

Chapter 3

Dataflow Model 논리 설계



3.1 Modeling 방법

■ 설계 방법

- 전통적인 방법:
 - gate-level design → schematic entry
- 현재의 설계 방법:
 - high-level description using HDL
→ synthesize into gates
→ translate into physical technology (ASIC 또는 FPGA)

■ 모델링 방법

- 구조적 모델링(structural modeling)
- 동작적 모델링(behavioral modeling)
 - dataflow 모델링 – 부울함수식 이용
 - RTL (register transfer level) 모델링
 - algorithmic 모델링



동작적 모델링

■ 동작적 모델링(Behavioral modeling)

- 회로의 기능(functionality)을 기술
 - 회로가 어떻게(how) 구성되는 것이 아니라
 - 회로가 무엇을(what) 수행할 것인지를 기술
- 전파지연은 대개 동작적 모델링에 포함되지 않으며 timing constraint를 부과할 때에 synthesis tool에 의해서 고려됨

■ 동작적 모델링의 장점

- 빠른 설계와 검증 가능
- 합성도구가 최적화된 게이트 수준의 회로를 생성하고 물리적 구현 기술로 맵핑시켜줌

■ Modeling의 혼합

- structural model과 behavioral model을 혼합하여 설계할 수 있음
- 계층적 설계에서 하위 계층은 behavioral model을 사용하여 주로 설계하고, structural model은 상위 계층에서 주로 사용됨



3.2 동작적 모델링을 위한 자료형

■ 변수(variable)

- 회로의 binary-encoded signal 을 표현
- behavior의 기술에 사용되는 value을 표현

■ 자료형(data types)

- net 형
 - physical circuit의 wire를 modeling (두 design object를 연결)
 - 값을 저장하지 못함
(예) wire, wor, wand, tri, tri1, tri0 ...
- variable 형
 - procedural language의 variable과 같은 역할
 - 값을 저장 (프로그래밍 언어의 변수와 같은 역할)
(예) reg, integer, ...
 - physical register를 modeling하는 데 많이 사용하지만, 조합회로 출력의 modeling도 가능함



자료형의 크기와 정수의 표현

- 변수의 default size
 - wire, reg형은 1-bit,
 - integer형은 host의 정수 크기 (대개 32-bit)
- 정수의 표현과 크기 지정
 - 형식: [size]'[base] [value]
 - base: 10진수(d), 2진수(b), 16진수(h), 8진수(o)로 표현 가능
 - size: 비트 크기를 명시적으로 지정 가능
 - 비트 크기가 지정되지 않으면 integer형 크기(32-bit)로 지정됨
- 정수 표현 예

15	// 32-bit 10진수
'd15	// 32-bit 10진수
16'd255	// 16-bit 10진수
4'b1101	// 4-bit 2진수
'b1101	// 32-bit 2진수
12'h5fc	// 12-bit 16진수



정수의 표현

- 밑줄문자와 숫자 표현
 - 숫자의 가독성을 높이기 위해서 밑줄문자(_) 사용 가능

```
8'b1011_0010
```
- 4값논리 숫자 표현

12'h75x	// 0111_0101_xxxx
8'hx	// 8비트 모두 x
8'bz	// 8비트 모두 z
'bx	// 32비트 모두 x



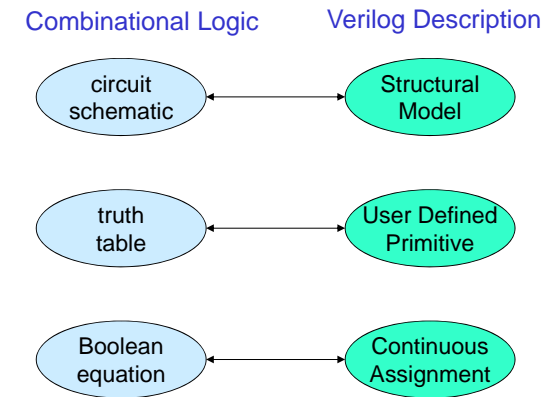
3.3 연속할당문과 부울함수식

- 연속할당문(Continuous assignment statement)
 - 부울함수식에 의해서 정해지는 값을 net에 할당함

```
assign y = ~a & b | a & ~c // y = a'b + ac'
```
- 연속할당문과 dataflow 모델링
 - 연속할당문은 조합회로를 부울함수식으로 모델링하는 데 사용함
 - dataflow 모델링
- 연속 할당문의 특징
 - 왼쪽은 net형(wire) 변수이어야 함 (variable형(reg) 변수는 안됨)
 - 연속할당문의 오른쪽 식의 피연산자들의 값이 바뀌자마자 식의 값을 다시 계산하여 왼쪽 net에 값을 할당함
 - 왼쪽 net이 다른 식의 피연산자로 사용될 때 왼쪽 net의 값이 바뀌면 위의 과정이 반복됨 (즉, 출력이 변경되면 이 출력을 입력으로 사용하는 할당문이 연속하여 수행됨)
 - 연속할당문에서도 delay값을 지정할 수 있음.

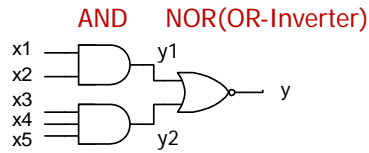


조합회로와 Verilog 기술



dataflow 모델링 예

- (예) 5-input AND-OR-Inverter(AOI)



```

module AOI5 (y, x1, x2, x3, x4, x5);
input  x1, x2, x3, x4, x5;
output y;
wire  y1, y2;

assign y1 = x1 & x2;
assign y2 = x3 & x4 & x5;
assign y = ~( y1 | y2 );
endmodule
    
```



- 연속할당은 정의된 순서에 관계없이 동시에(concurrent) 동작함

- 프로그램과 같이 순서대로 실행하는 것을 표현한 것이 아님
- 연속 할당은 하드웨어 회로를 입출력 관계식으로 나타낸 것임

- 다음은 앞의 예와 같은 회로임

```

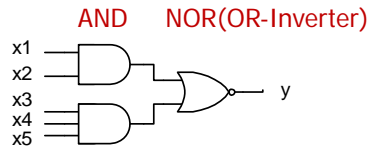
module AOI5 (y, x1, x2, x3, x4, x5);
input  x1, x2, x3, x4, x5;
output y;
wire  y1, y2;

assign y = ~( y1 | y2 );
assign y1 = x1 & x2;
assign y2 = x3 & x4 & x5;
endmodule
    
```



dataflow 모델링 예

- 하나의 부울함수식으로 나타낸 설계



```

module AOI5 (y, x1, x2, x3, x4, x5);
input  x1, x2, x3, x4, x5;
output y;

assign y = ~( (x1 & x2) | (x3 & x4 & x5) );
endmodule
    
```

- 중간의 wire 신호 선언 불필요
- 연산 우선순위 : &(and) > |(or)
 - **assign** y = ~(x1 & x2 | x3 & x4 & x5);



비트단위 논리 연산자

- 비트단위 논리연산자(bitwise logic operator)

연산자	연산	우선순위
~	NOT	1
&	AND	2
^	XOR	3
~^, ^~	XNOR	3 (C언어에 없음)
	OR	4

- 입력과 출력의 비트 수가 같아야 함.
- gate primitive에 대응되고 쉽게 합성 가능



여러 개의 할당문을 포함한 연속 할당문

■ 여러 개의 개별적인 연속할당문

(예) 1비트 비교기

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

■ 여러 개의 할당문을 포함한 연속할당문

- 하나의 assign 문에 여러 개의 할당문 포함 가능

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



Implicit 연속할당문

■ implicit 연속할당문

- assign을 사용하지 않고 wire 선언 시에 식을 변수에 지정함

■ (예)

- 보통의 연속할당문

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

- implicit 연속할당문

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



연속할당문과 전파지연

■ 전파지연(Propagation delay): #delay

- 모듈 인스턴스의 delay
 - nor #1 u1(y, x1, x2);
- 연속할당문의 delay
 - assign #2 y = ~(x1 | x2);
- wire의 delay
 - wire #10 y1 = x1 & x2;
 - 또는
 - { wire #10 y1;
 - assign y1 = x1 & x2;

■ Propagation delay의 simulation

- RHS의 값이 변하면 LHS 변수를 주어진 time delay후에 갱신하도록 scheduling함



3.4 조건 연산자와 멀티플렉서

■ 조건 연산자

- C언어와 같음
 - assign y = select ? data1 : data0

■ 2x1 멀티플렉서

- 선택신호의 값에 따라 두 입력 중 하나를 선택하여 출력하는 회로 (설계1) 비트단위 논리 연산자 사용

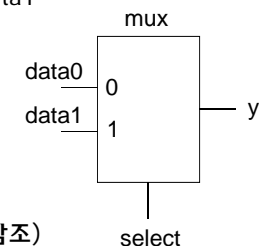
```
assign y = ~select & data0 | select & data1
```

- (설계 2) 조건 연산자 사용

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

■ 조건연산자와 2x1 멀티플렉서

- 조건연산자는 2x1 멀티플렉서를 사용하여 합성가능
- 멀티플렉서로 합성된 회로는 게이트 수준에서 최적화됨 (부울함수식 참조)

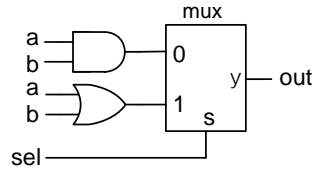


멀티플렉서 응용 회로

■ (예) 선택하여 출력하는 회로

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

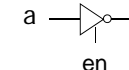
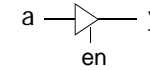
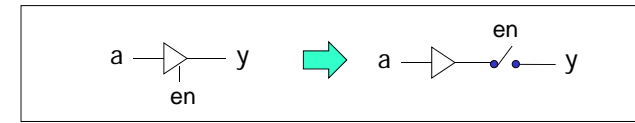
합성 예



3-상태 출력 회로

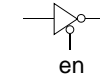
■ 3상태 버퍼/인버터

en=1일 때 연결됨



```
assign y = en ? a : 1'bz
```

```
assign y = en ? ~a : 1'bz
```



```
assign y = (~en) ? a : 1'bz
```

```
assign y = (~en) ? ~a : 1'bz
```



3-상태 출력을 가진 논리 회로

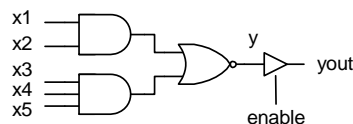
■ (예) 3상태 출력을 가진 5입력 AOI 회로

```
module AOI_5 (yout, x1, x2, x3, x4, x5, enable);
input  x1, x2, x3, x4, x5, enable;
output yout;
wire  y;

{ assign y = ~( (x1 & x2) | (x3 & x4 & x5) );
  assign yout = enable ? y : 1'bz; // 3-state buffer
}

endmodule
```

active
concurrently



conditional operator
? :



3-state output을 가진 논리 회로(계속)

```
module AOI_5 (yout, x1, x2, x3, x4, x5, enable);
input  x1, x2, x3, x4, x5, enable;
output yout; // output은 기본적으로 wire로 선언됨

assign yout = enable ? ~( (x1 & x2) | (x3 & x4 & x5) ) : 1'bz;
endmodule
```

```
module AOI_5 (yout, x1, x2, x3, x4, x5, enable);
input  x1, x2, x3, x4, x5, enable;
output yout;
wire yout = enable ? ~( (x1 & x2) | (x3 & x4 & x5) ) : 1'bz;
endmodule
```

wire 선언과 함께 정의된 equation은 assign이 없어도
continuous assignment문으로 취급됨.



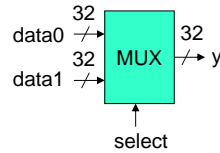
32비트 2x1 멀티플렉서

■ (예) 32비트 2x1 multiplexer

```

module mux2_32(y, data1, data0, select);
  output [31:0] y;
  input [31:0] data1, data0;
  input select;

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



- 입력 신호를 벡터형으로 선언하면 같은 조건식을 사용하여 1비트 MUX를 n비트 MUX로 쉽게 변경가능
- 부울함수 설계
 - **assign** y = ~select & data0 | select & data1; (틀린 설계)
 - **assign** y[0] = ~select & data0[0] | select & data1[0];
 - **assign** y[1] = ~select & data0[1] | select & data1[1];
 - ...
 - **assign** y[31] = ~select & data0[31] | select & data1[31];



3.5 Parameter를 사용한 constant 정의

■ parameter

- module 내에서 상수 값이 지정되는 기호상수를 정의함
 - **parameter** wsize = 32;
- module 인스턴스를 만들 때에 parameter 값을 재지정 가능
 - 모듈의 재사용이 용이함
 - portable, configurable, readable, extendable

■ (예) n-bit 2x1 multiplexer

```

module mux2(y, data1, data0, select);
  parameter wsize = 32;
  output [wsize-1:0] y;
  input [wsize-1:0] data1, data0;
  input select;

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



module instance에서의 parameter값을 재지정

■ module instance에서의 parameter값 재지정

- 3 가지 방법이 있음
 1. mux2 #(1) u1 (a, b, sel, out);
 2. mux2 #(.wsize(1)) u2 (a, b, sel, out);
 3. mux2 u3 (a, b, sel, out);
 - defparam u3.wsize = 1;

■ (cf) define 지시어

- **define** WSIZE 32
- 문장에서 **WSIZE** 를 사용하면 32로 대체됨
- module instance에서 변경 불가
- 대개 모듈 instance 생성 시에 변경할 필요가 없는 값의 상수 이름 지정에 사용함



연속할당문의 제한점

■ 연속할당문은

- 작은 부울함수식의 모델링 → 편리
- 큰 부울함수식의 모델링 → 적합하지 않음
 - 큰 회로를 equation으로 기술하는 과정에서 오류 가능성이 있음
- 큰 회로의 모델링은
 - 단순히 Boolean equation으로 기술하는 것이 아닌 완전한 behavioral modeling을 사용함.



3.6 여러 가지 연산자

■ 결합연산자

- 피연산자들을 결합(concatenation)하여 벡터를 만드는 데 사용
 - `{1'b1, 2'b01}` // 3'b101
 - `{ a, b, c }` //

a	b	c
---	---	---
 - 피연산자의 크기가 정해져 있어야 함
 - `{'b1, 'b01}` // 피연산자의 크기가 지정되지 않아서 **잘못임**
 - 반복결합
 - `{ 3 { A } }` // { A, A, A }
 - `{ A, 2{B} }` // { A, B, B }
 - `{ 2 { 3'b011} }` // 6'b011011
- A →

B →
- (예) 4 비트 신호 data를 32비트로 확장
 - `assign y = {28'b0, data};` 또는
 - `assign y = 32'b0 | data;`
 - (예) 32비트 2x1 멀티플렉서의 부울함수 설계?

등가연산자

■ logic equality operator: `==, !=`

- 결과는 0, 1 또는 x
- x 또는 z값과 비교할 때에는 결과가 x

■ case equality operator: `===, !==`

(C언어에 없음)

- 결과는 0 또는 1 (x는 없음)
- x 또는 z값도 정확하게 비교함

■ 예

```
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)
```



논리연산자

■ 논리연산자

`&&` 논리 AND
`||` 논리 OR
`!` 논리 NOT

- 결과는 1비트 - 참(1), 거짓(0)
- 피연산자가 0이 아닌 경우에 참으로 인식
- 피연산자가 x 또는 z값을 가질 경우에 거짓으로 인식

- 게이트를 사용하여 쉽게 합성됨

(cf) 비트단위 논리연산자:

- 결과의 크기 = 피연산자의 크기

예제

■ 부울함수식의 다른 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}; // 2비트 벡터신호
```

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

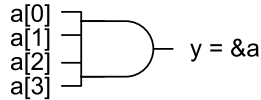


축소연산자

축소연산자(reduction operator) (C언어에 없음)

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

- 벡터 피연산자의 모든 비트들에 대해서 지정된 논리연산 수행
- 결과는 1비트



- (예) 3입력 AND
 - assign y = a & b & c; → assign y = &{a,b,c};
- (예) 3입력 NAND
 - assign y = ~(a & b & c); → assign y = ~&{a,b,c};



산술연산자와 관계연산자

산술연산자

- 덧셈(+), 뺄셈(-), 곱셈(*), 나눗셈(/), 나머지(%)
- 피연산자가 x 또는 z값을 가지면 결과는 x임

단항 +, - 연산자

관계연산자

- 크기 비교: >, >=, <, <=
- 피연산자가 x 또는 z값을 가지면 결과는 x임

쉬프트 연산자

- right-shift(>>), left-shift(<<)

산술연산자, 관계연산자, 쉬프트연산자의 합성

- 간단한 게이트가 아닌 각 연산을 수행하는 회로로 합성됨



3.7 예제: 2-bit 비교기

bitwise operator 사용

A=0x, B=1x
 A=00, B=x1
 A=x0, B=11

```
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
```

결합연산자, 관계 연산자 사용

- 두 signal을 합쳐서 2-bit 단위로 비교함

```
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
```



■ module port를 vector로 선언

```

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
    
```

■ n-bit로 일반화하기 쉬움

■ parameter를 사용하여 SIZE 지정

```

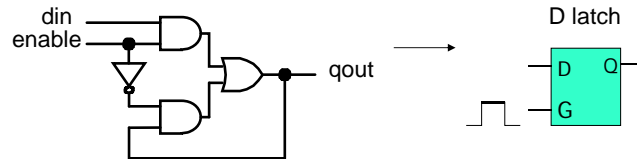
module cmp(lt, gt, eq, A, B);
  parameter SIZE = 32;
  input [SIZE-1:0] A, B;
  output lt, gt, eq;

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

- assign을 각 continuous assignment에 개별적으로 사용하지 않고 continuous assignment의 list에 한 번만 사용할 수 있다.
- parameter 상수는 module instance에서 재지정 가능

3.8 Feedback이 있는연속할당문

■ D 래치



- 동작: enable=1일 때 : qout = din
enable=0일 때 : qout은 그대로 유지(기억)

■ feedback이 있는 연속할당문

```

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

- synthesis tool이 feedback이 있는 연속할당문을 미리 준비된 latch로 합성할 수 있음

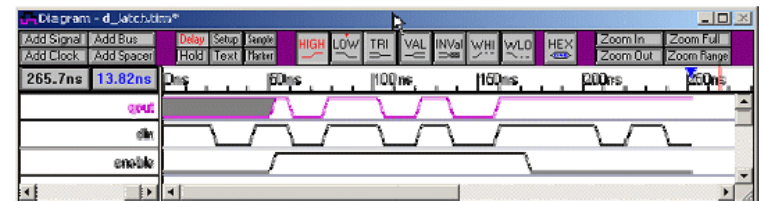
Transparent D-latch

■ (예) 조건연산자와 피드백이 있는 연속할당문을 사용한 transparent D-latch

```

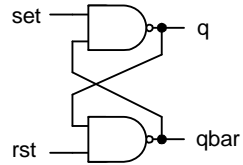
module d_latch(qout, din,enable);
  output qout;
  input din, enable;

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



SR Latches

SR Latch



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

(주의) Synthesis tool이 이러한 간접적인 형태의 feedback은 잘 처리하지 못할 수 있음 (미리 준비된 래치로 합성 못할 수 있음)
 → 이런 형태의 설계는 권장하지 않음



Reset 제어 입력이 있는 transparent latch

reset 제어입력

- 출력을 0으로 초기화하는 데 사용 (reset=0일때)
- conditional operator를 중첩 사용하여 기술가능

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

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

