Behavioral Modeling of Sequential Circuits: Key Study Guide

1. Introduction to Behavioral Modeling

Behavioral modeling represents digital circuits at a **functional and algorithmic level**, using a higher level of abstraction compared to structural or gate-level design. This approach is primarily used for describing sequential logic.

Key Point: Behavioral HDL models describe sequential circuits by their **behavior (functionality)** rather than their physical structure.

---

2. Always Block and Sensitivity List

Basic Structure

```
// General syntax
reg output_signal; // All signals assigned inside always must be declared as reg
always @([sensitivity_list])
begin
// local variable declarations here (optional)
// body: composed of procedural statements
end
```

Critical Concepts

- **All signals assigned inside an always block must be declared as `reg`** - Unlike a wire (net data type), a reg **retains its value** until a new value is assigned (variable data type)
- **Sensitivity List:** A list of signals monitored for changes
- **Execution:** Whenever a signal in the sensitivity list changes, statements inside the always block execute sequentially
- **Note:** `begin-end` are **required** when contents have more than one statement

Sensitivity List Rules

Use Case	Example	Purpose
Combinational logic	always @(*)	Sensitive to all input changes
Level-sensitive	always @(Clk)	Responds to clock level
Rising edge	always @(posedge Clk)	Responds to rising edge
Falling edge	always @(negedge Clk)	Responds to falling edge
Multiple events	always @(posedge Clk, negedge Reset)	Asynchronous reset

---

3. Latches vs. Flip-Flops vs. Registers

Component	Clock Sensitivity	Description	Use
Latch	Level-sensitive	Responds to clock level (HIGH or LOW)	Transparent operation
Flip-Flop	Edge-sensitive	Responds to clock edge (rising or falling)	Synchronous circuits
Register	Edge-sensitive	Multi-bit version of flip-flop (D and Q > 1-bit)	Data storage, shift operations

---

4. Example 1: Positive-Level D Latch

Module Code:

```
// Positive-level triggered D Latch with Active High Reset
module D_Latch (
input wire Clk, Reset, D,
output reg Q
);
// sensitivity list uses () for combinational-like behavior
always @()
begin
if (Reset)
Q <= 1'b0; // Reset when active
```

```
else
if (Clk) // Level-sensitive to clock
Q <= D; // Capture input when Clk is HIGH
end
endmodule
```

Key Observation: The sensitivity list `always @(*)` makes it level-sensitive, capturing data whenever the clock is HIGH.

Testbench Example:

```
// Clock generator - toggles every 5 ns
always
#5 clk = ~clk;

// Generate stimuli with time delays
initial begin
$display("Starting simulation at %0d ns...", $time);
D = 1'b0; #12 // Set input 12 ns before clock edge
D = 1'b1; #25 // Change after 25 ns
D = 1'b0; #10
$display("Finished simulation at %0d ns.", $time);
$stop;
end
```

Simulation Output Example:

```
Time = 0 ns Clk = 0 Reset = 1 D = 0 Q = 0

Time = 5 ns Clk = 1 Reset = 1 D = 0 Q = 0
(Reset still active)

Time = 10 ns Clk = 0 Reset = 0 D = 0 Q = 0
(Reset released)

Time = 15 ns Clk = 1 Reset = 0 D = 1 Q = 1
(Latch captures D=1)
```

Time = 20 ns Clk = 0 Reset = 0 D = 1 Q = 1 (Q holds value)

---

5. Example 2: Positive-Edge D Flip-Flop

Module Code:

```
// Positive-edge triggered D Flip-Flop with Active High Reset
module D_FF (
input wire Clk, Reset, D,
output reg Q
);
always @(posedge Clk) // Key difference: Edge-sensitive!
begin
if (Reset)
Q <= 1'b0;
else
Q <= D; // Capture input only on rising edge
end
endmodule

// Negative-edge triggered version (if needed):
// always @(negedge Clk)
```

Critical Difference from Latch:

- Latch: `always @(*)` - Level-sensitive
- Flip-Flop: `always @(posedge Clk)` - **Edge-sensitive**

Simulation Comparison:

Time (ns)	Latch Q	Flip-Flop Q	Note
15 (posedge)	1	1	Both capture D=1
20 (falling edge)	1	1	Latch releases, FF holds
25	1	1	FF maintains value
37 (D changes to 0 during Clk=1)	0	1	**Latch responds, FF doesn't**
40	0	1	Critical difference!
45 (next posedge)	0	0	FF finally captures

---

6. Non-Blocking vs. Blocking Assignments

Blocking Assignment (=)

- **Use in:** Combinational logic
- **Behavior:** Sequential execution within the always block
    1. Right-hand side evaluated
    2. Immediately assigned to left-hand side
    3. Next statement uses updated value

```
// Blocking: x gets new value before y is evaluated
reg x, y;
always @(*)
begin
x = a & b; // Step 1: x ← (a & b)
y = x | c; // Step 2: y ← (NEW x) | c
end
```

Non-Blocking Assignment (<=)

- **Use in:** Sequential logic
- **Behavior:** Concurrent evaluation
    1. All right-hand sides evaluated with OLD values
    2. All assignments deferred until time step completes
    3. Updates happen simultaneously

```
// Non-blocking: x and y evaluated with OLD x value
reg x, y;
always @(posedge Clk)
begin
x <= a & b; // Queue: x ← (a & b)
y <= x | c; // Queue: y ← (OLD x) | c (concurrent!)
// Both assignments apply together
end
```

Why This Matters

Blocking (Wrong for Sequential):

- Creates simulation/synthesis mismatches
- Order-dependent behavior
- Unpredictable hardware

Non-blocking (Correct for Sequential):

- Hardware independent of assignment order
- Simulation matches synthesis
- Predictable sequential behavior

---

7. Synchronous vs. Asynchronous Reset

Synchronous Reset

- Reset is sampled only at **clock edge**
- Safer for reliable synchronization
- Reset happens on next clock pulse

```
module D_FF_SyncReset (
input wire Clk, nReset, D,
output reg Q
);
// Notice: Only Clk in sensitivity list
always @(posedge Clk)
begin
if (!nReset) // Check reset at clock edge
Q <= 1'b0;
else
Q <= D;
end
endmodule
```

Asynchronous Reset

- Reset is **immediate**, independent of clock
- Faster reset response
- Can cause metastability issues if not careful

```
module D_FF_AsyncReset (
input wire Clk, nReset, D,
output reg Q
);
// Notice: Both Clk AND nReset in sensitivity list
always @(posedge Clk, negedge nReset)
begin
if (!nReset) // Reset responds immediately!
Q <= 1'b0;
else
Q <= D;
end
endmodule
```

Comparison Waveforms

Synchronous Reset: Q remains at current value until next clock edge, then resets
Asynchronous Reset: Q resets immediately when nReset goes LOW, regardless of clock

Convention: Reset signals are typically **active-low** (denoted by `n` prefix like `nReset`) by industry standard.

8. Incorporating Logic: Enable and Muxing

Enable Signal Implementation

```
module D_FF_Enable (
input wire Clk, nReset, D, En,
output reg Q
);
always @(posedge Clk) begin
if (!nReset)
Q <= 1'b0;
else
if (En) // Only update when enabled
Q <= D;
// NOTE: No "else" here produces a latch!
// Q holds its previous value when En=0
end
endmodule
```

Critical Point: Omitting the `else` clause creates an **implicit latch** - the register holds its value when the condition is false.

With Logic and Selection

```
// Mux between two inputs with logic
always @(posedge Clk) begin
if (!nReset)
Q <= 1'b0;
else
if (Sel)
Q <= B; // Select B
else
Q <= A & B; // Select AND result
end
```

---

9. 4-Bit Binary Counter with Parallel Load

Block Diagram Function Table:

Clear	CLK	Load	Count	Function
0	X	X	X	Clear to 0
1	↑	1	X	Load inputs
1	↑	0	1	Count next binary state
1	↑	0	0	No change

Module Implementation:

```
module Binary_Counter_4_Par_Load (
output reg [3:0] A_count, // Data output (4-bit)
output C_out, // Carry out (overflow)
input [3:0] Data_in, // Data input
input Count, // Active high to count
input Load, // Active high to load
input CLK, // Positive-edge sensitive
input Clear_b // Active low clear
);
// Carry out when counting and A_count = 15 (1111)
assign C_out = Count && (~Load) && (A_count == 4'b1111);
```

```
always @(posedge CLK, negedge Clear_b)
begin
    if (~Clear_b)              // Asynchronous reset
        A_count <= 4'b0000;
    else if (Load)             // Synchronous load
        A_count <= Data_in;
    else if (Count)            // Count increment
        A_count <= A_count + 1'b1;
    else                       // Hold value (redundant but explicit)
        A_count <= A_count;
end
```

```
endmodule
```

Key Features:

- **Asynchronous clear** using `negedge Clear_b`
- **Priority logic:** Clear > Load > Count > Hold
- **Carry output:** Generated combinationally for overflow detection
- **Integer arithmetic:** `A_count + 1'b1` for increment

---

10. Guidelines for Good Synthesis of Sequential Circuits

Essential Rules

1. **Use Non-Blocking Assignments (<=)** for sequential logic
   - ✓ Hardware is independent of assignment ordering
   - ✓ Simulation matches synthesis behavior
   - ✓ Predictable results
2. **Use Blocking Assignments (=)** for combinational logic
   - Only within `always @(*)` blocks
   - Sequential use of blocking is NOT recommended
3. **Never Mix Assignment Types** in same always block
   - Causes unpredictable behavior

    - Creates synthesis errors
4. **One Driver Per Signal**
    - Do NOT assign same variable from multiple always blocks
    - Creates race conditions (even with non-blocking!)
    - Exception: Use `assign` for combinational, one `always` for sequential
5. **Edge Sensitivity for Proper Sequencing**
    - Use `posedge` or `negedge` for flip-flops
    - Use `@(*)` only for combinational logic
    - Include both edges if needed: `always @(posedge Clk, negedge Reset)`

Summary Table

Rule	**Do**	**Don't**
Sequential assignments	`Q <= D;`	`Q = D;`
Combinational assignments	`y = a & b;`	`y <= a & b;`
Reset in sequential	Include in sensitivity	Omit from sensitivity
Multiple drivers	Use unique always blocks	Assign same signal twice
Assignment mixing	Keep separate	Mix `=` and `<=` together

---

11. Simulation Verification Strategy

Testbench Structure

`timescale 1 ns / 1 ps // Time unit / precision

module tb_DUT ();
// Declare signals
reg Clk, Reset, Input_signal;
wire Output_signal;

```
// Instantiate DUT (Device Under Test)
DUT_Module UUT (
    .Clk(Clk),
    .Reset(Reset),
    .Input(Input_signal),
    .Output(Output_signal)
);

// Clock generation (50% duty cycle, 10 ns period)
initial
    Clk = 1'b0;
always
    #5 Clk = ~Clk;         // Toggle every 5 ns

// Reset sequence
initial begin
    Reset = 1'b1;          // Assert reset
    #10 Reset = 1'b0;      // Release after 10 ns
end

// Test stimulus
initial begin
    $display("Starting test at %0d ns", $time);
    // Apply test patterns with delays relative to clock
    Input_signal = 1'b0; #12   // Delay before next clock
    Input_signal = 1'b1; #20
    // ... more test cases
    $stop;
end

// Monitoring
initial
    $monitor("Time=%0d ns: Clk=%b Reset=%b Input=%b Output=%b",
             $time, Clk, Reset, Input_signal, Output_signal);
```

endmodule

Key Simulation Directives

- `$display()` - Print message once
- `$monitor()` - Continuous monitoring (print on signal changes)
- `$stop` - Stop simulation
- `$time` - Current simulation time
- `#delay` - Time delay in simulation units

---

12. Synthesis Summary

Key Synthesis Behaviors:

- **posedge/negedge in sensitivity list** → Hardware flip-flop with automatic clock
- **Incomplete if-else statements** → Produces latches (be careful!)
- **Non-blocking assignments** → Synthesis-friendly, predictable hardware
- **Proper reset handling** → Synchronous (sampled at edge) or asynchronous (immediate)

Best Practice Checklist:

- ✓ Sensitivity list includes all inputs OR clock edges
- ✓ Use non-blocking assignments throughout sequential always
- ✓ Reset properly handled (synchronous or asynchronous)
- ✓ No race conditions (one driver per signal)
- ✓ Testbench validates expected behavior before synthesis
- ✓ Waveforms confirm simulation matches hardware expectations

---

Key Takeaways

1. **Behavioral modeling** describes circuits functionally at high abstraction
2. **Always blocks** execute based on sensitivity list changes
3. **Latches** are level-sensitive; **flip-flops** are edge-sensitive
4. **Non-blocking (<=)** for sequential; **blocking (=)** for combinational
5. **Asynchronous reset** is immediate; **synchronous reset** waits for clock
6. **Enable signals** create implicit latches if else clause omitted
7. **Counters and registers** combine multiple techniques for complex circuits
8. **Synthesis expects** non-blocking assignments in sequential logic
9. **Testbenches simulate** before synthesis to verify behavior
10. **Guard against race conditions** - one signal driver per always block