ETRI 0.5um CMOS Std-Cell DK:

C++ 템플릿 크래스와 SystemC 의 최소한 이해

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 14 일

ETRI 한국전자통신연구원
www.etri.re.kr

C++ 템플릿 크래스와 SystemC의 최소한 이해

목차:

[주] '디자인 킷'의 깃-허브 저장소 내용이 매주 갱신되고 있습니다. 기존에 받은 내용이 있다면 갱신 하십시요.

```
$ cd ~/ETRI050_DesignKit
$ git pull
```

[주] '디자인 킷'을 새로 받는 경우 앞서 발행한 설치 문서를 참고하십시요.

기술노트1, 2: 가상머신 리눅스 설치[pdf], 오픈-소스 반도체 설계도구 설치[pdf] [바로가기]

기술노트3: 하드웨어 기술 언어의 코딩 스타일[pdf][바로가기]

---

1. 개요

컴퓨팅 환경을 접하며 생활하는 요즘 굳이 전공을 따지지 않더라도 컴퓨팅 언어에 친숙하다. 반도체(하드웨어) 설계 역시 컴퓨팅 언어를 사용하여 설계 생산성을 한층 높여왔다. 이에 약간의 프로그래밍 언어에 대한 이해를 가졌다면 반도체 설계를 시작할 수 있다. 넓은 추상성을 포용하는 C++는 가장 널리 사용되는 컴퓨팅 언어로서 알고리즘의 기술은 물론 하드웨어 시스템 모델링까지 활용 된다. SystemC는 하드웨어 기술에 필요한 크래스와 병렬 시뮬레이션 커널을 갖춘 C++ 라이브러리다. ETRI 0.5um CMOS 공정 표준셀 디자인 킷(이하 '디자인 킷')에서 제공하는 각종 예제와 학습 자료들의 테스트벤치는 SystemC 기반으로 작성 되었다. C++의 템플릿 크래스의 최소한 이해, 그리고 베릴로그와 비교, 병렬 시뮬레이션 커널의 사건 구동 병렬 시뮬레이터가 작동 하는 원리를 정성적으로 설명한다.

SystemC를 처음 접하면 마치 새로운 언어처럼 보이나 실은 C++ 그 자체다. 다양한 템플릿 크래스와 매크로가 매우 현명하게 사용 되었을 뿐이다. 본 문서는 SystemC를 활용하여 시스템 수준 모델링을 시작하기 전에 C++ 템플릿 크래스의 최소한 이해를 돕기 위한 것이다. 본 문서에서 설명하는 예제의 파일들은 디자인 킷[바로가기]의 튜토리얼 중 아래 디렉토리에서 찾아볼 수 있다.

```
$ cd ~/ETRI050_DesignKit/Tutorials/2-2 Verilog SystemC in a Day
$ tree
.
├── ex1_template_class: C++ 템플릿 크래스의 최소한
│   └── templated_class.cpp
├── ex2_delta_cycle: 사건 구동 시뮬레이션 커널(병렬실행의 원리)
│   ├── Makefile
│   ├── sc_delta_cycle.h
│   ├── sc_delta_cycle_TB.gtkw
│   ├── sc_delta_cycle_TB.h
│   └── sc_main.cpp
├── ex3_dff_strange
│   ├── dff: D-플립플롭의 행위수준 Verilog 기술
│   │   ├── dff.v
│   │   ├── Makefile
│   │   ├── sc_dff_TB.h
│   │   └── sc_main.cpp
│   ├── sc_dff: D-플립플롭의 행위수준 SystemC 기술
│   │   ├── Makefile
│   │   └── Vdff.h
│   └── sc_dff_strange: 감응 지정이 행위에 미치는 영향
│       ├── Makefile
│       └── Vdff.h
└── ex4_rsff_recursive_event
    ├── rsff_gates: 게이트 수준 RS-래치 Verilog 기술
    │   ├── Makefile
    │   └── rsff.v
    ├── sc_rsff_gates: 게이트 수준 RS-래치 SystemC 기술
    │   ├── Makefile
    │   ├── sc_main.cpp
    │   ├── sc_rsff_TB.h
    │   └── Vrsff.h
    └── sc_rsff_gates_hier: SystemC 기술(계층적)
        ├── Makefile
        ├── sc_main.cpp
        ├── sc_nand.h
```

```
├─ sc_rsff.h
└─ sc_rsff_TB.h
```

2. C++의 크래스: 객체를 담는 그릇

SystemC를 맞이하기 전에 먼저 C++의 객체 표현 방식을 알아보자. C++를 객체 지향적 언어(Object Oriented Language)라고 한다. 객체 지향적 프로그래밍(OOP)은 현대 컴퓨팅 언어의 상식이 되었다. 대부분 컴퓨팅 언어에서 객체를 표현하는 형식으로 크래스(class)의 개념을 채용하고 있다. 크래스는 단적으로 복합 자료(변수)를 묶는 C 언어 구조체(structure)의 확장이다. 자료에 더하여 이 자료를 취급하는 행동(함수)을 함께 묶어 담는다.

2-1. 크래스의 예: "복소수 다루기"

복합 자료를 묶는 가장 널리 활용되는 예가 아마도 복소수(complex number) 체계일 것이다. 복소수 체계에서 한 객체는 두 개의 숫자 값, 허수(Image)와 실수(Real)로 표현된다.

```
Complex_Number = (Real) + j(Image)
```

이를 컴퓨팅을 위해 C 언어의 자료 구조체로 표현하면 다음과 같다.

```
struct Complex_t
{
    int Real;
    int Image;
}
```

복소수 체계를 다루는 방법은 일반적인 수체계와 상이하다. 예를 들어 복소값의 크기(power)는 켤레(conjugate) 복소수를 취한 제곱근 구하기다. 앞서 정의한 복소수 구조체 Complex_t 를 활용하여 기술하면 다음과 같다.

```
struct Complex_t X;
mult = (X.Re*X.Re) - (X.Im*X.Im);
power =  sqrt(abs(mult));
```

이 계산법은 구조체로 선언한 X가 복소수 일 때 유효하다. 따라서 복소수를 취급하는 방법(함수)을 아예 자료 구조체 내에 넣어두는 것이 현명하다.

```
class Conplex_t {
  int Re;
  int Im;
  int conjugate(){
      return  (Re*Re - Im*Im);
  }
  int power(){
      return (sqrt(abs(conjugate())));
  }
};
```

자료(소속 자료, member data)와 그 자료를 취급하는 방법(소속 함수, member function)을 한 그릇에 놓기 위한 구문 형식이 바로 크래스(class)다. 취급할 수 있는 숫자가 정수(int)뿐만 아니라 부동 소숫점 실수(float)도

있으므로 위의 크래스에 더하여 실수형 자료 선언으로 또 만들어야 한다면 매우 불합리하다. 이럴 때 아래와 같이 템플릿(template)을 사용한다. C++의 템플릿은 자료형에 맞춰 크래스를 재사용하는 아주 유용한 방법을 제공한다.

```
template<typename T>
class Complex_t {
 private:
    T Re;
    T Im;
 public:
    Complex_t(T re, T im) {     // Constructor
        Re = re; Im = im;
    }
    void put_Re(T x) { Re = x;}
    void put_Im(T x) { Im = x;}
    T get_Re() { return Re;}
    T get_Im() { return Im;}
    T conjugate(){ return((real*real)-(image*image));}
    T Power(){ return (sqrt(abs(conjugate())));}
};
```

객체를 선언하여 사례화(instantiation) 할 때 자료형을 템플릿에 지정한다.

```
Complex_t<int>    IntX;
Complex_t<float> FloatY;
```

복소수 표현에서 실수부와 허수부 만으로 의미가 없으므로 크래스의 내부 자료 Re 와 Im은 외부에 노출되지 않도록 private 에 두었다. 내부로 접근 하려면 외부 공개된 소속 함수들을 통한다. 이를 위해 공용 public 으로 지정했다.

```
IntX.put_Image(5);
float Re_of_Y = Float.Y.get_Real();
```

크래스와 동일한 이름을 갖는 특별한 함수가 있다. 크래스 객체를 선언 할 때 내부를 구축하는 역할을 한다. 이를 구성자(constructor)라 한다. 구성자 함수는 선언될 때 한번 호출될 뿐이다. 위의 예에서 구성자는 객체 선언과 함께 내부 자료를 초기화한다.

```
Complex_t<int>      intX(2,3);
Complex_t<float>    floatY(3.14, 4.5);
```

2-2. 템플릿 크래스의 예: "비트 벡터"

템플릿의 활용도는 매우 넓다. C++의 기본 자료형의 경우 비트 폭과 연산 방법이 정해져 있다. 하지만 임의 비트 폭(bit-width)을 갖는 RTL에서 하드웨어를 묘사하려면 별도의 객체 표현 방법이 필요하다. 표현 뿐만 아니라 이 자료를 다루는 연산자 또한 이에 맞춰져야 한다. 비트 폭이 N 디지털 데이터 자료형을 템플릿 크래스로 선언하면 다음과 같다.

```
template <u_int N>
class bit_vector_t {
    bool m_next_val[N];
    bool m_curr_val[N];
    int  nLen;
    char m_sz_val[N+1];

public:

    bit_vector_t():nLen((int)N) {
        for (int i=0; i<N; i++) m_next_val[i] = false;
        for (int i=0; i<N; i++) m_curr_val[i] = false;
    }
    bit_vector_t(const char* szVal):nLen((int)N) {
        write(szVal);
    }
    char* to_string();
    void write(const char* szVal);
    int length();

    // overload the | operator

    bit_vector_t operator | (const bit_vector_t& obj1, const
bit_vector_t& obj2);

    // overload the & operator

    bit_vector_t operator & (const bit_vector_t& obj1, const
bit_vector_t& obj2);
};
```

크래스 내부에서 2진 디지털 데이터를 표현하기 위해 C++의 bool 형 배열로 선언 하였다. 아울러 두가지 구성자를 두고 있는데, 하나는 모두 false 로 초기화 하는 경우와 다른 하나는 임의 값으로 초기화 하는 목적이다. 고정된 초기값을 갖는 10비트짜리 디지털 객체의 선언은 다음과 같다.

```
bit_vector_t<10> A;
bit_vector<10>   B;
```

임의 비트폭의 상수형 2진 자료(리터럴, literal)를 표현하는 방법은 객체를 정의할 때 약속한다. 위의 경우 문자열(char *)로 하기로 한다. 임의의 초기값을 갖는 10비트짜리 디지털 객체의 선언은 다음과 같다.

```
bit_vector_t<10> X("0101010101");
bit_vector_t<10> Y("1010101010");
```

객체의 소속 자료로 선언된 부울형 배열을 취급하는 방법을 소속 함수로 가지고 있다. 쓰기방법 write(), 읽어서 문자열로 표현하는 방법 to_string()등이 있다.

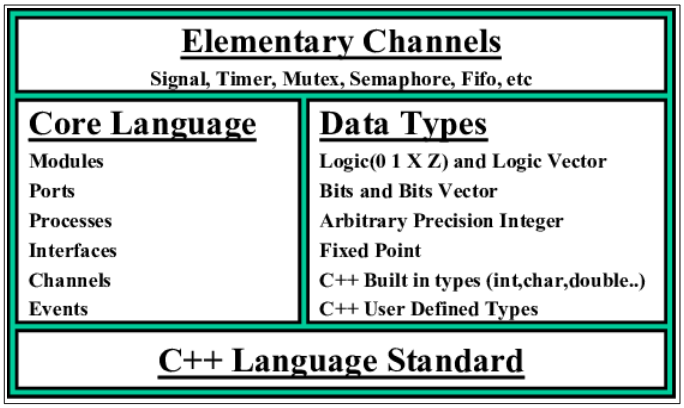
```
void bit_vector_t::write(const char* szVal) {
    if ((int)strlen(szVal)!=nLen) {
```

```
            fprintf(stderr, "Bit Vector NOT match!\n");
            return;
        }
        for (int i=0; i<strlen(szVal); i++)
            if (szVal[i]=='1')  m_curr_val[i] = true;
            else                m_curr_val[i] = false;
    }

    char* bit_vector_t::to_string() {
        for (int i=0; i<(int)N; i++)
            if (m_curr_val[i])  m_sz_val[i] = '1';
            else                m_sz_val[i] = '0';
        m_sz_val[N] = '\0';
        return m_sz_val;
    }
```

만일 써넣으려는 리터럴 상수가 객체의 선언된 자료형과 일치하지 않을 경우 오류로 간주하고 이를 실행 중 경고한다. C++언어 고유의 자료형 사이의 연산에 대하여 내부적으로 규칙을 가지고 있다(컴파일러는 자료형 변환으로 정밀도가 상실될 수 있다는 경고를 낸다). 하지만 예의 bit_vector<> 자료형은 비트 폭이 정확히 일치해야 하므로 연산 규칙을 만들어 주어야 한다. C++는 특별한 객체에 대하여 연산자와 규칙을 정의해 줄 수도 있다. 다음은 비트 단위 논리합(|)과 논리 곱(&) 연산자를 정의한 것이다. 다소 낯설기는 하지만 연산자에 |(), &()처럼 함수 표현이 가능하다. C/C++언어에서 소괄호 ()는 오직 함수의 의미라는 규칙을 매우 충실하게 따르고 있다.

```
// overload the | operator
bit_vector_t operator | (const bit_vector_t& obj1, const bit_vector_t& obj2)
{
    bit_vector_t<N> Temp;
    for (int i=0; i<N; i++)
        Temp.m_curr_val[i] = obj1.m_curr_val[i] | obj2.m_curr_val[i];
    return Temp;
}
// overload the & operator
bit_vector_t operator & (const bit_vector_t& obj1, const bit_vector_t& obj2)
{
    bit_vector_t<N> Temp;
    for (int i=0; i<N; i++)
        Temp.m_curr_val[i] = obj1.m_curr_val[i] & obj2.m_curr_val[i];
    return Temp;
}
```

크래스의 작은 부분만 살펴 봤다. 이정도 만으로도 C++는 하드웨어 모델링을 위해 만들어진 것은 아닐까라는 생각마져 든다. C++ 언어가 넓은 추상성을 가진 최고의 언어라는 점에 동의할 것이다.



[출처] UML for ESL Design - Basic Principles, Tools, and Applications

SystemC는 하드웨어를 묘사하기 위해 C++의 템플릿 크래스를 매우 현명하게 활용한 라이브러리다. 2진 논리를 표현할 수 있는 비트 형과 비트 벡터형은 다음과 같다.

```
sc_bit
    single bit value: '0', '1'
sc_bv<N>
    Vector of sc_bit values, N is number of bits
    Methods:
        range(x,y), to_int(), to_uint(), length()
        set_bit(i, d), get_bit(i), to_string()
        and_reduce(), nand_reduce(), nor_reduce() or_reduce(),
        xor_reduce(), xnor_reduce()
```

2진 수 비트 형은 물론 하드웨어의 베릴로그와 호환되는 다치 논리(multi-valued) 형도 있다.

```
sc_logic
    single bit value: '0', '1', 'X', 'Z'
sc_lv<N>
    Vector of sc_logic values, N is number of bits
    Methods:
        range(x,y),to_int(),to_uint(),length(),
        set_bit(i, d), get_bit(i), to_string(),
        and_reduce(), nand_reduce(), nor_reduce(), or_reduce(),
        xor_reduce(), xnor_reduce()
```

C++의 템플릿 크래스의 기초 정도만 이해하는 것으로도 SystemC를 활용하여 반도체 설계를 충분히 시작 할 수 있다. 최소한 첫 만남의 낯설음을 덜 수 있다. 이제 자주 활용하며 익혀보자. 하드웨어 기술자로서 또 다른 HDL을 다뤄야 한다는 부담을 갖는다면 피곤할 일이다. C++의 넓은 활용도를 생각해 보길 바란다. 하드웨어 기술자를 위한 SystemC를 개괄적으로 설명한 블로그 글이 있다. 일독을 권한다.

"Tour of SystemC for Hardware Engineers"
https://techne-atelier.com/digital-design/a-tour-of-systemc/

아울러 SystemC를 쉽게 설명한 동영상도 있다.

FORTE Design, Learn SystemC,
https://www.youtube.com/playlist?list=PLcvQHr8v8MQLj9tCYyOw44X1PLisEsX-J

2-3. C++ 템플릿 크래스 예제

간단한 예제를 통해 C++ 템플릿 크래스를 익혀보자. '디자인 킷'의 튜토리얼 폴더에 크래스 템플릿의 예제 폴더로 이동,

```
$ cd ~/ETRI050_DesignKit/Tutorials/2-2_Verilog_SystemC_in_a_Day/ex1_template_class
```

파일 목록을 보면 C++ 소스 파일이 한개 있다.

```
$ ll
total 12
drwxrwxr-x 2 goodkook goodkook 4096 Jul  7 11:09 ./
drwxrwxr-x 6 goodkook goodkook 4096 Jul  9 17:30 ../
-rw-rw-r-- 1 goodkook goodkook 3311 Jul  7 11:05 templated_class.cpp
```

GNU C++ 컴파일러로 소스 파일 templated_class.cpp를 읽어 컴파일 및 링크하여 얻은 실행화일의 이름은 기본적으로 a.out이다. 다른 이름으로 생성하려면 -o 옵션을 사용한다. 아래의 명령줄에서 실행 파일의 이름은 templated_class 다.

```
$ g++ -o templated_class templated_class.cpp
```

예제를 컴파일 하여 생성된 바이너리 파일을 실행 시킨다. 리눅스 명령줄 환경('쉘' 이라고 한다)에서 현재 디렉토리가 실행 파일 탐색 경로 PATH 에 포함되어 있지 않다. 현재 디렉토리에서 실행파일 templated_class를 찾아 실행한다.

```
$ ./templated_class
Float: Re=2.000000 Im=1.000000
Int. : Re=4 Im=3
[Conjugate] Float:3.000000 Int.:7
[Power] Float:1.732051 Int.:2
1010101010 | 0101010101 = 1111111111
Bit Vector NOT match!
Bit Vector NOT match!
1111100000 & 0001100011 = 0001100000
```

예제의 C++ 소스 파일 templated_class.cpp를 열어 내용을 보면 위의 실행이 어떻게 출력 되었는지 쉽게 이해될 것이다. 쉘 명령 less는 파일을 열어 화면에 페이지 단위로 보여준다.

```
$ less templated_class.cpp
```

간단한 복소수 객체 Complex_t 와 비트 벡터 객체 bit_vector_t를 템플릿 크래스로 정의하고 사용예를 볼 수 있다.

[주] 리눅스의 명령줄 환경(Command-Line Interface, CLI) 본-어게인 쉘(bash)의 환경변수 PATH를 출력해 보면 현재 디렉토리가 포함되어 있지 않은 것을 알 수 있다.

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/snap/
bin:/snap/bin
```

쉘 명령으로 현재 디렉토리를 환경변수에 포함 시킬 수 있다.

```
$ PATH=.:$PATH
$ echo $PATH
.:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/snap/
bin:/snap/bin
```

환경변수 설정은 해당 터미널의 명령줄 환경에서 만 유효하다. 터미널의 닫으면 무효화 된다. 터미널을 열 때마다 환경 변수를 지정하려면 리눅스 본-어게인 쉘 환경 설정 파일 ~/.bashrc 을 수정한다.

3. SystemC의 병렬 시뮬레이션 커널

컴퓨팅 언어로 알고리즘을 기술하는 목적을 생각해 보자. 디지털 전자회로의 작동은 트랜지스터를 매우 낮은 수준의 스위칭 동작으로 근사할 수 있다. 몇개의 트랜지스터를 사용하여 지연된 스위칭으로 디지털 논리회로(논리 게이트와 플립플롭)를 설명 할 수 있으며 수많은 논리 소자를 동원하여 거대한 컴퓨팅 시스템을 구성 할 수 있다. 반도체 설계를 위해 트랜지스터를 동원하기 보다 좀 더 높은 추상화된 수준에서 알고리즘을 기술 함으로써 높은 설계 생산성을 얻을 수 있다. 인간의 언어와 유사한 컴퓨팅 언어를 사용하여 알고리즘을 기술하고 이를 자동화된 도구(컴파일러 소프트웨어)를 써서 물리적 장치로 바꾸는 방법은 이미 소프트웨어 개발에서 널리 채택되고 있다. 디지털 하드웨어 또한 다르지 않다.

[주] 아래의 동영상은 트랜지스터 회로가 2진 부호를 받아 어떤 동작을 하는지 쉽게 설명한다.

HOW TRANSISTORS RUN CODE?
https://youtu.be/HjneAhCy2N4?si=PDoHQ4b3CiD_72eW

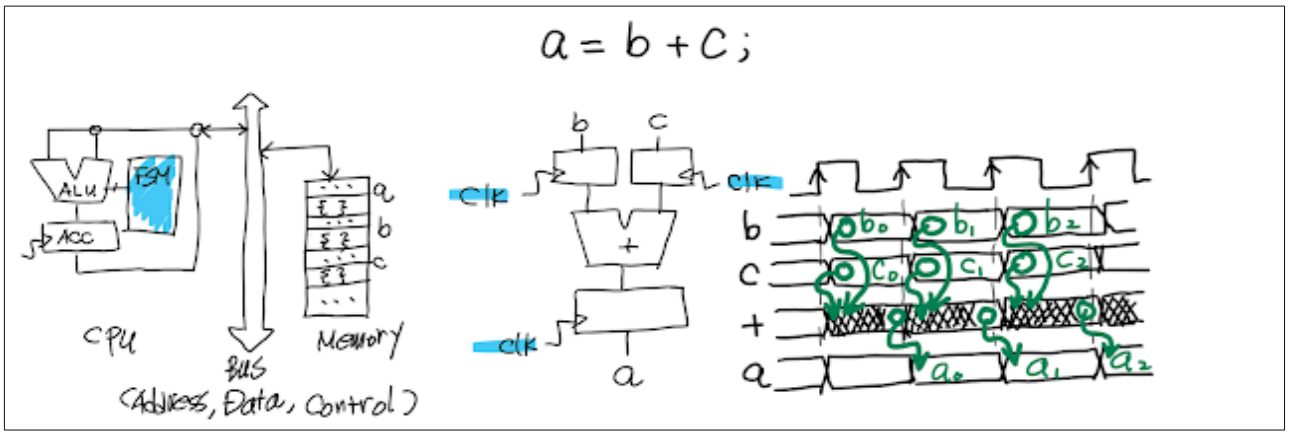
컴퓨팅 언어는 기본적으로 대수 표현을 차용한 문장과 기본 영어 문법을 원용한 제어구조로 구성된다. 전자회로를 대상으로 하므로 전기적 특성이 언어의 규칙에 반영되어야 한다. 대표적으로 문장의 할당 연산자 '='의 왼편과 오른편에 놓일 수 있는 객체의 입출력 속성은 확실히 구분되어야 한다. 이에 덧붙여 전자회로 하드웨어를 표현한 문장의 실행은 모두 동시에 이뤄지며 절차적 표현을 별도로 수용한다. 문법에 맞춰 작성된 다수의 문장이 의도한 대로 작동할 것이라는 보장은 없다. 큰 비용을 들여 제작된 실물의 동작에 오류가 발견되면 이미 늦다. 시뮬레이션은 실물이 제작되기 전에 그 동작을 미리 확인해 보는 행위다.

3-1. 레지스터 전송 수준

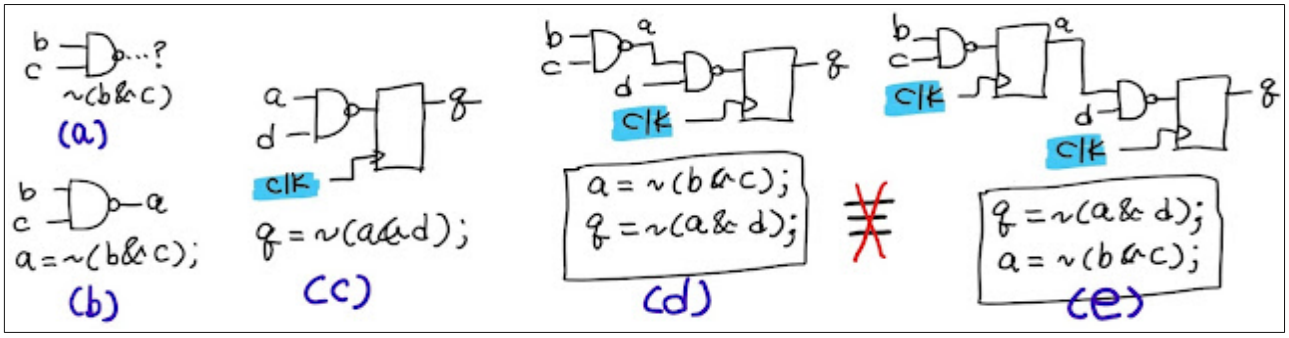
디지털 하드웨어를 레지스터 트랜스퍼 수준(Register Transfer Level, RTL)에서 기술하고 이를 자동화된 도구(합성기 또는 컴파일러)로 논리 회로 소자로 변환하는 방법론은 이미 정점에 이르렀다. RTL을 단적으로 표현하면 다음과 같다.

"비트 단위로 표현된 하드웨어 객체를 클럭의 동기에 맞춘 동작을 기술하는 수준"

C++의 템플릿 크래스로 임의 비트폭을 갖는 하드웨어 객체를 표현하는 방법은 앞선 절에서 설명하였다. 컴퓨팅 언어의 한 문장은 결국 할당 연산자 '=' 의 오른편에 놓인 수식을 평가하여 그 결과를 왼편에 놓인 객체에 전송하는 것으로 완료된다.



[주] 컴퓨터에서 한 문장이 수행되는 절차를 따져보면 메모리에서 값을 읽어 CPU에서 계산을 수행 한 후 다시 메모리에 저장한다. 메모리와 CPU는 모두 디지털 전자회로다. 단순한 문장이지만 이 표현에 많은 것들이 생략되어 있다. 컴퓨팅 언어의 문장은 매우 높은 수준의 추상적 표현으로 단지 자료가 이동하는 경로 만을 묘사했을 뿐이지 실제 전자회로가 작동하는 절차는 감춰져 있다.



높은 수준의 컴퓨팅 언어에서는 할당이 일어나는 기작(mechanism)을 따지지 않는 반면 디지털 하드웨어 언어는 이를 구분 한다. 위 그림에서 (a)는 완결된 문장이 아니다. 연산의 결과가 저장(매핑 또는 함수의 사상)되어야 문장이다. (b)와 (c)의 회로를 표현한 문장에서 할당 연산 '='의 의미는 완전히 다르다. (d)와 (e)처럼 문장의 순서에 따라 다른 회로가 될 수 있다. 병렬실행 문장은 순서가 달라도 동일한 회로이어야 한다. 할당에 대한 심도 있는 고려가 필요하다. 하드웨어를 묘사하는 언어에서 이 문제를 해결하고자 별도의 할당 연산자 <= 를 정의했지만 혼란만 더하고 있다. C++는 새로운 할당 연산자를 정의할 경우 변종의 언어가 탄생하는 결과를 초래하게 된다. 이 문제는 코딩 스타일로 해결하고 있다.

[참고] "기술문서3: 하드웨어 언어의 코딩 스타일"[바로가기]

상태 만을 표현하는 소프트웨어 언어에서 하드웨어의 사건(상승 또는 하강 엣지)을 검출하고 이를 동작에 반영하려면 별도의 실행 기작(mechanism)을 갖춰야 한다. 이를 위해 하드웨어 시뮬레이터는 문장의 실행이 문장이 놓인 순서에 따르는 절차적 운영 방식 외에 사건에 따라 지정된 함수를 호출하는 별도의 장치를 두고 있다. 이를 병렬 실행 시뮬레이션 커널(simulation kernel)이라고 한다. 하드웨어를 묘사한 문장을 실행하는 병렬 시뮬레이터는 사건 구동과 시간을 운영한다는 점을 기억해 두자. "사건과 동기"는 RTL의 기본 개념 중 하나이기도 하다.

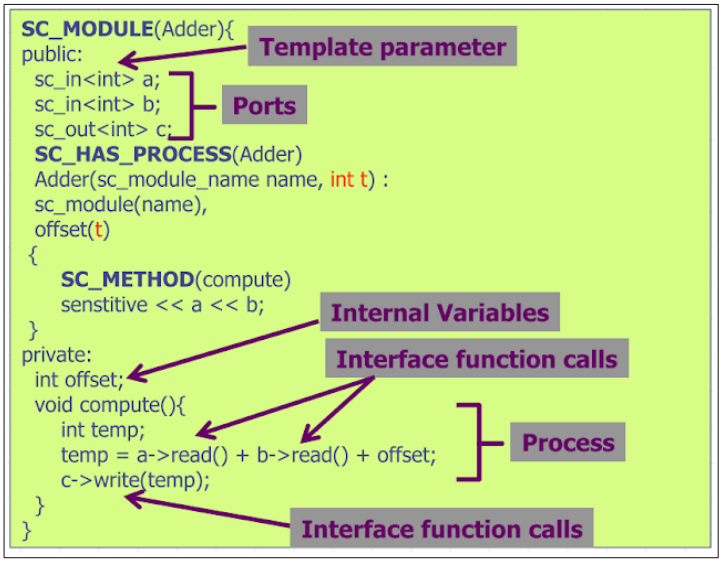
3-2. 병렬 실행 시뮬레이터의 기작

컴퓨팅 언어로 묘사한 하드웨어의 병렬성을 모의하는 방법은 의외로 단순하다. 기본 개념은 이미 소프트웨어 개발 기법으로 널리 활용되고 있는 비 선점형 다중처리(Non-preemptive multi-processing)다. 이 기법의 개념은 전통적인 마이크로 프로세서의 인터럽트 처리 기작과 흡사하다. 그래픽 사용자 인터페이스(GUI)에 활용되는 사건(버튼 또는 아이콘을 누르는 등)에 대한 대응 함수의 호출(Event & Call-Back function)과 다를 바 없다.

SystemC는 하드웨어 객체들을 묘사하기 위한 크래스들의 집합체이며 아울러 병렬 실행 시뮬레이션 커널을 가지고 있다. 시뮬레이션 커널의 기작을 정성적으로 살펴보기로 한다. SystemC를 구성하는 가장 기본 요소는 모듈과 채널이다.

a. 모듈

모듈은 설계의 기본 단위다. 외부와 통신을 위해 입출력 방향을 지정한 포트를 가지고 있으며 소속 함수를 사건에 감응 시켜(sensitize) 커널에 의해 불려 질 프로세스(sensitized member function)로 지정할 수 있다. 모듈은 계층적 구조를 취할 수 있다. 하위 모듈을 사례화 하고 채널로 모듈 사이를 연결한다. 아래 그림은 SystemC의 모듈의 구성을 보여 준다.



[그림출처] SystemC and Simulation Kernel

SystemC가 C++의 크래스 라이브러리라고 하지만 생소하다. 마치 새로운 언어 체계 처럼 보이는 것은 #define 매크로를 적극적으로 활용했기 때문이다. 복잡한 크래스 계승을 단순화 할 뿐만 아니라 하드웨어 언어와 비교하여 가독성을 높이기 위한 조치이기도 하다.

```
#include <systemc.h>
SC_MODULE(my_module) {
    // Module content (ports, signals, processes, etc.)
      SC_CTOR(my_module) {
          // Constructor code
  }
};
```

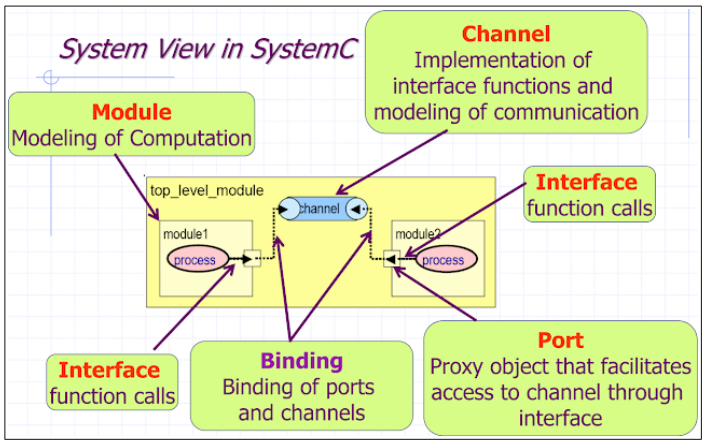
SC_MODULE() 과 SC_CTOR() 등은 모두 systemc.h 에 정의되어 있는 매크로 들이다. 풀어보면 다음과 같다.

```
struct my_module : ::sc_core::sc_module {
  // Module content
  my_module(sc_core::sc_module_name name) : sc_module(name) {
    // Constructor code
  }
};
```

하드웨어 묘사에 필요한 기초 크래스들을 계승하고 있는 것을 볼 수 있다. 모듈 뿐만 아니라 채널을 포함해 매우 다양한 속성을 갖는 하드웨어 객체들이 C++의 크래스로 기술되었다.

b. 채널

하드웨어의 동작을 기술한 두 프로세스(입출력과 동작이 포함된 함수 또는 모듈 등 뭐든 좋다.) 사이에 연결 통로가 있다고 하자. SystemC는 이 통로를 채널(channel)이라고 부른다. 이 채널은 템플릿 크래스를 활용하여 비트 단위 자료형을 담을 수 있고 사건을 감지하는 매개로서 두 저장소(새로 할당된 값 m_new_val 과 이미 가지고 있던 현재 값 m_curr_val)를 운용한다. 자료형을 다루는 소속 함수 외에 사건 감지와 보고 .notify_event() 그리고 갱신 .request_update() 등의 방법들을 가지고 시뮬레이션 커널과 유기적으로 소통한다.



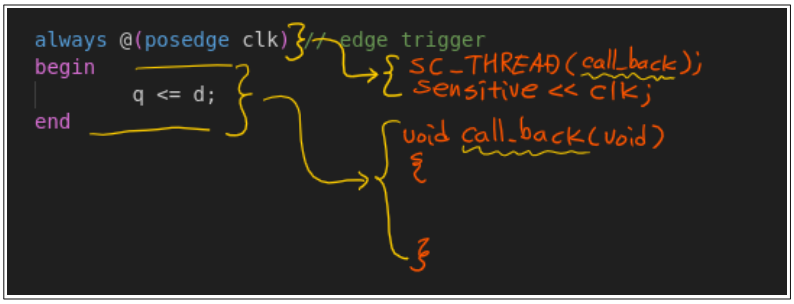
[그림출처] SystemC and Simulation Kernel

설계할 시스템의 필요에 따라 사용자 정의 채널을 만들 수도 있다. SystemC가 기본적으로 제공하는 채널로는 sc_fifo<T>, sc_signal<T>, sc_signal_resolved<>, sc_mutex<T>, sc_semapore<T> 등이 있다. 하드웨어의 묘사에 가장 기본적으로 사용되는 채널은 sc_signal<T>다. 하드웨어 기술 언어인 베릴로그의 wire 와 같다.

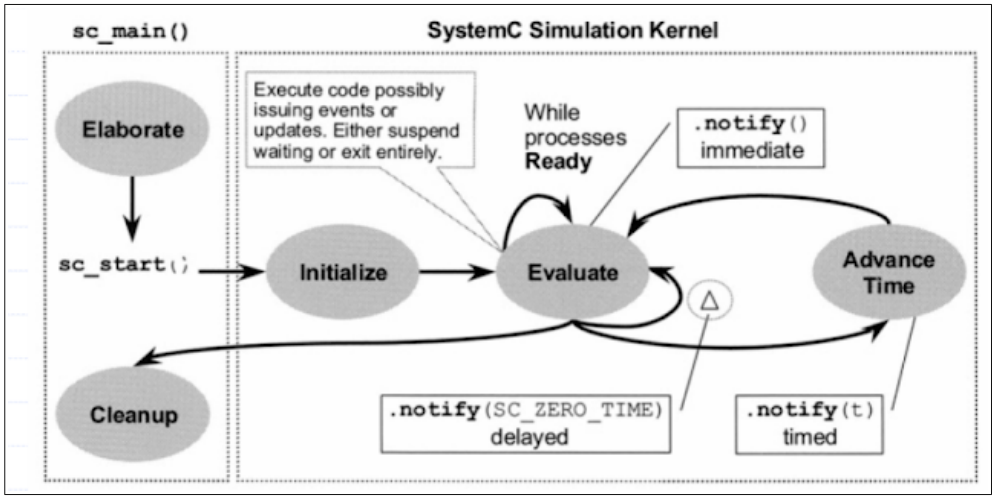
c. sc_main()

C/C++ 언어로 작성한 프로그램이 main() 으로 시작 하듯 SystemC로 작성한 하드웨어 시스템 모델의 시작은 sc_main()이다. 매우 단순한 구성으로 시험 대상(Design Under Test, DUT) 모듈을 사례화 하고 시뮬레이션 커널을 시작한다. DUT가 사례화 되면서 모듈 크래스의 구성자가 실행 되어 내부를 구성하는 절차(elaboration)를 수행한다. 이때 사건을 감시할 채널과 이에 구동될 소속함수를 지정한다.

> [주] 베릴로그 HDL은 병렬구문을 기본으로 순차구문을 always 영역에 기술 할 수 있도록 하였다. 병렬실행의 관점에서 always 영역은 병렬실행 구문 1개와 같다. 모든 구문이 순차실행인 C++에서 병렬실행의 단위는 함수다. 시뮬레이션 준비 과정에서 모듈의 소속 함수를 사건에 구동 될 함수로 지정한다. 베릴로그의 always와 그에 딸린 순차구문 영역을 C++에서 모듈의 사건과 구동 함수 연결은 아래와 같이 설명될 수 있다.



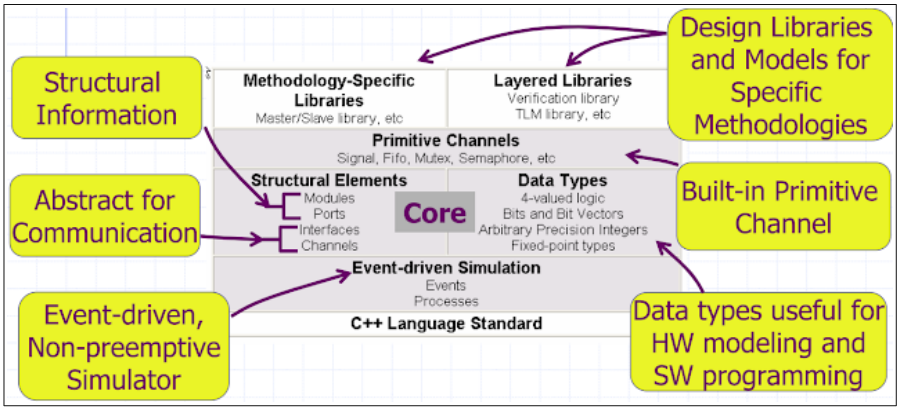
시뮬레이션이 개시되기 전 준비(elaboration)를 마치면 sc_start()를 호풀 함으로써 프로그램 실행권은 시뮬레이션 커널이 갖는다. 시뮬레이션 커널은 등록된 사건구동 함수를 모두 한번 씩 호출하여 채널에 최초 사건을 일으키도록 한다.



[출처] SystemC: From the Ground-Up

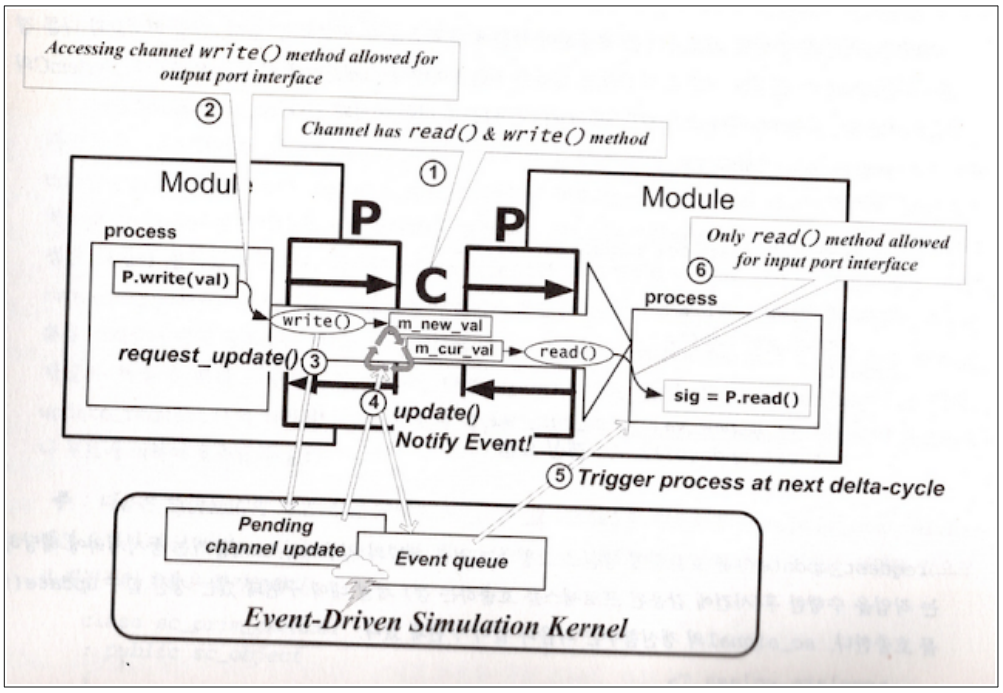
d. 병렬 시뮬레이션 커널의 스케쥴러

SystemC는 하드웨어를 묘사할 수 있도록 자료형을 갖췄을 뿐만 아니라 병렬 시뮬레이션 커널을 내장하고 있다.



[그림출처] SystemC and Simulation Kernel

SystemC의 시뮬레이션 커널은 비 선점형 다중 프로세스 처리 방식으로 프로세스 사이에 연결된 사건에 의해 구동되는 콜백 함수(Call-Back function)를 운용한다. 콜백 함수를 '프로세스(process)'라고 한다. 병렬 실행 시뮬레이션 커널의 스케쥴 절차를 도식적으로 표현하면 다음과 같다.



[출처] SystemC를 이용한 시스템 설계

병렬 실행 시뮬레이션 커널은 사건처리와 시뮬레이션 시간 진행을 관리한다. 최초(시뮬레이션 진행 시간 0)에 커널은 시뮬레이션 시간을 멈추고 프로세스들을 호출하여 채널에 쓰기 접근을 허용함으로써 스케쥴러가 개시된다.

(1) 두 프로세스 사이에 연결된 채널을 통해 통신한다. 프로세스에서 채널로 읽기 및 쓰기 접근을 위해 크래스의 소속함수 read()와 write()를 사용한다. 채널 내부에 두개의 저장소를 가지며 이를 비교하여 사건을 탐지한다.

(2) 한 프로세스에서 채널의 쓰기 방법 write()으로 새값을 써넣음으로서 사건 구동 스케쥴러가 개시된다. 이때 써넣게 되는 값은 m_new_val에 저장된다.

(3) 채널 내에서 두 저장소 m_new_val 과 m_cur_val이 다를경우 시뮬레이터에 사건이 발생 했음을 공지한다.

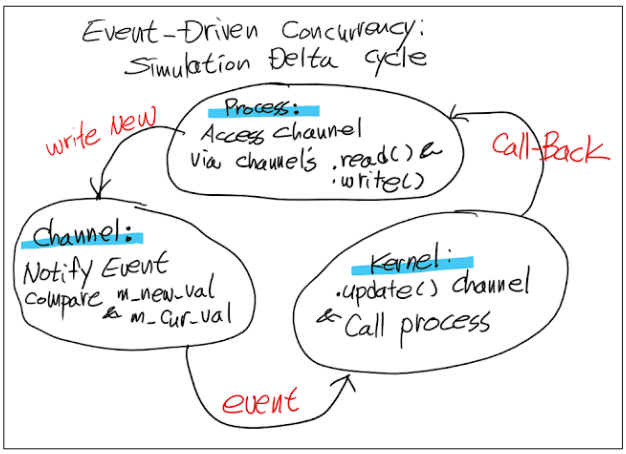
사건 공지 후 시뮬레이터의 실행 제어는 커널로 옮겨 간다.

(4) 시뮬레이터는 채널로부터 공지받은 사건들을 모두 수집하여 해당 사건에 감응이 지정된 프로세스를 호출한다.

(5) 프로세스를 호출하기 전에 채널의 새값을 현재 값으로 갱신한다.

(6) 사건에 감응되어 호출된 프로세스는 채널로 읽기 접근 접근한다. 이때 읽히는 값은 갱신된 값 m_cur_val 이다.

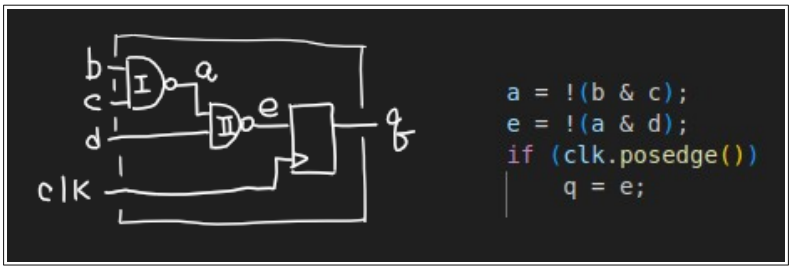
채널과 사건처리 커널은 실행을 선점하지 않고 서로 협조 관계(cooperative)에 있다. (1)~(3)의 과정은 채널에서 수행되고 (4)~(5)는 커널의 역활이며 (6)은 프로세스의 수행이다. 어느 프로세스에서 채널로 쓰기 접근함에 따라 사건이 발생하고 이에 감응이 지정된 함수를 호출하는 일련의 과정을 반복한다. 시뮬레이션 시간을 멈춘 상태에서 채널에서 발생한 사건을 모두 처리하는 동안을 "시뮬레이션 델타"라 한다. 모든 사건이 완료되면 스케쥴러는 시뮬레이션 시간을 진행한다. 시뮬레이션 델타가 완료된 후 다음 사건이 발생하지 않으면 시뮬레이션은 중지된다. 아래 그림은 시뮬레이션 델타 내에서 채널과 프로세스, 그리고 커널이 반복(Simulation Delta Cycle)되는 모습을 나타냈다. 하드웨어 시뮬레이터는 채널과 프로세스 그리고 커널 모두 실행 우선권이 없이(Non-Preemptive) 서로 협동하는(Cooperative) 다중 처리(multi-processing) 체계다. 실시간 운영체제(Real-Time Operating System, RTOS)의 멀티 태스킹(Multi-Tasking)과 다르다.



하드웨어 시뮬레이터는 협동형 비 선점 멀티 프로세싱이다.

3-3. 병렬실행 예제

예제를 통해 SystemC의 병렬 실행 시뮬레이션 커널의 작동 메커니즘을 정성적으로 이해해 보기로 한다. 다음과 같은 회로를 3개의 할당문으로 표현하였다. 하드웨어(논리 게이트 회로)를 표현한 할당문은 각각 병렬 실행 되어야 한다.



예제 디렉토리로 이동,

```
    $ cd ~/ETRI050_DesignKit/devel/Tutorials/2-
2_Verilog_SystemC_in_a_Day/ex2_delta_cycle
    $ ll
    total 28
    drwxrwxr-x 2 goodkook goodkook 4096 Jul 12 13:54 ./
    drwxrwxr-x 8 goodkook goodkook 4096 Jul 11 23:27 ../
    -rw-rw-r-- 1 goodkook goodkook 1274 Jul 12 13:15 Makefile
    -rw-rw-r-- 1 goodkook goodkook  505 Jul 11 23:10 sc_main.cpp
    -rw-rw-r-- 1 goodkook goodkook 1327 Jul 12 13:39 sc_delta_cycle.h
    -rw-rw-r-- 1 goodkook goodkook  723 Jul 11 23:33 sc_delta_cycle_TB.gtkw
    -rw-rw-r-- 1 goodkook goodkook 2535 Jul 11 23:24 sc_delta_cycle_TB.h
```

시험 대상(DUT)을 기술한 파일 sc_delta_cycle.h을 보면 다음과 같다. 회로는 3개의 문장을 사용하여 기술되었다. 사건 구동 프로세스 behavior 내에 할당 연산자의 오른편에 놓인 채널 sc_signal 들을 모두 감응에 지정하였다. C++의 문법으로 하드웨어를 기술 할 경우 할당 연산자 '='가 하드웨어의 추상성(조합회로 또는 순차회로)을 구분하지 못한다. C++는 절차적 언어로서 병렬실행 문장은 없으므로 하드웨어의 병렬성을 반영하지 못한다. 하드웨어 시뮬레이터는 이를 극복할 방법으로 협력형 병렬실행 커널(Cooperative multi-processing kernel)을 도입하였다. 병렬 실행은 사건 구동 프로세스(함수 또는 시그널 채널로 할당하는 문장) 단위로 이뤄지며 할당의 구분은 코딩 스타일에 의해 결정된다.

```
    #include <systemc.h>

    SC_MODULE(sc_delta_cycle)
    {
        // IO Ports
        sc_in<bool>     clk, b, c, d;
        sc_out<bool>    q;
        sc_signal<bool> a, e;     // Local Channels
        SC_CTOR(sc_delta_cycle):  // constructor
            clk("clk"), d("d"), q("q")
        {
```

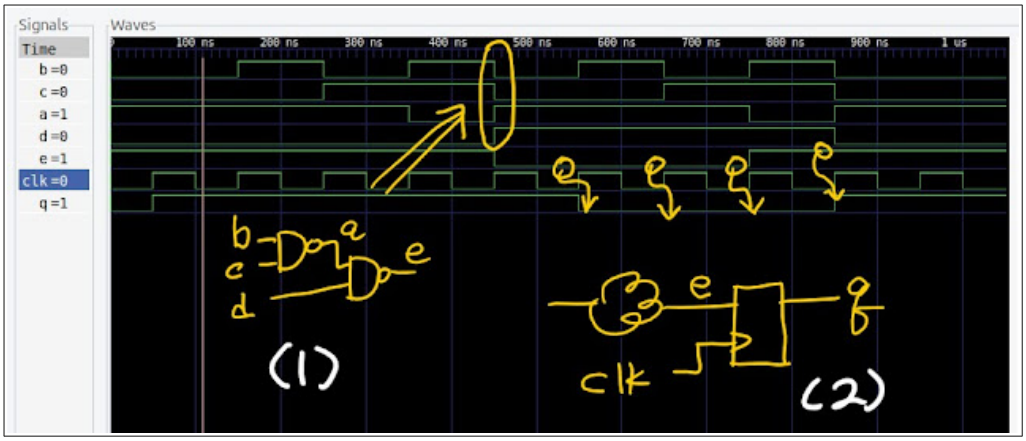
```
        SC_METHOD(behavior);
        sensitive << clk << a << b << c << d;  // exclude 'e' ?
    }
    void behavior(void)
    {
        printf("\n[%03d] clk=%c b=%c c=%c a=%c d=%c e=%c q=%c",
            (int)(sc_time_stamp()).to_double()/1000,
            clk.read()? '1':'0',
            b.read()? '1':'0',
            c.read()? '1':'0',
            a.read()? '1':'0',
            d.read()? '1':'0',
            e.read()? '1':'0',
            q.read()? '1':'0'
        );
        a = !(b & c);
        e = !(a & d);
        if (clk)
            q = e;
    }
};
```

예제 디렉토리에 준비된 Makefile로 시뮬레이터를 빌드하고 실행시켜보자.

```
$ make run
clang++ -I. -I../c_untimed -I/opt/systemc/include -g  -L/opt/systemc/lib \
  -o sc_test_d_TB -lsystemc sc_main.cpp
./sc_test_d_TB
    SystemC 3.0.2-Accellera --- Jun 13 2025 17:49:45
    Copyright (c) 1996-2025 by all Contributors,
    ALL RIGHTS RESERVED
[000] clk=1 b=0 c=0 a=0 d=0 e=0 q=0
[000] clk=0 b=0 c=0 a=1 d=0 e=1 q=0
      ......
[300] clk=0 b=0 c=1 a=1 d=0 e=1 q=1
[350] clk=1 b=0 c=1 a=1 d=0 e=1 q=1
[350] clk=1 b=1 c=1 a=1 d=0 e=1 q=1
[350] clk=1 b=1 c=1 a=0 d=0 e=1 q=1
[400] clk=0 b=1 c=1 a=0 d=0 e=1 q=1
[450] clk=1 b=1 c=1 a=0 d=0 e=1 q=1
[450] clk=1 b=0 c=0 a=0 d=1 e=1 q=1
[450] clk=1 b=0 c=0 a=1 d=1 e=1 q=1
[500] clk=0 b=0 c=0 a=1 d=1 e=0 q=1
[550] clk=1 b=0 c=0 a=1 d=1 e=0 q=1
[550] clk=1 b=1 c=0 a=1 d=1 e=0 q=0
......
Info: /OSCI/SystemC: Simulation stopped by user.
```

디지털 파형을 보면 다음과 같다.

```
$ make wave
```



동일한 할당 연산이지만 행위 기술방식(코딩 스타일)에 따라 즉시할당 또는 지연할당이 된다. 그림에서 입출력 파형으로 즉시할당과 지연 할당이 일어나는 모습을 볼 수 있다.

(1) 즉시할당: 논리게이트 입력측의 모든 채널 a, b, c, d 이 사건 감응 되었다. 채널에 발생한 사건은 즉시 출력 채널에 반영된다.

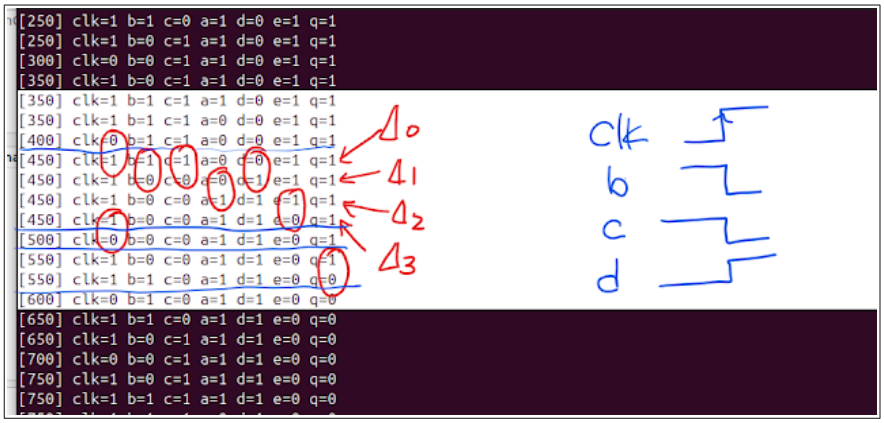
(2) 지연할당: e 에서 q 로 전송에 clk 만 감응에 참여 한다. 채널 e 의 현재 값이 출력 q 에 반영되는 시점은 다음 clk의 사건 처리 델타에서 반영 된다. 이는 플립플롭의 행동을 묘사한 것이다. 지연할당을 의미하는 '전송'은 즉시할당과 다르다.

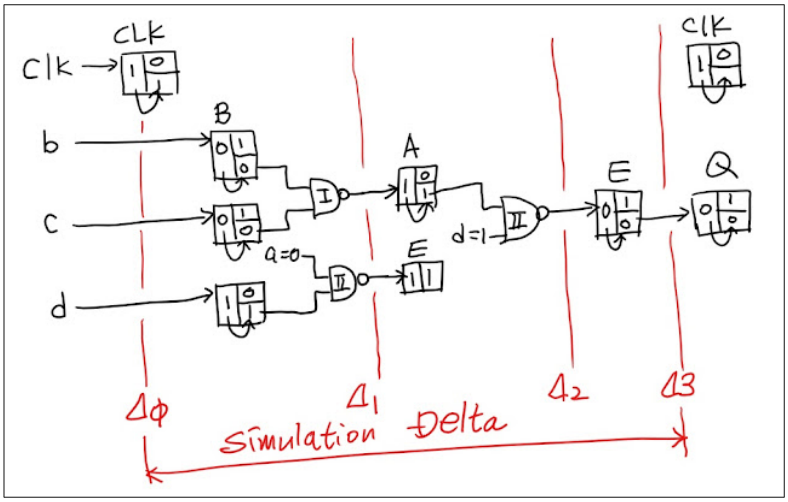
프로세스 behavior() 내 할당문의 순서를 바꿔서 시뮬레이션 해보자. 결과는 달라지지 않는다. 이는 세 문장이 병렬실행 되고 있음을 뜻한다. SystemC의 시뮬레이터의 기작으로 병렬실행을 설명해 보기로 한다. 아래 그림은 프로세스에서 채널로 접근과 채널 내에서 사건의 감지 그리고 커널에 의한 프로세스의 실행이 반복되는 과정(시뮬레이션 델타)을 설명한다.

시뮬레이션은 테스트벤치 sc_delta_cycle_TB.h 에서 clk의 상승 엣지를 기다려 채널 b, c, d에 새값 써넣기로 시작된다. 테스트 벤치의 프로세스에서 채널 b, c, d에 새 값을 써넣었다.

```
wait(clk.posedge_event());
b.write(0);
c.write(0);
d.write(1);
```

시뮬레이션 결과를 보여주는 위의 파형에서 시간 450ns 지점을 살펴본다. 채널 clk에 사건이 발생하였으므로 시뮬레이션 델타가 시작된다.





델타0:

테스트 벤치에서 채널 clk 에 값을 씀으로써 사건이 발생했다. 시뮬레이션 커널은 시간을 멈추고 시뮬레이션 델타 싸이클을 개시한다.

델타1:

채널: b, c, d 채널의 현재값과 새값이 다르므로 사건발생 공지

커널: b, c, d 채널 값 갱신 후 사건에 감응되도록 지정된 프로세스 호출

(모두 동일한 프로세스에 감응 되어 있으므로 세 채널 모두 프로세스 호출 전 갱신)

프로세스: 할당문 실행으로 내부 채널 a, e 에 새값 써넣기

델타2:

채널: 채널 a 는 현재값과 새값이 다르므로 사건 발생 공지

(채널 e 는 현재값과 새값이 동일 하므로 사건 없음)

커널: 채널 a 의 사건에 감응되도록 지정된 프로세스 호출

프로세스: 할당문 실행으로 채널 e 에 새값 써넣기

델타3:

채널 e: 현재값과 새값이 다르므로 사건 발생 공지 후 갱신

(채널 a는 현재값과 새값이 동일 하므로 사건 없음)

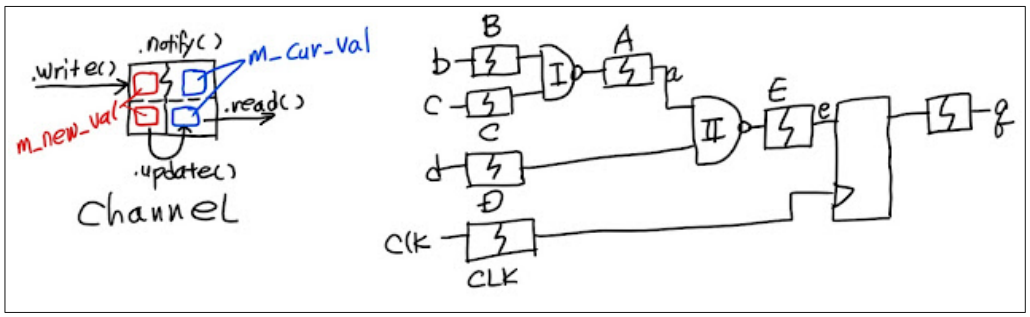
커널: e 채널 갱신 후 사건에 감응되도록 지정된 프로세스 호출

프로세스: 할당문 실행으로 채널 e 에 새값 써넣기

(q 가 갱신 되려면 clk의 상승 엣지 사건의 조건에 맞지 않으므로 q 값은 갱신되지 않음)

시뮬레이션 델타 완료:

시뮬레이션 델타 싸이클은 채널 a, b, c, d , e 에 더이상 사건이 탐지되지 않으면 종료 후 시간 진행



3-4. 감응 지정이 논리회로의 행동에 미치는 영향

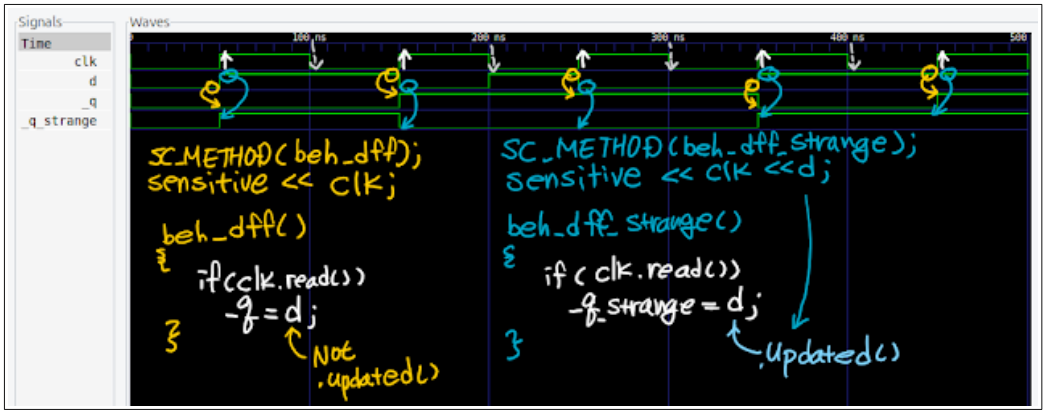
프로세스가 구동되는 기작의 원인은 감응으로 지정된 채널에 발생한 사건이다. 이때 프로세스를 호출하기 전 갱신은 감응으로 지정된 채널에 한한다는 점에 유의해야 한다. 위의 예에서 NAND 게이트의 출력은 현재 값이 할당되면 즉시 반영된다. 이는 조합 논리회로의 동작이다. 플립플롭의 경우 클럭의 상승 엣지에 맞춰 입력이 출력으로 전송 된다. 이는 순차 논리회로의 동작이다. 이 둘을 구분하는 방법은 감응의 지정 여부에 달렸다.

아래의 예를 보자. 프로세스 beh_dff()는 clk 채널에 감응이 지정되었지만 beh_dff_strange()는 clk와 d 채널 모두에 감응이 지정되었다. 시뮬레이션 결과 관찰을 VCD 파형보기 외에 프로세스가 구동될 때마다 채널의 현재값 변화를 읽어 출력하였다.

```
// Filename: ex3_dff_strange/sc_dff_strange/Vdff.h

#include <systemc.h>
SC_MODULE(Vdff)
{
    sc_in<bool>     clk, d;
    sc_out<bool>    q;
    sc_signal<bool> _q, _q_strange;
    sc_trace_file* fp;  // VCD file
    SC_CTOR(Vdff):    // constructor
        clk("clk"), d("d"), q("q")
    {
        SC_METHOD(beh_dff);
        sensitive << clk;

        SC_METHOD(beh_dff_strange);
        sensitive << clk << d;

        SC_METHOD(beh_output);
        sensitive << _q;

        // VCD Trace
        fp = sc_create_vcd_trace_file("Vdff");
```

```
        sc_trace(fp, clk, "clk");
        sc_trace(fp, d, "d");
        sc_trace(fp, q, "q");
        sc_trace(fp, _q, "_q");
        sc_trace(fp, _q_strange, "_q_strange");
    }
    void beh_dff()
    {
        printf("\n[%03d] beh_dff         : clk=%c d=%c",
            (int)(sc_time_stamp()).to_double()/1000,
            clk.read()? '1':'0', d.read()? '1':'0');

        if (clk.read())
            _q.write(d);
    }
    void beh_dff_strange()
    {
        printf("\n[%03d] beh_dff_strange: clk=%c d=%c",
            (int)(sc_time_stamp()).to_double()/1000,
            clk.read()? '1':'0', d.read()? '1':'0');

        if (clk.read())
            _q_strange.write(d);
    }
    void beh_output()
    {
        q.write(_q);
    }
};
```

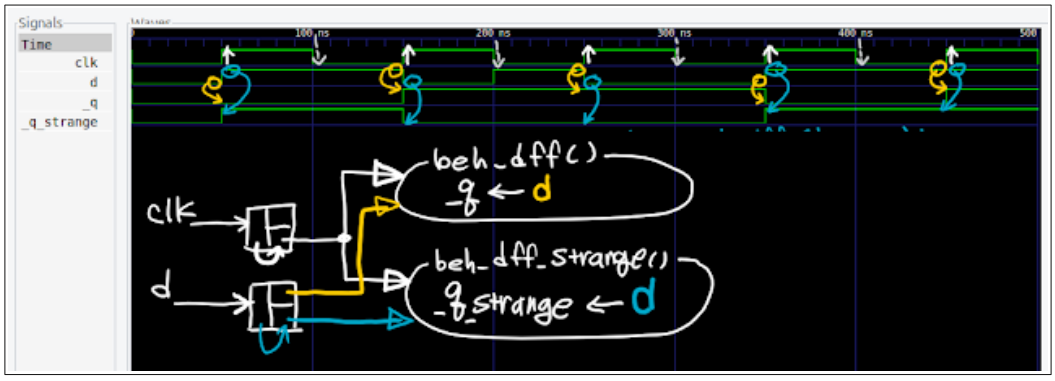
시뮬레이션을 수행해보면 다음과 같은 동작을 보여준다.



채널 clk와 d 에 사건이 동시에 발생했지만 프로세스가 호출되는 델타 시점이 다르다.

(1) 프로세스 beh_dff()를 호출 할 때 감응에 지정되지 않은 채널 d는 갱신되지 않는다. 따라서 출력 _q는 이전 값을 갖는다. 이는 플립플롭의 행동(클럭 동기에 맞춘 전송)을 반영한다.

(2) 프로세스 beh_dff_strange()는 clk는 물론 d 에도 감응 되었다. 체널 clk와 d 모두 새 값으로 갱신된 후 프로세스가 호출 된다. 새 값이 즉시 출력 _q_strange 에 반영 된다. 이는 조합 논리회로의 행동을 반영한다.



감응 목록에 넣는 것 만으로도 하드웨어의 행동에 영향을 줄 수 있다. 하드웨어 기술 언어의 지침에 코딩 스타일에 주의를 기울여야 한다고 강조하고 있는 이유다. 디지털 조합 회로를 기술하는 경우 입력 신호들을 모두 감응 목록에 포함시켜야 한다. 불완전한 감응 목록은 의도치 않은 결과를 낳을 수 있다. 다음은 멀티플렉서를 기술한 예다.

```
always @ (a, b)
begin
    if (sel)     // 'sel' must be added into sensitivity list
        y <= a;
    else
        y <= b;
end
```

선택을 제어하는 sel 이 감응 목록에 빠져 있다. 이로 인한 행동의 오류를 사건 구동 시뮬레이터의 기작으로 설명해 보라. 불완전한 감응 목록의 실수를 방지하고자 와일드 문자 '*' 를 사용하기도 한다.

```
always @ (*)
begin

   ......
end
```

불완전한 감응 목록은 베릴로그 언어에서 가장 흔히 저지르는 실수다. 베릴로그 언어가 SystemVerilog 로 개정되면서 이에 대한 대비로 always 외에 always_comb 와 always_latch, always_ff 가 추가 되었다. 행동을 기술한 내용이 설계자의 의도와 맞지 않을 경우 빌드(컴파일) 오류 메시지를 내보낸다. 일례로, 다음과 같이 always_comb로 조합 회로를 기술 한다고 명시 했지만 기술된 내용은 else가 빠져 래치가 되었다.

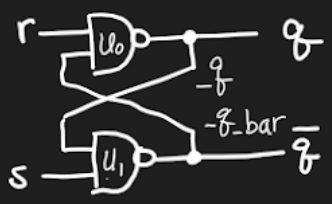
```
always_comb
begin
    if (sel)
        y <= a;
end
```

3-5. 재귀적 사건 반복이 금지된 이유

사건은 프로세스에서 체널에 새 값을 써넣어 발생한다. 만일 한 프로세스가 자신을 구동하는 채널에 새 값을 써넣을 경우 사건이 재귀적으로 반복되어 시뮬레이션 델타가 무한히 증가하게 만들 수 있다. 결국 시뮬레이터 시간 진행되지 못하게 된다. 디지털 정보를 저장하는 래치는 디지털 게이트를 재귀적으로 연결시킨 구조로 만든다. 전류가 게이트를 통과할 때 트랜지스터 내부의 저항과 컨덴서 성분이 작용하여 발생 하는 약간의 지연을 교묘히 이용한다. 전압의 유무로 정보를 취급하는 사건 구동 시뮬레이터는 이와 같은 전류-전압의 미세한 시간상 변화를 다룰 수 없다. 디지털 시뮬레이터는 전자회로를 매우 높은 추상화 수준에서 묘사한다는 점을 기억해 두자.

반복적인 사건이 반복되어 시뮬레이션 델타를 종료하지 못하는 경우의 예를 보기로 한다. 하드웨어 언어로 NAND 게이트를 상호 연결하여 RS-래치 회로를 어렵지 않게 묘사할 수 있다.

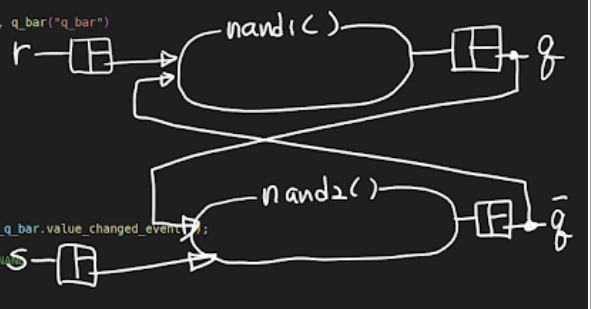
```
/* verilator lint_off UNOPTFLAT */

module rsff(r, s, q, q_bar);
input   r, s;
output  q, q_bar;

wire _q, _q_bar;

assign q = _q;
assign q_bar = _q_bar;

// ILLEGAL: Circular combinational logic generate Recursive events
assign _q = ~(r & _q_bar);
assign _q_bar = ~(s & _q);

endmodule
```



rsff.v: 게이트 수준 RS 래치의 베릴로그 모델

물론 SystemC 로도 묘사할 수 있다.

```
SC_MODULE(Vrsff)
{
    sc_in<bool>     r, s;
    sc_inout<bool>    q, q_bar;

    SC_CTOR(Vrsff) : r("r"), s("s"), q("q"), q_bar("q_bar")
    {
        SC_THREAD(nand1);
        sensitive << r << q_bar;

        SC_THREAD(nand2);
        sensitive << s << q;
    }

    void nand1(void)
    {
        while(true)
        {
            wait(r.value_changed_event() | _q_bar.value_changed_event());
            wait(SC_ZERO_TIME);
            _q.write(!(r & _q_bar));    // NAND
        }
    }

    void nand2(void)
    {
        while(true)
        {
            wait(s.value_changed_event() | _q.value_changed_event());
            wait(SC_ZERO_TIME);

            _q_bar.write(!(s & _q));    // NAND
        }
    }
};
```



Vrsff.v: 게이트 수준 RS 래치의 SystemC 모델

두 NAND 게이트의 입출력이 서로 연결 되어 있다. 두 프로세스 사이에 사건이 재귀적으로 반복되는 상황이 벌어진다. 디자인 킷의 예제를 이를 사건 구동 시뮬레이터의 기작으로 따져보자. 예제 디렉토리로 이동,

```
      $ cd ~/ETRI050_DesignKit/devel/Tutorials/2-2_Verilog_SystemC_in_a_Day/
ex4_rsff_recursive_event/sc_rsff_gates
```

```
      $ ll
      total 260
      drwxrwxr-x 2 goodkook goodkook   4096 Jul 13 20:02 ./
      drwxrwxr-x 5 goodkook goodkook   4096 Jul  8 14:36 ../
      -rw-rw-r-- 1 goodkook goodkook   1258 Jul 13 19:21 Makefile
      -rw-rw-r-- 1 goodkook goodkook    572 Jul 13 19:36 sc_main.cpp
      -rw-rw-r-- 1 goodkook goodkook   1939 Jul 13 16:05 sc_rsff_TB.h
      -rw-rw-r-- 1 goodkook goodkook   1898 Jul 13 20:01 Vrsff.h
```

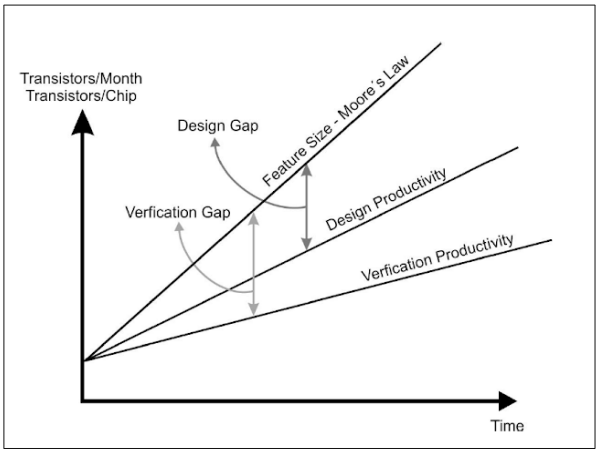
준비된 Makefile을 이용하여 시뮬레이터 빌드 및 실행,

```
    $ DELAY=ZERO_TIME make build
    $ make run
    ./sc_rsff_TB
        SystemC 3.0.2-Accellera --- Jun 13 2025 17:49:45
        Copyright (c) 1996-2025 by all Contributors,
        ALL RIGHTS RESERVED
Info: (I702) default timescale unit used for tracing: 1 ps (sc_rsff_TB.vcd)
[Time:010ns][Delta:003] NAND2(s=1, q    =0) -> q_bar=[new=1|curr=0]
[Time:010ns][Delta:005] NAND1(r=0, q_bar=1) -> q    =[new=1|curr=0]
......
[Time:060ns][Delta:032] NAND2(s=0, q    =1) -> q_bar=[new=1|curr=0]
[Time:060ns][Delta:032] NAND1(r=0, q_bar=0) -> q    =[new=1|curr=1]
[Time:060ns][Delta:034] NAND1(r=0, q_bar=1) -> q    =[new=1|curr=1]
[Time:070ns][Delta:037] NAND2(s=1, q    =1) -> q_bar=[new=0|curr=1]
[Time:070ns][Delta:037] NAND1(r=1, q_bar=1) -> q    =[new=0|curr=1]
[Time:070ns][Delta:039] NAND2(s=1, q    =0) -> q_bar=[new=1|curr=0]
[Time:070ns][Delta:039] NAND1(r=1, q_bar=0) -> q    =[new=1|curr=0]
[Time:070ns][Delta:041] NAND2(s=1, q    =1) -> q_bar=[new=0|curr=1]
[Time:070ns][Delta:041] NAND1(r=1, q_bar=1) -> q    =[new=0|curr=1]
[Time:070ns][Delta:043] NAND2(s=1, q    =0) -> q_bar=[new=1|curr=0]
[Time:070ns][Delta:043] NAND1(r=1, q_bar=0) -> q    =[new=1|curr=0]
[Time:070ns][Delta:045] NAND2(s=1, q    =1) -> q_bar=[new=0|curr=1]
[Time:070ns][Delta:045] NAND1(r=1, q_bar=1) -> q    =[new=0|curr=1]
[Time:070ns][Delta:047] NAND2(s=1, q    =0) -> q_bar=[new=1|curr=0]
[Time:070ns][Delta:047] NAND1(r=1, q_bar=0) -> q    =[new=1|curr=0]
......
```

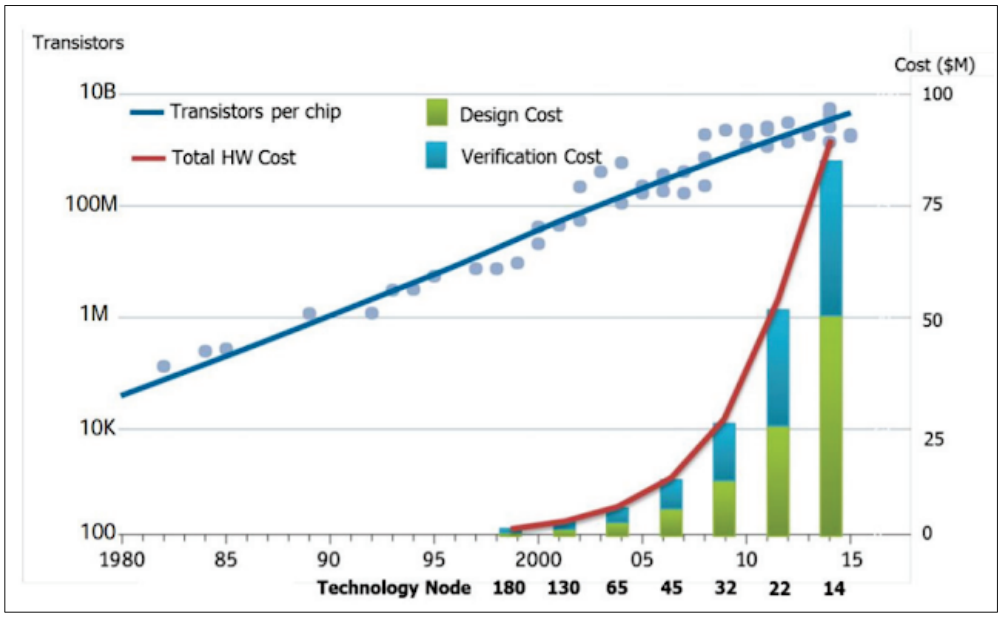
시간 60ns 에서 각각 0 이었던 r 과 s 입력을 70n 에서 모두 1로 변경하자 시뮬레이션 시간을 중지하고 사건 처리 시뮬레이션 델타가 시작된다. 두 NAND 게이트의 출력 q 와 q_bar의 새값이 현재 값과 다르므로 사건이 다시 게이트 입력으로 들어간다. RS-래치의 입력 r 과 s 는 변함이 없지만 q 와 q_bar의 변화는 재귀적으로 반복되므로 시뮬레이션 델타가 무한히 늘어나 교착 상태에 빠질 위험이 있다.

4. 맺음말

컴퓨팅 언어를 활용할 수 있게 되면서 반도체 설계의 생산성이 극적으로 향상되었다. 특히 크래스 개념은 전자회로의 묘사를 최고의 추상화 수준에서 가능하게 한다. 그럼에도 설계 생산성은 여전히 반도체 제조 집적도 기술을 따라잡지 못하고 있고 검증은 그보다 훨씬 미치지 못하고 있다. 이 격차의 극복 방안 역시 높은 추상성 추구다. 디지털 반도체 설계의 검증 방법에서 RTL 시뮬레이션이 차지하는 비중이 여전히 높지만 SystemC/C++를 활용한 시스템 수준 설계 방법론이 활발하여 상당한 성과를 거두고 있다. 아쉽게도 하드웨어 설계자에게 C++ 언어는 높은 장벽이 되고 있다. 이 글은 C++ 언어의 근간인 크래스와 시뮬레이션 커널의 기초를 정성적으로 설명하였다. 높은 추상화 수준에서 반도체를 설계하고 검증을 위한 시뮬레이션 환경 구죽의 입문에 도움이 되길 바란다.



설계 생산성은 무어의 법칙을 따라잡지 못하고 있다. 검증의 생산성은 이보다 더 쳐진다.



SoC 의 비용은 설계보다 검증이 차지하는 비중이 높다.
[출처] OpenROAD: Toward a Self-Driving, Open-Source Digital Layout Implementation Tool Chain

---

참고:

[1] SystemC Quick Reference card, http://www.eis.cs.tu-bs.de/klingauf/systemc/systemc_quickreference.pdf

[2] Verilog Quick Reference, https://web.stanford.edu/class/ee183/handouts_win2003/VerilogQuickRef.pdf

[3] C++ Quick Reference, https://www.hoomanb.com/cs/quickref/CppQuickRef.pdf

[4] RTL Coding Styles That Yield Simulation and Synthesis Mismatches, http://www.sunburst-design.com/papers/CummingsSNUG1999SJ_SynthMismatch.pdf

[5] IEEE 1364.1-2002 - IEEE Standard for Verilog Register Transfer Level Synthesis, https://ieeexplore.ieee.org/document/1146718

[6] IEEE Standard for Verilog Hardware Description Language, https://www.eg.bucknell.edu/~csci320/2016-fall/wp-content/uploads/2015/08/verilog-std-1364-2005.pdf

[7] FORTE Design, Learn SystemC, https://www.youtube.com/playlist?list=PLcvQHr8v8MQLj9tCYyOw44X1PLisEsX-J