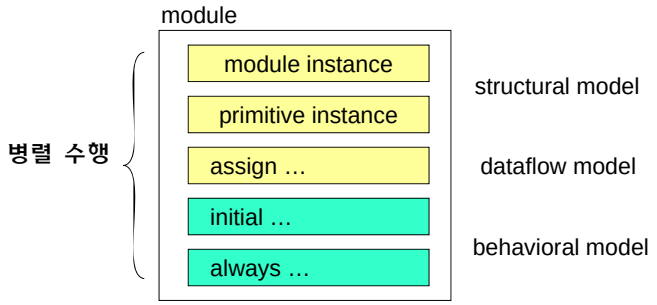
Chapter 4

Behavioral Model 논리 설계

4.1 initial과 always

- Verilog의 특징
  - 보통의 programming언어와 같은 procedural statement을 제공
    → 추상적인 behavioral model 기술에 사용
  - 순차적으로 수행하는 보통의 programming 언어와는 다르게 병렬적으로 수행하는 언어임



procedural statement

- procedural statement
  - 보통의 programming언어와 비슷한 방식으로 기술함
  - 보통 programming언어에서 사용하는 구문들 사용
    if, case, for, assignment 등 ...
  - 추상적인 behavioral model에 사용
- procedural statement의 종류
  - initial 문 – single-pass behavior
  - always 문 – cyclic behavior

initial 문

- initial 문 – single pass behavior

```
initial begin
  문장;
  문장;  ...
end
```

```
initial  문장;
```

  - simulation을 시작할 때에 한 번만 수행
  - synthesis되지 않음
  - 일반적으로 초기화, 입력파형 생성, 출력 관찰 등 simulation 수행시에 한 번만 수행되어야 하는 과정을 기술하는 데 사용
- 예

```
initial begin
  x = 1'b0;
  #40 x = 1'b1;
end
initial #100 $finish;
```

always문

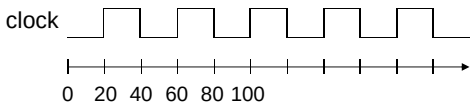
- always 문 – cyclic behavior

```
always 반복조건 begin
    문장;
    문장; …
end
```

  - 반복조건의 이벤트가 발생할 때마다 반복하여 수행

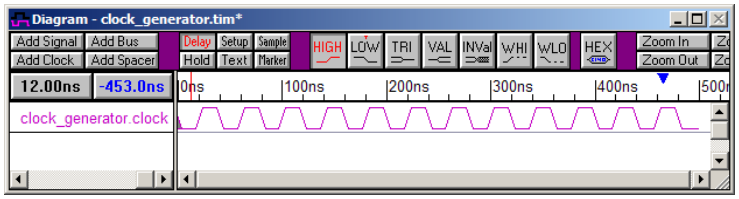
- 예: clock generator

```
initial clock = 1'b0;
always
   #20 clock = ~clock;
```




```
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
```



initial과 always의 비교

종류	구문	동작	논리합성
initial	initial begin . . . end	(single-pass behavior) simulation시작할 때 한 번 수행	x
always	always begin . . . end	(cyclic behavior) 이벤트가 발생할 때마다 반복 수행	O

4.2 타이밍 제어 - always 반복 조건

- 타이밍 제어

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

  - 지연제어와 레벨제어문은 논리합성이 지원되지 않음

- 지연 제어문

  # delay

  - 주어진 시간 delay가 지연된 후에 뒤에 있는 문장을 수행

이벤트 제어

- 이벤트 제어
  - @(*var*) ... var값의 변화시에 event 발생
- 이벤트 제어의 여러 가지 형태
  - @(*var*) 변수 var 값의 변화 → level-sensitive variable
  - @(**posedge** *var*) 변수 var의 positive edge (0→1 변화)
  - @(**negedge** *var*) 변수 var의 negative edge (1→0 변화)
  - @(*var1* **or** *var2*) 변수 var1 또는 var2 값의 변화
  - @(**posedge** *var1* **or negedge** *var2*)
  - @(**posedge** *var1* **or** *var2*) → synthesis tool은 혼합형태를 지원하지 않음

edge-triggered variable

sensitivity list

Verilog-2001에서는 or 대신에 , 를 사용할 수 있음

- @(var1, var2, var3)

level-sensitive와 edge-triggered 이벤트

- level-sensitive 이벤트

half-adder — a 또는 b가 변할 때 이벤트 발생

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

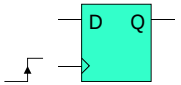
a 또는 b가 변할 때마다 sum과 cout값을 다시 계산

- edge-triggered 이벤트

clk의 positive edge에서 이벤트 발생

D flip-flop

```
always @(posedge clk)
   qout = din;
```



clk의 positive edge에서 din이 qout에 저장됨

4.3 procedural assignment문

- 예: 보수 출력이 있는 positive edge-triggerred D flip-flop

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

   always @ (posedge clk) begin
      q = data;
      qbar = ~q;
   end
endmodule
```

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

   assign  qbar = ~q;
   always @ (posedge clk) begin
      q = data;
   end
endmodule
```

두 할당문은 순서대로 수행함
두 할당문 순서가 바뀌면 동작이 달라짐

assign문과 always문의 병렬 수행

- 순차할당문(procedural assignment statement)
  - initial 또는 always 내에서 사용된 assignment(=) 문장
  - 순차적으로 할당문이 수행됨 → 할당문의 순서가 동작에 영향을 미칠 수 있음
  - LHS변수는 variable 형이어야 함 (ex) reg 또는 integer
- reg형 변수는
  - hardware register로 합성될 수 있으나 그렇지 않을 수도 있음
- always 반복조건과 회로
  - sensitivity list : @ 이벤트에 포함된 신호들
  - sensitivity list에 모든 입력을 포함한다면
    - 조합회로로 합성될 수도 있음
  - sensitivity list,에 모든 입력이 포함되지 않는다면
    - 내부에 기억장소가 포함된 순차회로로 합성됨

4.4 조건문

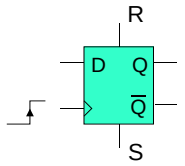
- if 문
  - if (조건식) 문장
  - if (조건식) 문장1
    else 문장2
  - if (조건식1) 문장1
    else if (조건식2) 문장2
    ...
    else if (조건식n) 문장n
    else 문장n+1
- case 문
  - case (수식)
    값1: 문장1;
    값2: 문장2;
    ...
    default: 기본문장; // 일치하는 값이 없으면 기본문장 수행
    endcase

if 문을 사용한 설계

- 예: flip-flop with synchronous set and reset

```
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
```



예제

- 예: flip-flop with asynchronous set and reset

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

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

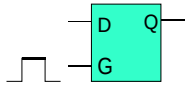
- set과 reset은 실제로 level-sensitive 동작:
  - 0일 때에 clk과 무관하게(asynchronous) set 또는 reset 동작
- clk만 edge-triggered 동작

예제

- 예: transparent D-latch

```
module tr_latch (q, enable, data);
   output   q;
   input    enable, data;
   reg      q;

   always @ (enable or data) begin
      if (enable) q = data;
   end
endmodule
```



- enable=0이면 q값은 변하지 않음 → 저장
- sensitivity list에 모든 입력이 있지만 문장에서 출력 q가 모든 경우에 대해서 정의되어 있지 않으므로 기억장소를 포함하여 논리 합성됨

예제

- 예: 1비트 4x1 멀티플렉서

```
module mux4(y, d3, d2, d1, d0, sel);
   output  y;
   input   d3, d2, d1, d0;
   input   sel;
   reg  y;

   always  @(d3 or d2 or d1 or d0 or sel) begin
      if (sel==0)  y = d0;
      else if (sel==1)  y = d1;
      else if (sel==2)  y = d2;
      else if (sel==3)  y = d3;
      else  y = 1'bx;
   end
endmodule
```

else y = d3;

case 문을 사용한 설계

- 1비트 4x1 멀티플렉서

```
module mux4(y, d3, d2, d1, d0, sel);
   output y;
   input   d3, d2, d1, d0;
   input [1:0]  sel;
   reg  y;

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


4.5 여러 가지 Behavioral Model

- Modeling styles
  - structural modeling
    - gate-level structure
  - behavioral modeling
    - continuous assignment – dataflow model
    - register transfer level(RTL) logic
    - algorithm-based model

} always문 사용

RTL Model

- RTL(Register Transfer Logic) model
  - 주로 clock에 동기 되는 동작하는 synchronous machine의 data flow를 modeling하는 데 사용
  - combinational logic의 modeling 가능
- 조합회로의 RTL modeling
  - combinational logic은 continuous assignment문과 같은 동작을 하는 asynchronous cyclic behavior로 modeling 가능

Verilog-2001에서는 @*로 대체 가능

RHS에 사용되는 모든 변수 포함

```
wire y1, y2;

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

continuous assignment 모델

```
reg y1, y2;

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

RTL 모델

RTL model – 예제

- 예: 2-bit comparator

```
module cmp2_RTL(lt, gt, eq, A1, A0, B1, B0);
   input A1, A0, B1, B0;
   output  lt, gt, eq;
   reg  lt, gt, eq;

   always @(A0 or A1 or B0 or B1) begin
      lt = {A1,A0} < {B1,B0};
      gt = {A1,A0} > {B1,B0};
      eq = {A1,A0} == {B1,B0};
   end
endmodule
```

  - always내의 문장은 순서대로 실행됨
    → RHS와 LHS 사이에 dependency가 있으면 순서에 영향을 받음

(cf) multiple continuous assignment들은 concurrent하게 실행됨
→ 기술된 순서에 관계 없음

Algorithm-Based Models

- Algorithm-based model
  - RTL model보다 더 추상적인 model
    - input-output 관계를 기술하며
    - register, datapath, computational resource 등의 내부 구조를 기술하지 않음
  - architectural synthesis
    - 내부 구조는 synthesis tool에 의해서 정해짐 → most challenging
  - hardware로 합성될 수 없는 경우도 있음

Algorithm-Based Model – 예제

- 예: 2-bit comparator

```
module cmp2_CA2(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;      } 조건을 만족하는
      else  lt = 1;                  } 출력을 assert(1)
   end
endmodule
```

  - 이 model은 reg 변수를 사용할 지라도 조합회로로 합성되며 hardware register를 필요로 하지 않음

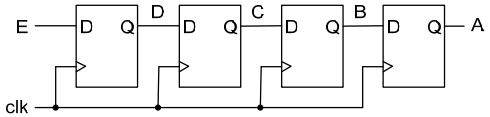
4.6 Blocking와 Nonblocking Assignment 문

- procedural assignment 문의 종류

종류	구문	동작
blocking assignment 문	변수 = 수식	(보통의 programming 언어와 같음) ▪ 블록 내의 할당문을 순서대로 수행 ▪ 할당문의 순서에 영향을 받을 수 있음 → sequential
nonblocking assignment 문	변수 <= 수식	(concurrent behavior를 modeling) ▪ 블록 내의 할당문의 수식들을 먼저 계산한 후에 LHS변수값을 갱신함 ▪ 할당문의 순서에 영향을 받지 않음 → concurrent

예 – 4-bit shift register

- 4비트 shift register
  - clock의 positive edge에서 register 값이 1비트씩 오른쪽으로 이동



- 4-bit shift register의 modeling
  - structural model 또는 dataflow model로 기술하기가 어려움
  - always @(posedge clk) 을 사용한 behavioral model로 기술함
  - assignment문 사용에 주의가 필요함

Blocking Assignment 문

- Blocking assignment: a = b
  - sequential, procedural assignment
  - blocking assignment문의 실행이 종료되어야 다음 문장이 실행됨
    → 새로 할당된 결과가 다음 문장의 실행에 사용됨
  - 실행 결과는 blocking assignment 문의 순서에 영향을 받을 수 있음
- 예: 4-bit shift register (순서: E→D→C→B→A)

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

before: EDCBA=1011x
after: EDCBA=11011
(옳음)

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

before: EDCBA=1011x
after: EDCBA=11111
(잘못됨)

Nonblocking Assignment

- Nonblocking assignment: a <= b
  - 먼저 procedure 내의 RHS의 값들을 계산한 후에 LHS 변수를 갱신함
    - t = T에 RHS 값 계산
    - t = T + Δ에 LHS 변수 값 갱신
  - nonblocking assignment 문들을 concurrent하게 실행하는 효과를 제공하며 문장들의 순서에 영향을 받지 않음
- 예: 4-bit shift register (순서: E→D→C→B→A)

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

before: EDCBA=1011x
after: EDCBA=11011
(옳음)

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

before: EDCBA=1011x
after: EDCBA=11011
(옳음)

Blocking vs. Nonblocking Assignments

- Multiple assignments에서 LHS와 RHS 간에 dependency가
  - 없는 경우: blocking, nonblocking 모두 사용 가능
  - 존재하는 경우: 순서에 영향이 있음
    - concurrent 동작 모델: nonblocking 사용
    - algorithmic 동작 모델: blocking 사용
- 권장하는 용도
  - Nonblocking assignment: edge-sensitive operation (synchronous 동작, register 출력저장)
  - Blocking assignment: 조합회로 출력

Port Names: Style

- Module의 port name의 순서
  - 정해진 규칙은 없음
  - 권장 순서 – 다음 순서(출력 우선) 또는 역순 (입력 우선)
    - bidirectional datapath signals
    - bidirectional control signals
    - datapath outputs
    - control outputs
    - datapath inputs
    - control inputs
    - synchronizing signals