San Francisco Bay University  
EE461 Digital Design and HDL

Week#3 Combinational Logic

**QA.**

**Code 1:**

always@(A or B) begin

if (A)

C = B ^ A;

else

C = D & E;

F = C | A;

end

**issues:**

**-Sensitivity list.**

**-Non-blocking assignments**.

**-Causal dependency**

**Fixed code1:**

always @(A or B or D or E) begin

if (A)

C = B ^ A;

else

C = D & E;

F = C | A;

end

**Code 2:**

always@(B)

C = |B;

always@(E)

C = ^E;

**issues:**

-Conflicting assignments to C.

- Combining both conditions into one block.

#### Fixed Code:

always @(B or E) begin

C = |B; // Reduction OR for B

C = ^E; // Reduction XOR for E, overwrites the previous value

end

### Code 3:

always@(posedge clock)

if(A)

Q <= D;

always@(Q or E)

case (Q)

0: F = E;

default: F = 1;

endcase

**Issues:**

-Sequential and combinational logic separation.

**-No initialization of Q**:

**-Simulation-specific issues**:

-clock example.

#### Fixed Code:

always@(posedge clock) begin

if (A)

Q <= D;

end

always@(Q or E) begin

case (Q)

0: F = E;

default: F = 1;

endcase

end

**Code 4**

module top;

wire B;

bar u1 (A, B);

bar u2 (C, B);

endmodule

module bar (input D; output wire E);

assign E = ~D;

endmodule

#### Issues:

**-Multiple drivers for** B.

- **Fix**.

#### Fixed Code:

module top;

wire B1, B2;

bar u1 (A, B1);

bar u2 (C, B2);

endmodule

module bar (input D, output wire E);

assign E = ~D;

endmodule

### Code 5:

module foo(input A, B; output reg E);

wire C, D;

always@(posedge clock)

E = B & D;

assign C = A ^ D;

assign D = C | B;

endmodule

#### Issues:

**-Conflicting assignments.**

**- Incorrect usage of wires.**

#### Fixed Code:

module foo(input A, B, input clock, output reg E);

reg D;

wire C;

always@(posedge clock) begin

E = B & D;

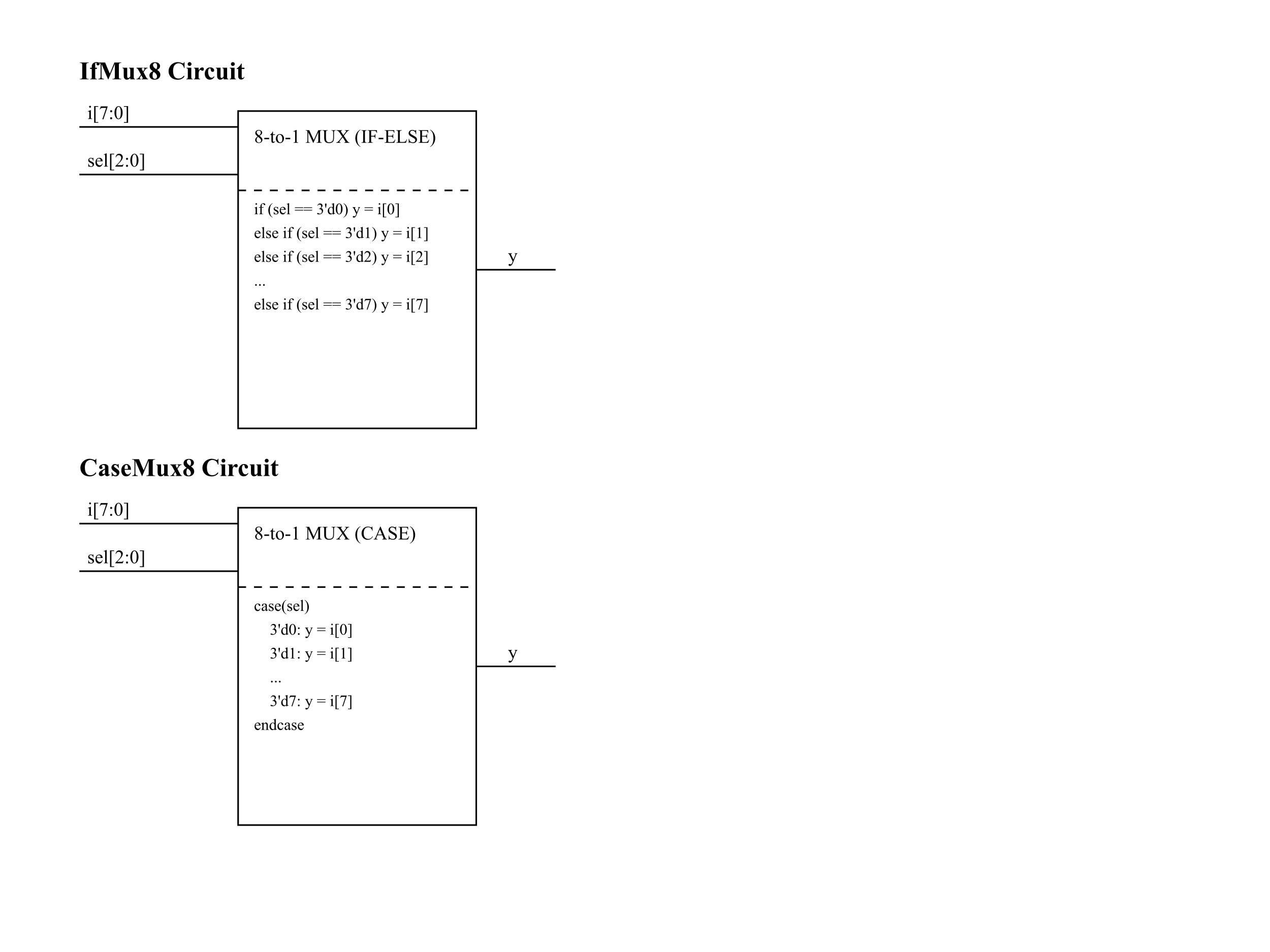
D <= C | B; // Sequential update for D

end

assign C = A ^ D;

endmodule

**Qb.** According to the modules IfMux8 and CaseMux8, draw the corresponding circuits and compare them.



**Comparing the two modules.**

1. **Functionality:** Both IfMux8 and CaseMux8 implement an 8-to-1 multiplexer. They take an 8-bit input i[7:0], a 3-bit select input sel[2:0], and output a single bit y based on the select input.
2. **Circuit structure:** The basic structure of both circuits is identical:

* 8-bit input bus (i[7:0])
* 3-bit select input (sel[2:0])
* Single-bit output (y)
* Internal logic block for multiplexing

1. **Implementation differences:**

* IfMux8 uses a series of if-else if statements.
* CaseMux8 uses a case statement.

1. **Behavioral equivalence:** Both modules produce identical behavior in simulation and synthesis, selecting the appropriate input bit based on the select lines.
2. **Synthesis considerations:**

* For this simple 8-to-1 multiplexer, modern synthesis tools will likely produce very similar or identical hardware for both implementations.
* The synthesized hardware will typically consist of a network of multiplexers or pass transistors controlled by decoded select signals.