1장 디지털 시스템 설계 개요

1.1 개요

디지털 회로의 설계는 예전에는 회로도 작성과 수작업으로 이루어졌다. 그리고 소규모 또는 중간규모의 집적회로(SSI/MSI)로 구현된 기본 디지털 IC들을 사용하여 이들을 인쇄회로기판(PCB) 위에서 연결하여 구현하였다. 대규모집적회로(VLSI) 기술이 등장함에 따라서 단일 칩 위에서 수십만개 또는 수백만개 이상의 트랜지스터가 들어가는 회로를 구현할 수 있게 되어서 복잡한 디지털 시스템을 단일 칩에 구현하는 것이 가능하게 되었다. 회로의 크기와 복잡도가 증가함에 따라서 회로의 설계와 검증이 매우 중요한 요소가 되었으며 이전의 방법으로는 설계와 검증을 하기가 매우 어려워졌다.

디지털 시스템 설계 영역에서도 컴퓨터 프로그램을 작성하는 것처럼 회로도(schematic)가 아닌 표준 언어를 사용하여 디지털 회로를 기술해야할 필요성을 깨닫게 되었으며 이에 따라서 하드웨어 기술 언어(hardware description language: HDL)가 등장하게 되었다. HDL의 등장은 매우 복잡하고 큰 디지털 회로의 설계와 검증을 편리하게 할 수 있도록 하였다.

HDL을 사용한 회로 설계는 이식성이 있으며 구현 기술과 무관하게 설계를 할 수 있다. 즉 구현기술 독립적이다. 이러한 특징은 이전에 수행한 설계에 새로운 구현 기술을 쉽게 적용할 수 있도록 한다. HDL을 지원하는 설계 도구들은 HDL로 기술한 회로의 동작을 시뮬레이션을 통하여 검증해주고 논리 간소화 등의 번거로운 과정을 자동적으로 수행하여 최적화된 게이트 수준의 회로로 자동적으로 합성(synthesis)하여 줌으로써 설계자들이 트랜지스터나 게이트 수준의 회로 설계가 아닌 회로의 기능과 동작에 초점을 맞추어서 설계를 할 수 있도록 하였다.

1.2 하드웨어 기술 언어(HDL)

현재 널리 사용되고 있는 HDL로는 *Verilog*와 *VHDL*이 있다. Verilog는 1983년 Gateway Design Automation사에서 개발되었으며 1995년에 IEEE 표준(IEEE-1364)으로 지정되었고 2001년과 2005년에 개정되었다. VHDL은 1983년에 DARPA의 지원에 의해서 개발되었으며 1987년에 IEEE 표준(IEEE-1076)으로 지정되었고 1993년, 2000년, 2002년과 2008년에 개정되었다. VHDL은 VHSIC(Very High Speed IC) Hardware Description Language의 약어이다. Verilog나 VHDL로 기술된 대규모 디지털 회로에 대한 시뮬레이터가 개발됨에 따라서 이 두 언어는 설계자들로부터 매우 빠른 속도로 호응을 받을

수 있었다. 이 이외에도 하드웨어 설계에 사용되는 대표적인 언어로서 *Spice*와 *System C*가 있다. Spice는 아날로그 회로를 설계하고 파형을 시뮬레이션하는 데 사용되며 대규모 시스템의 설계에는 적합하기 않다. System C는 C언어를 확장한 언어로서 시스템 설계에 적합하며 Verilog과 VHDL보다 추상적인 모델링이 가능하다.

초기에는 회로의 동작을 검증하는 것은 시뮬레이터에 의해서 수행되었으나 최종적인 게이트 수준의 회로 설계는 수작업에 의존해야 했다. 1980년대 후반에 논리합성(synthesis) 방법이 등장하여 설계 방법론을 급격히 바뀌었다. HDL을 사용하여 레지스터 전송 수준(register transfer level: RTL)에서 회로를 기술하면 논리합성 도구에 의해서 게이트 수준의 표현으로 자동적으로 변환하는 것이 가능하게 되었다. 논리합성의 등장은 디지털 시스템 설계에 HDL이 필수적인 도구로서 자리 잡게 하였으며 설계자가 게이트들에 대한 세세한 설계를 더 이상 직접 하지 않아도 되게 되었다.

회로를 HDL로 설계함으로써 동작을 프로그래밍 언어처럼 추상화된 수준에서 기술할 수 있도록 하여 하드웨어 설계에 대한 복잡도를 현저하게 감소시켰다. HDL은 시스템 수준의 설계에도 사용되기 시작했다. FPGA, ASIC 등을 위한 회로 설계에 HDL을 사용하며 더 나아가서 이들을 서로 연결하여 구성된 시스템의 동작을 검증하는 데에도 HDL을 사용할 수 있다.

1.3 설계 절차

일반적인 VLSI의 설계 과정은 그림 1-1과 같다. 설계는 회로의 명세를 작성하는 것으로 시작된다.

설계 명세

설계의 명세(specification)는 회로의 기능과 외부 인터페이스 등을 추상적으로 표현한 것이다. 회로의 기능은 조합회로는 부울 함수, 진리표로 표현할 수 있으며 순차회로는 상태전이 그래프(state transistion graph), 타이밍도(timing diagram), 알고리즘 상태도(algorithmic state machine: ASM) 등으로 나타낼 수 있다. 설계 명세에는 타이밍, 칩 면적, 전력소모, 테스트능력 등을 포함할 수 있다.

설계 분할

큰 회로는 여러 개의 작은 회로로 분할(partition)하여 계층적인 설계를 할 수 있으며 이 때에 큰 회로는 분할된 작은 회로들로 구성된다. 분할된 회로는 전체 회로보다 간단하기 때문에 설계하고 검증하기가 쉽다.

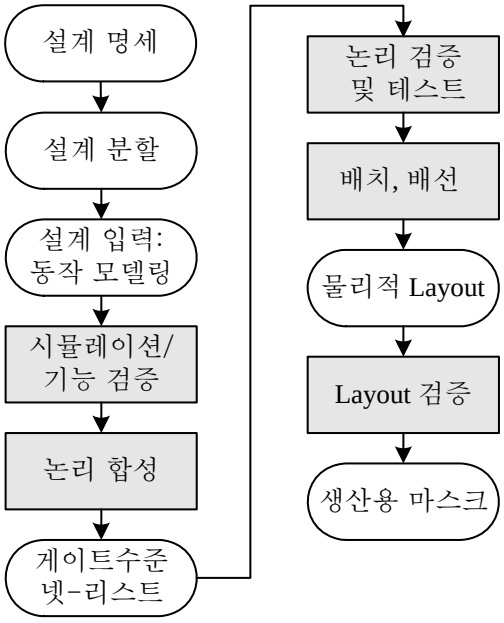


그림 1-1 HDL을 사용한 IC 설계 과정

설계 입력

설계 입력(entry) 단계에서 회로를 HDL을 사용하여 표현한다. 회로의 설계는 대개 회로를 구체적으로 게이트 수준에서 기술하는 방법 보다는 회로의 동작을 추상적으로 기술하는 방법을 많이 사용한다. 회로를 레지스터 전송 수준(RTL)에서 기술하는 것이 일반적인 설계 방법이다.

회로의 설계를 기술하는 방법에는 구조적 기술(structural description) 방법과 동작적 기술(behavioral description) 방법으로 나뉜다. **구조적 기술** 방법은 회로도와 같이 구성 요소와 이들의 연결로서 회로를 기술하는 방법이다. 회로가 복잡해질수록 구조 기술 방법으로 설계하기가 매우 어려워지며 특히 게이트 수준의 회로를 구조적 기술 방법으로 설계하는 것은 더욱 힘들다. **동작적 기술** 방법은 회로의 구조 대신에 동작을 기술하는 것으로서 회로가 어떻게 구현되는 지가 아니라 회로가 어떠한 일을 수행하는 지를 나타내는 것이다. 동작수준의 기술 방법으로 설계를 할 때에 세부적인 게이트 수준의 설계를 직접 수행할 필요가 없으며 게이트 수준의 설계는 논리합성 도구에 의해서 이루어진다.

시뮬레이션 및 기능 검증

설계된 회로의 기능(function)을 시뮬레이션에 의해서 검증한다. 검증은 (1) 테스트

계획 수립 (2) 테스트벤치(testbench) 개발 (3) 테스트 수행의 과정을 포함한다. 이 과정에서 설계의 잘못을 바로잡는다.

논리 합성 및 기술 맵핑

논리합성 도구는 논리 최적화 과정을 거쳐서 게이트들과 그들의 연결을 나타내는 게이트 수준의 넷리스트(net list)로 변환한다. 최근의 합성도구들은 동작적 기술 또는 RTL 수준의 HDL 표현을 최적화된 넷리스트로 합성해 준다. 합성 과정에서 넷리스트를 생성할 때에 구현할 ASIC의 표준 셀 라이브러리 또는 FPGA의 구성 정보가 이용된다.

합성후 검증

논리 합성된 게이트 수준의 회로에 대해서 동작 및 타이밍을 검증하여 합성 전의 검증 과정에서 발견되지 않았던 잘못을 바로잡는다.

배치, 배선 및 설계 규칙 검증

게이트들의 배치(placement)와 이들 간에 연결에 대한 배선(routing)을 도구를 사용하여 자동적으로 수행한 후에 물리적, 전기적 설계 규칙에 위배되는 지 검증을 한 후에 최종적인 칩 레이아웃(layout)을 완성한다.

구현

완성된 칩 레이아웃을 VLSI로 구현한다.

회로의 설계 과정에서 CAD 도구가 매우 커다란 역할을 한다. 설계의 검증과 합성 그리고 최종적인 칩 레이아웃을 완성하는 데 이르기까지 모두 CAD 도구에 의존한다. 특히 논리 합성 기술의 발전은 과거에 몇 년이 소요되는 설계 주기를 불과 몇 개월로 단축시켰다.

1.4 HDL의 중요성

HDL은 전통적인 회로도 기반의 설계에 비해서 다음과 같은 많은 장점을 가지고 있으며, 앞으로 CAD도구의 발전과 함께 대규모 디지털 시스템을 설계하는 유일한 방법이 될 것이다.

첫째, 설계를 특정한 제조기술에 관계없이 할 수 있고, 게이트 수준의 회로를 직접 설계하는 것 대신에 추상화된 수준에서 표현할 수 있기 때문에 설계하기가 쉽다. 논리합성도구가 회로의 동작을 추상화하여 기술한 것을 제조기술에 맞는 게이트 수준의

회로로 자동적으로 변환해주기 때문이다. 그리고 논리 합성 과정에서 회로를 최적화 해준다.

둘째, 회로의 기능 검증을 최종 구현 단계가 아닌 설계 초기에 끝낼 수 있다. 거의 모든 오류를 이 단계에서 없앨 수 있다. 게이트 수준의 넷리스트나 물리적인 레이아웃에서의 기능 오류를 발생시킬 확률을 최소화한다.

셋째, HDL로 회로를 기술하는 것은 컴퓨터 프로그래밍과 유사하다. 게이트 수준의 도면에 비해서 매우 간결한 표현 방법을 제공한다. 복잡한 설계에서 게이트 수준의 도면은 거의 알아보기가 어렵다.

1.5 Verilog의 특징

Verilog는 IEEE 표준으로 지정된 하드웨어 기술 언어로서 하드웨어 설계에 매우 유용한 다음과 같은 특징들을 제공한다.

- 배우고 사용하기 쉬운 하드웨어 기술 언어이다. 문법이 C언어와 유사하므로 C언어에 대한 경험이 있는 설계자는 배우기가 쉽다.
- 하나의 회로 모델 내에서 여러 추상화 수준의 기술 섞어서 모델링할 수 있다. 즉, 트랜지스터 스위치, 게이트, RTL 또는 더 상위 수준의 동작을 기술하는 코드를 혼합하여 사용하여 설계를 할 수 있다.
- 시뮬레이션을 위한 스티뮬러스(stimulus, 외부 입력)를 제공할 수 있으므로 외부 테스트 입력 제공 및 계층적인 설계를 위해서 Verilog만 배우면 충분하다.
- 대부분의 논리합성 도구들은 Verilog로 표현된 회로의 합성을 지원한다.
- 모든 제조업체들이 논리합성 시뮬레이션을 위한 Verilog 라이브러리를 제공한다.
- PLI(Programming Language Interface) 기능이 제공되어서 Verilog 내부 자료구조와 상호작용을 하는 C프로그램을 사용할 수 있도록 한다. 설계자는 PLI를 이용해 그들의 필요에 맞도록 Verilog 시뮬레이터를 조정할 수 있다.

1.6 IC 구현 기술

Verilog로 설계된 하드웨어는 그림 1-2와 같은 여러 가지 형태의 IC로 구현될 수 있으며 소요 수량, 개발 비용, 개발 기간, 필요 성능과 칩 면적 등의 사항을 고려하여 구현하는 IC 형태를 선택한다.

완전주문형(full-custom) IC는 회로의 배치와 배선을 직접 설계하는 방식으로서 성능을 최대화 하고 칩 면적을 최소화할 수 있으며 대량 생산 시에 제조 단가가 적게 든

다. 그렇지만 NRE(non-recurring engineering) 비용이라고도 불리는 개발 비용이 많이 들며 개발 기간이 많이 걸리고 설계를 위한 고급 전문 인력이 필요한 단점이 있다.

표준셀(standard cell)을 사용한 IC는 미리 설계되어 라이브러리에 저장된 표준 셀을 사용하고 이들 간의 배선을 최소화하도록 셀들을 배치하여 구현한다. 완전주문형 IC 보다 개발 비용을 적고 개발 기간을 단축시킬 수 있다.

게이트 배열(gate array)은 NAND와 NOR과 같은 기본적인 논리 게이트나 표준 논리 소자를 규칙적으로 배열한 배선 이전의 공정이 끝난 칩을 웨이퍼(wafer)상태로 보관한 다음에 설계에 따라서 배선 및 이에 따른 마스크 공정을 수행한다. 설계 비용 및 기간이 완전주문형이나 표준 셀 방식에 비해서 작지만 칩 면적을 비효율적으로 사용하고 전력 소모가 크고 속도가 떨어진다

프로그램 논리소자(programmable logic device: PLD)는 고정된 기본 구조를 가지고 있지만 사용자 요구에 맞게 연결 상태를 선택하여 하드웨어 구조를 프로그램하여 사용할 수 있는 논리소자이다. 설계 비용과 설계 기간이 짧으며 하드웨어 프로그래밍 과정을 통하여 하드웨어를 빠르게 구현할 수 있다. 그렇지만 집적도가 낮아서 칩 면적의 사용이 비효율적이며 개별 단가가 비싼 단점이 있다. 프로그램 논리소자로는 낮은 게이트 수를 갖는 PAL, PLA 등이 있으며 많은 게이트 수를 갖는 FPGA(field programmable gate array)와 CPLD(complex PLD) 등이 있다.

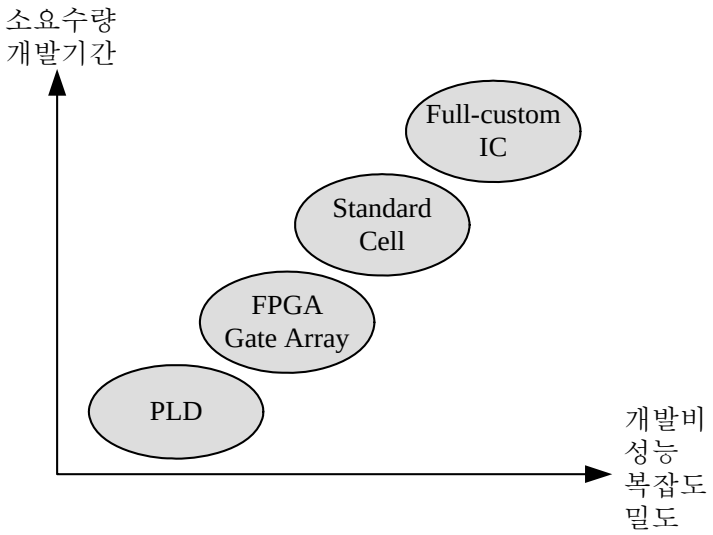


그림 1-2 여러 가지 IC구현

2장 Verilog 논리 설계의 기초

2.1 간단한 논리회로에 대한 Verilog 모델

간단한 논리회로를 Verilog로 나타내는 것으로서 디지털 시스템의 Verilog를 사용한 설계에 대한 소개를 시작하기로 하자. 그림 2.1은 두 개의 1비트 입력 a, b에 대한 합 sum과 캐리 cout을 출력시키는 반가산기(half adder) 회로이다.

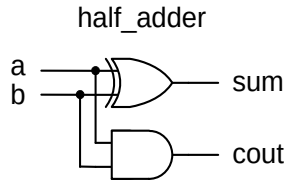


그림 2.1 반가산기 회로

이 반가산기 회로는 Verilog를 사용하여 예 2.1과 같이 표현된다.

예 2.1 반가산기

```
module halfadder(sum, cout, a, b);      // 회로(모듈)의 이름과 입출력 포트 이름
    input   a, b;           // 입력 포트 선언
    output  sum, cout;      // 출력 포트 선언

    xor (sum, a, b);        // XOR 게이트
    and (cout, a, b);       // AND 게이트
endmodule
```

이 Verilog 모델에 대한 설명은 다음과 같다.

1. 논리회로에 대한 Verilog 모델은 모듈로서 표현되며 모듈은 module로 시작하여 endmodule로 끝난다.
2. half_adder는 모듈 이름이며 괄호 안의 (sum, cout, a, b)는 모듈을 외부와 연결하는 입출력 포트들의 이름을 나타낸다. 그리고 괄호 다음에 콜론을 사용하여 module 선언을 끝낸다.

3. input과 output은 입출력 포트들의 방향을 선언한다. input으로 선언된 a와 b는 입력 포트이고 output으로 선언된 sum과 cout은 출력 포트이다.
4. xor와 and는 Verilog의 프리미티브(primitive)로서 XOR게이트와 AND게이트를 각각 나타낸다. 프리미티브 다음의 괄호 안에는 게이트의 입출력과 연결되는 신호들의 이름을 나열하는 데 게이트 프리미티브는 출력을 먼저 쓰고 그 다음에 입력들을 컴마로 구분하여 나열한다. 그림 2.1의 신호의 연결과 예 2.1의 Verilog 표현을 비교하면 쉽게 이해될 것이다.

2.2 기본적인 Verilog 언어 규칙

Verilog는 C언어와 매우 유사하며 다음과 같은 기본적인 규칙을 가지고 있다.

1. 대소문자를 구별하는 언어이다. 예를 들어, Cout과 COUT은 같지 않다.
2. 모듈, 신호 등의 이름으로 사용되는 식별자(identifier)는 알파벳(a-z, A-Z), 숫자(0-9), 밑줄문자(_), $ 문자로 구성된다. 단, 숫자와 $로 시작할 수 없다. 그리고 길이는 1024문자까지 가능하다.
3. Verilog의 각 문장은 대개 세미콜론으로 끝난다. 예외적으로 키워드 `endmodule`는 세미콜론으로 끝나지 않는다.
4. C언어와 같이 두개의 연속된 슬래쉬 `//`는 줄 끝까지가 주석(comment)임을 나타내어 줄 단위의 주석에 사용되며 `/*`와 `*/` 는 주석의 시작과 끝을 나타내어 여러 줄의 주석을 기술하는 데에 사용될 수 있다.

2.3 논리게이트와 프리미티브

Verilog는 논리회로를 구성하는 기본 요소인 게이트들에 대한 모델을 **프리미티브**(primitive)로서 미리 정의하여 제공한다. 프리미티브는 Verilog의 가장 기본적인 요소이다. Verilog에는 26개의 프리미티브가 정의되어 있는 데 이들 중에서 많이 사용되는 프리미티브는 표 2.1과 같다.

표 2.1 Verilog 프리미티브

n 입력 1 출력 게이트	`and, nand, or, nor, xor, xnor`
1 입력 n 출력 게이트	`buf, not`
1 입력 n 출력 게이트 (3 상태)	`bufif0, notif0` (제어입력이 0일 때에 enable) `bufif1, notif1` (제어입력이 1일 때에 enable)

논리회로를 게이트들과 이들의 연결로 구성된 회로도로 나타내는 것처럼 Verilog 프리미티브들을 사용하여 논리회로를 표현할 수 있다.

논리 게이트

논리 게이트용 프리미티브는 임의의 개수의 입력을 연결할 수 있으며 첫 번째 포트가 출력이다. 그림 2.2는 2입력 NAND 게이트와 4입력 NOR 게이트의 심볼와 이에 대한 Verilog 문장을 보여준다. 게이트 프리미티브가 어떤 모듈에서 사용되었을 때에, 사용된 게이트를 프리미티브의 **인스턴스**(primitive instance)라고 부른다. 프리미티브와 프리미티브 인스턴스의 관계는 객체지향 프로그램에서 클래스(class)와 객체(object)와의 관계와 비슷하다. 회로를 설계할 때에 같은 프리미티브의 인스턴스를 여러 개 사용할 수 있다. Verilog에서는 각 인스턴스를 구별하기 위해서 그림 2.2의 nand와 같이 프리미티브 이름 다음에 **인스턴스 이름**을 쓰는 것이 일반적이다. 그렇지만 프리미티브 인스턴스의 경우에는 그림 2.2의 nor와 같이 인스턴스 이름을 생략해도 무방하다. 인스턴스 이름은 같은 모듈 내에서는 고유한 이름이어야 한다.

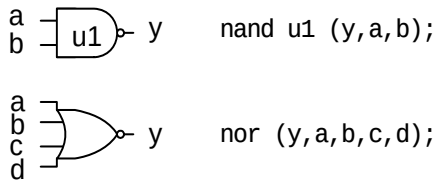


그림 2.2 n 입력 게이트

버퍼와 인버터

`buf`와 `not`은 각각 **버퍼**(buffer)와 **인버터**(inverter)를 나타내는 프리미티브이다. 버퍼는 출력의 논리 값이 입력과 같지만 팬 아웃(fan out)을 증가시키는 용도로 주로 사용된다. 버퍼와 인버터는 원래 입력과 출력이 1개이지만 프리미티브 `buf`와 `not`은 같은 값을 갖는 여러 개의 출력을 사용할 수 있으며 물리적으로 팬 아웃을 증가시키는 용도로 사용할 수 있다. 그림 2.3은 인버터와 버퍼의 회로도와 이에 대한 Verilog 문장을 보여준다. `buf`와 `not` 역시 출력 포트들을 먼저 나열하며 마지막에 입력을 쓴다.

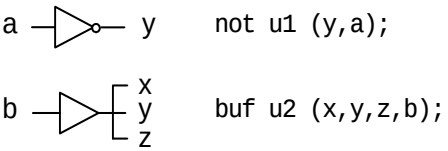


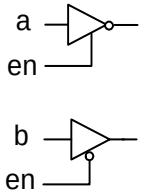
그림 2.3 인버터와 버퍼

3상태 버퍼와 인버터

일반적으로 논리회로의 출력은 0 또는 1의 값을 제공한다. 일부 논리회로는 이러한 출력 값 이외에 출력과 외부와의 연결이 끊어진 것과 같은 개방(open) 상태의 출력을 제공하기도 한다. 이러한 개방된 출력 상태를 **고임피던스**(Hi-Z 또는 Z) **상태**라고 하고 고임피던스 출력을 제공하는 출력을 **3상태 출력**(tri-state output)이라고 한다. 3상태 출력을 가진 회로는 출력과 외부와의 연결을 제어하는 제어 입력이 포함되며 회로의 출력이 외부와 연결되는 경우를 **인에이블**(enable)되었다고 하고 그렇지 않은 경우를 **디스에이블**(disable)되었다고 한다.

3상태 출력은 버스(bus)와 같이 여러 개의 출력이 물리적으로 함께 연결되는 경우에 주로 사용된다. 버스 신호는 물리적으로는 여러 개의 출력이 서로 연결되지만 하나의 출력을 제외한 나머지 출력이 디스에이블 상태가 되게 하면 실질적으로 하나의 출력만 버스에 연결되게 된다.

`bufif0`와 `bufif1`, `notif0`와 `notif1`은 3상태 출력을 제공하는 버퍼와 인버터를 나타내는 Verilog 프리미티브로서 buf와 not과 마찬가지로 같은 값을 갖는 여러 개의 출력을 사용할 수 있다. 프리미티브 이름의 0과 1은 인에이블 상태의 제어입력 값을 나타낸다. 그림 2.4는 1-인에이블 3상태 인버터 및 0-인에이블 3상태 버퍼와 이에 대한 Verilog 문장을 보여준다. 3상태 버퍼와 인버터 프리미티브의 포트들은 출력들, 입력, 제어입력 순서로 나열된다.




```
notif1 u1 (y,a,en);
bufif0 u2 (y,b,en);
```

그림 2.4 3상태 인버터와 버퍼

2.4 와이어

논리회로는 게이트들을 여러 단계로 연결하여 구현될 수 있으며 내부 게이트들의 입출력들이 서로 연결된다. 그림 2.5가 이러한 종류의 논리회로의 예이다. 이러한 요소들 간의 연결선을 **넷**(net)라고 한다. 넷에 연결되는 입력은 넷에 연결된 출력 값을 가지게 된다.

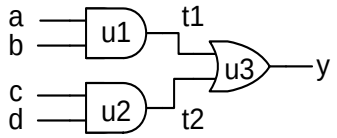


그림 2.5 내부 연결선이 포함된 논리회로

넷은 대부분 키워드 wire에 의해서 정의된다. 넷에는 wire 이외에도 넷의 물리적 특성에 따라서 키워드 wand, wor, tri, triand, trior, trireg 등이 사용된다. wand와 wor는 각각 wired AND와 wired OR 특성의 출력에 연결되는 넷에 사용되며, tri 및 tri로 시작되는 넷 유형은 3상태 출력에 연결되는 넷에 사용된다.

예 2.2는 그림 2.5의 회로에 대한 Verilog 모델이다. 이 회로는 내부에 2개의 연결선이 있으며 이 연결선을 나타내기 위해서 2개의 wire 신호 t1과 t2를 선언하여 사용하였다.

예 2.2 wire를 포함한 Verilog 모델

```
module circuit1(y, a, b, c, d);
   input   a, b, c, d;          // 입력 포트 선언
    output  y;                  // 출력 포트 선언
    wire    t1, t2;             // 와이어 선언 - 내부 연결선

    and u1 (t1, a, b);          // AND 게이트
    and u2 (t2, c, d);
    or u3 (y, t1, t2);          // OR 게이트
endmodule
```

2.5 모듈 포트

모듈 포트(port)는 모듈 외부와의 인터페이스이다. 포트의 이름은 모듈 이름 다음의 괄호 안에 정의된다. 정의된 모든 포트들은 바로 뒤이어 키워드 input, output, inout을 사용하여 포트의 방향을 선언한다. input은 입력 포트를, output은 출력포트를, inout은 입력과 출력 특성을 함께 갖는 양방향 포트를 나타낸다. 모듈 포트의 방향을 선언하는 순서는 포트 이름이 정의된 순서와 같지 않아도 된다. 예를 들어서 다음의 두 선언은 서로 같은 것이다.

```
module circuit2(y, a, b);
    output y;
    input a, b;
    . . .
endmodule
```

```
module circuit2(y, a, b);
    input a, b;
    output y;
    . . .
endmodule
```

모듈 포트는 별도의 선언이 없으면 기본적으로 wire로 취급된다. input과 inout은 일반적으로 추가적인 선언 없이 wire로 사용되지만 output은 경우에 따라서 나중에 배우게 될 reg로 선언하는 경우가 많이 있다.

Verilog-2001에서는 예 2.3와 같이 포트 이름을 포트 방향과 함께 선언하는 ANSI C 언어 스타일로도 모듈의 입출력 포트를 선언할 수 있다. 입출력 포트 선언은 예 2.3의 두 번째 코드와 같은 스타일로 입출력 포트를 줄을 구분하여 나타내기도 한다. Verilog-2001이 새로운 형식의 포트 선언을 추가했을지라도 이 책에서의 포트 선언은 예전 스타일을 사용할 것이다.

예 2.3 반가산기 (Verilog 2001 스타일의 모듈 포트 선언)

```
module halfadder(output sum, cout,  input a, b);
    xor (sum, a, b);        // XOR 게이트
    and (cout, a, b);       // AND 게이트
endmodule
```

```
module halfadder(
    output sum, cout,
    input a, b
);
    xor (sum, a, b);        // XOR 게이트
    and (cout, a, b);       // AND 게이트
endmodule
```

2.6 계층적인 설계와 모듈

디지털 시스템 설계는 시스템이 복잡해짐에 따라서 계층적인 설계를 하는 것이 일반적이며 설계 방법에는 **하향식**(top-down) 방법과 **상향식**(bottom-up) 방법이 있다. 하향식 방법은 전체 시스템을 설계와 검증이 용이한 블록들로 나눈 다음에 블록들을 상호 연결하여 전체 시스템을 설계한다. 나누어진 블록들은 더 간단한 작은 블록으로 나누어

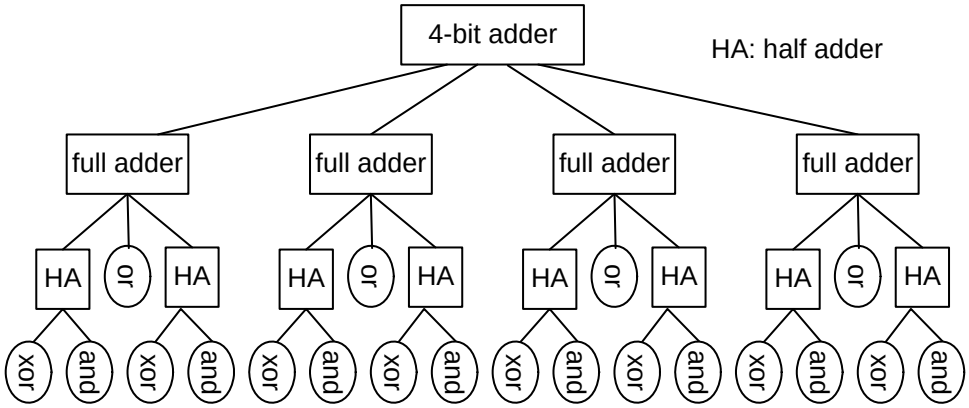


그림 2.6 4비트 가산기의 계층적 설계

서 앞에서와 마찬가지 방법으로 설계를 하며 이러한 분할 과정은 더 이상 나누어지지 않을 때까지 반복하여 수행한다. 더 이상 나누어지지 않는 블록이 프리미티브(primitive)이다. 상향식 방법은 먼저 필요한 기본 블록을 만들고 이들을 이용하여 더 큰 블록을 만든다. 이러한 방법으로 최상위 블록까지 설계한다. 실제 설계에서는 두 가지 방법이 혼용된다. 하향식 설계 방법을 사용하여 설계를 하되 많이 사용되는 기본적인 블록들은 미리 설계되기도 한다.

Verilog에서 모듈은 이러한 계층적인 설계를 지원한다. 블록은 Verilog에서 모듈로 설계되는데 모듈은 게이트와 같은 프리미티브 뿐 만 아니라 이미 설계된 다른 모듈을 사용하여 설계될 수 있다. 그림 2.6은 4비트 가산기에 대한 계층적인 설계를 나타내는 그림이다. 이 그림에서 사각형으로 표시된 것이 Verilog 모듈로 표현할 수 있는 블록이고 타원형으로 표시된 것이 프리미티브이다.

4비트 가산기의 계층적인 설계를 예로 들어서 다른 모듈을 사용하는 회로의 설계에 대해서 설명하고자 한다. 그림 2.7은 반가산기의 내부 회로도와 블록도이다. 반가산기에 대한 Verilog 모델은 예 2.1에서 이미 소개되었다. 반가산기가 다른 회로에서 사용될 때에는 회로도에는 반가산기의 내부 회로 대신에 반가산기의 블록도를 사용하여 나타낸다. 블록도에 입출력 포트의 이름을 표시하며 이 이름은 Verilog 모듈에서 모듈

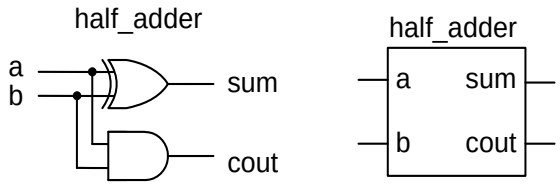


그림 2.7 반가산기의 내부 회로와 블록도

포트 이름에 해당한다.

전가산기

전가산기는 3개의 1비트 입력 a, b, c를 더하여 합 sum과 캐리 cout을 출력하는 논리회로이다. 반가산기는 2개의 1비트 입력을 더하여 합과 캐리를 출력하는 회로이다. 우리가 세 개의 숫자를 더할 때에 앞의 두 숫자를 먼저 더한 다음에 이 결과에 마지막 숫자를 더하는 것처럼 전가산기는 2개의 반가산기를 사용하여 이러한 방법으로 구현할 수 있다. 먼저 a와 b를 더하고 이들의 합인 s와 세 번째 입력 c를 더하여 세 입력의 합 sum을 구할 수 있다. 세 입력을 더하는 각 단계에서 캐리가 발생하면 최종적으로도 캐리 출력이 발생하므로 두 반가산기의 캐리출력의 OR로 캐리출력을 나타낼 수 있다. 그림 2.8은 이러한 방식으로 2개의 반가산기와 OR게이트를 사용하여 설계한 전가산기의 회로도이다.

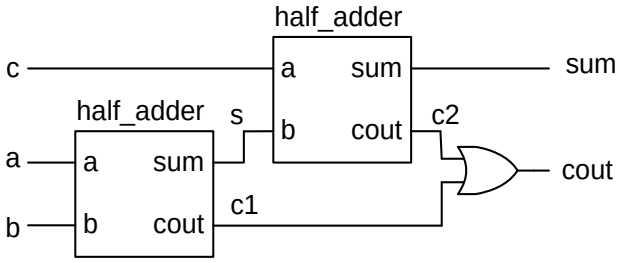


그림 2.8 전가산기 회로

모듈 인스턴스

프리미티브 뿐 만 아니라 모듈도 다른 모듈을 설계하는 데 사용될 수 있다. 이미 정의된 모듈이 다른 모듈에 포함되어 사용되었을 때에 사용된 모듈을 **모듈 인스턴스** (module instance)라고 한다. 모듈과 모듈 인스턴스는 각각 객체지향 프로그램에서 클래스와 객체의 관계와 같다.

모듈 인스턴스는 다음과 같이 모듈 이름, 인스턴스 이름, 괄호 안에 표기하는 모듈 포트에 연결되는 신호 이름의 리스트로 표현한다.

```
halfadder u1 (s1, c1, a, b);
```

괄호 안의 리스트에서 신호 이름의 순서는 모듈 정의에서의 포트 이름의 순서와 같다. 여기서 신호 s1과 c1은 모듈 halfadder의 포트 sum과 cout에 연결되며 신호 a와 b는 포트 a와 b에 연결된다. 그림 2.8의 오른쪽 반가산기에서 블록도 안쪽의 포트 이름 a,

b, sum, cout은 모듈을 정의할 때의 포트 이름이고, 이 반가산기의 각 포트에 연결된 신호인 c, c1, sum, c2는 모듈 인스턴스에 실제로 연결된 신호 이름이 된다.

모듈 인스턴스를 만드는 방법은 프리미티브 인스턴스의 경우와 같지만 인스턴스 이름을 반드시 사용해야 하는 점이 프리리티브 인스턴스와 다르다. 프리미티브 인스턴스는 인스턴스 이름을 생략할 수 있었음을 기억하자.

예 2.4는 그림 2.8의 전가산기 회로에 대한 Verilog 모델이다. 모듈 fulladder 내에서 전가산기 회로의 내부 연결선인 s, c1, c2를 나타내기 위해서 세 개의 wire 신호를 선언하였다.

예 2.4 전가산기

```
module fulladder(sum, cout, a, b, c);
    input   a, b, c;            // 입력 포트 선언
    output  sum, cout;          // 출력 포트 선언
    wire    s1, c1, c2;         // s1: 첫째 합, c1: 첫째 캐리, c2:둘째 캐리

    halfadder u1 (s1, c1, a, b);
    halfadder u2 (sum, c2, c, s1);
    or (cout, c1, c2);
endmodule
```

4비트 가산기

4비트 가산기는 그림 2.9과 같이 4개의 전가산기를 직렬로 연결하여 구성할 수 있다. 아래 자리에서 발생한 캐리가 윗자리에 더해져야 하므로 각 전가산기의 캐리 출력은 다음 자리의 전가산기의 입력에 연결된다. 이러한 방식으로 설계한 가산기를 **리플캐리**(ripple carry) **가산기**라고 한다.

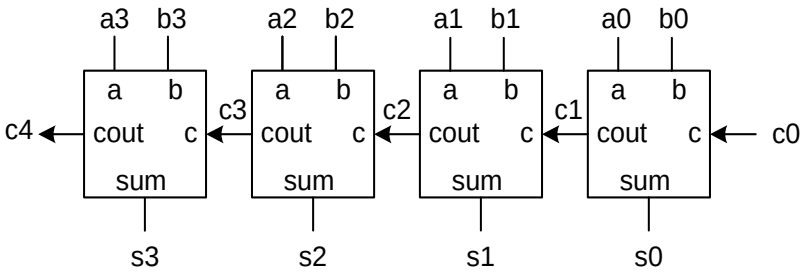


그림 2.9 4비트 리플캐리 가산기

예 2.5는 이 회로에 대한 Verilog 모델이다. 이 모듈은 두 개의 4비트 입력과 캐리 입력용으로 9개의 입력 포트를, 4비트 덧셈 출력과 최종 캐리 출력용으로 5개의 출력 포트를 정의하였고 반가산기들 간에 캐리를 전달하는 데 필요한 3개의 와이어 c1, c2, c3를 선언하였다.

예 2.5 4비트 리플캐리 가산기

```
module add_rca4(c4,s3,s2,s1,s0,a3,a2,a1,a0,b3,b2,b1,b0, c0);
    input   a3,a2,a1,a0,b3,b2,b1,b0,c0;    // 입력 포트: a3-a0, b3-b0, c0
    output  c4,s3,s2,s1,s0;                // 출력 포트: s3-s0, c4
    wire    c3,c2,c1;                      // 중간 캐리

    fulladder u1 (s0,c1,a0,b0,c0);
    fulladder u2 (s1,c2,a1,b1,c1);
    fulladder u3 (s2,c3,a2,b2,c2);
    fulladder u4 (s3,c4,a3,b3,c3);
endmodule
```

2.7 벡터

Verilog에서 선언되는 신호는 기본적으로 1비트 크기를 가지며 1비트 신호를 **스칼라**(scalar)라고 한다. Verilog에서는 여러 비트의 신호를 나타내기 위해서 **벡터**(vector) 형을 제공하며 다음과 같이 대괄호를 사용하여 벡터의 첨자 범위와 함께 선언한다.

```
output [3:0] s;     // 4비트 출력포트
input [4:1]  a;     // 4비트 입력포트
wire [0:7]   w;     // 8비트 와이어 신호
```

벡터를 선언할 때에 대괄호 안에 콜론을 사용하여 첨자의 범위를 표시하는 데 왼쪽 첨자가 최상위비트(MSB)의 첨자이고 오른쪽 첨자가 최하위비트(LSB)의 첨자이다. 최상위비트의 첨자가 가장 큰 값일 수도 있고 가장 작은 값일 수도 있다. 그림 2.10은 위에서 선언한 벡터형 신호의 각 비트와 첨자와의 관계를 보여준다.

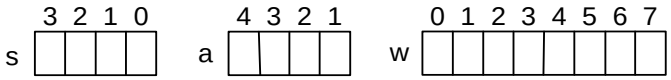


그림 2.10 벡터와 첨자

벡터형의 신호에서 특정한 위치의 비트는 다음과 같이 대괄호 안에 첨자를 표시하여 선택하여 사용할 수 있다.

```
s[3]            // s의 최상위비트, s가 2진수 1100인 경우에 s[3]는 1이다.
s[0]            // s의 최하위비트, s가 2진수 1100인 경우에 s[0]는 0이다.
a[2]            // a의 2번째 하위비트, a가 2진수 0010인 경우에 a[2]는 1이다.
w[0]            // w의 최상위비트
```

또한, 다음과 같이 콜론을 사용하여 첨자의 범위를 표시하여 여러 비트의 신호에서 연속된 일부분의 비트들을 선택할 수 있다.

```
s[1:0]          // s의 하위 2비트, s가 2진수 1101인 경우에 s[1:0]는 01이다.
w[0:3]          // w의 상위 4비트, w가 11000011인 경우에 w[0:3]는 1100이다.
```

이러한 벡터를 사용하면 여러 비트의 신호를 하나의 이름으로 나타낼 수 있으므로 여러 비트의 신호를 많이 사용하는 경우에 Verilog 표현을 간결하게 할 수 있다.

예 2.5의 4비트 가산기 모델에서 입력 a3, a2, a1, a0은 4비트 벡터 a로, 입력 b3, b2, b1, b0는 4비트 벡터 b로, 출력 s3, s2, s1, s0는 4비트 벡터 s로, 그리고 내부의 캐리 c3, c2, c1은 3비트 벡터 c로 나타낼 수 있으며 예 2.6은 4비트 가산기를 벡터를 사용하여 표현한 모델이다.

예 2.6 4비트 리플캐리 가산기 – 벡터 사용

```
module add_rca4(c4, s, a, b, c0);
    input   [3:0] a, b;         // 2개의 4비트 입력
    input   c0                  // 캐리 입력
    output  [3:0] s;            // 4비트 출력
    output  c4;                 // 캐리 출력
    wire    [3:1] c;            // 중간 캐리

    fulladder u1 (s[0],c[1],a[0],b[0],c0);
    fulladder u2 (s[1],c[2],a[1],b[1],c[1]);
    fulladder u3 (s[2],c[3],a[2],b[2],c[2]);
    fulladder u4 (s[3],c4,a[3],b[3],c[3]);
endmodule
```

2.8 모듈 포트의 연결

모듈을 설계할 때에 프리미티브와 마찬가지로 이미 설계된 다른 모듈을 인스턴스로서 사용할 수 있으며 와이어와 같은 넷(net)를 사용하여 프리미티브 또는 모듈 인스턴스들을 서로 연결한다. 모듈 인스턴스의 각 포트에는 신호들이 연결된다.

모듈을 정의할 때에 사용한 포트의 이름을 **형식 이름**(formal name)이라고 하며 모듈 인스턴스의 포트에 실제로 연결하는 신호의 이름을 **실제 이름**(actual name)이라고 한다. 형식 포트이름과 실제 포트이름을 연관시키는 방법은 위치에 의한 방법과 이름에 의한 방법의 두 가지가 있다.

위치에 의한 포트 연결

위치에 의한 포트 연결 방법은 모듈이 정의될 때의 형식 포트이름의 순서대로 연결되는 신호의 이름을 나열한다.

```
module fulladder(sum, cout, a, b, c);
    ...
    halfadder u1 (s1, c1, a, b);    // 모듈 인스턴스
    ...
endmodule

module halfadder(sum, cout, a, b);  // 모듈 정의의 헤더
    ...
endmodule
```

형식이름과 실제이름의 크기가 같지 않은 것을 허용하기는 하지만 크기를 같지 않게 사용하는 것은 것은 바람직하기 않다. 모듈의 일부 포트는 연결하지 않은 채로 그대로 둘 수 있다. 특히 출력 포트는 사용되지 않을 수도 있다. 실제 신호와 연결되지 않는 포트는 다음과 같이 빈 칸으로 두면 된다.

```
halfadder u1 (s1, , a, b);      // 모듈 포트 cout이 연결되지 않은 모듈 인스턴스
```

이름에 의한 포트 연결

많은 수의 포트를 가진 모듈을 사용할 때에 모듈 정의에서 포트가 정의된 순서를 기억하기가 쉽지 않으며 연결되지 않는 포트가 많이 있을 수 있다. Verilog에서는 신호의 이름을 위치 대신에 다음과 같은 형식으로 형식이름인 포트 이름과 실제 이름인 외부 신호를 대응시켜서 나열하여 연결할 수 있다.

.*formal_name*(*actual_name*)

이 방법에서는 포트 이름의 나열 순서는 중요하지 않으며 생략되는 포트는 연결되지 않는다. 예 2.7는 모듈 인스턴스가 위치에 의한 포트 연결을 사용한 예 2.4와 같은 회로를 모듈 인스턴스를 이름에 의한 포트 연결로 바꾸어서 나타낸 것이다. halfadder 모듈의 형식 포트 이름은 예 2.1에서 정의된 것이다. 두 halfadder 모듈 인스턴스의 포트 이름 나열 순서를 순서에 관계없다는 것을 보이기 위해서 서로 다르게 정의를 하였으나 같은 순서로 나열하는 것이 바람직하다.

예 2.7 전가산기 – 이름에 의한 포트 연결

```
module fulladder(sum, cout, a, b, c);
    input   a, b, c;            // 입력 포트 선언
    output  sum, cout;          // 출력 포트 선언
    wire    s1, c1, c2;         // s1: 첫째 합, c1: 첫째 캐리, c2:둘째 캐리

    halfadder u1 ( .a(a), .b(b), .cout(c1), .sum(s1) );
    halfadder u2 ( .cout(c1), .sum(s1), .a(a), .b(b) );
    or (cout, c1, c2);
endmodule
```

2.9 구조적 모델링

지금까지 소개한 방법과 같이 모듈을 프리미티브와 다른 모듈들을 사용하여 프리미티브 또는 모듈 인스턴스들 간의 연결로 표현하는 방법을 **구조적 모델링**(structural modeling)이라고 한다. 비교기 설계를 통해서 구조적 모델링의 예를 하나 더 보기로 하자. 그림 2.11은 비교기의 입출력을 나타내는 블록도이다.

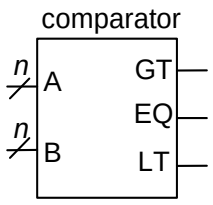


그림 2.11 *n*-비트 비교기의 블록도

비교기는 같은 크기의 두 입력 A, B를 비교하여 비교 결과에 따라서 표 2.2와 같이 세 출력 EQ, LT, GT 중 하나가 1이 되고 나머지 출력들은 0이 된다.

표 2.2 비교기의 동작

입력 A, B	EQ	LT	GT
A = B	1	0	0
A < B	0	1	0
A > B	0	0	1

1비트 비교기

제일 먼저 가장 간단한 1비트 비교기를 설계하고 이를 이용하여 큰 크기의 비교기를 설계하는 상향식 설계를 해보자. 1비트 비교기의 동작은 표 2.3의 진리표로 나타낼 수 있다.

표 2.3 1비트 비교기 진리표

A	B	EQ	LT	GT
0	0	1	0	0
0	1	0	1	0
1	0	0	0	1
1	1	1	0	0

1비트 비교기는 다음의 부울함수로 표현되며 이 회로는 예 2.8의 모듈 comparator1과 같이 게이트 프리미티브만을 사용하여 나타낼 수 있다.

$$EQ = A' \cdot B' + A \cdot B$$
$$LT = A' \cdot B$$
$$GT = A \cdot B'$$

예 2.8 1비트 비교기

```
module comparator1(eq, lt, gt, a, b);
    input   a, b;                   // 입력 포트 선언
    output  eq, lt, gt;             // 출력 포트 선언
    wire    nota, notb, t1, t2;
```

```
    not u1 (nota, a);              // nota는 a'
    not u2 (notb, b);              // notb는 b'
    and u3 (lt, nota, b);          // lt = a'b
    and u4 (gt, a, notb);          // gt = ab'
    and u5 (t1, nota, notb);       // a'b'
    and u6 (t2, a, b);             // ab
    or u7 (eq, t1, t2);            // eq = a'b' + ab
endmodule
```

2비트 비교기와 4비트 비교기

2비트 비교기는 2개의 1비트 비교기의 결과를 사용하여 설계할 수 있다. 2개의 1비트 비교기의 출력들을 각각 eq0, lt0, gt0와 eq1, lt1, gt1이라고 할 때에 2비트 비교기의 출력은 다음과 같은 부울함수로 표현될 수 있으며 2비트 비교기는 이 함수들을 적용하여 예 2.9와 같이 두 개의 1비트 비교기와 게이트 프리미티브들을 사용하여 설계할 수 있다.

EQ = eq1 · eq0 // EQ는 두 비트가 모두 같을 때에 1임
LT = lt1 + eq1 · lt0 // LT는 상위비트가 작거나
상위비트가 같고 하위비트가 작을 때에 1임
GT = gt1 + eq1 · gt0 // GT는 상위비트가 크거나
상위비트가 같고 하위비트가 클 때에 1임

예 2.9 2비트 비교기

```
module comparator2(eq, lt, gt, a, b);
    input [1:0]  a, b;             // 입력 포트 선언, 2 비트
    output  eq, lt, gt;            // 출력 포트 선언
    wire    eq0, lt0, gt0, eq1, lt1, gt1, t1, t2;

    comparator1 u1 (eq0, lt0, gt0, a[0], b[0]);
    comparator1 u2 (eq1, lt1, gt1, a[1], b[1]);
    and (eq, eq1, eq0);     // eq = eq1 · eq0
    and (t1, eq1, lt0);
    or (lt, lt1, t1);       // lt = lt1 + eq1 · lt0
    and (t2, eq1, gt0);
    or (gt, gt1, t2);       // gt = gt1 + eq1 · gt0
endmodule
```

4비트 비교기도 마찬가지 방법으로 2개의 2비트 비교기를 사용하여 설계할 수 있다. 이 때에 2비트 비교기의 출력을 사용하여 4비트 비교기의 출력을 얻는 부울 함수는 앞에서 2비트 비교기를 구성할 때에 사용한 식과 같다. 이 부울함수는 일반적으로 m비트 비교기와 n비트 비교기를 사용하여 (m+n)비트 비교기를 구성할 때에 m비트 비교기의 출력과 n비트 비교기의 출력을 사용하여 (m+n)비트 비교기의 출력을 얻을 때에도 사용된다. 이 부울함수를 모듈로 설계하면 재사용 할 수 있으므로 더 쉽게 설계를 할 수 있다. 예 2.10은 두 개의 작은 비교기의 출력을 사용하여 큰 비교기의 출력을 얻는 부울함수를 모듈로 작성한 것이다.

예 2.10 두 비교기 출력을 사용하여 결합된 비교기의 출력을 생성하는 모듈

```
module combine(eq, lt, gt, eq0, lt0, gt0, eq1, lt1, gt1);
    input   eq0, lt0, gt0, eq1, lt1, gt1;
    output  eq, lt, gt;
    wire    t1, t2;

    and (eq, eq1, eq0);     // eq = eq1·eq0
    and (t1, eq1, lt0);
    or (lt, lt1, t1);       // lt = lt1 + eq1·lt0
    and (t2, eq1, gt0);
    or (gt, gt1, t2);       // gt = gt1 + eq1·gt0
endmodule
```

그리고 예 2.11는 예 2.8에서 설계한 모듈 comparator1과 예 2.10에서 설계한 모듈 combine을 함께 사용하여 2비트 비교기 comparator2와 4비트 비교기 comparator4를 계층적으로 설계한 것이다.

예 2.11 4비트 비교기

```
module comparator4(eq, lt, gt, a, b);   // 4비트 비교기
    input [3:0]  a, b;              // 입력 포트 선언, 4 비트
    output  eq, lt, gt;             // 출력 포트 선언
    wire    eq0, lt0, gt0, eq1, lt1, gt1;

    comparator2 u1 (eq0, lt0, gt0, a[1:0], b[1:0]);
    comparator2 u2 (eq1, lt1, gt1, a[3:2], b[3:2]);
    combine u3 (eq, lt, gt, eq0, lt0, gt0, eq1, lt1, gt1);
endmodule
```

```
module comparator2(eq, lt, gt, a, b);   // 2비트 비교기
    input [1:0]  a, b;              // 입력 포트 선언, 2 비트
    output  eq, lt, gt;             // 출력 포트 선언
    wire    eq0, lt0, gt0, eq1, lt1, gt1, t1, t2;

    comparator1 u1 (eq0, lt0, gt0, a[0], b[0]);
    comparator1 u2 (eq1, lt1, gt1, a[1], b[1]);
    combine u3 (eq, lt, gt, eq0, lt0, gt0, eq1, lt1, gt1);
endmodule
```

2.10 4값 논리

Verilog는 0, 1, x, z의 네 값을 갖는 4값 논리 시스템을 사용한다. Verilog에서 0과 1은 신호의 0과 1에 대응되며 실제회로의 신호는 0과 1의 값만을 가진다. 그렇지만 시뮬레이터는 두 값 x와 z를 추가로 사용한다. 두 값의 의미는 다음과 같다.

- 논리 값 x는 시뮬레이터가 신호의 값이 0과 1인지를 결정할 수 없는 애매한 상태, 즉 값을 **알 수 없는**(unknown) **상태**를 나타낸다. 이 값은 서로 다른 값을 갖는 두 출력이 서로 연결되어 있을 때에 발생한다.
- 논리 값 z는 와이어(wire)가 어떠한 출력으로부터도 출력을 공급받지 못하는 상태인 **고임피던스 상태**를 나타낸다.

표 2.3 4값 논리에 대한 부울 연산 진리표

and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

xor	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

	buf	not
0	0	1
1	1	0
x	x	x
z	x	x

Verilog에서는 4값 논리를 사용하므로 미리 정의된 게이트 프리미티브들의 동작은 0과 1 뿐 만 아니라 x와 z에 대해서도 정의되어 있다. 표 2.3은 주요 게이트들의 4값 논리 연산에 대한 진리표이다.

4값 논리에서도 AND연산은 한 입력이 0이면 다른 입력의 값에 관계없이 출력이 0이고 OR연산은 한 입력이 1이면 다른 입력의 값에 관계없이 출력이 1이다. 3상태 출력을 제공하지 않는 모든 연산은 출력이 z가 되는 경우는 없으며 모든 연산을 할 때에 입력이 z이면 입력이 x일 때와 같은 출력 값을 생성한다.

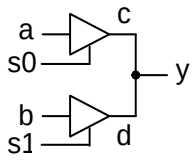


그림 2.12 여러 개의 출력이 연결된 회로

그림 2.12와 같이 여러 개의 출력이 서로 연결되었을 때의 출력 값은 출력이 서로 연결되어 있지 않을 때의 출력 값에 따라서 표 2.4와 같이 정해진다. 한 출력이 z일 때에는 다른 출력 값이 최종 출력 값이 된다. 두 출력 값이 같을 때에는 이 값이 그대로 최종 출력 값이 되며, 같지 않을 때에는 출력 값이 x가 된다.

표 2.4 여러 개의 출력이 연결되었을 때의 출력 값

	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

4값 논리에 대한 파형은 대개 그림 2.13과 같이 나타낸다. x는 0과 1사이의 값에 빗금을 치거나 음영을 넣어서 값을 알 수 없음을 나타내고 z는 0과 1의 중간 위치로 나타낸다.

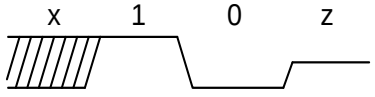


그림 2.13 4값 논리 값에 대한 파형

2.11 설계 검증

설계된 논리회로는 논리 시뮬레이션 방법을 사용하여 원래의 설계 명세를 만족하게 동작하는 지 검증한다. 논리 시뮬레이션은 회로의 입력에 적절한 **스티뮬러스**(stimulus) **패턴**을 공급하고 시뮬레이션을 통하여 회로가 올바르게 동작하는 지를 관찰한다. 이를 위해서 Verilog에서는 그림 2.14와 같은 구조로 동작을 검증할 테스트 모듈의 인스턴스를 포함하고 스티뮬러스 패턴을 발생시켜서 테스트 모듈의 입력에 연결하고 테스트 모듈의 출력값을 확인하는 문장을 포함한 테스트벤치(testbench) 모듈을 작성해야 한다.

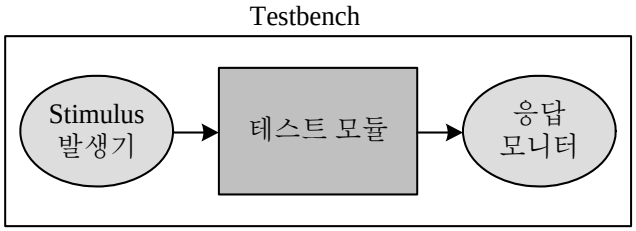


그림 2.14 테스트 모듈을 검사하기위한 테스트벤치

설계를 계층적으로 수행하는 경우에 전체 시스템을 더 간단한 작은 단위로 분할하고 이를 더 아래 단계로 반복하는 하향식 설계를 주로 하는 반면에 설계의 검증은 제일 먼저 가장 하위 단계의 모듈을 먼저 검증하고 이 모듈을 사용하여 설계한 모듈을 그 다음에 검증하고 전체 시스템은 가장 나중에 검증을 하는 상향식 방법을 사용한다.

테스트벤치 작성

테스트 모듈을 검증할 테스트벤치를 Verilog로 작성할 때에 initial등을 사용하여 시뮬레이션 시작부터 시간의 흐름에 따른 테스트 입력 값의 변화를 지정한다. 예 2.12의 모듈 test_halfadder는 앞에서 작성한 반가산기 모듈 halfadder에 대한 테스트벤치이다.

이 테스트벤치에 대한 설명은 다음과 같다.

1. 이 모델에는 새로운 형인 reg가 소개되었다. reg는 값을 저장할 수 있는 신호를 선언하기 위해서 사용한다. reg로 선언된 a와 b는 프로그램에서의 변수와 같이 새로운 값이 주어지기 전까지는 이전에 주어진 값을 보관하고 있다. 테스트 모듈의 입력을 제공하기 위해서 사용되는 신호들은 대개 reg 형으로 선언된다.
2. 테스트 모듈의 출력과 연결되는 신호 sum과 count는 값이 테스트 모듈에서 공급되

예 2.12 반가산기 halfadder 모듈에 대한 테스트벤치

```
module tb_halfadder;        // 테스트벤치는 입출력 포트가 없음
    reg   a, b;             // 입력 포트 선언, 2 비트
    wire  sum, cout;        // 출력 포트 선언

    halfadder u1 (sum, cout, a, b);     // 테스트 모듈

    initial #100 $finish;       // 시뮬레이션 최대시간

    initial begin               // 스티뮬러스 패턴
        #10 a = 0; b = 0;       // a=0, b=0
        #10 b = 1;              // a=0, b=1
        #10 a = 1;              // a=1, b=1
        #10 b = 0;              // a=1, b=0
    end
endmodule
```

므로 저장할 필요가 없으며 보통의 net형인 wire로서 선언하였다.

3. 테스트 모듈의 인스턴스를 선언하고 테스트 모듈의 입력에 값을 공급하는 신호와 출력을 관찰하는 신호를 연결한다.
4. initial은 시뮬레이션을 시작할 때에 수행할 명령어들을 나타내기 위해서 사용한다. initial 다음에 수행할 명령어를 작성하며, 수행할 명령어가 여러 개이면 begin과 end 블록 내에 수행할 명령어들을 작성한다.
5. 첫 번째 initial에서는 시뮬레이션의 종료시간을 지정한다. #100은 지연시간을 나타내며 $finish는 시뮬레이션을 종료시키는 시스템 태스크이다. 즉, 100 시뮬레이션 시간이 지난 후에 시뮬레이션이 종료된다.
6. 두 번째 initial에서는 스티뮬러스 패턴을 생성한다. 시간 10 경과 후에 a와 b를 모두 0으로 초기화 하고 시간 10이 지날 때마다 입력 값을 바꾼다. 테스트 모듈이 입력이 2개인 조합회로이므로 입력은 00, 01, 11, 10의 모든 4가지 조합을 생성하였다.
7. 두 개의 initial 문장은 시간 0에서 동시에 시뮬레이션을 시작한다.

스티뮬러스 패턴을 잘못 만들면 회로 설계의 잘못을 찾지 못할 수 있다. 그러므로 검사가 필요한 모든 경우를 검사할 수 있도록 체계적이며 정교하게 스티뮬러스 패턴을 만들어야 한다.

대부분의 Verilog 시뮬레이터들은 그래픽 환경에서 사용되며 신호의 값을 파형으로 보여주는 기능이 있다. 그림 2-15는 예 2-12의 테스트벤치를 사용하여 시뮬레이션한 결과의 파형을 보여준다. 음영인 부분은 값이 x인 상태를 나타낸다. 시간 10까지는 모든 신호가 x 값을 가지며 a와 b의 값이 지정됨에 따라서 sum과 cout값도 정해진다. 그리고 파형은 시뮬레이션 종료시간인 시간 100까지만 표시되었다.

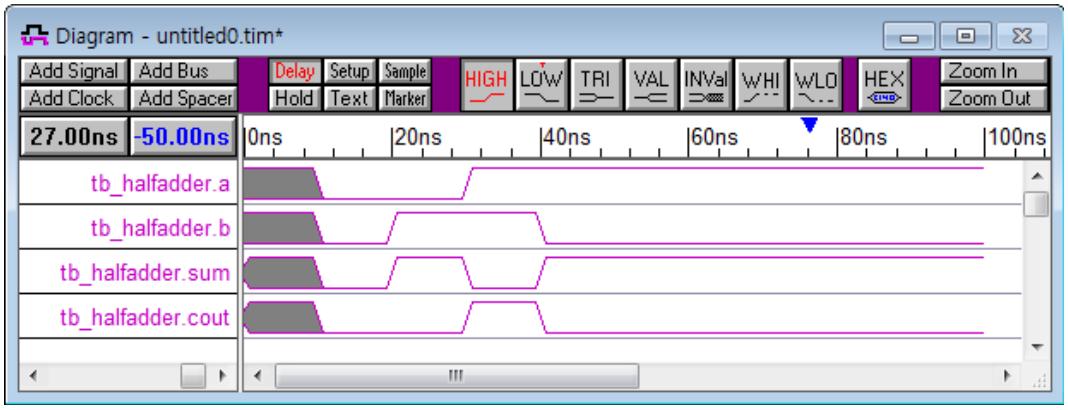


그림 2-15. halfadder에 대한 테스트벤치의 시뮬레이션 파형

Verilog에는 $display와 $monitor와 같은 출력문을 제공하는 시스템 태스크가 제공되며 이 문장을 사용하여 텍스트로 시뮬레이션 결과를 출력할 수 있다. **$display**는 문자열, 신호, 변수와 수식 등의 값을 출력하며 **$monitor**는 신호의 값이 변할 때마다 신호들을 계속하여 출력한다. 형식은 C언어의 printf와 유사한 형태로 사용하며 신호나 변수의 출력에 사용되는 주요 형식지정자로 `%d`는 10진수, `%e`는 실수의 과학적 표기법 `%f`는 실수의 십진 표기법, `%s`는 문자열, `%b`는 2진수, `%t`는 시간 형식 출력을 나타낸다.

예 2.13는 예 2.12의 테스트벤치에 $display와 $monitor 태스크를 사용하여 시뮬레이션 결과를 텍스트로 출력하는 기능을 포함하도록 수정한 것이다. 코드에서 굵게 표시한 부분이 추가한 부분이다.

예 2.13 텍스트 출력을 포함한 테스트벤치

```
module tb_halfadder;        // 테스트벤치는 입출력 포트가 없음
    reg   a, b;             // 입력 포트 선언, 2 비트
    wire  sum, cout;        // 출력 포트 선언

    halfadder u1 (sum, cout, a, b);     // 테스트 모듈
```

```
    initial $display("time\ta b cout sum");
    initial $monitor($realtime, "\t%b %b %b %b", a, b, cout, sum);

    initial #100 $finish;       // 시뮬레이션 최대시간
    initial begin               // 스티뮬러스 패턴
        #10 a = 0; b = 0;       // a=0, b=0
        #10 b = 1;              // a=0, b=1
        #10 a = 1;              // a=1, b=1
        #10 b = 0;              // a=1, b=0
    end
endmodule
```

이 예를 사용하여 시뮬레이션을 하면 다음과 같이 텍스트로 출력한다.

```
time    a b cout sum
0       x x x x
10      0 0 0 0
20      0 1 0 1
30      1 1 1 0
40      1 0 0 1
```

FPGA 합성을 목적으로 제공되는 CAD 도구에서는 테스트 벤치에서 시뮬레이션 목적으로 사용하는 Verilog 구문을 지원하지 않는 경우가 많다. 앞에서 소개한 예약어 initial과 지연시간을 나타내는 #100과 시스템 태스크 $finish가 이러한 예에 속한다.

파형 편집기 사용

그래픽 기반의 시뮬레이터에서는 스티뮬러스 패턴을 파형 편집기에서 직접 그려서 공급할 수 있다. 이러한 경우에는 테스트벤치를 Verilog로 작성할 필요가 없으며 테스트 모듈이 최상위 모듈이 된다.

2.12 전파지연

실제의 물리적 회로는 입력 변화와 출력 변화 사이에 **전파 지연**(propagation delay) 시간을 가지고 있다. Verilog의 프리미티브는 기본적으로 지연 시간이 0이다. 이것은 입력이 변화되는 즉시 출력도 변화됨을 뜻한다. 회로의 타이밍 검증은 궁극적으로 회로에서의 전파 지연을 고려해야 하지만 회로를 기능(function) 수준에서 동작을 검사할

때에는 흔히 지연시간을 0 또는 1로 단순화하여 시뮬레이션을 한다.

Verilog에서 전파 지연의 모델링은 #을 사용하여 할 수 있다. 예 2-14는 모든 게이트가 지연시간을 가지도록 한 예이다.

예 2.14 전파지연을 갖는 반가산기 회로의 Verilog 모델

```
module halfadder(sum, cout, a, b);      // 회로(모듈)의 이름과 입출력 포트 이름
    input   a, b;           // 입력 포트 선언
    output  sum, cout;      // 출력 포트 선언

    xor #4 u1 (sum, a, b);      // XOR 게이트
    and #2 u2 (cout, a, b);     // AND 게이트
endmodule
```

프리미티브 인스턴스의 전파지연은 프리미티브 이름 다음에 #지연시간을 삽입하여 나타낸다. 예 2.14에서 AND게이트과 XOR게이트는 각각 지연시간이 2와 4이다. 그림 2-16은 이러한 전파지연을 갖는 halfadder에 대한 시뮬레이션 파형이다. cout과 sum 출력이 입력 변화에 대해서 지연되어 변화함을 확인할 수 있다.

전파지연시간을 이같이 모델링 할 수 있을 지라도 실제의 전파지연시간은 구현기술에 따라서 다르며 ASIC의 표준 셀이나 FPGA의 내부 정보에 의해서 정해진다. 설계자들은 구현기술과 독립적으로 회로를 설계할 필요가 있으며 시뮬레이션이 아닌 구현 목적으로는 이러한 전파지연을 나타낼 필요는 없다. 타이밍에 대한 정보는 논리 합성 도구 내에 포함되어 있으며 논리합성 후의 검증 과정에서 전파지연이 고려된 파형을 관찰할 수 있다.

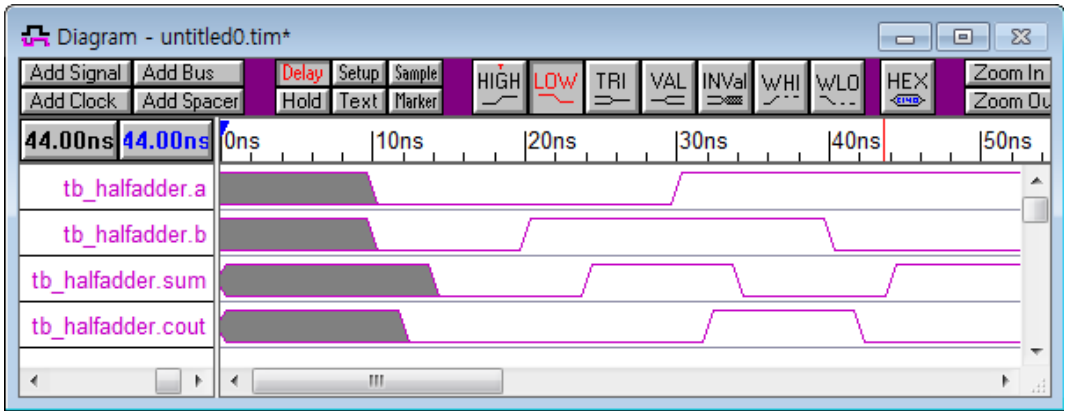


그림 2-16 전파지연을 갖는 halfadder에 대한 시뮬레이션 파형

전파지연 모델

시뮬레이션에서 전파 지연은 **관성지연**(inertial delay)와 **전달지연**(transport delay)의 두 가지 방법으로 모델링할 수 있다.

관성지연 모델에서 게이트의 지연시간보다 짧은 시간동안의 입력 펄스는 무시되어 출력의 변화가 발생하지 않는다. 이 모델은 물리적 회로에서 캐패시턴스(capacitance) 특성으로 인해서 출력이 순간적으로 변할 수 없음을 모델링한 것으로서 Verilog의 프리미티브 게이트의 전파 지연을 모델링하는 데 사용된다.

전달지연 모델에서는 입력의 펄스폭에 관계없이 입력의 변화가 출력에 그대로 전달된다. 이 모델은 wire를 통한 신호의 전달을 모델링하는 데 사용된다. wire는 대개 전파지연을 무시할 수 있지만 길이가 긴 경우에는 무시될 수 없을 경우가 있다. 이 경우에 wire를 선언할 때에 다음과 같이 지연시간을 함께 선언할 수 있으며 이 지연은 기본적으로 전달지연모델을 따른다.

```
wire #2 long_wire;
```

그림 2-17은 두 지연 모델의 차이점을 입출력 파형의 비교를 통해서 나타낸 것이다. 오른쪽 그림에서 게이트 전파지연 시간보다 짧은 펄스폭을 갖는 입력은 관성지연 모델에서는 무시되어 출력의 변화가 없지만 전달지연에서는 입력의 변화가 그대로 출력으로 전달된다.

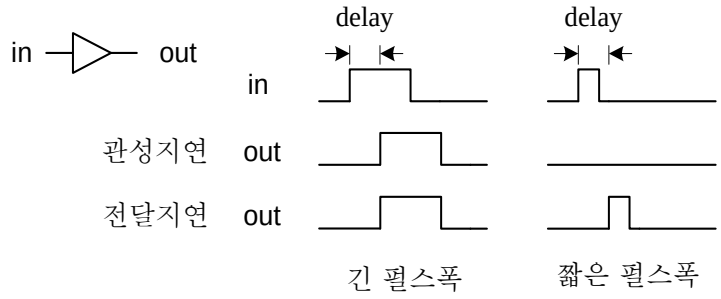


그림 2.17 관성지연과 전달지연

그림 2.18은 예 2.14의 반가산기에 펄스폭이 3인 입력 변화가 가해졌을 때에 실제의 시뮬레이터에서 파형을 보여준 것이다. 전파지연이 4인 cout 출력은 변했으나 전파지연이 2인 sum 출력은 변화가 없음을 보여준다. 그림 2.19는 펄스폭이 5인 입력 변화가 가해졌을 때의 파형이다. 여기서는 두 출력이 모두 변했음을 확인할 수 있다.

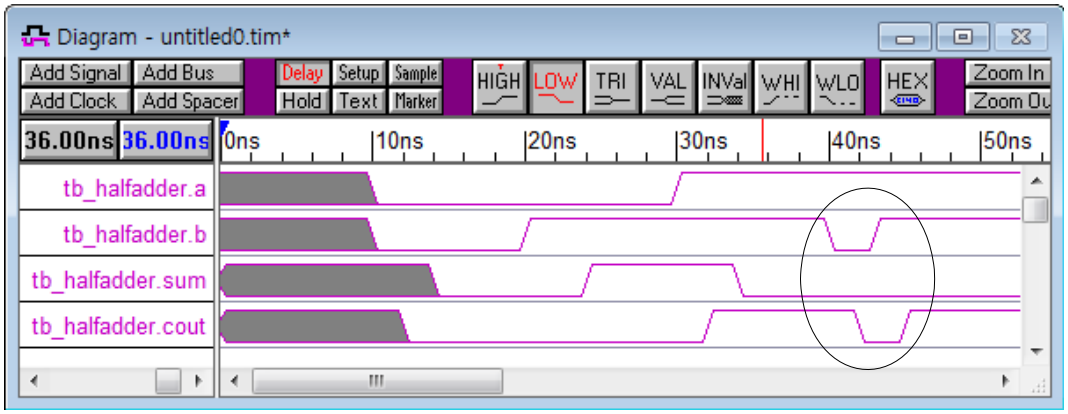


그림 2.18 짧은 펄스폭의 입력에 대한 시뮬레이션 파형

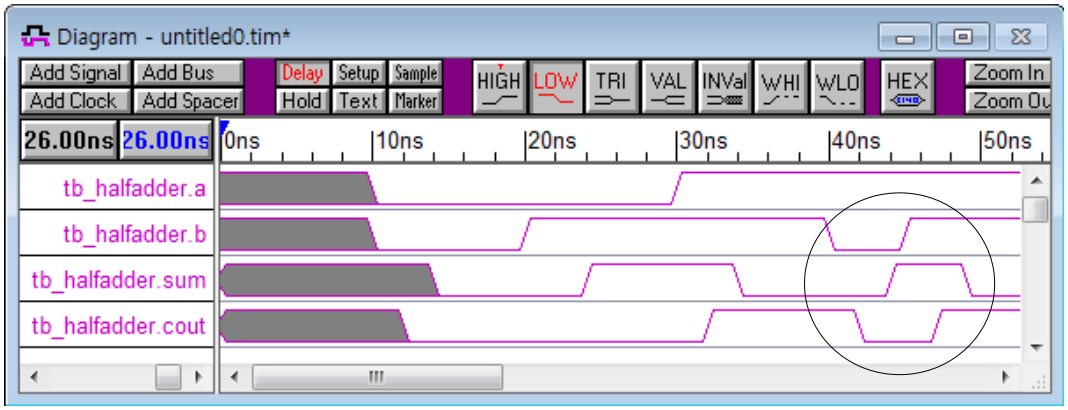


그림 2.19 긴 펄스폭의 입력에 대한 시뮬레이션 파형

2.12 이벤트 구동 시뮬레이션

시뮬레이션 동안의 신호의 값의 변화를 **이벤트**(event)라고 부른다. 논리 시뮬레이션은 시뮬레이션 동안 내내 수행하는 것이 아니라 이벤트가 발생할 때마다 수행한다. 이벤트 사이의 시간 동안에는 아무 일도 하지 않는다. 이러한 시뮬레이션을 **이벤트 구동 시뮬레이션**이라고 한다.

테스트 모듈의 입력에 이벤트가 발생하면 내부 회로의 동작을 수행하고 내부의 신호와 출력 신호 값을 갱신한다. 회로에 지연시간이 있는 경우에 지연시간 경과 후에 신호의 변화에 대한 이벤트가 발생하도록 이벤트 스케쥴링을 한다. 지연시간이 없는 회로에 대해서는 미소시간 Δt 후에 이벤트가 발생하도록 스케쥴링을 한다. 시뮬레이

터는 현재 시간에 발생하도록 스케쥴링된 모든 이벤트에 대해서 처리를 한 후에 대기 중인 이벤트가 있는 다음 시간으로 이동하여 대기 중인 이벤트를 처리한다. 시뮬레이션 시간이 종료되었거나 대기 중인 이벤트가 없으면 시뮬레이션을 종료한다.

2.13 진리표 모델

Verilog는 and, or, not과 같은 기본적인 프리미티브들을 제공하며 이러한 프리미티브를 사용하여 더 복잡한 논리회로를 구성할 수 있다. 일반적으로 조합회로는 진리표(truth table)로 순차회로는 상태전이표(state transition table)로 동작을 나타낼 수 있다. Verilog에서는 프리미티브나 다른 모듈을 사용하지 않고 대신에 진리표 또는 상태전이표를 기술하여 사용자가 직접 프리미티브를 정의할 수 있다. 예 2.15는 **사용자 정의 프리미티브**(user-defined primitive, UDP)의 예이다.

예 2.15 사용자정의 OR게이트 udp_or

```
primitive udp_or(y, a, b);
    output  y;
    input   a, b;
    table   // a  b : y
        0 0 : 0;
        0 1 : 1;
        1 0 : 1;
        1 1 : 1;
    endtable
endprimitive
```

UDP의 정의는 primitive로 시작하여 endprimitive로 끝나며 출력이 스칼라이어야 한다. 조합회로의 UDP는 table과 endtable 사이에 다음 형식으로 진리표의 각 엔트리를 나열한다.

입력값들 : 출력값;

Verilog는 4값 논리를 사용하므로 x 또는 z가 입력이 될 수 있으며 이러한 입력에 대해서 어떻게 동작하는지도 기술해야 한다. UDP에서는 z값은 허용하지 않으므로 z값이 입력되면 x값으로 취급되며 입력이 x인 경우의 동작도 기술할 필요가 있다. 특히 출력 값이 x가 아닌 경우에는 반드시 기술해야 한다. 예 2.16은 이러한 점을 반영하여

수정한 것이다. 4값 논리에 대한 OR 연산은 표 2.3을 참고하기 바라며 이 예에서 출력이 x인 경우는 생략하였다.

예 2.16 x값 입력을 고려한 사용자정의 OR게이트 udp_or

```
primitive udp_or(y, a, b);
    output  y;
    input   a, b;
    table   // a  b : y
        0 0 : 0;
        0 1 : 1;
        1 0 : 1;
        1 1 : 1;
        1 x : 1;    // 추가
        x 1 : 1;    // 추가
    endtable
endprimitive
```

그리고 **? 표기법**을 사용하면 예 2.16의 UDP는 더 간단하게 기술할 수 있다. ?는 입력으로 0, 1, x를 모두 사용할 수 있는 **don't care조건**을 나타낸다. x는 값을 알 수 없는 것이므로 ?와는 다르다. 예 2.17는 예 2.16을 ?를 사용하여 다시 작성한 것으로서 table의 엔트리 수가 3개로 감소하였다.

예 2.17 ? 표기법을 사용한 사용자정의 OR게이트 udp_or

```
primitive udp_or(y, a, b);
    output  y;
    input   a, b;
    table   // a  b : y
        0 0 : 0;
        ? 1 : 1;
        1 ? : 1;
    endtable
endprimitive
```

순차회로에 대한 UDP 정의는 상태도로 나타내는 데 다소 복잡하므로 생략한다. UDP는 대부분의 논리합성 도구에서 사용자를 위해서는 지원하지 않으므로 시뮬레이션이 아닌 구현을 목적으로 Verilog를 사용할 때에는 UDP를 사용하지 않아야 한다.

3장 Verilog 데이터흐름 모델 논리 설계

3.1 구조적 모델링과 동작적 모델링

Verilog는 구조적 모델링과 동작적 모델링을 지원한다. **구조적 모델링**(structural modeling)은 2장에서 이미 배운 바와 같이 프리미티브 게이트나 기존에 설계된 모듈들을 서로 연결하여 회로를 설계하는 방식이다. 그렇지만 프리미티브를 사용한 게이트 수준의 모델링 방법은 회로 설계가 불편하고 설계된 회로의 동작을 이해하기 쉽지 않다. 이 방법을 사용하면 수십 개의 게이트를 갖는 회로를 설계할 때에도 어려움이 따르는 데 현재의 많은 ASIC들은 단일 칩에 수백만 게이트를 가지고 있는 실정이다.

동작적 모델링(behavioral modeling)은 회로의 내부 구조 대신에 회로가 수행하는 기능을 기술한다. 즉 회로가 어떻게(how) 구성되는 지가 아니라 회로가 무엇을(what) 수행할 것인지를 기술한다. 회로의 입력과 출력 간의 관계를 기술하고, 회로의 내부와 물리적 구현에 대한 상세한 사항을 기술하지 않는다. 전파지연시간은 회로의 동작 모델에 포함되지 않으며 물리적 구현시의 타이밍 제약이 부과될 때에 합성 도구에 의해서 고려된다. 동작 모델링을 사용하면 설계자는 빠른 설계와 검증을 할 수 있으며, 합성도구를 사용하여 설계를 최적화하고 물리적 구현 기술로 맵핑시킬 수 있다.

현재의 디지털 시스템 설계 방법은 큰 회로를 몇 개의 기능 단위(functional unit)로 나누고 이들이 포트를 통해 연결되도록 전체적인 구조(architecture)를 설계하고 각각의 기능 단위는 동작적 모델로 기술한다. 동작적 모델로 기술된 각 기능 단위는 논리합성도구에 의해서 자동적으로 게이트 수준의 회로로 합성된다. 따라서 설계자는 게이트 수준의 회로를 직접 구현할 필요가 없다. 게이트 수준의 회로는 ASIC의 표준 셀 라이브러리나 FPGA의 라이브러리와 같은 물리적 기술로 변환되어 최종 구현이 이루어지게 된다.

동작적 모델링은 구조적 모델링과 함께 사용할 수 있다. 어떤 모듈을 설계할 때에 회로의 일부분은 구조적 모델링 방법으로, 다른 부분은 동작적 모델링 방법을 사용하여 기술할 수 있다. 구조적 모델링 방법은 설계된 하위 모듈을 사용하여 모델링하는 상위 수준의 모델링에 많이 사용된다.

동작적 모델링에서 회로의 동작은 부울 함수 수식으로 기술할 수도 있고 알고리즘과 같은 추상적인 표현을 사용하여 나타낼 수도 있다. 이 중에서 부울 함수와 같은 수식으로 기술하는 모델링을 **데이터흐름 모델링**(data flow modeling)이라고 부른다. 이 장에서는 데이터흐름 모델링에 대해서 다루고 동작적 모델링의 나머지 부분은 다음 장에서 소개한다.

3.2 동작적 모델링을 위한 자료형

동작적 모델을 보기에 앞서서 Verilog 모델에서 정보를 표현하는 자료형에 대해서 먼저 보기로 하자. 컴퓨터 프로그램에서처럼 Verilog에서도 정보를 다루기 위해서 **변수**(variable)를 사용한다. Verilog에서 변수는 2진수로 표현된 **신호**(signal)를 나타낼 수 있다. Verilog에서는 물리적인 신호이외에도 동작 모델링에 필요한 값을 표현하기 위해서도 변수를 사용한다.

Verilog의 자료형은 net 형과 variable 형의 두 종류가 있다. **net 형** 변수는 물리적 회로의 와이어와 같은 동작을 하여 설계 개체들을 서로 연결하는 데 사용된다. **variable 형** 변수는 프로그램이 수행되는 동안 값을 저장하는 프로그래밍 언어에서의 변수와 같은 역할을 수행한다. variable 형은 verilog-1995에서는 register 형이라고 불렀으나 하드웨어 레지스터로 합성되지 않을 수도 있으므로 verilog-2001부터 variable 형이라고 부른다. net 형의 자료형은 여러 가지가 있지만 주로 wire 형을 사용하며 variable 형은 **reg 형**과 **integer 형**을 주로 사용한다. wire와 reg는 기본적으로 1비트의 크기를 가지며 integer 형은 기본적으로 32비트의 크기를 갖는다.

정수의 표현

variable 형 변수에 저장되는 정수는 다음과 같이 ' 뒤에 진수를 나타내는 문자를 함께 사용하여 10진수(d), 16진수(h), 8진수(o), 2진수(b)로 표현할 수 있으며 ' 앞에 비트 크기를 지정할 수 있다. 비트 크기가 지정되지 않으면 기본적으로 integer형의 크기인 32비트로 지정된다.

```
4'b1101         // 4비트 2진수
'b1101          // 2진수 (32비트)
12'h5fc         // 12비트 16진수
'o755           // 8진수 (32비트)
15 또는 'd15    // 10진수 (32비트)
16'd255         // 16비트 10진수
```

긴 자리 숫자에 대한 가독성을 위하여 다음과 같이 _를 사용할 수 있다. _는 임의의 위치에서 사용할 수 있으며 Verilog는 _를 무시한다.

```
9'b1_0110_1101          // 9비트 2진수
```

그리고 4값 논리에서 소개한 x와 z를 다음과 같이 신호의 값을 표현하는 데에 사용할 수 있다.

```
12'h75x          // 12비트, 하위 4비트가 x임
10'bz3           // 10비트, 상위 6비트가 z임
8'hx             // 8비트 모두 x임
'bx              // 32비트 모두 x임
```

16진수에서의 x 또는 z는 4개의 비트를 이 값으로 지정하며, 8진수에서는 3개의 비트를, 2진수에서는 1비트를 이 값으로 지정한다. 표현된 수의 비트 수가 지정된 비트 크기보다 작으면 상위 비트들은 0으로 채워진다. 그렇지만 표현된 수의 최상위 값이 x 또는 z이면 상위 비트들은 이 값으로 채워진다.

3.3 연속할당문과 부울함수식을 사용한 모델링

우리는 2장에서 부울함수 식을 Verilog 프리미티브를 사용하여 기술한 예를 보았으며 이를 통해서 간단한 부울함수 식도 제법 복잡하게 기술됨을 확인할 수 있었다. Verilog의 **연속할당문**(continuous assignment statement)을 사용하면 부울함수로 표현되는 논리회로를 매우 간단하게 기술할 수 있다. 회로의 데이터흐름 모델링은 연속할당문을 사용하여 기술한다. 연속할당문은 다음과 같이 **assign**으로 시작되며 C언어에서 사용하는 것과 같은 연산자와 할당문을 사용한다.

```
assign y = ~a & b | a & ~c;       // 부울함수 y = a' b + a c'
```

여기서 &는 비트단위 AND연산자를, |는 비트단위 OR연산자를, ~는 비트단위 NOT연산자를 나타낸다. 예 3.1은 예 2.8의 1비트 비교기를 연속할당문을 사용하여 나타낸 동작적 모델이다. 여러 개의 연속할당문은 정의되는 순서에 관계없이 동시에 동작하는 회로를 기술한다.

예 3.1 1비트 비교기의 데이터흐름 모델

```
module comparator1(eq, lt, gt, a, b);
    input   a, b;                   // 입력 포트 선언
    output  eq, lt, gt;             // 출력 포트 선언

    assign lt = ~a & b;
    assign gt = a & ~b;
    assign eq = ~a & ~b | a & b;
endmodule
```

그림 3.1은 2장에서 배운 구조적 모델링, 사용자 정의 프리미티브와 지금 소개한 연속할당문과 조합회로의 세 가지 표현 방법과의 연관 관계를 보여준다.

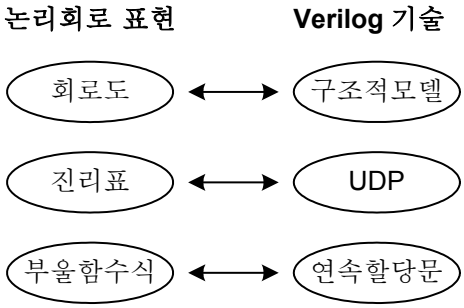


그림 3.1 조합회로의 각 표현에 대한 Verilog 기술

비트단위 논리연산자

다음과 같은 비트단위 논리연산자가 있으며 XNOR 연산자를 제외하면 C언어와 같다.

~	NOT 연산자
&	AND 연산자
|	OR 연산자
^	XOR 연산자
^~ 또는 ~^	XNOR 연산자

벡터 또는 정수에 대한 비트단위 논리연산은 비트 단위로 이루어지며 연산결과는 피연산자의 길이와 같다. 그리고 비트단위 논리 연산자의 우선순위는 ~, &, ^, | 순서이다. XNOR연산자는 XOR연산자와 우선순위가 같다.

각 비트단위 논리연산자는 대응되는 게이트들이 있으므로 이 연산자들을 사용하여 표현된 식은 게이트 수준의 회로로 쉽게 합성될 수 있다.

여러 개의 할당문을 포함한 연속할당문

여러 개의 부울 함수 식은 예 3.1과 같이 여러 개의 assign을 사용하여 여러 개의 연속할당문으로 나타낼 수 있지만 assign문은 하나 이상의 할당문들을 컴마로 구분하여 함께 나타낼 수 있다. 다음은 예 3.1의 3개의 연속할당문을 하나의 assign문으로 나

타낸 것이다.

```
assign  lt = ~a & b,
        gt = a & ~b,
        eq = ~a & ~b | a & b;
```

묵시적인 연속할당문

연속할당문을 통해서 값이 지정되는 신호는 일반적으로 wire형으로 선언한 후에 assign문을 사용하여 값을 할당한다. output 신호는 기본적으로 wire형으로 다루어지므로 별도로 wire형으로 선언할 필요가 없다.

연속할당문은 assign을 사용하지 않고 다음과 같이 wire형으로 선언할 때에 직접 수식을 지정하여 나타낼 수 있다.

```
wire out = a & b | c & d;
```

위의 표현은 다음의 두 문장을 합쳐서 기술한 것이다.

```
wire out;
assign out = a & b | c & d;
```

연속할당문과 전파지연

우리는 2장에서 프리미티브 인스턴스의 전파 지연시간을 다음과 같이 모듈 이름 다음에 #을 사용하여 지정할 수 있음을 배웠다.

```
nor #5 u1 (y, a, b);
```

연속할당문도 다음과 같이 assign 다음에 #을 사용하여 회로의 전파지연 시간을 지정할 수 있다.

```
assign #10 out = a & b | c & d;
```

이 문장은 오른쪽 수식의 결과가 시간이 10이 경과한 후에 왼쪽 변수에 저장됨을 나타낸다. 연속할당문의 전파지연은 관성지연 특성을 가져서 전파지연 시간보다 짧은 펄스 신호는 무시한다.

전파지연은 연속할당문 대신에 넷을 선언할 때에도 지정할 수 있다. 다음의 세 가지 예는 같은 효과를 나타낸다.

```
wire #10 out = a & b | c & d;          // 묵시적 할당문에 전파지연 지정

wire #10 out;                          // 넷을 선언할 때에 전파지연 지정
assign  out =  out = a & b | c & d;

wire out;                              // 연속할당문에서 전파지연 지정
assign #10 out = a & b | c & d;
```

3.4 조건연산자와 멀티플렉서

조건연산자

C언어와 같은 조건연산자(? :)를 사용할 수 있으며, 조건에 따라서 출력의 부울함수 식이 바뀌는 회로를 연속할당문으로 간결하게 나타낼 수 있다. 다음 문장은 그림 3.2의 2×1 멀티플렉서를 조건 연산자를 사용하여 연속할당문으로 나타낸 것이다.

```
assign y = select ? data1 : data0;
```

이 식에서 select가 1이면 data1 값이 y에 출력되고, 0이면 data0 값이 y에 출력된다.

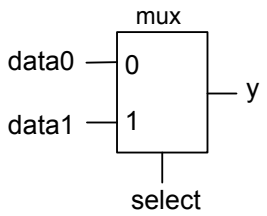


그림 3.2 2×1 멀티플렉서

조건연산자를 사용한 식은 이처럼 2×1 멀티플렉서를 사용하여 합성할 수 있다. 예를 들어서 다음의 연속할당문으로 기술된 회로는 멀티플렉서를 사용하여 그림 3.3와 같이 합성할 수 있으며 멀티플렉서 사용하여 합성된 회로는 게이트 수준에서 추가적으로 최적화될 수 있다.

```
assign out = sel ?  a | b : a & b;
```

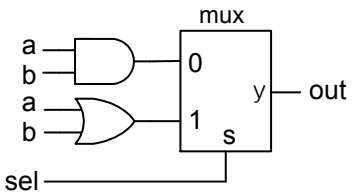
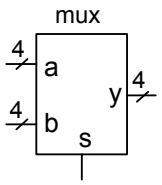


그림 3.3 조건연산자를 사용한 설계의 합성 예

n비트 2×1 멀티플렉서

멀티플렉서(multiplexer)는 여러 개의 입력들 중에서 선택신호의 값에 따라서 한 입력을 선택하여 출력으로 전달하는 조합회로이다. 두 개의 데이터 입력을 갖는 2×1 멀티플렉서는 조건연산자를 사용하여 쉽게 나타낼 수 있음을 보았다. 멀티플렉서의 데이터 입력과 출력은 여러 비트를 가진 벡터 신호일 수 있으며 이 경우에도 마찬가지로 조건연산자를 사용하여 나타낼 수 있다.

그림 3.4는 두 개의 4비트 데이터 입력 a, b와 4비트 출력 y를 갖는 4비트 2×1 멀티플렉서의 블록도와 동작을 나타내는 진리표를 보여준다.



s	y
0	a
1	b

(a) 블록도 (b) 동작

그림 3.4 2×1멀티플렉서의 블록도와 동작

이 회로는 예 3.2와 같이 벡터 자료형과 조건연산자를 사용하여 나타낼 수 있다.

예 3.2 조건연산자를 사용한 4비트 2×1멀티플렉서

```
module mux2_4(a, b, s, y);
    input [3:0] a, b;       // 2개의 4비트 데이터 입력
    input  s;               // 선택 입력
    output [3:0] y;         // 4비트 데이터 출력

    assign y = s ? b : a;   // s=0이면 y=a, s=1이면 y=b
endmodule
```

3상태 출력을 가진 회로

그림 3.5는 3상태 출력을 갖는 회로이다.

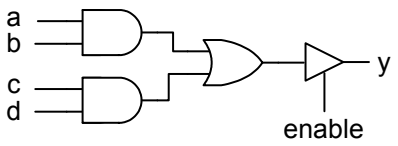


그림 3.5 3상태출력을 갖는 회로

이 회로는 다음과 같은 연속할당문으로 나타낼 수 있다.

```
assign y = enable ? a & b | c & d : 1'bz;
```

이 식은 enable이 1이면 y에는 조합회로 a & b | c & d의 결과가 출력되며 enable이 0이면 y는 z값을 가져서 고임피던스 상태가 된다. 이처럼 3상태 출력은 조건 연산자를 사용하여 나타낼 수 있다. 이 회로는 다음과 같이 조합회로 부분과 3상태 출력부분을 구분하여 나타낼 수도 있다.

```
wire yout;

assign yout = a & b | c & d;
assign y = enable ? yout : 1'bz;
```

3.5 parameter와 `define을 사용한 상수 정의

parameter를 사용한 상수 정의

설계에 따라서 바뀔 수 있는 상수 값은 다음과 같이 parameter를 사용하여 기호상수로 정의를 하면 다른 모듈에서 이 모듈을 인스턴스로서 사용할 때에 기호상수의 값을 바꾸어 사용할 수 있다. 따라서 이식성, 소스의 가독성, 확장성, 재구성 가능성 측면에서 장점을 가지고 있다.

```
parameter SIZE = 4;
```

예 3.3은 parameter를 사용하여 상수를 정의하여 예 3.2를 다시 작성한 것이다.

예 3.3 parameter를 사용한 2×1 멀티플렉서

```
module mux2(a, b, s, y);
    parameter SIZE = 4;              // parameter상수 정의
    input [SIZE-1:0]    a, b;        // 데이터 입력
    input  s;                        // 선택 입력
    output [SIZE-1:0] y;             // 데이터 출력

    assign y = s ? b : a;            // s=0이면 y=a, s=1이면 y=b
endmodule
```

모듈 인스턴스에서의 parameter 재지정

parameter를 포함한 모듈을 인스턴스로서 사용할 때에 모듈에서 정의된 parameter의 값을 모듈 인스턴스에서 바꾸어서 다시 지정할 수 있다. parameter값을 지정하지 않으면 모듈에서 정의된 파라메터 값이 사용된다.

모듈 인스턴스에서 다음과 같이 모듈 이름 다음에 #(값)을 넣어서 parameter 값을 지정할 수 있다.

```
mux2 #(1) u1 (d0, d1, sel, out);
```

parameter가 여러 개인 경우에는 #(값1, 값2, 값3)와 같이 모듈에서 정의된 순서대로 parameter 값들을 컴마로 구분하여 넣는다. 모듈 인스턴스에서 parameter 값을 재지정하는 다른 두 가지 방법이 있다. 다음과 같이 #( ) 안에 또는 defparam을 사용하여 parameter 이름과 값을 함께 명시하여 parameter 값을 재지정할 수 있다.

```
mux2 #(.SIZE(16)) u2 (a, b, sel, y);        // u2의 SIZE를 16으로 지정

mux2 u3 (x, y, sel, z);                     // u3의 SIZE를 8로 지정
defparam u3.SIZE = 8;
```

여러 개의 parameter가 정의된 모듈의 인스턴스에서 일부 parameter의 값만 다시 지정하려면 이름과 값을 함께 명시하는 방법을 사용하는 것이 편리하다. 예 3.4는 mux2 모듈의 parameter값 SIZE를 1로 재지정한 모듈 인스턴스를 사용하여 그림 3.3의 회로를 설계한 것이다.

예 3.4 parameter값을 재지정한 모듈 인스턴스를 사용한 그림 3.3의 회로

```
module circuit(a, b, s, y);
    input   a, b;
    input   s;
    output  y;

    mux2 #(.SIZE(1)) u1 (a & b, a | b, s, y);

endmodule
```

변경가능한 값을 parameter로 지정하여 모듈을 정의하면 다른 모듈을 설계할 때에 필요에 따라서 parameter 값을 변경하여 모듈을 재사용할 수 있으므로 모듈을 재사용하기가 훨씬 용이해진다. 프로그램을 작성할 때에 라이브러리 함수를 사용하는 것 처럼 하드웨어를 설계할 때에도 미리 설계된 모듈을 사용할 수 있으며 많은 모듈들이 **IP**(intellectual property) 형태로 제공된다. IP는 다양한 요구조건에 맞추어 쉽게 재사용할 수 있도록 대부분 parameter를 사용한 모듈로 설계된다.

`define을 사용한 상수 정의

`define은 C언어의 #define과 같이 매크로 정의 기능을 수행하며 상수를 정의하는 데에도 사용할 수 있다. 다음과 같은 형식으로 `define을 사용하여 매크로 상수를 정의할 수 있으며 정의된 매크로는 `SIZE와 같이 이름 앞에 `을 붙여서 사용한다.

```
`define SIZE 4
```

예 3.5는 예 3.3을 parameter 대신에 `define을 사용하여 상수를 정의한 것이다.

예 3.5 `define 상수 정의를 사용한 2×1 멀티플렉서

```
module mux2(a, b, s, y);
    `define SIZE  4;              // parameter상수 정의
    input [`SIZE-1:0]   a, b;       // 데이터 입력
    input  s;                       // 선택 입력
    output [`SIZE-1:0] y;           // 데이터 출력

    assign y = s ? b : a;           // s=0이면 y=a, s=1이면 y=b
endmodule
```

3.6 피드백이 있는 연속할당문과 래치

피드백이 있는 연속할당문

기억소자는 출력이 입력으로 피드백(feedback)되는 회로로 구성된다. 그림 3.6은 D래치의 회로도이다. D래치는 **투명 래치**(transparent latch) 라고 부르기도 한다.

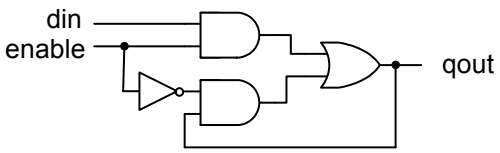


그림 3.6 D 래치의 회로도

이 회로를 부울함수를 사용하여 연속할당문으로 기술한 Verilog 문장은 다음과 같다.

```
assign qout = enable & din | ~enable & qout;
```

이 회로는 enable이 1일 때에는 출력 qout은 입력 din과 같게 되며 enable이 0일 때에는 출력 qout은 직전의 출력 qout과 같게 되어 현재의 출력상태가 그대로 유지된다. 즉 기억하는 상태가 된다. 앞에서 설명한 이 회로의 동작은 다음과 같이 조건연산자를 사용하여 연속할당문으로 나타낼 수 있으며 예 3.6은 그림 3.6의 D 래치 모듈의 완전한 Verilog 문장이다.

```
assign qout = enable ? din : qout;
```

예 3.6 피드백이 있는 연속할당문을 사용한 D래치 – 조건연산자 사용

```
module d_latch(qout, din,enable);
    output  qout;
    input   din, enable;

    assign qout = enable ? din : qout;
endmodule
```

그림 3.7은 예 3.6의 회로에 대한 시뮬레이션 파형이다. enable이 1일 때 입력 din의 파형이 출력 qout에 그대로 나타남을 확인할 수 있으며 enable이 0으로 된 후에는 qout의

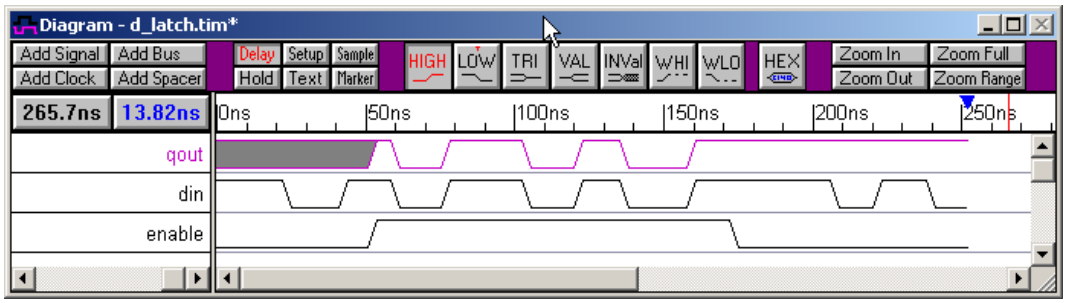


그림 3.7 D래치에 대한 시뮬레이션 파형

상태가 입력의 변화에 관계없이 그대로 유지됨을 알 수 있다.

그림 3.8은 D 래치의 심볼이다. D 래치는 순차회로의 기본 기능 단위로서 대개 미리 설계가 되어 있다. 위의 연속할당문과 같이 같은 신호가 입력과 출력에 모두 나타나 있으면 피드백이 있다는 것을 알 수 있으며, 합성도구는 피드백이 있는 연속할당문을 미리 준비된 래치를 사용하여 합성한다.

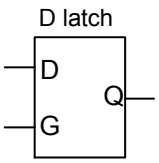


그림 3.8 D 래치 심볼

예를 들어서, 다음과 같이 피드백이 있는 연속할당문으로 기술된 회로는 그림 3.9과 같이 D 래치를 사용하여 합성될 수 있다.

```
assign y = en ? (a & b | c) : y;
```

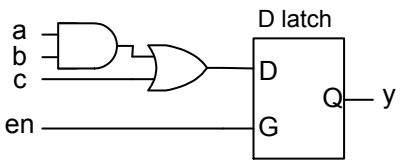


그림 3.9 피드백이 있는 식의 D 래치를 사용한 합성

S-R 래치

가장 간단한 기억소자는 SR(set-reset) 래치로서 NAND게이트 또는 NOR게이트를 사용하여 구성할 수 있는 데 NAND게이트를 사용한 SR 래치의 회로는 그림 3.10과 같다.

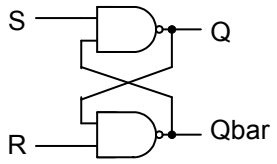


그림 3.10 SR 래치 회로

래치는 두 개의 출력 Q와 Qbar가 있는 데 정상적인 동작 상태에서는 Q와 Qbar는 서로 다른 상태가 되며 Q의 값을 래치가 기억하고 있는 상태로 다룬다. 이 회로는 S와 R이 모두 1일 때에는 인버터 2개가 서로 연결되어 있는 형태가 되어서 현재의 게이트 출력값이 그대로 보존된다. S가 0이 되면 Q가 1로, Qbar가 0으로 되어 1을 저장하고 있는 상태, 즉 set 상태가 된다. R이 0이 되면 Qbar가 1로, Q가 0으로 되어 0을 저장하고 있는 상태, 즉 reset 상태가 된다. S와 R이 모두 0이면 Q와 Qbar가 모두 1이 되며 Q와 Qbar가 같은 값이 되므로 기억되는 내용을 0인지 1인지를 정의할 수 없는 상태가 된다. 따라서 대부분 경우에는 S와 R이 모두 0인 상태가 되지 않도록 한다.

그림 3.10의 SR래치는 다음과 같이 연속할당문으로 나타낼 수 있다. 여기서 S는 set으로, R은 reset으로 표기하였다.

```
assign q = ~(set & qbar);
assign qbar = ~(reset & q);
```

위의 두 연속할당문을 결합하면 q와 qbar 모두 피드백이 있는 회로의 출력이라는 것을 알 수 있지만 각 연속할당문은 피드백이 없는 문장이 된다. 이렇게 피드백이 없는 문장은 합성도구가 래치를 사용한 합성을 하지 않을 수 있으며 래치를 사용하기 위해서 그림 3.10과 같은 회로를 사용자가 직접 설계하여 사용하는 것은 바람직하지 않다.

리셋 제어입력이 있는 D 래치

그림 3.7의 파형에서 보듯이 enable이 1이 되기 전까지의 출력은 그 이전에 저장된 내용이 없으므로 알 수 없는 상태가 된다. 이러한 점을 없애기 위해서 리셋 제어입력 reset을 추가하여 출력 값을 0으로 초기화하는 데에 사용할 수 있다. 예 3.7은 reset이

0일 때에는 출력 qout이 0이 되고 reset이 1일 때에는 출력이 예 3.6의 D 래치 회로의 출력과 같게 되는 회로를 설계한 것으로서 조건 연산자를 중첩하여 표현할 수 있다.

예 3.7 조건연산자를 사용하여 나타낸 리셋 제어입력이 있는 D 래치

```
module d_latch_reset(qout, din,enable, reset);
    output  qout;
    input   din, enable;

    assign qout = (~reset) ? 0 : (enable ? din : qout);
endmodule
```

조건 연산자는 우결합성을 가지며 다른 연산자에 비해서 우선순위가 낮으므로 위의 예의 식에서 두 개의 괄호는 모두 없어도 되지만 이해를 돕기 위해서 괄호를 사용하였다.

그림 3.11은 예 3.7에 대한 시뮬레이션 파형이다. reset의 초기값을 0으로하여 출력 qout을 0으로 초기화한 후에 reset을 1로 만들고 D 래치를 정상적으로 동작시켰다. 그림 3.7과는 다르게 시뮬레이션 초기에 출력 qout에 don't care 상태가 발생하지 않는다.

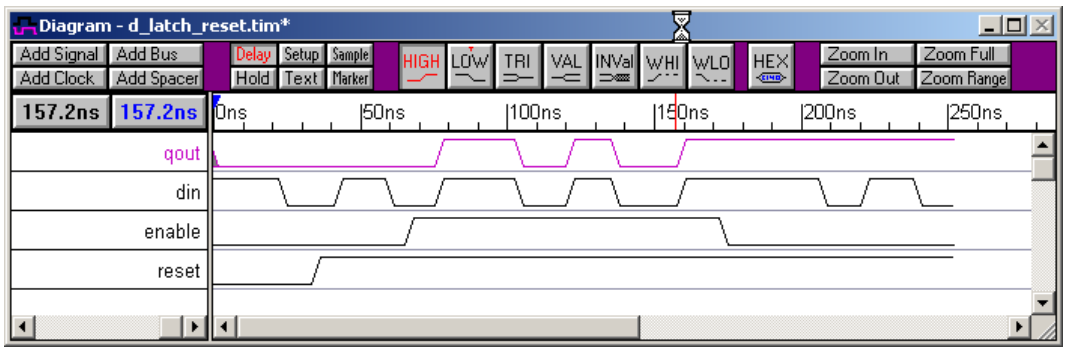


그림 3.11 리셋 제어입력이 있는 D래치의 시뮬레이션 파형

연속할당문의 제한점

연속할당문은 간단한 부울 함수, 3상태 출력 동작과 D래치 회로와 같이 작은 회로를 모델링하는 데에는 매우 편리하다. 그렇지만 입력 개수가 많고 복잡한 회로는 부울 함수로 표현하는 것이 쉽지 않고 부울 함수로 표현했더라도 설계의 내용을 이해하기 어려운 경우가 많다. 큰 회로는 연속할당문을 사용한 데이터흐름 모델링 방법 보

다는 완전한 동작적 모델링을 사용하는 것이 설계하기도 쉽고 설계된 내용을 이해하기도 쉽다.

그리고 Verilog는 AND, OR와 같은 비트단위 논리연산자와 조건연산자 이외에도 다음에서 소개하는 여러 가지 연산자들을 제공한다. 연속할당문을 기술할 때에 이러한 연산자들을 사용하면 회로의 데이터 흐름 모델링을 훨씬 간결하게 나타낼 수 있다.

3.7 여러 가지 연산자

Verilog에서는 동작적 모델링을 위해서 다양한 연산자를 제공한다. 우리는 이미 비트단위 논리연산자와 조건연산자에 대해서 배웠다. Verilog 연산자는 대부분 C언어의 연산자와 같지만 C언어에서 제공하지 않는 연산자들도 제공한다.

결합연산자

결합연산자(concatenation operator) { } 는 다음과 같이 여러 개의 피연산자를 결합하여 더 큰 크기의 벡터를 만드는 데 사용한다.

```
{ 1'b1, 2'b01 }     // 결과: 3'b101
{ a, b, c }         // | a | b | c |
```

결합연산자를 사용하기 위해서는 피연산자의 크기가 정해져 있어야 한다. 그리고 다음과 같이 숫자를 앞에 사용하면 반복하여 결합이 된다.

```
{ 3{ A } }          // { A, A, A }
{ A, 2{B} }         // { A, B, B }
{ 2{3'b011} }       // 6'b011011
```

등가연산자

등가연산자는 값의 비교 방법에 따라서 다음과 같이 두 종류가 있다.

- 논리 등가 연산자: == (등가), != (부등가)
  결과는 0, 1 또는 x가 가능하며 x 또는 z값과 비교할 때에 결과가 x이다.
- 케이스(case) 등가 연산자: === (등가), !== (부등가)
  x 또는 z값도 정확하게 비교하며 결과는 0 또는 1이다.

다음은 여러 가지 등가연산자의 사용 예이다.

```
4'b1010 == 4'b1xxz  // x
4'b1010 != 4'b1xxz  // x
4'b0010 == 4'b1xxx  // 0 (거짓)
4'b1010 === 4'b1xxz // 0 (거짓)
4'b1010 !== 4'b1xxz // 1 (참)
4'b1xxz === 4'b1xxz // 1 (참)
```

등가 연산자 ==와 !=는 비교기 회로를 사용하여 합성될 수 있다. 이에 비해서 케이스 등가연산자는 대개 하드웨어로 합성이 되지 않으며 시뮬레이션 용으로만 사용한다.

논리연산자

다음과 같이 C 언어와 같은 논리연산자가 제공된다.

- `&&` 논리 AND
- `||` 논리 OR
- `!` 논리 NOT

논리연산자의 결과는 피연산자의 비트 크기와 관계없이 1비트로서 참(1)과 거짓(0)을 나타낸다. 피연산자가 0이 아닌 경우에 참으로 인식하지만 피연산자가 x 또는 z값을 가질 경우에 거짓으로 인식한다. 이에 비해서 비트단위 논리연산자는 결과의 크기가 피연산자의 크기와 같다. 논리연산자는 논리 게이트를 사용하여 쉽게 합성된다.

예 3.8은 같은 부울함수 식을 결합연산자, 등가연산자, 논리연산자를 사용하여 여러 가지 방법으로 나타낸 것이다.

예 3.8 같은 부울함수식의 여러 가지 표현

```
assign lt = ~a & b;
assign eq = ~a & ~b | a & b;
```

```
assign lt = {a, b} == 2'b01;
assign eq = {a, b} == 2'b00 || {a, b} == 2'b11;
```

```
wire [1:0] ab = {a, b};

assign lt = (ab == 2'b01);
assign eq = (ab == 2'b00) || (ab == 2'b11);
```

축소연산자

축소연산자(reduction operator)는 벡터 피연산자의 모든 비트에 대해서 논리연산 수행하여 1비트의 결과를 제공하는 것으로서 다음과 같은 것들이 있다.

- &x reduction AND
- ~&x reduction NAND
- |x reduction OR
- ~| reduction NOR
- ^x reduction XOR
- ~^, ^~ reduction XNOR

예를 들어서 다음 두 연속할당문은 같은 동작을 나타낸다.

```
assign y = a[0] & a[1] & a[2] & a[3];
assign y = &a;
```

축소연산자는 벡터의 모든 비트들이 관련된 게이트의 입력에 연결되는 회로로 합성이 된다. 그림 3.11은 4비트 벡터 a에 대해 축소연산자 &를 수행했을 때의 합성 예이다.

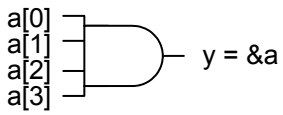


그림 3.11 축소연산자의 합성

산술연산자, 관계연산자와 쉬프트연산자

산술연산자는 덧셈(+), 뺄셈(-), 곱셈(*), 나눗셈(/), 나머지(%) 연산자가 제공된다. 산술연산 시에 피연산자가 x 또는 z값을 가지면 결과는 x이다. 그리고 단항연산자로 + 와 - 연산자를 사용할 수 있다.

크기 비교를 위해서 관계연산자 >, >=, <, <= 를 사용하며 피연산자가 x 또는 z값을 가지면 결과는 x이다.

시프트(shift)연산자는 C언어와 다르게 다음과 같이 4개의 연산자를 제공된다.

- >>, << : 논리시프트 연산 (빈자리에 0이 채워짐)
- >>>, <<< : 산술시프트 연산 (>>>는 최상위비트가 보존되며, <<<는 <<와 같다.)

산술연산자, 관계연산자, 시프트연산자로 기술된 하드웨어는 간단한 게이트가 아닌 각 연산을 수행하는 미리 설계된 회로로 합성이 된다.

지금까지 소개한 연산자들의 우선순위는 표 3.1과 같다.

표 3.1 연산자 우선순위

우선순위	종류	연산자
1	단항	! ~ + -
2	산술연산	* / %
3		+ -
4	시프트	>> << >>> <<<
5	관계	< <= > >=
6		== != === !==
7	비트논리	&
8		^
9		|
10	논리	&&
11		||
12	조건	?:

3.8 예제 – 비교기의 여러 가지 모델링

이 절에서는 2장에서 구조적 모델링을 사용하여 설계한 비교기를 연속할당문을 사용하여 여러 가지 형태로 설계한 예를 소개한다. 예 3.9은 2비트 비교기를 비트논리연산자를 사용하여 부울함수로 나타낸 것이다.

예 3.9 비트논리연산자를 사용한 2-bit 비교기

```
module cmp2(lt, gt, eq, A1, A0, B1, B0);
   input A1, A0, B1, B0;
   output  lt, gt, eq

   assign  lt = ~A1 & B1 | ~A1 & ~A0 & B0 | ~A0 & B1 & B0;
   assign  gt = A1 & ~B1 | A0 & ~B1 & ~B0 | A1 & A0 & ~B0;
   assign  eq = ~A1 & ~A0 & ~B1 & ~B0 | ~A1 & A0 & ~B1 & B0
                     | A1 & A0 & B1 & B0  | A1 & ~A0 & B1 & ~B0;
endmodule
```

예 3.9와 같은 비트논리연산자를 사용한 설계는 부울함수가 비트 수에 따라서 변경되어야 하므로 일반적인 n비트 비교기에 대해서 사용하기가 어렵다. 예 3.10과 같이 결합연산자를 사용하여 입력들을 2비트로 결합한 후에 관계연산자를 사용하여 비교하면 구체적인 부울함수 식을 유도할 필요가 없으므로 예 3.9의 표현보다 간결하며 실수할 가능성이 훨씬 줄어든다.

예 3.10 결합연산자와 관계연산자 사용한 2-bit 비교기

```
module cmp2_CA1(lt, gt, eq, A1, A0, B1, B0);
   input A1, A0, B1, B0;
   output  lt, gt, eq

   assign  lt = {A1,A0} < {B1,B0};
   assign  gt = {A1,A0} > {B1,B0};
   assign  eq = {A1,A0} == {B1,B0};
endmodule
```

관계연산자는 미리 설계된 비교기를 사용하여 합성될 수 있다. 예 3.11은 입력 포트를 벡터로 선언한 것으로서 2비트 비교기를 임의의 n-bit 비교기로 일반화하기 쉽다.

예 3.11 포트를 벡터로 선언한 2-bit 비교기

```
module cmp2_CA2(lt, gt, eq, A, B);
   input  [1:0]  A, B;
   output  lt, gt, eq

   assign  lt = A < B;
   assign  gt = A > B;
   assign  eq = A == B;
endmodule
```

설계의 일반화를 위해서 모듈을 정의할 때에 비교 데이터의 비트 수를 예 3.12과 같이 parameter를 사용하여 지정할 수 있다.

예 3.12 parameter를 사용한 2-bit 비교기

```
module comparator(lt, gt, eq, A, B);
   parameter SIZE = 2;
   input  [SIZE-1:0]  A, B;
   output  lt, gt, eq

   assign   lt = A < B,
            gt = A > B,
            eq = A == B;
endmodule
```

예 3.12에서 설계된 비교기 comparator는 다음과 같이 모듈 인스턴스를 만들때에 파라메터를 재지정하여 임의의 길이의 데이터 비교에 사용될 수 있다.

```
   comparator #(16) u1 (lt, gt, eq, A, B);      // parameter 재지정
```

4장 Verilog 동작적 모델 논리 설계

우리는 앞에서 구조적 모델과 데이터흐름 모델과 사용한 설계에 대해서 배웠다. 디지털 시스템의 설계가 복잡해짐에 따라서 게이트나 모듈들 간의 연결에 대해서 기술하는 구조적 모델과 입출력 간의 관계식으로 기술하는 데이터흐름 모델링 방법이외에 보통의 프로그램을 작성하는 것처럼 입력으로부터 출력을 얻는 과정, 즉 알고리즘을 기술하는 방법을 사용하게 되었다. 그리고 구조적 모델이나 데이터흐름 모델링으로 나타내기 어려운 레지스터를 포함한 회로의 동작을 표현하는 방법도 필요하게 되었다. 이 장에서는 이러한 레지스터 기반과 알고리즘에 기반을 둔 동작적 모델 논리 설계에 대해서 다룬다.

4.1 순차 처리문

논리회로는 입력으로부터 출력을 얻는 과정을 보통의 프로그램에서와 같이 순차적인 절차로 기술할 수 있으며 **initial문** 또는 **always문**과 같은 순차 처리문이 이러한 절차를 기술하기 위해서 사용된다. initial문은 처음에 한 번만 수행되는 동작을, always문은 신호가 변할 때마다 반복하여 수행되는 동작을 기술한다.

순차 처리문 내에서는 보통의 프로그래밍 언어에서와 같이 할당문, 조건문, 반복문 등이 사용될 수 있다. 순차 처리문은 내부에서 기술한 문장들은 순차적으로 실행되지만 다른 순차처리문과 모듈/프리미티브 인스턴스 및 연속할당문들과는 병렬로 수행된다. 그림 4.1은 병렬로 수행되는 Verilog의 문장들을 나타낸다.

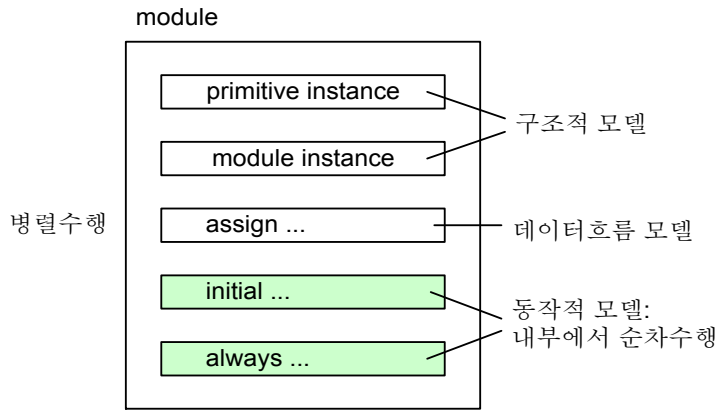


그림 4.1 병렬로 수행되는 Verilog 문장들

initial문

테스트벤치를 설명할 때에 이미 initial문에 대해서 보았다. initial 문은 시뮬레이션 시작할 때에 한 번만 실행하며 여러 개의 initial문이 있으면 이들은 독립적으로 실행한다. initial문의 형식은 다음과 같다. 실행하는 문장이 여러 개가 있으면 begin과 end를 사용하여 블록으로 묶어야 하며 블록 내의 문장들은 순차적으로 실행된다.

```
initial 문장;              // 하나의 문장 포함

initial begin              // 여러 개의 문장들을 포함
   문장;
   문장;
   ...
end
```

initial문은 일반적으로 신호의 초기화, 모니터링을 위한 파형출력과 시뮬레이션 수행시에 한 번만 수행되어야 하는 과정을 기술하는 데 사용한다. initial문은 주로 시뮬레이션 용도로 사용하며 합성도구에서는 지원되지 않을 수 않다.

always문

always문은 반복조건이 만족할 때마다 문장을 반복하여 수행하며 사용 형식은 다음과 같다. 실행되는 문장이 여러 개이면 begin과 end를 사용하여 블록으로 설정해야 하며 이 문장들은 순차적으로 실행된다.

```
always 반복조건
   문장;

always 반복조건  begin            // 여러 개의 문장들을 포함
   문장;
   문장;
   ...
end
```

예 4.1은 initial과 always문을 사용하여 기술한 클록 발생 회로이다. 예 4.1에서 initial문은 clock의 초기값과 시뮬레이션 종료시간을 지정하는 데에 사용되었다. always 문은 20 단위시간 마다 clock 값이 반대로 바뀌는 동작을 기술하였으며 주기가 40 단위시간인 클럭을 발생시킨다.

예 4.1 클럭발생회로

```
module clock_generator;
    reg     clock;

    initial begin
        clock = 0;              // clock을 0으로 초기화
        #500 $finish;           // 500 단위시간 후에 시뮬레이션 종료
    end
    always
        #20 clock = ~clock;     // 20 단위시간 마다 clock 신호 반전
endmodule
```

예 4.1에서 시간지연을 나타내는 #20이 없다면 always문은 멈추지 않고 계속하여 반복하게 실행될 것이다. #20은 20 단위시간 동안 동작 수행을 멈춘다. 그림 4.2는 예 4.1의 회로에서 생성되는 clock의 파형을 나타낸다.

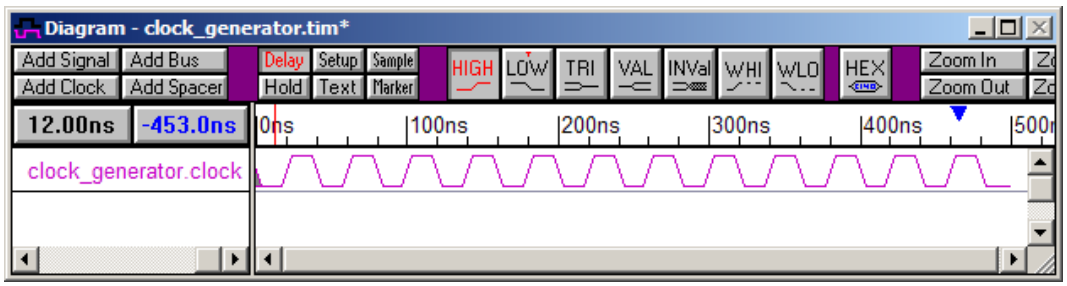


그림 4.2 클럭발생회로의 clock 파형

하드웨어의 동작은 전원이 공급되었을 때부터 시작하여 지속적으로 반복되는 동작으로 볼 수 있다. 조합회로는 입력이 변화하면 출력이 변할 수 있으며, 순차회로는 대개 클록이 특정 조건을 만족할 때에 출력이 변한다. 이러한 하드웨어의 동작은 출력이 변할 수 있는 조건이 발생하였을 때에 출력 값을 다시 계산하여 지정하는 동작이 반복되는 것으로 기술할 수 있다. always문은 이러한 동작을 나타내는 데 적합하다.

always문의 반복조건은 조합회로나 순차회로의 출력 값이 갱신되는 시점을 기술하며 이벤트 발생, 시간 지연 등을 표현하는 타이밍 제어문으로 나타낸다. always문은 구조적인 모델링이나 수식으로 동작을 기술하기 힘든 순차회로의 동작을 기술하는 데에 특히 유용하다. 표 4.1은 initial 문과 always 문의 특징을 비교한 것이다.

표 4.1 initial과 always의 비교

종류	형식	동작	논리합성
initial	initial begin . . . end	(1회 동작) 시뮬레이션을 시작할 때에 한 번 수행	지원되지 않음
always	always 반복조건 begin . . . end	(반복 동작) 반복조건을 만족할 때마다 반복 수행	지원됨

4.2 타이밍 제어

타이밍 제어문은 표 4.2과 같이 **지연 제어**, **이벤트 제어**, **wait문**의 3가지 형태가 있다. 타이밍 제어문을 만나면 수행을 멈추었다가 타이밍 제어문에서 주어진 조건을 만족하면 다시 수행을 재개한다. 이 세 가지 타이밍 제어문 중에서 이벤트 제어문 만 합성 도구에서 지원을 하며 나머지는 시뮬레이션 목적으로만 사용할 수 있다. 타이밍 제어문은 always 문의 반복 조건 뿐 만 아니라 순차 처리문의 임의의 문장과 함께 사용하여 문장의 동작에 대한 타이밍을 제어할 수 있다.

표 4.2 타이밍 제어

종류	형식	내용
지연 제어	#10	10 단위시간이 지연됨
이벤트 제어	@(a)	신호 a의 변화를 기다림
레벨 제어	wait (a==0)	신호 a가 0과 같기를 기다림

지연 제어

지연 제어는 다음과 같은 형식으로 지연 시간을 기술한다.

```
# 시간
```

지연 제어를 만나면 주어진 시간이 지연된 후에 문장을 수행을 한다. 지연 제어는 논리 합성이 지원되지 않는다. 테스트벤치를 작성할 때에 이미 지연 제어를 사용하였다. 지연 제어는 예 4.2의 첫 번째 예와 같이 문장 앞에 표시하는 방법과 두 번째 예와 같이 할당문 사이에 표시하는 두 가지 종류가 있다. 세 번째 예는 두 방법을 혼합한

것이다.

예 4.2 지연 제어

```
#10 y = x;            // 10 단위시간 후에 y = x 실행
z = #20 x + y;        // x + y를 계산하고 20 단위시간 후에 z에 할당
#15 w = #10 x + z;    // 15 단위시간 후에 x + z를 계산하고 10 단위시간 후에 w에 할당
```

문장 앞에 지연시간을 표시하면 표시된 시간 만큼 지연된 후에 문장이 실행된다. 할당문 사이에 지연시간을 표시하면 할당문의 오른쪽 수식을 실행하여 얻은 결과를 표시된 시간 만큼 지연된 후에 왼쪽 변수에 저장한다. 지연 제어는 시뮬레이션 용으로 사용되며 논리 합성이 지원되지 않는다.

이벤트 제어

이벤트는 신호가 변화되는 것을 의미한다. 이벤트 제어는 다음과 같이 괄호 안에 변화하기를 기다리는 신호를 기술한다.

```
@ (var)                    // var가 변할 때에 문장 수행
```

이벤트 제어문을 만나면 괄호 안의 신호가 변할 때까지 기다리며 해당 신호가 변하면 이어지는 문장의 수행을 계속한다. 이벤트는 신호가 0에서 1로 변하거나 1에서 0으로 변할 때에 모두 발생한다.

예 4.3 이벤트 제어

```
always @(a)
    x = a & b;        // a가 변할 때마다 x값이 다시 계산됨
```

예 4.3은 이벤트 제어를 사용한 always문의 예이다. 신호 x는 a의 변화에는 영향을 받아서 a & b의 값이 할당되지만 b의 변화에는 영향을 받지 않는다. x가 b의 변화에 영향을 받지 않으므로 논리합성될 때에 기억장치가 포함될 수 있다.

0에서 1로와 1에서 0으로의 두 가지의 신호 변화들 중 하나만 나타내려면 다음과 같이 posedge와 negedge를 함께 사용한다.

```
@ (posedge clock)          // clock이 0에서 1로 변할 때에 수행 (상승에지)
@ (negedge clock)          // clock이 1에서 0으로 변할 때에 수행 (하강에지)
```

이러한 이벤트 제어문은 에지트리거(edge-triggerred) 동작을 하는 플립플롭이나 순차회로의 동작을 기술할 때에 매우 유용하다.

그림 4.3의 D플립플롭은 clock의 상승 에지에서 입력 D의 값이 출력 Q에 저장되는 동작을 수행한다. D 플립플롭의 동작은 예 4.4와 같이 이벤트를 posedge clock으로 사용한 always 문을 사용하여 나타낼 수 있다.

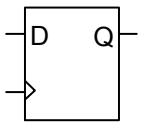


그림 4.3 D 플립플롭

예 4.4 에지트리거 이벤트 제어 – D플립플롭

```
always @(posedge clock)
    q = d;        // clock의 상승에지에서 q = d 수행
```

클록의 에지에서 출력이 변화하는 예 4.4와 같은 구문은 합성도구에서 미리 설계된 D 플롭플롭으로 합성된다.

여러 개의 신호들 중에서 하나라도 변할 때에 이벤트가 발생하도록 하려면 다음과 같이 or를 사용하여 신호들을 나열하며 Verilog-2001에서는 or 대신에 ,를 사용하여 신호들을 나열할 수 있다.

```
@(var1 or var2 or var3)          // var1, var2, var3 중 하나라도 변할 때에 수행
@(var1, var2, var3)              // 위 구문의 Verilog-2001 표현
@(posedge clock or negedge reset)   // clock의 상승에지 또는 reset의 하강에지
                                       에서 수행
@(posedge clock, negedge reset)     // 위 구문의 Verilog-2001 표현
```

이벤트 제어문의 괄호 안에 나열된 신호들의 목록을 **감지 목록**(sensitivity list)이라고 부른다.

다음과 같이 보통 이벤트와 에지트리거 이벤트를 감지 목록에 함께 기술할 수 있다. 그렇지만 이러한 형태의 이벤트 제어는 합성도구가 하드웨어로 제대로 합성을 해주지 못할 수가 있으므로 사용하지 않는 것이 바람직하다.

```
@(posedge clock or var)      // clock의 상승에지 또는 var가 변할 때에 수행
```

이벤트 제어를 사용한 조합회로 모델링

always 문은 플립플롭이나 래치와 같이 기억장소를 포함한 회로의 모델링에 주로 사용하지만 조합회로의 모델링에도 사용할 수 있다. 조합회로는 모든 입력에 대해서 출력이 정의되며, 입력이 변할 때마다 출력값을 다시 지정해야 한다. 모든 입력을 감지목록에 포함시키고 always 문 안에 입출력 관계를 나타내는 수식을 기술하면 입력이 변할 때마다 입출력관계 수식이 실행되어 출력값이 재지정되므로 조합회로 동작을 나타낼 수 있다. 예 4.5는 조합회로인 반가산기를 always문을 사용하여 기술한 것이다.

예 4.5 조합회로의 always 문을 사용한 모델링 – 반가산기

```
always @(a or b) begin       // 모든 입력을 감지목록에 포함
    sum = a ^ b;
    cout = a & b;
end
```

Verilog-2001에서는 always 문 내의 문장에서 모든 입력들을 감지 목록에 나열하는 것 대신에 예 4.6과 같이 @* 으로 간편하게 나타낼 수 있다.

예 4.6 조합회로의 always 문을 사용한 모델링 – 반가산기 (Verilog-2001)

```
always @* begin      // 모든 입력(a, b)을 감지목록에 포함
    sum = a ^ b;
    cout = a & b;
end
```

복잡한 조합회로를 always 문으로 기술할 때에 일부 입력을 감지목록에 넣지 않는 실수를 하기 쉬우므로 Verilog-2001이 지원되는 경우에는 감지목록에 모든 입력들을 명시적으로 나열하는 것 대신에 예 4.6과 같이 @*을 사용하는 것을 권장한다.

레벨 제어

wait문은 수식이 참이 될 때까지 문장의 수행을 지연시킨다. wait문의 형식은 다음과 같으며 괄호 안에 조건식을 기술한다.

```
wait(a == 0)                 // a가 0일 때에 수행
wait(enable)                 // enable이 참(1)일 때에 수행
```

wait문은 신호가 특정한 레벨이면 문장을 수행하도록 하는 **레벨 제어**를 할 수 있다. wait문은 대개 논리합성이 지원되지 않는다.

예 4.7의 always문은 타이밍 제어에 wait문을 사용한 예로서 enable이 1인 동안 20 단위시간마다 count값이 증가하는 동작을 수행한다.

예 4.7 wait문을 사용한 타이밍 제어

```
always wait(enable)
    #20 count = count + 1;
```

4.3 순차 할당문

기호 '='는 보통의 프로그래밍 언어에서와 같이 오른쪽 수식의 결과를 왼쪽의 변수에 저장하는 할당문에 사용된다. 우리는 이미 assign과 함께 사용하는 연속할당문을 보았다.

할당문은 always와 initial과 같은 순차 처리문 안에서 사용될 수 있으며 순차 처리문 안에 여러 개의 할당문이 있는 경우에 보통의 프로그래밍 언어에서와 같이 순서대로 수행하므로 순차처리문에서 사용되는 할당문을 **순차할당**(procedural assignment)**문**이라고 한다. 순차할당문에서 값이 저장되는 왼쪽 변수는 시뮬레이션동안 값을 저장해야 하므로 reg나 integer와 같은 variable형으로 선언되어야 한다. 변수가 실제 신호와 관계가 있으면 reg로 선언하고, 그렇지 않으면 integer로 선언하는 것이 일반적이다. 예 4.8은 예 4.5의 반가산기를 완전한 Verilog 모듈로 작성한 예이다.

예 4.8 반가산기

```
module halfadder(a, b, sum, cout);
    input a, b;
    output sum, cout;
    reg sum, cout;      // 순차할당문의 왼쪽 변수

    always @(a or b) begin
        sum = a ^ b;
        cout = a & b;
    end
endmodule
```

이 예에서 sum과 cout은 always 내의 순차할당문에서 저장되는 변수이므로 reg형으로 선언하였다. sum과 cout은 reg형으로 선언되지 않으면 컴파일을 할 때에 오류 메시지가 출력된다. reg형으로 선언되지 않는 output 신호는 wire 형으로 간주되므로 sum과 cout은 output과 reg로 이중으로 선언해야 한다. Verilog-2001에서는 모듈의 output이며 reg형인 신호는 다음과 같이 한꺼번에 선언할 수 있다.

```
output reg sum, cout;       // Verilog 2001에서 지원
```

always 문 안에서 사용하는 순차할당문들은 예 4.8과 같이 앞에 있는 순차할당문의 결과를 다음 순차할당문에서 사용하지 않는 경우에는 순서를 바꾸어도 관계없지만, 앞의 결과를 사용하는 경우에는 소프트웨어에서와 같이 순차할당문의 사용 순서가 결과에 영향을 미친다.

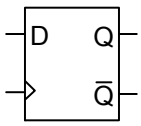


그림 4.4 보수 출력을 갖는 D 플립플롭

그림 4.4의 블록도로 표시되는 보수출력을 갖는 D 플립플롭은 클럭의 상승 에지에서 입력 D가 출력 Q에 저장되고 출력 $\overline{Q}$에는 출력 Q의 보수가 출력되며 예 4.9와 같이 동작을 나타낼 수 있다.

예 4.9 상승에지 트리거 D 플립플롭

```
module dff(clock, data, q, qbar);
    input   data, clock;
    output  q, qbar;
    reg     q, qbar;        // 순차할당문의 왼쪽 변수

    always @(posedge clock) begin
        q = data;           // 정상출력
        qbar = ~q;          // 보수출력
    end
endmodule
```

여기서 보수출력을 기술하는 두 번째 할당문은 첫 번째 할당문의 결과를 입력으로 사용하고 있으며, 두 할당문의 순서를 바꾸면 다른 결과가 나오며 잘못 기술한 것이 된다.

assign과 함께 사용되는 연속할당문은 서로 병렬로 수행하는 하드웨어를 나타내는 것이며 기술하는 순서에 관계없이 같은 동작을 수행하지만 always나 initial 내에서 사용되는 순차할당문은 병렬로 수행되는 하드웨어 동작을 나타내는 것이 아니라 하드웨어의 입출력 관계를 프로그램 작성 방법으로 기술하는 데 사용하는 것으로서 보통의 프로그램처럼 순서대로 수행되므로 할당문의 순서가 중요하다. 할당문의 순서에 따라서 서로 다른 동작을 할 수 있으므로 주의해야 한다.

예 4.10은 예.4.9에서 출력 qbar를 always 내의 순차할당문 대신에 assign을 사용한 연속할당문으로 기술한 예이다.

예 4.10 상승에지 트리거 D 플립플롭 – qbar를 연속할당문으로 기술

```
module dff(clk, data, q, qbar);
    input   data, clock;
    output  q, qbar;
    reg     q;              // 순차할당문의 왼쪽 변수

    assign qbar = ~q;       // 보수출력
    always @(posedge clock) begin
        q = data;           // 정상출력
    end
endmodule
```

여기서 qbar에 대한 연속할당문은 always문과 병렬로 동작하며 always문과의 상대적인 위치에 상관없이 항상 q의 보수값을 출력한다. 연속할당문의 왼쪽변수 qbar는 reg형이 아니어야 하므로 reg형 선언에서 제외시켰다.

reg형 변수와 하드웨어 레지스터와 차이를 구분하는 것이 중요하다. reg형 변수는 예 4.9와 예 4.10과 같이 하드웨어 레지스터를 모델링하는 데에 사용되며 많은 경우에 하드웨어 레지스터로 합성이 되지만 예 4.8과 같이 always를 사용하여 동작을 기술하는 조합회로의 출력에 대해서도 사용한다.

4.4 조건문

순차처리문 내에서는 보통의 프로그래밍 언어에서처럼 조건문과 반복문을 사용할 수 있다. 조건문에는 if문과 case문 등이 있다.

if문

if문은 다음과 같이 세 가지 형태로 사용된다.

```
if (조건식) 문장            // if문: 조건식이 참일 때 문장 수행

if (조건식) 문장1           // if-else문:
else 문장2                  // 조건식이 참일 때 문장1, 거짓일 때 문장2 수행

if (조건식1) 문장1          // 중첩된 if-else문:
else if (조건식2) 문장2     // 조건에 따라서 여러 문장 중 하나를 수행
else if (조건식3) 문장3
...
else 기본문장
```

각 조건을 만족할 때에 수행하거나 else에 대한 문장이 여러 개이면 begin과 end를 사용한 블록문 안에 기술한다.

그림 4.5와 같은 reset과 set 제어신호가 있는 D 플립플롭에 대한 동작적 모델을 if - else 문을 사용하여 기술하여 보자. reset과 set 제어신호는 동기식과 비동기식으로 동작하는 두 종류가 있다. 동기식 제어신호를 사용하는 D 플립플롭은 클럭의 상승에지에서 reset이 0이면 출력 Q가 0이 되고 set이 0이면 출력 Q가 1이 되며 두 제어신호가 모두 1이면 입력 데이타가 출력 Q에 저장된다. 예 4.11은 이러한 동기 set, reset 제어신호가 있는 D 플립플롭에 대한 동작적 모델이다.

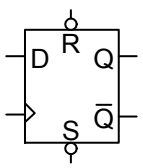


그림 4.5 set과 reset 제어입력이 있는 D 플립플롭

예 4.11 동기 set, reset 제어신호가 있는 D 플립플롭

```
module dff_s (q, qbar, data, set, reset, clk);
    output  q, qbar;
    input   data, set, reset, clk;
    reg     q;

    assign  qbar = ~q;
    always @ (posedge clk) begin
        if (reset == 0) q = 0;
        else if (set == 0) q = 1;
        else q = data;
    end
endmodule
```

동기식 제어신호를 사용하는 경우에 출력은 모두 클럭과 동기가 되어 변화되므로 예 4.11의 감지목록에는 posedge clk만 사용한다.

예 4.12는 reset과 set 제어신호가 클럭과 비동기로 동작하는 D 플립플롭에 대한 동작적 모델이다. 비동기식 제어신호를 사용하는 D 플립플롭은 reset이 0이면 클럭과 관계없이 출력 Q는 즉시 0이 되고 set이 0이면 출력 Q는 즉시 1이 된다. 그리고 두 제어신호가 모두 1이면 클럭의 상승에지에서 입력 데이타가 출력 Q에 저장된다.

예 4.12 비동기 set, reset 제어신호가 있는 D 플립플롭

```
module dff_s (q, qbar, data, set, reset, clk);
    output  q, qbar;
    input   data, set, reset, clk;
    reg     q;

    assign  qbar = ~q;
    always @ (negedge reset or negedge set or posedge clk) begin
        if (reset == 0) q = 0;
        else if (set == 0) q = 1;
        else q = data;
    end
endmodule
```

비동기 제어신호를 사용하는 경우에 reset 또는 reset이 0이 될 때에 클럭에 관계없이

출력이 즉시 0 또는 1로 바뀌므로 예 4.12의 감지 목록에 negedge reset고 negedge set이 추가된다. 클럭신호와 비동기 제어신호의 차이점은 비동기 제어신호는 감지목록에도 포함되며 동작을 기술하는 if 문의 조건식에도 나타나지만 클럭은 감지목록에만 포함되며 동작기술문에 나타나지 않는 것이다.

예 4.13 transparent D 래치

```
module dlatch(enable, data, qout);
    input enable, data;
    output qout;
    reg qout;

    always @(enable or data)
        if (enable) qout = data;
endmodule
```

예 4.13는 3장의 예 3.6에서 소개했던 D 래치를 if문을 사용하여 기술한 예이다. 이 모델에서 enable이 1일 때에 할당문이 실행되어 data가 qout에 전달되며 enable이 0일 때에는 할당문이 실행되지 않는다. 할당문이 실행되지 않으면 reg형의 변수 qout은 변하지 않고 이전 값을 그대로 유지한다. 이처럼 always 문을 사용하면 예 3.6과 같이 피드백을 명시적으로 나타낼 필요가 없다. D 래치는 enable과 data의 변화가 출력 qout에 즉시 영향을 줄 수 있으므로 두 신호가 감지목록에 포함된다.

예 4.14 4비트 2×1 멀티플렉서

```
module mux2_1(sel, a, b, y);
    input sel;
    input [3:0] a, b;
    output [3:0] y;
    reg [3:0] y;

    always @(sel or a or b) begin    // verilog-2001은 @* 사용 가능
        if (sel==0) y = a;
        else y = b;
    end
endmodule
```

예 4.14는 if-else문을 사용하여 조합회로인 4비트 2×1 멀티플렉서를 기술한 예이다. 조합회로를 always 문으로 기술하기 위해서는 이전에 언급한 바와 같이 다음 조건을 만족시켜야 하며 예 4.14는 이 조건을 만족한다.

- 모든 입력이 always의 감지목록에 포함되어야 한다. (Verilog-2001 에서는 @* 사용)
- 입력의 모든 경우에 대해서 출력이 정의되어야 한다.

만약에 감지목록에 포함되지 않는 입력이 있거나 출력이 정의되지 않은 경우가 존재하면 기억소자가 포함되어 논리합성 되므로 always 문을 사용하여 조합회로 설계 시에 주의를 요한다. 예를 들어 2×1 멀티플렉서를 다음과 같이 기술하면 sel이 1‘bx인 경우에는 y가 정의되어 있지 않으므로 이전 값이 유지되는 것으로 간주되어 기억소자를 포함할 수도 있다. 그러므로 if문을 사용한 조합회로의 기술은 예 4.14와 같이 else로 끝나도록 해야 한다.

```
always @(sel or a or b) begin    // verilog-2001은 @* 사용 가능
    if (sel==0) y = a;
    else if (sel==1) y = b;
end
```

예 4.15 1비트 4x1 멀티플렉서

```
module mux4_1(sel, d0, d1, d2, d3, y);
    input [1:0] sel;        // 선택신호
    input d0, d1, d2, d3;   // 4개의 1비트 입력
    output y;
    reg y;

    always @(sel or d0 or d1 or d2 or d3) begin // verilog-2001은 @* 사용 가능
        if (sel==0) y = d0;
        else if (sel==1) y = d1;
        else if (sel==2) y = d2;
        else y = d3;
    end
endmodule
```

예 4.15는 중첩된 if-else문을 반복 사용하여 1비트 4×1 멀티플렉서를 기술한 예이다. 이 예에서도 모든 입력이 감지목록에 포함되었고 모든 경우에 출력이 할당되었다.

case문

case문은 수식의 값에 따라서 실행할 문장을 선택하는 다중 분기문이다. case문의 형식은 다음과 같다. endcase 다음에 세미콜론이 없음을 유의하자

```
case (수식)
    값1: 문장1;
    값2: 문장2;
    값3: 문장3;
    ...
    default: 기본문장;   // 일치하는 값이 없으면 기본문장 수행
endcase
```

수식의 값에 따라서 해당되는 문장이 수행되며 수식의 값과 일치하는 값이 없으면 default로 지정된 문장을 수행한다. 여러 개의 값들이 같은 문장을 수행한다면 다음과 같이 여러 개의 값들을 콤마로 구분하여 나열한다.

```
    값1, 값2, ... : 문장
```

4×1 멀티플렉서는 case 문을 사용하여 예 4.16과 같이 간결하게 기술할 수 있다.

예 4.16 1비트 4×1 멀티플렉서

```
module mux4_1(sel, d0, d1, d2, d3, y);
    input [1:0] sel;
    input d0, d1, d2, d3;
    output reg y;

    always @(*) begin
        case (sel)
          0: y = d0;
          1: y = d1;
          2: y = d2;
          3: y = d3;
          default: y = 1'bx;
        endcase
    end
endmodule
```

여기서 0부터 3까지의 모든 경우에 대한 수행문을 기술했지만 Verilog는 0과 1 이외에도 x와 z를 사용하는 4값 논리를 사용하므로 0부터 3까지의 값에 속하지 않는 경우가 발생할 수 있다. 그러므로 default에 대한 문장도 포함해야 논리 합성 시에 불필요한 기억소자가 사용되는 것을 방지할 수 있다. 예 4.16의 case 문은 다음과 같이 마지막 경우인 3을 default 경우로 하여 작성을 해도 무방하다.

```
always @(*) begin
    case (sel)
      0: y = d0;
      1: y = d1;
      2: y = d2;
      default: y = d3;  // case 3
    endcase
end
```

4.5 여러 가지 동작적 모델

Verilog 모델링 형태에는 다음과 같은 종류가 있다.

- 구조적 모델링: gate-level구조
- 동작적 모델링: 데이터흐름 모델 - 연속할당문
  레지스터 전송(Register Transfer Level: RTL) 모델
  알고리즘 기반 모델

이들 중에서 RTL모델과 알고리즘 기반 모델은 always문을 사용하여 기술한다.

레지스터 전송(RTL) 모델

레지스터 전송(RTL) 모델은 주로 클럭에 동기되어 동작하는 동기식 회로의 데이터 흐름을 모델링하는 데에 사용한다. 클럭 에지에 동기되어 변하는 하드웨어 레지스터 값을 지정하는 문장들로 구성되며 if - else 문이나 case 문을 함께 사용하여 기술할 수 있다. RTL 모델은 회로의 내부 구조를 미리 정한 상태에서 작성되며 구조가 정해져 있으므로 하드웨어로 합성하기가 용이하다. 동기식 회로의 일반적인 RTL 모델 작성 방법은 순차회로 설계에서 다룬다.

RTL 모델은 조합회로를 설계하는 데에도 사용할 수 있다. 연속할당문으로 기술된 조합회로의 RTL 모델링은 연속할당문을 다음과 같이 always문을 사용하여 변환하여

수행한다. 여기에서 연속할당문의 오른쪽 부분에 사용되는 모든 변수들은 RTL모델에서는 감지목록에 있어야 하고 왼쪽에 사용되는 변수들은 reg자료형으로 선언되어야 한다.

연속할당문 사용 예

```
wire y1, y2;

assign  y1 = expr1;
assign  y2 = expr2;
. . .
```

RTL 모델 예

```
reg y1, y2;

always @(v1 or v2 … ) begin
   y1 = expr1;
   y2 = expr2;
   . . .
end
```

그리고 예 4.14부터 예 4.16까지의 멀티플렉서에 대한 설계 예와 같이 if - else 문이나 case 문을 사용하면 연속할당문을 사용할 때보다 간결한 수식을 사용하여 조합회로를 나타낼 수 있다.

예 4.17은 예 3.11에서 연속할당문으로 기술한 2비트 비교기를 앞에서 소개한 방법을 적용하여 RTL 모델로 나타낸 것이다.

예 4.17 2비트 비교기 – RTL 모델

```
module cmp2_RTL(lt, gt, eq, A, B);
   input [1:0] A, B;
   output  lt, gt, eq;
   reg  lt, gt, eq;

   always @(A or B) begin   // verilog-2001은 @* 사용 가능
      lt = A < B;
      gt = A > B;
      eq = A == B;
   end
endmodule
```

예 4.17에서 always내의 문장은 순서대로 실행된다. always 내의 순차할당문에서 RHS와 LHS사이에 종속관계가 있으면 있으면 순서에 영향을 받는다. 이에 비해서 연속할당문들은 기술된 순서에 관계없이 병렬도 실행된다. 예 4.17에서는 각 순차할당문들 간에는 종속관계가 없으므로 할당문의 기술 순서를 바꾸어 써도 무방하다.

예 4.18은 2비트 비교기를 if - else 문을 사용한 RTL 모델로 나타낸 것이다. 예 4.17

에 있는 할당문의 수식에 포함되어 있는 연산자가 if - else 문의 조건식으로 사용되었고 할당문의 수식은 0 또는 1을 사용하였다.

예 4.18 2비트 비교기 – 또다른 RTL 모델

```
module cmp2_RTL2(lt, gt, eq, A, B);
   input [1:0] A, B;
   output  lt, gt, eq;
   reg  lt, gt, eq;

   always @(A or B) begin   // verilog-2001은 @* 사용 가능
        if (A < B) begin lt = 1; gt = 0; eq = 0; end
        else if (A > B) begin lt = 0; gt = 1; eq = 0; end
        else begin lt = 0; gt = 0; eq = 1; end
   end
endmodule
```

알고리즘 기반 모델

알고리즘 기반 모델은 RTL 모델보다 더 추상적인 모델로서 입력으로부터 출력을 결정하는 알고리즘을 사용하여 입력과 출력과의 관계를 기술하며 내부의 레지스터, 데이터경로, 산술장치 등의 내부 구조에 대해서는 명시적으로 기술하지 않는다. 알고리즘 기반 모델은 단지 입출력 간의 관계를 나타내는 것이며 알고리즘의 진행 순서와 하드웨어 구조와는 관련이 없다. 내부 구조는 논리 합성 도구에 의해서 정해지며 논리 합성 시에 효율적인 구조로 합성할 수 있도록 하는 방법을 개발하는 것은 아직도 도전할 만한 여지가 많다. 알고리즘 기반 모델로 기술된 모델이 하드웨어로 합성될 수 없는 경우도 있음에 유의하자.

예 4.19 2비트 비교기 – 알고리즘 기반 모델

```
module cmp2_ALG(lt, gt, eq, A, B);
   input  [1:0]  A, B;
   output  lt, gt, eq;
   reg  lt, gt, eq;

   always @ (A or B) begin
      lt = 0;  gt = 0;  eq = 0;        // deassert(0) 초기화
      if (A==B)  eq = 1;
      else if ( A > B)  gt = 1;
      else lt = 1;
   end
endmodule
```

예 4.19은 2비트 비교기를 알고리즘 기반 모델로 작성한 것이다. 이 예의 always문 내에서 lt, gt, eq의 세 출력을 0으로 초기화하였다가 입력의 크기 비교 결과에 따라서 한 출력을 1로 만들었다. 기술하는 방법이 보통의 프로그램과 매우 비슷하다. 위의 알고리즘 기반 모델에서 출력이 처음에 0으로 초기화되었다가 뒤에 1로 바뀌었다고 해서 하드웨어의 동작도 이와 같은 동작을 하는 것은 아니다. 하드웨어의 동작은 always문 내에서 수행한 마지막 결과의 동작으로 기술되는 것이다.

이처럼 알고리즘 기반 모델은 소프트웨어와 같이 출력 변수를 기본 초기값으로 초기화시킨 후에 필요에 따라서 값을 갱신하는 방식으로 입출력 관계를 기술한다. 이에 비해서 RTL 모델은 출력 변수 값이 각 경우에 대하여 한 번만 할당된다.

4.6 블록킹과 비블록킹 할당문

순차처리문에서 사용하는 할당문('=')은 순차적으로 실행되므로 순서에 영향을 받는다. 순차할당문은 앞에 있는 할당문의 수행이 완료되어야 다음 문장이 수행되므로 **블록킹(blocking) 할당문**이라고 한다.

순차처리문 내에서 병렬로 수행되는 동작을 기술하기 위해서 기호 '<='를 사용하는 **비블록킹(nonblocking) 할당문**을 사용한다. 비블록킹 할당문은 순차처리문 내에 있는 할당문들의 모든 오른쪽 수식(RHS)의 값을 순서대로 계산한 다음에 계산된 결과들을 왼쪽 변수(LHS)에 저장한다. 비블록킹 할당문을 사용하면 할당문 사이에 종속관계가 있을지라도 새로 계산된 값이 아닌 이전의 값이 오른쪽 수식의 계산에 사용되므로 할당문의 순서에 영향을 받지 않는다. 표 4.2에서 두 할당문의 동작을 비교하였다.

표 4.3 블록킹 할당문과 비블록킹 할당문

종류	구문	동작
블록킹 할당문	변수 = 수식	(보통의 programming 언어와 같음) - 블록 내의 할당문을 순서대로 수행 - 할당문의 순서에 영향을 받을 수 있음 - 순차수행 모델링
비블록킹 할당문	변수 <= 수식	(concurrent behavior를 modeling) - 블록 내의 할당문의 수식들을 먼저 계산한 후에 LHS변수값을 갱신함 - 할당문의 순서에 영향을 받지 않음 - 병렬수행 모델링

비블록킹 할당문에 대한 시뮬레이션은 다음과 같이 수행한다. always 문이 시간 t=T에서 수행될 때에 always 문 내의 비블록킹 할당문들은 오른쪽 수식을 먼저 계산을 하고 미소시간 지연 후인 t=T+△에 계산결과를 왼쪽 변수에 저장한다. 이렇게 함으로써 각 할당문에 병렬로 수행하는 동작을 시뮬레이션할 수 있다.

플립플롭은 클럭 에지 직전의 입력값에 의하여 출력이 결정된다. 여러 개의 플립플롭을 사용하는 회로에서 각 플립플롭의 출력이 다른 플립플롭의 입력으로 사용되는 경우에 갱신되기 전의 출력 값이 다른 플립플롭의 입력으로 사용된다. 블록킹 할당문을 사용하면 할당문의 RHS를 계산할 때에 갱신되기 전의 입력을 사용하므로 이러한 동작이 잘 표현된다.

4비트 쉬프트 레지스터

그림 4.6과 같은 4비트 쉬프트 레지스터는 클럭의 상승에지에서 레지스터 값이 1비트씩 오른쪽으로 동시에 이동한다. 플립플롭을 포함한 회로의 모델링은 구조적 모델 또는 데이터흐름 모델로 기술하기가 어려우며 always 문을 사용한 동작적 모델로 기술한다. 4비트 쉬프트 레지스터의 always문을 사용한 여러 가지 기술 방법에 대해서 비교해보자.

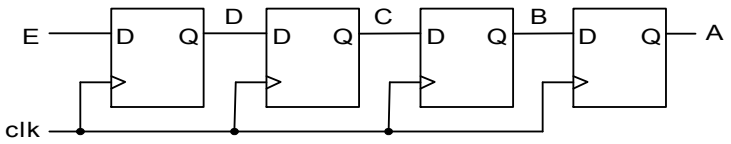


그림 4.6 4비트 쉬프트 레지스터

예 4.20은 4비트 쉬프트 레지스터를 4개의 always 문을 사용하여 나타내었다. 각 플립플롭이 동시에 동작하므로 별도의 always문을 사용하여 플립플롭을 기술하였다.

예 4.20 여러 개의 always문을 사용한 4비트 쉬프트 레지스터

```
module shift4 (clk, E, D, C, B, A);
    input clk, E;
    output D, C, B, A;
    reg D, C, B, A;

    always @(posedge clk)
        A = B;
```

```
    always @(posedge clk)
        B = C;
    always @(posedge clk)
        C = D;
    always @(posedge clk)
        D = E;
endmodule
```

4개의 always 문은 같은 감지 목록을 사용하여 같은 시점에서 반복하는 동작을 수행하므로 같은 always문에 포함시켜서 할당문을 기술할 수 있다. 예 4.21은 4비트 쉬프트 레지스터를 하나의 always문으로 기술한 것으로서 두 가지의 예를 들었다.

예 4.21 블록킹 할당문을 사용한 4비트 쉬프트 레지스터

```
always @(posedge clk) begin
    A = B;
    B = C;
    C = D;
    D = E;
end
```

```
always @(posedge clk) begin
    D = E;
    C = D;
    B = C;
    A = B;
end
```

예 4.21은 블록킹 할당문을 사용하였으며 두 예에서 사용된 할당문은 순서가 서로 다르다. 블록킹 할당문은 순서대로 실행되므로 할당문의 순서에 영향을 받을 수 있으며 두 모델의 결과는 다르게 나온다. 왼쪽 예는 쉬프트 레지스터의 동작을 정상적으로 기술한 것이지만 오른쪽 예는 순서대로 수행하면 A, B, C, D 모두 E의 값으로 할당되어서 잘못 기술된 것이다. 이러한 잘못은 비블록킹 할당문을 사용하면 없앨 수 있다.

예 4.22 비블록킹 할당문을 사용한 4비트 쉬프트 레지스터

```
always @(posedge clk) begin
    A <= B;
    B <= C;
    C <= D;
    D <= E;
end
```

```
always @(posedge clk) begin
    D <= E;
    C <= D;
    B <= C;
    A <= B;
end
```

예 4.22는 4비트 쉬프트 레지스터를 비블록킹 할당문을 사용하여 기술한 두가지 예이

다. 비블록 할당문을 사용하면 왼쪽 변수에 할당되기 전에 오른쪽 수식에 모두 계산이 되므로 위의 두 가지 예는 할당문의 순서에 관계없이 같은 결과를 제공하며 두 모델 모두 올바르게 동작한다.

always 문 내에 여러 개의 할당문이 있는 경우에 왼쪽과 오른쪽 변수 간에 종속관계가 없으면 어느 할당문을 사용해도 관계가 없지만, 종속관계가 존재하는 경우에는 순서에 영향이 있으므로 병렬 동작 모델과 에지트리거 동작을 하는 RTL 수준 동작 모델은 대개 비블록킹 할당문을 사용하고 알고리즘 기반 모델은 블록킹 할당문을 사용한다.

포트 이름 순서

Verilog에서 모듈의 포트이름의 순서에 대한 규칙은 없다. 다만 기능별로 포트이름을 나열하는 순서를 정하는 것이 일관성이 있으므로 바람직하다. 다음은 포트 이름 선언하는 순서의 한 가지 예이다.

- 입력 신호
- 출력 신호
- 입출력 신호(양방향)

그리고 같은 방향의 포트들 내에서는 다음과 같은 순서로 포트이름을 선언한다.

- 클럭신호(동기신호)
- 리셋(reset)
- 인에이블(enable)
- 제어신호
- 데이터, 주소