ETRI 0.5um CMOS Std-Cell DK:

하드웨어 기술 언어의 코딩 스타일

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 4 일

ETRI 한국전자통신연구원
www.etri.re.kr

하드웨어 기술 언어의 코딩 스타일

목차:

---

1. 개요

Verilog로 설계하고 SystemC로 검증 하는 설계 방법론을 소개한다. 설계의 예는 디지털 회로의 가장 단순한 D-플립플롭이다. D-플립플롭은 이미 이전 강좌에서 트랜지스터 회로 수준에서 설계 했었다[바로가기]. 디지털 저장장치(메모리 또는 플립플롭)를 일일이 트랜지스터의 지연된 스위칭 동작으로 구현했던 것과 비교하면 높은 추상화 수준의 설계 방법론이 생산적일 수 있다는 점을 알게될 것이다. 하드웨어의 행동을 언어로 표현 하므로서 얻는 장점을 확장하여 Verilog 설계와 SystemC 검증을 통하여 "시스템 수준"이라는 새로운 관점에서 살펴본다.

Verilog와 SystemC의 기본 구성요건과 시뮬레이터를 구성하는 과정을 소개하고 소프트웨어로 하드웨어의 병렬실행을 모의하는 방법을 살펴볼 것이다. 설계자로서 도구의 사용자 이지만 도구가 작동하는 원리를 이해하면 학습진도를 가속화 할 수 있을 뿐만 아니라 높은 추상화 수준에서 시스템 모형화(system modeling)를 시작할 때 기초가 될 것이다. 본 문서에서 설명하는 예제의 파일들은 디자인 킷[바로가기]의 튜토리얼 중 아래 디렉토리에서 찾아볼 수 있다.

```
~$ cd ~/ETRI050_DesignKit/Tutorials/2-3_Lab1_dff

~/ETRI050_DesignKit/devel/Tutorials/2-3_Lab1_dff$ tree
.
├── dff : D-플립플롭의 행위기술
│   ├── dff_Config.txt
│   ├── dff.v
│   ├── Makefile
│   ├── sc_dff_TB.gtkw
│   ├── sc_dff_TB.h
│   ├── sc_main.cpp
│   └── Vdff.gtkw
├── dff_gate : D-플립플롭의 게이트 기술 및 테스트벤치 재사용
│   ├── dff.v
│   ├── Makefile
│   ├── sc_dff_TB.gtkw
│   └── Vdff.gtkw
├── dffrs : 비동기 셋과 리셋을 가진 D-플립플롭
│   ├── dffrs.v
│   ├── Makefile
│   ├── sc_dffrs_TB.gtkw
│   ├── sc_dffrs_TB.h
│   ├── sc_main.cpp
│   └── Vdffrs.gtkw
└── shifter : 코딩 생산성 향상 팁: FOR 반복문과 `define 매크로
    ├── Makefile
    ├── sc_main.cpp
    ├── sc_shifter_TB.gtkw
    ├── sc_shifter_TB.h
    ├── shifter.v
    └── Vshifter.gtkw
```

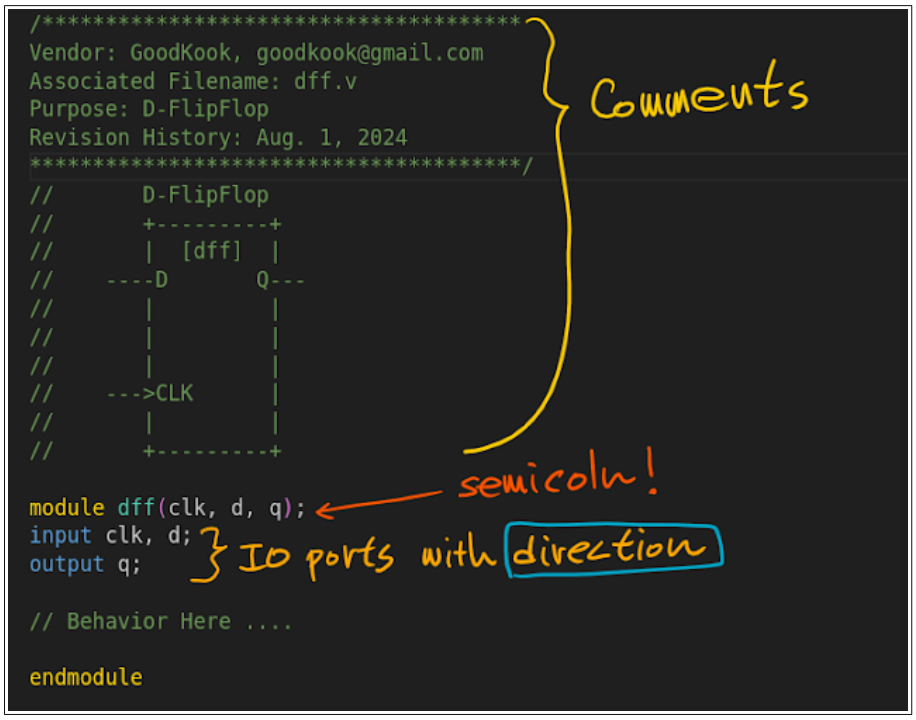
2. 베릴로그로 작성하는 D-플립플롭

Verilog는 하드웨어의 행동을 묘사하기 위해 등장한 컴퓨팅 언어다. 하드웨어를 다루는 만큼 병렬실행(concurrency) 구문을 기본으로 행위의 기술을 위해 순차실행(procedural)을 수용한다. 베릴로그 시뮬레이터는

하드웨어의 병렬성을 흉내내기 위해 사건 구동(event-driven)과 지연 할당(deferred assign) 기법을 활용한 소프트웨어다. Verilog 언어에서 이 기법들이 어떻게 동원되는지 살펴본다.

2-1. 모듈

베릴로그의 설계(기술)단위는 모듈(module)이다. 이름이 "dff"인 모듈의 외형(boundary)을 기술하면 아래와 같다.




```
/*************************************
Vendor: GoodKook, goodkook@gmail.com
Associated Filename: dff.v
Purpose: D-FlipFlop
Revision History: Aug. 1, 2024
*************************************/
//       D-FlipFlop
//       +---------+
//       |  [dff]  |
//    ----D       Q---
//       |         |
//       |         |
//       |         |
//    --->CLK      |
//       |         |
//       +---------+

module dff(clk, d, q);
input clk, d;
output q;

// Behavior Here ....

endmodule
```

- 모듈을 정의하는 베릴로그의 키워드는 **module** 이다. 모듈 정의는 세미콜론(;)으로 끝난다.
- 모듈 기술의 끝은 **endmodule** 이다.

> [주] 세미콜론(;)은 문장의 끝을 표시하는 마침 부호다. C 언어와 같다. 모듈의 끝을 표시하는 **endmodule** 은 문장의 끝이 아니므로 문장 마침표가 없다. C 언어에서 제어 영역을 묶기 위해 중괄호 {...}를 사용 하듯 베릴로그는 예약어 **begin ... end** 를 사용한다.

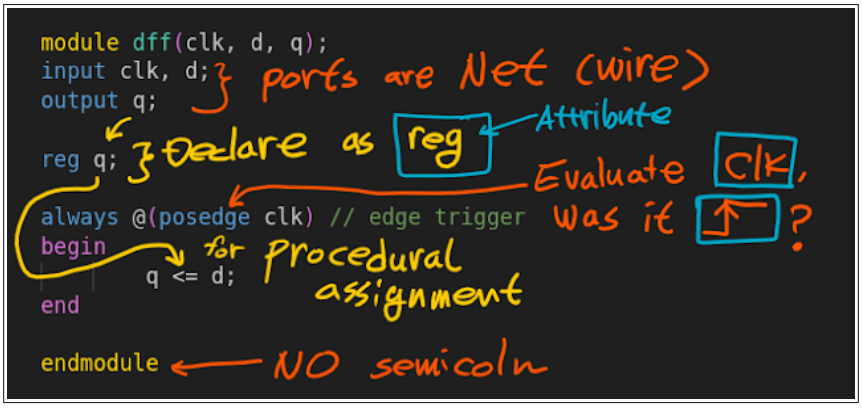
- 모듈 dff 는 입출력 포트로 clk, d, q를 가지고 있다.
- 포트(port)는 입출력 방향(direction)을 명시해야 한다. 방향을 표시하는 예약어는 input, output, inout으로 3종류다.

> [주] 하드웨어 언어는 전자회로를 기술하고 있다는 점을 항상 기억해 두자. 전류의 입력(current source)과 출력(current sink)을 반드시 구분해 주어야 한다. 입력 포트는 할당(=)의 오른편 rhs(right-hand side)에, 출력 포트는 왼편 lhs(left-hand side)에 만 놓일 수 있다.

- 주석문(comments)은 C 언어의 것과 동일하다.

2-2. D-플립플롭의 행위 기술

D-플립플롭의 행동(behavior)는 클럭으로 지정된 신호의 에지 사건(edge event)에 의해서 만 반응한다. 아래의 예는 clk의 상승 엣지에 반응하는 플립플롭을 기술했다.



- 순차적 행동의 묘사(procedural behavior description)는 always 구역 내에서 기술된다.
- 순차구문의 할당연산(<=) 왼편에 놓일 신호는 reg 속성을 가져야 한다. 출력 포드 q 는 네트(net 또는 wire)이지만 순차구문 구역 내에서 사용되기 위해 reg 로 속성이 부여되었다.

  [주] 마치 중복 선언된 것처럼 보이지만 하드웨어 객체 속성을 지정한 것이다. 베릴로그는 선언(declare)과 속성부여(attribute)에 애매한 면을 가지고 있다.

  [주] 레지스터 reg 속성을 가졌다고 해서 반드시 플립플롭을 의미하는 것은 아니다. 단지 언어적으로 순차할당(procedural assignment)의 왼편에 놓을 수 있다는 뜻이다.

- 행위의 기술(behavior description)로부터 D-플립플롭으로 해석될 수 있다.

  [주] 묘사(행위 기술)를 해석하여 전자회로를 유추(inference)해 내는 일은 합성기(synthesizer)의 중요한 역할 중 하나다. 합성기의 또 다른 역할로 최적화(optimization)가 있다.

- 예약어 posedge 는 입력 신호 clk의 상태를 평가(evaluate)하여 상승 엣지 사건인지 판별해 주는 연산자다. 이 평가가 참이면 always 구역내의 순차구문이 실행된다.

  [주] 사건에 의한 always 구역의 실행을 사건구동(event-driven)이라 한다. 소프트웨어 작성 기법에서 사건과 콜백(event & call-back function) 또는 마이크로프로세서의 인터럽트(interrupt) 메커니즘과 완벽히 같은 의미다.

3. SystemC 테스트벤치

우리는 정보화 사회에 살면서 컴퓨팅 언어를 어느 정도 들어 알고 있다. 대부분 컴퓨팅 언어들은 기본적인 대수 표현법을 차용하고 기초 영문 단어와 문법를 채택하고 있어서 따로 배울 필요가 있나 할 정도다. 컴퓨팅 언어에 의한 묘사는 결국 컴파일러라고 하는 자동 변환 소프트웨어에 의해 기계가 이해할 수 있는 형식으로 바뀐다. 따라서 컴퓨팅 언어는 단순 명료할 수 밖에 없다. 컴퓨팅 언어의 문법 보다 특정 컴퓨팅 언어가 묘사하려는 대상의 특성을 이해하는 것이

중요하다. 베릴로그는 하드웨어를 기술하려는 목적으로 발명된 컴퓨팅 언어다. 하드웨어 중에서도 디지털 회로 요소들의 행동을 묘사할 목적을 가지고 있다. 연산자와 제어 구문의 의미는 다른 컴퓨팅 언어와 다를 바 없다. 다만 문장의 실행 방식이 하드웨어의 성격대로 동시성(문장의 순서에 무관한)을 가진다는 점이 가장 큰 차이라 할 것이다. 우리가 알고리즘을 묘사한 원시 코드를 보면서 어렵다고 느끼는 것은 내용을 모르는 것이지 컴퓨팅 언어의 문법 탓이 아니라는 점을 알아야 한다.

역사적으로 묘사하려는 대상과 목적에 따라 다양한 컴퓨팅 언어가 발명 되었고, 목적이 분명히 나뉘는 소프트웨어와 하드웨어로 양분되어 발전해왔다. 하지만 설계의 규모가 기하급수적으로 증가함에 따라 생산성 문제의 해결 방안으로 설계 방법론에 높은 추상성이 요구 되었다. 이에 현존하는 컴퓨팅 언어 중 가장 광범위하게 사용되면서 높은 그리고 폭넓은 추상성을 가진 컴퓨팅 언어 C++로 합치려는 시도가 이뤄져 이에 탄생한 것이 SystemC 다. 자료구조와 알고리즘의 묘사를 보다 수월하고 재사용성을 높이기 위해 제정된 STL(Standard Template Library)이 있듯이 SystemC는 하드웨어의 동시성을 "시스템 수준"에서 묘사하기 위한 C++의 크래스 라이브러리로써 IEEE Std.1666 으로 제정되었다.

[주] "시스템 수준(System Level)"이라는 문구가 매우 다양하게 인용 되어 때로 남용을 넘어 오해를 낳기도 한다. "시스템 수준"을 한마디로 말하자면 알고리즘의 묘사를 크래스 객체로 두고 이를 소속 함수로 다루겠다는 의미다. 합성을 목적으로 하드웨어를 기술한 레지스터 트랜스퍼 수준(RTL)은 "시스템 수준"의 하위에 놓인다. 아주 단순히 말하면 여러 알고리즘들이 모여 한 집합체를 이루는 시스템에서 한 알고리즘을 수행할 때 해당 객체의 소속함수를 호출 하는 것이다. 해당 객체가 구현된 추상성의 수준(하드웨어 인지 또는 소프트웨어 인지)은 객체 내에서 해결할 문제일 뿐 "시스템 수준"에서는 개의치 않는다. 시스템 수준에서는 각 알고리즘들이 어떻게 구현되었는지 알고 싶지도 않다. 하드웨어로 구현되었더라도 소요 클럭 수와 동작 주파수 그리고 비트 폭은 관심없다. 시스템 수준에서 검증(verification)은 단지 적절한 때에 원하는 값을 얻고자 할 뿐이며, 구조 탐색(architecture exploration)은 입력에 대하여 출력을 얻기까지 어느 정도 자원(소요 클럭 수, 하드웨어 량)을 사용하는지 따져보기 위함이다.

베릴로그로 작성된 D-플립플롭을 DUT(Design Under Test) 삼아 테스트벤치(testbench)를 SystemC로 작성한다. 비록 매우 단순한 DUT와 테스트벤치 이지만 앞으로 시스템 수준의 검증환경을 구축하는 첫걸음이다. 일반적인 테스트벤치의 구성은 다음과 같다.

- DUT의 사례화(instantiate)
- DUT의 입력에 줄 신호의 생성
- DUT의 출력 검토

[주] '사례화'라는 용어가 생소하다. 조금 의미를 담아 표현하면 기술해 놓은 함수 또는 모듈 객체를 "존재하게 한다"고 하겠다.

SystemC 가 C++의 크래스 라이브러리라고 하지만 처음 접하면 마치 또다른 언어처럼 생소하다. C++의 크래스에 대한 기초 지식을 동원하여 이해 해보자. 당장 이해가 않되는 부분은 일단 받아들이자.

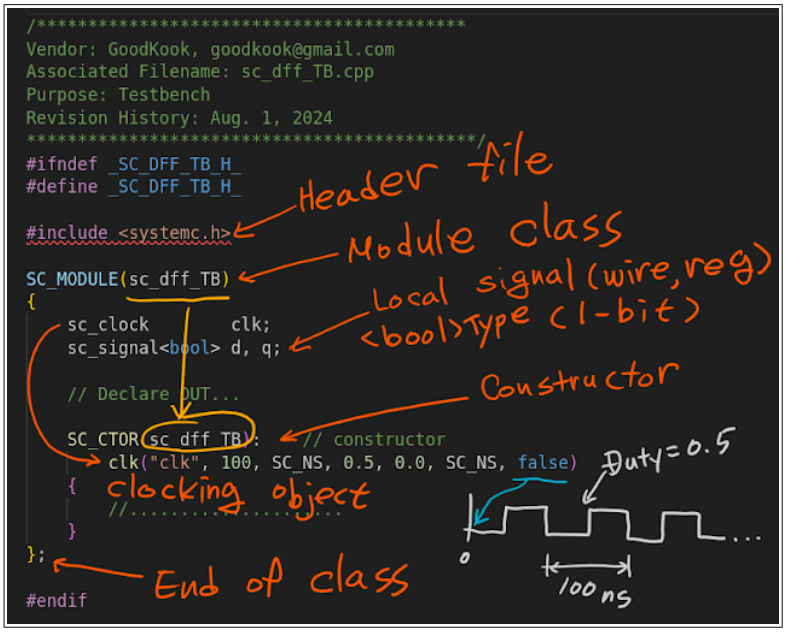
[주] SystemC를 C++를 이용하여 베릴로그를 흉내냈다는 식으로 소개 되곤 한다. 이런 이유로 굳이 SystemC를 해야 하나 하는 생각이 일견 들것이다. 이는 SystemC를 시스템 수준 모델링 능력을 파악하지 못한 채 매우 일부분인 RTL로 만 접했기 때문이다. 시스템 수준 모델링은 천천히 알아가기로

하고 기왕 배워놓은 C/C++를 하드웨어 설계에 활용한다고 여기자. C++도 익히고 하드웨어 설계도 배우면 일석이조 아닌가.

[주] SystemC를 설명하는 아주 좋은 동영상이 있다. 6편의 짧은 동영상으로 구성 되어 약 2시간 분량이다. SystemC가 생소하다면 잠시 시간을 내서 들어보길 권한다. Learn SystemC [바로가기]

3-1. SystemC 모듈의 기본구성

SystemC의 모듈의 구성은 아래와 같다. 외형적으로 보면 마치 또 다른 언어 체계 같지만 뜯어보면 C++의 크래스다.



- SystemC 크래스 라이브러리를 사용하기 위해 systemc.h 헤더 파일 인클루드(include)
- SC_MODULE 은 C++의 크래스를 재정의한 매크로다.

```
#define SC_MODULE(name) class name: ....
```

- 크래스 끝에 세미콜론(;) 이 있다.

[주] C++의 크래스는 C 의 구조체를 매우 확장한 것이다. 크래스는 자료의 정의뿐만 아니라 객체 자료형 정의 typedef 까지 수행한다.

- SC_CTOR() 은 C++의 구성자(constructor) 지정 매크로다. 당연히 모듈 크래스와 동일한 이름을 가져야 한다.
- 하드웨어 객체 sc_signal<>은 베릴로그의 wire 또는 reg 를 모사한 C++의 크래스다. 하드웨어의 비트 단위 선언을 위해 템플릿(template)을 사용하고 있다.
- sc_clock 은 특별한 하드웨어 객체로 클럭 신호를 생성해 준다.

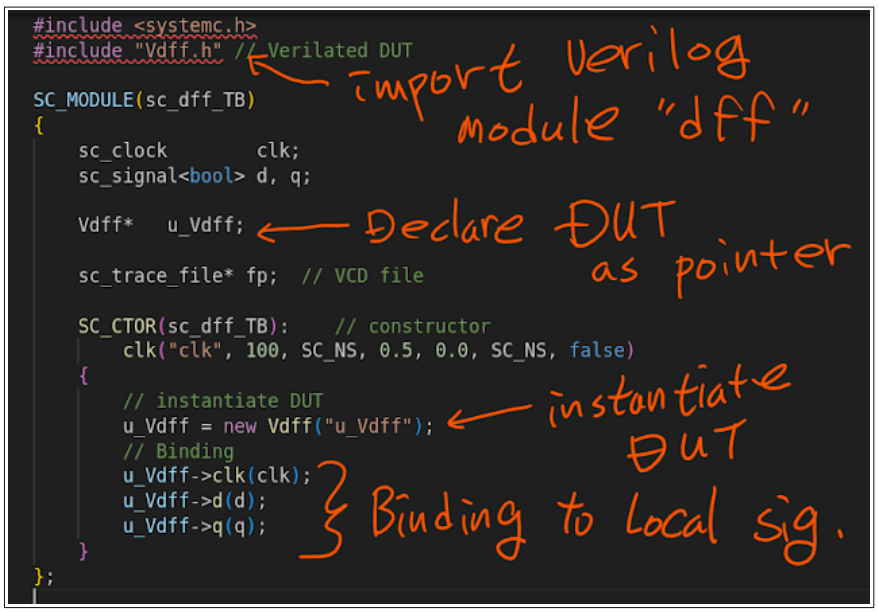
[주] SystemC의 구문을 보면서 매우 생소하다면 C++에 익숙치 못한 탓이다. 이참에 C++의 크래스를 좀더 이해해 보는 기회가 되기 바란다. 어쨌든 아래 구문은 모두 C++ 라는 점을 기억하자.

3-2. DUT의 사례화와 연결

DUT의 사례화(instantiation)와 지역신호 연결(binding)은 두 언어의 문법적 차이만 있을 뿐이지 완전히 동일한 의미다. 아래와 같은 베릴로그의 신호 연결(binding)이 있다고 하자.

```
dff u_dff (
    .clk(clk),
    .d(d),
    .q(q));
```

SystemC에서 연결을 수행하는 과정을 살펴보면 다음과 같다. C++의 포인터 매핑이다.



- Verilog로 작성된 DUT를 C++에 직접 불러올 수 없으므로 언어 변환을 통하여 C++ 언어체계로 들여온다. 변환된 DUT의 헤더 파일이 Vdff.h 다.

  [주] 매우 넓은 추상성을 가진 SystemC는 Verilog의 RTL 표현과 등가적 표현이 가능할 뿐만 아니라 사건구동 병렬 시뮬레이터 커널이 라이브러리에 내장되어 있다. Verilator는 베릴로그를 SystemC/C++ 로 변환해 주는 오픈-소스 도구다. 베릴로그 모듈 명에 대문자 V 를 붙여 SystemC 모듈 크래스를 생성한다.

- 테스트벤치 모듈 크래스 내에 DUT를 선언하고 사례화 한다. DUT는 포인터로 선언되었고 C++의 객체 동적 할당 연산자 new로 사례화 한다.
- 동적 할당된 DUT의 포트들을 SystemC 테스트벤치의 지역 하드웨어 객체 sc_clock, sc_signal<>와 연결한다.

- DUT의 사례화와 연결은 모두 테스트벤치 크래스의 구성자 내에서 이뤄진다. 이 과정을 내부 구축(elaboration)이라고 한다. 베릴로그의 initial 에 해당하는 절차도 구성자 내에서 수행한다.

3-3. 테스트 벡터 생성

DUT에 입력을 넣고 시뮬레이션을 실시하여 그 동작을 확인한다. 예제 D-플립플롭의 입력은 클럭 clk와 d 다.

a. 클럭 객체 sc_clock

SystemC는 주기적으로 끝없이 반복되는 클럭을 생성하는 특별한 크래스 객체 sc_clock 를 가지고 있다. 주기를 가지고 파형을 자동 발생시키기 위해 초기화 해준다. SystemC 간단 참고문서[1]를 보면 sc_clock 객체의 초기화 방법은 다음과 같다.

```
sc_clock("ID", period, duty_cycle, offset, first_edge_positive);
```

D-플립플롭 예제에서 클럭 객체는 모듈 크래스의 소속 데이터(member data)로 선언 되었다.

```
sc_clock    clk;
```

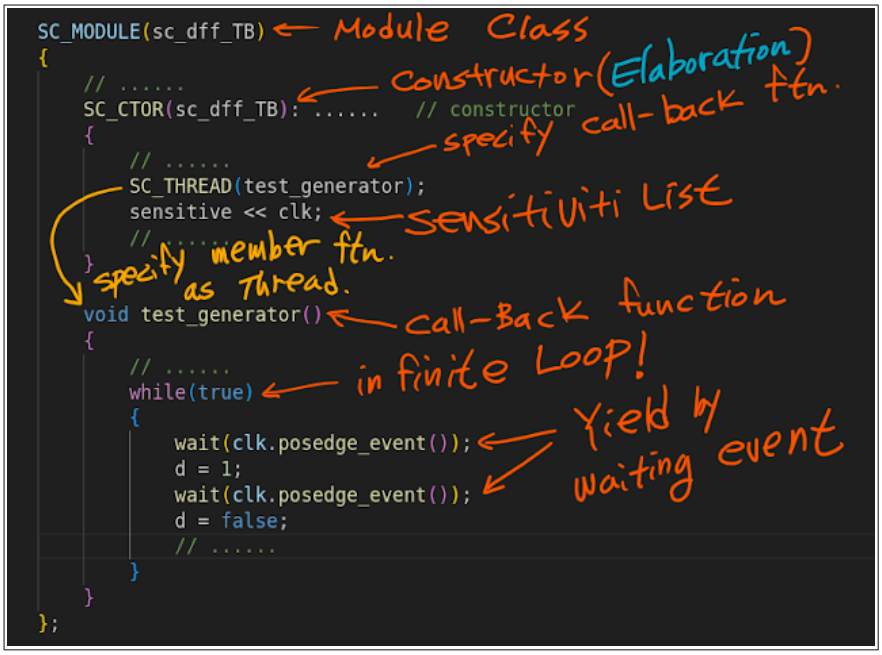
이어 구성자가 실행될 때 클럭 객체를 초기화 한다.

```
SC_CTOR(sc_dff_TB):    // constructor
        clk("clk", 100, SC_NS, 0.5, 0.0, SC_NS, false)
{ ..... }
```

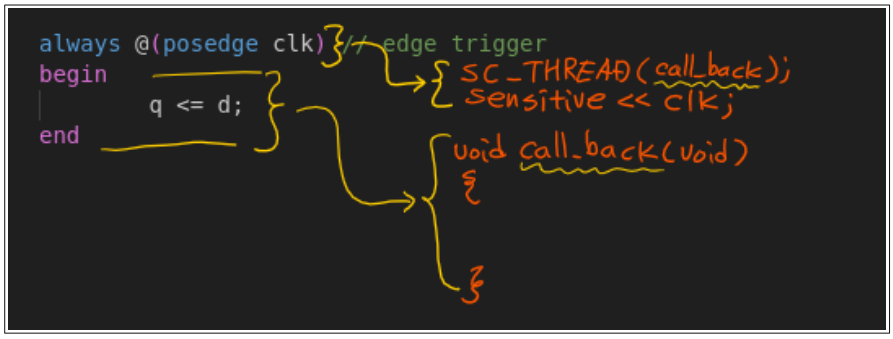
D-플립플롭의 다른 입력 d 는 시험 절차에 따라 발생 시킨다. 모듈 크래스의 소속함수 (member function) test_generator() 에 시험 절차(순서)에 따른 d 의 생성을 기술 하였다.

b. 사건구동 함수 지정

SystemC는 하드웨어 묘사를 위한 다양한 객체들의 크래스와 사건 구동 병렬실행기(event-driven simulation kernel)를 내장하고 있다. 사건탐지에 쓸 신호를 지정하고 이에 구동될 함수(콜백, call-back)를 지정한다. 이 지정은 시뮬레이션이 가동되기 전에 미리 구축 되어야 하므로 모듈 크래스의 구성자에서 수행한다. 이는 시뮬레이션 전에 준비하는 상세화(내부구축, elaboration)과정의 일부다.



위의 SystemC예에서 보인 쓰레드 함수와 사건 감응신호 지정은 베릴로그의 always @(sensitivity_list) 에서 감응 리스트의 지정과 동일한 의미다.



[주] 하드웨어를 기술하는 언어(Verilog, VHDL 등)는 병렬 실행 구문을 기본으로 순차 실행 구문 구역을 지정하지만 C++ 는 병렬 실행의 개념을 가지고 있지 않다. SystemC는 병렬 실행을 모의하기 위해 사건 구동 시뮬레이션 커널을 내장하고 있다. 베릴로그에서 사건에 의하여 구동될 순차 구문 구역이 always 에 이어지지만 모두 순차구문인 C++로 사건 구동을 구현해야 하는 SystemC는 사정이 다르다. 감응(sensitive)과 이에 반응하여 호출될 함수(call-back function)를 따로 지정한다. 이는 마이크로 컨트롤러 프로그래밍에서 인터럽트 벡터 설정(인터럽트 번호에 서비스 루틴을 지정하는 절차)과 같다. 사건에 구동될 함수는 호출 인수와 되돌림 값이 모두 void 인 모듈 크래스의 소속함수(member function)다. 사건에 대하여 콜백 함수를 호출하는 주체는 사건구동 시뮬레이션 커널이다. 함수를 운영하는 방식은 메쏘드(method) 혹은 쓰레드(thread)가 있다. SC_METHOD() 로 지정된 함수는 감응 신호에 사건이 발생할 때마다 호출된다. SC_THREAD()로 지정된 함수는 준비과정(elaboration process)에서 한번 호출되므로 함수 내에 무한 반복 구간을 두고 사건 대기 구문을 가지고 있어야 한다. 소프트웨어에서 병렬 프로그래밍 기법으로 흔히 사용하는 멀티 쓰레딩(multi-threading)과 같다.

테스트 입력 d 를 발생 시키는 쓰레드 함수 test_generator()의 무한 반복 구문 while(true) {...} 내에 사건을 대기하는 wait(....) 를 두고 있다. SystemC 의 wait(...) 는 감응으로 지정한 신호에 사건이 발생할 때까지 대기한다. 이는 실행 제어권을 시뮬레이션 커널에 양보(yield)하는 역할을 한다. SystemC 간단 참고문서[1]에 wait(...)의 사용 방법을 찾아보면 다음과 같다.

```
wait(); // Wait for event as specified in static sensitivity list
wait(event_expression); // Temporary overrides the static sensitivitiy
list
wait(time);
wait(time, event_expression);
```

하드웨어 객체 clk 에 상승 엣지(또는 하강엣지) 사건이 발생할 때까지 시뮬레이션을 대기 시킬 수 있다.

```
wait(clk.posedge_event());
wait(clk.negedge_event());
```

c. 하드웨어 객체 sc_signal<>

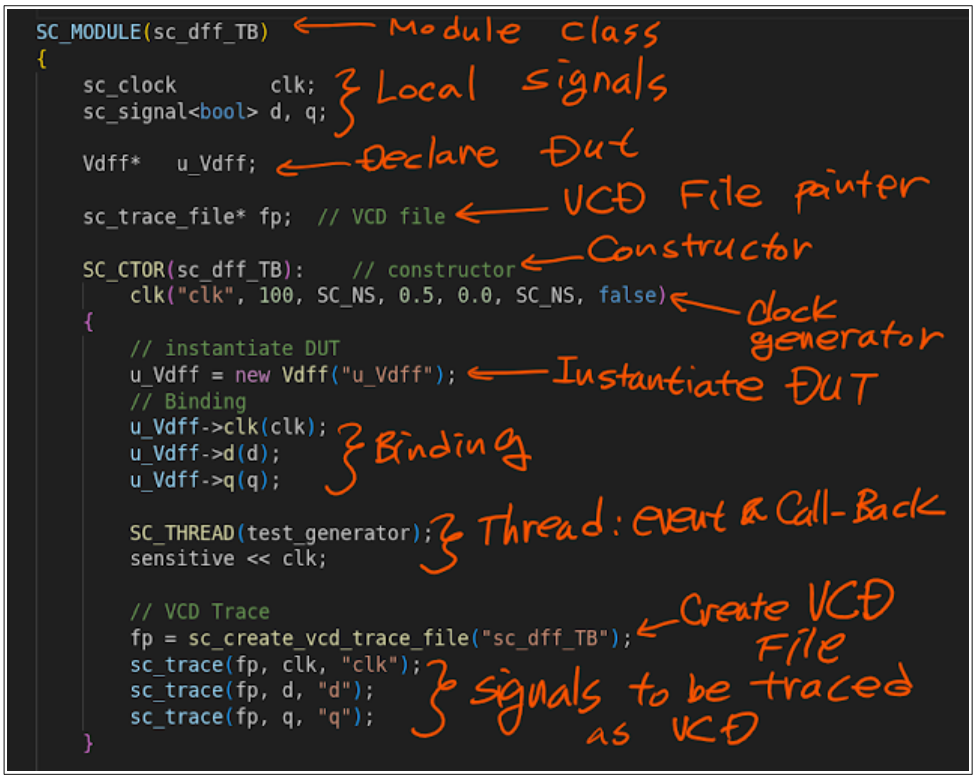
예제에서 DUT의 입력에 연결한 d 는 sc_signal<> 로 선언된 하드웨어 객체다. 이 객체에 접근하는 방법은 읽기 .read() 와 쓰기 .write() 가 있다. 소속함수로 접근 할 수 있고 할당 연산자 = 가 중복정의(오버로드, overload) 되어 있다. 예를 들어, d = 1; 과 d.write(1); 은 동일 하다. 하드웨어 언어에서 모든 할당은 하드웨어 객체를 상대로 하는 것이 기본 이지만 C++ 언어는 모두 변수 할당이다. 하드웨어 객체에 대한 접근과 변수 접근을 구분하기 위해서 라도 할당 연산자 대신 소속함수(메쏘드) .read()와 .write()를 통해 접근하는 방식을 쓰도록 한다.

> [주] 하드웨어에서 부품간 연결에 단순히 "전선"이라 부르는 "와이어"를 사용한다. 이 "와이어"를 컴퓨팅 언어로 표현하려면 고려할 사항이 많다. 소프트웨어에서 "전선"의 개념 없고 "변수"만 있을 뿐이다. 게다가 전류가 흐르는 전선에는 시간적 순서가 없다. 디지털 회로에서 조합회로와 순차회로를 구분 하는 이유는 "순서" 때문이다. 하드웨어 객체 sc_signal<>는 C++라는 소프트웨어 개발 언어로 하드웨어의 "와이어"를 모사하기 위한 템플릿-크래스로서 단순한 변수 이상의 의미를 가지고 있으며 다양한 접근 방법(method)을 갖추고 있다. 이 하드웨어 객체 sc_signal<>는 사건 구동 병렬실행 시뮬레이션 커널의 중요 매개이기도 하다. 프로그래밍 언어의 변수와 하드웨어 객체를 구분하는 근본적인 이유는 소프트웨어로 하드웨어의 행동을 모의하려고 하기 때문이다. 하드웨어의 행동을 소프트웨어로 흉내 내기 위해 사용된 기법이 사건구동(event-driven)과 지연 할당(deferred assignment)이다. 이에 대해서는 하드웨어 시뮬레이터의 내부(simulator kernel)를 다룰 때 살펴보기로 한다.

3-4. VCD 파형

디지털 하드웨어 설계의 입출력을 관찰하는 고전적인 방법은 역시 파형보기(waveform view)다. VCD(Value Changed Dump)는 디지털 파형을 기록하는 베릴로그 표준 형식이다. SystemC는 하드웨어 객체의 변화를 VCD 로

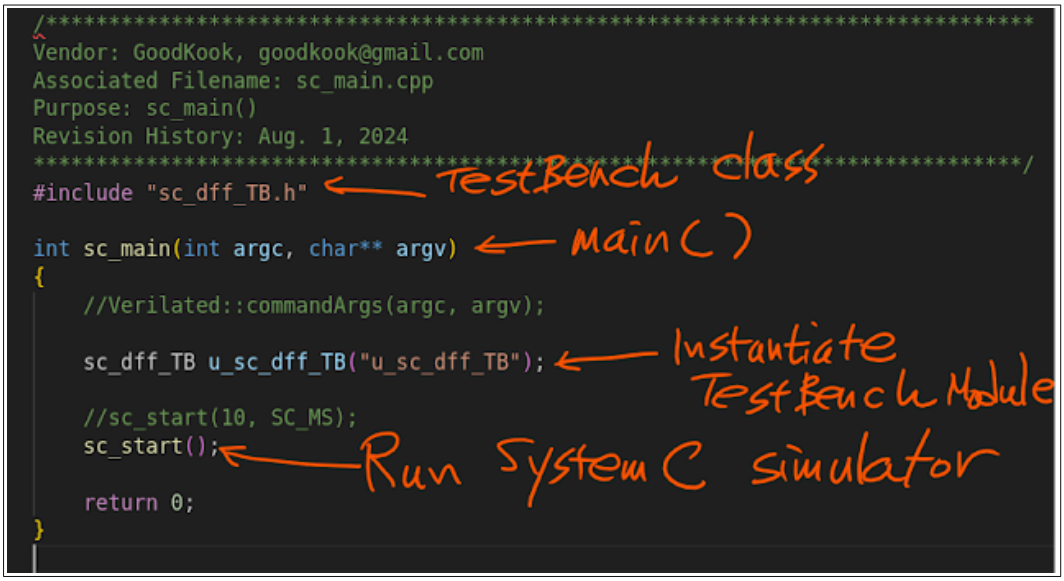
기록하는 방법을 제공한다. 베릴로그에서는 VCD 추적을 initial 구역에 지정 했었다. 모듈 크래스 구성자에서 VCD를 기록하도록 지정할 수 있다.



- VCD 파일명은 "sc_dff_TB.vcd" 다.
- VCD 파일에 기록할 신호는 clk, d, q 다.

3-5. int sc_main(int argc, char** argv)

C++ 프로그램의 시작은 main() 호출이다. SystemC의 시작은 sc_main()이다. 헤더 파일로 기술한 테스트 벤치 모듈 클래스를 들여와서 사례화 한 후 시뮬레이터를 개시한다.



시뮬레이터는 아래 조건 중 하나를 충족할 때 중지된다.

- 지정된 시뮬레이션 시간에 도달했을 때
- 모든 채널(하드웨어 객체)에 사건이 발생하지 않을 때
- 사용자가 sc_stop()을 호출 했을 때

4. 메이크(make) 유틸리티

Verilog 설계에 SystemC 테스트벤치를 씌운 시뮬레이터를 만들기까지 Verilator 변환도구가 동원되었고 최종적으로 GNU C++ 컴파일러를 사용하여 실행 파일을 만들어낸다. 일반적인 소프트웨어 개발용 표준 라이브러리에 더하여 특수한 라이브러리들을 동원하면 명령줄 옵션이 매우 복잡해진다. 개발 중 반복되는 컴파일 명령을 매번 명령줄에 입력하기도 어렵고 개발자의 피로도가 쌓여 실수를 낳게 되므로 스크립트를 활용 하는 것이 좋다.

그래픽 환경에서 통합된 개발도구의 활용이 늘고 있다. 통합 환경이 사용자 편의성을 높이긴 하지만 결국 스크립트의 실행이다. 필요할 때마다 외부 라이브러리를 추가하고 다양한 옵션을 자유롭게 활용 하려면 통합 환경의 사용법을 익혀야 하는데 결국 메이크 스크립트(Makefile)를 마주하게 된다. 메이크(make) 유틸리티는 명령줄에서 쓰이는 가장 강력한 스크립트 활용 방법이다. 긴 세월동안 수많은 개발자들에 의해 향상된 make 유틸리티의 내용은 매우 방대하지만 기본 사용법만 익혀도 활용도는 매우 높다.

[주] make의 사용법을 다룬 글들이 많지만 임대영(RAXIS)의 GNU Make 강좌글을 추천한다.

`https://doc.kldp.org/KoreanDoc/html/GNU-Make/GNU-Make.html`

4-1. 목표와 의존 관계

명령줄에서 아무런 인수 없이 make 를 실행하면 현재 디렉토리에서 Makefile 을 기본으로 읽어 그 내용을 수행한다. Makefile 의 가장 단순한 사용법은 콜론(:) 을 사이에 둔 의존 관계(dependency)의 표현이다. 콜론의 왼편에 놓인 목표(target)를 얻기 위해 오른편에 놓인 파일들에 의존(dependent)한다는 의미다. 바로 이어 목표에 도달하는

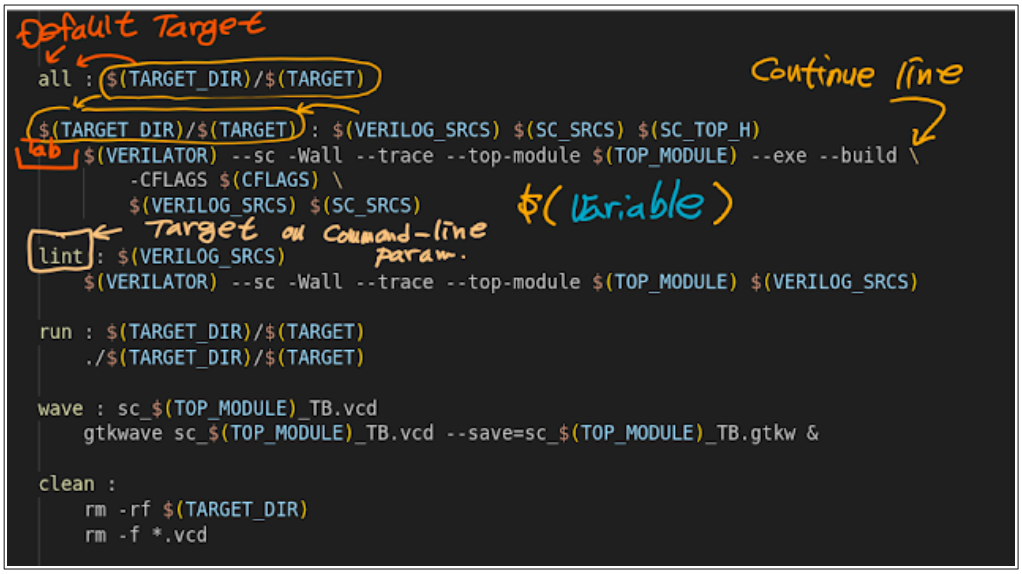
방법을 기술 한다. 이때 들여쓰기(indent)는 반드시 공백없는 탭(tab) 문자가 선행되어야 한다. 의존성은 목표 파일의 존재 여부와 두 파일의 날짜와 시간을 살펴 정한다. 예를 들어 아래와 같은 간단한 의존 관계를 보자.

```
# Simple 'Makefile'
hello : hello.c
    gcc -o hello hello.c
```

목표 파일 hello는 파일 hello.c 에 의존하는 관계에 있다. 목표인 hello 가 존재하지 않거나 hello.c 의 날짜가 목표보다 최신일 경우 바로 아래에 놓인 절차를 수행하라는 뜻이다. 위의 경우 hello.c 를 gcc 로 컴파일 하여 hello 를 만든다. 첫번째 칼럼에 # 은 주석이다. Makefile 내에 다수의 목표를 놓을 수 있다. 명령줄에서 make를 실행 할 때 명령줄 인수를 주어 여러 목표 중 하나를 선택 할 수 있다. 예를들어 명령줄에서 아래와 같이 인수를 주고 메이크 하였다고 하자.

```
$ make lint
```

Makefile 에 lint를 목표로 하는 절차가 있었다면 이를 찾아 수행한다. 다음은 간단한 Makefile의 예다.



의존 관계에서 콜론(:)의 왼편이 명령줄의 인수와 일치하는 목표를 찾아 이에 도달하는 절차를 수행한다. 위의 예에서 보듯이 목표 lint에 의존관계에 있는 파일들을 검사한다. 의존 관계에 있는 파일들의 목록은 변수 VERILOG_SRCS 에 지정 되었다. 목표에 도달하기 위한 절차는 VERILATOR 변수로 지정한 명령을 수행하는 것이다.

4-2. 내부 변수

Makefile 스크립트는 내부적으로 변수를 사용할 수 있다. 변수를 아래와 같이 선언해 주었다면 위의 수행 절차가 이해 될 것이다.

```
# Makefile variables:
VERILOG_SRCS = dff.v
SC_SRCS      = sc_main.cpp
```

```
SC_TOP_H      = sc_dff_TB.h
VERILATOR     = verilator
TOP_MODULE    = dff
TARGET        = V$(TOP_MODULE)
TARGET_DIR    = obj_dir
```

4-3. 다중 목표

위의 예에서 Makefile에 all, lint, run, wave, clean 등 5가지 명령줄 목표를 가지고 있다. 목표 all 은 make에 예약되었고 나머지는 사용자 지정 목표다.

a. all

목표 all 은 Makefile에서 예약되었다. 명령줄에서 인수없이 make 만 수행할 경우 자동으로 목표 all 을 찾아 수행한다. 위의 예에 따르면 목표 all 은 $(TARGET_DIR)/$(TARGET) 에 의존한다. 만들 방법이 제시되지 않고 있으므로 $(TARGET_DIR)/$(TARGET)가 목표인 의존성을 찾아 만들기(make)를 수행한다.

[주] Verilator 는 Verilog 를 SystemC/C++ 로 변환 한다. 옵션으로 --exe 와 --build 옵션을 주면 변환한 DUT와 테스트벤치 그리고 main() 이 포함된 C++ 원시 파일을 읽어 실행 파일(시뮬레이터)를 생성한다. 이때 GNU C++/clang++ 컴파일러를 사용한다.

b. lint & build

목표 lint 는 Verilator로 하여금 변환만 수행하도록 한다. Verilog의 무결성(문법적 오류는 물론 효과 없는 코드, 자료형 불일치 등)을 검사한다. 목표 build는 DUT와 테스트벤치를 묶어 시뮬레이터(실행파일)을 만든다.

c. run

목표 run 은 Verilator와 C++ 컴파일러로 만들어진 실행 파일(시뮬레이터)를 실행한다. 리눅스 운영체제에서 실행되는 응용프로그램 이므로 실행 환경의 설정이 필요할 수 있다. DUT와 테스트벤치를 묶어 시뮬레이터를 빌드하는 과정에서 SystemC를 사용 하였으므로 크래스 헤더 파일을 들여오고 실행시 동적 라이브러리를 적재하기 위한 환경 변수를 설정해 주어야 한다.

```
export SYSTEMC            = /opt/systemc
export SYSTEMC_HOME       = $(SYSTEMC)
export SYSTEMC_INCLUDE    = $(SYSTEMC_HOME)/include
export SYSTEMC_LIBDIR     = $(SYSTEMC_HOME)/lib
export LD_LIBRARY_PATH  :=$(LD_LIBRARY_PATH):$(SYSTEMC_LIBDIR)
```

SystemC를 설치하면 크래스 헤더와 라이브러리를 특정 위치에 놓는다. 따로 지정하지 않았다면 /opt/systemc 다. 시뮬레이터를 빌드하는 컴파일러에게 이 경로를 환경변수를 통해 알려준다. 아울러 SystemC의 시뮬레이션 엔진(simulation kernel)은 동적 라이브러리 libsystemc.so (shared object)로 설치되었다. 컴파일된 시뮬레이터(바이너리 파일)를 실행 할 때 시뮬레이션 커널과 함께 실행 되어야 하므로 동적 라이브러리가 놓인 경로 /opt/systemc/lib 를 리눅스 배쉬 쉘(bash-shell)의 환경변수 LD_LIBRARY_PATH에 추가 시켜야 한다. export 는 Makefile의 외부변수 지정을 의미한다.

d. wave

목표 wave 는 시뮬레이션을 수행한 후 기록된 VCD 파형을 보기 위한 것이다. VCD 파형 보기 소프트웨어로 gtkwave를 사용하였다.

e. clean

목표 clean 은 현재 디렉토리를 정리한다. 빌드하는 과정에서 여러 중간 파일들이 생성된다. 개발 중 파일명이 변경 되기도 하고 임시 저장된 파일도 있다. 보관을 위해 디렉토리 청소가 필요할 때 보존되어야 할 파일들과 지워도 좋을 파일을 분명히 해줄 필요가 있다.

5. 하드웨어 언어의 코딩 스타일

"내 칩 서비스" 표준 셀 디자인 킷(이하 디자인 킷)과 함께 제공된 예제의 시뮬레이션을 수행해 보면서 코딩 스타일이 설계 자동화 도구(HDL 시뮬레이터, 합성기 등)에 미치는 영향을 살펴 보기로 한다. '디자인 킷'은 깃-허브를 통해 내려 받을 수 있다.

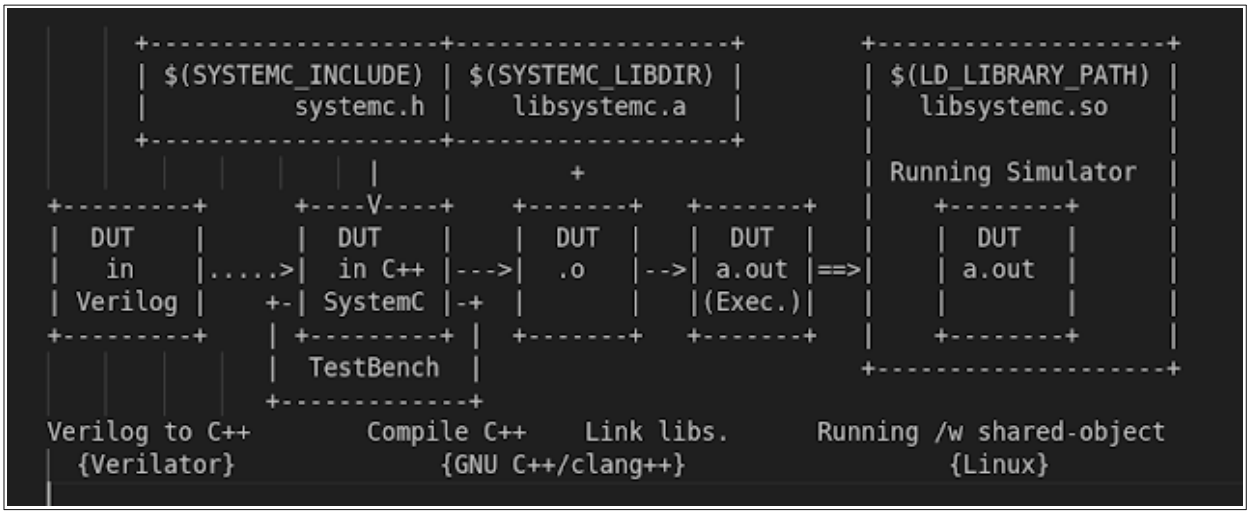
```
https://github.com/GoodKook/ETRI-0.5um-CMOS-MPW-Std-Cell-DK
```

예제의 수행은 모두 오픈-소스 반도체 설계 도구를 활용한다. 오픈-소스 반도체 설계 도구와 디자인 킷의 설치 방법을 설명한 문서를 참조한다.

"ETRI 0.5um CMOS Std-Cell DK: 오픈-소스 반도체 설계 도구 설치 "[내려받기]

5-1. D-플립플롭 시뮬레이터의 빌드와 실행

레지스터 트랜스퍼 수준(RTL)에서 베릴로그로 기술한 D-플립플롭을 언어 변환 도구 Verilator를 사용하여 SystemC/C++ 모델로 변환하고 이를 SystemC 테스트벤치와 빌드(컴파일 및 링크)한다. 모두 C++ 이므로 GNU의 빌드 도구들(GCC/G++)이 동원된다.



예제의 폴더로 이동 후 Verilog RTL로 기술한 D-플립플롭 과 SystemC 테스트벤치를 묶어 시뮬레이터를 빌드한다. 먼저 예제 디렉토리로 이동하여 베릴로그 DUT와 SystemC 테스트 벤치를 살펴본다.

```
$ cd ~/ETRI050_DesignKit/Tutorials/2-3_Lab1_dff/dff
```

시험할 대상(DUT, Desugn Under Test)은 RTL에서 베릴로그로 기술된 모듈 dff 다.

```
/*************************************
Associated Filename: dff.v
Purpose: D-FlipFlop
*************************************/
module dff(clk, d, q);
input clk, d;
output q;
reg q;

always @(posedge clk) // edge trigger
begin
    q <= d;
end

endmodule
```

SystemC 테스트벤치는 다음과 같다. 시험할 DUT가 RTL 이므로 테스트를 수행에 필요한 시험입력 또한 이에 준하는 추상화 수준에서 작성 되어야 한다.

```
/*************************************************************
Associated Filename: sc_dff_TB.h
Purpose: Testbench
*************************************************************/
#ifndef _SC_DFF_TB_H_
#define _SC_DFF_TB_H_

#include <systemc.h>
#include "Vdff.h" // Verilated DUT

SC_MODULE(sc_dff_TB)
{
    sc_clock        clk;
    sc_signal<bool> d, q;
    Vdff*   u_Vdff;
    sc_trace_file* fp;      // VCD file
    SC_CTOR(sc_dff_TB):     // constructor
        clk("clk", 100, SC_NS, 0.5, 0.0, SC_NS, false)
    {
        // instantiate DUT
        u_Vdff = new Vdff("u_Vdff");
        // Binding
        u_Vdff->clk(clk);
        u_Vdff->d(d);
        u_Vdff->q(q);

        SC_THREAD(test_generator);
        sensitive << clk;
```

```
        // VCD Trace
        fp = sc_create_vcd_trace_file("sc_dff_TB");
        sc_trace(fp, clk, "clk");
        sc_trace(fp, d, "d");
        sc_trace(fp, q, "q");
    }

    void test_generator()
    {
        int test_count =0;
        d.write(0);
        while(true)
        {
            wait(clk.posedge_event());
            d = 1;
            wait(clk.posedge_event());
            d = false;
            wait(clk.negedge_event());
            d = 1;
            wait(clk.posedge_event());
            d = 0;
            wait(clk.posedge_event());
            d = 1;
            wait(clk.posedge_event());
            wait(clk.posedge_event());
            sc_close_vcd_trace_file(fp);
            sc_stop();
        }
    }
};
#endif
```

SystemC는 높은 시스템 수준에서 낮은 RTL 까지 폭넓은 추상화 수준을 지원한다. SystemC 모듈은 먼저 두개의 헤더 파일을 들여오고(include) 있는데, "systemc.h"는 SystemC의 하드웨어 묘사용 크래스들을 사용하기 위한 것이며, "Vdff.h"는 베릴로그에서 C++ 로 변환된 모델을 기술한 크래스 정의다. 위의 SystemC 테스트벤치는 가장 기본적인 구성을 갖추고 있다. DUT를 사례화 하고 지역 신호에 연결 하며 테스트 입력을 생성한다. 이번 예제는 워낙 단순한 DUT라 출력 검사 부분은 생략 되었다.

예제 디렉토리에 준비되어 있는 Makefile을 가지고 시뮬레이터를 빌드한다.

```
$ make build
    verilator --sc -Wall --trace --top-module dff --exe --build \
        -CFLAGS -std=c++17 \
        dff.v sc_main.cpp
        .........…
- V e r i l a t i o n   R e p o r t: Verilator 5.027 ..…
- Verilator: Built from 0.011 MB sources in 2 modules, ..…
- Verilator: Walltime 0.770 s (elab=0.004, cvt=0.008, bld=0.744)...
```

오류 없이 시뮬레이터(실행 파일)가 만들어 졌으면 실행시켜 보자.

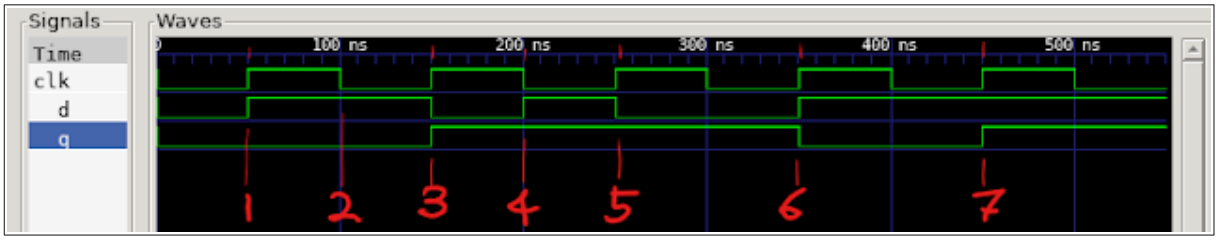
```
$ make run
./obj_dir/Vdff
```

```
SystemC 3.0.0-Accellera --- Jun 18 2024 08:49:55
  Copyright (c) 1996-2024 by all Contributors,
          ALL RIGHTS RESERVED
Info: (I702) default timescale unit used for tracing: 1 ps (sc_dff_TB.vcd)
Info: /OSCI/SystemC: Simulation stopped by user.
```

VCD 로 기록된 시뮬레이션의 결과를 살펴보자.

```
$ make wave
```

DUT가 단순하고 시뮬레이션도 아주 짧다. D 플립플롭의 RTL 묘사는 클럭 엣지의 사건에 감응되어 입력 d 를 출력 q 로 전송한다. 입력의 이전 값을 취하고 있는 점에 주목하며 시뮬레이션 결과 파형을 살펴보자.



(1) 입력 d 와 clk 가 동시에 사건이 발생했다. 클럭의 상승 엣지 사건 (posedgge clk)에 구동된 순차 할당문은 입력 d 의 할당 이전 값을 취한다.

(2) clk 에 하강 엣지 사건이 발생 했지만 이에 감응된 프로세스는 없다. D 플립플롭의 행동을 묘사한 always @() 는 상승 엣지에서만 반응한다.

(3) 이번에도 clk의 상승 엣지 사건와 입력 d의 할당이 동시에 일어났다. 출력 q 를 앞서 (1)의 경우와 비교해 보면 동일하게 d 의 이전의 값을 취하고 있다.

(4) 출력 q 는 clk의 하강 엣지 사건에 영향을 받지 않고 이전 값을 유지한다.

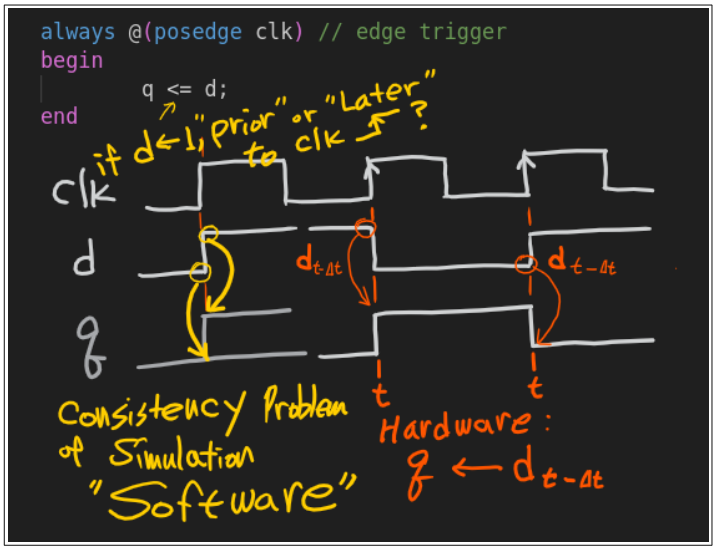
(5) 앞서 (3)의 경우와 같다.

(6) 앞서 (1)의 경우와 같다. clk의 상승 엣지 사건에 감응하여 실행된 할당문은 d 의 이전 값을 취한다.

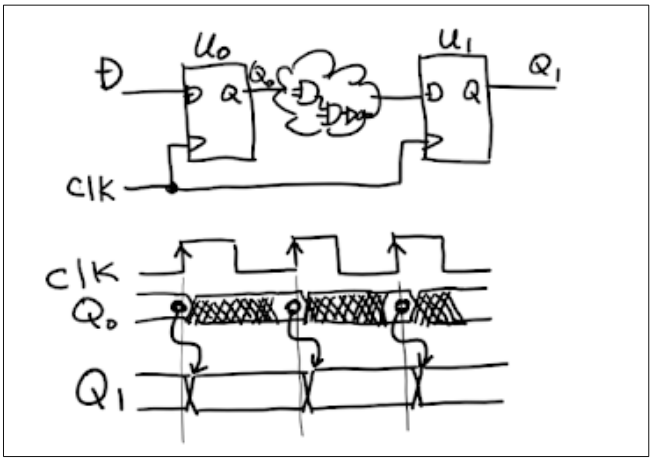
(7) 앞서 (3), (5)의 경우와 같다. 입력 d 의 이전 값을 취하여 출력 q에 전송되었다.

5-2. 레지스터 전송 수준

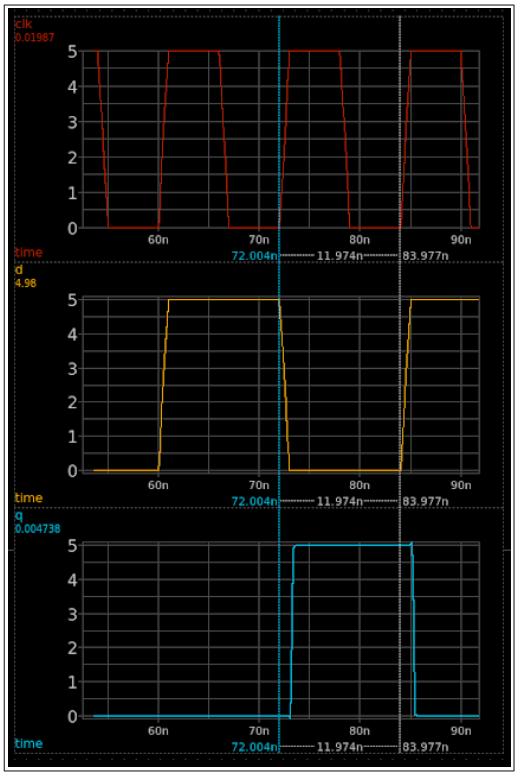
앞서 D-플립플롭 시뮬레이션의 VCD 파형에서 (1), (3), (5), (6) 시점을 보면 clk의 상승 엣지 사건과 동시에 입력 d 의 사건이 발생했다. 이때 엣지 트리거 플립플롭의 출력 q는 클럭 사건이 일어난 이전 시점의 d 값을 취하고 있다. 이는 RTL 시뮬레이터가 순차 논리 회로의 동작 원리를 따르고 있음을 보여준다.



흔히 RTL(레지스터 전송 수준)이라고 하는 추상화 수준은 디지털 하드웨어에 근접하여 "클럭 동기"에 맞춰 작동하는 디지털 회로를 묘사 했다는 의미가 포함되어 있다. 한 클럭으로 연동된 두 플립플롭 사이에 조합 회로들이 놓여 있다.

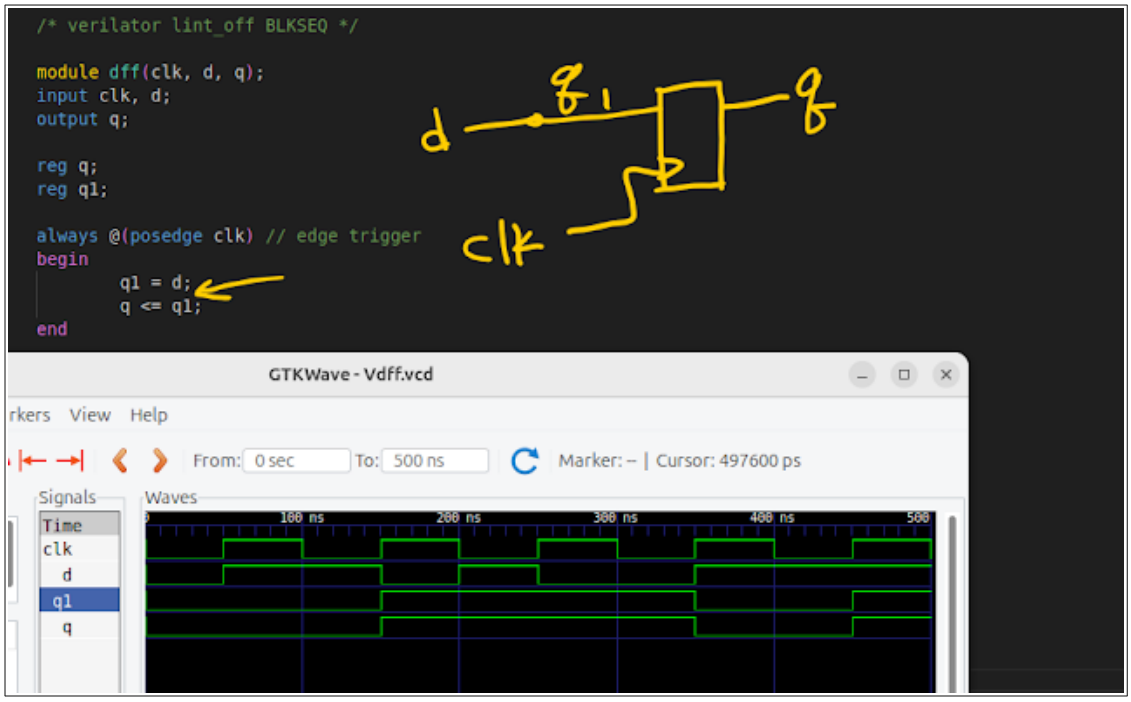


플립플롭 U0의 출력 Q0은 클럭의 상승 엣지에서 시작된다. 이 출력 값은 조합회로를 통과하며 지연으로 인한 불안정을 보이기 시작한다. 플립플롭 U1은 클럭의 다음번 상승 엣지에서 조합회로를 통과하며 지연된 값을 취한다. RTL이란 연속적인 클럭의 엣지를 사이에 두고 플립플롭 사이에 이뤄지는 신호의 전송을 표현한 것이다. 아래 그림은 클럭 엣지 트리거 D-플립플롭을 트랜지스터의 조합으로 구성한 후 SPICE 회로 시뮬레이터로 얻은 파형의 모습이다. 시뮬레이션 소프트웨어들이 하드웨어를 묘사하고 동작을 전자회로에 맞게 모의하고 있음을 단적으로 보여준다.

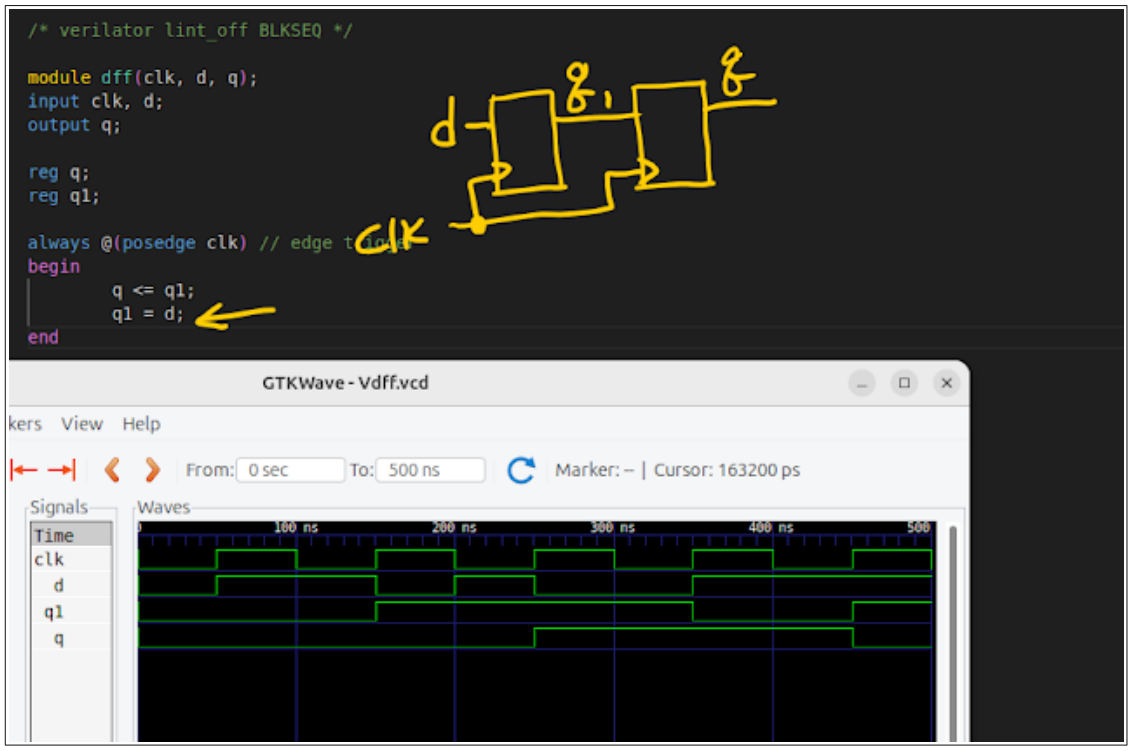


5-3. 컴퓨팅 언어에 의한 하드웨어의 묘사

하드웨어 D-플립플롭을 Verilog로 RTL에서 기술하고 C++ 로 작성한 테스트벤치와 함께 시뮬레이터를 만들고 실행해 동작을 파형으로 관찰하였다. 그 결과 컴퓨팅 언어로 전자회로의 동작을 모의하고 있음을 알게 되었다. 하지만 묘사하려는 대상이 하드웨어의 행동이라는 점을 염두에 두어야 한다. 하드웨어를 언어적으로 표현하기 위해 사용하는 객체는 소프트웨어 언어의 변수와는 사뭇 다르다. 하드웨어 언어에서도 객체를 심볼로 표현하지만 행동의 묘사방법(코딩 스타일)에 따라 단순한 전선(또는 조합회로)이 될 수도 있고 저장소(또는 순차회로)가 되기도 한다. 하드웨어를 표현한 문장은 전선 혹은 저장소 모두 병렬 실행이 원칙이다. 다음은 베릴로그의 순차구문 구역 내에 연속적인 할당을 보여주는 예다.



입력 d가 q1을 거쳐 연속적으로 q 까지 할당 되었다. 이때 d 와 q1의 연속적인 할당은 하나의 전선으로 합쳐진다. 만일 연속 할당 구문의 순서를 바꿀 경우 다음과 같은 결과를 낳는다.



할당 문의 순서가 역전되므로 그로 인하여 저장소가 늘어났다. 이는 always 구역이 소프트웨어 처럼 순차 실행을 하고 있기 때문이다. 위의 예에서 두가지 할당 연산자가 사용되었는데, '='은 즉시 할당, <= 은 지연 할당이라 한다. 즉시

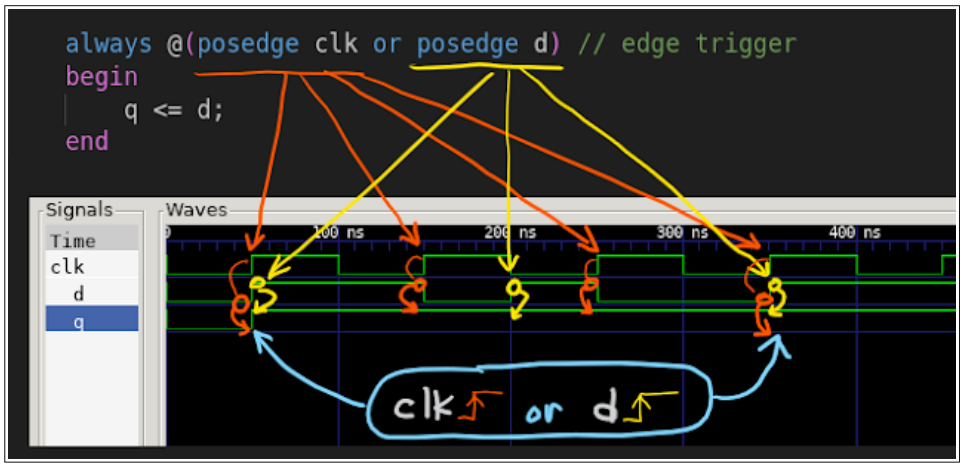
할당은 저장이 없는 조합회로(전선)를 표현할 때, 지연할당 연산자는 저장소(플립플롭)를 묘사할 때 적용된다. 하지만 순차구문 구역 내에서 할당 구문의 순서에 따라 즉시 할당문도 저장소를 만들 수 있다는 점에 주의해야한다. 할당 구문 순서에 상관없이 모두 연속적인 플립플롭을 묘사하고자 한다면 모두 지연 할당 연산자를 적용한다.

5-4. 하드웨어 행위 묘사의 코딩 스타일

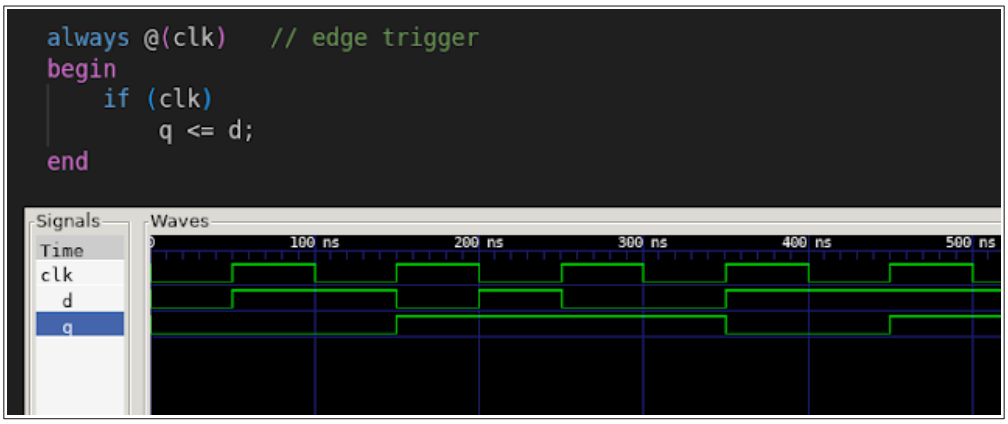
컴퓨팅 언어로 알고리즘 또는 하드웨어를 묘사 할 때 대상에 따라 주의가 필요하다. 하드웨어를 대상으로 할 경우 이에 제시되는 언어의 문장구성과 문장의 실행방식에 주의해야 한다. 문법 오류없이 기술 되었다고 해서 의미를 제대로 담았다고 보장 할 수는 없기 때문이다. 의미를 명확히 해주기 위한 지침이 필요한데 이를 코딩 스타일(coding style)이라 한다. 앞서 할당 연산자의 사용 만으로도 다른 의미가 될 수 있다는 점을 알게 되었다. 하드웨어 언어의 또 다른 주의점으로 문장의 실행이다. 하드웨어는 모든 문장이 병렬 실행을 원칙으로 하며 always 구역은 한개의 병렬 구문(여러개 순차 구문을 담은 절)과 같다. 언어로서 Verilog의 시뮬레이션 동작은 사건 감응과 구동이라는 원칙하에 일관성을 가지고 설명될 수 있다.

> [주] 감응 목록에 넣고 빼기에 의해 할당문의 실행 결과가 다를 수 있다[4]. 코딩 스타일은 컴퓨팅 언어로서 문법적 규칙과 문장의 실행 방식(병렬실행과 순차실행) 외에 합성(synthesis)을 위한 작성 양식이다. 사건구동과 감응 목록은 하드웨어의 행동을 컴퓨팅 언어로 기술하기 위한 소프트웨어적 기법이며 실제 디지털 회로를 정확히 반영하지 않는다는 점을 염두에 두어야 한다. 심지어 합성기 마다 다른 회로를 생성해서 시뮬레이션과 일치하지 않을 수 있다. 이런 이유로 EDA 업계를 중심으로 합성용 RTL 코딩 스타일의 표준이 제정되었다[5].

순차구문 영역 always의 감응 리스트에 posedge clk 와 d 도 함께 주어 졌다면 문제가 발생한다. 할당문은 지정된 감응에 충실히 반응하여 아래와 같은 결과를 보여줄 것이다. 이는 우리가 원하던 플립플롭의 동작이 아니다.

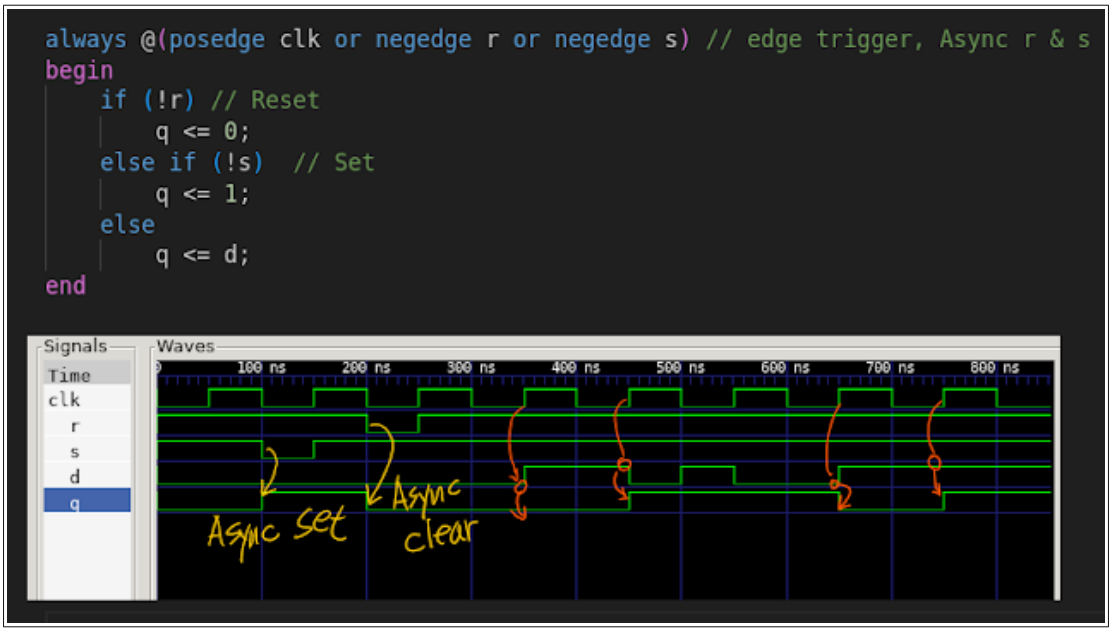


감응목록에 따라 다른 결과를 보여주기도 하지만 시뮬레이션의 실행 속도에 영향을 주기도 한다. 아래의 경우 D-플립플롭의 행동을 적절하게 기술하고 있다. 하지만 always 구역이 clk의 상승과 하강 엣지 사건에 모두 반응한다. 하강 엣지에서 불필요하게 always 구역을 수행하므로써 시뮬레이터의 부담을 증가시킨다.



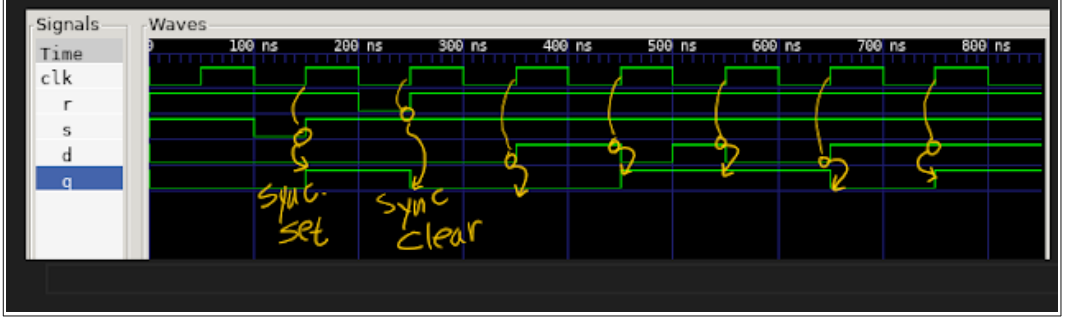
아래의 예는 마치 레벨 트리거 래치(level trigger latch)처럼 보인다. 감응신호 clk와 d 의 모든 사건에 always 가 반응 하지만 할당은 if 문의 조건을 따른다.

비동기 셋과 리셋을 가진 D 플립플롭을 기술하면 아래와 같다. clk의 상승 엣지 사건과 무관하게 q 가 셋 또는 리셋되고 있다. 플립플롭 동작은 clk의 상승 엣지 사건에 의해 작동한다.

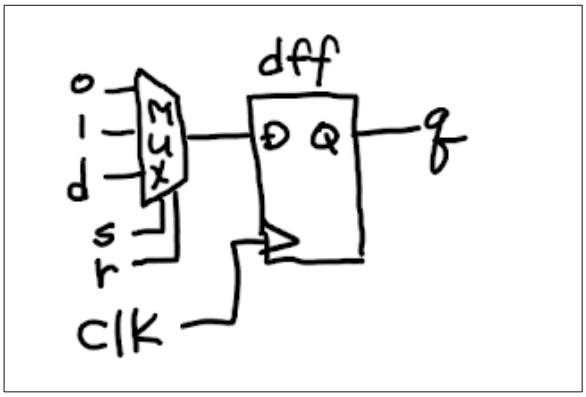


감응 목록에서 r 과 s 를 빼면 모두 clk에 동기 시킬 수 있다. 클럭 동기 셋과 리셋을 가진 플립플롭은 권장하지 않는다.

```
always @(posedge clk) // edge trigger, Sync r & s
begin
    if (!r) // Reset
        q <= 0;
    else if (!s)  // Set
        q <= 1;
    else
        q <= d;
end
```



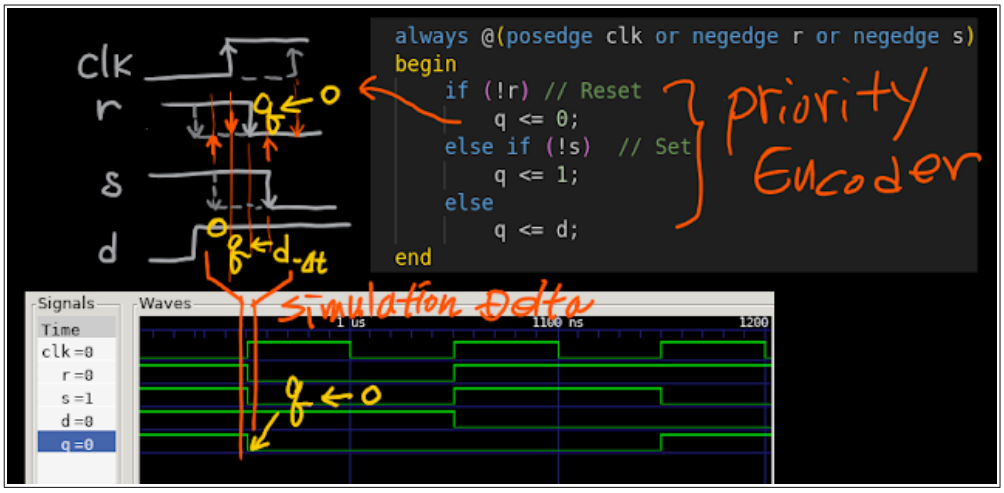
동기 셋과 리셋을 가진 플립플롭을 표준셀로 가진 경우는 흔치 않다. 설계에 동기가 필요할 경우 대부분 입력 d 에 멀티 플렉서를 달아 구현한다.



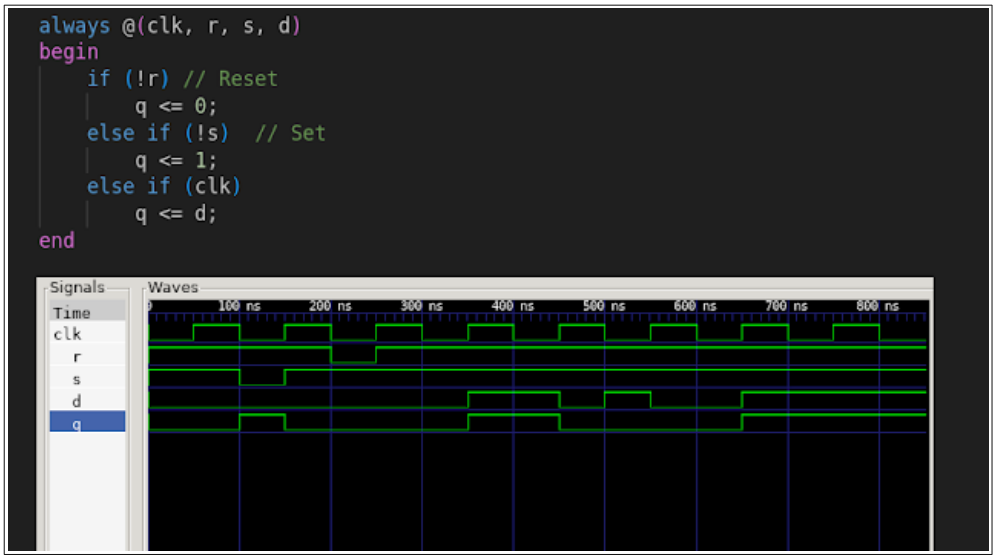
5-5. 시뮬레이션 델타

컴퓨팅 언어는 문법 뿐만 아니라 문장의 실행 방식을 정하고 있다. 문장이 실행된 후 그 결과의 일관성은 매우 중요하다. 특히 병렬 실행문과 순차실행문을 모두 수용하는 하드웨어 기술 언어의 경우 코딩 스타일에 따라 다른 결과를 가져올 수 있고 그 차이는 실행방식의 규정으로 설명 될 수 있어야 한다.

감응 목록에 지정된 다수의 신호가 동시에 시건을 일으킨 경우를 상정해 보자. 각 사건에 대응하여 순차구역이 실행된다. 시뮬레이션 델타 시간 내에서 사건의 순서에 무관한 결과를 얻을 수 있다. 아래의 예에서 clk의 상승 엣지와 r 과 s의 하강 엣지 사건이 동시에 일어 났다. always 구역은 각 사건에 대해 모두 반응할 것이다. 시뮬레이터가 어느 사건을 먼저 처리하든 그 결과는 동일해야 한다. 사건 구동 방식에서 할당과 사건의 수집 그리고 콜-백 그리고 갱신의 과정을 시뮬레이션 델타(simulation delta)라한다. 모든 사건의 처리가 완료되기까지 시뮬레이션 시간은 멈춰있다. 아래의 예에서 감응으로 지정된 신호에 사건이 발생하면 순차구문 if~else 의 논리에 따라 우선순위가 결정(priority encoder)되어 있을 뿐 시뮬레이션은 일관성을 유지한다.



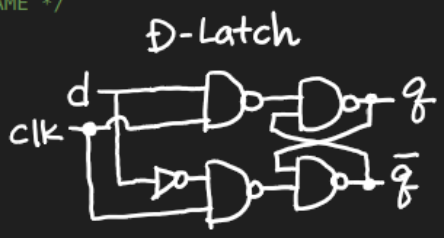
감응 목록이 아래와 같은 경우 시뮬레이션 결과를 설명해 보자. 실제 디지털 회로가 되기 어렵지만 사건구동 시뮬레이션으로 일관성 있는 설명을 해보기 바란다.



5-6. 추상성 수준: 행위 묘사 vs 회로 묘사

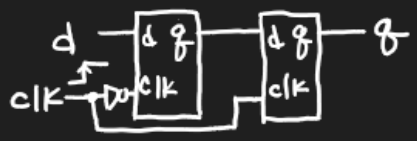
앞서 D-플립플롭을 클럭의 엣지 사건에 반응하는 행위로 기술 했다면 이번에는 논리 연산자를 사용하여 기술해본다. 기초적인 디지털 정보 저장 장치 D-래치를 베릴로그 논리 연산자로 표현하면 아래와 같다. assign은 병렬 할당 구문을 표현한다. 두 병렬 문장에서 할당 왼편과 오른편의 와이어가 서로 교차 결선되어 있지만 하드웨어를 표현한 병렬 실행 구문의 동작은 순서에 무관하다.

```
/* verilator lint_off UNOPTFLAT */
/* verilator lint_off DECLFILENAME */
module d_latch(clk, d, q);
input clk, d;
output q;

wire _q, q_bar, nand0, nand1;

assign q = _q;

assign nand0 = ~(d & clk);
assign nand1 = ~(~d & clk);

assign _q = ~(nand0 & q_bar);
assign q_bar = ~(nand1 & _q);
endmodule
```

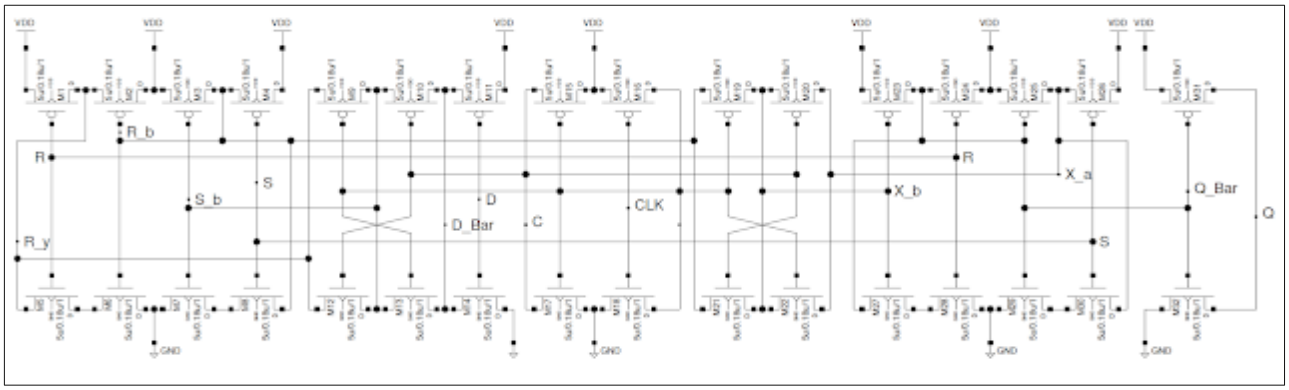


두개의 D-레치를 연속 사용하면 엣지 트리거 방식의 플립플롭을 구현하면 아래와 같다.

```
module dff(clk, d, q);
    input clk,d;
    output q;
    wire _q;

    d_latch master(.d(d),  .clk(~clk), .q(_q));
    d_latch slave (.d(_q), .clk(clk),  .q(q));
endmodule
```

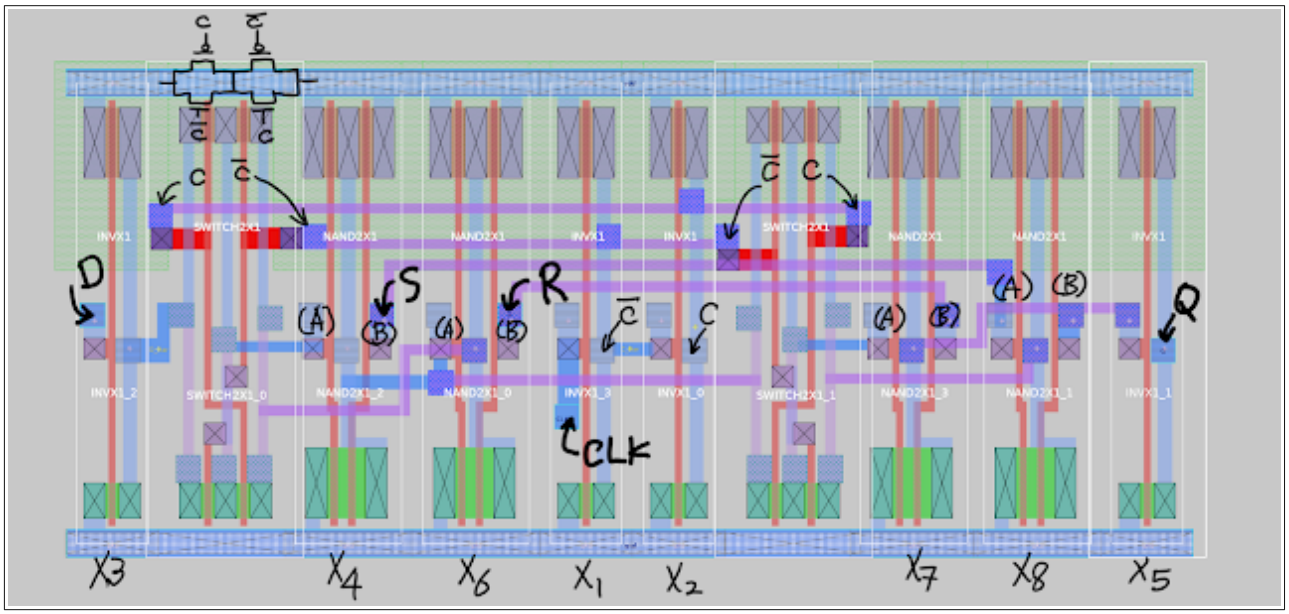


D-플립플롭을 클럭 신호의 상승 엣지 사건에 반응하여 정보를 저장한다는 행위의 표현보다 좀더 실제 전자회로에 가까운 논리 게이트 소자로 기술 하였다. 디지털 정보의 저장장치는 서로다른 두 NAND 게이트의 입출력 포트를 교차 연결하므로서 전류의 지연된 궤환으로 전압을 유지시킨 수 있다는 물리적 현상에 기초한다. 논리소자(게이트)는 다시 트랜지스터의 회로로 표현하면 아래와 같다. 트랜지스터의 조합으로 표현되었지만 여전히 개념적인 표현이다.

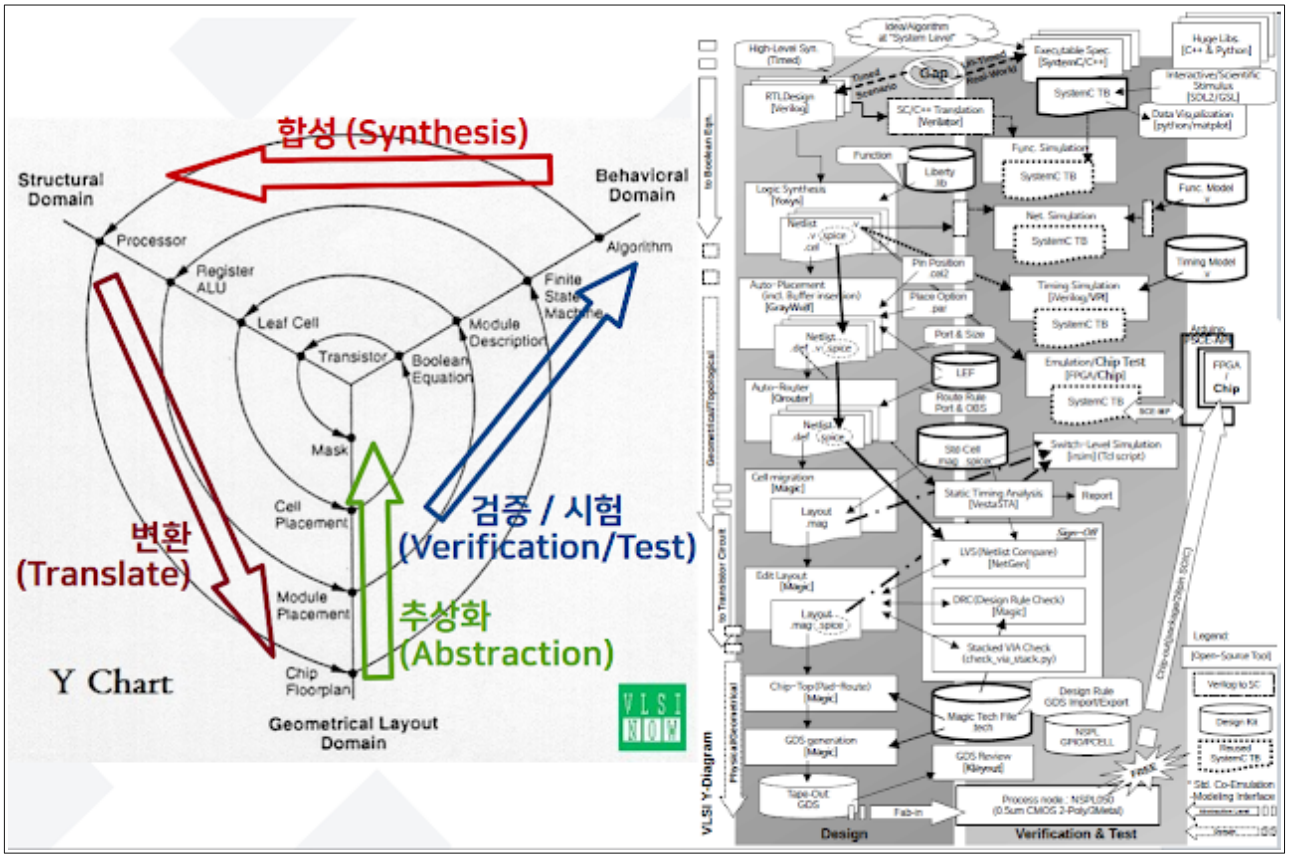


실제 반도체 제조에 사용될 도면(레이아웃)은 아래와 같다. 수십개의 트랜지스터를 사용해야 하는 D-플립플롭을 단 한줄의 할당문으로 기술하므로서 직관적며 높은 설계 생산성을 가질 수 있다. 전자부품의 동작을 행위 수준에서 기술

되었을 때 추상화 수준이 높다고 한다. 실제 제조가능한 도면으로 표현 되었을 때 추상화 수준이 낮다고 한다. 이 구체적인 전자회로는 전류의 흐름을 제어하여 동작을 일으키고 유지되는 전압으로 정보를 담는다.



반도체 설계의 과정은 추상화 수준을 낮추는 과정이다. 추상화 수준의 변환에 반도체 설계 자동화 도구들이 동원된다. 합성기, 자동 배치배선기 등은 컴퓨팅 언어로 기술된 전자회로의 행위를 낮은 추상화 수준으로 변환(또는 합성)해주는 도구들이다. 추상화 수준이 변경되어 얻은 회로도의 등가성 확인은 필수다. 서로다른 추상화 수준으로 표현된 회로의 등가성을 확인하는 절차를 검증이라 한다. 검증에는 HDL 시뮬레이터, SPICE 회로 시뮬레이터 등이 동원된다.



5-7. 테스트벤치 재사용

목적에 따라 동일한 기능(알고리즘)을 상이한 언어 또는 다른 추상화 수준에서 표현할 수 있다. 자동화 도구에 의해 추상화 수준이 변경될 수도 있고 수동으로 변경 될 수도 있다. 어떤 경우든 그 동작의 결과는 동일해야 한다. 검증은 서로다른 두 표현이 낳는 동작의 등가성을 보장하는 절차다. 구문 형식이 다를 뿐 동일한 실행방식을 취하는 소프트웨어 언어에 비해 실행 방식조차 완전히 바뀌는 하드웨어언어를 감안 한다면 변환(또는 합성)의 과정에서 오류가 끼어들 여지는 더욱 다분하다. 이점에서 변환(translation)과 합성(synthesis)의 차이를 감지할 수 있다. 자동화 도구는 설계 생산성을 높여주고 인간의 오류를 줄여주는데 획기적으로 기여하지만 완벽하다고 장담할 수 없으므로 반드시 검증의 과정이 필요하다. 검증의 가장 확실한 방법은 설계를 시험하기 위해 마련된 테스트벤치를 활용한 시뮬레이션이다. 추상화 수준이 변경 되었더라도 동일한 테스트벤치를 사용하는 것이 원칙이다. 다양한 이유로 추상화 수준이 달라진 경우 다른 테스트벤치를 요구하기도 한다. 별도의 테스트벤치를 작성하는 과정은 매우 비생산적일 뿐만 아니라 유연성이 낮아 불완전한 검증으로 이어질 가능성이 매우 높다. 앞서 살펴본 대로 하드웨어를 표현할 수 있는 다양한 크래스 라이브러리를 갖추고 있는 SystemC는 높은 시스템 수준의 알고리즘에서 RTL 까지 매우 폭넓은 추상화 수준을 포용한다.

행위를 묘사한 D-플립플롭을 시험하기 위해 작성한 테스트벤치를 게이트 수준의 표현에서도 변경 없이 동일하게 적용할 수 있다. 추상화 수준을 넘나들며 테스트벤치를 재사용하므로서 검증의 품질을 한층 높일 수 있음은 자명하다. 테스트벤치 재사용을 D-플립플롭 예제를 통해 살펴보자. 디자인 킷 예제에서 게이트 수준으로 묘사한 D-플립플롭 디렉토리로 이동한 후 파일 목록을 보면 별도의 테스트 벤치는 없다.

```
$ cd ~/ETRI050_DesignKit/devel/Tutorials/2-3_Lab1_dff/dff_gate
$ ll
total 24
drwxr-xr-x 2 goodkook goodkook 4096 Jul  4 16:50 ./
drwxr-xr-x 6 goodkook goodkook 4096 Jul  3 22:08 ../
-rw-r--r-- 1 goodkook goodkook  894 Jul  3 22:13 dff.v
-rw-r--r-- 1 goodkook goodkook 1636 Jul  3 22:09 Makefile
-rw-rw-r-- 1 goodkook goodkook  622 Jun 21 19:02 sc_dff_TB.gtkw
-rw-rw-r-- 1 goodkook goodkook 1131 Jul  3 22:49 Vdff.gtkw
```

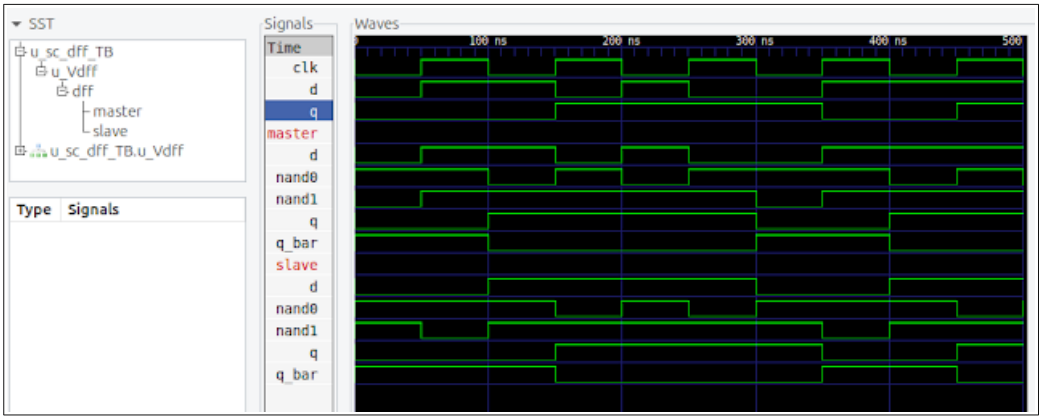
다음은 Makefile의 일부다. 행위 묘사 D-플롭플롭을 시험하기 위해 사용했던 SystemC 테스트 벤치 ../dff/sc_dff_TB.h 와 ../dff/sc_main.cpp 를 재사용하고 있다.

```
VERILOG_SRCS = dff.v
SC_SRCS      = ../dff/sc_main.cpp
SC_TOP_H     = ../dff/sc_dff_TB.h
VERILATOR    = verilator
CFLAGS       = -g
    ......
build : $(TARGET_DIR)/$(TARGET)

$(TARGET_DIR)/$(TARGET) : $(VERILOG_SRCS) $(SC_SRCS) $(SC_TOP_H)
    $(VERILATOR) --sc -Wall --trace \
        --top-module $(TOP_MODULE) --exe --build \
          -CFLAGS $(CFLAGS) $(VERILOG_SRCS) $(SC_SRCS)
```

게이트 수준 D-플립플롭 시뮬레이터를 빌드하고 실행하여 VCD 파형을 보면 행위 모델과 동일한 결과를 보여줄 것이다. 추상화 수준이 다른 두 모델의 등가성을 한 테스트벤치를 통해 확인 할 수 있다.

```
$ make build
verilator --sc -Wall --trace --top-module dff --exe --build \
          -CFLAGS -g \
          dff.v ../dff/sc_main.cpp
    ......
make[1]: Leaving directory......
- V e r i l a t i o n   R e p o r t: Verilator 5.037 ....
- Verilator: Built from 0.022 MB sources in 3 modules,....
- Verilator: Walltime 0.347 s (elab=0.000, cvt=0.006, bld=0.337)....


$ make run
./obj_dir/Vdff
    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_dff_TB.vcd)
    Info: /OSCI/SystemC: Simulation stopped by user.
```

```
$make wave
```



5-8. 설계 생산성을 높이는 팁: FOR 반복문과 `define 매크로

컴퓨팅 언어를 사용하여 알고리즘을 기술할 때 생산성 향상을 위해 여러가지 기법이 동원 된다. 대표적으로 반복적인 구문을 위해 사용하는 for-반복문이 있다. 그외 재사용성을 높이기 위해 define 매크로를 활용하고 조건부 컴파일 기법을 동원한다. 베릴로그 하드웨어 기술 언어도 그와 동일한 기법을 적용 할 수 있다. 다음은 쉬프트 레지스터를 기술한 예다. 다단 쉬프트 레지스터를 묘사하기 위해 for-반복문을 사용했다. 쉬프트 단수를 `define 매크로에 정의해 둠으로써 필요에 따라 쉽게 조정할 수 있는 유연성을 갖추고 있다.

```
// filename: shifter.v
`define NUM_REG 4
`define BIT_WIDTH 8

module shifter(clk, rst, din, qout);
input clk, rst;
input [`BIT_WIDTH-1:0] din;
output [`BIT_WIDTH-1:0] qout;

reg qout;
reg [`BIT_WIDTH-1:0] x[`NUM_REG];

always @(posedge clk or negedge rst) // edge trigger, Async rst
begin
    if (!rst) begin // Reset
        for (integer i = 0; i < `NUM_REG; i++)
            x[i] <= 0;
    end else begin
        for (integer i = 1; i<`NUM_REG; i++)
            x[i+1] <= x[i];
        x[0] <= din;
        qout <= x[3];
    end
end

endmodule
```

6. 맺음말

Verilog로 설계하고 SystemC로 검증 하는 설계 방법론을 간단한 D-플립플롭의 예를 들어 소개했다. 트랜지스터의 회로에 비하여 추상화 수준을 비교해 볼 수 있을 것이다. 설계자의 안목 또한 중요한 요소가 된다. 반도체 설계에 컴퓨팅 언어를 사용하므로써 얻는 장점이 많다. 컴퓨팅 언어 기반의 설계 방법론이 성숙되어 알고리즘 개발자도 높은 추상화 수준에서 반도체(하드웨어)를 쉽게 시작할 수 있다. 하지만 최종 목표가 전자회로라는 점을 인식하고 있어야 한다. Verilog 도 컴퓨팅 언어다. 컴퓨팅 언어로는 고도의(혹은 기이한 트릭) 행위의 표현이 가능 하지만 실제로 트랜지스터 회로로 전환 될 수 없을 수도 있다.

하드웨어의 행위를 묘사하는 설계와 더블어 검증 또한 매우 중요하다. 제대로 검증되지 않은 설계가 하드웨어로 구현 되어 일으킬 손실은 막대하다는 점은 굳이 반도체 뿐만은 아니다. 검증은 설계보다 높고 넓은 추상성을 갖춰야 한다는 점도 이해했을 것이다. 시뮬레이션 소프트웨어에서 병렬성을 구현하는 방법을 간략히 살펴봤다. 코딩 스타일에 따라 시뮬레이션의 결과가 달라질 수 있는 이유를 설명 할 수 있을 것이다. 시뮬레이션 소프트웨어가 병렬성을 일관성있게 처리하는 방식을 알면 시스템 모델링에 큰 도움이 된다.

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] "VLSI 레이아웃 설계 기초" [8] Std-Cell 제작 실습: DFF-SR, https://fun-teaching-goodkook.blogspot.com/2024/07/vlsi-8-std-cell-dff-sr.html

[8] GNU Make강좌, https://doc.kldp.org/KoreanDoc/html/GNU-Make/GNU-Make.html

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