CONTROL CIRCUITS
AND
COUNTERS

Control in Digital Systems

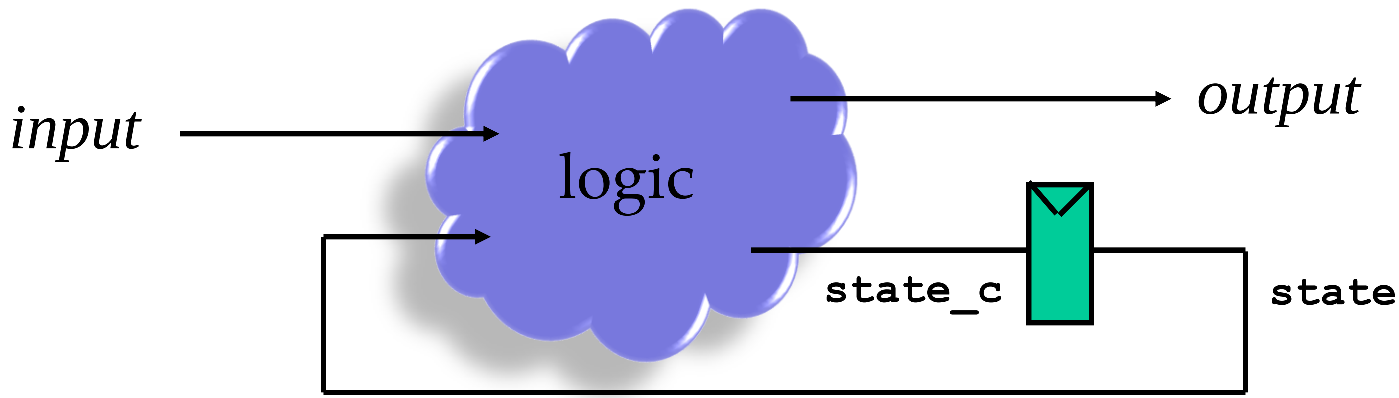
- Three primary components of digital systems
  - Datapath (does the work)
  - **Control (manager, controller)**
  - Memory (storage)

Control in Digital Systems

- Control blocks in chips
  - Typically small amount of HW
  - Typically substantial verilog code
  - Therefore:
    - We typically do not care about small circuit area
    - We typically do not care about fast circuits
      - A possible exception is in cases of "data-dependent control". For example, if arithmetic is required to make control decisions e.g., "change states if *sum* < *phase*"
  - Verilog code can be complex
    - Many opportunities for bugs
    - May be impossible to test all states and transitions
    - Often the focus of testing by Verification Engineers

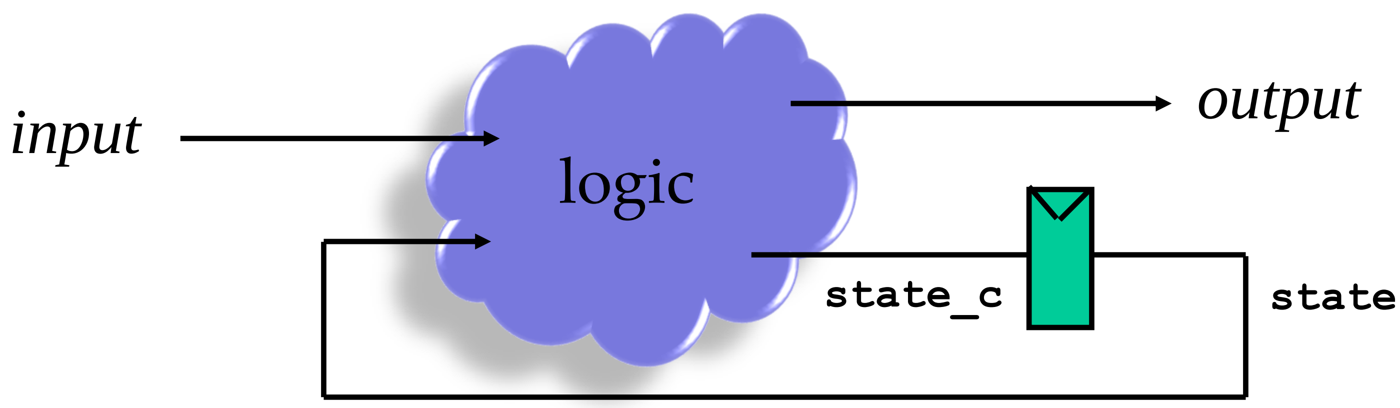
Sequential Logic

- *Combinational circuits'* outputs are a function of the circuit's inputs and a time delay
- *Sequential circuits'* outputs are a function of the circuit's inputs, previous circuit state, and a time delay



Control with Finite State Machines

- Of course we can design all standard finite state machines we learned in basic logic design classes
  - Moore type FSM
    - outputs depend only on the present state (**`state`** below)
  - Mealy type FSM
    - outputs depend on the present state and the inputs



Writing Verilog for State Machines

- Design process
  - **Think**
  - Determine all necessary registers (state, count, etc.)
  - **Think**
  - Draw state diagrams if helpful
  - Draw block diagrams if helpful
  - Draw timing diagrams
  - Pick descriptive signal names
  - **Think**

  - Then…
    - Write verilog
    - Test it

#1 Design Goal for Controllers: Clarity

- Clear code → bugs will be less likely
- It is even more important to use good signal naming conventions in control logic than with other digital circuits
  - Ex: *state_c* → *state*
- Reduce the amount of state if possible (clarity)
  - Ex: It may be better to have one global state machine instead of two separate ones
- Increase the amount of state if helpful (clarity)
  - Ex: It may be better to have two separate global state machines instead of a single global one
  - Ex: Instantiate separate counters to count independent events

I. Counters

- Typically the output is equal to the *state* of the counter
- In normal operation, the *next state* is the *present state* plus or minus a fixed number
  - Or reset to an initial state
  - Or hold the value to reduce power when idle
- Fixed simple circular counter
  - Example: 0 → 1 → 2 → 3
  - This would be a good choice if you want something to happen once every four cycles, as an example

I.a. Counters—Count Up

- Count Up Counter
  - Example: 0 → 1 → 2 → ... → 23

reset

```
if (count == 8'd023) begin
    ...do something...
```

Is it better to count up 0→23 or down 23→0? Or no difference?

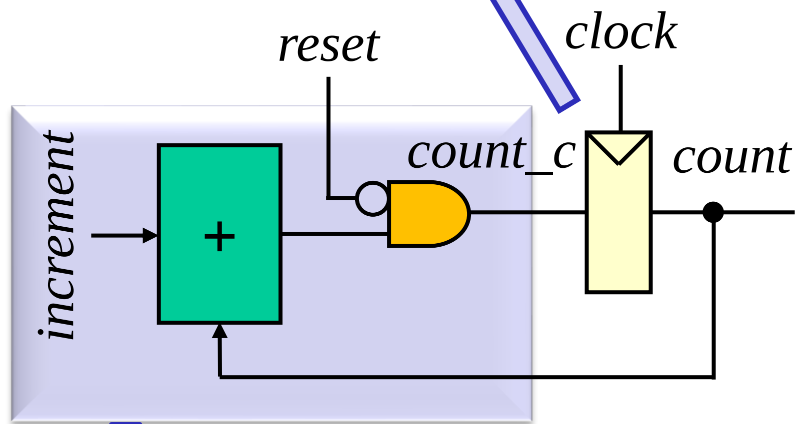
- The reset hardware is simple, probably using reset-able FFs
- If the counter's ending count is programmable, detection of the finishing condition requires a more general comparator

I.a. Counters—Count Up

- Example: 0 → 1 → 2 → … → 23
- When count gets to 23, hold that value to reduce power dissipation
- Fixed constants: increment = 1, stop at 23 base 10

```
// Example Code version #1
reg [7:0] count;    // real FF register
reg [7:0] count_c;  // combinational logic

always @(count or reset) begin
   // default section is good practice
   // Try hold for default
   count_c = count;   // hold

   if (count != 8'd023) begin
      count_c = count + 8'h01;   // increment
   end

   if (reset == 1'b1) begin // high priority
      count_c = 8'b0000_0000;
   end
end
```

```
always @(posedge clock) begin
   count  <= #1 count_c;
end
```



I.a. Counters—Count Up Example Code Version #2

```
// Example Code version #2
reg [7:0] count;   // real FF register
reg [7:0] count_c; // combinational logic

always @(count or reset) begin
   // default increments in this example
   count_c = count + 8'h01;  // increment

   if (count == 8'd023) begin
      count_c = count;  // hold
   end

   if (reset == 1'b1) begin // high priority
      count_c = 8'b0000_0000;
   end
end
```

- What happens if count is 24 – 255?
  - What do you want to happen?
- Notice that inputs to the "next state logic" will be the registered **`count`**, not **`count_c`**

I.b. Counters—Count Down

- Count Down Counter
  - Example: 17 → 16 → 15 → ... → 0

    preset/reset

```
if (count == 8'd000) begin
    ...do something...
```

- The reset hardware is more complex especially if multiple starting values are needed
- Detection of the finishing condition is requires very simple hardware, conceptually a single NOR gate

I.b. Counters—Count Down

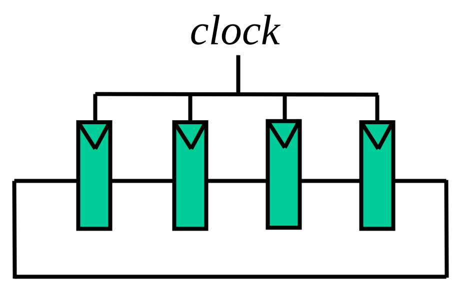
- Example: 17 → 16 → 15 → ... → 0
- When count gets to 0, hold that value to reduce power dissipation, and assert *done* output signal
- Fixed constants: increment = –1, stop at 0

```
always @(posedge clock) begin
   count  <= #1 count_c;
end
```

```
reg [7:0] count;    // real FF register
reg [7:0] count_c; // combinational logic
reg       done;    // output ==1 when done

always @(count or reset) begin
   // default section is good practice
   // Let default be decrement
   count_c = count - 8'h01;   // decrement
   done    = 1'b0;

   if (count == 8'h00) begin
      count_c = count; // could also be  = 8'h00
      done    = 1'b1;
   end

   if (reset == 1'b1) begin // highest priority
      count_c = 8'd017;
   end
end
```

I.c. One-Hot Ring Counter

- One-hot encoding
- One flip-flop for each state
- Requires circuits to initialize in a one-hot configuration
- Minimal hardware, no adders required!
- Zero-time state decode (i.e., each state is fully decoded by a FF output)



1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1
1	0	0	0