
Digital Design: A Systems Approach

Lecture 7: Data Path State Machines

Readings

- L7: Chapter 16
- L8: Chapter 17

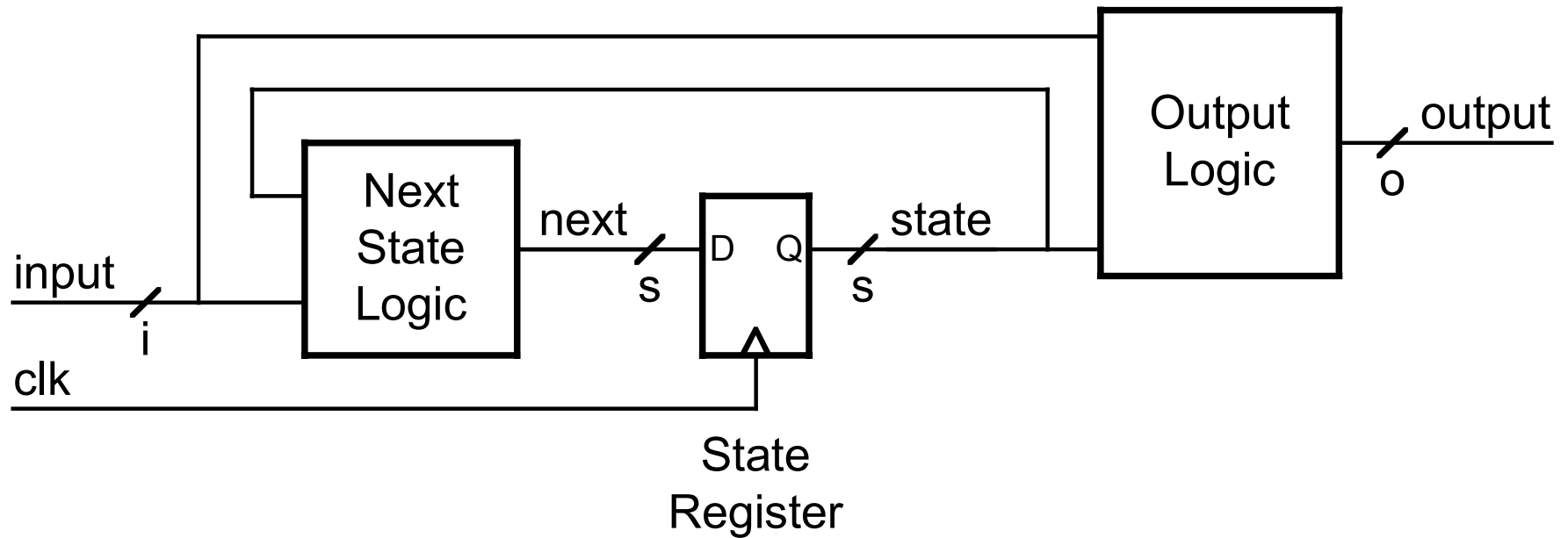
Review

- Lecture 1 – Digital abstraction
- Lecture 2 – Combinational logic design
- Lecture 3 – Combinational building blocks
- Lecture 4 – Numbers and arithmetic
- Lecture 5 – Quiz 1 Review

- Lecture 6 – Sequential Logic, FSMs
- Lecture 7 – Datapath FSMs

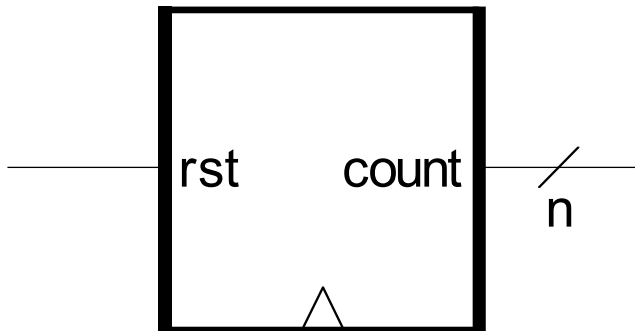
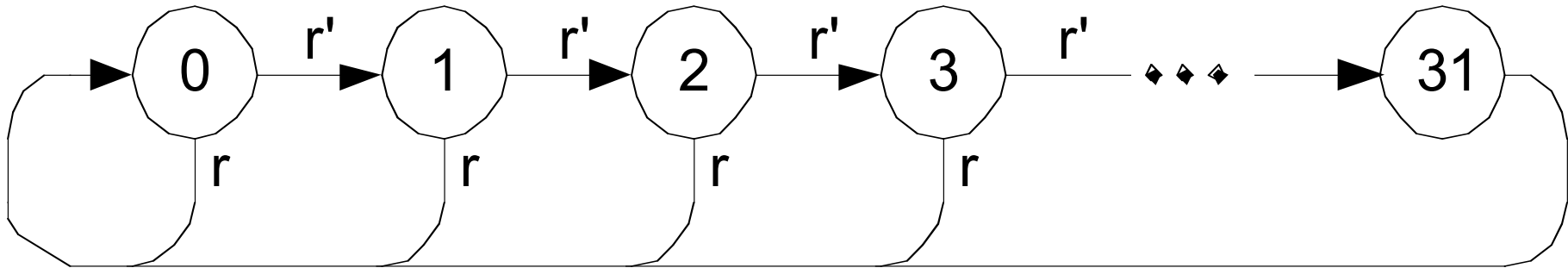
- Lecture 8 - Factoring FSMs
- Lecture 9 - Microcode

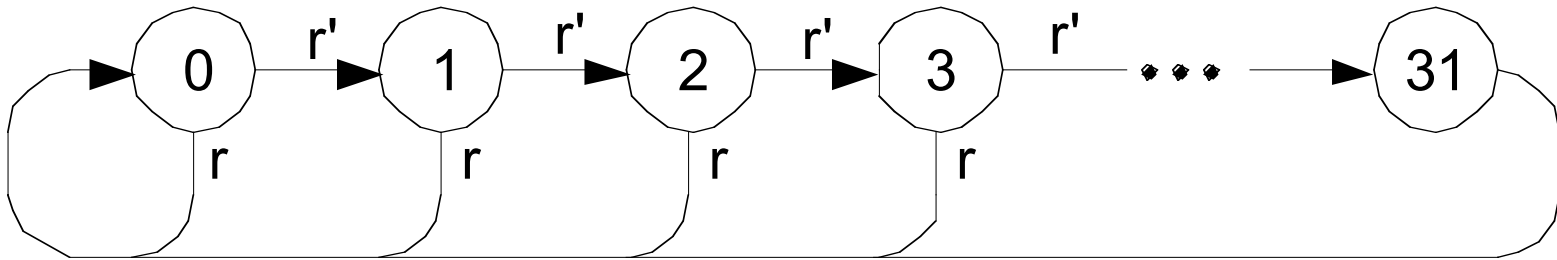
An FSM is a state register and two functions



A Simple Counter

- Suppose you want to build an FSM with the following state diagram





```

module Counter1(clk,rst,out) ;
  input rst, clk ; // reset and clock
  output [4:0] out ;
  reg      [4:0] next ;

```

```

  DFF #(5) count(clk, next, out) ;

```

```

  always@(rst, out) begin
    casex({rst,out})
      6'b1xxxxx: next = 0 ;
      6'd0: next = 1 ;
      6'd1: next = 2 ;
      6'd2: next = 3 ;
      6'd3: next = 4 ;
      6'd4: next = 5 ;
      6'd5: next = 6 ;
      6'd6: next = 7 ;

```

```

    ...

```

```

      6'd30: next = 31 ;

```

```

      6'd31: next = 0 ;

```

```

      default: next = 0 ;

```

```

    endcase

```

```

  end

```

```

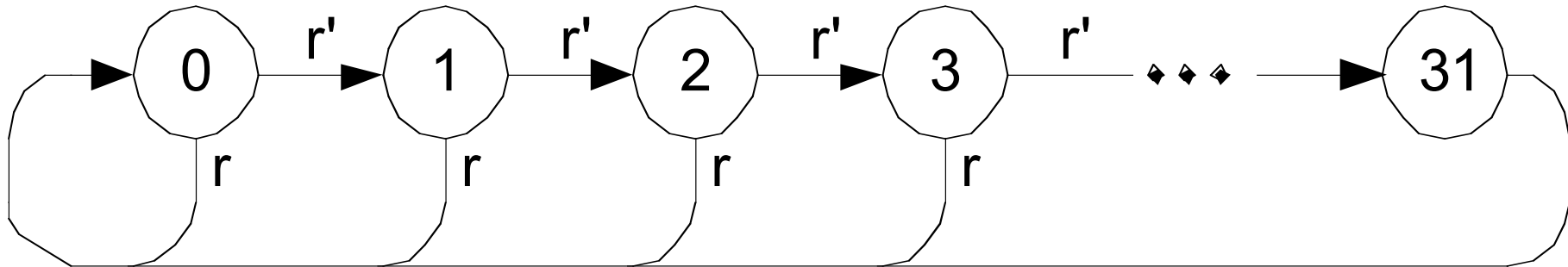
endmodule

```

(c) 2005-2012 W. J. Dally

State	Next State	
	~rst	rst
0	1	0
1	2	0
2	3	0
.		
.		
.		
30	31	0
31	0	0

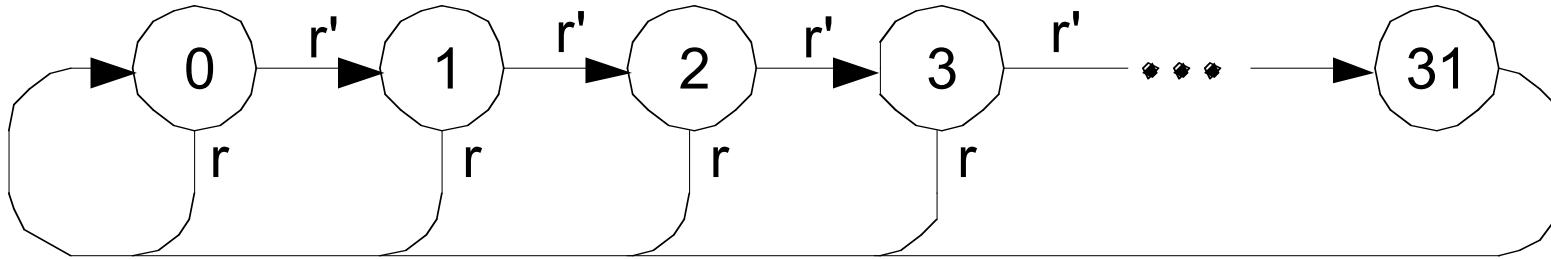
Datapath Implementation



State	Next State	
	$\sim\text{rst}$	rst
0	1	0
1	2	0
2	3	0
⋮		
30	31	0
31	0	0

- You can describe the next-state function with a table
- However, it can more compactly be described by an expression:
$$\text{next} = r ? 0 : \text{state} + 1 ;$$
- This “counter” is an example of a sequential “datapath” – a sequential circuit where the next state function is generated by an expression rather than a table.

Verilog description



```
module Counter(clk, rst, count) ;  
    parameter n=5 ;  
    input rst, clk ; // reset and clock  
    output [n-1:0] count ;  
  
    wire    [n-1:0] next = rst? 0 : count + 1 ;  
  
    DFF #(n) count(clk, next, count) ;  
endmodule
```


Make Table Symbolic

State	Next State	
	$\sim\text{rst}$	rst
0	1	0
1	2	0
2	3	0
· · ·		
30	31	0
31	0	0

State	Next State	
	$\sim\text{rst}$	rst
a	$a+1$	0

Alternate description (symbolic table)

```
module Counter1(clk,rst,out) ;
    input rst, clk ; // reset and clock
    output [4:0] out ;
    reg      [4:0] next ;

    DFF #(5) count(clk, next, out) ;

    always@(rst, out) begin
        casex({rst,out})
            6'b1xxxxx: next = 0 ;
            6'd0: next = 1 ;
            6'd1: next = 2 ;
            6'd2: next = 3 ;
            6'd3: next = 4 ;
            6'd4: next = 5 ;
            6'd5: next = 6 ;
            6'd6: next = 7 ;
            ...
            6'd30: next = 31 ;
            6'd31: next = 0 ;
            default: next = 0 ;
        endcase
    end
endmodule
```

```
module Counter1(clk,rst,out) ;
    input rst, clk ; // reset and clock
    output [4:0] out ;
    reg      [4:0] next ;

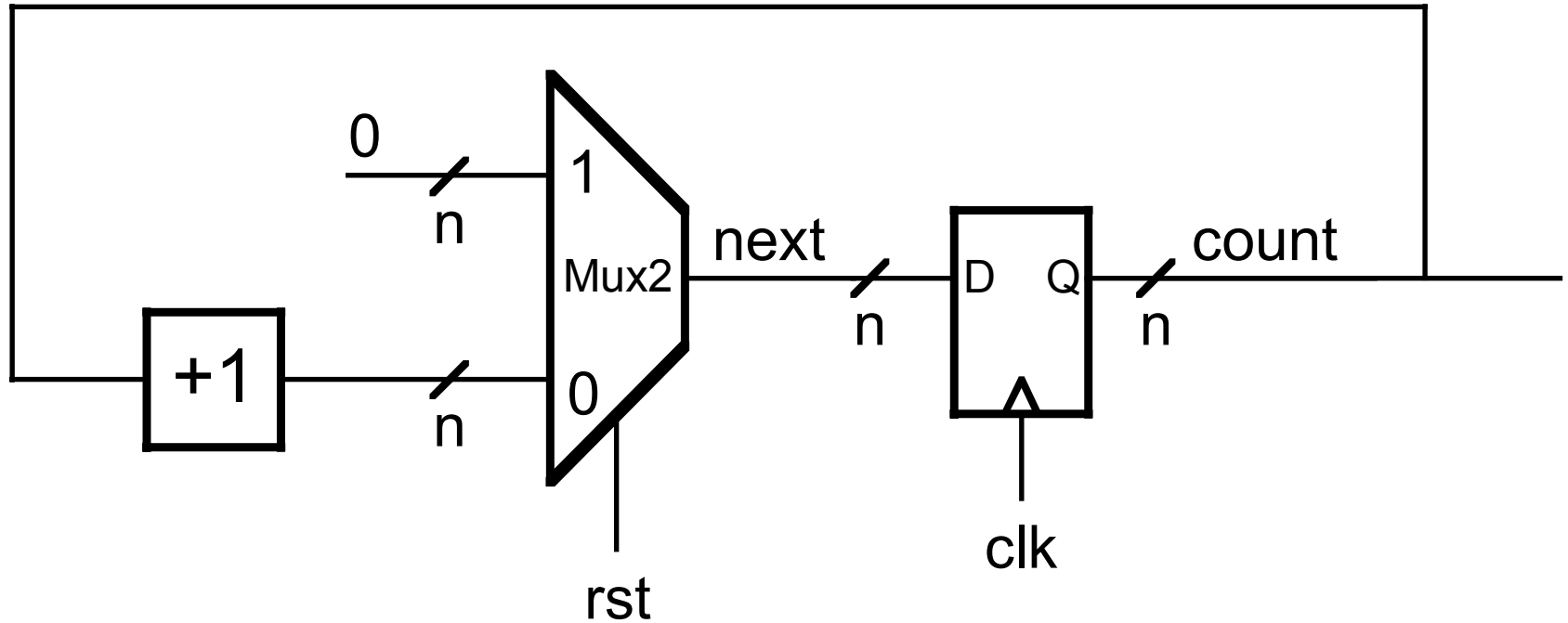
    DFF #(5) count(clk, next, out) ;

    always@(rst, out) begin
        case(rst)
            1'b1: next = 0 ;
            1'b0: next = out+1 ;
        endcase
    end
endmodule
```

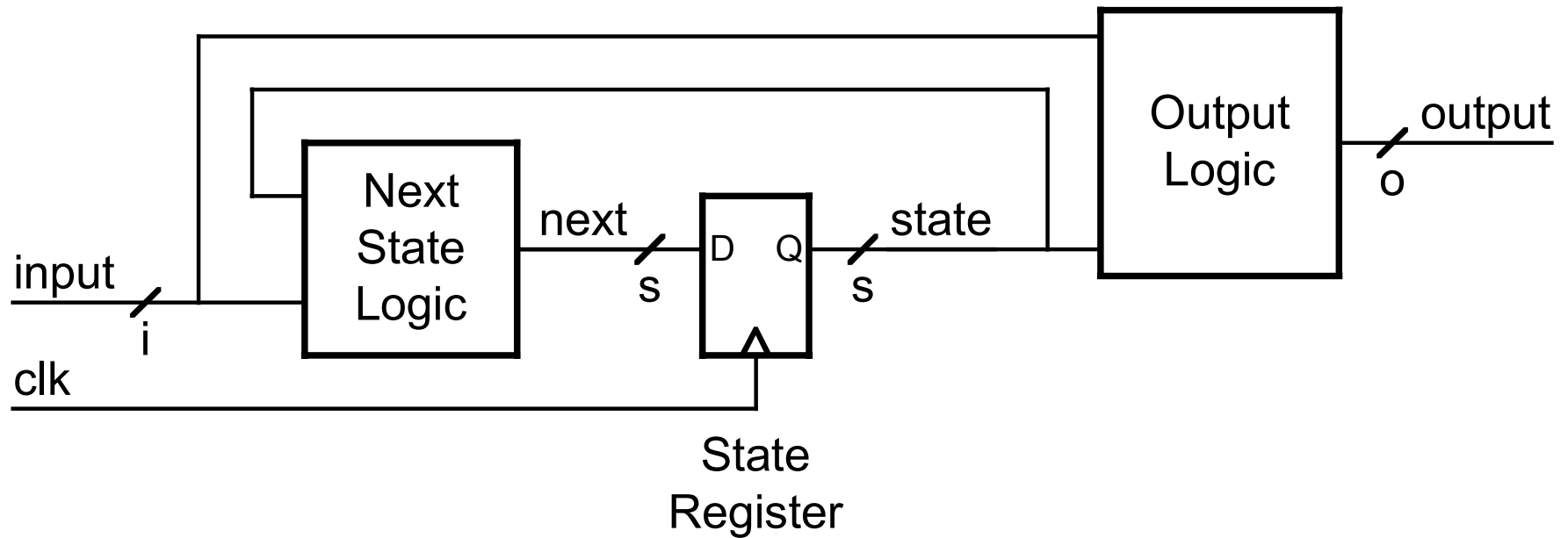
Schematic

A simple counter

$$\text{next_state} = \text{rst} ? 0 : \text{state} + 1$$



Sequential Datapath



An Up/Down/Load (UDL) Counter

A Deluxe Counter that can:

- count up (increment)
- count down (decrement)
- be loaded with a value

Up, down, and load guaranteed to be one-hot. Rst overrides.

if rst, next_state = 0

if (!rst & up) next_state = state+1

if (!rst & down) next_state = state-1

if (!rst & load) next_state = in

else next_state = state

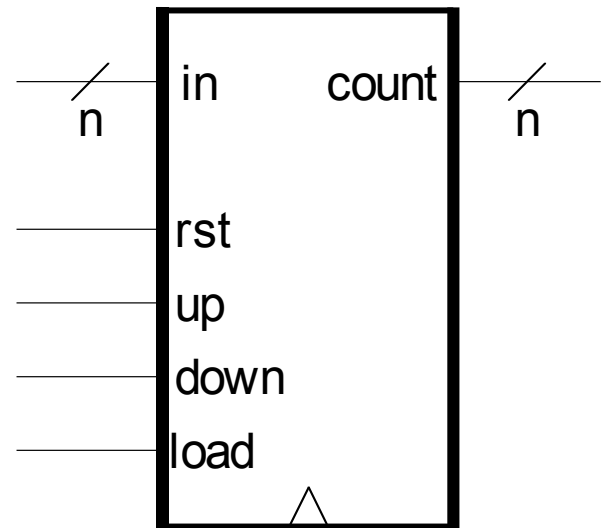


Table Version

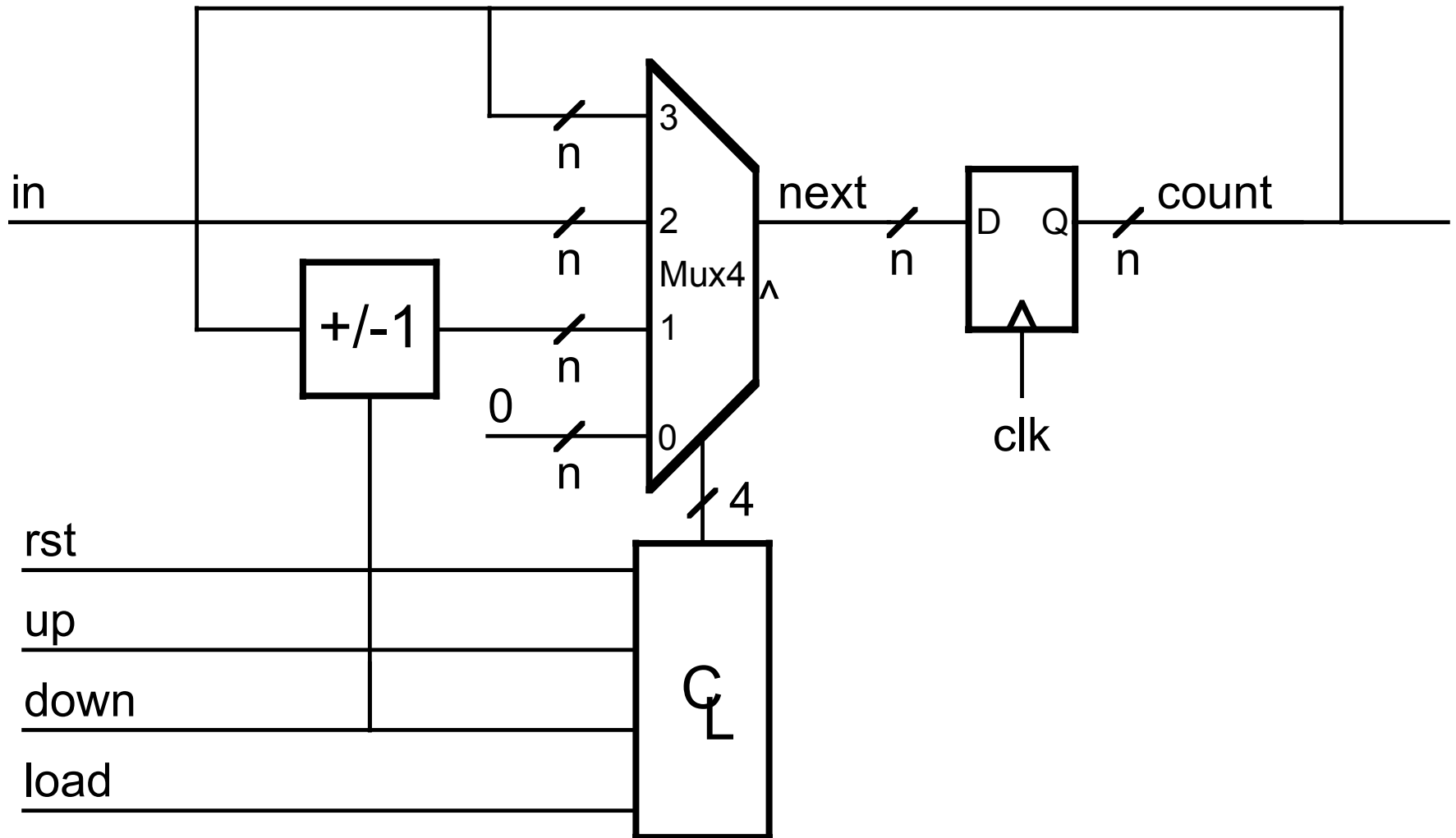
State	Next State			
	rst	up	down	load
0	0	1	31	in
1	0	2	0	in
2	0	3	1	in
▪ ▪ ▪				
30	0	31	29	in
31	0	0	30	in

Symbolic Table Version

State	Next State				
	rst	up	down	load	
q	0	q+1	q-1	in	q

State	In	Rst	Up	Down	Load	Next
q	x	1	x	x	x	0
q	x	0	1	0	0	q+1
q	x	0	0	1	0	q-1
x	y	0	0	0	1	y
q	x	0	0	0	0	q

Schematic of UDL Counter




```

module UDL_Count1(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] out ;
    reg [n-1:0] next ;

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

    always@(rst, up, down, load, out) begin
        casex({rst, up, down, load})
            4'b1xxx: next = {n{1'b0}} ;
            4'b0100: next = out + 1'b1 ;
            4'b0010: next = out - 1'b1 ;
            4'b0001: next = in ;
            4'b0000: next = out ;
            default: next = {n{1'bx}} ;
        endcase
    end
endmodule

```

```

module UDL_Count1(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] out, outpm1 ;
    reg [n-1:0] next ;

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

    assign outpm1 = out + {{n-1{down}},1'b1} ; // down ? -1 : 1

    always@(rst, up, down, load, in, out, outpm1) begin
        casex({rst, up, down, load})
            4'b1xxx: next = {n{1'b0}} ;
            4'b01xx: next = outpm1 ;
            4'b001x: next = outpm1 ;
            4'b0001: next = in ;
            default: next = out ;
        endcase
    end
end
endmodule

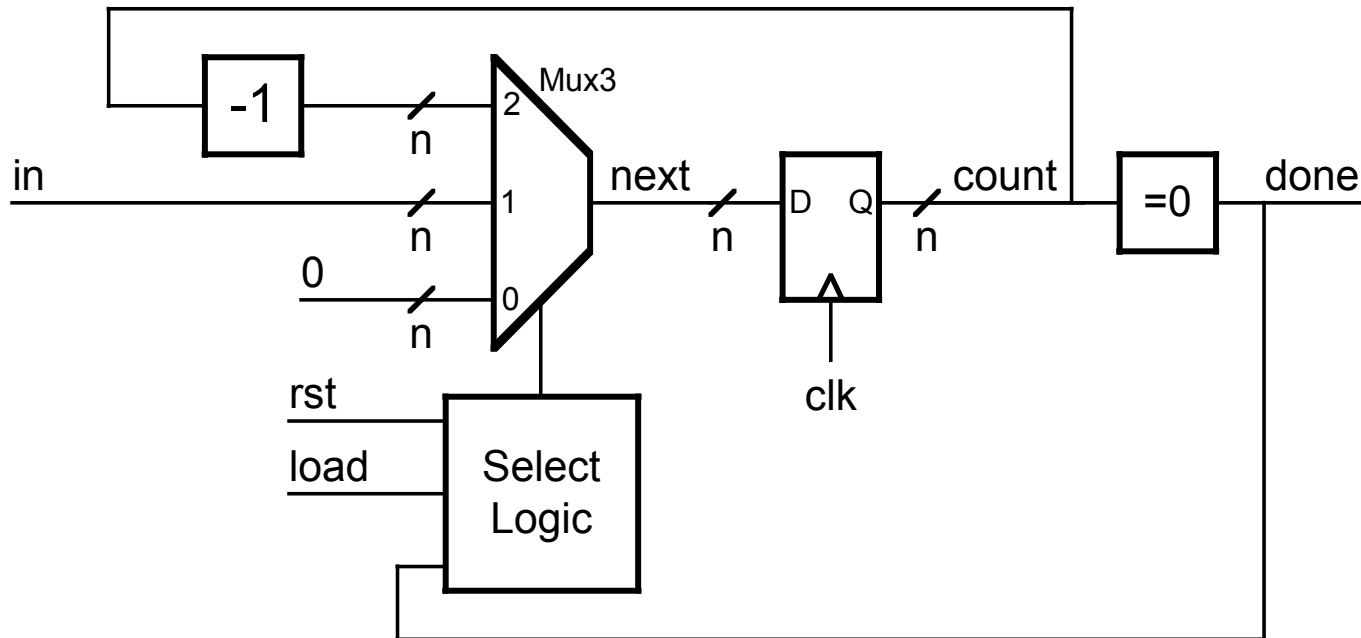
```

Timer module

load – loads count

done – asserted when count = 0

count decrements unless load or done is true



```

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) ;

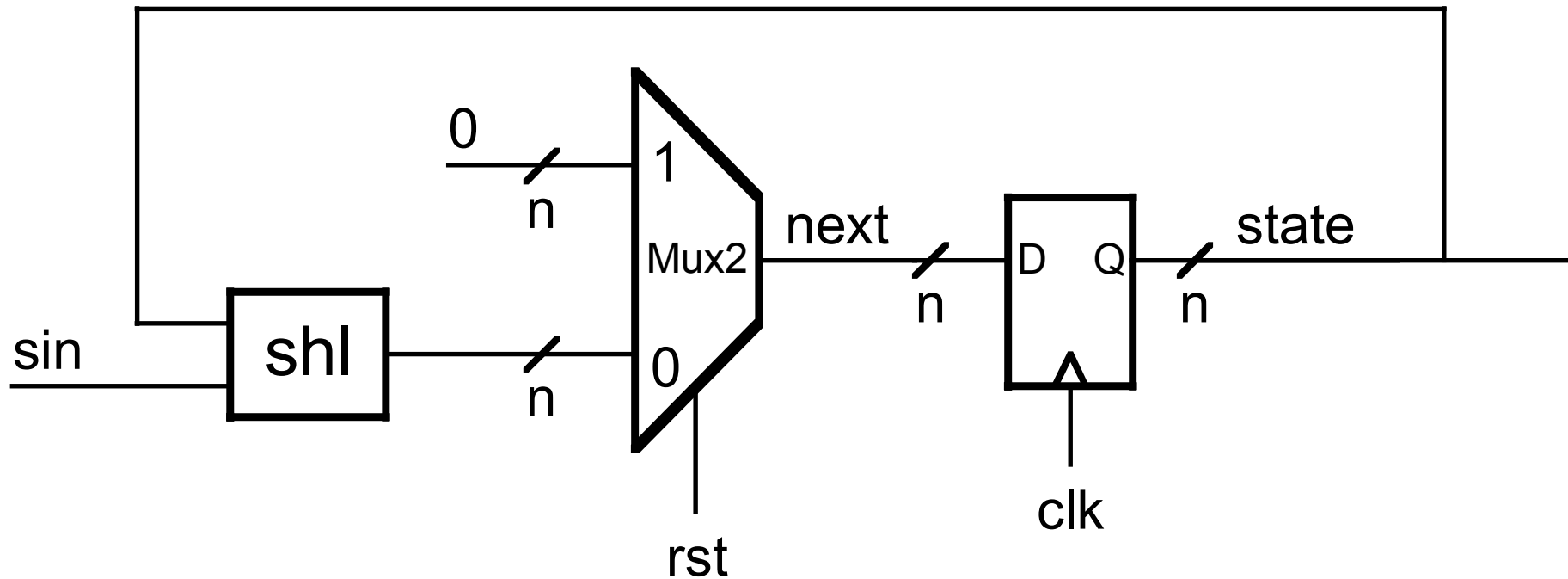
    always@(rst, load, in, out) begin
        casex({rst, load, done})
            3'b1xx: next_count = 0 ; // reset
            3'b001: next_count = 0 ; // done
            3'b01x: next_count = in ; // load
            default: next_count = count-1'b1; // count down
        endcase
    end

    assign done = (count == 0) ;
endmodule

```

Shift Register

`next_state = rst ? 0 : {state[n-2:0],sin} ;`



```
module Shift_Register1(clk, rst, sin, out) ;  
    parameter  $\bar{n}$  = 4 ;  
    input clk, rst, sin ;  
    output [n-1:0] out ;  
  
    wire [n-1:0] next = rst ? {n{1'b0}} : {out[n-2:0],sin} ;  
  
    DFF #(n) cnt(clk, next, out) ;  
endmodule
```

```

module LRL_Shift_Register1(clk, rst, left, right, load, sin, in, out) ;
    parameter n = 4 ;
    input clk, rst, left, right, load, sin ;
    input [n-1:0] in ;
    output [n-1:0] out ;
    reg [n-1:0] next ;

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

    always @(*) begin
        casex({rst,left,right,load})
            4'b1xxx: next = 0 ;                // reset
            4'b01xx: next = {out[n-2:0],sin} ; // left
            4'b001x: next = {sin,out[n-1:1]} ; // right
            4'b0001: next = in ;                // load
            default: next = out ;               // hold
        endcase
    end
endmodule

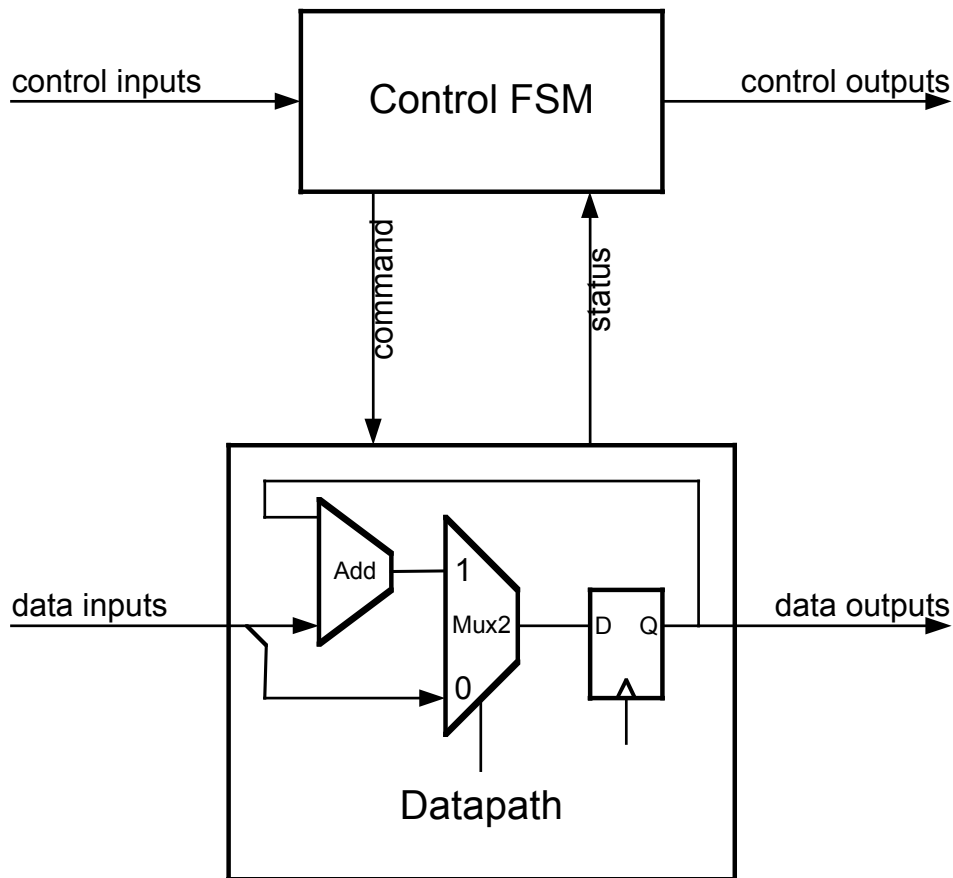
```

	rst	down			done			lrlsrout		
		load		sin		udlcout			srout	
		up			cout					

#	1	0	1	0	1	0000	1	0000	0000	0000
#	0	0	1	0	1	0000	1	0000	0000	0000
#	0	0	1	0	1	0001	1	0001	0001	0001
#	0	0	1	0	1	0010	1	0010	0011	0011
#	0	0	0	1	1	0011	1	0011	0111	0111
#	0	0	0	1	1	0100	1	0010	1111	1011
#	0	1	0	0	1	0101	1	0001	1111	1101
#	0	0	0	0	0	0110	0	0101	1111	0101
#	0	0	1	0	1	0111	0	0101	1110	0101
#	0	0	1	0	1	1000	0	0110	1101	1011
#	0	0	1	0	1	1001	0	0111	1011	0111
#	0	0	1	0	1	1010	0	1000	0111	1111
#	0	0	1	0	1	1011	1	1001	1111	1111
#	0	0	1	0	1	1100	1	1010	1111	1111

Datapath/Control Partitioning

Datapath state – determined by a function – e.g., mux, arithmetic, ...
Control state – determined by state diagram or state table



Consider a vending machine controller

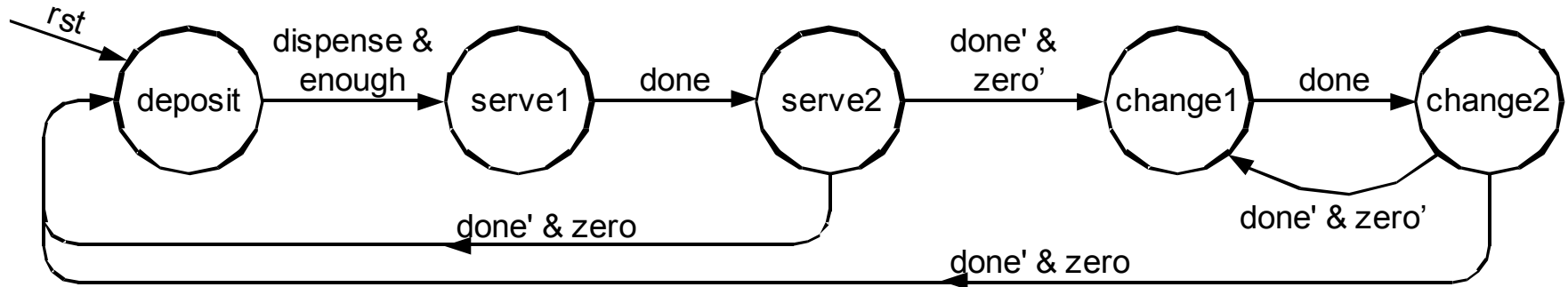
- Receives coins (nickel, dime, quarter) and accumulates sum
- When “dispense” button is pressed serves a drink if enough coins have been deposited
- Then returns change – one nickel at a time.

Partition task

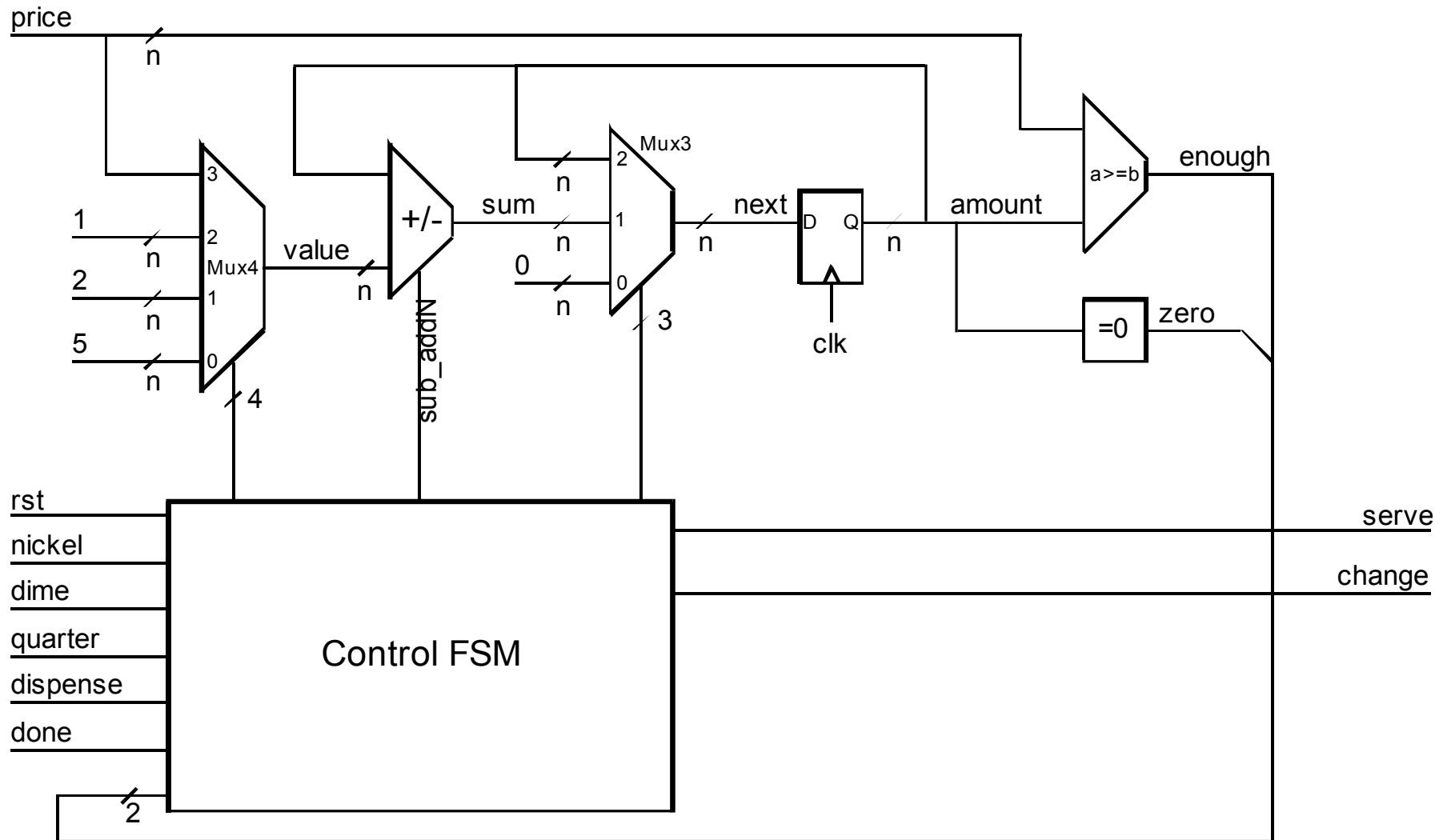
Datapath – keep track of amount owed user

Control – keep track of sequence – deposit, serve, change

State diagram for control portion



Block diagram of data path



```

//-----
// VendingMachine - Top level module
// Just hooks together control and datapath
//-----
module VendingMachine(clk, rst, nickel, dime, quarter, dispense, done, price,
                      serve, change) ;
    parameter n = `DWIDTH ;
    input clk, rst, nickel, dime, quarter, dispense, done ;
    input [n-1:0] price ;
    output serve, change ;

    wire enough, zero, sub ;
    wire [3:0] selval ;
    wire [2:0] selnext ;

    VendingMachineControl vmc(clk, rst, nickel, dime, quarter, dispense, done,
                              enough, zero, serve, change, selval, selnext, sub) ;

    VendingMachineData #(n) vmd(clk, selval, selnext, sub, price, enough, zero) ;
endmodule

```

```
//-----
module VendingMachineControl(clk, rst, nickel, dime, quarter, dispense,
done,
```

```
    enough, zero, serve, change, selval, selnext, sub) ;
input clk, rst, nickel, dime, quarter, dispense, done, enough, zero ;
output serve, change, sub ;
output [3:0] selval ;
output [2:0] selnext ;
wire [`SWIDTH-1:0] state, next ; // current and next state
reg [`SWIDTH-1:0] next1 ;        // next state w/o reset
```

```
// outputs
wire first ; // true during first cycle of serve1 or changel
wire serve1 = (state == `SERVE1) ;
wire changel = (state == `CHANGE1) ;
wire serve = serve1 & first ;
wire change = changel & first ;
```

```
// datapath controls
wire dep = (state == `DEPOSIT) ;
// price, 1, 2, 5
wire [3:0] selval = {(dep & dispense),
                    ((dep & nickel) | change),
                    (dep & dime),
                    (dep & quarter)} ;
```

```
// amount, sum, 0
wire selv = (dep & (nickel | dime | quarter |
                    (dispense & enough))) |
            (change) ;
wire [2:0] selnext = {(selv | rst),selv,rst} ;
```

```
// subtract
wire sub = (dep & dispense) | change ;
```

```
// only do actions on first cycle of serve1 or changel
wire nfirst = !(serve1 | changel) ;
DFF #(1) first_reg(clk, nfirst, first) ;
```

```
// state register
```

```
DFF #(`SWIDTH) state_reg(clk, next, state) ;
```

```
// next state logic
```

```
always @(state or zero or dispense or done or enough) begin
    casex({dispense, enough, done, zero, state})
        {4'b11xx,`DEPOSIT}: next1 = `SERVE1 ; // dispense & enough
        {4'b0xxx,`DEPOSIT}: next1 = `DEPOSIT ;
        {4'bx0xx,`DEPOSIT}: next1 = `DEPOSIT ;
        {4'bxx1x,`SERVE1}: next1 = `SERVE2 ; // done
        {4'bxx0x,`SERVE1}: next1 = `SERVE1 ;
        {4'bxx01,`SERVE2}: next1 = `DEPOSIT ; // ~done & zero
        {4'bxx00,`SERVE2}: next1 = `CHANGE1 ; // ~done & ~zero
        {4'bxx1x,`SERVE2}: next1 = `SERVE2 ; // done
        {4'bxx1x,`CHANGE1}: next1 = `CHANGE2 ; // done
        {4'bxx0x,`CHANGE1}: next1 = `CHANGE1 ; // ~done
        {4'bxx00,`CHANGE2}: next1 = `CHANGE1 ; // ~done & ~zero
        {4'bxx01,`CHANGE2}: next1 = `DEPOSIT ; // ~done & zero
        {4'bxx1x,`CHANGE2}: next1 = `CHANGE2 ; // done
    endcase
end
```

(c) 2005-2012 W.J. Dally

```
reset next state
assign next = rst ? `DEPOSIT : next1 ;
endmodule
```

```

module VendingMachineData(clk, selval, selnext, sub, price, enough, zero) ;
    parameter n = 6 ;
    input clk, sub ;
    input [3:0] selval ; // price, 1, 2, 5
    input [2:0] selnext ; // amount, sum, 0
    input [n-1:0] price ; // price of soft drink - in nickels
    output enough ; // amount > price
    output zero ; // amount = zero

    wire [n-1:0] sum ; // output of add/subtract unit
    wire [n-1:0] amount ; // current amount
    wire [n-1:0] next ; // next amount
    wire [n-1:0] value ; // value to add or subtract from amount
    wire ovf ; // overflow - ignore for now

    // state register holds current amount
    DFF #(n) amt(clk, next, amount) ;

    // select next state from 0, sum, or hold
    Mux3 #(n) nsmux({n{1'b0}}, sum, amount, selnext, next) ;

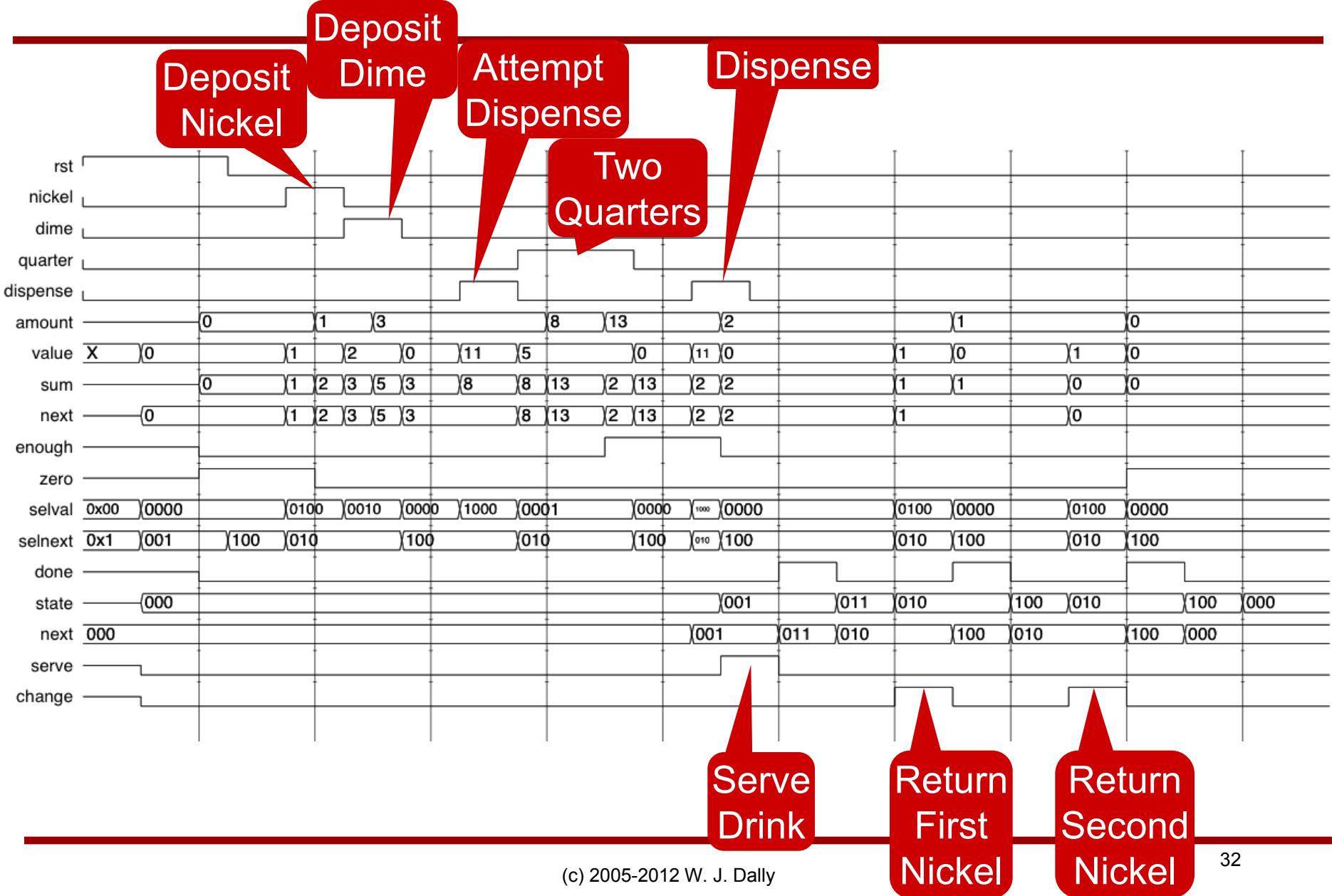
    // add or subtract a value from current amount
    AddSub #(n) add(amount, value, sub, sum, ovf) ;

    // select the value to add or subtract
    Mux4 #(n) vmux(`QUARTER, `DIME, `NICKEL, price, selval, value) ;

    // comparators
    wire enough = (amount >= price) ;
    wire zero = (amount == 0) ;
endmodule

```

Waveforms for vending machine



Summary

- Datapath state machines
 - Next state function specified by an expression, not a table
$$\text{next} = \text{rst} ? 0 : (\text{inc} ? \text{next} + 1 : \text{next}) ;$$
 - Common “idioms”
 - Counters
 - Shift registers
- Datapath and control partitioning
 - Divide state space into control (deposit, serve, change) and data
 - FSM determines control state
 - Datapath computes amount
 - Status and control signals
 - Special case of factoring