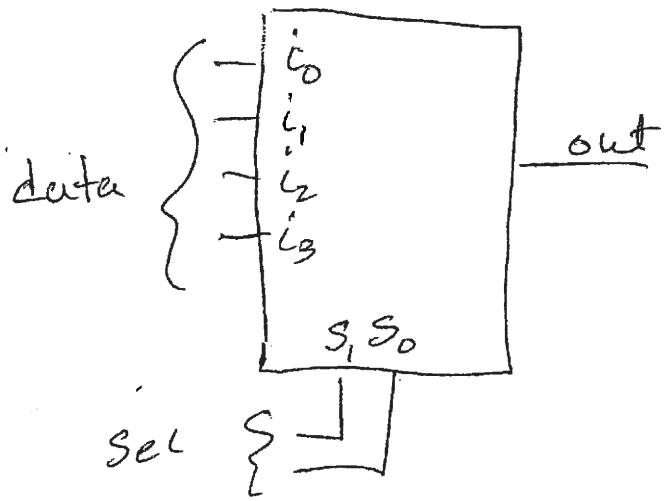
Example 6-2 *4-to-1 Multiplexer, Using Logic Equations*

```
// Module 4-to-1 multiplexer using data flow. logic equation
// Compare to gate-level model
module mux4 to 1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3; // data inputs
input s1, s0;         // selection

//Logic equation for out
assign out =    (~s1 & ~s0 & i0) |
                (~s1 & s0 & i1) |
                (s1 & ~s0 & i2) |
                (s1 & s0 & i3) ;

endmodule
```



$S_1 S_0$	out
0 0	i_0
0 1	i_1
1 0	i_2
1 1	i_3

Example 6-3 *4-to-1 Multiplexer, Using Conditional Operators*

```
// Module 4-to-1 multiplexer using data flow. Conditional operator.
// Compare to gate-level model
module multiplexer4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;

// Use nested conditional operator
assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0) ;

endmodule
```

Example 6-4 *4-bit Full Adder, Using Dataflow Operators*

```
// Define a 4-bit full adder by using dataflow statements.
module fulladd4(sum, c_out, a, b, c_in);

// I/O port declarations
output [3:0] sum;
output c_out;
input[3:0] a, b;
input c_in;

// Specify the function of a full adder
assign {c_out, sum} = a + b + c_in;

endmodule
```

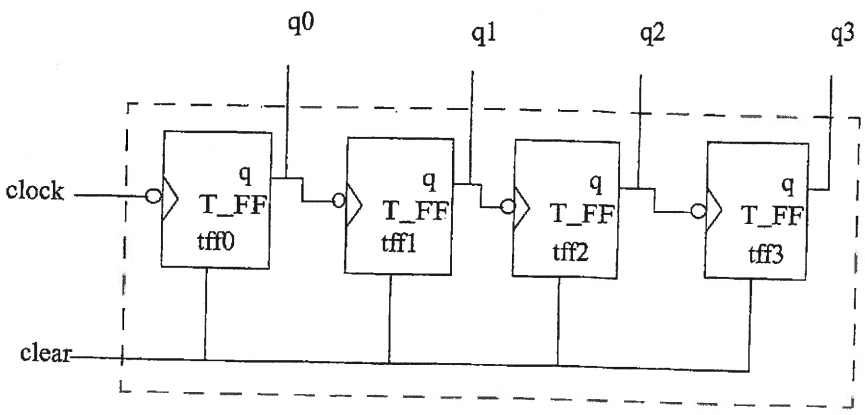


Figure 6-2 4-bit Ripple Carry Counter

Example 6-6 Verilog Code for Ripple Counter

```
// Ripple counter
module counter(Q , clock, clear);

// I/O ports
output [3:0] Q;
input clock, clear;

// Instantiate the T flipflops
T_FF tff0(Q[0], clock, clear);
T_FF tff1(Q[1], Q[0], clear);
T_FF tff2(Q[2], Q[1], clear);
T_FF tff3(Q[3], Q[2], clear);

endmodule
```

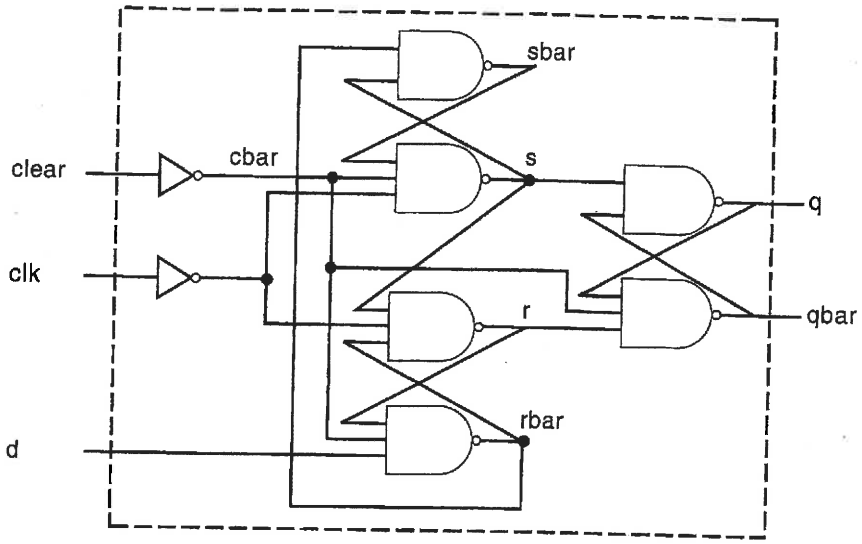


Figure 6-4 Negative Edge-Triggered D-flipflop with Clear

Example 6-8 Verilog Code for Edge-Triggered D-flipflop

```
// Edge-triggered D flipflop
module edge_dff(q, qbar, d, clk, clear);

// Inputs and outputs
output q,qbar;
input d, clk, clear;

// Internal variables
wire s, sbar, r, rbar,cbar;

// dataflow statements
//Create a complement of signal clear
assign cbar = ~clear;

// Input latches; A latch is level sensitive. An edge-sensitive
// flip-flop is implemented by using 3 SR latches.
assign  sbar = ~(rbar & s),
        s = ~(sbar & cbar & ~clk),
        r = ~(rbar & ~clk & s),
        rbar = ~(r & cbar & d);

// Output latch
assign  q = ~(s & qbar),
        qbar = ~(q & r & cbar);

endmodule
```

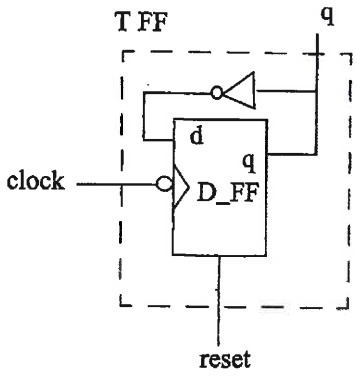


Figure 6-3 T-flipflop

Example 6-7 Verilog Code for T-flipflop

```
// Edge-triggered T-flipflop. Toggles every clock
// cycle.
module T_FF(q, clk, clear);

// I/O ports
output q;
input clk, clear;

// Instantiate the edge-triggered DFF
// Complement of output q is fed back.
// Notice qbar not needed. Unconnected port.
edge_dff ff1(q, ,~q, clk, clear);

endmodule
```

Behavioral level of abstraction

Highest level

Defn (testbench)

a program designed to exercise a digital system to verify its functionality

Notes:

1) written in Verilog at the behavioral level

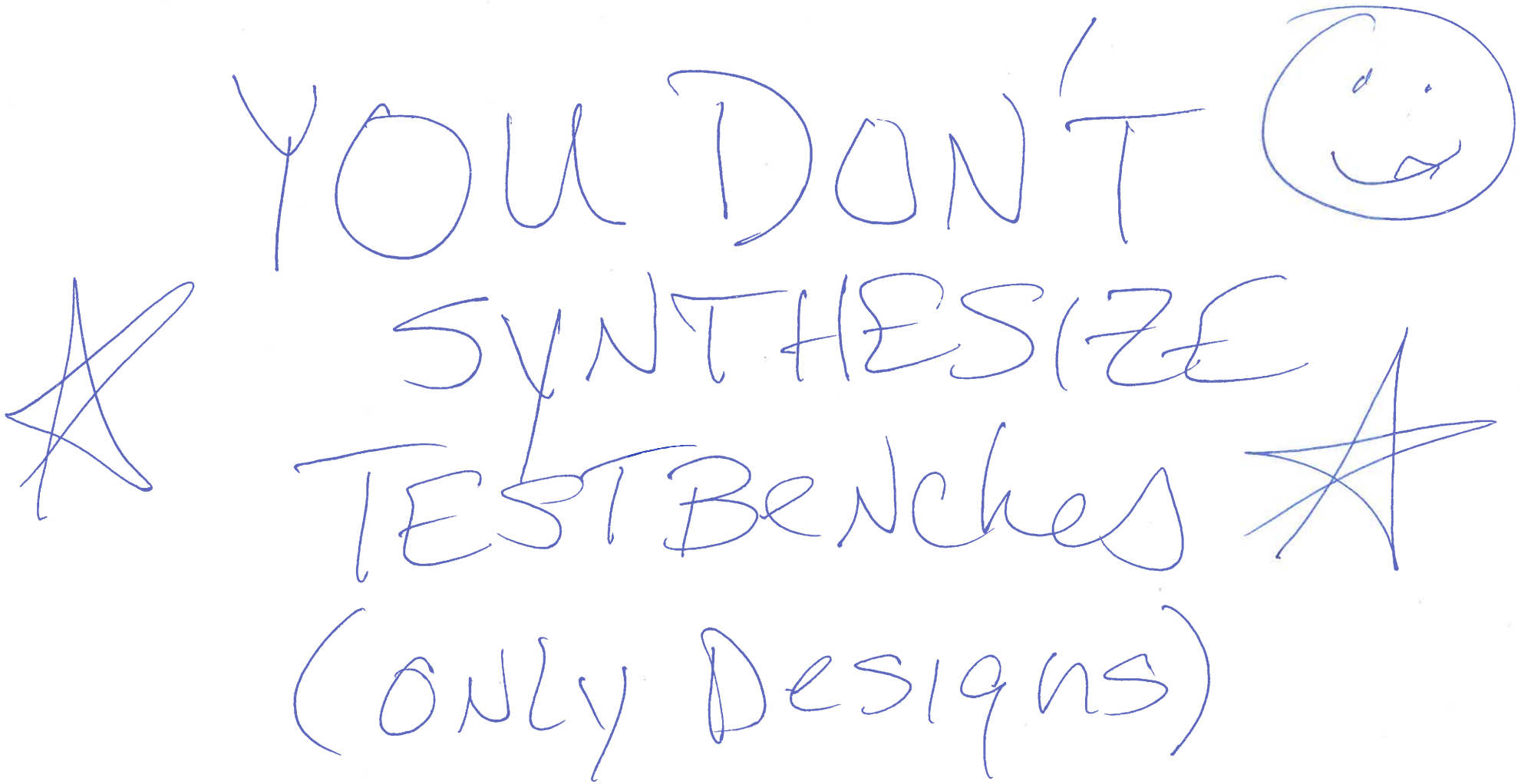
2) does not verify timing

3) testbenches have 2 purposes

- generate test patterns to exercise the digital design
- collect and output the circuit's response
- (optional) compare outputs with expected outputs

Not everything in Behavioral modeling is synthesizable

But some of that can be used in testbenches



Behavioral models have 2 structured procedures

- initial statements
- always statements

Some observations

1) no limit on the number of these in a module

2) all of them execute concurrently and begin at time t=0.

3) cannot nest them

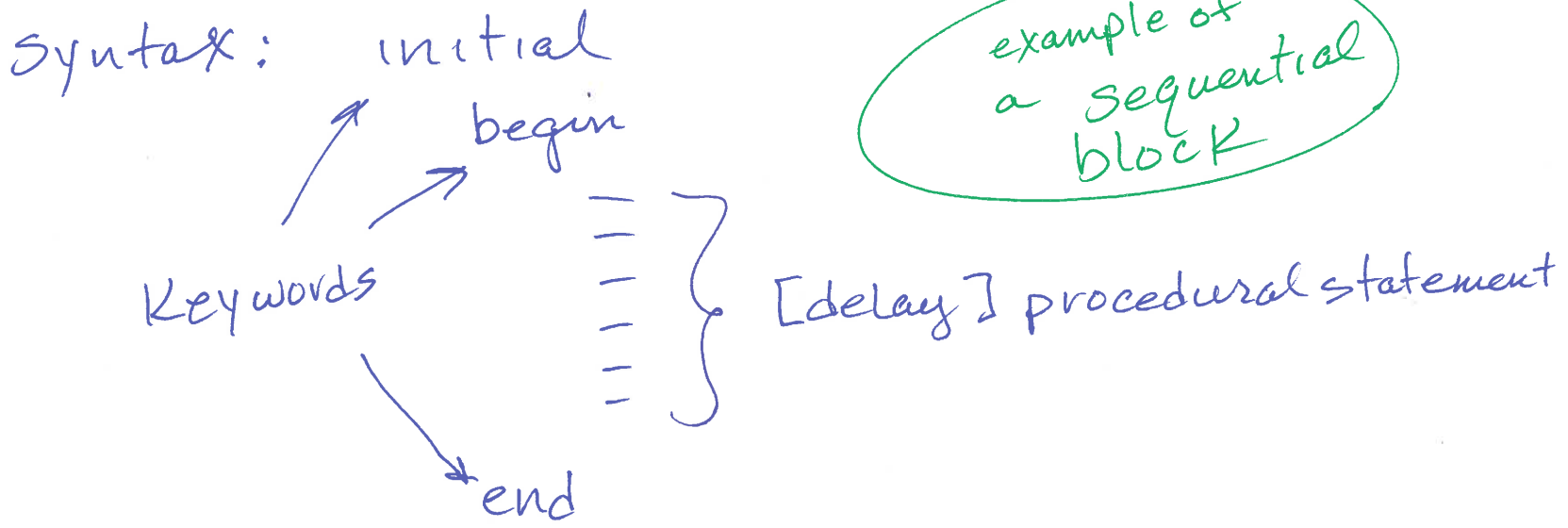
syntax: initial ← Keyword

[delay] procedural statement

- assignment (blocking or non-blocking)
- continuous assignment
- conditional statement
- case statement
- Loop
- event trigger
- parallel block
- wait statement
- disable statement

when there are than 1 procedural statements,

Syntax:



example of a sequential block

---

initial statements are

not

synthesizable

Example 7-1 *initial Statement*

```
module stimulus;

reg x,y, a,b, m;

initial
    m = 1'b0; //single statement; does not need to be grouped

initial
begin
    #5 a = 1'b1; //multiple statements; need to be grouped
    #25 b = 1'b0;
end

initial
begin
    #10 x = 1'b0;
    #25 y = 1'b1;
end

initial
    #50 $finish;

endmodule
```

time	statement executed
0	m = 1'b0;
5	a = 1'b1;
10	x = 1'b0;
30	b = 1'b0;
35	y = 1'b1;
50	$finish;

always statement

- begins executing at time t=0
- no limit on the number in a module
- no nesting
- if a sequential block, need begin/end
- operates in a continuous loop
  (always means always)

[in contrast, initial states execute only once.....]

---

what follows in

important

reg variables can only be assigned values

- when declared
- inside an initial or an always statement

```
module
 reg a;
     a = 1'b1;
endmodule
```

```
module
 reg a = 1'b1;
 reg b;

   initial
       b = 1'b0;
endmodule
```

example 7.5

always Statement

```
module clock_gen (output reg clock);

//Initialize clock at time zero
initial
        clock = 1'b0;

//Toggle clock every half-cycle (time period = 20)
always
        #10 clock = ~clock;

initial
        #1000 $finish;

endmodule
```

When do they execute?

```
module FA_Mix (A, B, Cin, Sum, Cout);
  input A, B, Cin;
  output Sum, Cout;
  reg Cout;
  reg T1, T2, T3;
  wire S1;

  xor X1 (S1, A, B);          // Gate instantiation.

  always
    begin                     // Always statement.
#5  T1 = A & Cin;
    T2 = B & Cin;
    T3 = A & B;
    Cout = (T1 | T2) | T3;
  end

  assign Sum = S1 ^ Cin;      // Continuous assignment.
endmodule
```

Two kinds of assignments

- blocking
- non-blocking

- blocking (syntax: a = b;)

  LHS updates before next statement executes

- non-blocking (syntax: a <= b;)

  RHS is recorded

  LHS updates at the end of an initial/always procedure

```
initial
  begin
    A=3; B=2;
  end

initial
  begin
    #10 A=B;
        B=A;
  end
```

A = B = 2

```
initial
  begin
    A =3; B =2;
  end

initial
  begin
    #10 A <=B;
        B<=A;
  end
```

A = 2
B = 3 they swapped

Rules

1. do not use non-blocking assignments in continuous assignment statements

2. do not mix blocking & non-blocking in ~~them~~ the same initial/always procedure

you can put #delay on a line by itself

e.g.

```
always
  begin
  #5;    // ————————————
     A=B;

  #10;
     C=D;
  end
```

Can use expressions

e.g. #(on_delay/2) A = B;