ECE 385 – Digital Systems Laboratory

Lecture 6 –Lab 5 Intro

Zuofu Cheng

Fall 2019

Link to Course Website

ECE ILLINOIS

ILLINOIS

Lab 4 Hints

- Read tutorials (Introduction to Quartus, Introduction to SystemVerilog) and make a project for the 8-bit logic processor.
  - Full design files are provided for 4-bit version, including testbench
  - Extend to 8-bit (code is well commented), use 8-bit test bench included (1 demo point)
  - Show TA RTL block diagram (generated from Quartus) for 8-bit logic processor
- Design 3 16-bit adders and test on the FPGA board
  - Will need state machine for testing (provided) and pin assignments (made with pin planner tool)
  - Analyze performance, power, and area for each adder
  - Should only need to write combinational logic (test setup is provided for you)
  - Demo on FPGA board using built in switches and LEDs (don't need switchbox)
- Working in 3022/4072 ECEB:
  - FPGA PAR (place and route) operation is extremely disk intensive
  - Performance may be very slow when running on network drive
  - Use C:\ECE385Temp – make new folder with your NETID (will have private permissions)
  - Transfer to network drive or USB after **each session,** C:\ECE385Temp is **not saved or mirrored**

Lab 4 Goals

- Carry ripple adder and its structures
- Carry look-ahead adder
- Carry select adder
- Demo platform already provided
- Should only need to write combinational logic for adders!
- **Remember you need to add pin assignments to demo on FPGA!**
- For fun and to satisfy curiosity – try making module using + operator to see what hardware gets synthesized
- Code submission reminder:
  - Due immediately after the demo (entire project folder)
  - Need to provide sufficient comments (will be graded)

SystemVerilog Summary – So far

- Made of instances of modules (e.g. 7400 chips) the logic connections in between (e.g. wires)
- Behavior of modules defined by always_comb, always_ff, and assign procedure
- Always_comb and assign are used to describe combinational logic: both ways of writing do the same
- Recommend always_comb over assign
  - Always_comb lets you enforce dependencies (blocking assignment)
  - Begin/end necessary when block exceeds a line (similar to {} in C/C++)
  - Procedure blocks resolve in unknown order
- SystemVerilog operators
  - & | ^ ~ (bitwise AND, OR, XOR, NOT)

```
logic a, b, c;

assign a = b | c;

always_comb
begin
   a = b | c;
end
```

SystemVerilog Summary – Continued

- For now, always use "logic" type – think of this as "a place that has a logic level"
- Logic type actually can take on 4 different values (0, 1, z – high impedance, x – unknown/don't care)
- Immediates (for type logic) are specified {nbits}'{radix}{value}
- Note that SystemVerilog has 2 different types of arrays: packed and unpacked
- Packed arrays are typically used for busses/n-bit signals
  - Use packed arrays whenever you care about adjacent bits – hence "packed"
  - E.g. logic [7:0] a creates an 8-bit wide logic can do operations with a single bit or a range of bits such as split/concatenate
  - Packed dimension comes **before** variable
- Unpacked arrays are used for addresses
  - Use unpacked arrays when you want multiple addresses
  - Unpacked arrays come **after** variable, and can be single number e.g. logic a [10];
  - Unpacked arrays have no special relationship between adjacent elements, they all have independent addresses (like a RAM)

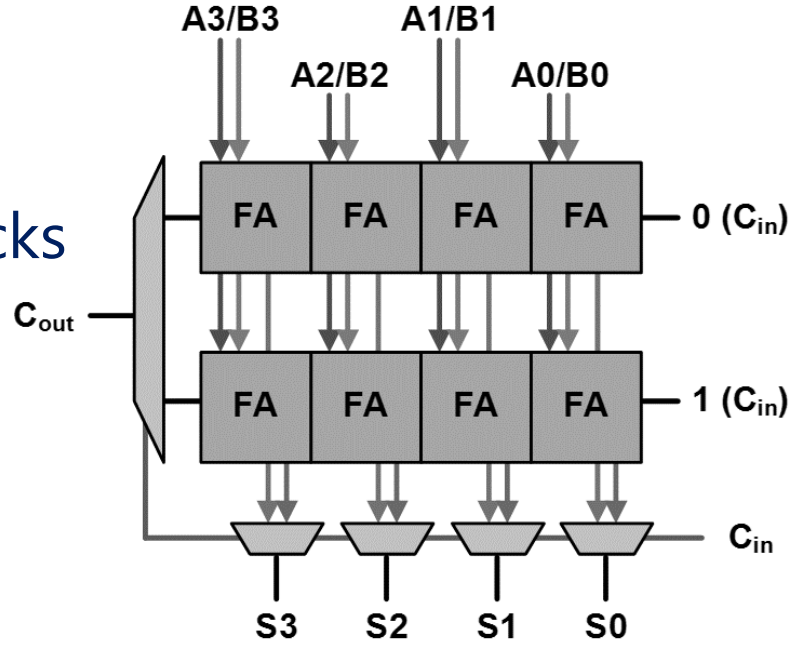
```
logic [7:0] a;
logic [3:0] b;
logic [1:0] c;
logic [15:0] d;

assign b = a[7:4];
assign d = {6'b000000, a, c};
```

Lab 4 Notes

- Report should contain simulation traces
  - Relevant output e.g. P/G bits for CLA
  - Note: simulation does not take into account timing (all combinational logic is assumed to take Δt time (minimum simulation time unit)
  - Possible to do simulation with timing (post place & route) but in practice simulation performance is very slow
- Include timing analysis in report
  - Demo setup (top-level) takes inputs from register and stores output to register
  - Why is this important?

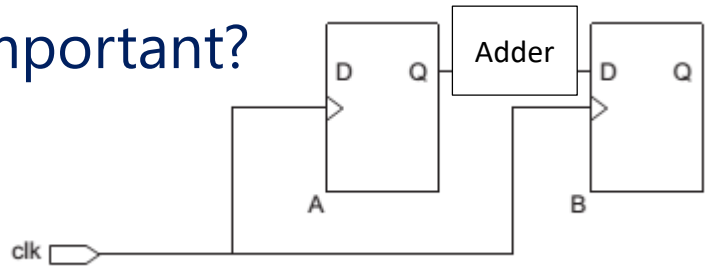
Carry Select Adder

- An alternative approach to dealing with carry delay is to pre-compute the result for both C = 0 and C = 1 for each n bit block
- This requires twice as many full adders (and additional multiplexors) but reduced carry propagation delay
- 4-bit version show on right
- Create 16-bit version from 4-bit
  - For this lab, can ripple 4 x 4-bit blocks
  - Why might this not be ideal?



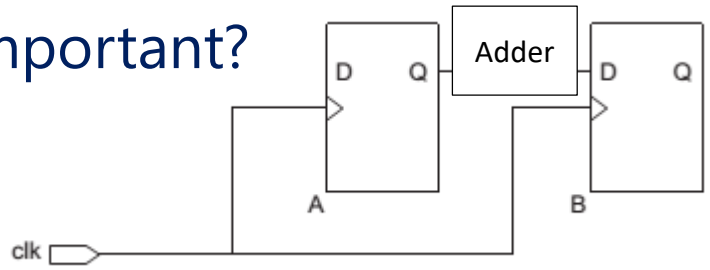
Timing Analysis

- Report should contain simulation traces
  - Relevant output e.g. P/G bits for CLA
  - Note: simulation does not take into account timing (all combinational logic is assumed to take Δt time (minimum simulation time unit)
  - Possible to do simulation with timing (post place & route) but in practice simulation performance is very slow
- Include timing analysis in report
  - Demo setup (top-level) takes inputs from register and stores output to register
  - Why is this important?



Lab 4 Notes

- Report should contain simulation traces
  - Relevant output e.g. P/G bits for CLA
  - Note: simulation does not take into account timing (all combinational logic is assumed to take Δt time (minimum simulation time unit)
  - Possible to do simulation with timing (post place & route) but in practice simulation performance is very slow
- Include timing analysis in report
  - Demo setup (top-level) takes inputs from register and stores output to register
  - Why is this important?



Experiment 5: Goals

- Design a 2's Complement 8-bit Multiplier unit in SystemVerilog using **logic operations**.
- Do not use SystemVerilog Arithmetic Operations.
  - *i.e.* Do not use A = A + B; or A = A – B;
  - This includes shift operations (should be created using logic not SV operators)
- Follow Experiment-5 description in Lab Notes
  - Bring your code on a memory stick
  - Bring a detailed block diagram of your design
    - Components, ports and interconnections labeled
- A Note on Cooperation with others
  - You are encouraged to cooperate with others in the class. However, each team must write its own SystemVerilog source and Lab Report.
  - You are permitted to exchange ideas, algorithms, high level flow charts and so on – BUT NO Entities, Components, Architectures, or Files of any kind.

Quick Re-cap about Two's Complement

- **Two's complement** is equivalent to taking the one's complement (invert all bits) and then adding one
  - The fundamental arithmetic operations are identical to those for unsigned binary numbers
  - Zero has only a single representation
  - Leading bit carries the sign of the number (0 – positive, 1 – negative)
  - 2's complement is a weighted representation where he weight of each bit is a power of two, except for the most significant bit, whose weight is the negative of the corresponding power of two:

$$A_{10} = -a_{k-1}*2^{k-1} + a_{k-2}*2^{k-2} + \ldots a_1*2^1 + a_0*2^0$$

8-bit two's-complement integers

Bits	Unsigned value	2's complement value
0111 1111	127	127
0111 1110	126	126
0000 0010	2	2
0000 0001	1	1
0000 0000	0	0
1111 1111	255	−1
1111 1110	254	−2
1000 0010	130	−126
1000 0001	129	−127
1000 0000	128	−128

8-bit 2's Complement Multiplication (pen & paper)

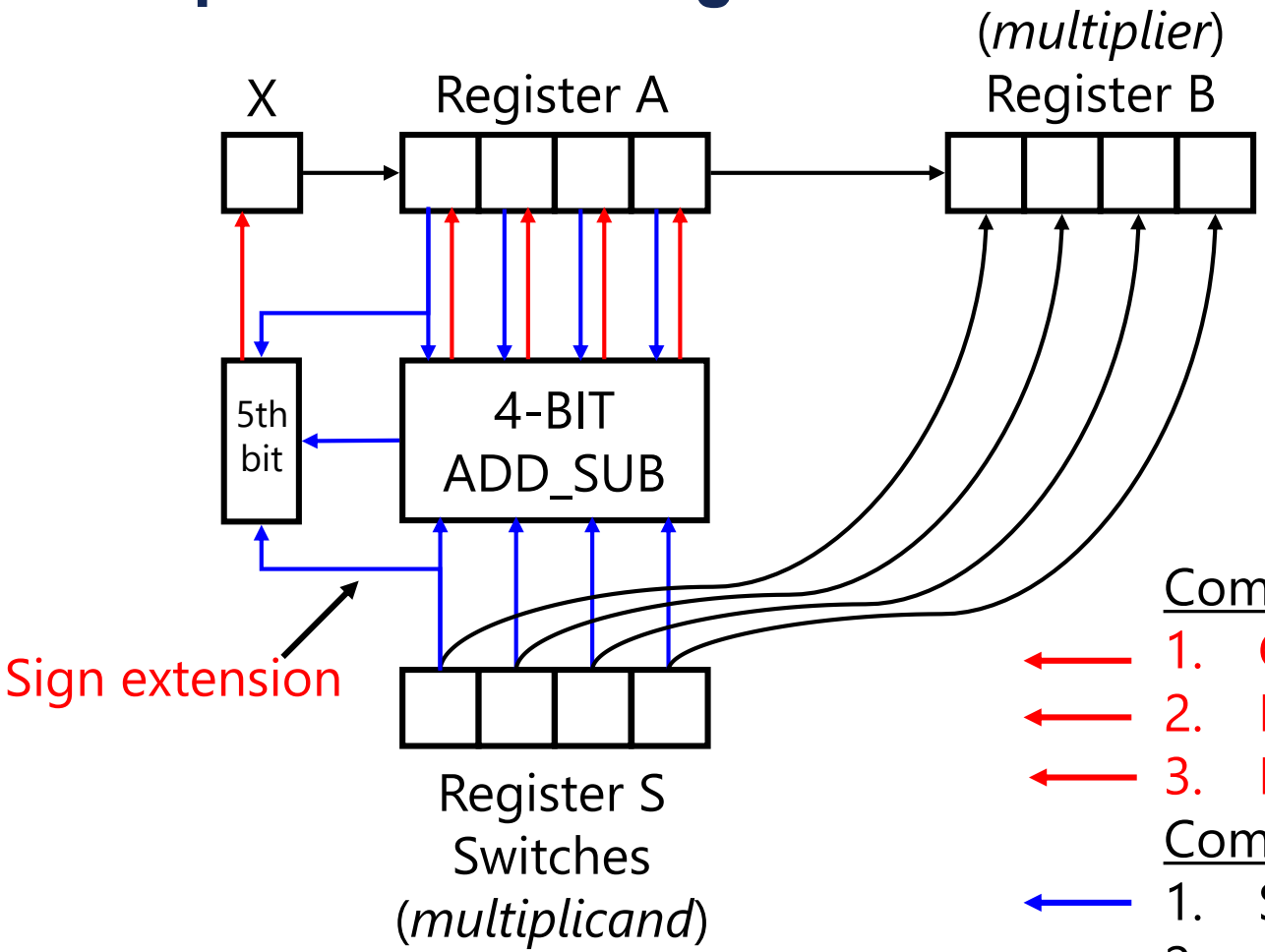
```
      00000111                    7  (multiplicand)
    x 11000101              x (-)59  (multiplier)
    ----------              --------
      00000111                (-)413
     +00000000x
    +00000111xx
   +00000000xxx
  +00000000xxxx
 +00000000xxxxx
+00000111xxxxxx
-00000111xxxxxxx   Subtract last partial product
1111111001100011    since it is from the sign bit
```

8-bit 2's Complement Multiplication (add & shift)

Multiply 0000 0111 × 1100 0101

Function	X	A_{7-0}	B_{7-0}	M	Next Step in words
	0	0000 0000	11000101	B_0 1	Since M = 1, multiplicand will be added to A.
ADD	0	0000 0111	11000101	1	Shift XAB by one bit after ADD complete
SHIFT	0	0000 0011	1 1100010	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	0	0000 0001	11 110001	1	Add S to A since M = 1.
ADD	0	0000 1000	11 110001	1	Shift XAB by one bit after ADD complete
SHIFT	0	0000 0100	011 11000	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	0	0000 0010	0011 1100	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	0	0000 0001	00011 110	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	0	0000 0000	100011 11	1	Add S to A since M = 1
ADD	0	0000 0111	100011 11	1	Shift XAB by one bit after ADD complete
SHIFT	0	0000 0011	1100011 1	1	Subtract S from A since 8th bit M = 1.
SUB	1	1111 1100	1100011 1	1	Shift XAB after SUB complete
SHIFT	1	1111 1110	01100011	1	8th shift done. Stop. 16-bit Product in AB.

4-bit Multiplier: Block Diagram



Commands (push buttons)

1. ClearXA_loadB
2. RUN (Execute)
3. Reset

Commands (control)

1. SHIFT XAB
2. ADD (A+S → XA)
3. SUB (A–S → XA)

SHIFT, ADD *and* SUB *are derived signals from State Controller. They last only one clock cycle*

If-else and Case

- Statements in the same **if** or **case** block are processed sequentially and thus implies priority
  - Different **if**/**case** blocks are processed in parallel
- All cases should be covered to avoid latch inference or synthesis failure
  - If statements should synthesize cascade of 2:1 MUX (therefore need **else** statement)
  - Case statements should synthesize N:1 MUX (cover all cases or use **default**)
- Example: When conditions overlap
  - Left: Not all cases are covered, will fail to synthesize (inside `always_comb`) or synthesize latches (inside Verilog style `always`)
  - Right: All cases covered, synthesizes correctly into MUXes

```
if (x == 4'b1111)
   z = b;
...
```

```
if (x == 4'b1111)
   z = b;
else
   z = a;
```

Default conditions

- Need to cover all cases when synthesizing MUXes
- Examples of how to do this
  - All three examples are synthesizable and create a 2:1 MUX

```
if (x == 4'b1111)
   z = b;
else
   z = a;
```

```
case (x)
   4'b1111  : z = b;
   others   : z = a;
```

```
z = a;
if (x == 4'b1111)
     z = b;
```

- Note: Example 3 works because of blocking assignment (always use blocking assignment for combinational logic!)

Consider: ALU from Lab 3/4

- Consider two different cases:
- Both synthesize working combinational logic – what is difference?

```
always_comb
     begin
       unique case (F)
            3'b000   : F_A_B = A_In & B_In ;
            3'b001   : F_A_B = A_In | B_In ;
            3'b010   : F_A_B = A_In ^ B_In ;
            3'b011   : F_A_B = 1'b1;
            3'b100   : F_A_B = A_In &~ B_In ;
            3'b101   : F_A_B = A_In |~ B_In ;
            3'b110   : F_A_B = A_In ^~ B_In ;
            3'b111   : F_A_B = 1'b0;
            default  : F_A_B = 1'b0;
       endcase
   end
```

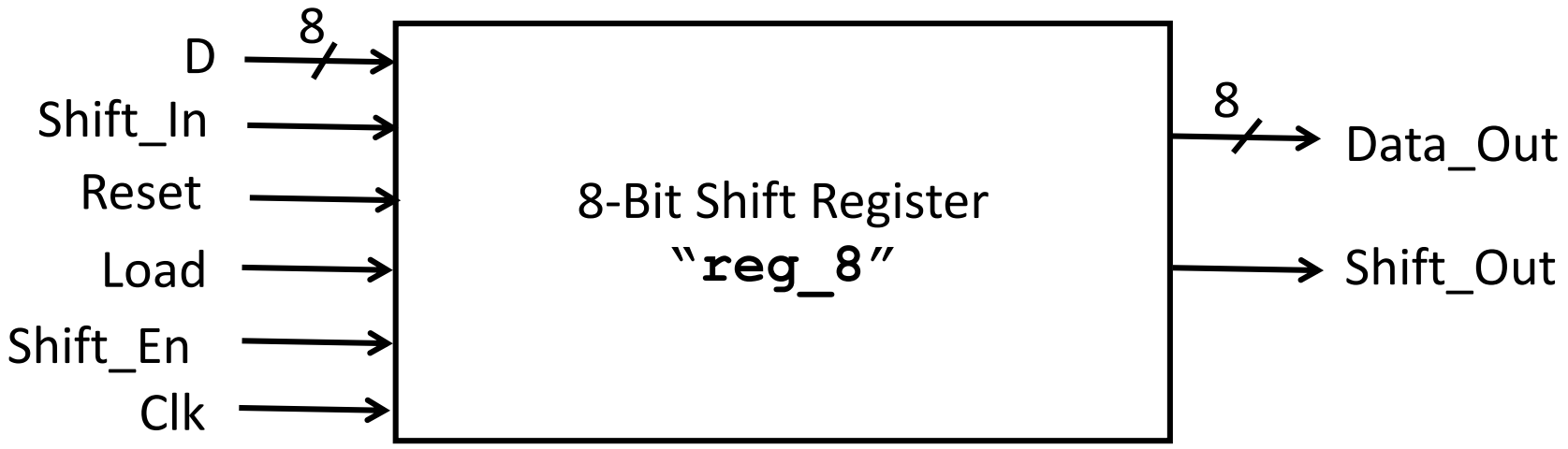
```
always_comb
      begin
          if (F == 3'b000)
               F_A_B = A_In & B_In;
          else if (F == 3'b001)
               F_A_B = A_In | B_In;
          else if (F == 3'b010)
               F_A_B = A_In ^ B_In;
          else if (F == 3'b011)
               F_A_B = 1'b1;
          else if (F == 3'b100)
               F_A_B = A_In &~ B_In;
          else if (F == 3'b101)
               F_A_B = A_In |~ B_In;
          else if (F == 3'b110)
               F_A_B = A_In ^~ B_In;
          else
               F_A_B = 1'b0;
end
```

If-else vs Case Statements

- Case statement creates 8:1 MUX:
- If statement creates cascade of 2:1 MUX:

Consider: 8-bit Shift Register

```
//8-Bit Shift Register

module reg_8 ( input logic          Clk, Reset, Shift_In,
                                    Load, Shift_En,
               input logic  [7:0]   D,
               output logic         Shift_Out,
               output logic [7:0]   Data_Out);
```



8-bit Shift Register: Asynchronous Reset

```
always_ff @ (posedge Clk or posedge Reset)
begin

    if (Reset)                  // Asynchronous Reset
        Data_Out <= 8'h0;
    else if (Load)
        Data_Out <= D;
    else if (Shift_En)
        Data_Out <= { Shift_In, Data_Out[7:1] };
end

assign Shift_Out = Data_Out[0];
```

8-bit Shift Register: Synchronous Reset

```
always_ff @ (posedge Clk)
begin

    if (Reset)                  // Synchronous Reset
        Data_Out <= 8'h0;
    else if (Load)
        Data_Out <= D;
    else if (Shift_En)
        Data_Out <= { Shift_In, Data_Out[7:1] };
end

assign Shift_Out = Data_Out[0];
```

8-bit Shift Register: 2 - Always

```
logic [7:0] Data_Next;

always_ff @ (posedge Clk) begin
    Data_Out <= Data_Next;
end

always_comb begin
Data_next = Data_Out;
    if (Reset)                  // Synchronous Reset
     Data_next = 0;
    else if (Load)
     Data_next = D;
    else if (Shift_En)
     Data_next = { Shift_In, Data_Out[7:1] };
end
```

Why Use 2-Always Method?

- Most modules (even synchronous ones) have some combinational logic (think shift register, state machine, counter)
- Consider two modules, does count ever hit 0xAF (175)?

```
always_ff @ (posedge Clk)
begin
     count <= count + 1;
     if (count == 8'hAF)
        count <= 0;
end
```

```
always_ff @ (posedge Clk)
begin
     count <= count_next;
end

always_comb
begin
    count_next = count + 1;
    if (count == 8'hAF)
        count_next = 8'h00;
end
```

State Machines in SystemVerilog

- FSM is the backbone of hardware design
- As HDL is mostly concurrent, any algorithm that requires sequential, step-by-step operations is better implemented as FSM
- Techniques of FSM implementation in HDL are well-studied and well established

- Reference: Clifford E. Cummings, "Synthesizable Finite State Machine Design Techniques Using the New SystemVerilog 3.0 Enhancements," *SNUG 2003*

General State Machine Tips

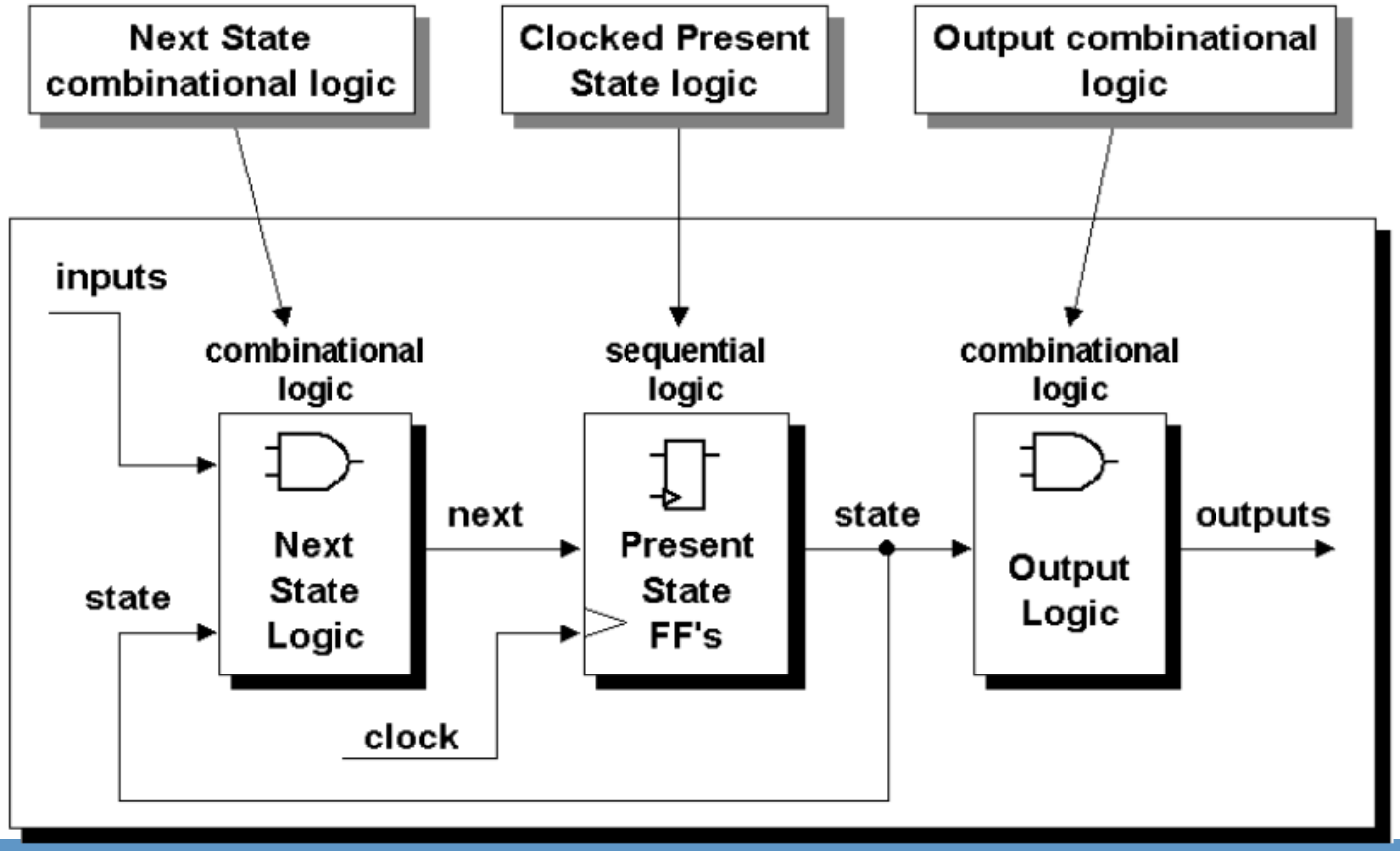
- Make each FSM design a separate module
- Use `enum` or `parameter` to encode the states
  - Some people use `` `define ``, which is global, and it causes problems when multiple FSMs share some state names
  - `enum` is preferred because most simulation tools display state names rather than state encodings in waveform
- Make `state` and `next_state` declarations right after the `enum` or `parameter` assignments
- This makes the state variables easily readable by either designers or synthesis tools
- Easier to change encoding style (which will be introduced shortly)

State Machine Patterns

- Three common patterns for writing state machines
  - **2-Always:** Two always block style with combinational outputs **(Recommended)**
  - **1-Always:** One sequential always block style with registered outputs **(Not Recommended)**
  - **3-Always:** Three always block style with registered outputs **(OK)**
- **Use either 2-Always or 3-Always**
  - Choice depends on whether outputs need to be registered or not
  - Registered outputs
    - Outputs appear one cycle later
    - Why might this be useful?

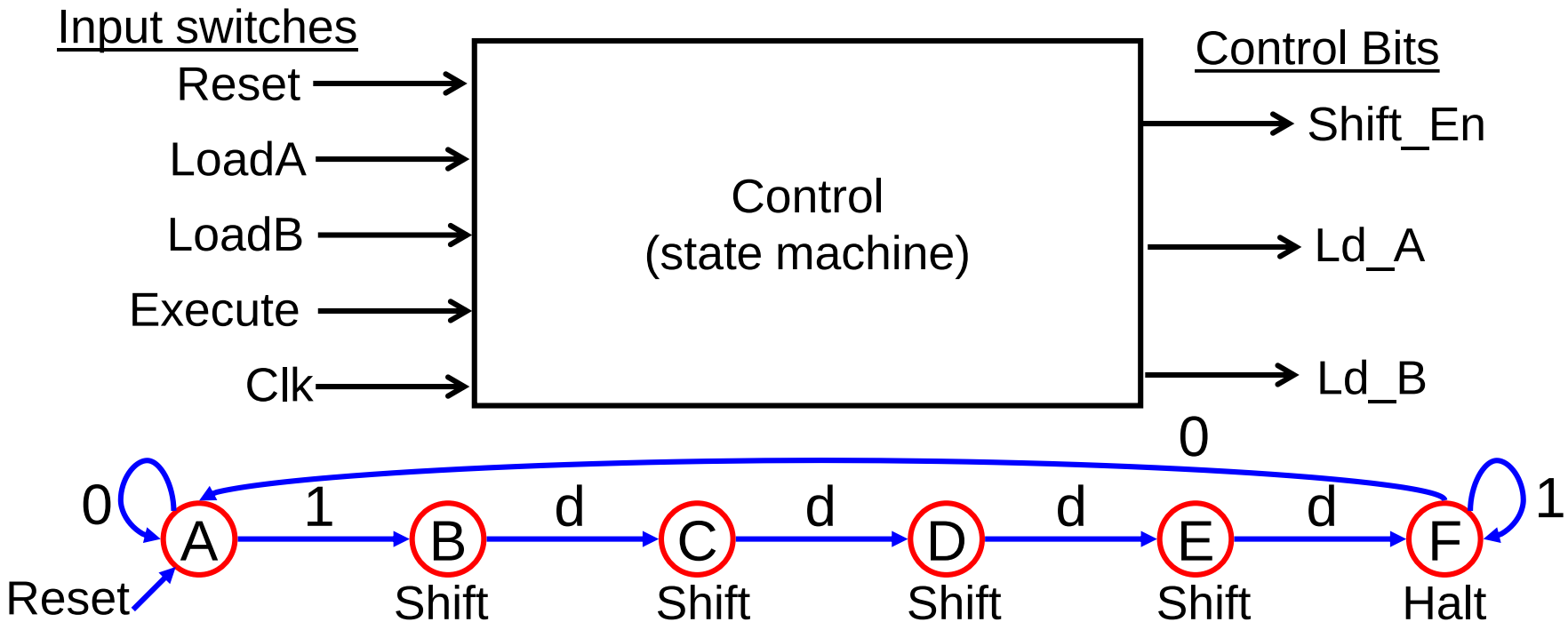
"2 – Always" State Machine

- Two `always` procedures:
  - One for all combinational logic
  - One for all sequential logic



2-Always Example

```
module control ( input         Clk, Reset, LoadA, LoadB,
                               Execute,
                 output logic  Shift_En, Ld_A, Ld_B );
```



2-Always Example

- Declare `enum` for state register and next state
- Declare register behavior (only register is `curr_state`)

```
// declare signals curr_state, next_state of type enum
// with enum values of A, B, ..., F as the state values
enum logic [3:0] {A,B,C,D,E,F,G,H,I,J} curr_state, next_state;

always_ff @ (posedge Clk or posedge Reset )
begin
    if (Reset)  // Asynchronous Reset
        curr_state <= A;   // A is the reset/start state
    else
        curr_state <= next_state;
end
```

2-Always Example

- Combinational procedure has both output and **next state** logic

```
// This is the next state logic
always_comb
begin
    next_state = curr_state; //should never happen

    unique case (curr_state)
        A :  if (Execute)
                next_state = B;
        B :     next_state = C;

        // ..... similarly for D, E, F, ...

        J :  if (~Execute)
                next_state = A;
    endcase
...
```

2-Always Example

- Combinational procedure has both **output** and next state logic

```
case (curr_state)
     A:  begin
         Ld_A = LoadA;
          Ld_B = LoadB;
          Shift_En = 1'b0;
     end
     F:  begin
          Ld_A = 1'b0;
          Ld_B = 1'b0;
          Shift_En = 1'b0;
     end
     default:  begin
          Ld_A = 1'b0;
          Ld_B = 1'b0;
          Shift_En = 1'b1;
    end
endcase
end //end of always_comb
```