Datapath Sequential Logic

Dr. Eric Becker

CS 4341

Fall 2017

Datapath? What? Huh? Durrrrr?

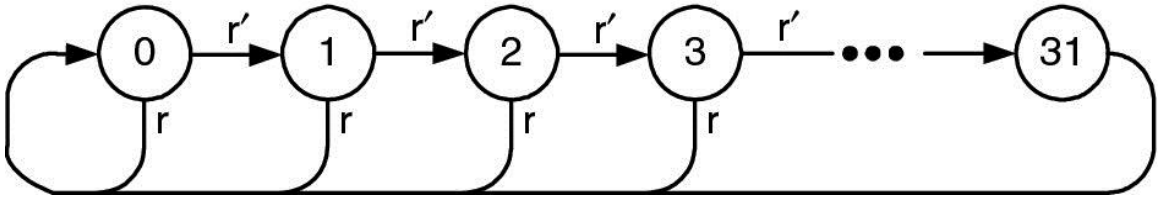
- The next-state function of the FSM can be represented as an expression rather than a table
- This expression is the *datapath*
- Then, in the circuit, the next state comes from:
  - Arithmetic circuits
  - Multiplexers
  - Combinational Logic
  - Sound familiar?

Counters

- Simple
- Up/Down/Load
- A timer
- A definition of a Saturated Counter

Simpler Counter

- A counter:
  - When r is true, go back to 0.
  - When r is not true, go to the next state, which increments by one
- The datapath would be:
  - next state= current state + 1

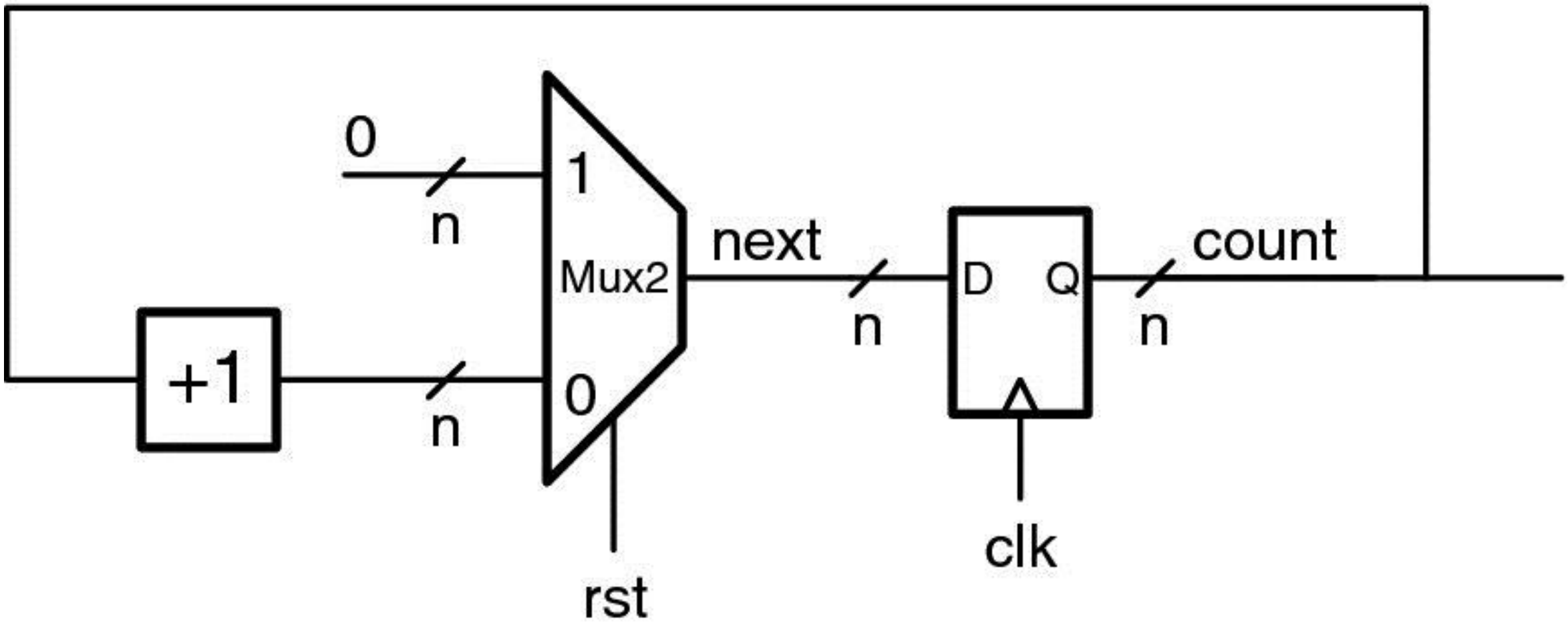


Simpler Counter

- Psuedo Code:
  - If reset is true,
    - next state is zero
  - Else
    - next state is count + 1
- Verilog:
  - assign next = rst ? 0: count+1;

```
module Counter(clk, rst, out) ;
  parameter n=5 ;//Count of 0 to 31
  input rst, clk ; // reset and clock
  output [n-1:0] out ;

  wire   [n-1:0] next = rst? 0 : out+1 ;

  DFF #(n) count(clk, next, out) ;
endmodule
```

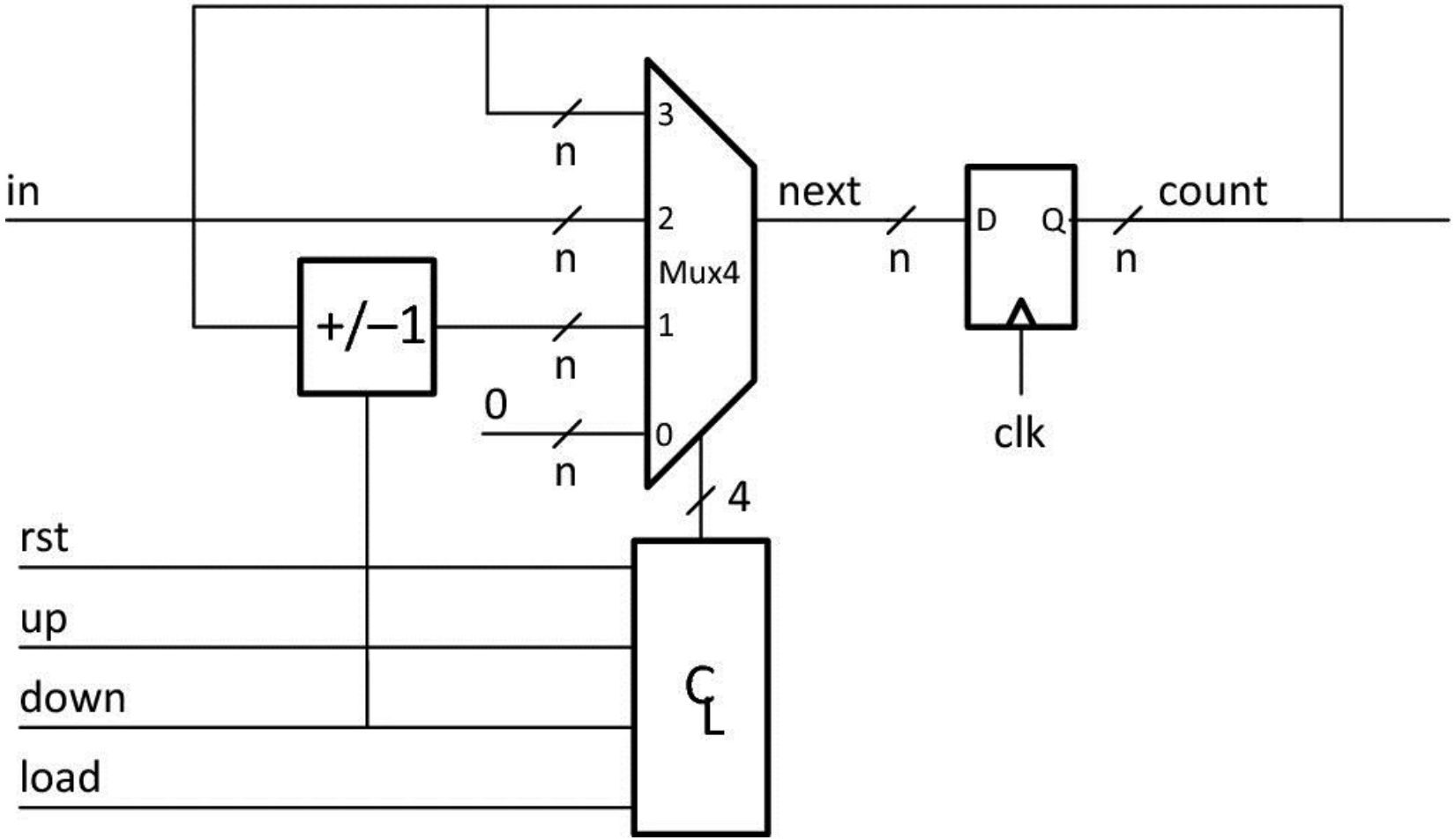
Simple Counter



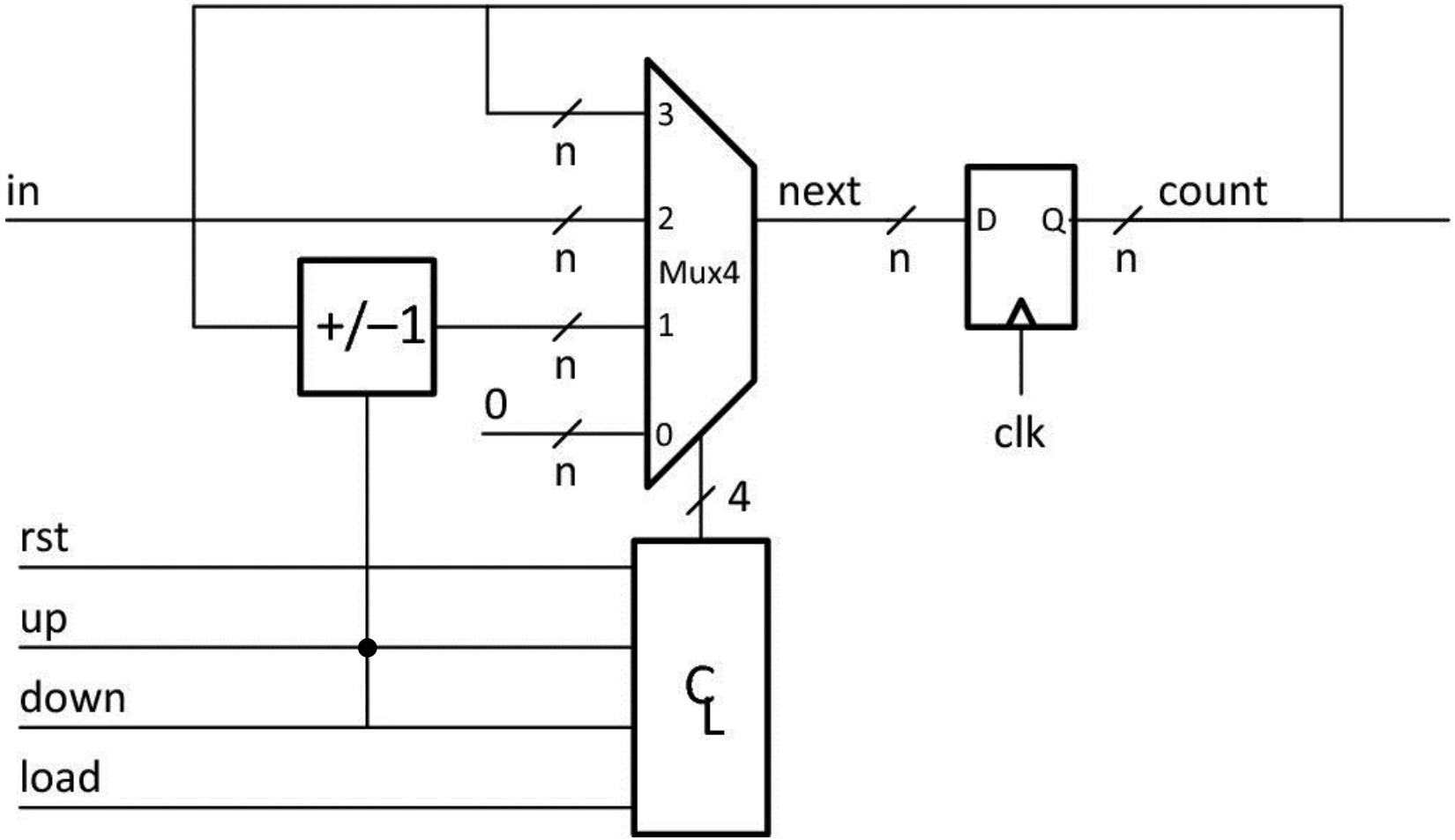
Up/Down/Load Counter

- An Up/Down/Load counter has 5 commands:
  - Reset-The next state is 0 ,(1000)
  - Up-The next state is count+1,(0100)
  - Down-The next state is count-1,(0010)
  - Load-The next input sets the current state(0001)
  - Do Nothing-Count stays the same (0000)

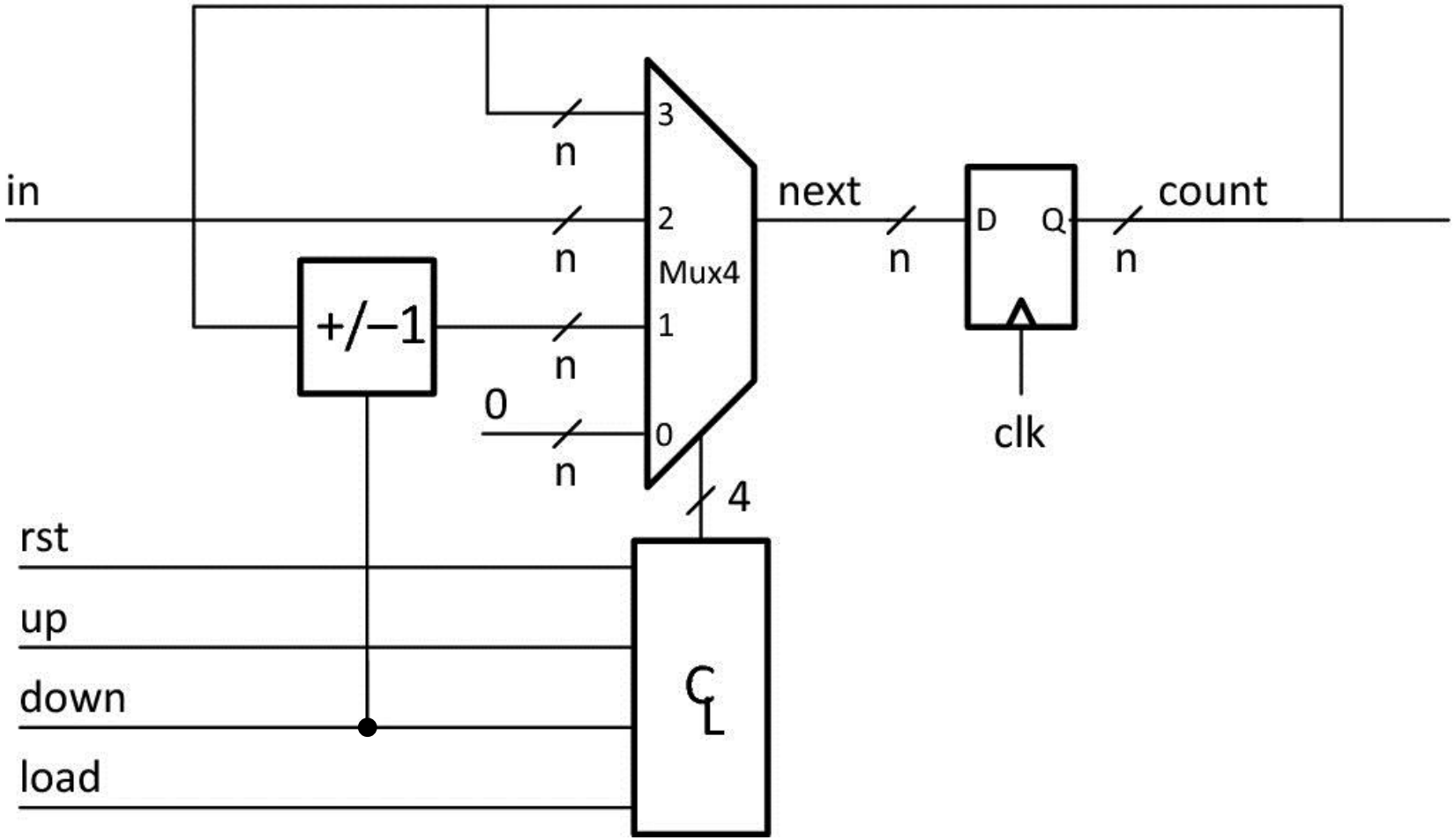
Up/Down/Load Counter



An Aside: How many bits needed?



An Aside: How many bits needed?



Verilog Code

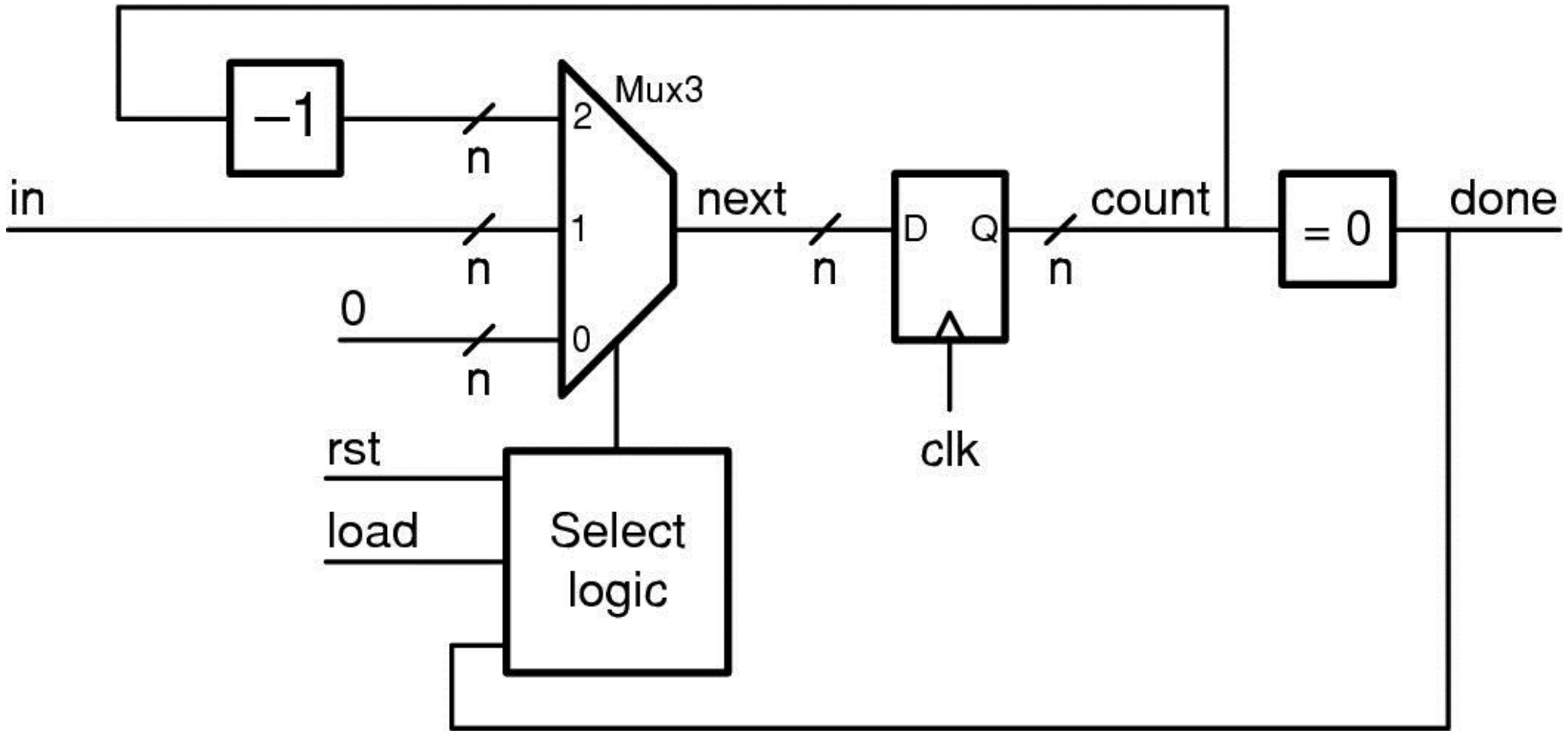
```
module UDL_Count3(clk, rst, up, down, load, in, out) ;
  parameter n = 4 ;
  input clk, rst, up, down, load ;
  input [n-1:0] in ;
  output [n-1:0] out ;
  wire [n-1:0] next, outpm1 ;

  DFF #(n) count(clk, next, out) ;
  assign outpm1 = out + {{n-1{down}},1'b1} ;//IF DOWN
  Mux4 #(n) mux(out, in, outpm1,
                {n{1'b0}},//A ZERO
                {(~rst & ~up & ~down & ~load),//ALL OFF
                 (~rst & load),//LOAD
                 (~rst & (up | down)),//UP OR DOWN
                   rst}, //RESET
                  next) ;//OUTPUT
endmodule
```

Time for Verilog

Timer

- A timer is like the Traffic Light
- The mark is set, and the finite state machine counts down
- When machine reaches zero
- Machine stopes decrementing

Timer


−1
n
Mux3
2
in
1
n
next
n
D
Q
n
count
= 0
done
0
0
n
rst
clk
load
Select
logic

Timer

```
module Timer(clk, rst, load, in, done) ;
  parameter n=4 ;
  input clk, rst, load ;
  input [n-1:0] in ;
  output        done ;
  wire [n-1:0] count, next_count ;
  wire done ;

  DFF #(n) cnt(clk, next_count, count) ;
  Mux3 #(n) mux(count-1'b1, //Select 2, Counting Down
                        in, //Select 1
                 {n{1'b0}}, //Select 0
                {~rst & ~load & ~done, //Choice 2//Count Down
                 load & ~rst,          //Choice 1//Load Value
                 rst | (done & ~load)},//Choice 0//Reset
                next_count) ;//Count
  assign done = ~(|count) ;//Are Done
endmodule
```

Saturated Counter?

- Let’s look at the description.
  - The counter will not decrement when count equal to zero
  - The counter will not increment when count equals count max.
- From the counter:
  - Reset, up, down, load…sounds like the UDL counter
  - Needs to have an additional load-max count value.
  - Two options: Larger multiplexer or more than one multiplexer.
  - Also has to have an additional state!
- From Saturation
  - Has to know it cannot go below zero. Zero is a constant. No problem
  - Has to know it cannot go above max count.
  - Needs to have a register to store the max_count.
- Therefore:
  - The increment/decrement needs to know if it is up or down (down wire)
  - The increment/decrement needs to know it cannot go below zero
  - The increment/decrement has to know the current count
  - The increment/decrement has to know the max_count

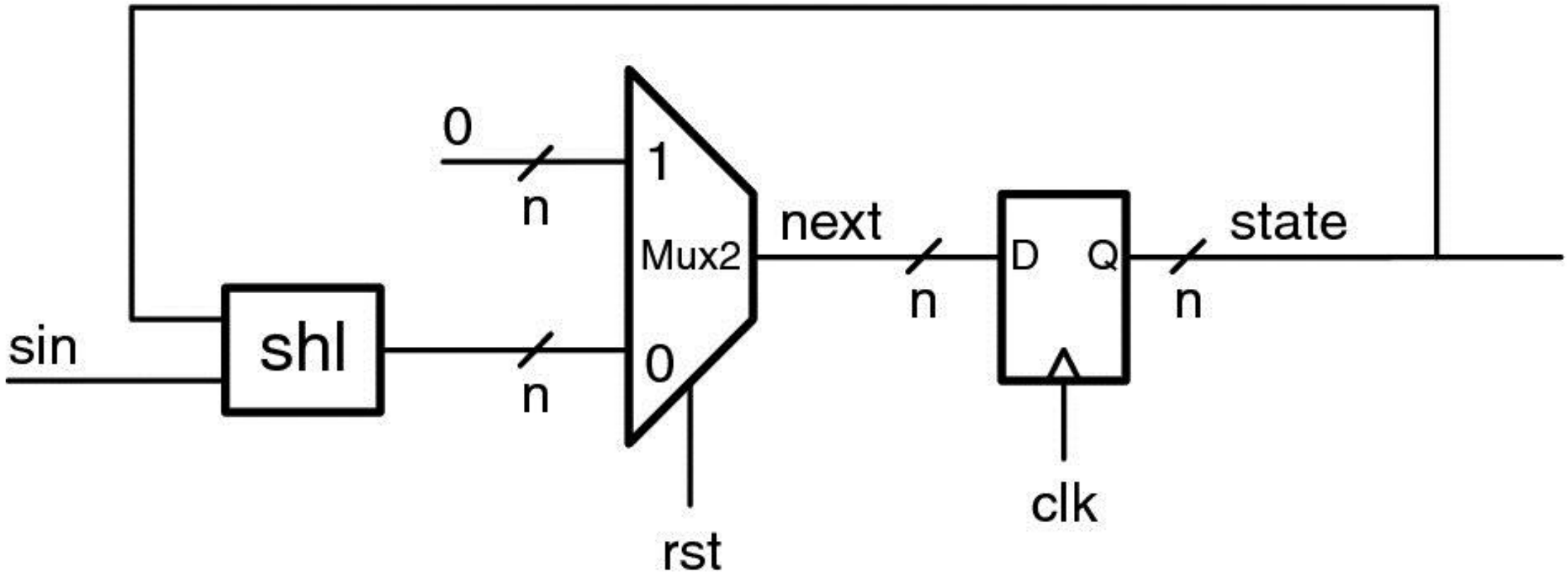
Shift Registers

- Simple Shift
- Left/Right/Load Shift
- Universal Shifter/Counter

Simple Shift Register

- Unless reset,
- the shift register shifts one bit to the left
- The new bit is the input
- Concatenate the rightmost n-1 bits with the input.

Simple Shift Register



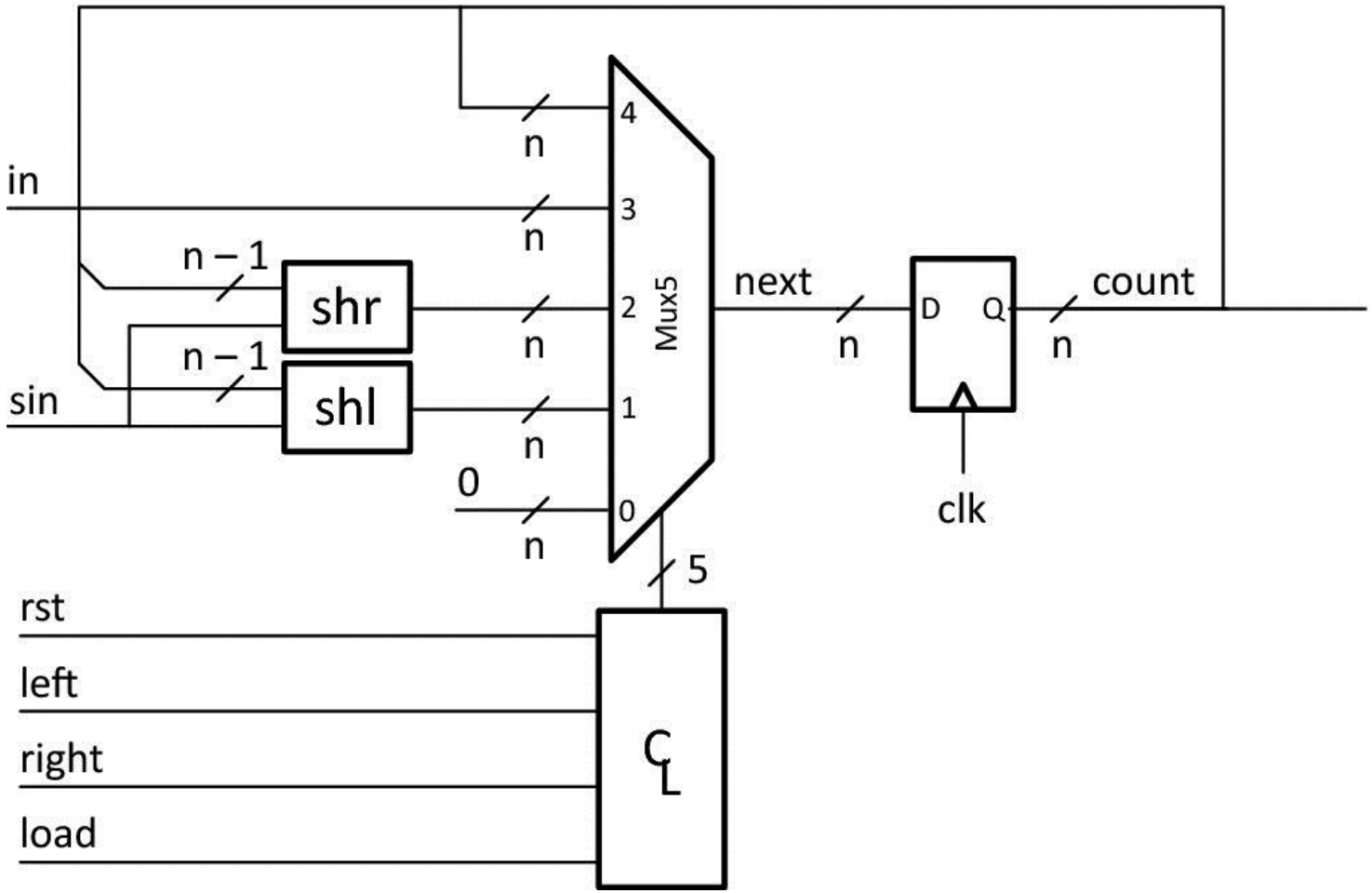
Verilog

```
module Shift_Register1(clk, rst, sin, out) ;
  parameter n = 4 ;
  input clk, rst, sin ;
  output [n-1:0] out ;

  //If Reset Go to 0, else Concatenate LSB and Input
  wire [n-1:0] next = rst ? {n{1'b0}} : {out[n-2:0],sin} ;

  DFF #(n) cnt(clk, next, out) ;
endmodule
```

Left/Right/Load Shift Register

- Instead of up/down/load
- Left/Right/Load
- Sounds familiar?
- On shift left, concatenate trimmed LSB with Input
- Verilog:{out[n-2:0],sin}
- On Shift Right, concatenate Input with trimmed MSB
- Verilog: {sin ,out[n-1:1],}

Left/Right/Load Shift Register



Universal Shifter/Counter

- In theory, this can be all one module
- A large multiplexer could handle:
  - Reset, Left, Right, Up, Down, Done, Load
  - Use an arbiter to determine rank of commands.
  - Use a clock and DFF to count the states
- Is it wasteful?
  - Perhaps. Depends on need.
  - Such a module can become another module by simply setting the relative commands to 0.

Universal Shifter/Counter Comments

```
//-------------------------------------------------------------
// Universal Shifter/Counter
// inputs take priority in order listed
// rst - resets state to zero
// left - shifts state to the left, sin fills LSB
// right - shifts state to the right, sin fills MSB
// up - increments state
// down - decrements state - will not decrement through zero.
// load - load from in
//
// Output done indicates when state is all zeros.
//-------------------------------------------------------------
```

Universal Shifter/Counter Variables

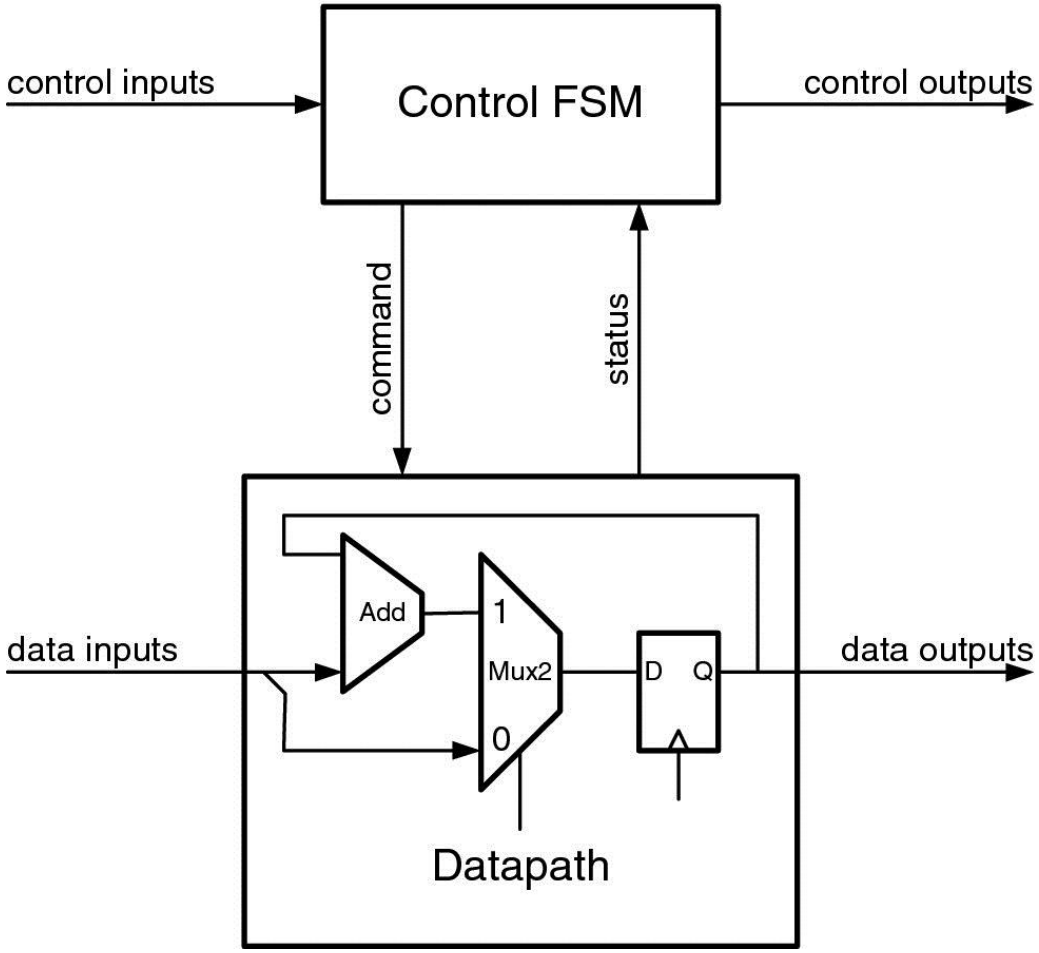
```
module UnivShCnt(clk, rst, left, right, up, down, load, sin, in, out, done)
;
  parameter n = 4 ;
  input clk, rst, left, right, up, down, load, sin ;
  input [n-1:0] in ;
  output [n-1:0] out ;
  output done ;
  wire [6:0] sel ; // multiplexer select, formed by arbiting the commands
  wire [n-1:0] next, outpm1 ;
```

The Program

```
  assign outpm1 = out + {{n-1{down}},1'b1} ; // COUNTING
  DFF #(n) cnt(clk, next, out) ;
  RArb #(7) arb({rst, left, right, up, down & ~done, load, 1'b1}, sel) ;
  Mux7 #(n) mux({n{1'b0}}, {out[n-2:0],sin}, {sin,out[n-1:1]},
                outpm1, outpm1, in, out, sel, next);
//SHIFTING IN LINE ON MULTIPLEXER. DR. BECKER THINKS IT SLOPPY.
  assign   done = ~(|out) ;
endmodule // UnivShCnt
```

Control and Data Partitioning

- Like most computer projects, systems have two key areas
  - Data, the information or sequence
  - Functionality, the control or commands
- Data Path: Count, Timing, Shifted Binary Number
- Control FSM: Find the next instruction, manipulate data

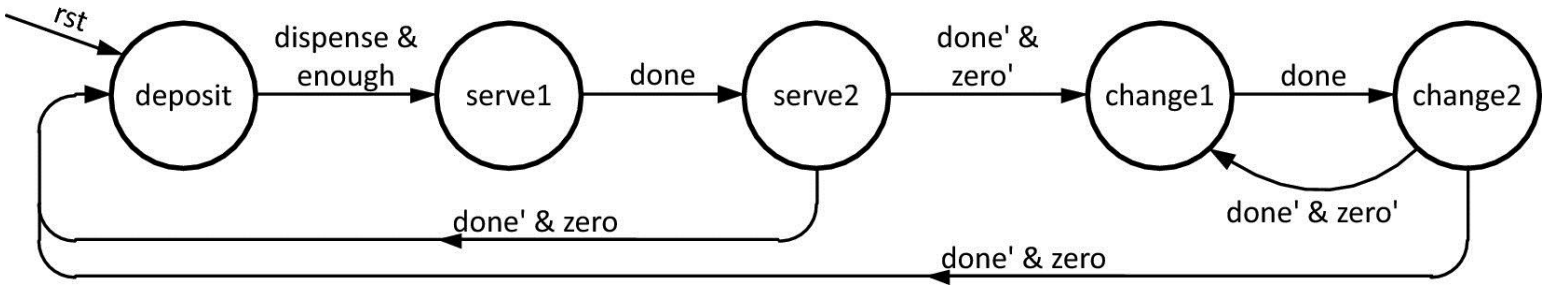


Control and Data Partitioning

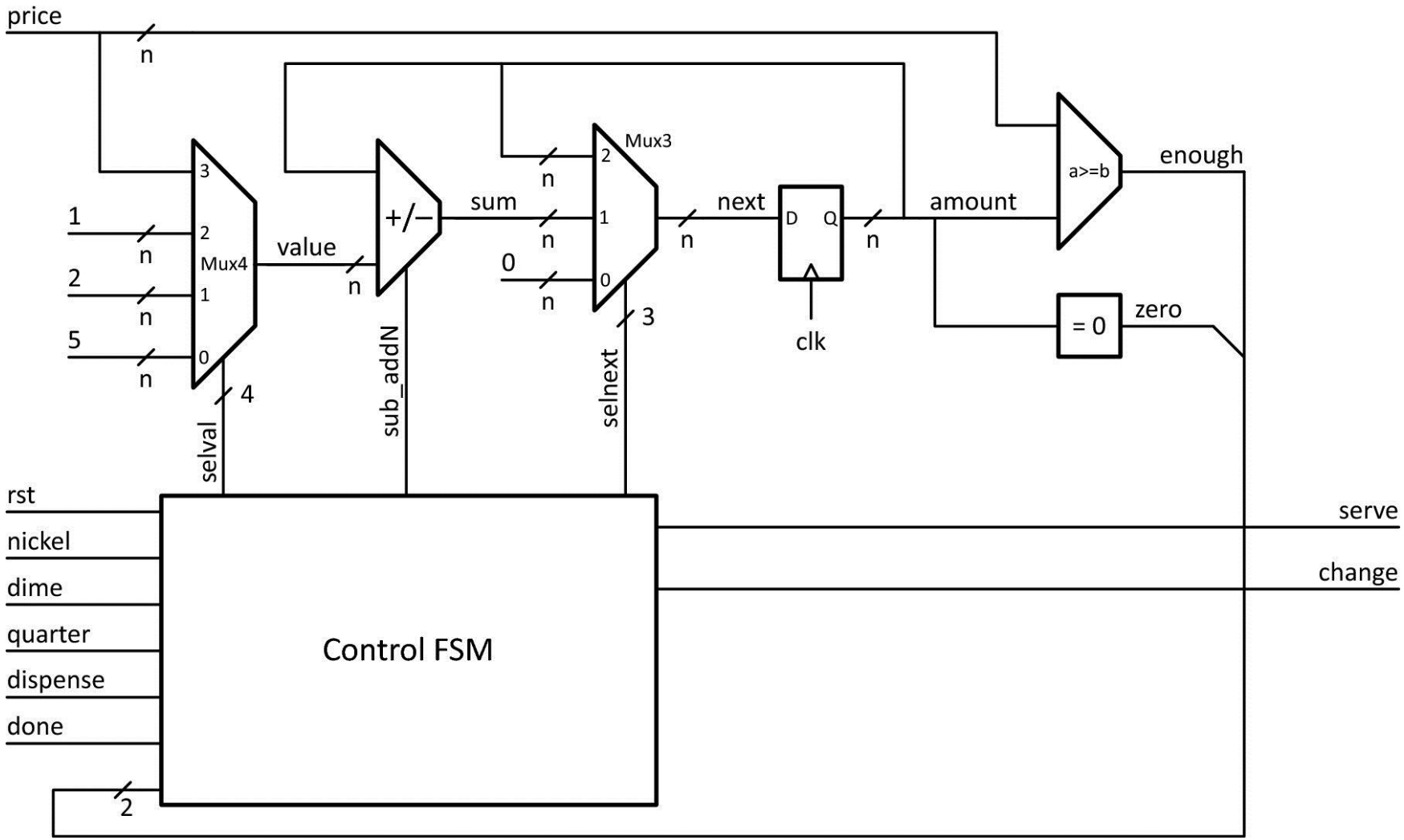
- The previous examples
  - have a finite state machine to control the data (count, time, shift),
  - and combinational logic to control the circuit (Mulitplexer and lines)

- But, really, another state machine should manipulate the data.

The Vending Machine



The Vending Machine



The Vending Machine

- *Commands*
- Rst-Reset
- Nickel-add 1
- Dime-Add 2
- Quarter-Add 5
- Dispense-Get Drink
- Done-No more

The Vending Machine

- *Wires*
  - Price-Price of a drink
  - Value-Current Coin
  - Sum-Current Total
  - Next-Next State
  - Amount-The Current State, the total amount
  - Enough-Arbiter says Amount greater than Price
  - Zero-If Amount is equal to Zero

The Vending Machine

- *Outputs*
- Serve-Serve the Drink?
- Change-Does Change exist?

The Vending Machine

- What changes *AMOUNT*
- Reset: Amount=0
- Nickel: Add 1 to amount
- Dime: Add 2 amount
- Quarter: Add 5 amount
- Dispense: Amount=Amount-Price
- Change: Return Nickels (decrement Amount by 1)
- Otherwise: Amount stays the same.