"내 칩 제작 서비스" 오픈-소스 디자인 킷/C++에서 GDS 까지:

FIR 필터 설계 (1 부)

연구과제명	반도체 기술 개발 지원 고경력 전문인력 활용 사업(25JB1710)
연구기간	2025 년 6 월~2026 년 12 월
연구책임자	고상춘
기록자	국일호
확인자	
작성일자	2025 년 7 월 19 일

ETRI 한국전자통신연구원
www.etri.re.kr

"내 칩 제작 서비스" 오픈-소스 디자인 킷/연구노트9: C++에서 GDS까지

FIR 필터 설계 (1부: RTL)

목차

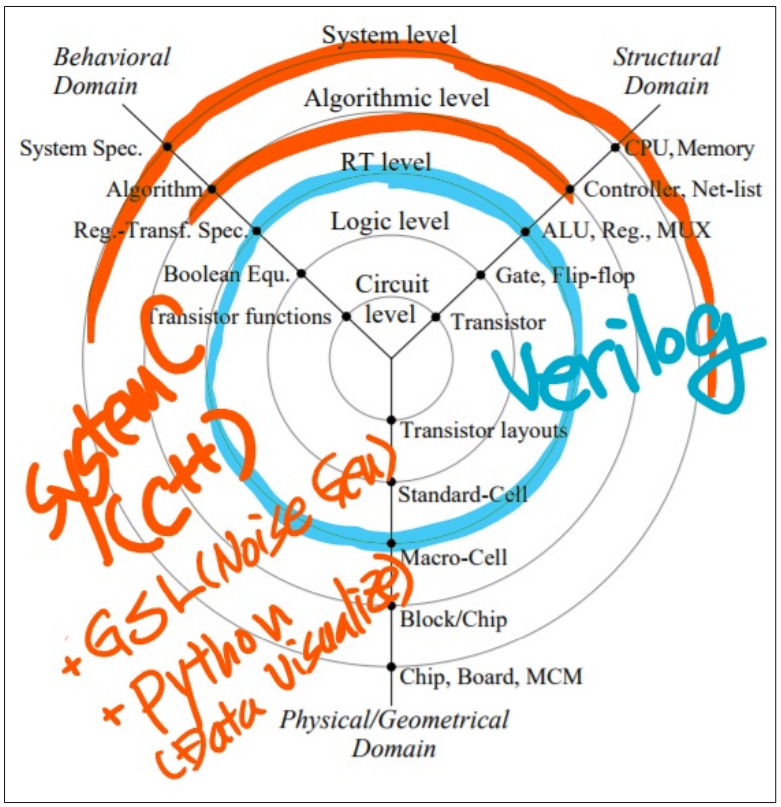
---

by GoodKook, goodkook@gmail.com

I. 개요

앞서 D-플립플롭을 트랜지스터 수준에서 설계하고 레이아웃을 그려 봤다[연구노트8]. 하드웨어 기술 언어(HDL)의 코딩 스타일을 다루면서 RTL과 그 예로서 쉬프트 레지스터를 학습 했다[연구노트3]. 이어 RTL 설계의 검증을 위해 높은 추상화 수준의 테스트벤치 작성의 필요성을 이해했다[연구노트4]. 컴퓨팅 언어가 발달하여 급기야 인간의 언어(사고방식)에 가깝게 되어 "객체 지향적"이라는 말이 자연스럽다. 프로그래밍 언어로서 C++는 최고의 유연성과 넓은 추상화 수준을 갖추게 되었다. SystemC는 C++를 시스템(하드웨어 및 소프트웨어) 설계에 현명하게 적용한 템플릿 크래스 라이브러리로 IEEE1666의 표준이다.

반도체 설계의 추상화 수준을 알고리즘 수준까지 끌어올려 본다. 디지털 FIR 필터를 C++ 언어로 기술하고 검증하며 이를 베릴로그(Verilog) RTL로 설계한다. SystemC의 타임드 모델링으로 FIR 필터의 파이프라인 병렬처리 구조를 탐색하고 테스트 벤치를 작성한다. 디지털 신호처리 알고리즘을 검증하기 위해 매우 긴 테스트 신호가 생성되어야 한다. 필터의 성능 시험을 위한 잡음은 C++ GSL(GNU Scientific Library)을 활용하여 생성한다. 테스트 벤치에 C++와 SystemC 를 사용함으로써 얻는 혜택은 그동안 쌓인 라이브러리의 활용도 있다. 발전된 컴퓨터 성능을 활용하는 현대 알고리즘들은 대용량의 입출력 신호를 다룬다. 디지털 신호처리(DSP, Digital Signal Processing)도 그중 하나다. 이번에 학습할 디지털 FIR 필터 역시 매우 방대한 량의 입출력 신호를 처리한다. 많은 양의 입출력 신호의 시각화(visualization)에 파이썬(Python)을 활용한다. 본 학습의 목표는 C++언어로 기술된 FIR 필터로부터 파이프 라인 구조를 도출하고 PE(Processing Element)를 베릴로그로 설계하여 최종적으로 "내 칩 제작 서비스"를 통해 칩으로 제작할 수 있는 레이아웃 도면 GDS를 생성하는 과정을 다룬다. RTL 설계에 더하여 시스템 수준 검증 기법을 포함한다.



II. 설계 사양(Design Specification)

설계하려는 FIR 필터의 시스템 사양은 다음과 같다.

설계 목표: 디지털 데이터의 저역 통과 필터 설계 (Digital Low-Pass Filter)
입력: 8 비트 부호 없는 디지털 데이터 (8-bit unsigned integer)
출력: 16비트 부호 없는 디지털 데이터(16-bit unsigned integer)
필터 특성:
    샘플링 주파수: 4800Hz
    차단 주파수: 1000Hz
    차단 이득: -20dB 이상
알고리즘: FIR
입출력 인터페이스: 1샘플/1클럭 (파이프라인)

위의 사양을 만족하는 필터를 디지털 하드웨어로 설계한다. 이 사양서에 입출력 신호의 인터페이스 조건(In/Out interface timing requirement)에 유의한다. RTL 설계에 앞서 컴퓨팅 언어 C++ 로 알고리즘을 기술하고 필터의 요구 사양(필터 특성)을 만족하는지 시험한다. C++로 작성하고 시험한 코드는 RTL 설계시 검증용 참고(golden reference)로 쓰인다. 문서가 아닌 실행형 설계사양(executable specification)이다.

II-1. FIR 필터 알고리즘(FIR filter algorthm)

DSP의 디지털 FIR 필터는 궤환이 없는 유한 길이의 선형성을 갖는 알고리즘으로 하드웨어 구현이 용이하다. 배열 구조 병렬처리(array parallel processing)에 적용되는 전형적인 예이기도 하다. 이산(discrete-time) 계수 곱의 합으로 표현하면 아래와 같다.

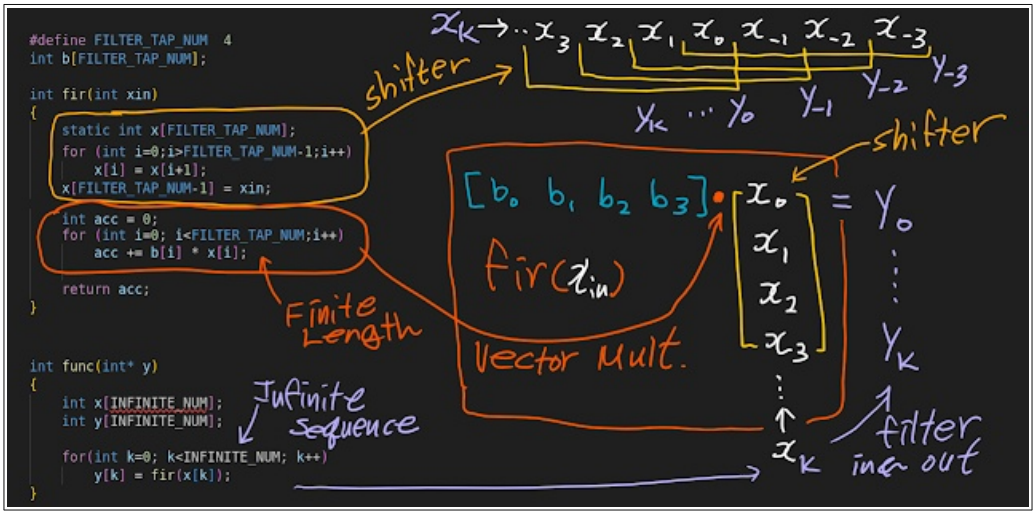
$$y[n] = b_0 x[n] + b_1 x[n-1] + \cdots + b_N x[n-N]$$
$$= \sum_{i=0}^{N} b_i \cdot x[n-i],$$

where:

- $x[n]$ is the input signal,
- $y[n]$ is the output signal,
- N is the filter order; an N^{th}-order filter has $N+1$ terms on the right-hand side
- b_i is the value of the impulse response at the i'th instant for $0 \le i \le N$ of an N^{th}-order FIR filter. If the filter is a direct form FIR filter then b_i is also a coefficient of the filter.

[출처] Finite Impulse Response, https://en.wikipedia.org/wiki/Finite_impulse_response

이산 FIR 필터의 수식을 따지고 보면 매우 단순한 행렬의 벡터 곱(vector multiplication)이다. 어렵지 않게 C++ 언어로 구현할 수 있다.

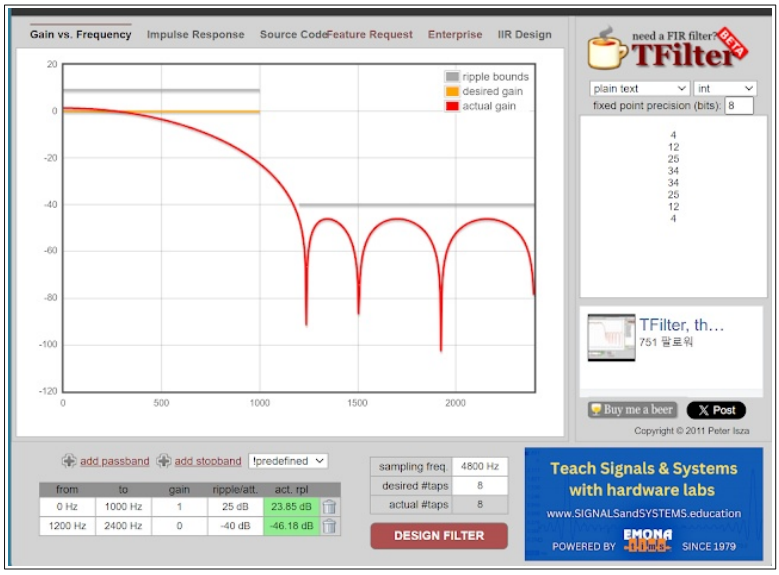


입력 벡터에 곱해지는 계수에 따라 다양한 효과를 볼 수 있다. 결국 FIR 디지털 필터의 설계는 이 계수를 구하는 것이다. 필터의 사양(주파수 특성)에 따라 계수가 구해지면 상수로 고정된다. 응용에 따라 이 계수 테이블을 가변하여 다양한 임무를 수행 할 수 있다. 필터 계수 구하기에 복잡한 수학이 동원된다.

[주] 입력에 대하여 계수를 곱하는 계산 구조로 따지면 인공지능의 신경망에서 추론(inference)에 사용하는 뉴런과 다를 바 없다[But what is a Neural Network?]. 학습(training)은 이 계수를 효과적으로 갱신해 내는 것으로 상당한(복잡한) 계산이 요구된다.

II-2. FIR 필터 계수 구하기

필터 계수 구하기는 이번 학습의 범위가 아니므로 외부 도구의 도움을 받기로 한다. 필터의 계수는 T-Filter 웹 도구를 통하여 구한다.



[출처] T-Filter, Free on-line FIR Filter Design

저역 통과 필터(Low-Pass Filter)로서 탭의 갯수는 8개, 샘플링 주파수 4.8Khz인 입력 신호에 대하여 차단 주파수는 1kHz로 잡았다. 필터 계산은 8-비트 정수형으로 한다. 설계 옵션에 필터 탭의 수가 작고 정밀도가 낮아 계수가 갖는 수의 범위가 크지 않다. 취급할 수의 범위(dynamic range of numbers)에 따라 비트 폭(bit-width)이 결정된다. 이는 결국 하드웨어의 규모와 소모 전력 등에 영향을 미치게 되므로 설계사양(design specification)의 중요한 요소다.

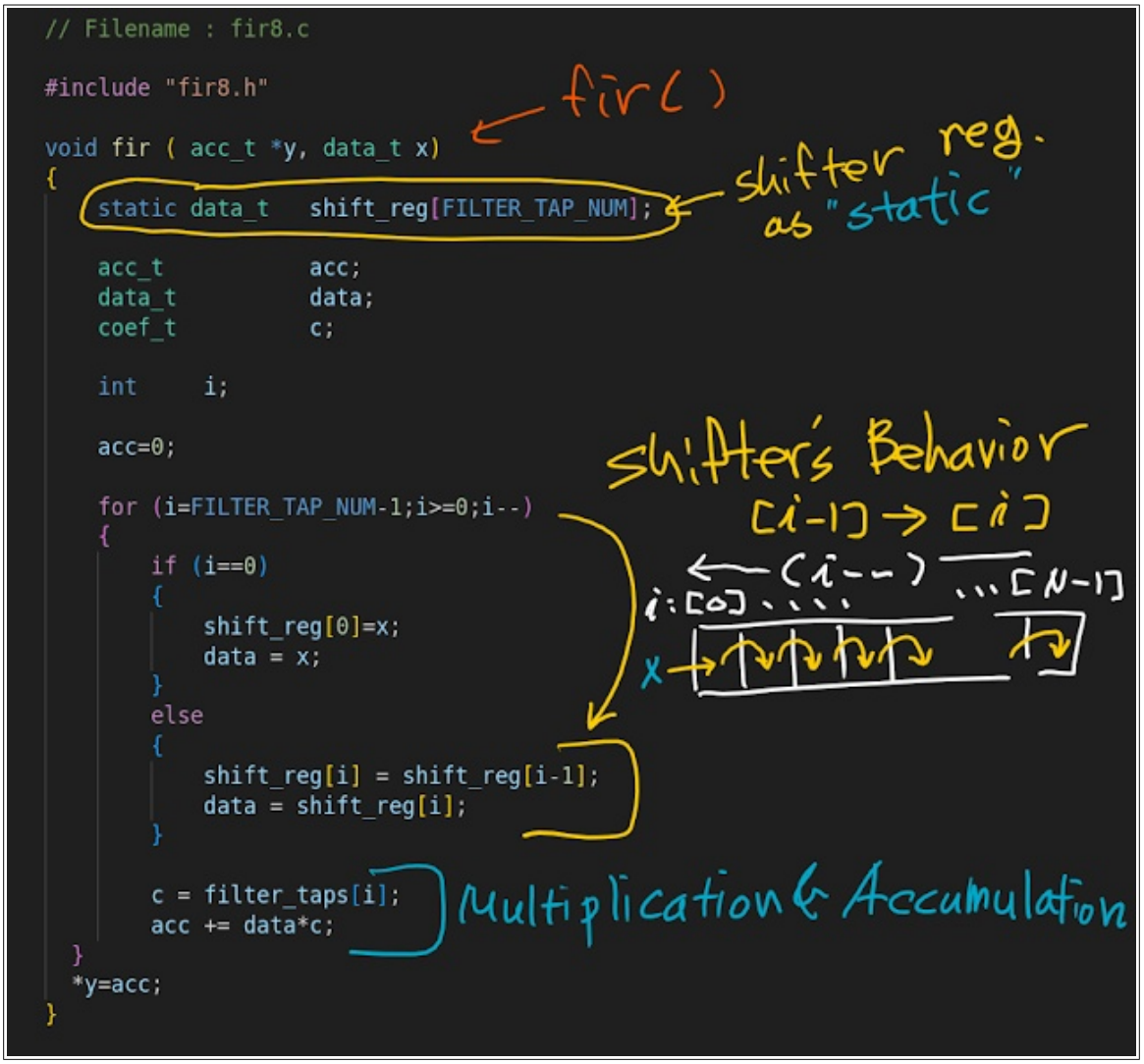
II-3. FIR 필터의 언-타임드 C++ 코드(Un-timed C++ code)

앞서 웹 도구 T-Filter를 통해 얻은 계수를 가지고 필터의 성능(차단 대역 등 설계 요구 사양에 부합하는지) 검증을 위해 C++ 코드를 작성한다. 하드웨어 설계를 검증할 때 비교 표준(golden reference)으로 사용될 것이다. 필터의 계수들을 filter_taps[FILTER_TAP_NUM]에 저장해 둔 헤더 파일 fir8.h 은 아래와 같다.

```
// Filename : fir8.h

#ifndef _FIR8_H_
#define _FIR8_H_

#include <stdint.h>

#define F_SAMPLE        4800
#define FILTER_TAP_NUM  8

static uint8_t filter_taps[FILTER_TAP_NUM] = {
  4, 12, 25, 34, 34, 25, 12, 4 };

typedef uint8_t   coef_t;
typedef uint8_t   data_t;
typedef uint16_t  acc_t;

void fir( acc_t* y, data_t x);

#endif
```

FIR 필터 계산의 C++ 코드는 아래와 같다[fir8.cpp]. 쉬프트 레지스터와 계수 곱의 누적을 for 반복문 안에 기술하였다. 즉시 할당과 순차 실행되는 C++에서 쉬프트 레지스터를 표현하기 위해 반복문의 색인 순서(역순이다)에 유의한다. 쉬프트 레지스터 shift_reg[]는 함수 fir() 되돌려진 후에도 계속 값을 유지하고 있어야 하므로 정적 할당(static allocation) 되었다.

> [주] 논리회로를 배울 때를 상기해보자. 소위 "카운터"와 "쉬프터"라는 디지털 회로를 가장 많이 다뤘었다. 이 두 회로가 조합논리와 결합된 순차회로의 대표적인 예이기도 하지만 컴퓨터 시스템을 구성하는 가장 널리 응용되는 회로이기 때문이다. CPU의 "프로그램 카운터", 대부분 알고리즘에 등장하는 반복문 제어의 중심은 "카운터 회로"다. 신호처리 알고리즘에서 이산 차동식, x[t] ← x[t-1]의 개념, 파이프라인 병렬처리 알고리즘의 하드웨어 구현은 결국 "쉬프터 회로"다.



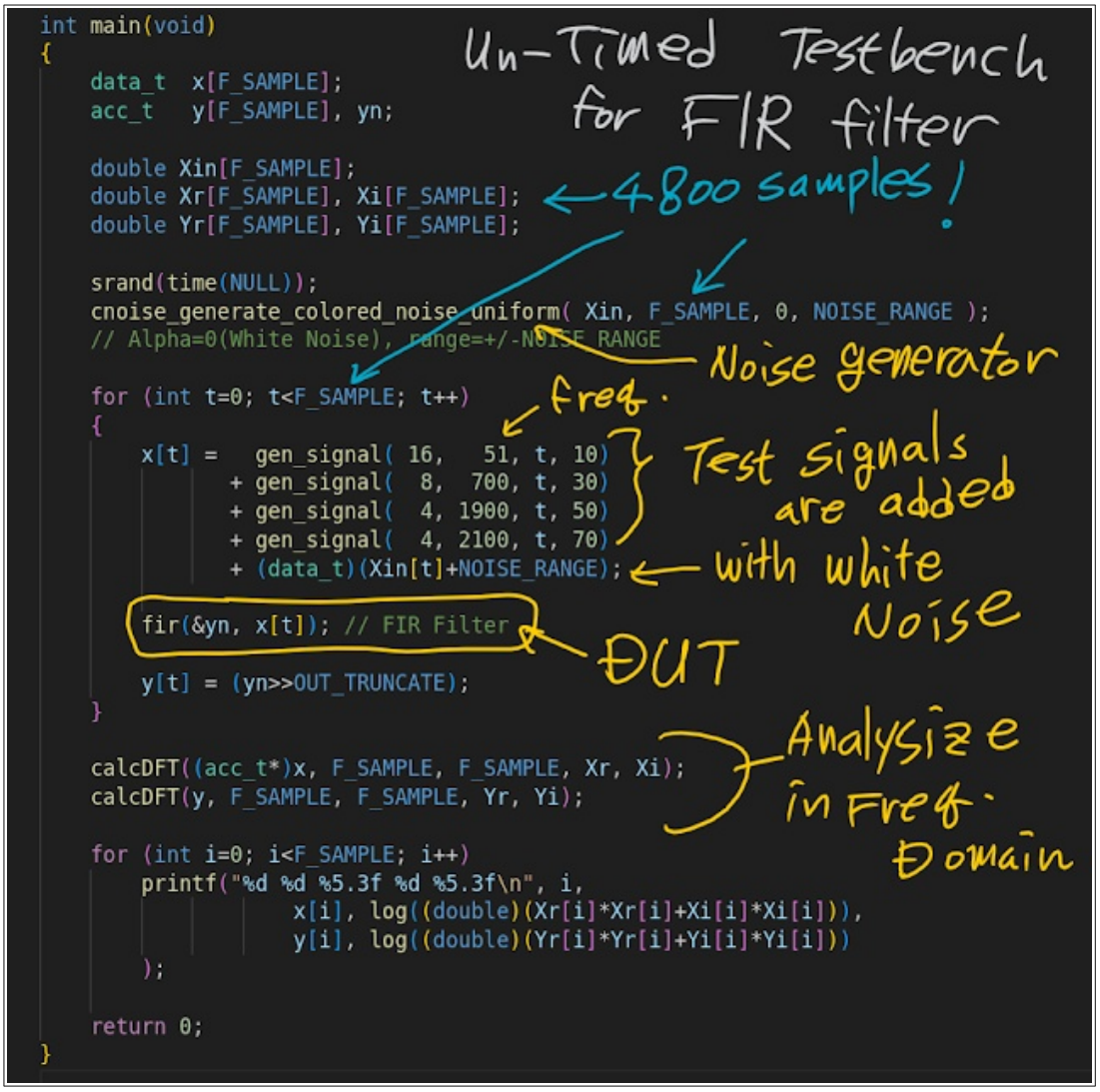
하드웨어 설계를 목표로 하고 있지만 알고리즘 개발 단계에서 C++ 코드는 클럭과 병렬성의 개념이 포함되지 않은 언-타임드(un-timed) 추상적 모델(abstracted model)이다.

II-4. FIR 필터의 언-타임드 C++ 테스트 벤치(Un-timed testbench)

웹 도구로 구한 필터 계수를 사용하여 작성한 FIR 필터의 C++ 코드를 시험하기 위한 테스트 벤치를 작성한다. 예제에서 파일명은 fir8_tb.cpp 다. 입력으로 사용할 대량의 잡음 신호는 cnoise* 라이브러리를 사용하였다. 이 C++ 는 백색 잡음 생성에 GSL(GNU scientific Library)**를 활용한다.

* cnoise, Miroslav Stoyanov, Oak Ridge National Laboratory, https://people.sc.fsu.edu/~jburkardt/c_src/cnoise/cnoise.html

** GSL(GNU scientific library), https://www.gnu.org/software/gsl/




```
int main(void)
{
    data_t  x[F_SAMPLE];
    acc_t   y[F_SAMPLE], yn;

    double Xin[F_SAMPLE];
    double Xr[F_SAMPLE], Xi[F_SAMPLE];
    double Yr[F_SAMPLE], Yi[F_SAMPLE];

    srand(time(NULL));
    cnoise_generate_colored_noise_uniform( Xin, F_SAMPLE, 0, NOISE_RANGE );
    // Alpha=0(White Noise), range=+/-NOISE_RANGE

    for (int t=0; t<F_SAMPLE; t++)
    {
        x[t] =   gen_signal( 16,   51, t, 10)
               + gen_signal(  8,  700, t, 30)
               + gen_signal(  4, 1900, t, 50)
               + gen_signal(  4, 2100, t, 70)
               + (data_t)(Xin[t]+NOISE_RANGE);

        fir(&yn, x[t]); // FIR Filter

        y[t] = (yn>>OUT_TRUNCATE);
    }

    calcDFT((acc_t*)x, F_SAMPLE, F_SAMPLE, Xr, Xi);
    calcDFT(y, F_SAMPLE, F_SAMPLE, Yr, Yi);

    for (int i=0; i<F_SAMPLE; i++)
        printf("%d %d %5.3f %d %5.3f\n", i,
                    x[i], log((double)(Xr[i]*Xr[i]+Xi[i]*Xi[i])),
                    y[i], log((double)(Yr[i]*Yr[i]+Yi[i]*Yi[i]))
        );

    return 0;
}
```

DUT(FIR 필터)의 성능 시험을 목적으로 백색잡음(white noise)에 몇가지 주파수를 갖는 신호를 혼합하여 생성하였다.

II-5. 실습: FIR 알고리즘의 시험(Lab. Un-timed algorithm test)

필터 알고리즘의 시험은 순수 C 프로그램 수준에서 실시한다. 실습의 예제 코드는 '디자인 킷'[6]에 포함되어 있다. 하드웨어의 시간 개념(클럭과 병렬성)이 포함되지 않은 C++ 코드로 작성된 모델을 "언-타임드(un-timed)" 라고 한다.

a. 테스트벤치 빌드(Build Testbench)

GNU C 컴파일러로 테스트 벤치를 빌드(build) 한다.

```
$ cd ~/ETRI050_DesignKit/Tutorials/2-5_Lab3_FIR8/c_untimed
```

```
$ gcc -I. -I/opt/local/include -o fir8_tb fir8.cpp fir8_tb.cpp calcDFT.cpp cnoise.cpp -lm -lgsl
```

GCC 가 수학(math library) 라이브러리와 GSL을 불러와 링크 시키도록 명령줄에 -lm 와 -lgsl 가 추가되었다.

[주] 알고리즘을 기술한 fir8.cpp을 검증하기 위한 테스트벤치 fir8_tb.cpp 를 컴파일 하려면 명령줄에 상당히 긴 명령을 입력해야 한다. 개발 과정에서 매번 이 명령을 입력하려면 상당히 피곤하다. make 유틸리티와 Makefile 스크립트를 작성해 놓도록 하자. 예제의 디렉토리에 빌드와 실행 그리고 결과 데이터의 확인까지 일련의 과정을 수행하는 Makefile을 작성해 놓았다. Makefile 작성법은 [연구노트3]참조한다. Makefile 스크립트로 테스트벤치 빌드 명령은 다음과 같다.

```
$ make build
gcc -I. -DFIR_MAC_VERSION -lm -lgsl -o fir8_tb \
      fir8.cpp fir8_tb.cpp calcDFT.cpp cnoise.cpp -lm -lgsl
```

b. 테스트벤치 실행(Run Testbench)

GNU C++로 컴파일 하여 얻은 테스트벤치를 실행한다.

```
$ ./fir8_tb
0 167 36.857 668 36.263
1 80 33.651 2324 22.740
2 132 32.146 5663 22.891
...…
4797 110 31.941 15522 22.948
4798 94 32.146 15909 22.891
4799 39 33.651 15813 22.740
```

FIR 필터 알고리즘 시험용 테스트 벤치는 한 회에 4800개의 시계열(time domain) 입력 자료를 생성하고 필터를 거친 출력을 얻는다. 수없이 나열된 숫자들을 보고 그 결과를 판단 할 수는 없다. 대량 데이타를 저장해 두었다가 가시화(visualization) 도구를 통하여 분석(analysis) 하기로 한다. 명령줄에서 > 기호(리다이렉션, re-direction)를 사용하여 표준 출력을 임의 파일로 바꿔 줄 수 있다.

```
$ ./fir8_tb > fir8_tb_out.txt
```

테스트벤치의 출력을 일반 파일(plain text file) fir8_tb_out.txt 로 저장한다. 파이썬(Python)은 다양한 데이터 가시화 라이브러리를 갖춘 알고리즘 분석의 좋은 도구다.

[주] Makefile 스크립트로 테스트벤치 실행 명령은 다음과 같다.

```
$ make run
./fir8_tb | tee fir8_tb_out.txt
0 134 36.937 536 36.268
1 86 33.120 1952 23.962
2 158 29.823 5014
  .......
```

c. 파이썬(Python) 데이터 시각화(Data Visualization)

FIR 필터의 성능 분석에 파이썬을 활용한다. FIR 필터 알고리즘을 시험한 결과를 시각화하기 위한 파이썬 코드는 예제에서 plotDFT.py 다.

```
#!/usr/bin/env python3

import os, sys
import matplotlib.pyplot as plt
import numpy as np
```

첫 줄에 이 스크립트가 파이썬3(python3)용 이라는 것을 표시하고 있다. 파이썬 스크립트는 별도의 지정이 없는 한 가장 먼저 만나는 줄에서 시작한다. 사용할 파이썬 라이브러리들을 불러온다(import libraries). matplotlib 는 다차원 배열의 데이타를 시각화 해주는 라이브러리다. numpy는 다차원 배열(행렬) 데이타를 다룰 때 유용하다. 라이브러리 이름 뒤의 as 는 긴 라이브러리 이름을 쓰기 번거롭지 않도록 줄여 준다.

```
if len(sys.argv)!=2:
    print('plotDFT.py <option>')
    print('     Plot graph from text file. <option> is one of followings,')
    print('        input, inputDFT, filter, filterDFT')
    sys.exit(1);
```

명령줄에서 실행 인수(Command-line argument)의 갯수를 확인한다. 만족하지 않을 경우 도움말을 출력하고 운영체제로 나간다(스크립트 실행종료).

```
x  = []
y1 = []     # Input Time seq
y2 = []     # Input spectrum
y3 = []     # Output Time seq
y4 = []     # Output spectrum
```

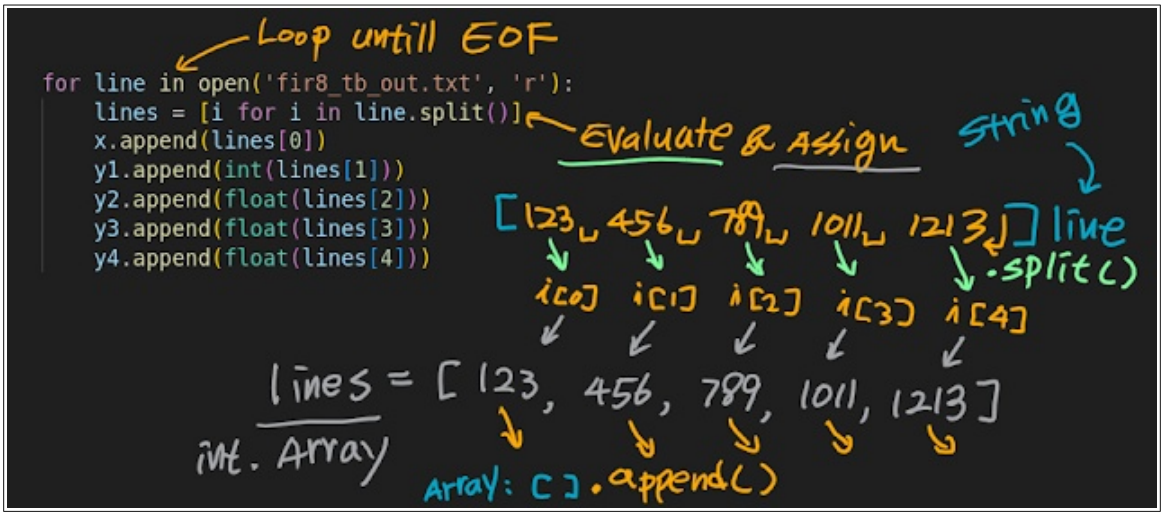
numpy 라이브러리를 사용할 수 있도록 변수들을 배열로 선언했다. 파이썬의 객체는 할당되는 값에 의해 자료형이 정해진다. 이는 객체의 자료형에 민감한 C/C++ 프로그래머에게 낯설지만 프로그래밍 입문자에게 매우 편리한 면이라고 하겠다. C/C++ 프로그래머가 보기에 파이썬은 지나치게 추상적인데, '척하면 알아서...' 해석해 주는 느낌마저 든다. 예를 들어 아래와 같은 반복문이 있다.

```
for x in range(1, 10):
    print(x)
```

변수 x 는 자료형 선언이 없어도 range(1, 10) 가 숫자 1에서 10까지 반복하는 함수이므로 알아서 정수형이다. 게다가 range()가 함수라고 하기에도 별나다. 1에서 시작하여 10이 되기 전까지 숫자들을 배열로 만들어 준다. 위의 반복문은 아래처럼 된다. 함수 range()의 리턴 값이 '알아서' 정수형 배열이다.

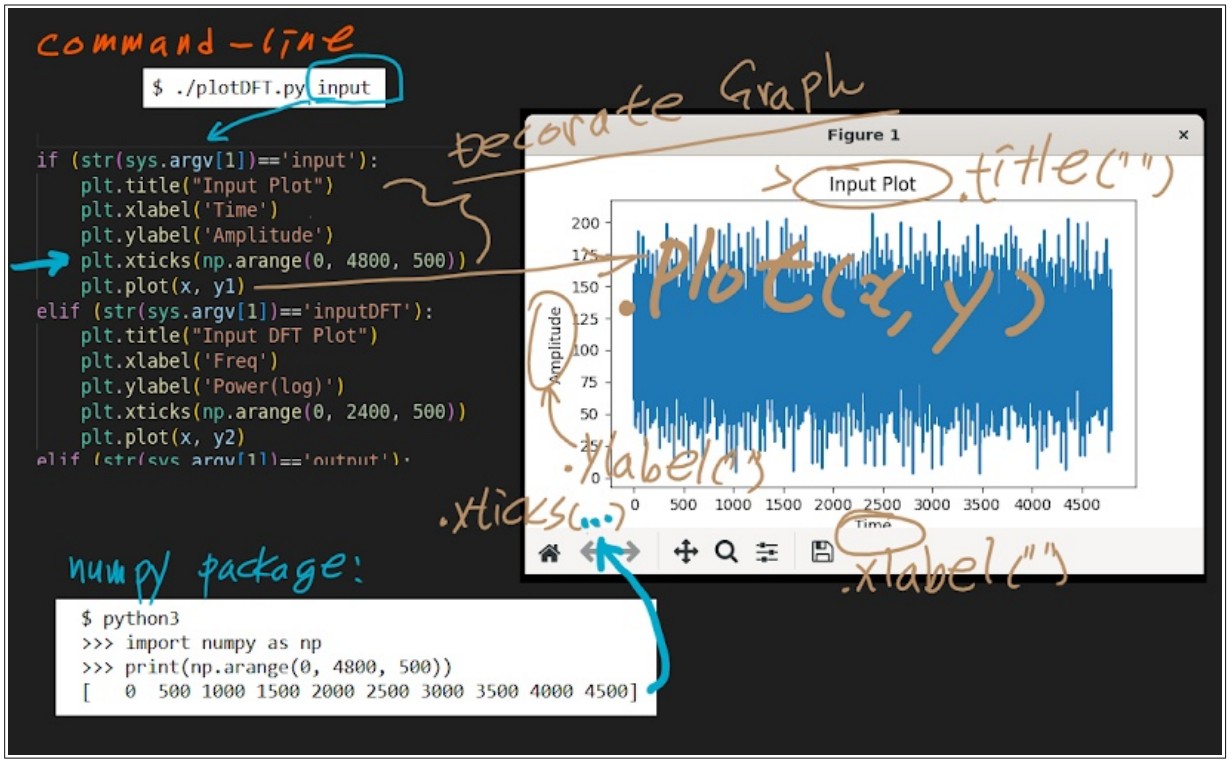
```
for x in [1,2,3,4,5,6,7,8,9]:
    print(x)
```

파이썬의 높은 추상성은 프로그래밍(코딩)을 시작하는 장벽을 낮추지만 외워야 할 것들이 늘어난다. 아래의 for 반복문은 FIR의 테스트벤치 출력을 저장한 파일 fir8_tb_out.txt 에서 1줄을 읽어 각 변수에 저장한다. 파일을 열어 문자열을 읽고 open(), 이를 나눠서 .split(), 각 변수에 넣기 .append() 까지 일련의 과정을 아주 간단히 수행한다.



파이썬의 극단적 추상성이자 인간 친화적인 면모를 볼 수 있다. for 반복문의 구간이 in 에서 파일을 취급하고 있으므로 파일의 끝(End of File)까지다. 한 줄에 여러 숫자를 공백 문자로 분리하자는 법칙(rule)은 없지만 보통 그렇게 하면 편리하니 그런 것으로 하고 문자열 자료형에 .split()를 만들어 뒀다.

다음은 테스트벤치의 실행으로 얻은 DUT의 입출력 자료들을 읽어 시각화 하는 파이썬 코드의 일부다. matplot은 다양한 데아타 가시화 기능을 갖춘 패키지다. 대량의 데이타를 수월하게 그래프로 그릴 수 있다. 테스트벤치를 실행시켜 얻은 입출력 데이타를 matplot 패키지의 .plot(x, y)로 단번에 그래프로 표시해 준다. 그래프를 꾸며주기 위해 몇가지 방법이 더 사용되었을 뿐이다.



그 외 명령줄에서 전달되는 인수를 받아오기 위해 sys 패키지의 argv[]를 참조하고 x-축에 눈금을 그릴 요량으로 벡터 데이터 생성에 numpy의 .arange(...)를 활용했다. 예제에 사용된 방법들은 패키지의 극히 일부에 지나지 않는다.

[주] Makefile 스크립트에 작성된 테스트벤치 실행 결과의 시각화 명령은 다음과 같다.

테스트 신호 시간축에서 보기,

```
$ make plot_x
```

테스트 입력 주파수 축 분석,

```
$ make plot_fx
```

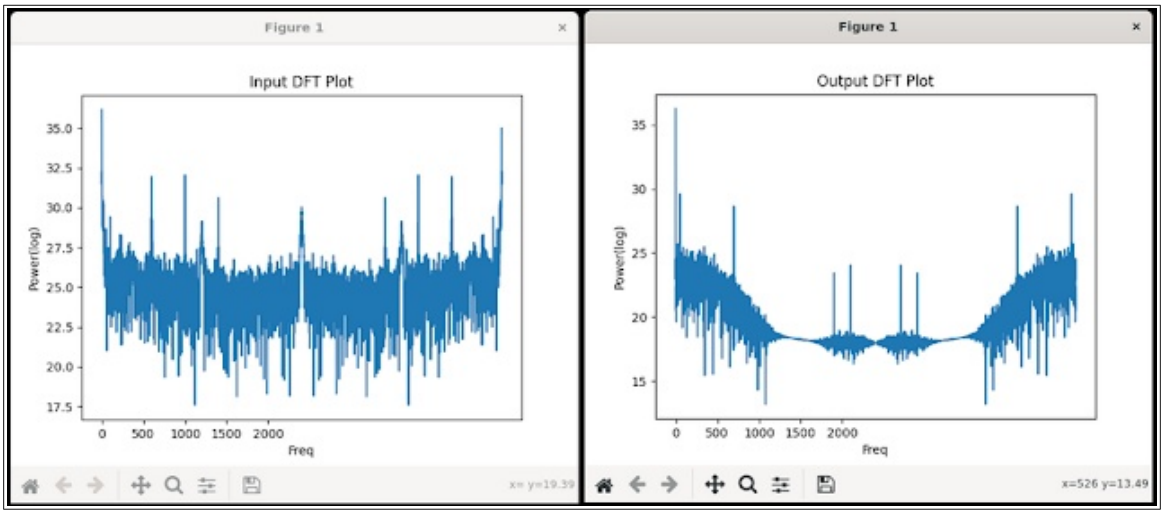
FIR 필터 출력의 시간 축 보기,주파수 분석,

```
$ make plot_fy
```

FIR 필터 출력의 주파수 분석,

```
$ make plot_fy
```

입력과 출력의 주파수 분석을 비교해 보면 필터의 효과를 한눈에 확인할 수 있다.



d. 파이썬 참고목록

프로그래밍 언어를 이해하는 파이썬 학습자라면 아래 목록 중 [1] 또는 [2] 하나로 충분하다. [3]은 과학기술 분야 응용에 많은 예제 코드들을 담고 있다. 전통적 C 언어로 작성했더라면 수십 줄의 코딩이 필요했을 일을 몇 줄의 파이썬 코딩으로 가능하다. 이러한 추상성은 오늘날 파이썬이 최고 인기를 구가하는 이유다. 라이브러리 공급자와 사용자 사이에 통념과 상식에 기반한 암묵적 동의를 바탕으로 최고의 추상성을 추구한다. 누구나 코딩의 부담에서 벗어나 알고리즘에 집중할 수 있다. 파이썬의 사용자가 증가하면서 이들이 만들어 공개적으로 제공하는 라이브러리들 역시 다양하다. 공식적으로 배포되는 표준 라이브러리(Python Standard Library)는 물론 다수의 사용자에 의해 개발되어 배포되는 패키지들도 이루 헤아릴 수 없을 정도다. 파일 입출력, 복합 자료형 등 기본 내장 라이브러리 부터 그래픽,

과학기술, 수치해석, 통계, 운용체제 등 응용 프로그램 제작에 유용하게 사용될 수 있는 라이브러리들을 망라한다. 자체적으로 라이브러리의 설치운용 기능을 갖추고 있어 파이썬은 이제 개발 플랫폼이 됐다.

[1] A Byte of Python(한글 번역본), https://byteofpython-korean.sourceforge.net/byte_of_python.pdf
[2] A Byte of Python, https://github.com/swaroopch/byte-of-python/releases/latest
[3] Python Programming And Numerical Methods: A Guide For Engineers And Scientists, https://pythonnumericalmethods.berkeley.edu/
[4] Python Standard Libraries, https://docs.python.org/3/library/index.html
[5] The Python Package Index (PyPI), https://pypi.org/
[6] Matplotlib: Visualization with Python, https://matplotlib.org/
[7] NumPy: The fundamental package for scientific computing with Python, https://numpy.org/
[8] Sys: System-specific parameters and functions, https://docs.python.org/3/library/sys.html
[9] But what is a Neural Network?, https://www.3blue1brown.com/lessons/neural-networks , https://github.com/3b1b/videos/blob/master/_2017/nn/part1.py

III. 하드웨어 구조 탐색

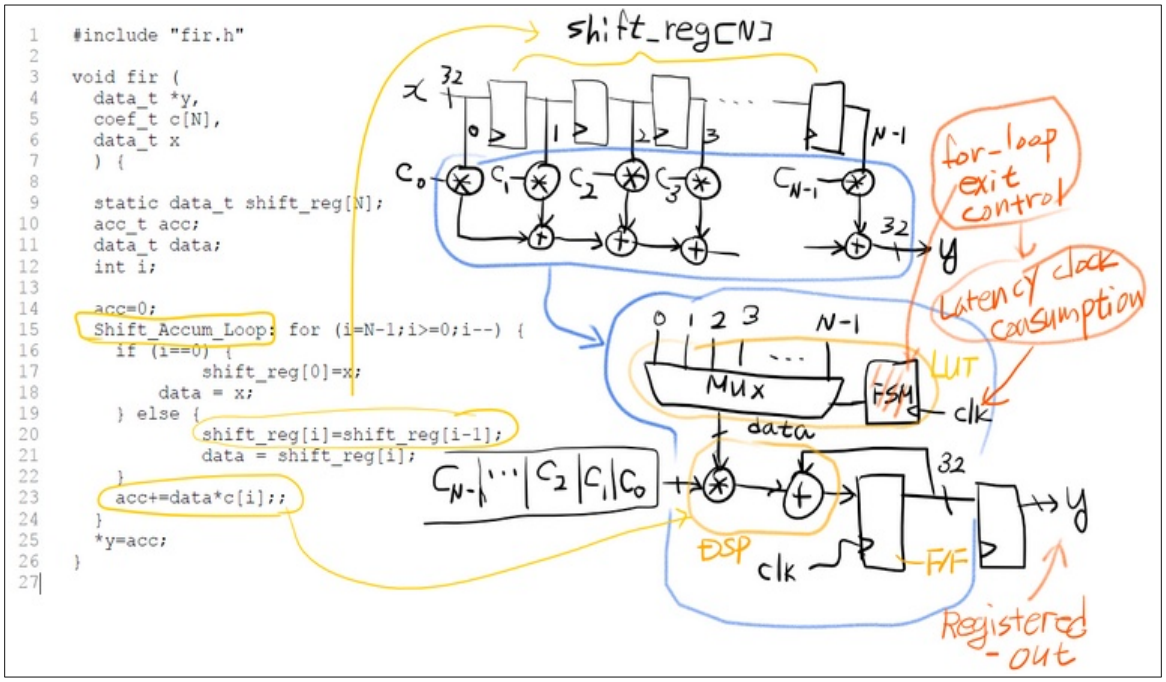
하드웨어로 구현할 목적이라면 알고리즘은 레지스터 전송 수준(RTL, Register-Transfer Level)으로 기술되어야 한다. 합성기(synthesizer)라는 자동화 도구를 사용할 수 있기 때문이다. RTL에서는 클럭 단위로 하드웨어의 동작을 묘사한다. 아울러 각 객체들(신호선, 와이어)은 비트 단위로 상세히(clock & bit-detailed) 기술된다. 알고리즘을 기술한 언-타임드(un-timed) C++ 코드를 하드웨어 언어로 바로 전환 하기에 추상화 수준의 간격이 너무 크다. 게다가 하드웨어의 동시실행성(concurrency)은 알고리즘 C++ 코드에 포함되지 않은 개념이다. 최근 알고리즘 C++ 를 RTL로 합성해 주는 HLS(High Level Synthesis) 자동화 도구 가 부상하고 있지만* 여전히 수동 변환이 대다수를 차지한다.

* Towards Automatic High-Level Code Deployment on Reconfigurable Platforms: A Survey of High-Level Synthesis Tools and Toolchains, IEEE Access, Vol.8,2020, https://ieeexplore.ieee.org/abstract/document/9195872

III-1. FIR 알고리즘의 타임드 모형(Timed model)

FIR 필터의 언-타임드 C++에서 병렬성을 찾아내고 파이프라인 구조를 도출해 본다. 언-타임드 C++로 작성된 알고리즘을 하드웨어로 구현하는 첫 단계는 변수들 사이의 의존 관계(dependency)를 따져 병렬성(parallelism)을 탐지해 내는 일이다. 이를 근거로 동시처리(concurrency)가 가능한 요소(processing element)로 분할하고 클럭 기반의 스케쥴링을 수립한다.

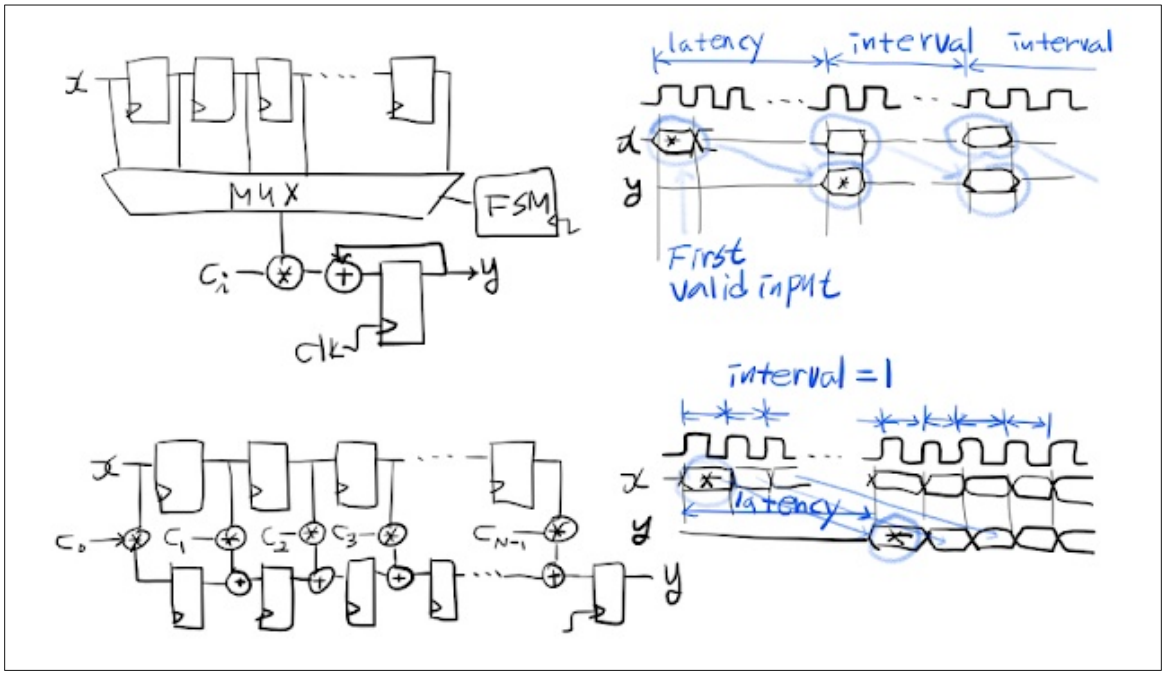
FIR 필터 알고리즘의 골자인 for 반복문을 다루는 방법에 두 가지 구조를 고려해 볼 수 있다. 첫째 구조는 누산기를 두고 반복문의 제어 절차를 FSM(유한 상태 머신 제어기)으로 구현하는 방법이다. 둘째는 반복문 for 에서 반복 변수 i를 펼쳐 놓고 자료의 의존 관계를 살펴보는 것이다. 곱셈과 누적 연산 사이에 순방향에 만 의존하므로 파이프라인 병렬처리 구조(pipelined-parallelism)가 가능하다. 두 구조는 각각 장단점이 있으므로 조건과 용도에 맞게 선택한다.

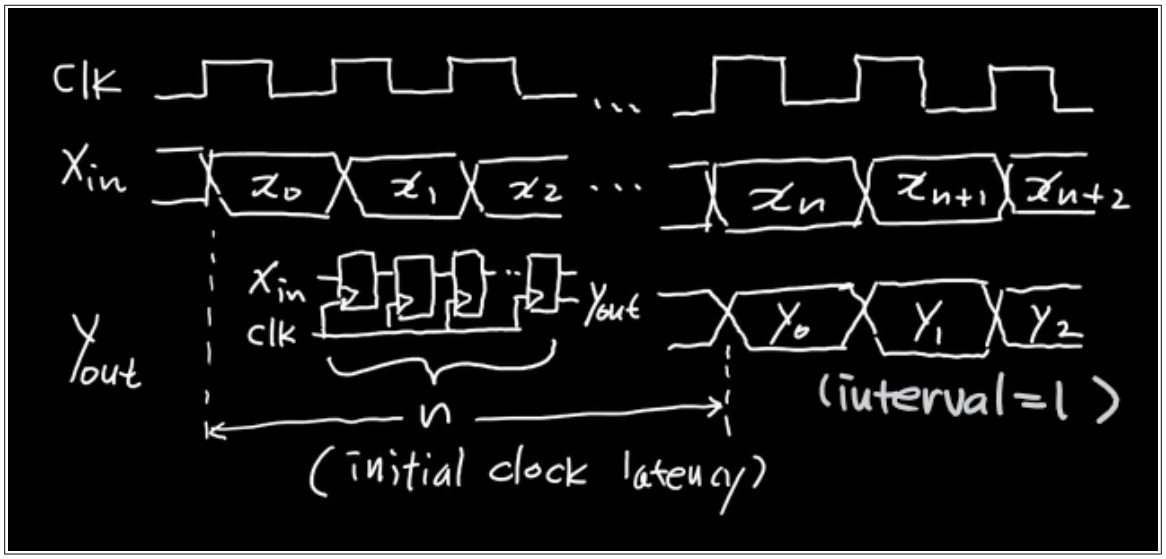


[주] RTL에서 논리회로 네트리스트로 변환하는 합성기가 발전할 수 있던 요인은 하드웨어 언어의 RTL과 디지털 회로의 추상화 수준의 간격이 크지 않았기 때문이다. 병렬성과 클럭의 개념은 순차적 알고리즘에서 RTL로 합성을 어렵게 한다. 프로그래밍 언어에서 따지지 않던 "클럭 싸이클(clock cycle)" 과 "레이턴시(latency)"는 하드웨어 설계의 중요 고려 사항이다. 입출력 신호의 인터페이스 타이밍 요건은 알고리즘을 하드웨어로 구현할 때 구조를 결정하는 중요한 요소다.

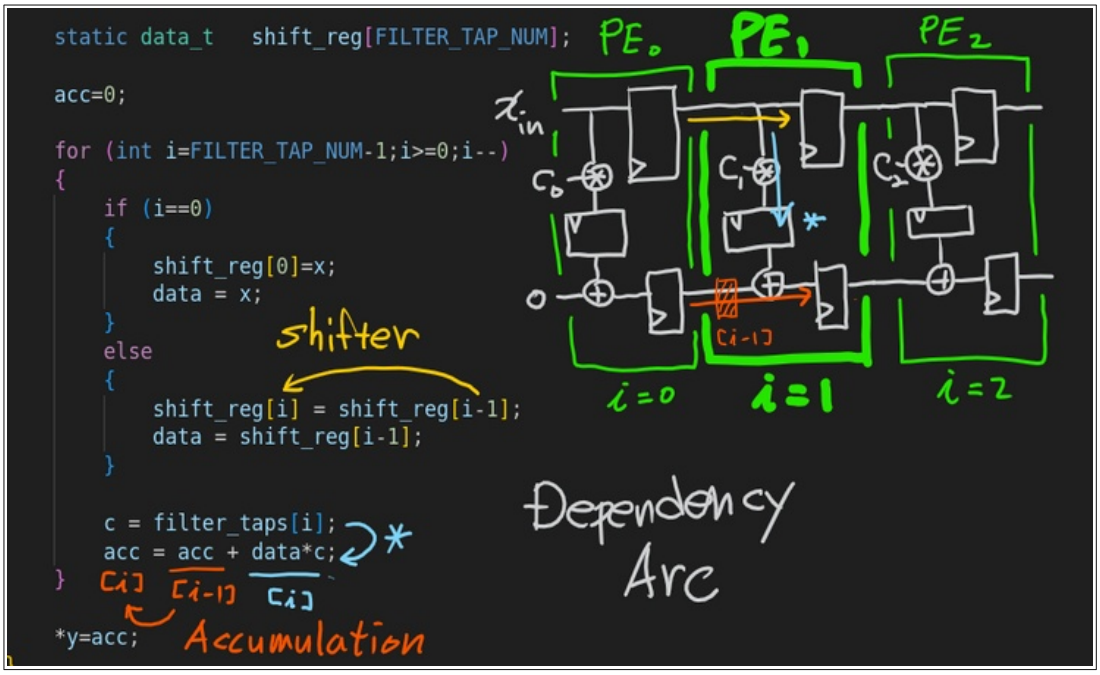
FIR 필터의 입출력 인터페이스 요구사항을 살펴보자. 매 클럭마다 입력이 들어오고 실시간으로 출력을 내야 한다(interval=1). 프레임 단위로 입력을 저장해 두었다가 출력하는 방식이라면 계산기의 속도를 높여 실시간 처리가 가능하다. 이에 비하여 파이프라인 병렬처리 구조의 경우 입력과 출력의 데이터 생성율(In/Out data rate)을 계산기의 클럭율(clock rate)과 일치시킬 수 있다. 계산에 사용되는 클럭율을 낮춰 얻을 수 있는 큰 장점은 전력 효율이다. 굳이 전용 하드웨어가 필요한 이유다. 단점은 하드웨어의 규모가 커진다. 오늘날 반도체 공정이 발달하여 집적도가 높고 낮은 전력으로(나노와트 이하) 트랜지스터를 구동할 수 있으며 스위칭 할 때 소모 전력이 매우 낮다. 현대적 설계 방법론은 하드웨어의 규모 증가보다 적은 클럭수를 추구한다.

아래와 같은 두가지 하드웨어 구조를 하드웨어 사용량과 입출력 인터페이스, 입력에 대한 유효 출력이 나오기까지 처리에 필요한 클럭 수(IO through rate)의 관점에서 비교해보자. 초기 레이턴시(initial latency)는 첫 유효 입력이 처리되어 출력으로 나오기까지 소요되는 클럭 갯수다. 이어진 연속 입력에 대하여 출력이 나오기까지 소요되는 클럭 갯수는 인터벌(interval)이다. 첫째 구조는 쉬프트 레지스터에 저장해 놓은 입력을 반복적으로 곱셈과 누적 연산을 수행한다. 최소의 연산기를 사용하고 있다. 범용 계산기에서 프로그래밍으로 처리하는 방식과 유사하다. FSM이 CPU의 반복 명령을 대신한다. 둘째 구조는 파이프라인 병렬처리다. 초기 지연 클럭 수는 쉬프트 레지스터의 갯수 만큼이다. 인터벌은 1이다.





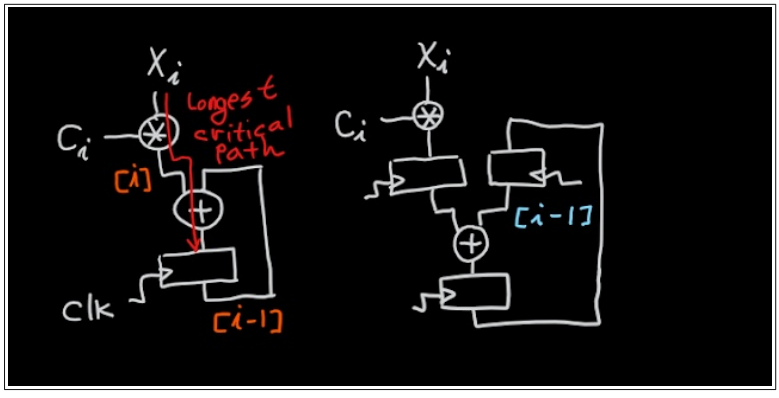
FIR 필터의 계산(알고리즘)을 C++의 단일 함수로 기술한 fir8.cpp을 병렬성(파이프라인 구조) 도출의 관점에서 살펴보기로 한다. for 반복문 내에서 변수들의 반복색인(loop index)을 따져 보면 변수 shift_reg 와 acc의 현재색인 [i] 값이 각각 이전색인 [i-1]의 값과 의존관계(dependency arc)에 있다. 계산이 끝날 때까지 이 의존관계를 유지한다. 이를 근거로 쉬프트 레지스터(shift register)와 누산기(multiplier-accumulator)를 파이프라인 병렬구조의 처리 단위(PE, Processing Element)로 삼을 수 있다. Xin은 지역 변수 shift_reg[] 로 정적 선언되어 있고 할당의 좌우 색인 관계가 쉬프트 레지스터로 명확하다. 누산(accumulation)의 경우 반복문 내에서 이전값과 덧샘을 수행하고 있는 점에 유의한다.



다수의 비트폭을 갖는 디지털 연산기를 하드웨어로 구현할 때 회로의 복잡도를 주의 깊게 살펴봐야 한다. 대수 연산은 조합회로 합성이 일반적이다. 회로의 복잡도가 매우 높기 때문에 (전기 신호의 전달에) 긴경로(longest path)의 원인이 된다. 어떤 회로가 긴경로를 가지면 시스템 동작 속도(레지스터 사이의 전송에 사용되는 클럭의 주기)에 치명적인 영향(critical path)을 준다. 위의 C++ 코드에서 for 반복문 내에 곱셈에 이어 덧셈이 연속적으로 일어나고 있다. 곱셈기는 매우 복잡한 조합회로다.

```
acc = acc + data*c;
```

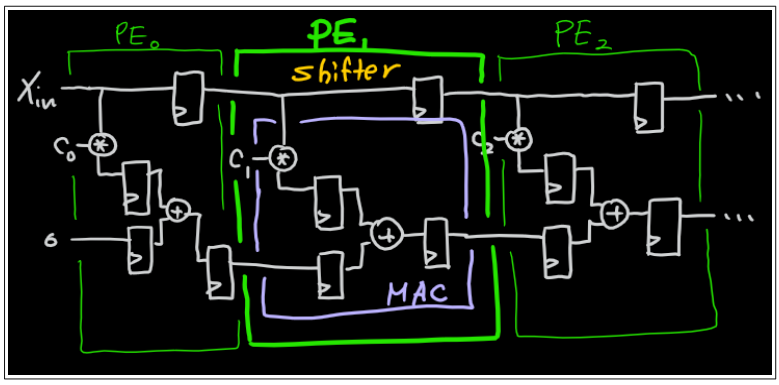
조합 회로로 구현되는 곱셈과 덧셈이 연속적으로 이어지면 전기 신호가 통과할 트랜지스터의 갯수가 늘어난다. 결국 전류가 지나는 경로상에 큰 저항이 놓인 것과 같다. 복잡한 조합회로를 논리식으로 보면 경우의 수가 증가한다. 이는 불필요한 트랜지스터의 스위칭을 유발하여 전력 소모를 증가 시키고 고장 탐지를 어렵게 한다.



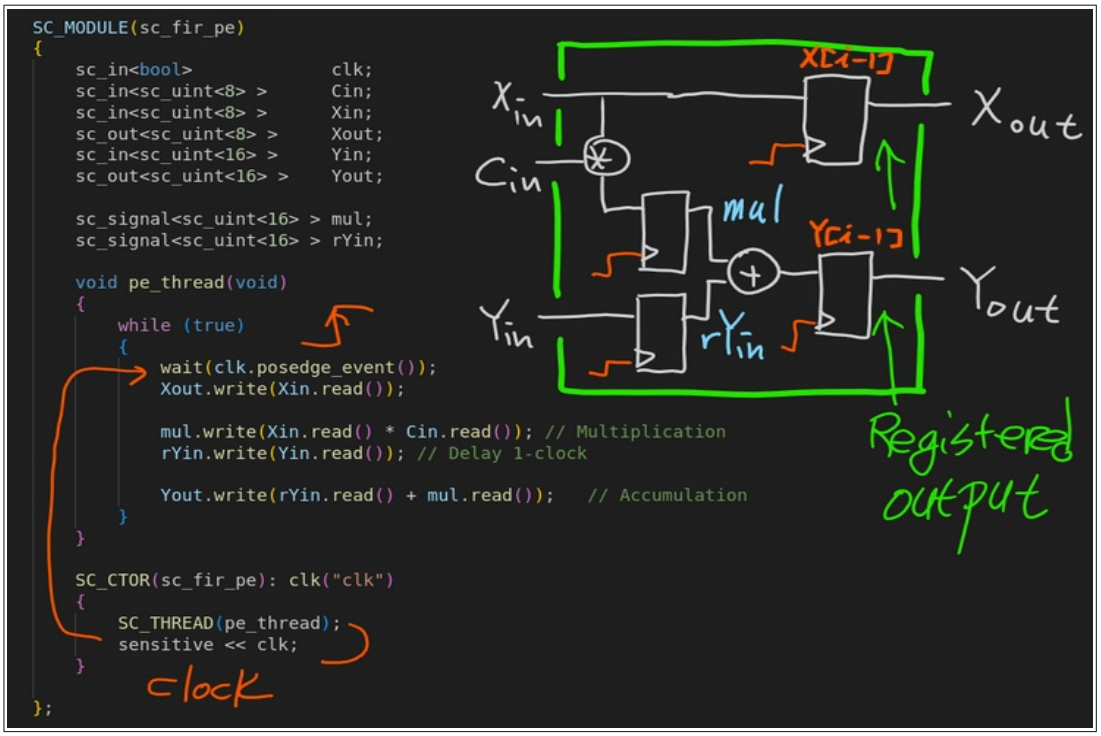
MAC(multiplication accumulator)의 구조에서 곱셈과 덧셈을 이어놓을 경우 회로 경로가 길어지는 것을 막기 위해 곱셈과 덧셈 사이에 레지스터를 추가한다. 위험 경로(critical path)를 끊어주기 위해 레지스터를 추가하면 회로 동작에 클럭이 필요하다. 위험 경로가 길어지는 것보다 클럭을 한 개 더 쓰는 편이 유리하다.

III-2. 처리 요소(Processing Element)

C++ 알고리즘을 구현할 하드웨어의 구조가 결정되었다고 곧바로 RTL 설계에 돌입할 수도 있겠지만 추상화 수준 차이를 극복하기는 쉽지 않다. 먼저 구조에 맞도록 알고리즘을 병렬 처리 단위로 분할한 후 병렬처리 스케쥴(parallel processing schedule)에 부합하는지 확인되어야 한다. 이번 FIR 필터의 입출력 스케쥴은 클럭 당 1 샘플이다. 언타임드 C++ 의 for 반복문을 파이프라인 병렬 처리 구조로 풀어 놓으면 아래와 같다. MAC 연산기가 누산형에서 파이프라인 구조로 궤환 경로가 풀려있다(Loop un-lock).



배열을 반복적인 처리요소 PE로 분할했다. 파이프라인 병렬처리 MAC 연산기는 계수와 PE 갯수의 변경에 따라 다양한 응용이 가능하다. PE의 반복 재사용을 고려하여 모두 레지스터 출력이 되도록 아래와 같은 분할이 좋다. 파이프라인 병렬처리 처리요소 PE 를 SystemC 로 기술하면 아래와 같다. 클럭 사건에 반응하여 처리요소가 행동하도록 기술되었다.



SystemC의 모듈 크래스 SC_MODULE() 와 구성자 매크로 SC_CTOR(), 그리고 템플릿 크래스 객체 sc_in<>, sc_out<>, sc_signal<> 들이 조금 생소하지만[연구노트4] C++와 하드웨어 모델링의 눈으로 보면 어렵지 않게 이해될 것이다. 타임드 SystemC로 기술한 FIR 필터의 PE를 살펴보자. 예제의 파일명은 sc_fir_pe.h 다.

PE의 모듈 크래스의 이름은 sc_fir_pe 다. 모듈 크래스는 동일한 이름의 구성자를 가진다.

```
SC_MODULE(sc_fir_pe)
{
    ......
    SC_CTOR(sc_fir_pe): clk("clk")   // 구성자(Constructor)
    {
        ...
    }
};
```

입출력 포트로 1비트 clk, 부호없는 8비트 정수형 Cin과 Xin, Xout, 16비트 정수형 Yin, Yout 가 선언 되었다. 입출력 방향과 비트 폭을 템플릿 크래스로 명시되었다. 하드웨어를 기술한 것이다.

```
    sc_in<bool>             clk;
    sc_in<sc_uint<8> >      Cin;
    sc_in<sc_uint<8> >      Xin;
    sc_out<sc_uint<8> >     Xout;
    sc_in<sc_uint<16> >     Yin;
    sc_out<sc_uint<16> >    Yout;
```

모듈 내 지역 신호들도 템플릿을 사용하여 선언되었다.

```
    sc_signal<sc_uint<16> > mul;
    sc_signal<sc_uint<16> > rYin;
```

구성자는 모듈의 내부를 구성한다. 클럭 입력 clk의 사건에 감응되어 호출될 콜백 함수(call-back function) pe_thread 를 지정했다. 콜백 함수의 형식은 SC_THREAD() 다.

```
    SC_CTOR(sc_fir_pe): clk("clk")   // 구성자(Constructor)
    {
        SC_THREAD(pe_thread);
        sensitive << clk;
    }
```

모듈 크래스의 소속함수 pe_thread()에 PE의 행동이 묘사되었다. SC_THREAD()로 지정된 콜백 함수는 시뮬레이션 커널에 의해 호출될 것이다. SystemC의 병렬 시뮬레이션 커널은 협동형 다중처리(cooperative multi-processing)다. 감응으로 지정된 clk에 사건이 발생하면 콜백 함수가 호출된다. 콜백 함수의 무한 반복문 내에 사건에 대기하는 wait()를 두고 있다. 이는 협동형 다중처리에서 실행권을 양보하는 yield() 함수와 같다. 단, 재개할 조건을 갖는다. 아래의 경우 시뮬레이션 커널은 clk에 상응 엣지 사건이 감지되면 콜백 함수를 재개한다. 실행이 양보 되었던 콜백 함수가 clk의 상승 엣지 사건에 재개되면 Xin과 Cin을 곱하고 rYin과 더한다. rYin은 1클럭 이전의 누적 값이다. 파이프라인에서 이전 PE로부터 전달 받은 값이므로 누적 연산을 수행한다.

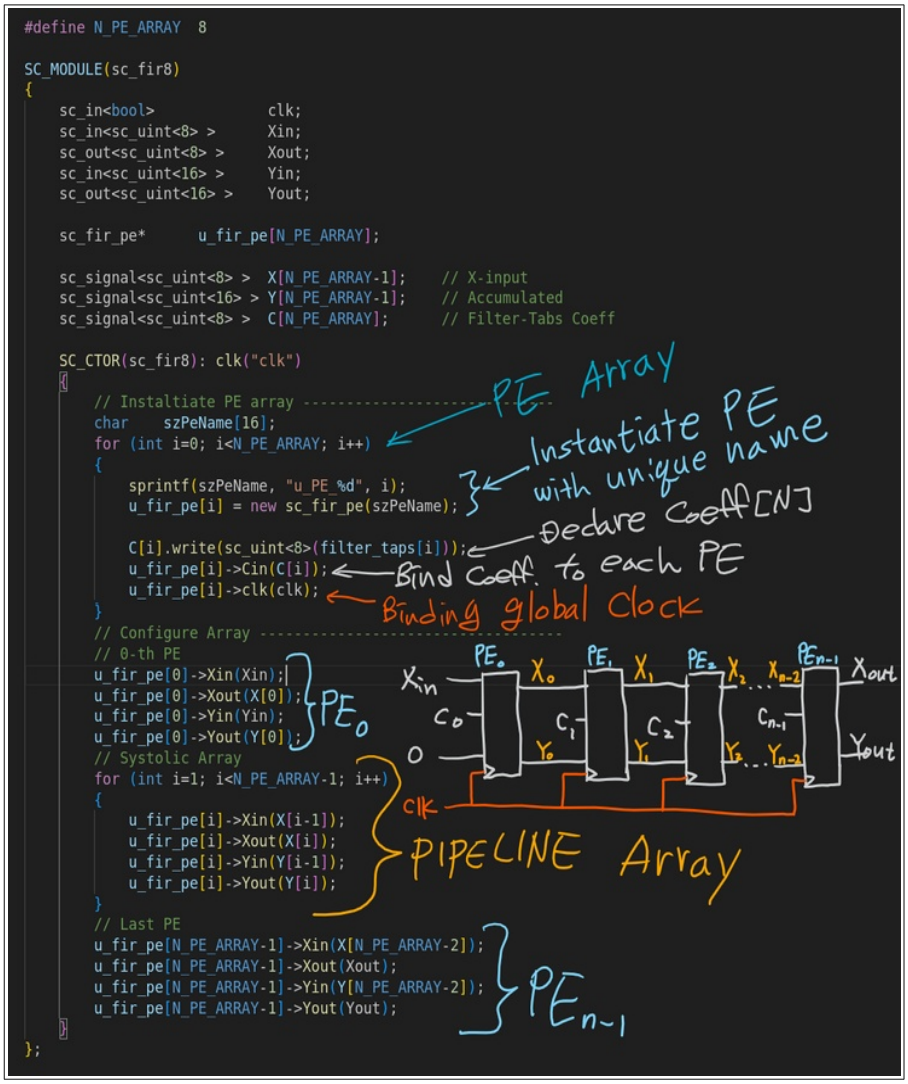
```
    void pe_thread(void)
    {
```

```
        while (true)
        {
            wait(clk.posedge_event());
            Xout.write(Xin.read());  // Xin→Xout Pipeline

            mul.write(Xin.read() * Cin.read()); // Multiplication
            rYin.write(Yin.read());             // Delay 1-clock
            Yout.write(rYin.read() + mul.read());   // Accumulation
        }
    }
```

입출력 포트의 지정, 비트 단위의 객체선언 그리고 클럭의 엣지 사건등을 묘사한 RTL 수준의 기술이다. SystemC는 C++로서 하드웨어를 묘사할 수 있음을 보여준다.

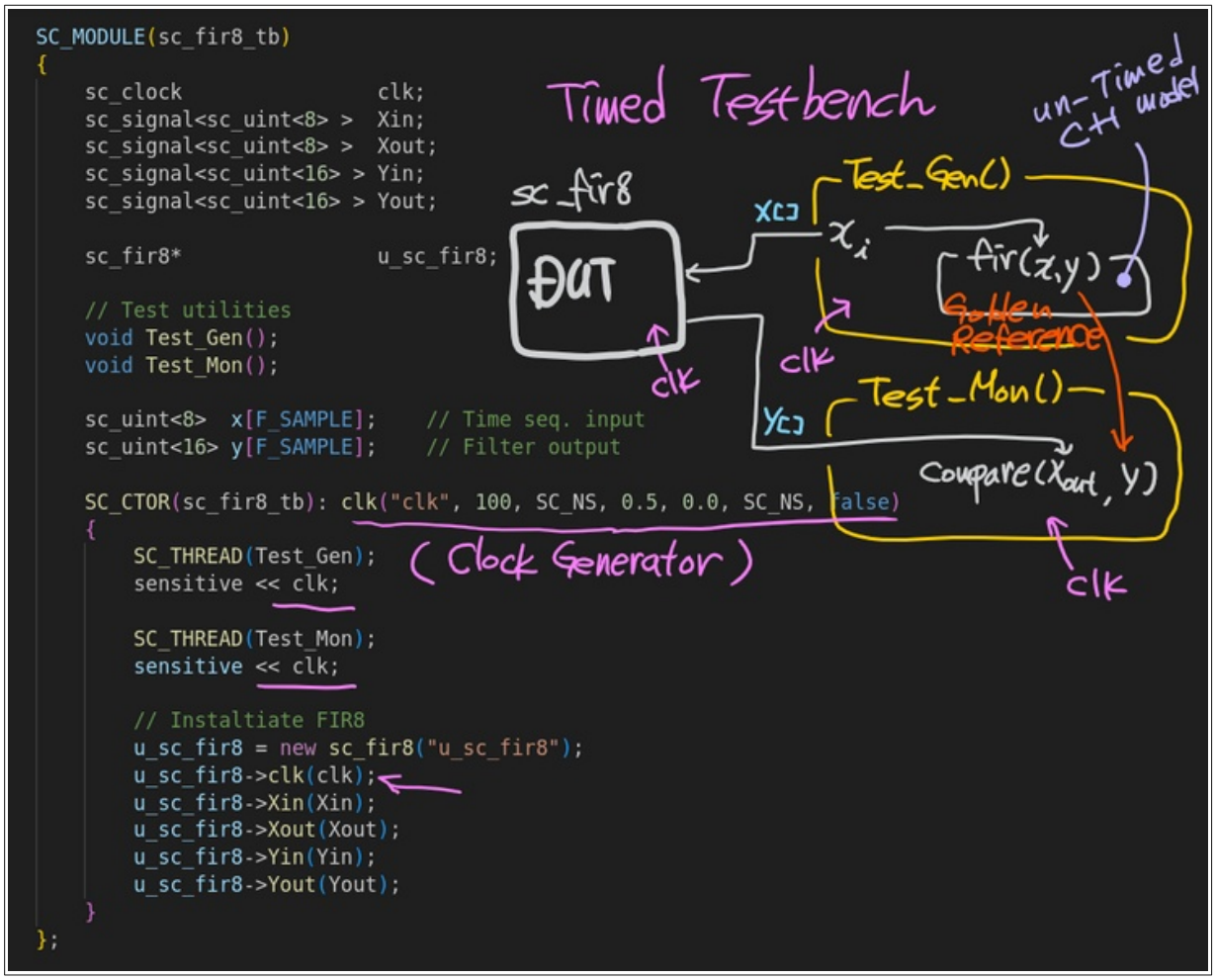
III-3. 파이프라인 구조 모델링

PE를 연속 배열한 파이프라인 처리 구조의 FIR 필터를 기술한 구조적 SystemC 모델은 아래와 같다. 모듈의 구성자에 다수의 PE들을 사례화 하고 지역 신호로 연결함으로서 파이프라인 구조의 FIR 필터를 구성하였다.



III-4. 언-타임드 실행형 사양과 타임드 테스트벤치

SystemC로 기술한 파이프라인 구조 FIR 필터를 DUT로 하는 테스트벤치는 아래와 같다. 예제 파일명은 fir8_tb.cpp다. DUT가 타임드 RTL 이다. 테스트벤치 역시 클럭에 동기를 맞춰 테스트 값을 생성하고 DUT을 출력을 검사한다. 알고리즘 개발에 사용되었던 언타임드 FIR 알고리즘 모델 fir()가 검사 표준(golden reference)으로 활용된다.



테스트벤치에 시험 입력을 생성하는 Test_Gen()과 Test_Mon() 가 소속 함수로 포함되어 있다. 이들 두 함수는 모두 클럭 clk 에 동기하여 작동한다. DUT 인 sc_fir8 이 타임드 모델이기 때문이다. Test_Gen()과 Test_Mon()는 테스트벤치 sc_fir8_tb.cpp 에 기술되어있다.

```
void sc_fir8_tb::Test_Gen()
{
    double      X_in[F_SAMPLE]; // Noise
     ...
    // Generate tests & reference from C-Model
    srand(time(NULL));   // 난수 발생기 설정
    cnoise_generate_colored_noise_uniform(X_in,F_SAMPLE,0,NOISE_RANGE );
                  // 백색잡음: Alpha=0(White Noise), range=+/-NOISE_RANGE
```

```
    for (t=0; t<F_SAMPLE; t++)
    {
        x[t] =  sc_uint<8>(AMPLITUDE/16.0*
              (cos((2*M_PI/F_SAMPLE)*51.0*t + (float)(rand()%10)/ 10.0)+1))
              + ......
              + sc_uint<8>(X_in[t]+NOISE_RANGE);

        fir(&yn, x[t]); // C-Model FIR Filter
        y[t] = yn; // Golden Reference
    }
    while(true)
    {
        wait(clk.posedge_event());  // 클럭 clk의 상승 엣지에 맞춤
        Xin.write(x[t]);            // 타임드 모델의 입력 써넣기
        t = ((++t) % F_SAMPLE);
    }
}

void sc_fir8_tb::Test_Mon()
{
    int         n = 0;
    uint16_t    yout;

    FILE *fp = fopen ( "sc_fir8_tb_out.txt", "w" );

    while(true)
    {
        wait(clk.posedge_event());     // 클럭의 상승 엣지에 동기
        yout = (uint16_t)Yout.read();  // 타임드 모델의 출력
        if (yout==0)    continue;

        if (y[n]!=yout)     // 언타임드 알고리즘과 타임드 모델의 출력 비교
            printf("Error:");
        printf("[%4d] y=%d / Yout=%d\n", n, (uint16_t)y[n], yout);
        fprintf(fp, "%5d %5d\n", (uint16_t)x[n], (uint16_t)yout);

        n++;
        if (n==F_SAMPLE)
        {
            fflush(fp);
            fclose(fp);
            sc_stop();
        }
    }
}
```

테스트벤치는 언-타임드 C++ 알고리즘을 참조 모델(golden reference)로 사용한다. 테스트벤치 내에 참조 모델이 호출 가능한 함수로 들어와 있다. 이를 실행형(executable specification) 사양이라 한다.

III-5. 실습. FIR 필터의 타임드 시뮬레이션

FIR 필터의 SystemC/C++ 타임드 테스트벤치를 실행하여 확인할 사항은 아래와 같다.

- 파이프라인 병렬처리 타이밍과 클럭당 입출력(throughput rate)
- 타임드 모델 검증: 언타임드 알고리즘의 결과와 타임드 모델의 결과 비교

SystemC/C++ 모델의 빌드와 실행 그리고 시각화를 위한 파이썬 실행은 미리 준비해 둔 메이크 스크립트 Makefile를 통해 이뤄진다. 예제는 '디자인 킷'의 Tutorials 에 포함되어 있다.

예제 디렉토리로 이동,

```
$ cd ~/ETRI050_DesignKit/Tutorials/2-5_Lab3_FIR8/sc_timed
```

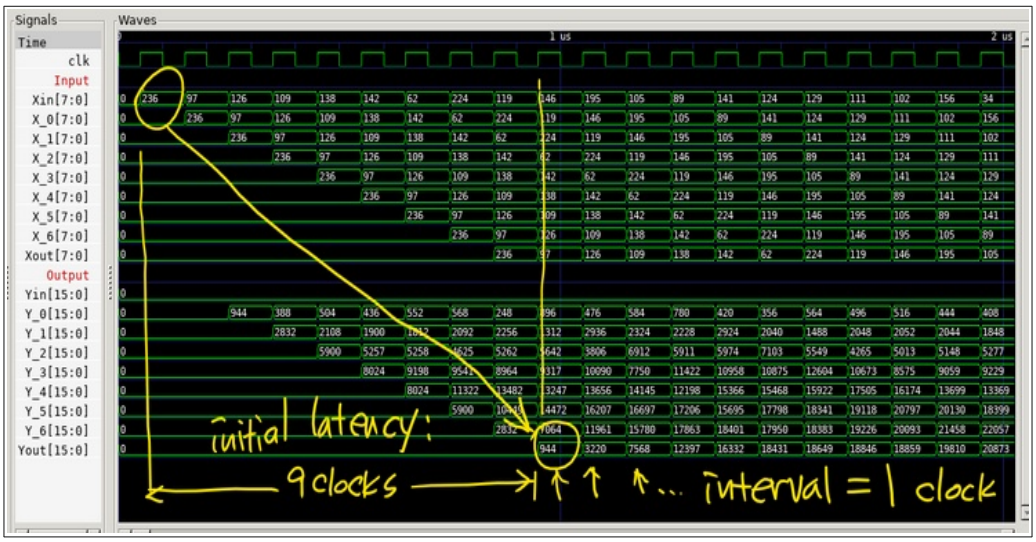
SystemC/C++ 테스트 벤치 빌드,

```
$ make build
clang++ -I. -I../c_untimed -I/opt/systemc/include -g -DVCD_TRACE_FIR8 \
 -DVCD_TRACE_FIR8_TB -DFIR_MAC_VERSION -L/opt/systemc/lib \
       -o sc_fir8_tb -lsystemc -lm -lgsl sc_main.cpp sc_fir8_tb.cpp \
         ../c_untimed/fir8.cpp ../c_untimed/cnoise.cpp
```

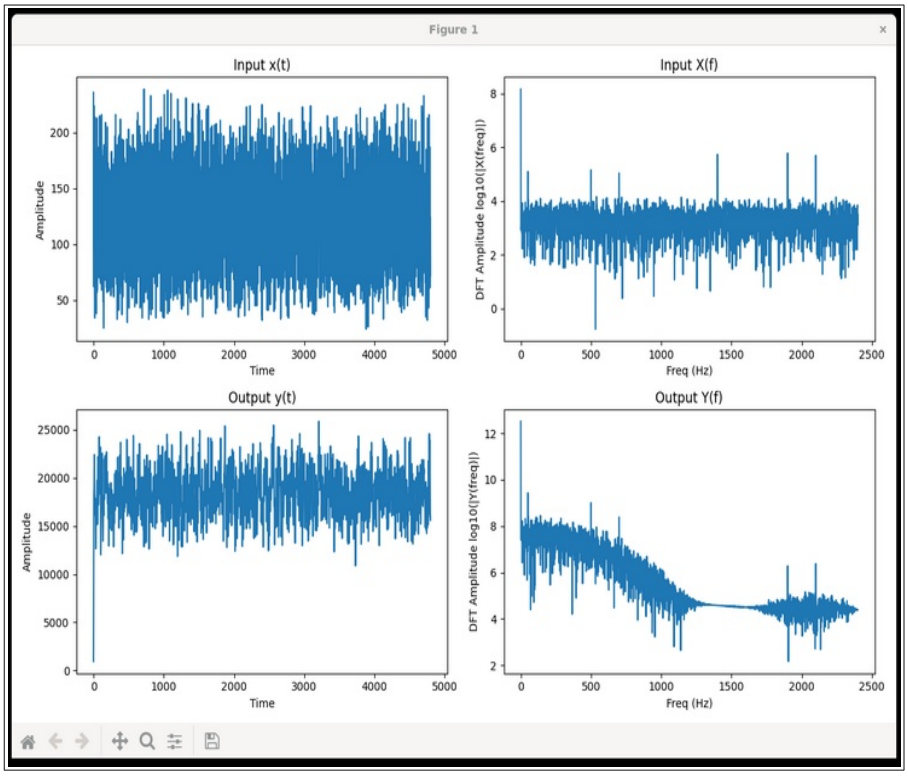
SystemC로 작성한 타임드 모델과 테스트 벤치에 필요한 알고리즘은 c_untimed 의 것을 가져왔다. 이어서 실행,

```
$ make run
./sc_fir8_tb
        SystemC 3.0.0-Accellera --- Aug  9 2024 12:41:46
        Copyright (c) 1996-2024 by all Contributors,
        ALL RIGHTS RESERVED
    [    0] y=944 / Yout=944
    [    1] y=3220 / Yout=3220
    ......…
    [4798] y=15985 / Yout=15985
    [4799] y=15617 / Yout=15617
```

언타임드 표준 모델과 타임드 모델의 출력을 비교한 결과 오류 없이 수행되었다. 타임드 모델이 클럭에 동기되어 동작하는 모습은 VCD 파일로 기록 되었다.



초기 지연은 9개 클럭 싸이클이 소요되었고 이어서 1클럭 마다 출력을 내고 있다. 파이프라인 병렬처리의 처리 요구 타이밍 스케줄을 만족한다. FIR 필터의 입출력 데이타에 대한 시각화 프로그램은 파이썬 프로그램으로 작성된 sc_plotDFT.py 다.



FIR 필터 출력의 주파수 분석 DFT(Discrete Fourier Transform)은 파이썬 numpy 라이브러리를 활용하여 작성되었다[참조:Discrete Fourier Transform(DFT)].

```
# DFT
#
import numpy as np
def DFT(x):
    """
    Function to calculate the
    discrete Fourier Transform
    of a 1D real-valued signal x
    """
    N = len(x)
    n = np.arange(N)
    k = n.reshape((N, 1))
    e = np.exp(-2j * np.pi * k * n / N)
    X = np.dot(e, x)
    return X
```

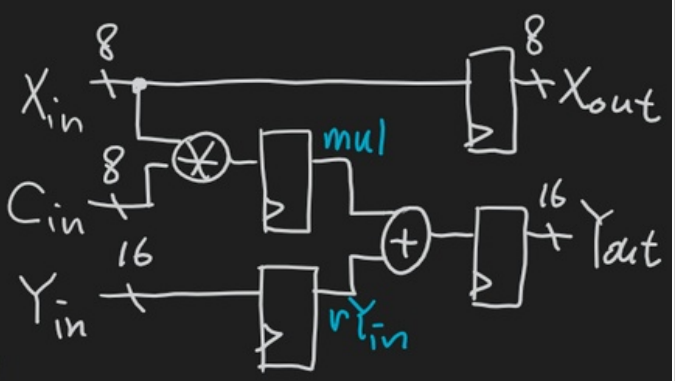
IV. 베릴로그 RTL

SystemC의 목적이 RTL 기술은 아니다. 하지만 FIR 필터의 구조가 단순하여 SystemC 타임드 모델링 단계에서 이미 RTL 수준이 되었다. RTL의 SystemC(또는 C++)에서 베릴로그로 변환(합성) 해주는 도구가 있으나 이번 예제는 수동 변환을 해보기로 한다.

IV-1. 처리요소의 RTL 베릴로그 (Processing Element RTL Verilog)

FIR 필터의 처리요소의 베릴로그 모델은 다음과 같다.

```
module fir_pe (clk, Cin, Xin, Xout, Yin, Yout);
input           clk;
input   [ 7:0]  Cin;
input   [ 7:0]  Xin;
output  [ 7:0]  Xout;
input   [15:0]  Yin;
output  [15:0]  Yout;

    reg     [ 7:0]  Xout;
    reg     [15:0]  Yout;
    reg     [15:0]  rYin;
    reg     [15:0]  mul;

    always @(posedge clk)
    begin : Multiply
        mul <= Xin * Cin; // Multiply
    end

    always @(posedge clk)
    begin : Hold_Yin
        rYin <= Yin; // Hold 1-Clock
    end

    always @(posedge clk)
    begin : Registered_Out
        Xout  <= Xin;
        Yout  <= mul + rYin;    // Accumulation
    end

endmodule
```



타임드 모델에서 도출한 PE와 동일한 동작을 한다. SystemC 모듈 sc_fir_pe.h 과 기술 방법 상 차이를 비교해 보기 바란다.

IV-2. 파이프라인 배열 구조의 RTL 베릴로그 (Pipeline Array RTL Verilog)

처리요소 fir_pe를 직렬로 연결한 파이프라인 구조 베릴로그 기술은 다음과 같다. FIR 필터 계수는 상수 배열(coustants array)로 localparam 으로 정의 됐다. 처리요소 fir_pe는 하위 모듈(sub-module)로 for 반복문으로 사례화(instanciate)했다. SystemC의 구조적 기술과 비교해보면 내용(추상화 수준)면에서 일치한다. 베릴로그가 하드웨어 기술 전용 언어인 만큼표현은 매우 간결하다. 타임드 모델 SystemC 모듈 sc_fir8.h 과 기술 방법 상 차이를 비교해 보기 바란다.

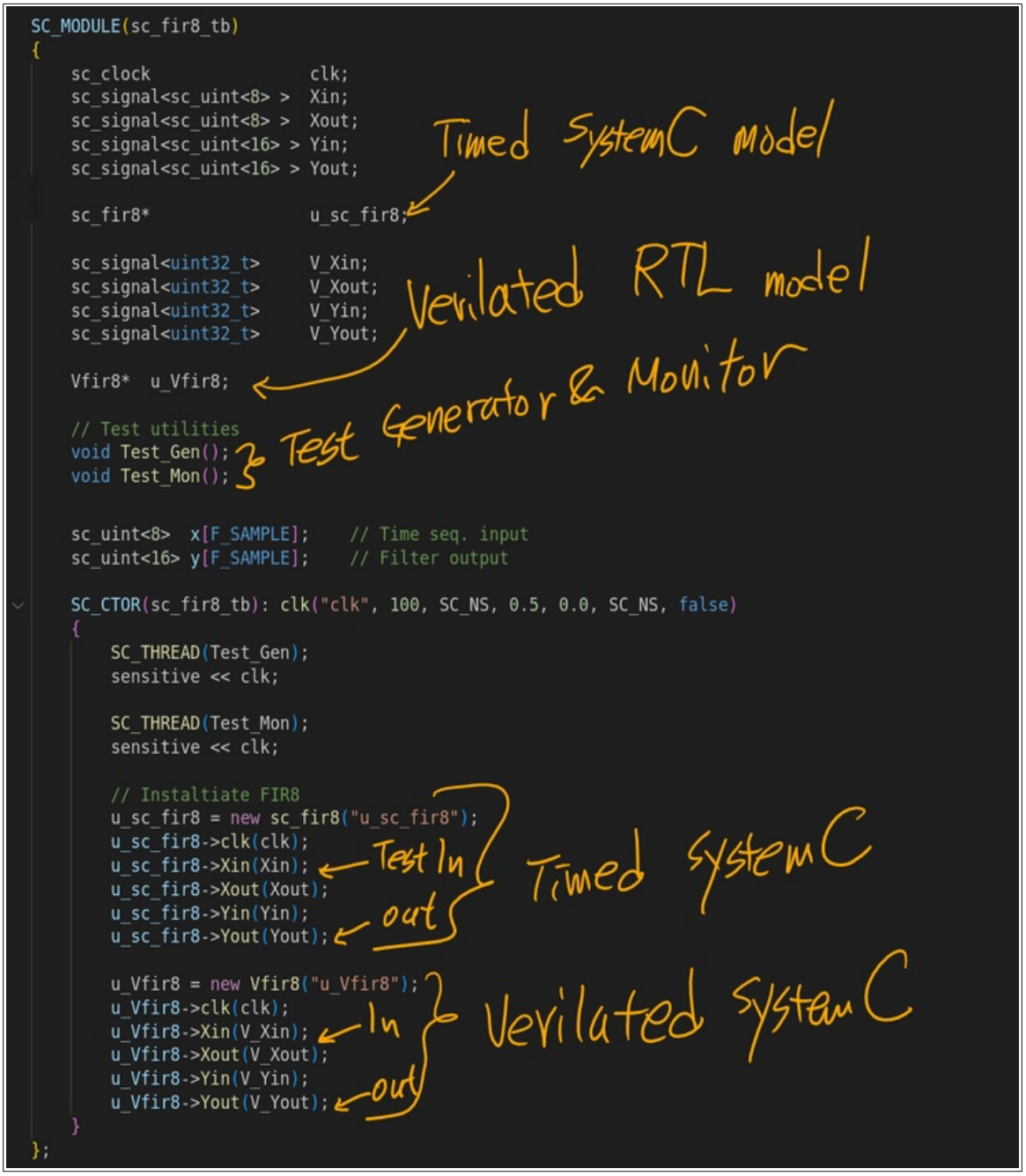
```
`define N_PE_ARRAY  8

module fir8(clk, Xin, Xout, Yin, Yout);
input           clk;
input  [ 7:0]   Xin;
output [ 7:0]   Xout;
input  [15:0]   Yin;
output [15:0]   Yout;

wire [ 7:0] X[`N_PE_ARRAY+1];
wire [15:0] Y[`N_PE_ARRAY+1];

localparam [7:0] C[`N_PE_ARRAY] = {4, 12, 25, 34, 34, 25, 12, 4 };

assign X[0] = Xin;
assign Y[0] = Yin;

for(genvar i=0; i<`N_PE_ARRAY; i++)
begin : gen_fir_pe
    fir_pe u_fir_pe (
        .clk(clk),
        .Cin(C[i]),
        .Xin(X[i]),
        .Xout(X[i+1]),
        .Yin(Y[i]),
        .Yout(Y[i+1]));
end

assign Xout = X[`N_PE_ARRAY];
assign Yout = Y[`N_PE_ARRAY];

endmodule
```

IV-3. 병행 시뮬레이션 테스트 벤치 (Co-simulation Testbench)

앞서 SystemC로 타임드 테스트벤치를 마련해 놓았으므로 베릴로그로 작성된 PE의 RTL 모델을 검증하기 위해 별도의 테스트벤치를 작성할 필요는 없다. SystemC로 작성한 타임드 테스트 벤치를 베릴로그 RTL 모델의 검증에 재사용(Re-Usable Testbench)한다. 베릴로그는 테스트벤치를 위해 RTL 보다 다소 향상된 추상화 수준을 지원한다. 방대한 량의 테스트 벡터를 읽기 위한 파일 입출력이 가능 하다. 하지만 C++ 의 추상화 수준에 비할 바가 못 된다. 게다가 알고리즘 검증에 적용된 C++ 테스벤치를 RTL 검증에 적용함으로서 검증의 연속성을 유지하게 되므로 신뢰도를 한층 높일 수 있다.

베릴로그와 C++는 완전히 다른 언어다. 단지 문법의 차이는 차치하더라도 문장의 실행 방식에 있어서 비교할 수 없다. 하드웨어의 동작을 모사 해주는 SystemC는 실행 방식의 격차를 좁혀 준다. Verilator는 베릴로그로 묘사된 하드웨어를 C++로 변환해 주는 오픈-소스 도구다. 베릴로그의 문법을 모두 수용하지는 못하지만 RTL은 완벽하게

변환한다. C++로 변환된 모델은 베릴로그 시뮬레이터보다 실행 속도도 빠르다. 무엇보다도 베릴로그 HDL을 C++의 영역으로 옮김으로써 최고 추상화 수준의 테스트 환경을 구축 할 수 있다. 실행 방식과 목적이 다른 컴퓨팅 언어(소프트웨어 개발용 언어와 하드웨어 묘사용 언어)를 한데 엮어 시뮬레이션을 수행하는 기법을 Co-Simulation(병행 시뮬레이션)이라 한다.

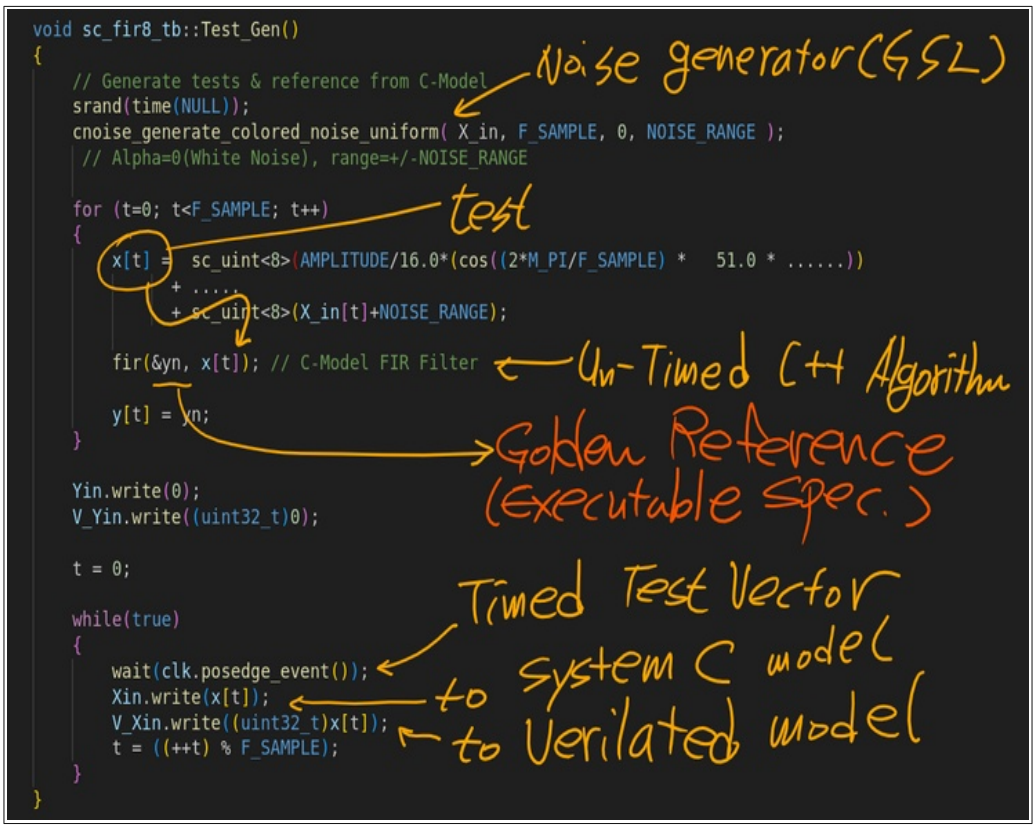



```
SC_MODULE(sc_fir8_tb)
{
    sc_clock                clk;
    sc_signal<sc_uint<8> >  Xin;
    sc_signal<sc_uint<8> >  Xout;
    sc_signal<sc_uint<16> > Yin;
    sc_signal<sc_uint<16> > Yout;

    sc_fir8*                u_sc_fir8;

    sc_signal<uint32_t>     V_Xin;
    sc_signal<uint32_t>     V_Xout;
    sc_signal<uint32_t>     V_Yin;
    sc_signal<uint32_t>     V_Yout;

    Vfir8*  u_Vfir8;

    // Test utilities
    void Test_Gen();
    void Test_Mon();

    sc_uint<8>  x[F_SAMPLE];    // Time seq. input
    sc_uint<16> y[F_SAMPLE];    // Filter output

    SC_CTOR(sc_fir8_tb): clk("clk", 100, SC_NS, 0.5, 0.0, SC_NS, false)
    {
        SC_THREAD(Test_Gen);
        sensitive << clk;

        SC_THREAD(Test_Mon);
        sensitive << clk;

        // Instaltiate FIR8
        u_sc_fir8 = new sc_fir8("u_sc_fir8");
        u_sc_fir8->clk(clk);
        u_sc_fir8->Xin(Xin);
        u_sc_fir8->Xout(Xout);
        u_sc_fir8->Yin(Yin);
        u_sc_fir8->Yout(Yout);

        u_Vfir8 = new Vfir8("u_Vfir8");
        u_Vfir8->clk(clk);
        u_Vfir8->Xin(V_Xin);
        u_Vfir8->Xout(V_Xout);
        u_Vfir8->Yin(V_Yin);
        u_Vfir8->Yout(V_Yout);
    }
};
```

앞서 만든 타임드 SystemC 테스트벤치는 sc_fir8 모델 만을 위한 것이었다. 이제 베릴로그 모델도 준비되었으므로 두 모델을 함께 테스트벤치에 사례화 한다. Verilator는 베릴로그 모델을 SystemC 모델로 변환 하면서 베릴로그 모듈명에 V를 붙인다. 베릴로그 모델 FIR 필터의 모듈명이 fir8 이었다. 변환된 SystemC 모듈명은 Vfir8 이다.

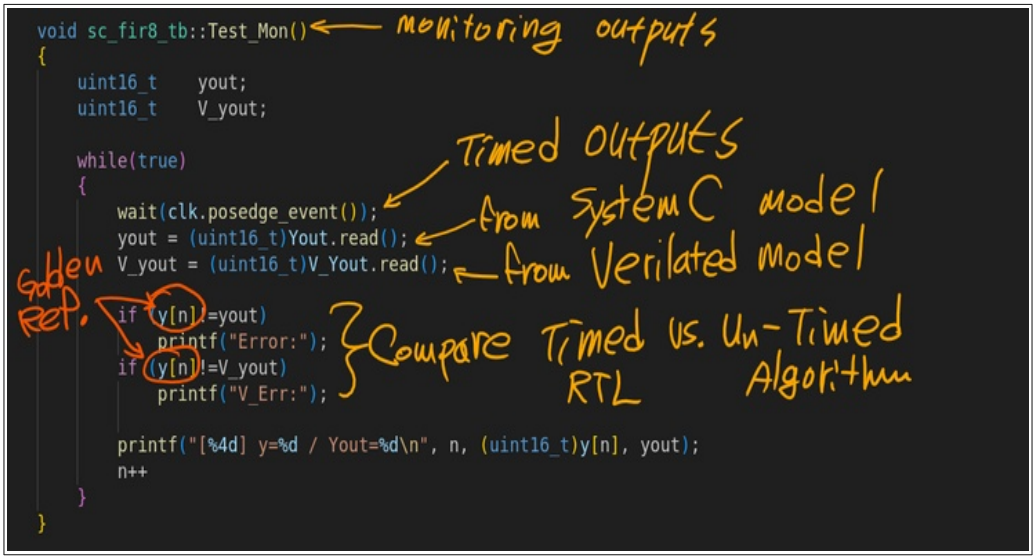
Verilator 가 변환한 SystemC 모델들은 현재 디렉토리에서 obj_dir에 저장된다. 타임드 SystemC 모델과 베릴로그에서 변환된 모델을 사례화한 테스트벤치는 위와 같다.

타임드 테스트 벡터 생성 쓰레딩 함수 Test_Gen()는 다음과 같다. 언-타임드 C++ 알고리즘의 fir()로 테스트 신호를 미리 준비 시킨 후 클럭의 동기에 맞춰 두 타임드 모델에 입력을 주고 있다.




```
void sc_fir8_tb::Test_Gen()
{
    // Generate tests & reference from C-Model
    srand(time(NULL));
    cnoise_generate_colored_noise_uniform( X_in, F_SAMPLE, 0, NOISE_RANGE );
     // Alpha=0(White Noise), range=+/-NOISE_RANGE

    for (t=0; t<F_SAMPLE; t++)
    {
        x[t] =  sc_uint<8>(AMPLITUDE/16.0*(cos((2*M_PI/F_SAMPLE) *   51.0 * ......))
              + .....
              + sc_uint<8>(X_in[t]+NOISE_RANGE);

        fir(&yn, x[t]); // C-Model FIR Filter

        y[t] = yn;
    }

    Yin.write(0);
    V_Yin.write((uint32_t)0);

    t = 0;

    while(true)
    {
        wait(clk.posedge_event());
        Xin.write(x[t]);
        V_Xin.write((uint32_t)x[t]);
        t = ((++t) % F_SAMPLE);
    }
}
```

타임드 출력 모니터링 함수 Test_Mon()는 아래와 같다. 시뮬레이션 시작 전에 생성해둔 알고리즘의 출력을 표준으로 삼아 두 타임드 모델의 출력과 비교한다.




```
void sc_fir8_tb::Test_Mon()
{
    uint16_t    yout;
    uint16_t    V_yout;

    while(true)
    {
        wait(clk.posedge_event());
        yout = (uint16_t)Yout.read();
        V_yout = (uint16_t)V_Yout.read();

        if (y[n]!=yout)
            printf("Error:");
        if (y[n]!=V_yout)
            printf("V_Err:");

        printf("[%4d] y=%d / Yout=%d\n", n, (uint16_t)y[n], yout);
        n++
    }
}
```

IV-4. 실습. FIR 필터의 병행 시뮬레이션 (Lab. FIR filter Co-Simulation)

병행 시뮬레이션(Co-Simulation)은 추상화 수준이 상이한 모델을 함께 엮어 시험하는 검증 기법이다. 앞서 시험했던 알고리즘 필터 함수 fir() 부터 타임드 모델 sc_fir8 에 덧붙여 베릴로그 RTL 모듈 fir8 까지 한데 모은 테스트 벤치를 가지고 시뮬레이션을 수행한다. SystemC/C++ 모델의 빌드와 시뮬레이션 실행은 미리 준비해 둔 메이크 스크립트 Makefile을 사용한다. 예제는 '내 칩 제작서비스 오픈-소스 디자인 킷'의 Tutorials 에 포함되어 있다.

예제 디렉토리로 이동,

```
$ cd ~/ETRI050_DesignKit/Tutorials/2-5_Lab3_FIR8/rtl_verilog
```

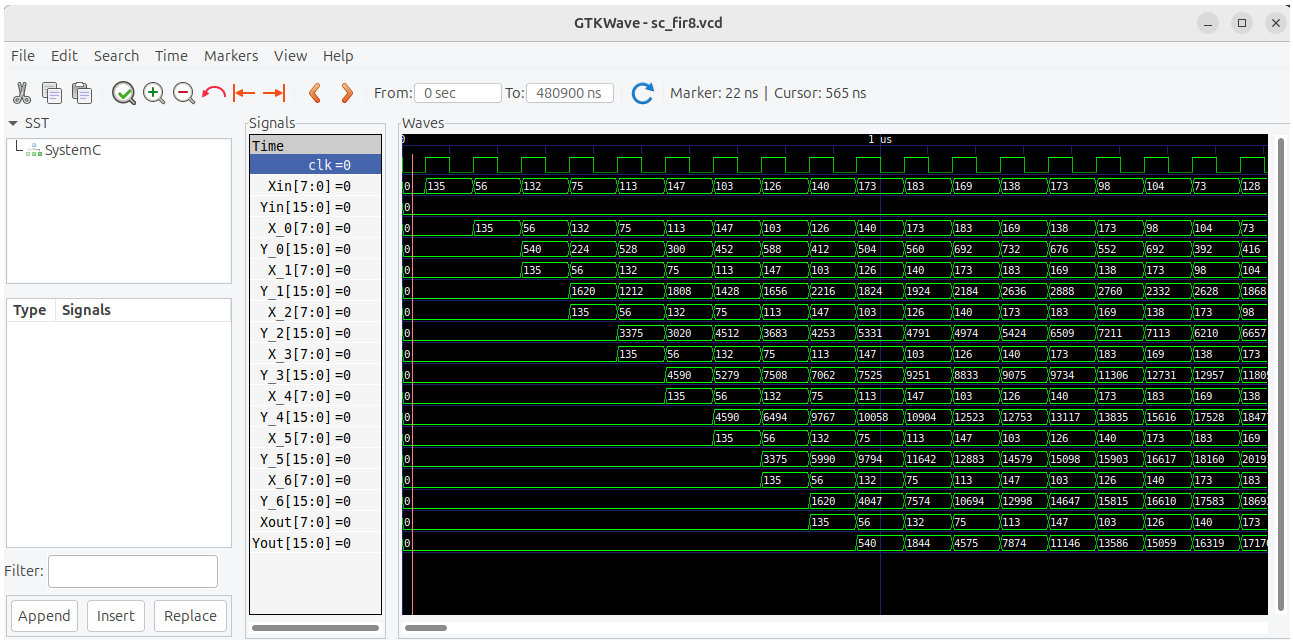
SystemC/C++ 시뮬레이터 빌드,

```
$ make build

  verilator --sc -Wall --trace --top-module fir8 --exe --build \
    -CFLAGS -I../sc_timed -CFLAGS -DVCD_TRACE_FIR8 -CFLAGS \
    -DVERILATED_CO_SIM -CFLAGS -DFIR_MAC_VERSION \
    -LDFLAGS -lgsl \
     fir8.v fir_pe.v \
     ../sc_timed/sc_main.cpp ../sc_timed/sc_fir8_tb.cpp \
     ../c_untimed/fir8.cpp ../c_untimed/cnoise.cpp
  make[1]: Entering directory '.../2-5_Lab3_FIR8/rtl_verilog/obj_dir'
                    ... ...
- V e r i l a t i o n   R e p o r t: Verilator 5.035 ...
- Verilator: ...into 0.074 MB in 10 C++ files needing 0.000 MB
- Verilator: Walltime 13.931 s (elab=0.014, cvt=0.046, bld=13.788) …
```

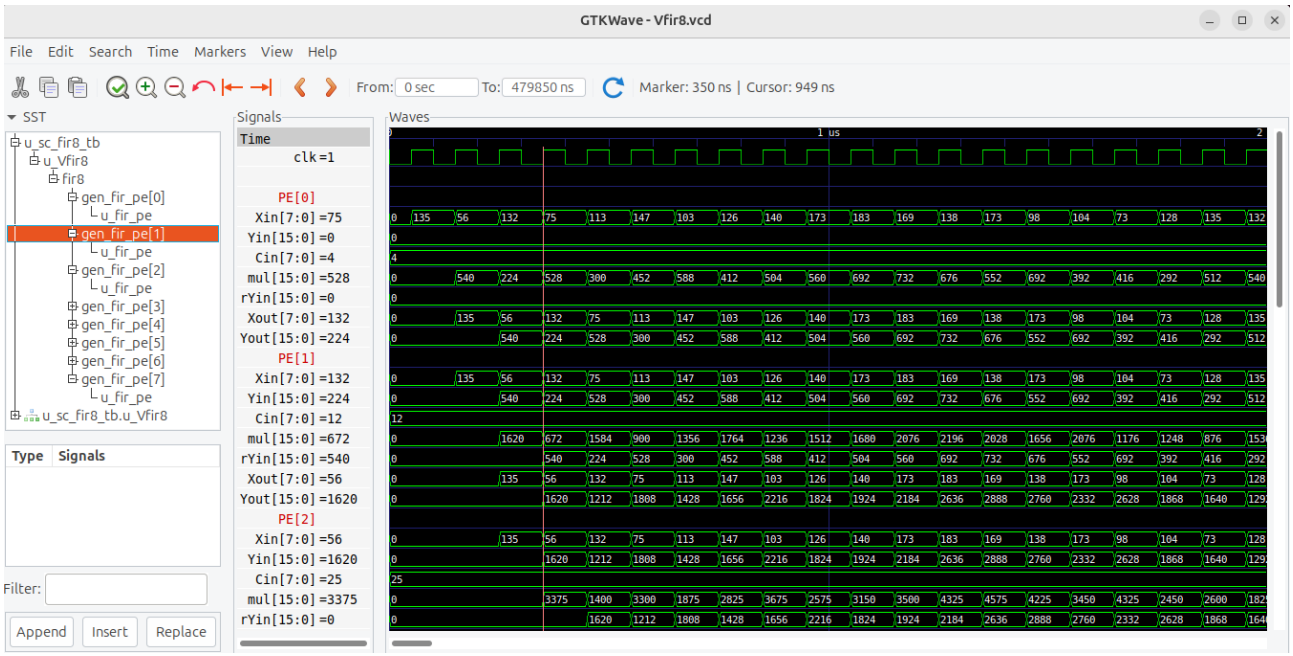
fir8 시뮬레이터 실행,

```
$ make run
./obj_dir/Vfir8
    SystemC 3.0.0-Accellera --- Aug  9 2024 12:41:46
    Copyright (c) 1996-2024 by all Contributors,
    ALL RIGHTS RESERVED
Info: (I703) tracing timescale unit set: 100 ps (sc_fir8.vcd)
Warning: (W509) module construction not properly completed: did you
forget to add a sc_module_name parameter to your module constructor?: module
'u_sc_fir8_tb'
In file: ../../../src/sysc/kernel/sc_module.cpp:376
[   0] y=704 / Yout=704
[   1] y=2304 / Yout=2304
[   2] y=5728 / Yout=5728
....…
[4798] y=20499 / Yout=20499
[4799] y=19391 / Yout=19391

Info: /OSCI/SystemC: Simulation stopped by user.
```

언-타임드 표준 모델과 타임드 모델 그리고 RTL 모델의 비교 결과 오류 없이 수행되었다. 타임드 SystemC 모델은 계층화된 모듈로 sc_fir8 아래에 sc_fir_pe 가 8개 파이프라인 배열 구성이다. 시뮬레이션을 수행하는 동안 VCD에 기록된 하위 모듈의 입출력 신호들을 볼 수 있다.



베릴로그 모델 역시 계층화된 모듈로 fir8 아래에 fir_pe 가 8개 파이프라인 배열 구성이다. VCD에 기록한 하위 모듈의 입출력 신호들을 볼 수 있다.

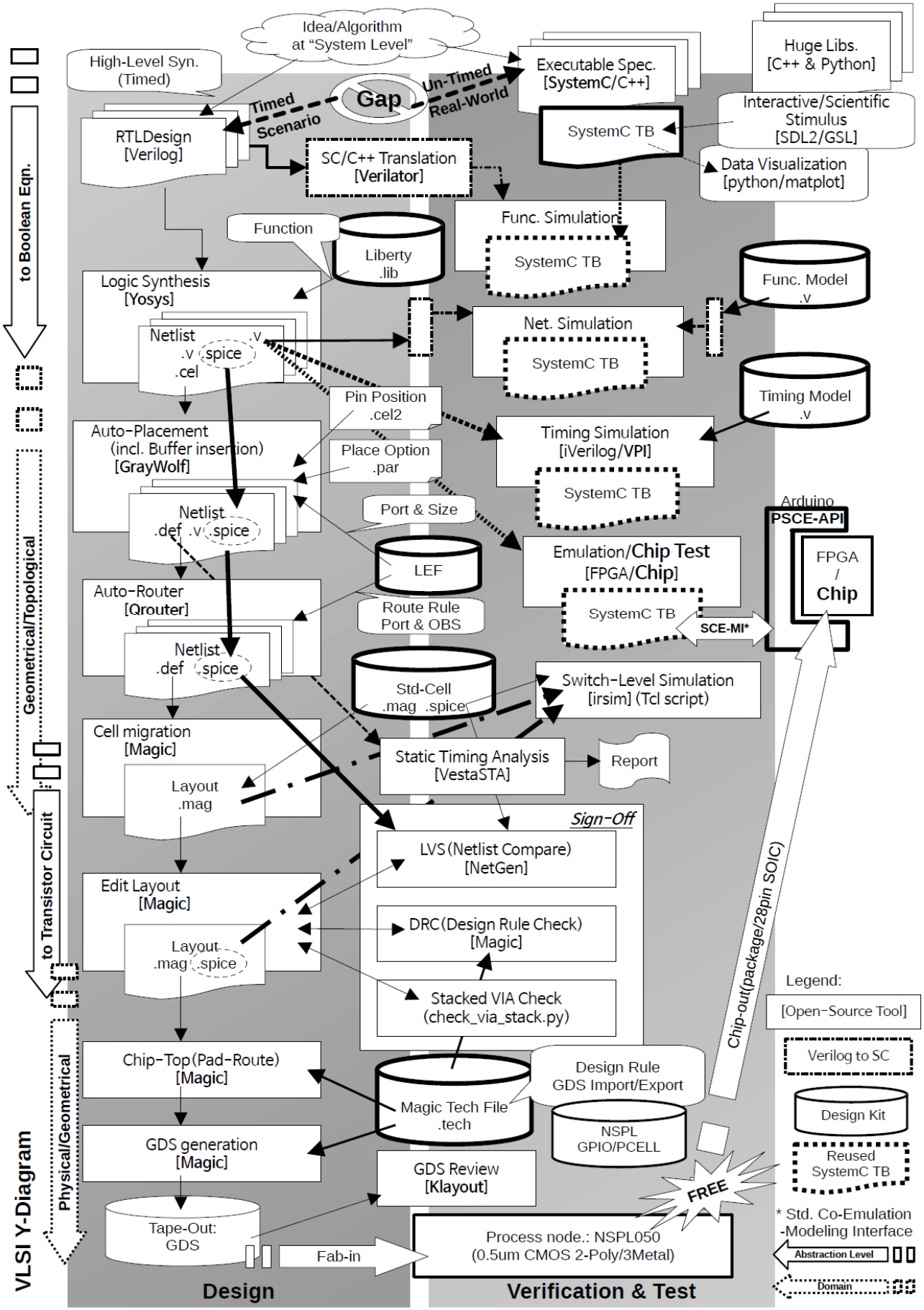


V. 맺음말

오픈-소스 도구를 사용하여 디지털 FIR 필터의 파이프라인 병렬처리 하드웨어 설계 과정을 살펴봤다. FIR 필터 알고리즘의 문서 사양을 토대로 언-타임드 C++ 로 기술하고 시험했다. 병렬 구조를 도출하여 SystemC 타임드 모델링 후 최종적으로 베릴로그 RTL로 기술했다. 베릴로그 RTL 모델의 검증에 언-타임드 알고리즘과 타임드 모델을 모두 엮은 병행 시뮬레이션(Co-Simulation)기법을 적용하였다. FIR 필터의 성능 시험에 GSL이 사용되었고 대용량 시뮬레이션 데이터의 시각화를 위해 파이썬의 패키지를 활용하였다. 설계와 검증에 사용된 도구 Verilator와 SystemC 그리고 GSL은 모두 C++ 언어 기반의 오픈-소스 도구들이다. 여기에 더하여 파이썬의 방대한 패키지(라이브러리)를 하드웨어 설계에 활용할 수 있다. 베릴로그 HDL을 C++ 로 변환해 주는 도구 Verilator와 시스템 수준 모델링 도구 SystemC를 사용한 반도체 설계 방법론은 기존에 축적된 방대한 자원들을 활용할 수 있게해 준다. 보다 실제적인 테스트 입력 데이터의 생성이 가능하며 가시화를 통해 방대한 시험 데이터의 분석이 용이하다.

상용 도구의 사용법이 마치 고급 기술로 여겨지기도 한다. 사용하면서 닥치는 각종 장애를 해결하기도 쉽지 않다. 물어볼 곳 조차 마땅치 않다. 질문에 쓸만한 답을 얻으려면 비용을 지불해야 한다. 심지어 보안유지 서약까지 요구하니 그들만의 닫힌 세계다. 그동안 우리는 소프트웨어 개발도구의 눈부신 발달을 목격했다. 오픈-소스의 공개 정책과 집단 지성의 공이 크다. 다행히 반도체 설계에도 오픈-소스 바람이 불어 괄목할 만한 발전을 이루고 있다.

반도체 설계에 Verilator-SystemC 방법론이 다소 낯설 수 있으나 우리는 이미 C++를 비롯한 컴퓨팅 언어의 코딩에 친숙해져 있다. 예제 프로젝트를 수행해 보면 반도체 설계 장벽이 그리 높지 않다는 것을 알게 될 것이다. 코딩은 도구 사용법 일 뿐이다. 도구의 사용자로서 무엇을 설계할지 고민해보자. SoC를 부르짖기를 한세대가 지났다. 그사이 세계는 CPU에서 벗어나 인공지능 같은 알고리즘 전용 칩이 대세다. 전부 디지털 컴퓨팅이다. 다음편은 베릴로그 RTL의 합성과 네트리스트 시뮬레이션 그리고 GDS 생성을 다룬다.


Idea/Algorithm at "System Level"
High-Level Syn. (Timed)
Executable Spec. [SystemC/C++]
Huge Libs. [C++ & Python]
Timed Scenario
Gap
Un-Timed Real-World
RTLDesign [Verilog]
SystemC TB
Interactive/Scientific Stimulus [SDL2/GSL]
SC/C++ Translation [Verilator]
Data Visualization [python/matplot]
to Boolean Eqn.
Function
Liberty .lib
Func. Simulation
SystemC TB
Logic Synthesis [Yosys]
Func. Model .v
Netlist .v .spice .cel
Net. Simulation
SystemC TB
Timing Model .v
Pin Position .cel2
Auto-Placement (incl. Buffer insertion) [GrayWolf]
Timing Simulation [iVerilog/VPI]
SystemC TB
Place Option .par
Netlist .def .v .spice
Port & Size
Arduino PSCE-API
FPGA / Chip
LEF
Emulation/Chip Test [FPGA/Chip]
SystemC TB
SCE-MI*
Auto-Router [Qrouter]
Route Rule Port & OBS
Netlist .def .spice
Std-Cell .mag .spice
Switch-Level Simulation [irsim] (Tcl script)
Geometrical/Topological
Cell migration [Magic]
Static Timing Analysis [VestaSTA]
Report
Layout .mag
Sign-Off
LVS (Netlist Compare) [NetGen]
Edit Layout [Magic]
DRC (Design Rule Check) [Magic]
Layout .mag .spice
Stacked VIA Check (check_via_stack.py)
to Transistor Circuit
Chip-out(package/28pin SOIC)
Legend:
[Open-Source Tool]
Chip-Top (Pad-Route) [Magic]
Design Rule GDS Import/Export
Verilog to SC
Magic Tech File .tech
NSPL GPIO/PCELL
Design Kit
GDS generation [Magic]
Reused SystemC TB
GDS Review [Klayout]
FREE
* Std. Co-Emulation -Modeling Interface
Physical/Geometrical
Tape-Out: GDS
Fab-in
Process node.: NSPL050 (0.5um CMOS 2-Poly/3Metal)
Abstraction Level
Domain
VLSI Y-Diagram
Design
Verification & Test