_j |\boldsymbol{Z}(i,j)| \tag{2}$$

OMP 3 단계 (정합) : 전 단계에서 구한 행 번호 $i_{\max}$와 0 Hz(DC)를 기준으로 대칭이 되는 번호를 구하여 서포트 벡터(Support Vector, SV)를 구한다.

$$\boldsymbol{S_k} = \boldsymbol{S_{k-1}} \cup \{i_{\max}, A_c - i_{\max} + 1\} \tag{3}$$

여기서 A_c는 측정행렬 A의 열 크기 이고, 서포트 벡터의 원소 수는 매 반복마다 $2(k+1)$씩 증가한다.

OMP 4 단계 (선택) : 측정행렬 A에서 서포트 벡터에 해당하는 열만을 선택하여 새로운 행렬 A_S를 생성한다.

$$\boldsymbol{A_S} = \boldsymbol{A}(:, \boldsymbol{S_k}) \tag{4}$$

Table 1. Reordering table for the Fourier transform data

col / row	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
1	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
2	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
4	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79

여기서 A_S행렬의 열은 매 반복마다 $2(k+1)$씩 증가한다.

OMP 5 단계 (최소제곱해) : $\boldsymbol{A_S}$행렬과 켤레전치를 취한 행렬 $\boldsymbol{A_S^H}$을 활용하여 $\boldsymbol{C}$행렬을 구한다.

$$\boldsymbol{C = A_S^H A_S} \tag{5}$$

OMP 6 단계 (최소제곱해) : 전 단계에서 구한 행렬의 역행렬을 구하여 사영행렬 $\boldsymbol{G}$를 구한다.

$$\boldsymbol{G = C^{-1} A_S^H} \tag{6}$$

OMP 7 단계 (최소제곱해) : 복원신호의 스펙트럼 $\hat{\boldsymbol{X}}_k$를 계산한다.

$$\hat{\boldsymbol{X}}_k = \boldsymbol{GY} \tag{7}$$

여기서 $\hat{\boldsymbol{X}}_k$행렬의 행은 매 반복마다 $2(k+1)$씩 증가한다.

OMP 8 단계 (최소제곱해) : $\hat{\boldsymbol{X}}$를 이용하여 압축 신호 $\hat{\boldsymbol{Y}}$를 생성한다.

$$\hat{\boldsymbol{Y}}_k = \boldsymbol{A_S}\hat{\boldsymbol{X}}_k \tag{8}$$

OMP 9 단계 (나머지) : 수신된 압축 신호에서 추정한 압축 신호 성분을 제거한 나머지 신호 R(residue)을 구한다.

$$\boldsymbol{R_k} = \boldsymbol{Y} - \hat{\boldsymbol{Y}}_k \tag{9}$$

사용자가 설정한 반복횟수 만큼 k를 증가시키면서 1단계부터 다시 반복하고, 최종으로 복원신호의 스펙트럼 $\hat{\boldsymbol{X}}$와 서포트 벡터를 구한다.

3. 관련 연구

OMP 알고리즘을 디지털 로직으로 구현한 다수의 연구[3-7]가 있으며, ASIC 보다는 FPGA를 대상으로 한 연구가 주를 이룬다. 특히 B. Knoop[3]은 Vivado HLS를 활용하여 OMP 알고리즘을 로직으로 설계하고, 기존의 연구와 성능을 비교하여 표로 정리하였다. 비교 기준으로는 OMP 알고리즘 변형여부가 있다. A. Septimus[4]는 OMP 알고리즘을 식 (1) ~ (3)에 해당하는 서포트 벡터를 찾는 단계와 식 (5) ~ (9)에 해당하는 압축신호를 복원하는 단계로 나누었다. 그람-슈미트 직교 과정을 $\boldsymbol{A_S}$행렬에 적용해 직교 행렬 $\boldsymbol{Q_S}$를 구하고, 식 (9)의 연산을 $\boldsymbol{R_k} = (\boldsymbol{I} - \boldsymbol{Q_S Q_S^T})\boldsymbol{R_{k-1}}$으로 유도하였다. 이때, 식 (7)은 OMP 반복 동안 한 번만 계산된다. P. Blache[5]와 H. Rabah[6]은 변형된 촐레스키 분해를 적용하여 루트 연산 없이 역행렬을 계산하였다. 또한 Matlab Simulink을 토대로 하는 Xilinx System Generator를 활용하여 그래픽 인터페이스 환경에서 로직을 설계하였다. L. Bai[7]은 $\boldsymbol{A_S}$ 행렬을 $\boldsymbol{A_S} = \boldsymbol{Q_S R_S}$으로 QR 분해한 뒤 식 (7)을 $\hat{\boldsymbol{X}}_k = \boldsymbol{R_S^{-1} Q^H Y}$으로 단순화하여 계산량을 줄이고, 후진 대입법(back substitution)을 적용하여 $\boldsymbol{R_S^{-1}}$를 계산하였다. B. Knoop[3]은 변형된 그람-슈미트 QR 행렬 분해를 적용하여 압축신호 복원 단계를 수행하고 블록 단위 역행렬 방법을 활용해서 $\boldsymbol{R_S^{-1}}$를 계산하였다. 알고리즘 변형 여부 이외에도 알고리즘 파라미터인 측정행렬의 크기와 Sparsity 값, 로직 처리시간(latency), 동작 주파수, 자원사용량 등이 비교 요소로 사용되었다. 또한 정규화한 제곱평균오차(Normalized Mean Square Error, NMSE)가 컴퓨터 시뮬레이션과 FPGA 수행 결과를 비교하는 성능척도로 사용되었다.

4. 디지털 로직 구현

4.1 제약조건

4.1.1 동작 클록

디지털 신호처리 플랫폼에 사용된 ADC의 표본화 주파수는 248 MHz이다. 디지털 로직 역시 같은 속도로 동작해야 입력 데이터의 손실을 방지할 수 있다.

4.1.2 처리시간(latency) 및 시작주기(initial interval)

Vivado HLS를 활용한 로직의 성능척도는 입력된 데이터가 출력되기 까지 시간인 처리시간과 데이터가 입력되는 시간 간격인 시작주기가 있다. 입력 데이터를 실시간으로 처리하는 로직을 설계 하려면 처리시

간 보다는 시작주기 조건을 만족해야 한다. 신호처리 로직의 시작주기를 전처리 1 단계에 사용되는 Xilinx FFT IP[8]에 따라 결정하였다. 길이가 256이고, 스트리밍 구조를 갖는 Xilinx FFT IP의 시작주기는 1 클록이지만, 256개 표본이 모두 출력되기 까지는 256 클록이 필요하다. 즉, 처리시간이 256 클록이다. 신호처리 로직의 동작클록을 ADC 속도에 맞추어 248 MHz로 설정한 경우, 매 클록은 1/248 MHz ≈ 4 ns이므로 FFT IP의 처리시간은 256 × 4 ns = 1024 ns가 된다. 따라서 FFT IP의 출력을 입력받는 OMP 로직은 256 클록인 1024 ns 시간마다 데이터를 처리할 수 있어야 한다. 하지만 동작 클록이 낮을수록 로직 구현(implementation) 과정이 수월한 것을 고려해 OMP 알고리즘의 동작 주파수를 124 MHz로 낮추어 설정하였다. 이 때 매 클록은 1/124 MHz ≈ 8 ns이며, 시작주기는 1024 ns/8 ns = 128 클록이 되어야 한다.

4.2 고정 소수점 모델링

로직설계에 앞서 먼저 Matlab으로 알고리즘을 구현하고, 비트수 변화에 따른 OMP 알고리즘의 성능을 모의실험 하였다. 실험 결과 14 비트 이상의 고정 소수점 변수를 사용한 경우 95 % 이상의 신호 복원 성능을 확인할 수 있었다. 실험 결과에 따라 결정된 입출력 데이터의 크기와 고정 소수점 형식은 Table 2와 같다.

Table 2. Input and output data format

변수	행렬 크기	형식	비트수/소수	구분
$\boldsymbol{R}$, $\boldsymbol{Y}$	20 × 32	복소수	14 / 8	입출력
$\boldsymbol{A_N^H}$	48 × 20	복소수	14 / 8	입력
$\boldsymbol{A}$	20 × 96	복소수	14 / 8	입력
$\hat{\boldsymbol{X}}$	2(k+1) × 32	복소수	14 / 10	출력

4.3 구조 설계

OMP 알고리즘은 현재의 출력이 다음의 입력으로 사용되는 궤한 구조이기 때문에, 입력 데이터가 내부 과정을 거쳐 출력되기까지 동일 연산이 반복 횟수 k 만큼 반복되어 수행되어야 한다. 따라서 ADC 데이터를 실시간으로 처리하기 위해서 Fig. 1과 같이 전체 신호처리 로직을 파이프라인 구조로 설계하였다. 이 때 k 값은 0과 1이 된다.

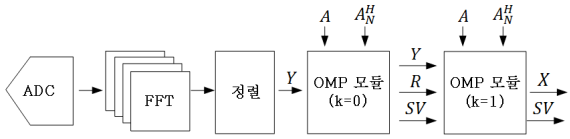

Fig. 1. The proposed pipeline architecture

4.4 입출력 모듈 설계

입출력 데이터는 주로 비트폭이 큰 복소 행렬 이므로, 입출력 포트는 모두 블록램(BRAM) 인터페이스로 설계하였다. Fig. 1을 예로 들면 정렬 블록과 k = 0 OMP 모듈 사이, 두 OMP 모듈 사이, k = 1 OMP 모듈 출력에 각각 데이터를 저장할 블록램이 필요하다.

식 (1) ~ (3)의 정합 연산을 빠르게 처리하기 위해 입력인 $\boldsymbol{Y}$행렬 또는 $\boldsymbol{R}$행렬과 $\boldsymbol{A_n^H}$행렬을 다수의 벡터로 나누어 다량의 데이터가 동시에 입력되도록 하였다. 예를 들어 $\boldsymbol{R}$행렬을 열 방향으로 크기가 20 × 8인 4개의 부분 행렬로 나누고, 각 부분 행렬을 행 방향으로 크기가 1 × 8인 20개의 벡터로 나누면, 매 클록마다 20 × 4 = 80개의 표본을 동시에 모듈로 입력받을 수 있다. 그러면 식 (1)에서 연산의 결과인 $\boldsymbol{Z}$행렬의 원소를 매 클록마다 4개 씩 계산할 수 있어 행렬 곱을 고속으로 처리 할 수 있다.

입력 행렬을 다수의 벡터로 나누는 Vivado HLS directive 소스 코드는 Fig. 2와 같다. 소괄호로 표시한 dim이 1이면 행 방향, 2이면 열 방향으로 variable에 표기된 포트가 개별 벡터로 나뉘어 입력된다.

식 (1)을 계산할 때 정합 모듈에서는 블록램의 동일한 데이터를 여러 번 반복해서 읽는다. 이로 인해 데이터의 병목 지점이 발생하고 불필요한 게이트가 소모 된다. 따라서 한번 읽은 데이터를 로직 내부 버퍼에 저장하여 같은 데이터를 다시 읽지 않도록 하였다.

k = 1 OMP 모듈의 출력인 $\boldsymbol{X}$행렬의 크기 또한 4 × 32이다. $\boldsymbol{X}$행렬을 열 방향으로 크기가 4 × 8인 4개의 부분 행렬로 나누고, 행의 수만큼 출력 포트를 만들어 데이터가 원활히 출력되도록 하였다.

```
#pragma HLS ARRAY_RESHAPE variable=( ) complete dim=( )
```

Fig. 2. Matrix multiplication directive source

4.5 행렬곱 모듈 설계

모든 OMP 하위 모듈에는 행렬곱 연산이 포함되어

있다. 행렬곱 연산은 Fig. 3과 같이 3개의 for 반복문을 이용해서 구현할 수 있다. 예를 들어 크기가 $r \times m$인 행렬과 $m \times c$인 행렬을 곱할 때, 바깥 반복문은 r번, 중간 반복문은 c번, 안쪽 반복문은 m번 수행되어야 한다. Vivado HLS로 행렬곱을 구현할 때, 하드웨어 적으로 가장 효율적인 구조는 모든 연산이 m 반복문에서 수행되도록 하는 것이다.

```
for r
    for c
    #pragma HLS PIPELINE
        for m
        #pragma HLS UNROLL
            ...
        end
    end
end
```

Fig. 3. HLS source code for matrix multiplication

행렬곱 연산은 총 $r \times c \times m$번의 반복이 필요하고, 행렬의 크기가 증가할수록 총 반복 횟수가 배수배로 증가하기 때문에 시작주기 같은 제약조건을 만족할 수 없다. 이때, c 반복문에 파이프라인 구조를 적용하면 m 반복문은 자동으로 풀어지고(unroll), 전체 반복 횟수는 $r \times c$ 만큼 감소하여 처리 속도가 눈에 띄게 향상된다. 대신 m 반복문에 사용되는 자원은 m배 더 증가한다.

이처럼 고속처리가 필요한 경우 반복문 내에 자원 사용량을 늘려 처리 속도를 증가시킬 수 있다. 하지만 로직의 면적이 증가하여 게이트간 라우팅 단계에서 많은 timing violation이 발생할 수 있다. 따라서 제약조건에 맞게 자원 사용량과 행렬 곱 연산 속도의 적절한 균형을 맞추는 작업이 필요하다.

4.6 하위 모듈 상세 설계

신호처리 로직 내부의 전체적인 신호 흐름은 Fig. 4와 같다. 총 7개의 모듈로 구성되어 있으며, 입력 데이터가 순차적으로 모듈을 거치도록 모듈단위 파이프라인 설계(dataflow)를 하였다.

정합 모듈에는 3개의 for 반복문이 있고, 각 반복문의 번호를 바깥부터 차례대로 r, c, m라 할 때, 모든 연산은 m 반복문에서 수행되도록 하였다. Fig. 5에 r과 c반복문 번호에 따라 식 (1) $\boldsymbol{Z}$행렬의 원소가 생성되는 순서를 나타내었다. c 값이 증가 할 때 마다 $\boldsymbol{Z}$행렬의 0행부터 3행까지 4개의 원소가 계산된다. c 반복문의 번호가 31이 될 때 까지 이 값을 화살표 방향으로 누적하여 더한 값을 s라 하면, 그 벡터의 최댓값과 그에 해당하는 r의 번호를 저장한다. 이때 토너먼트 방식을 활용하여 최댓값을 찾았다. 다시 $\boldsymbol{Z}$행렬의 4행부터 7행의 원소를 누적 한 s벡터를 구하고 최댓값을 찾아 이전 최댓값과 비교한다. 이런 방식으로 최댓값을 찾고, 그 값에 해당하는 r값을 서포트 벡터로 지정한다.

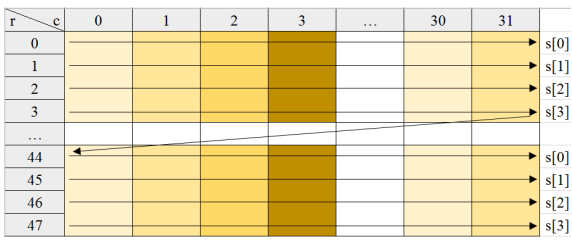

Fig. 5. Example of the matching step operation

입력 $\boldsymbol{Y}$행렬은 정합 모듈 이후 $\boldsymbol{X}$와 $\boldsymbol{R}$행렬을 계산하는 모듈에서도 사용된다. Vivado HLS 매뉴얼[2]에는 모듈간 파이프라인 동작이 수행되려면 한 모듈의 출력이 다음 모듈을 건너뛰어 전달되지 않을 것을 권장하고 있다. 따라서 $\boldsymbol{Y}$행렬을 비롯한 신호들이 되도록

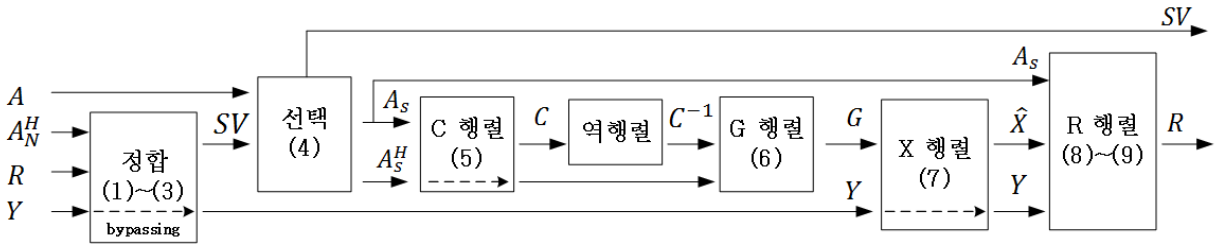

Fig. 4. Internal data flow diagram for the proposed logic

모듈 내부를 그대로 지나쳐 다음 모듈로 전달되도록 설계하였다.

다음 선택 블록에서는 $\boldsymbol{A}$ 행렬에서 서포트 벡터에 해당하는 열만을 가져와 새로운 행렬 $\boldsymbol{A}_S^H$과 $\boldsymbol{A}_S$를 생성한다. 이러한 연산은 대수연산이 필요하지 않다.

다음 블록에서는 선택 블록의 출력을 서로 곱하여 $\boldsymbol{C}$행렬을 생성한다. 정합 모듈과 같이 가장 안쪽 반복문이 풀리도록 두 번째 반복문에 파이프라인 directive를 적용하였다.

역행렬 블록은 $\boldsymbol{C}$행렬의 역행렬을 구한다. 이때, $\boldsymbol{C}$행렬은 정방행렬이고 대각원소는 항상 실수 이며, 대각원소를 기준으로 대칭되는 원소는 서로 복소 켤레(complex conjugate)를 취한 값이다. 또한 $\boldsymbol{C}$행렬의 역행렬 또한 동일한 성질을 갖는다. 이러한 특성을 활용하면 역행렬 연산에 필요한 연산과 로직 자원을 절감할 수 있다. $\boldsymbol{C}$행렬은 반복횟수 k에 따라 크기가 2배씩 증가한다. k = 0인 경우, 행렬의 크기는 2 × 2이고 다음과 같은 역행렬 공식을 적용할 수 있다.

$$C^{-1}=\begin{bmatrix}c_{11}\,c_{12}\\ c_{21}\,c_{22}\end{bmatrix}^{-1}=\frac{1}{c_{11}c_{22}-c_{12}c_{21}}\begin{bmatrix}c_{22} & -c_{12}\\ -c_{21} & c_{11}\end{bmatrix} \tag{10}$$

k = 1인 경우, 행렬의 크기는 4 × 4이고, 적용할 수 있는 역행렬 공식은 다음과 같다.

$$C^{-1}=\begin{bmatrix}c_{11}\,c_{12}\,c_{13}\,c_{14}\\ c_{21}\,c_{22}\,c_{23}\,c_{24}\\ c_{31}\,c_{32}\,c_{33}\,c_{34}\\ c_{41}\,c_{42}\,c_{43}\,c_{44}\end{bmatrix}^{-1}=\frac{1}{\det \boldsymbol{C}}\begin{bmatrix}b_{11}\,b_{12}\,b_{13}\,b_{14}\\ b_{21}\,b_{22}\,b_{23}\,b_{24}\\ b_{31}\,b_{32}\,b_{33}\,b_{34}\\ b_{41}\,b_{42}\,b_{43}\,b_{44}\end{bmatrix} \tag{11}$$

여기서 역행렬 원소 b와 $\det \boldsymbol{C}$의 값은 다음과 같이 계산할 수 있다.

$$\begin{aligned}
b_{11} &= c_{22}c_{33}c_{44}+c_{23}c_{34}c_{42}+c_{24}c_{32}c_{43}-c_{22}c_{34}c_{43}\\
&\quad -c_{23}c_{32}c_{44}-c_{24}c_{33}c_{42}\\
b_{12} &= c_{12}c_{34}c_{43}+c_{13}c_{32}c_{44}+c_{14}c_{33}c_{42}-c_{12}c_{33}c_{44}\\
&\quad -c_{13}c_{34}c_{42}-c_{14}c_{32}c_{43}\\
b_{13} &= c_{12}c_{23}c_{44}+c_{13}c_{24}c_{42}+c_{14}c_{22}c_{43}-c_{12}c_{24}c_{43}\\
&\quad -c_{13}c_{22}c_{44}-c_{14}c_{23}c_{42}\\
b_{14} &= c_{12}c_{24}c_{33}+c_{13}c_{22}c_{34}+c_{14}c_{23}c_{32}-c_{12}c_{23}c_{34}\\
&\quad -c_{13}c_{24}c_{32}-c_{14}c_{22}c_{33}\\
b_{21} &= c_{21}c_{34}c_{43}+c_{23}c_{31}c_{44}+c_{24}c_{33}c_{41}-c_{21}c_{33}c_{44}\\
&\quad -c_{23}c_{34}c_{41}-c_{24}c_{31}c_{43}\\
b_{22} &= c_{11}c_{33}c_{44}+c_{13}c_{34}c_{41}+c_{14}c_{31}c_{43}-c_{11}c_{34}c_{43}\\
&\quad -c_{13}c_{31}c_{44}-c_{14}c_{33}c_{41}\\
b_{23} &= c_{11}c_{24}c_{43}+c_{13}c_{21}c_{44}+c_{14}c_{23}c_{41}-c_{11}c_{23}c_{44}\\
&\quad -c_{13}c_{24}c_{41}-c_{14}c_{21}c_{43}\\
b_{24} &= c_{11}c_{23}c_{34}+c_{13}c_{24}c_{31}+c_{14}c_{21}c_{33}-c_{11}c_{24}c_{33}\\
&\quad -c_{13}c_{21}c_{34}-c_{14}c_{23}c_{31}\\
b_{31} &= c_{21}c_{32}c_{44}+c_{22}c_{34}c_{41}+c_{24}c_{31}c_{42}-c_{21}c_{34}c_{42}\\
&\quad -c_{22}c_{31}c_{44}-c_{24}c_{32}c_{41}\\
b_{32} &= c_{11}c_{34}c_{42}+c_{12}c_{31}c_{44}+c_{14}c_{32}c_{41}-c_{11}c_{32}c_{44}\\
&\quad -c_{12}c_{34}c_{41}-c_{14}c_{31}c_{42}\\
b_{33} &= c_{11}c_{22}c_{44}+c_{12}c_{24}c_{41}+c_{14}c_{21}c_{42}-c_{11}c_{24}c_{42}\\
&\quad -c_{12}c_{21}c_{44}-c_{14}c_{22}c_{41}\\
b_{34} &= c_{11}c_{24}c_{32}+c_{12}c_{21}c_{34}+c_{14}c_{22}c_{31}-c_{11}c_{22}c_{34}\\
&\quad -c_{12}c_{24}c_{31}-c_{14}c_{21}c_{32}\\
b_{41} &= c_{21}c_{33}c_{42}+c_{22}c_{31}c_{43}+c_{23}c_{32}c_{41}-c_{21}c_{32}c_{43}\\
&\quad -c_{22}c_{33}c_{41}-c_{23}c_{31}c_{42}\\
b_{42} &= c_{11}c_{32}c_{43}+c_{12}c_{33}c_{41}+c_{13}c_{31}c_{42}-c_{11}c_{33}c_{42}\\
&\quad -c_{12}c_{31}c_{43}-c_{13}c_{32}c_{41}\\
b_{43} &= c_{11}c_{23}c_{42}+c_{12}c_{21}c_{43}+c_{13}c_{22}c_{41}-c_{11}c_{22}c_{43}\\
&\quad -c_{12}c_{23}c_{41}-c_{13}c_{21}c_{42}\\
b_{44} &= c_{11}c_{22}c_{33}+c_{12}c_{23}c_{31}+c_{13}c_{21}c_{32}-c_{11}c_{23}c_{32}\\
&\quad -c_{12}c_{21}c_{33}-c_{13}c_{22}c_{31}
\end{aligned} \tag{12}$$

$$\det \boldsymbol{C}=c_{11}b_{11}+c_{12}b_{21}+c_{13}b_{31}+c_{14}b_{41} \tag{13}$$

이때, $\boldsymbol{C}$ 행렬의 특성에 따라 $\det \boldsymbol{C}$는 실수 값이 된다. 고정소수점 $\det \boldsymbol{C}$의 역수 연산은 부동소수점 연산기를 사용하여 구현하였다.

다음 $\boldsymbol{G}$ 및 $\boldsymbol{X}$행렬 계산 블록은 모두 복소 행렬 곱으로, 정합 모듈과 유사하게 구현 하였다.

$\boldsymbol{R}$ 행렬 계산 모듈에서는 OMP 알고리즘의 8, 9단계가 같이 수행되도록 설계하였다. 정합 모듈과 같이 4개의 표본이 동시에 계산되도록 설계하였다.

5. 로직 최적화

불필요한 로직자원의 소모를 줄이고 게이트 간 경로가 원활히 지정되도록 최적화를 수행하였다.

일반적인 복소곱 연산에는 4개의 곱셈기가 필요하다. 하지만, 복소곱 연산을 식 (14)와 같이 변경하면 3개의 곱셈기로도 연산이 가능하여 곱셈기 자원을 절약할 수 있다. Vivado HLS DSP 라이브러리에는 3개의 곱셈기

를 사용하는 복소 곱셈기인 hls::cmpy 함수가 있다.

$$W=(A_r+jA_i)(B_r+jB_i) \quad (14)$$
$$=A_rB_r-A_iB_i+j(A_rB_i+A_iB_r)$$
$$=(P_1-P_0)+j(P_1+P_2)$$

여기서 $P_0=(A_r+A_i)B_i$, $P_1=(B_r+B_i)A_r$, $P_2=(A_i-A_r)B_r$, 이다.

설계한 로직의 처리시간과 시작주기가 주어진 제약조건을 만족했더라도 구현 과정에서 라우팅이 제대로 되지 않아 과도한 timing violation이 발생할 수 있다. 이런 경우 FPGA에서 로직이 제대로 동작하지 않는다. 이 문제를 해결하고자, 행렬곱 반복문 내의 시작주기를 증가시켰다. 그러면 Vivado HLS가 반복문 내 연산을 여유 있게 스케줄링 하여 게이트 간 critical path가 감소하는 효과가 있다. 대신 반복문의 처리시간이 증가하여 시작주기 제약 조건에 어긋날 수 있다. 하지만 반복문에 포함된 로직 사용량을 늘려서 이를 해결할 수 있다.

Fig. 6은 게이트간 critical path를 완화하는 Vivado HLS directive 소스코드이다. 시작주기를 1에서 2로 증가 시키면 처리시간 또한 2배 증가하지만, UNROLL factor directive를 통해 반복문의 반복횟수를 factor배 만큼 줄임으로써 자원 사용량을 2배 증가시켜 모듈의 처리시간을 그대로 유지할 수 있다.

```
#pragma HLS PIPELINE II=2
#pragma HLS UNROLL factor=2
```

Fig. 6. HLS directive source for efficient routing

6. 로직 구현 결과

6.1 합성(synthesis) 결과

Vivado HLS 합성 시 적용한 주요 옵션은 Table 3과 같다. Vivado에서 구현 시 발생하는 timing violation을 최대한 줄이기 위해 bind, schedule, uncertainty 3가지 옵션을 설정하였다.

신호처리 모듈 합성 결과를 요약하면 Table 4와 같다. 표에서 Clock은 목표 클록 이고, Est. 값은 최대로 설정 가능한 클록을 나타낸다. 시작주기(interval)가 제약사항인 128 클록 이하이며, 모듈 간 파이프라인 기법이 적용된 것을 알 수 있다.

Table 3. Vivado HLS synthesis options

Command	Parameters
Config_bind	effort = high
Config_schedule	
Clock period	7.142 ns
Uncertainty	30 %
Part	xcku115-flvb2104-2-i

Table 4. Vivado HLS synthesis summary

Loop	Clock/Est.	Latency	Interval	Type
0	7.14 / 4.98	131 clk	106 clk	dataflow
1	7.14 / 5.61	124 clk	104 clk	dataflow

Table 5. Sub-module performances

Loop	SubModule	Latency	BRAM	DSP	FF	LUT
k=0	Matching	105	0	240	23599	29311
	Selection	4	0	0	1243	3863
	C Matrix	12	0	30	2072	2055
	C^{-1} Matrix	23	0	7	1031	2639
	G Matrix	8	0	30	4712	3204
	X Matrix	12	0	60	3610	8027
	R Matrix	12	0	60	6294	8124
k=1	Matching	103	0	240	27716	36696
	Selection	7	0	0	2860	8201
	C Matrix	20	0	60	3998	4213
	C^{-1} Matrix	41	0	292	10358	14991
	G Matrix	12	0	60	8840	8880
	X Matrix	20	0	240	13023	13607

Table 5에는 하위 모듈 별 처리시간과 자원사용량 값을 막대그래프와 함께 나타내었다. 두 OMP 모듈 모두 정합 단계에서 240개 이상의 DSP가 소모되고, 100 클록 이상의 처리시간이 필요한 것을 확인할 수 있다. 주목할 점은 역행렬 계산에 필요한 DSP 사용량이 k = 0 모듈 대비 k = 1 모듈에서 대폭 증가한 것이다. 이는 C행렬의 크기가 2배 증가했기 때문으로, OMP 알고리즘에서 정합과 역행렬 계산에 많은 FPGA

자원과 처리시간이 소모됨을 알 수 있다. 또한 파이프라인 설계가 적용되어 Table 4에 제시된 상위 OMP 모듈의 처리시간 보다 하위 OMP 모듈 간의 처리시간의 합이 큰 것을 확인할 수 있다.

6.2 C/RTL cosimulation 결과

Vivado HLS에는 C와 HDL을 동시에 시뮬레이션 할 수 있는 C/RTL cosimulation 기능이 있으며, 그 결과를 Table 6에 나타내었다. 이때, Table 4와 Table 6의 처리시간과 입력주기 값이 서로 다른 것을 주의해야 한다. 주로 C/RTL cosimulation의 시작주기 값이 합성 결과 값보다 크게 나오며, 반드시 C/RTL cosimulation 결과 값을 기준으로 설계를 해야 한다. Fig. 7은 모의 압축 신호에 대해 Vivado HLS와 Matlab에서 복원한 신호의 실수부와 허수부 스펙트럼을 나타낸 것이다. 실선은 Vivado HLS 결과이고 점선은 Matlab 결과이다. 두 값의 NMSE는 약 1.8×10^{-3}으로 계산되었다.

Table 6. C/RTL cosimulation result

Loop	Latency	Interval
0	183 clk	111 clk
1	211 clk	112 clk

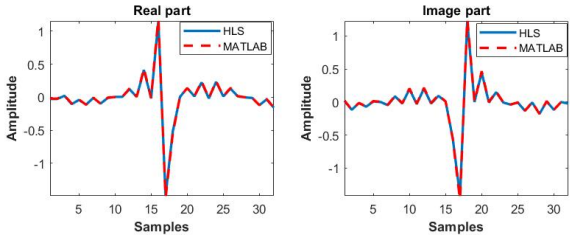

Fig. 7. Result comparison of HLS and Matlab

6.3 타이밍 다이어그램

6.3.1 입력 신호

Fig. 8에는 OMP 모듈의 입력 데이터 타이밍을 나타내었다. 설계한 로직의 제어신호에는 동작의 완료 알리는 ap_done, 시작 신호인 ap_start, 로직 대기상태를 알려주는 ap_idle, 입력을 받을 준비가 되었음을 알리는 ap_ready, 클록 입력인 ap_clk가 있다. ap_start가 high가 되면 로직이 동작하기 시작한다. 그리고 ap_done 신호가 high가 되기 전에 ap_ready 신호가 high가 되어 새로운 데이터가 입력되는 것을 확인할 수 있다. 이는 모듈단위 파이프라인 구조가 제대로 동작을 하고 있음을 의미한다. 시작주기는 ap_ready 신호가 high가 됐을 때의 시간 T1과 T2간의 차이에 해당하고, 처리시간은 ap_start와 ap_done 신호가 high가 될 때의 시간인 L1과 L2의 차가 된다.

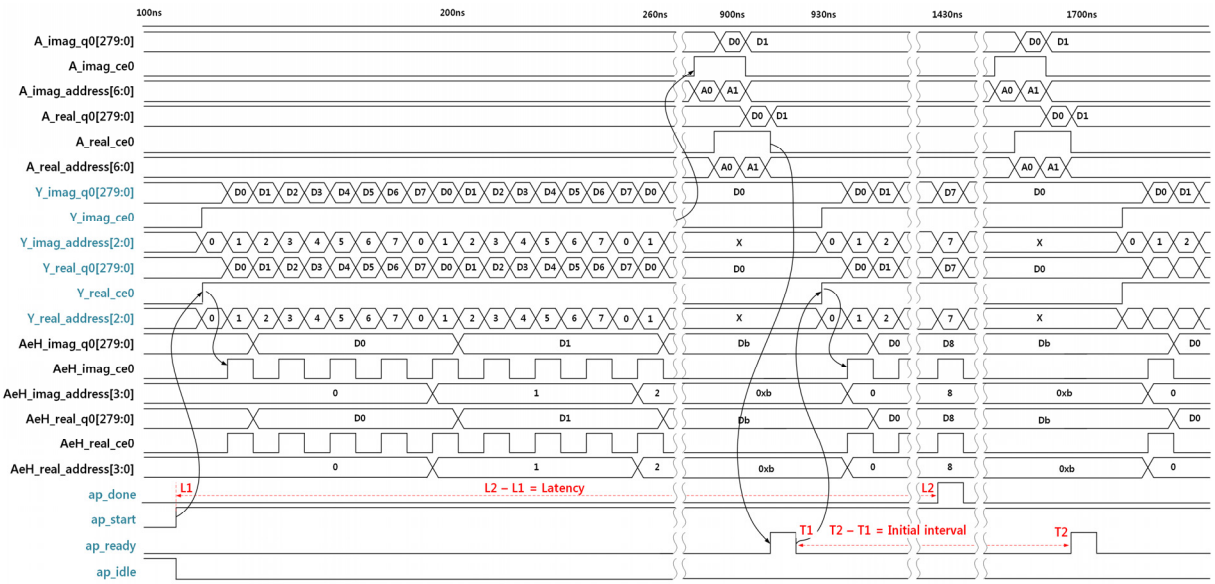

Fig. 8. Timing diagram of the input data with k = 0

데이터 $\boldsymbol{Y}$는 총 4개의 포트를 통해 병렬로 입력되지만 그림에는 단일 포트만 나타내었다. 또한 $\boldsymbol{Y}$ 데이터는 복소수 이므로 각각 실수와 허수 포트로 나뉜다. 데이터를 읽어올 블록램을 제어하기 위한 주소인 address[2:0]와 ce(chip enable) 신호가 동시에 high가 되고 한 클록 뒤에 실제 데이터가 연이어 입력되는 것을 확인할 수 있다. 데이터 포트 q0[279:0]를 통해서는 Fig. 2의 영향으로 한 표본 당 14비트인 20개의 표본이 280비트 씩 동시에 입력되는 것을 확인할 수 있다.

데이터 $\boldsymbol{A}_n^H$ 역시 4개의 AeH 포트를 통해 병렬로 입력되며, 각 포트 마다 실수와 허수 값을 입력 받는다. 5절에서 언급한 타이밍 완화 기법이 적용되어, $\boldsymbol{Y}$ 데이터와는 달리 ce 신호가 띄엄띄엄 high가 되는 것을 확인할 수 있다.

데이터 $\boldsymbol{A}$는 ce가 high일 때, $\boldsymbol{A}$행렬을 저장한 블록램의 address[6:0] 번지에서 한 클록 뒤에 q0[279:0] 포트를 통해 각각 280비트씩 연이어 입력되는 것을 확인할 수 있다. $\boldsymbol{A}$행렬의 열 크기가 96이므로 블록램의 주소는 7비트가 할당되었다.

6.3.2 출력 신호

Fig. 9에서 볼 수 있듯이 대략 900 ns 시점에서 정합 모듈의 결과인 서포트 벡터 값이 valid 신호와 함께 sv[31:0] 포트를 통해 출력 된다. 그 후 $\boldsymbol{Y}$ 데이터가 d[279:0] 포트를 통해 출력 되고, 연이어 $\boldsymbol{R}$ 데이터가 출력 된다. 출력 데이터를 저장할 블록램의 입력인 ce 신호와 주소 address[2:0]가 함께 출력되는 것을 확인할 수 있다. $\boldsymbol{Y}$와 $\boldsymbol{R}$행렬 모두 4개 포트로 출력 되지만 그림에는 한 개의 포트만 타나내었다. 모든 출력이 완료되면 ap_done 신호가 high가 되는 것을 확인할 수 있다. 이 후 대략 900 ns에 입력된 데이터의 서포트 벡터 값이 1700 ns 부근에서 출력되기 시작한다.

6.4 기존 연구와의 비교

논문에서 제안한 로직과 B. Knoop[3] 로직의 성능을 비교한 결과를 Table 7에 나타내었다. 제안한 로직의 알고리즘 파라미터는 N = 48, M = 20, K = 2이고, Kintex Ultrascale FPGA(xcku115-flvb2104-2-i)를 타겟으로 하였다. B. Knoop[3]은 N = 128 M = 32, K = 5에 대해 Virtex7 FPGA를 사용하였다. 제안한 로직은 B. Knoop[3]에 비해 블록램(BRAM)은 16배, DSP 코어 약 2.5배, 플립플롭(FF)은 약 5.6배, 룩업테이블(LUT)은 약 9.7배로 보다 많은 자원을 사용하지만 처리시간은 약 10배 빠른 것을 알 수 있다. 이는 실시간 처리를 목표로 디지털 로직을 설계했기 때문이다.

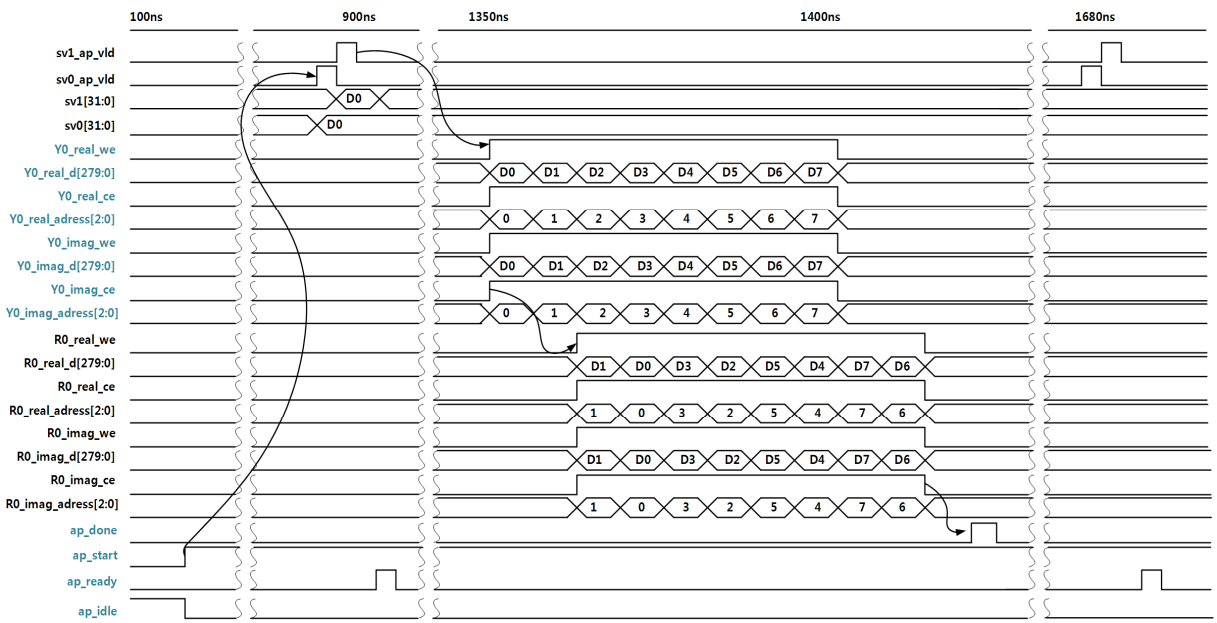

Fig. 9. Timing diagram of the input data with k = 0

Table 7. Comparison of implementation results of the proposed architecture and the existing design

Reference	Problem Size			Time (μs)	Freq. (MHz)	Format (Qm.n)	Target	Resource Utilization				NMSE
	M	N	K					BRAM	DSP	FF	LUT	
[3]	32	128	5	10.7	103	Q4.14	Virtex-7	4	518	22564	18330	1.4×10^{-7}
Proposed	20	48	2	1.024	124	Q4.14	Kintex Ultrascale	64	1319	126057	177883	1.8×10^{-3}

7. 결 론

본 논문에서는 orthogonal matching pursuit 알고리즘을 바탕으로 압축신호를 실시간으로 탐지하고 복원하는 신호처리 로직을 Xilinx Vivado HLS를 활용하여 설계하였다. 설계한 로직은 124 MHz의 속도로 동작하며, 실시간으로 압축신호를 입력받아 1.024 μs 시간단위로 원 신호의 스펙트럼을 복원할 수 있다. 타깃 FPGA인 Xilinx Kintex ultarscale 은 두 개의 실리콘 웨이퍼가 적층되어 있는 구조로 단일 웨이퍼 기준 DSP 자원 소모량은 60 %, 플립플롭은 18 %, 룩업테이블은 55 % 인 것을 확인할 수 있었다.

후 기

이 연구는 방위사업청 및 국방과학연구소의 재원을 지원받아 수행되었습니다.

References

[1] Moshe Mishali and Yonina C. Eldar, “From Theory to Practice: Sub-Nyquist Sampling of Sparse Wideband Analog Signals,” IEEE J. Select. Top. Signal Process., Vol. 4, No. 2, pp. 375-391, 2010.

[2] Vivado Design Suite User Guide: High-Level Synthesis(UG902), Xilinx, 2017.

[3] B. Knoop, J. Rust, S. Schmale, D. Peters-Drolshagen, and S. Paul, “Rapid Digital Architecture Design of Orthogonal Matching Pursuit,” 24th European Signal Processing Conference(EUSIPCO), pp. 1857-1861, 2016.

[4] A. Septimus and R. Steinberg, “Compressive Sampling Hardware Reconstruction,” Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 3316-3319, May, 2010.

[5] P. Blache, H. Rabah, and A. Amira, “High-Level Prototyping and FPGA Implementation of the Orthogonal Matching Pursuit Algorithm,” 11th International Conference on Information Science, Signal Processing and their Applications(ISSPA), pp. 1336-1340, July, 2012.

[6] H. Rabah, A. Amira, B. K. Mohanty, S. Almaadeed, and P. K. Meher, “FPGA Implementation of Orthogonal Matching Pursuit for Compressive Sensing Reconstruction,” IEEE Transactions on Very Large Scale Integration(VLSI) Systems, Vol. 23, No. 10, pp. 2209-2220, Oct., 2015.

[7] L. Bai, P. Maechler, M. Muehlberghuber, and H. Kaeslin, “High-Speed Compressed Sensing Reconstruction on FPGA using OMP and AMP,” 19th IEEE International Conference on Electronics, Circuits and Systems(ICECS), pp. 53-56, Dec., 2012.

[8] Fast Fourier Transform v9.0 LogiCORE IP Product Guide(PG109), Xilinx, 2017.