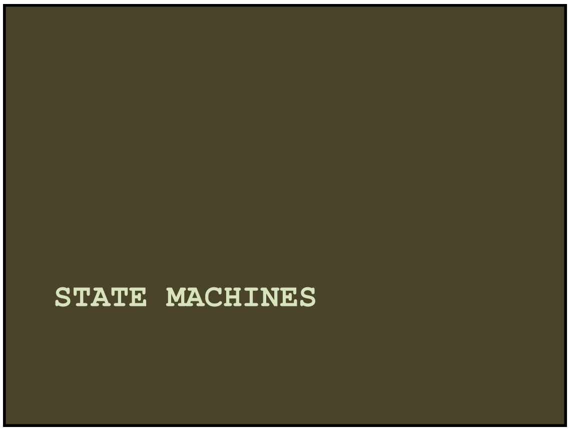
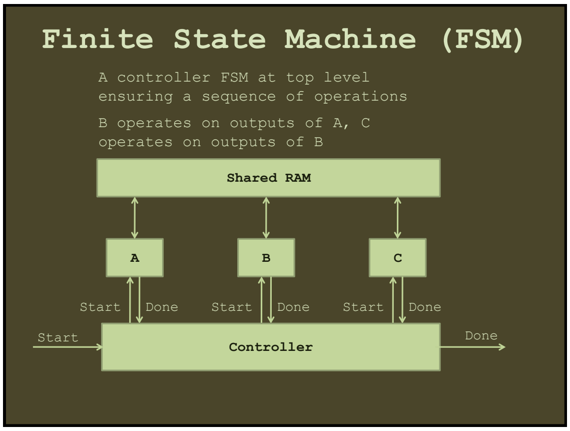


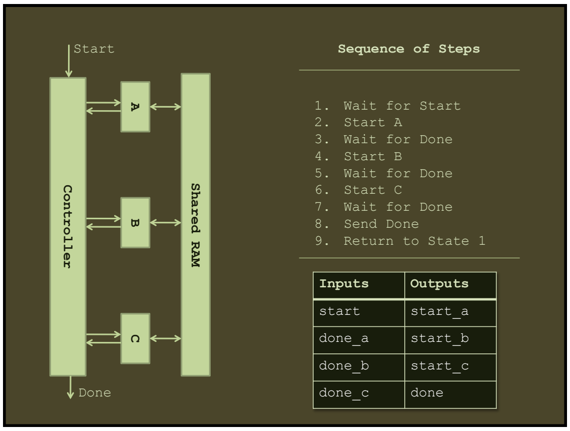
STATE MACHINES



Finite State Machine (FSM)

A controller FSM at top level ensuring a sequence of operations

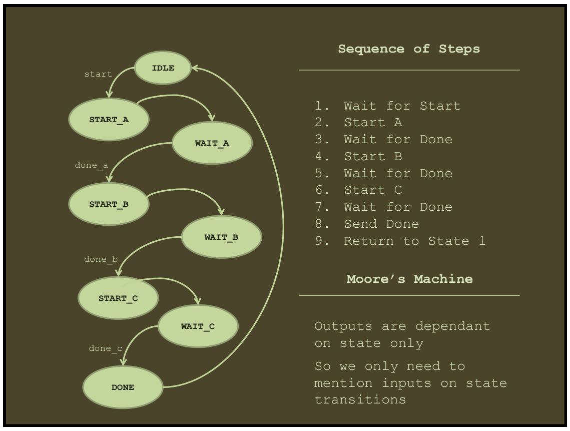
B operates on outputs of A, C operates on outputs of B



Sequence of Steps

1. Wait for Start
2. Start A
3. Wait for Done
4. Start B
5. Wait for Done
6. Start C
7. Wait for Done
8. Send Done
9. Return to State 1

Inputs	Outputs
start	start_a
done_a	start_b
done_b	start_c
done_c	done



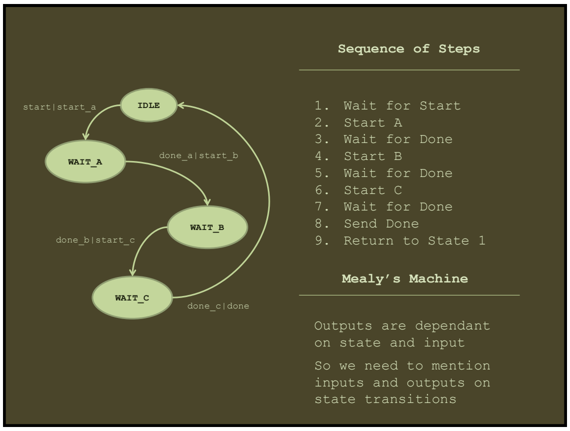
Sequence of Steps

1. Wait for Start
2. Start A
3. Wait for Done
4. Start B
5. Wait for Done
6. Start C
7. Wait for Done
8. Send Done
9. Return to State 1

Moore's Machine

Outputs are dependant on state only

So we only need to mention inputs on state transitions



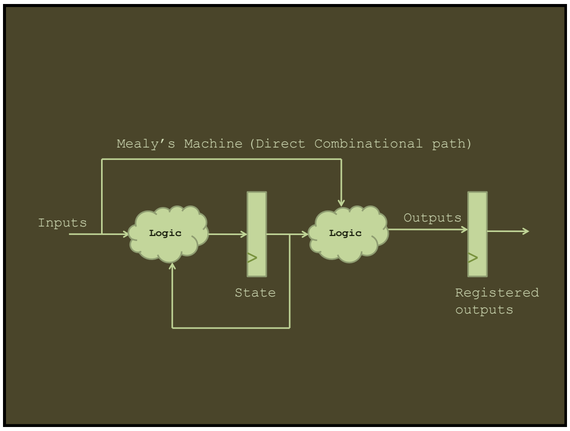
Sequence of Steps

1. Wait for Start
2. Start A
3. Wait for Done
4. Start B
5. Wait for Done
6. Start C
7. Wait for Done
8. Send Done
9. Return to State 1

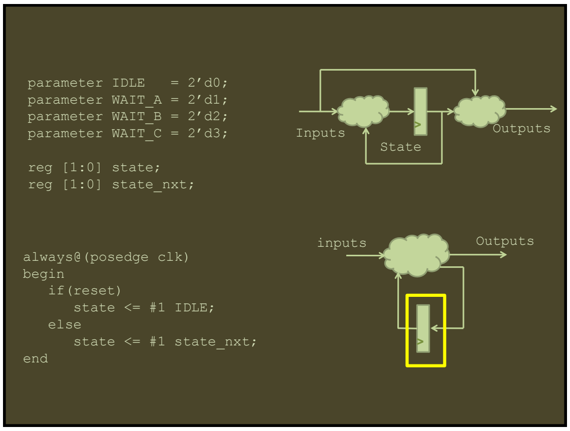
Mealy's Machine

Outputs are dependant on state and input

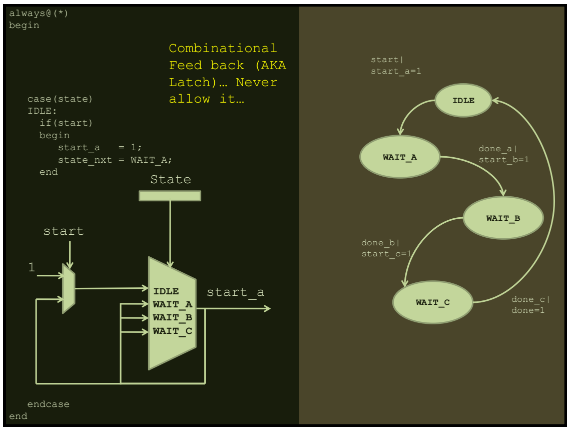
So we need to mention inputs and outputs on state transitions



Mealy's Machine (Direct Combinational path)

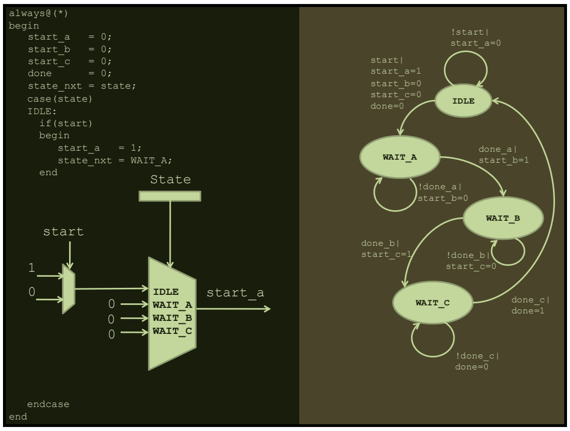


```
parameter IDLE   = 2'd0;
parameter WAIT_A = 2'd1;
parameter WAIT_B = 2'd2;
parameter WAIT_C = 2'd3;

reg [1:0] state;
reg [1:0] state_nxt;


always@(posedge clk)
begin
   if(reset)
      state <= #1 IDLE;
   else
      state <= #1 state_nxt;
end
```

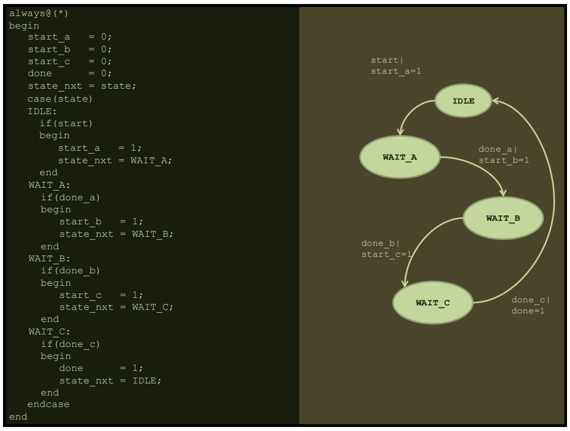


```
always@(*)
begin

   case(state)
   IDLE:
     if(start)
     begin
        start_a   = 1;
        state_nxt = WAIT_A;
     end

   endcase
end
```

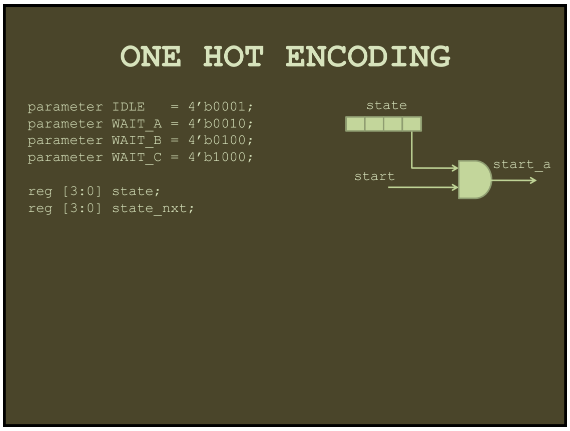
Combinational Feed back (AKA Latch)… Never allow it…



```
always@(*)
begin
   start_a   = 0;
   start_b   = 0;
   start_c   = 0;
   done      = 0;
   state_nxt = state;
   case(state)
   IDLE:
     if(start)
     begin
        start_a   = 1;
        state_nxt = WAIT_A;
     end

   endcase
end
```

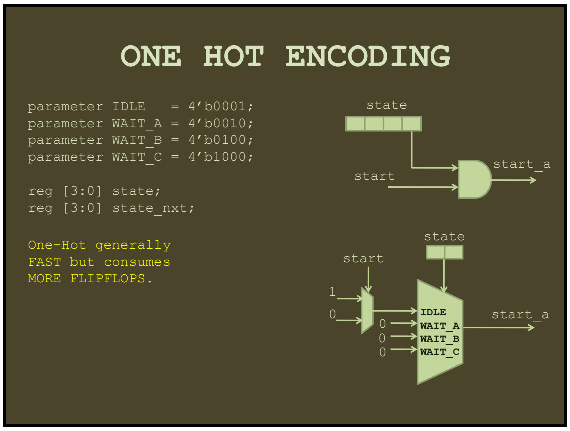


```
always@(*)
begin
   start_a   = 0;
   start_b   = 0;
   start_c   = 0;
   done      = 0;
   state_nxt = state;
   case(state)
   IDLE:
     if(start)
     begin
        start_a   = 1;
        state_nxt = WAIT_A;
     end
   WAIT_A:
     if(done_a)
     begin
        start_b   = 1;
        state_nxt = WAIT_B;
     end
   WAIT_B:
     if(done_b)
     begin
        start_c   = 1;
        state_nxt = WAIT_C;
     end
   WAIT_C:
     if(done_c)
     begin
        done      = 1;
        state_nxt = IDLE;
     end
   endcase
end
```



ONE HOT ENCODING

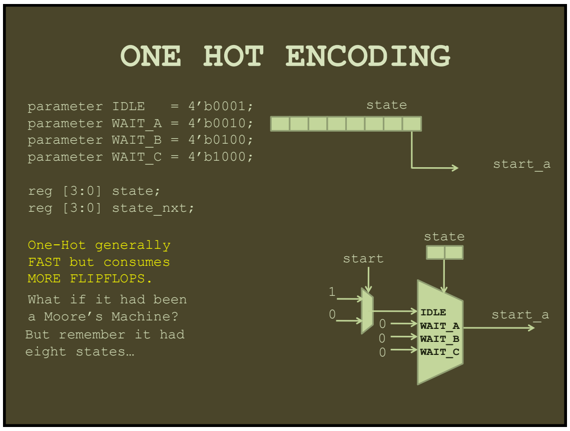
```
parameter IDLE   = 4'b0001;
parameter WAIT_A = 4'b0010;
parameter WAIT_B = 4'b0100;
parameter WAIT_C = 4'b1000;

reg [3:0] state;
reg [3:0] state_nxt;
```



ONE HOT ENCODING

```
parameter IDLE   = 4'b0001;
parameter WAIT_A = 4'b0010;
parameter WAIT_B = 4'b0100;
parameter WAIT_C = 4'b1000;

reg [3:0] state;
reg [3:0] state_nxt;
```

One-Hot generally FAST but consumes MORE FLIPFLOPS.



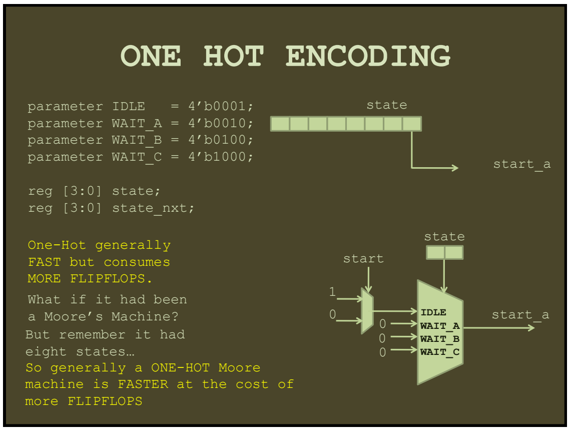
ONE HOT ENCODING

```
parameter IDLE   = 4’b0001;
parameter WAIT_A = 4’b0010;
parameter WAIT_B = 4’b0100;
parameter WAIT_C = 4’b1000;

reg [3:0] state;
reg [3:0] state_nxt;
```

One-Hot generally FAST but consumes MORE FLIPFLOPS.

What if it had been a Moore’s Machine?

But remember it had eight states…



ONE HOT ENCODING

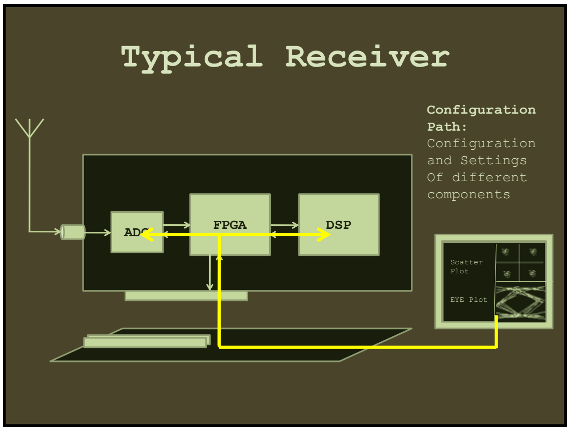
```
parameter IDLE   = 4’b0001;
parameter WAIT_A = 4’b0010;
parameter WAIT_B = 4’b0100;
parameter WAIT_C = 4’b1000;

reg [3:0] state;
reg [3:0] state_nxt;
```

One-Hot generally FAST but consumes MORE FLIPFLOPS.

What if it had been a Moore’s Machine?

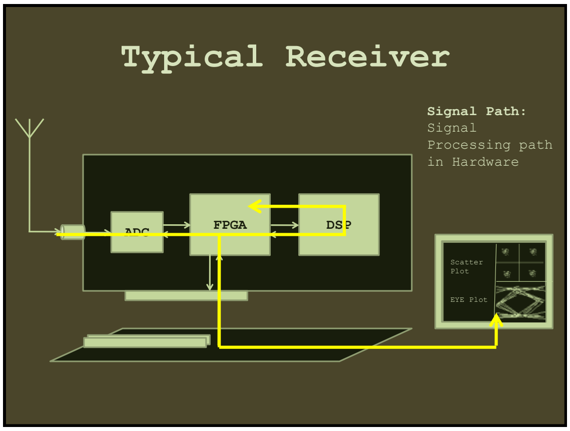
But remember it had eight states…

So generally a ONE-HOT Moore machine is FASTER at the cost of more FLIPFLOPS



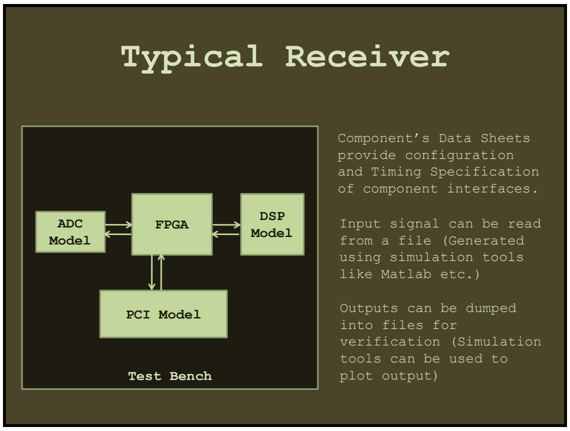
Typical Receiver

Configuration Path:
Configuration and Settings Of different components



Typical Receiver

Signal Path:
Signal Processing path in Hardware

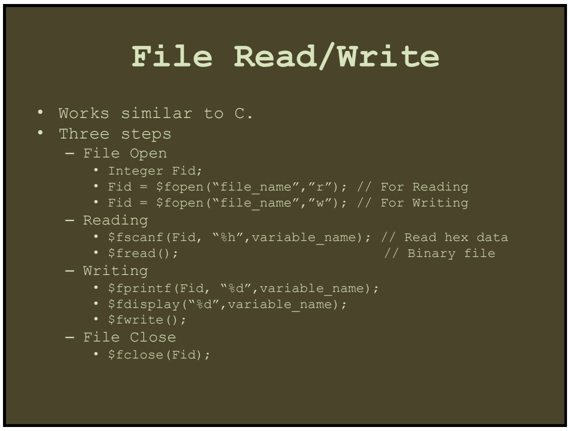


Typical Receiver

Component’s Data Sheets provide configuration and Timing Specification of component interfaces.

Input signal can be read from a file (Generated using simulation tools like Matlab etc.)

Outputs can be dumped into files for verification (Simulation tools can be used to plot output)



File Read/Write

- Works similar to C.
- Three steps
  - File Open
    - Integer Fid;
    - Fid = $fopen(“file_name”,”r”); // For Reading
    - Fid = $fopen(“file_name”,”w”); // For Writing
  - Reading
    - $fscanf(Fid, “%h”,variable_name); // Read hex data
    - $fread();   // Binary file
  - Writing
    - $fprintf(Fid, “%d”,variable_name);
    - $fdisplay(“%d”,variable_name);
    - $fwrite();
  - File Close
    - $fclose(Fid);