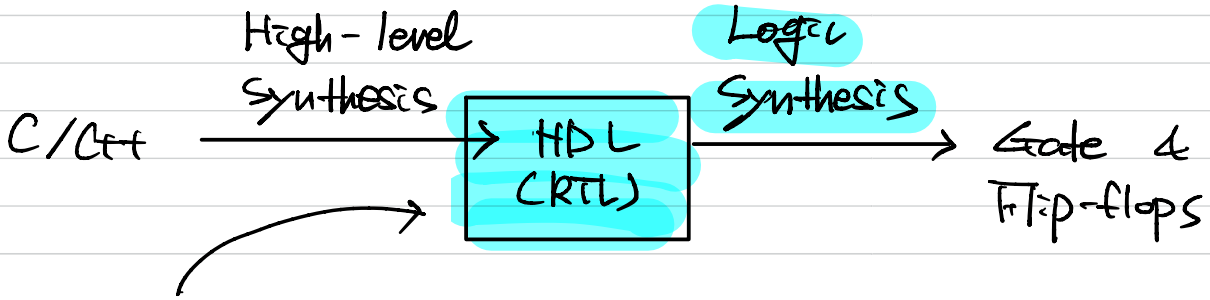


should be functionally correct while allowing a synthesis tool to design a final gate level structures.

Description Method

Gate Primitives

Continuous assign Stmt

Procedural Stmt

Types of Logic

Combinational Logic

Sequential Logic

Combinational Logic

Any change of any input value may have an immediate effect on the resulting output.

Sequential Logic

output depends not only on the circuit's present inputs, but also on the circuit's present state, which is stored in the circuit.

1. Combinatronal Logic

1.1 Using Gate Primitives

A logic synthesis tool ignores timing specifications Rather, it provides means of specifying timing requirements such as clock period. The tool will then try to design the logic so that all set-up times are met within that clock period.

1.2 Using Assign Stmt

```
assign  y = (a === 1'bx) ? c : 1;
```

unknow (Don't care)

is not synthesizable because there are no unknown in real hardware

```
assign  y = (a == b) ? 1'bx : c;
```

↓ optimized

```
assign  y = c;
```

1.3 Combinational Logic Using Procedural Stmt.

1.3.1 Rule

Def) Let **input set** of the always block be the set of all registers, wires, and inputs used on the right-hand side of the procedural statements.

Def) Let **sensitivity list** of the always block be the list of names appearing in the event statement ("@").

Rule 1) Every element of input set must appear in the sensitivity list without any edge specifiers (e.g., posedge).

Rule 2) the output of the combinational logic must be assigned in each and every one of the possible control path.

1.3.2 Inffered Latch

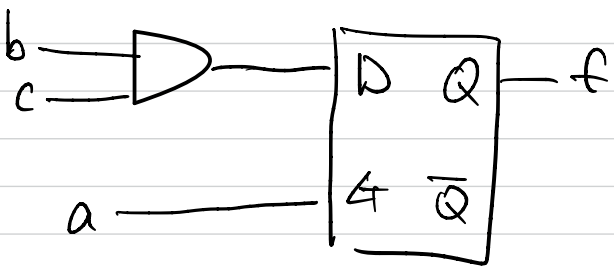
Example of Good description.

```
module mux ( output reg f,
             input      a, b, c);

  always @(*) begin
    f = c; ← guarantees Rule 2)
    if (a == 1)  f = b;
  end
end module
```

Example of Bad description.

```
module synInferedLatch (output reg f,
                        input      a, b, c);

  always @(*) begin
    if (a == 1)  f = b & c;
  end
end module
```

f will keep previous value if a is not 1.
→ Latch.



1.3.3 Specifying Don't Care Situation

Logic synthesis tools make great use of don't care situation to optimize a logic circuit.

```
module synCaseWithDC ( output reg f,
                       input      a, b, c);

  always @(*)
    case ( {a, b, c} )
      3'b001 :  f = 1'b1;
        :
      default :  f = 1'bx;
    endcase
endmodule
```

Be Careful !!

Don't cares give rise to differences bet. simulations and synthesis.

① (0, 1, X, Z) are possible value in simulation
(0, 1) are in synthesis.

② comparing to X make sense in simulation
doesn't in synthesis

DO NOT COMPARE WITH X or Z.

1.3.4 Procedural Loop Construct

	Sythesizable	used for
for	O	Combinational.
while	O	Sequential
forever	O	Sequential
repeat	X	X

```
module syn_Xor_8 ( output reg [1:8] xout,
                   input     [1:8] xin1, xin2);
  reg [1:8] i;
  always @(*)
    for (i = 1; i <= 8; i = i+1)
      xout[i] = xin1[i] ^ xin2[i];
end module.
```

```
always @(*) begin
  for (i = 0; i < dataWidth; i = i+1)
    matchCount = matchCount + ~(messageI[i] ^ patternI[i]);
end
```

Looks sequential logic

but it's combinational!!

1.4 Sequential Logic using Procedural statement.

If there is a sequential element like latch, flipflops in procedural statement, the synthesis tools infer it as sequential logic.

Latch vs. Flip Flops

Latch

Level sensitive meaning that their behavior is controlled by a gate input.

Flip Flop

Edge-triggered meaning that their behavior is controlled by a positive or negative edge of clock.

1.4.1 Latch Inferences

Rule 1) At least one control path must exist that does not assign to an output.

Rule 2) The sensitivity list must not contain any edge-sensitive specification.

```
always @(*)
    if (~reset)
        Q = 0;
    else if (g)
        Q = d;
```

Does it matter?

< Latch with Reset >

1.4.2 Flip Flop Inference

NINE RULES

1. always statement must specify the edges of each signal.

2. The first statement following the always must be an if.

3. The test for the set and reset conditions are done first using else-if constructs. The expressions for set and reset cannot be indexed; they must be one-bit variables.

4. If a negative edge was specified, then the test should be:

   if (~reset) ... or if (reset == 1'b0)

5. If a positive edge was specified, then the test should be:

   if (set) ... or if (set == 1'b1) ...

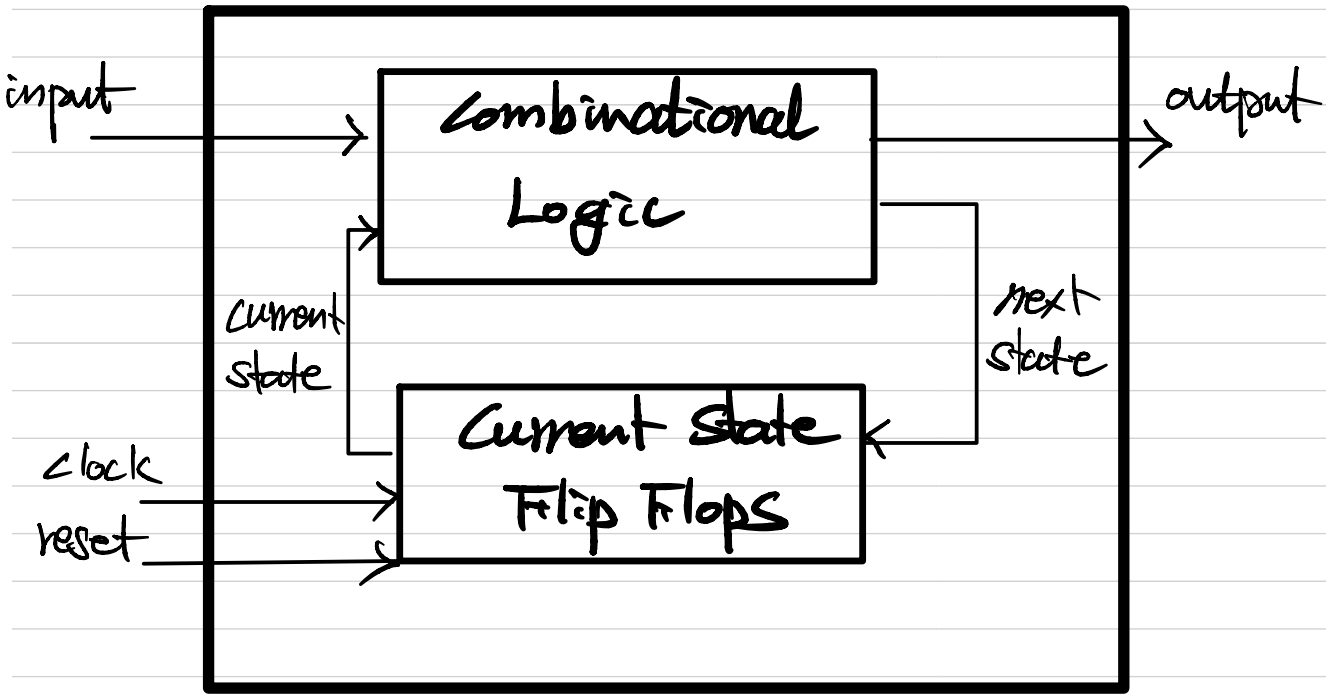
5. After all of the set and resets are specified, the final statement specifies the action that occurs on the clock edge.

6. All procedural assignment must be either be blocking or non-blocking assignments.

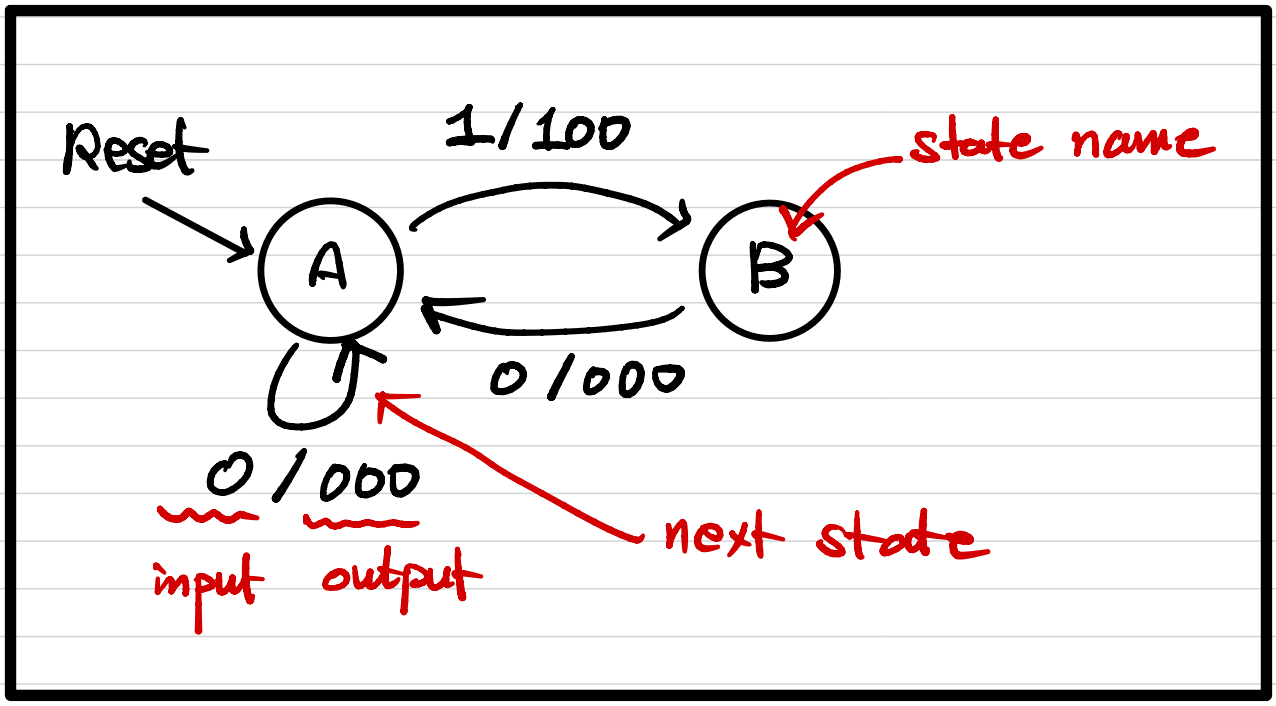
7. The sensitivity list includes only the edges for clock, reset, and preset conditions.

8. Any register assigned will be implemented using flip flops.

1.5 Describing Finite State Machine



Standard Model of FSM.



State Transition Diagram.

1.5.1 Explicit Style using case construct.

```
reg [2:0] currentState, nextState;

localparam [2:0]  A = 3'b000,
                  B =
                  :
                  H = 3'b111;

always @(*)
   case (currentState)
      A : begin
            nextState =  ... ;
            out       =  ... ;
      end

      default : begin
            nextState =  ... ;
            out       =  3'bxxx;
      end
   endcase

always @(posedge clock or negedge reset)
   if (~reset) currentState <= A;
   else        currentState <= nextState;
```

wire

flip flop

1.6.2 Implicit Style

```
always begin
  @(posedge clock)
    temp    = dataIn + 1;
  @(posedge clock)
    temp    = temp & 2;
  @(posedge clock)
    dataOut = temp - 1;
end
```

Let's not use ...

1.6 FSM and Datapath.

A system consists of datapath and FSM where datapath can do computations and store results in registers, and FSM will control the datapath.

1.6.1

A target computation

```
for (x=0, i=0; i<=10; i=i+1)
    x = x + y;

if (x<0) y=0;
else     x=0;
```

1.6.2

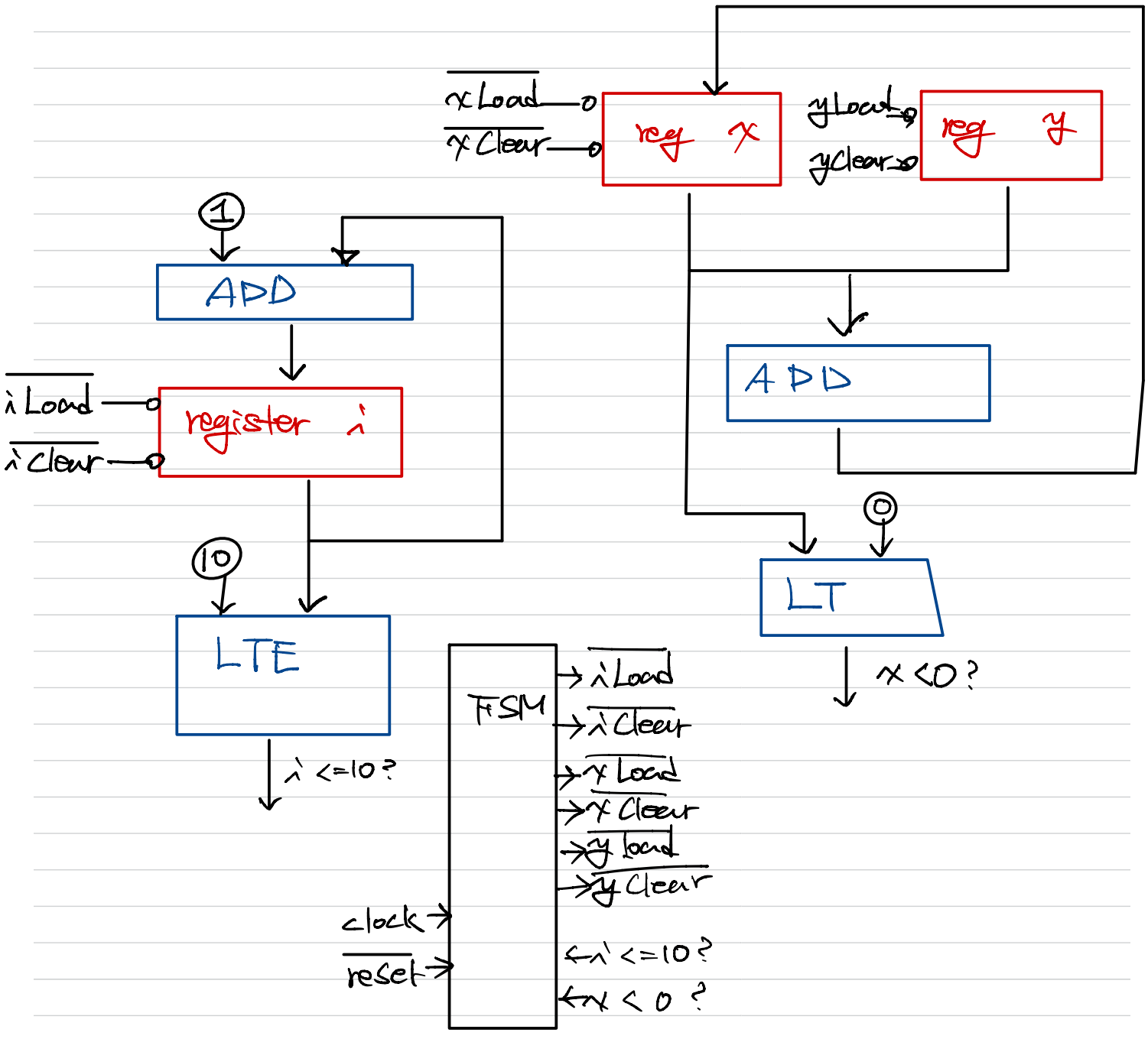
A data path and FSM

reg reg LTE ADD

```
for ( x = 0, i = 0 ; i <= 10 ; i = i + 1 )
    x = x + y ;
    if (x < 0)      y = 0 ;
    else            x = 0 ;
```

reg

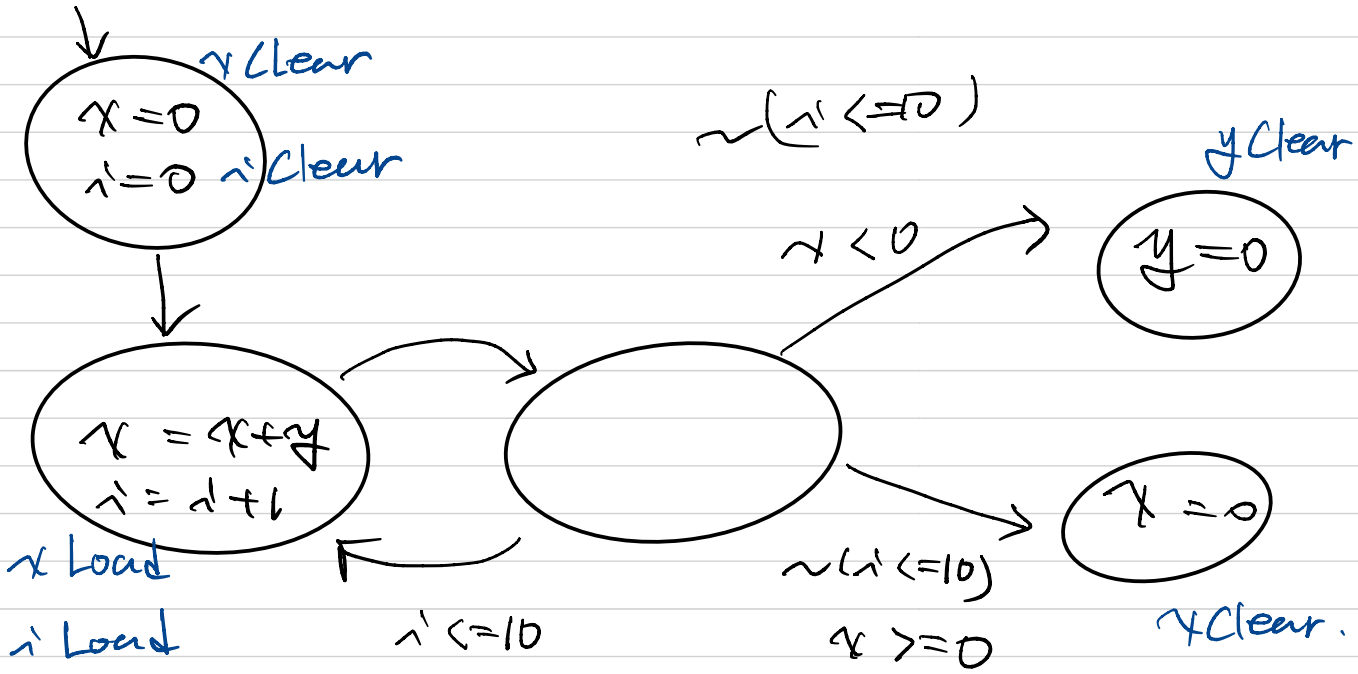
LT



1.6.3

Specifying the FSM

```
for (x=0, i=0; i<=10; i=i+1)
    x = x + y;

if (x<0) y=0;
else     x=0;
```



1.7 Summary

Type of Logic	Output Assigned To	Edge specifiers
Combi.	An output must be assigned +	