Digital Circuits and Systems Lecture 8 Model Dapath FSM

Tian Sheuan Chang

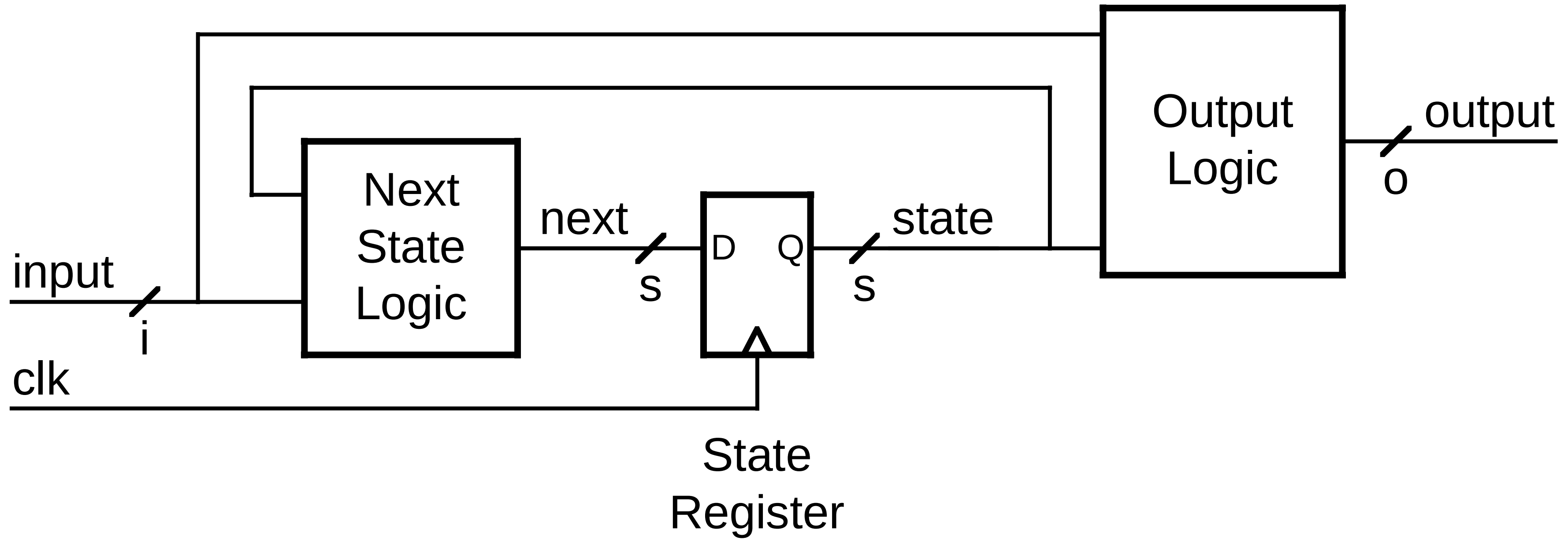
FSM 百百種，但大部分的next state 用counter 類簡單計算就可以，不用table

Outline [Dally ch. 16]

- Counter
  - Simple counter, up down counter
- Shift register
- Datapath and control partitioning
  - Vending machine example

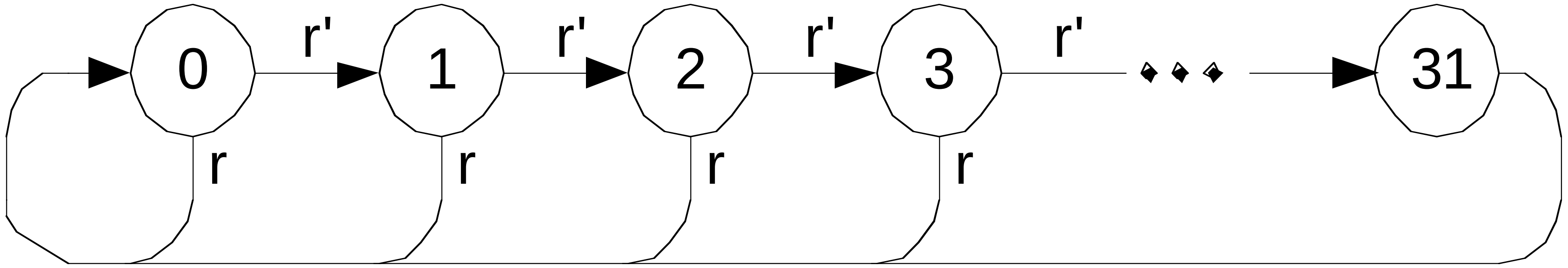
COUNTER AS FSM
COUNTER 當FSM控制

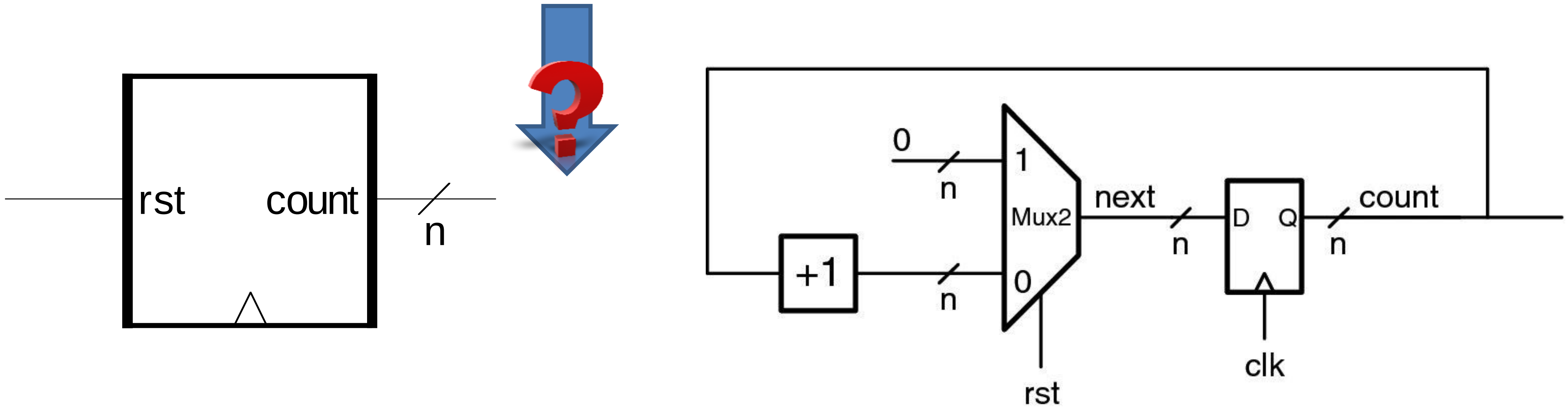
An FSM is a state register and two functions



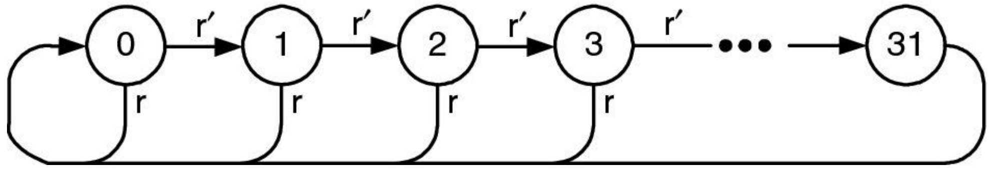
Counter 當FSM

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





第一種寫法: 表格式 Table based



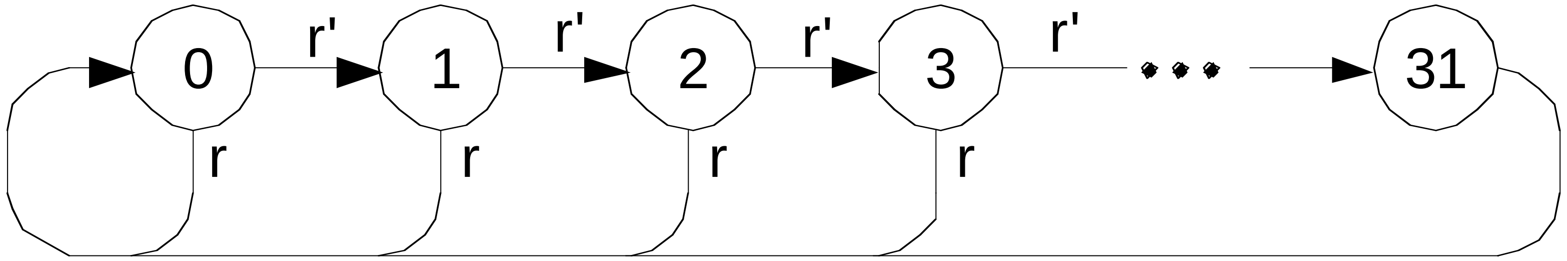
State	Next State	
	~rst	rst
0	1	0
1	2	0
2	3	0
. . .		
30	31	0
31	0	0

```
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) ;//5-bit counter

  always_comb begin //any change
    casez({rst,out})
      6'b1?????: next = 0 ;
      6'd0: next = 1 ;
      6'd1: next = 2 ;
      6'd2: next = 3 ;
      …
      6'd30: next = 31 ;
      6'd31: next = 0 ;
      default: next = 0 ;
    endcase
  end
endmodule
```

表格很好，但這個變化很規則，可以更簡單描述

next = r ? 0 : state + 1 ;

第二種寫法: 用算數作FSM Datapath FSM



```
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	
	~rst	rst
0	1	0
1	2	0
2	3	0
. . .		
30	31	0
31	0	0

State	Next State	
	~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
    casez({rst,out})
      6'b1?????: 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
```

Quiz: what's the difference Between these 2 coding examples?

Alternate description (symbolic table, ReWrite for coding Style)

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

  always@(posedge clk or posedge rst)
    if(rst) out <=0;
    else out <= next;

  always@(*) begin
    case({out})
      5'd0: next = 1 ;
      5'd1: next = 2 ;
      5'd2: next = 3 ;
      5'd3: next = 4 ;
      5'd4: next = 5 ;
      5'd5: next = 6 ;
      5'd6: next = 7 ;
      …
      5'd30: next = 31 ;
      5'd31: next = 0 ;
   endcase
  end
endmodule
```

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

  always@(posedge clk or posedge rst)
    if(rst) out <=0;
    else out <= next;

  always@(*) begin
     next = out+1 ;
   end
endmodule
```

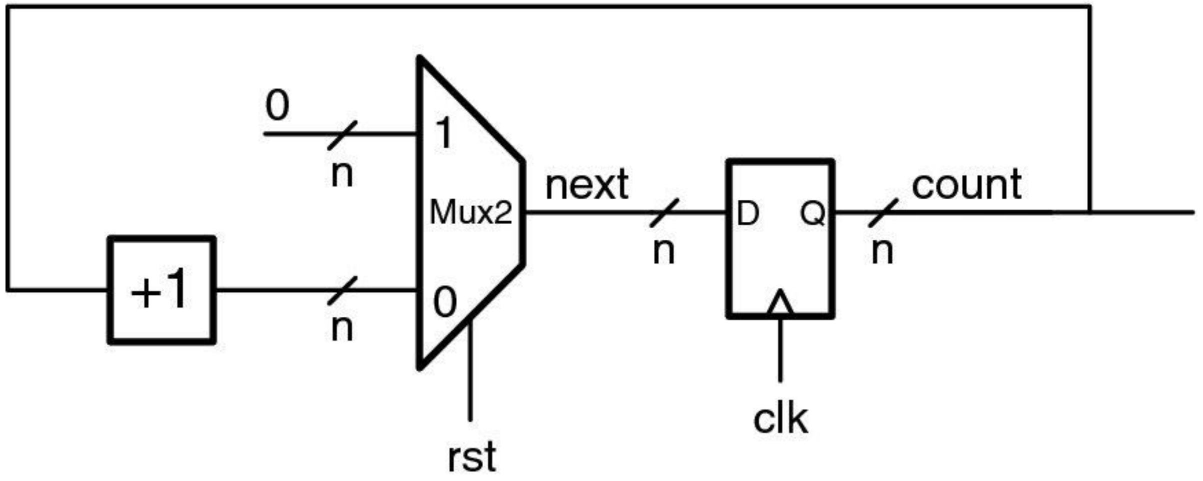
Quiz: what's the difference Between these 2 coding examples?

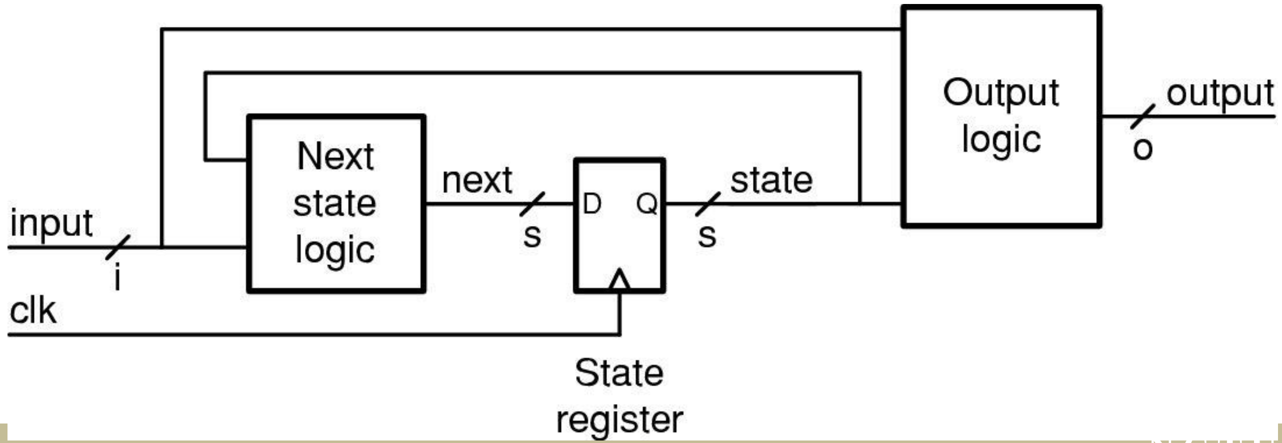
Schematic: Counter as FSM

A simple counter

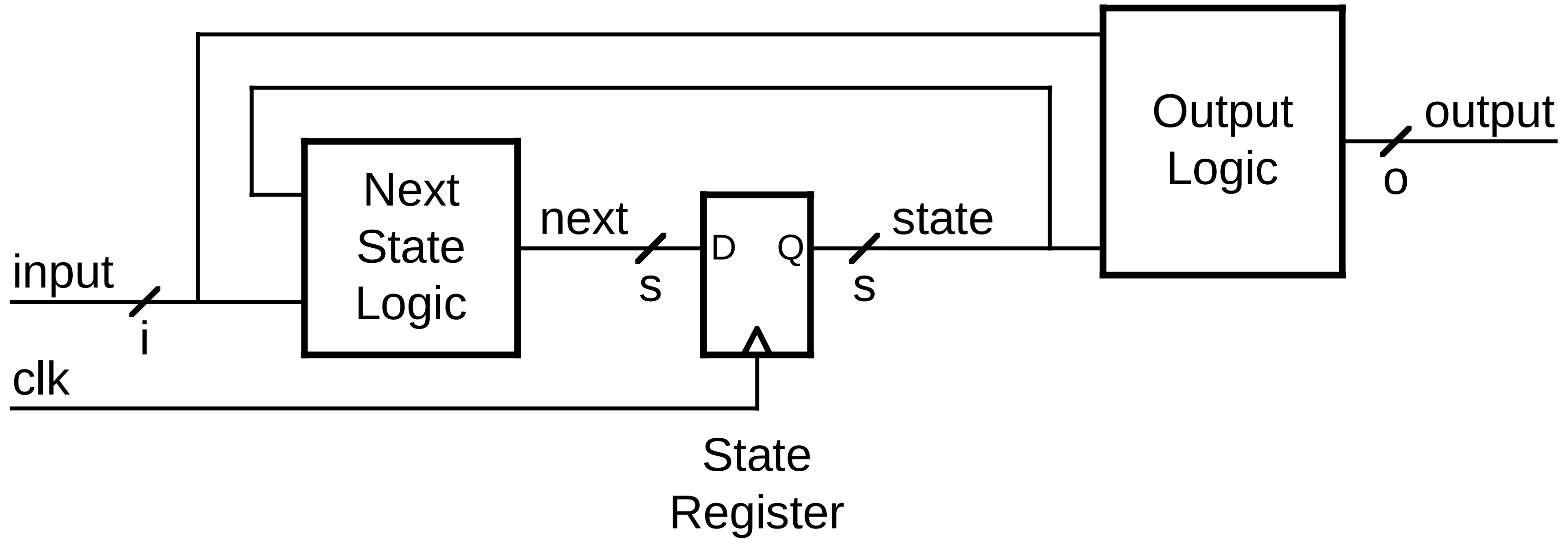
next_state = rst ? 0 : state + 1

MUX and D-FF can be replaced
By resettable D-FF





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
```

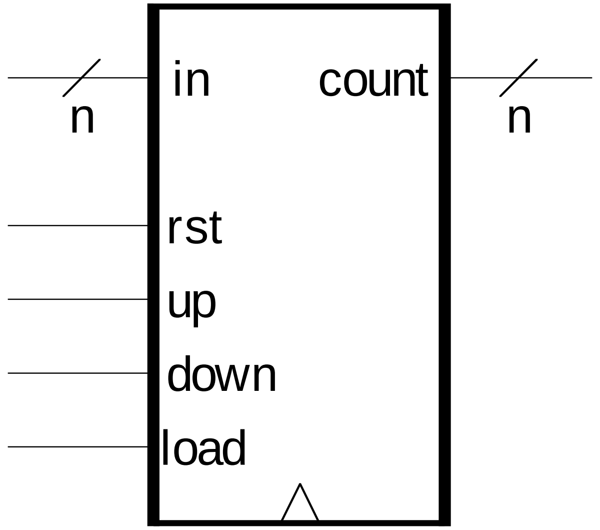


Table Version

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	else
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

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

```
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_comb begin
    casez({rst, up, down, load})
      4'b1???: next = {n{1'b0}} ; //reset
      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}} ;//unknown
    endcase
  end
endmodule
```

Up or down，需要兩個加減法器，可以更簡化嗎?

Check p.11 async rst

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

```
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
    casez({rst, up, down, load})
      4'b1???: next = {n{1'b0}} ;
      4'b01??: next = outpm1 ;
      4'b001?: next = outpm1 ;
      4'b0001: next = in ;
      default: next = out ;
    endcase
  end
endmodule
```

只用一個加減法器

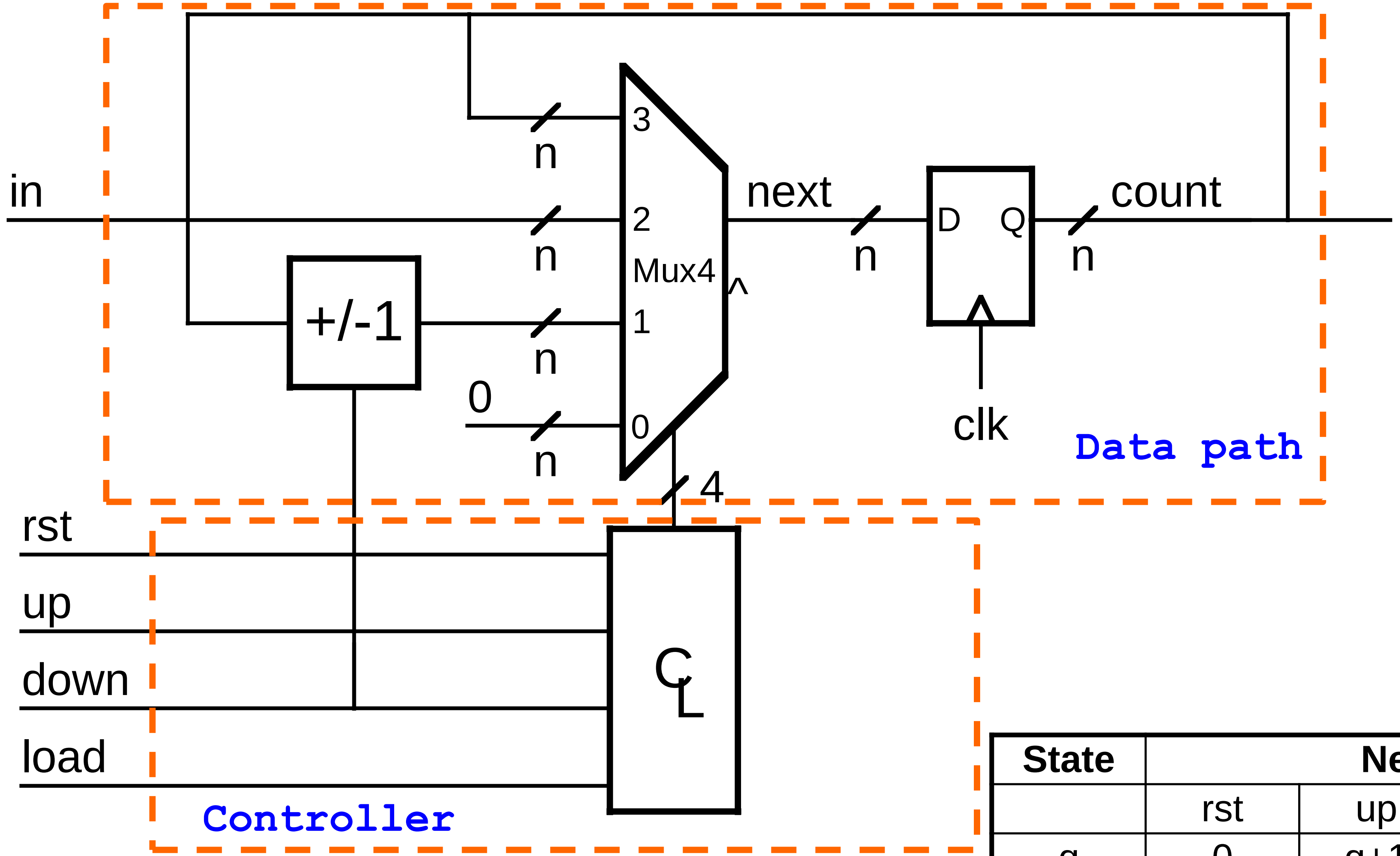
} 把可以共用的資源提出來

Quiz: what's the difference Between these 2 coding examples?

Check p.11 async rst

(c) 2005-2012 W. J. Dally

Schematic of UDL Counter



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

EXERCISE- Divide by 3 with Data Path

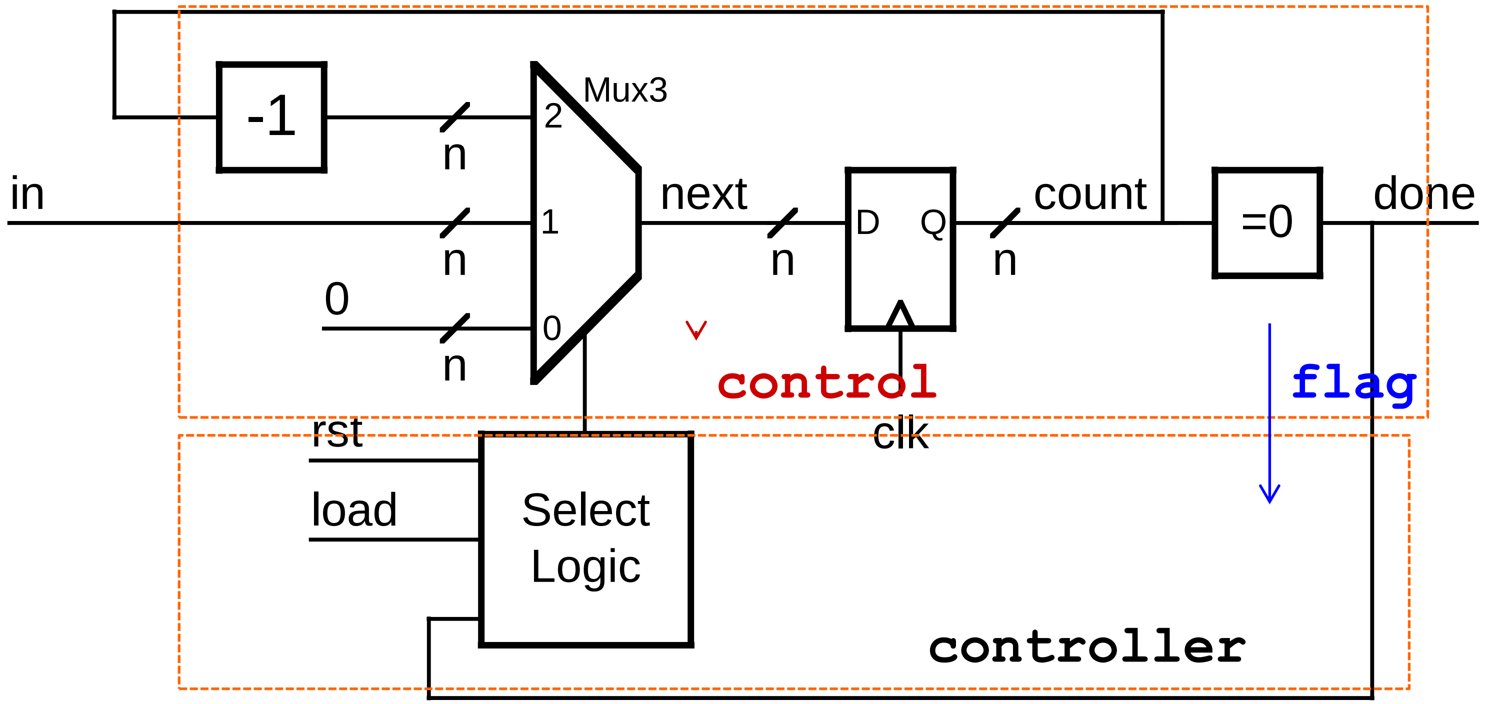
- *Output: (out) goes high once for each third cycle that input (in) is high*
- Example
  - IN: 0101010011100
  - OUT: 0000001000010
  - Note one cycle delay

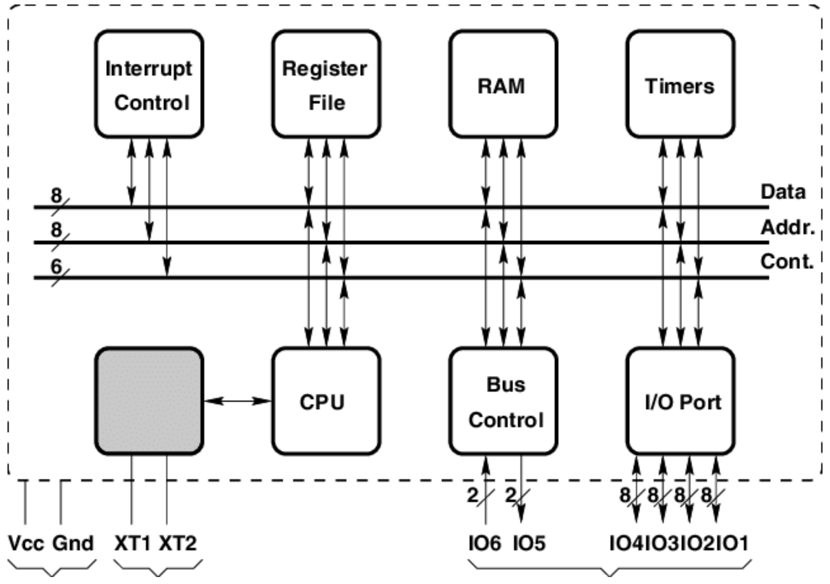
- **Draw the data path**
  - **What arithmetic operators are demanded?**
  - **What are the inputs and outputs?**
  - **What are the control signals?**

- How do we make the output go high in the same cycle as the third input 1?

Timer module (Set a Value and Decrease)

load – loads count
done – asserted when count = 0
count decrements unless load or done is true





8051

Data path: adder+mux+D-FF+comparator
Controller: combinational logic with inputs and flags
Exercise: realize divide-by-3 function on this module (Ex. 16.1)

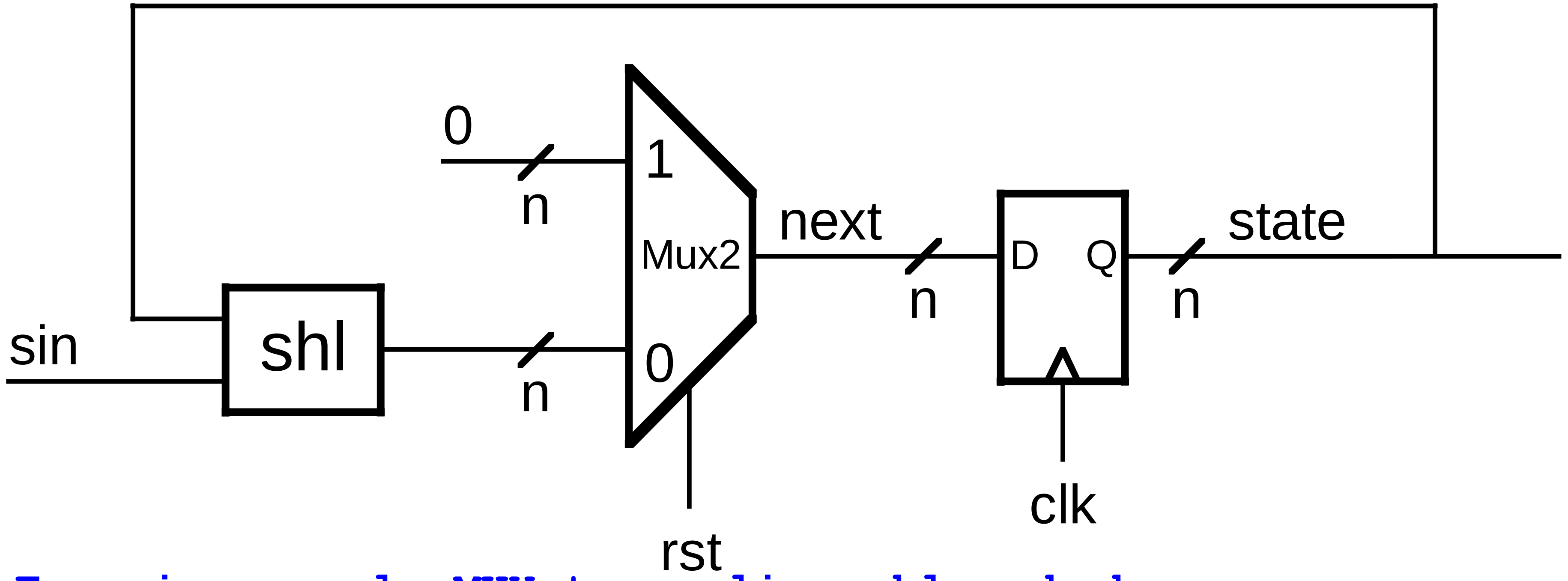
```
module Timer(clk, rst, load, in, done) ;
  parameter n=4 ;
  input clk, rst, load ;
  input [n-1:0] in ;
  output done ;
  wire [n-1:0] count, next_count ;
  wire done ;

  DFF #(n) cnt(clk, next_count, count) ;

  always@(rst, load, in, out) begin
    casez({rst, load, done})
      3'b1??: next_count = 0 ;  // reset
      3'b001: next_count = 0 ;  // done
      3'b01?: next_count = in ; // load
      default: next_count = count-1'b1; // count down
    endcase
  end

  assign done = (count == 0) ;
endmodule
```

Check p.11 async rst

SHIFT REGISTERS

Shift Register (+ Shift left + Shift right)

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



Exercise: apply MUX to realize shl and shr, which can be merged with rst.

Quiz: what function can be achieved if D-FF's are connected serially with parallel data out?

```
module Shift_Register1(clk, rst, sin, out) ;
  parameter 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
```

Check p.11 async rst

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

  wire [n-1:0] next;
  assign next = {out[n-2:0],sin} ;
  always_ff@(posedge clk or posedge rst)
    if(rst) out <= {n{1'b0}} ;
    else out <= next;

endmodule
```

A better coding style

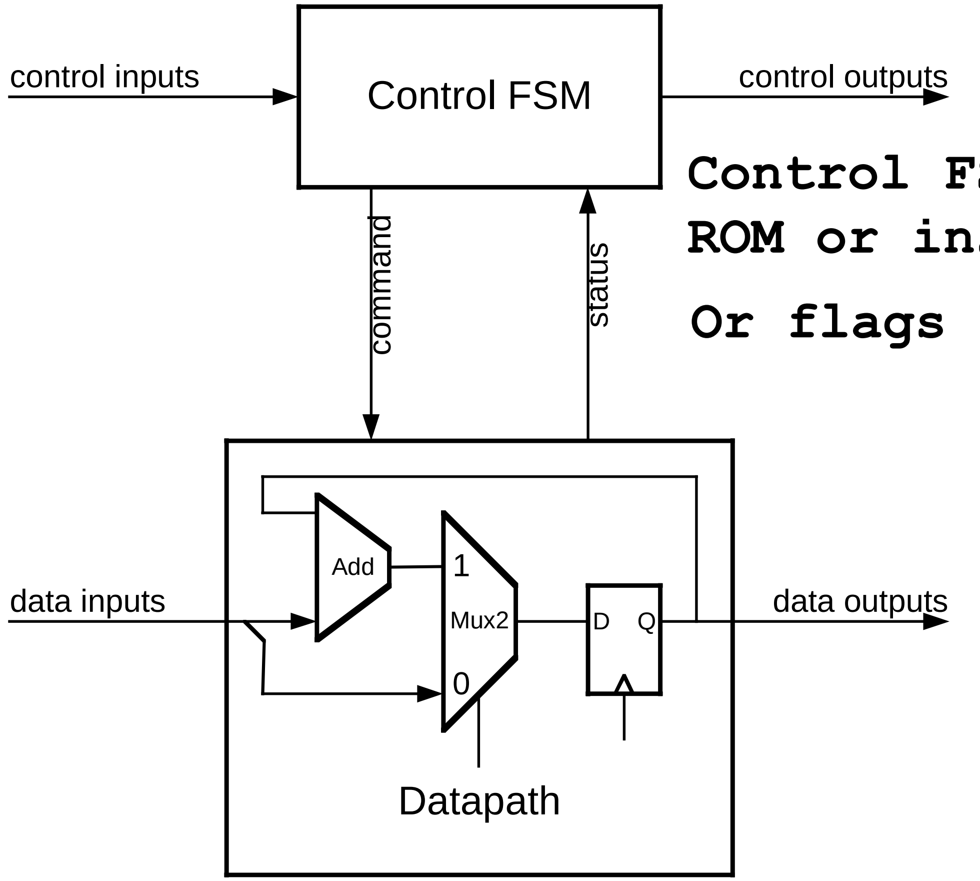
```
module LRL_Shift_Register1(clk, rst, left, right, load, sin, in,
out) ;
  parameter n = 4 ;
  input clk, rst, left, right, load, sin ;
  input [n-1:0] in ;
  output [n-1:0] out ;
  reg [n-1:0] next ;
  reg    [4:0] out ;

  always_ff@(posedge clk or posedge rst)
    if(rst) out <=0;
    else out <= next;

  always_comb begin
    casez({left,right,load})
      3'b1??: next = {out[n-2:0],sin} ; // left
      3'b01?: next = {sin,out[n-1:1]} ; // right
      3'b001: next = in ;               // load
      default: next = out ;             // hold
    endcase
  end
endmodule
```

DATAPATH AND CONTROL PARTITIONING
所有數位設計都可拆成資料流計算與控制兩部分

Datapath/Control Partitioning

Datapath – determined by a function – e.g., mux, arithmetic, …

Control – determined by state diagram or state table



Control FSM can be realized by ROM or instruction memory

Or flags

- 規格
  - The vending machine accepts *nickels, dimes, and quarters*. Whenever a coin is deposited into the coin slot, a pulse appears for one clock cycle on one of *three lines indicating the type of coin: nickel, dime, or quarter*.
  - The price of the item is set on an n-bit switch internal to the machine (in units of nickels), and is input to the controller on the n-bit signal *price*.
  - When sufficient coins have been deposited to purchase a soft drink, the status signal *enough* is asserted. Any time *enough* is asserted and the user press a ***disperse*** button; signal *serve* is asserted for exactly one cycle to serve the soft drink. After asserting serve, the FSM must wait until signal done is asserted, indicating that the mechanism has finished serving the soft drink.

- After serving, the machine returns change (if any) to the user. It does this one nickel at a time, asserting the signal change, for exactly one cycle and waiting for signal done to indicate that a nickel has been dispensed before dispensing the next nickel or returning to its original state. Any time the signal done is asserted, we must wait for done to go low before proceeding.

Consider a vending machine controller

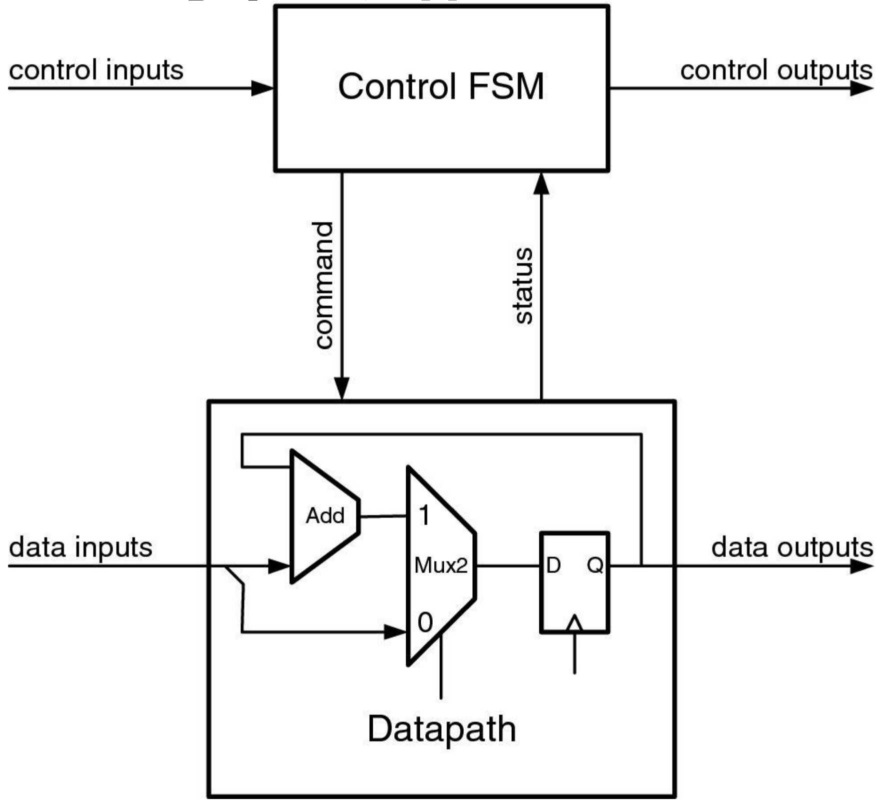
- 規格
  - Receives coins (nickel, dime, quarter) and accumulates sum 算錢
  - When "dispense" button is pressed serves a drink if enough coins have been deposited 給飲料
  - Then returns change – one nickel at a time. 找零
- 如何設計
  - 給定規格，先定義出資料流計算(datapath)部分和控制訊號(state diagram)兩部分
  - //given system specs, you need to define both operations (data path) and control signals (state diagram)
- Partition task
  - Datapath – keep track of amount owed user 算錢
  - Control – keep track of sequence – deposit, serve, change 動作流程

先處理 Datapath: data state

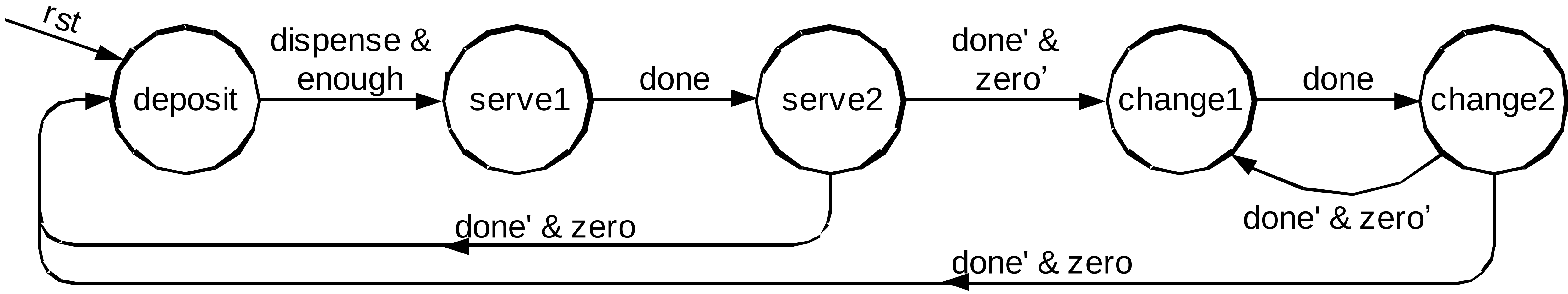
- Data state:
  - amount of money 目前投了多少錢 (in unit of nickels): *amount*
- 影響amount的運算
  - Reset: amount =0;
  - Deposit a coin: amount = amount + value
    - Value = 1, 2, 5 for a nickel, dime, or quarter
  - Return one nickel of change: amount = amount -1
  - Otherwise: no change

再處理control part: data 改變的時間點

- 先確定輸出入
- 輸入
  - 資料 : (nickel, dime, quarter), price
  - 動作: dispense, done
  - 系統: rst, clk
- 輸出
  - serve, change
- 內部Status/command signal
  - Status: enough, zero
  - Command: 決定state時一起看



再處理control part :State diagram

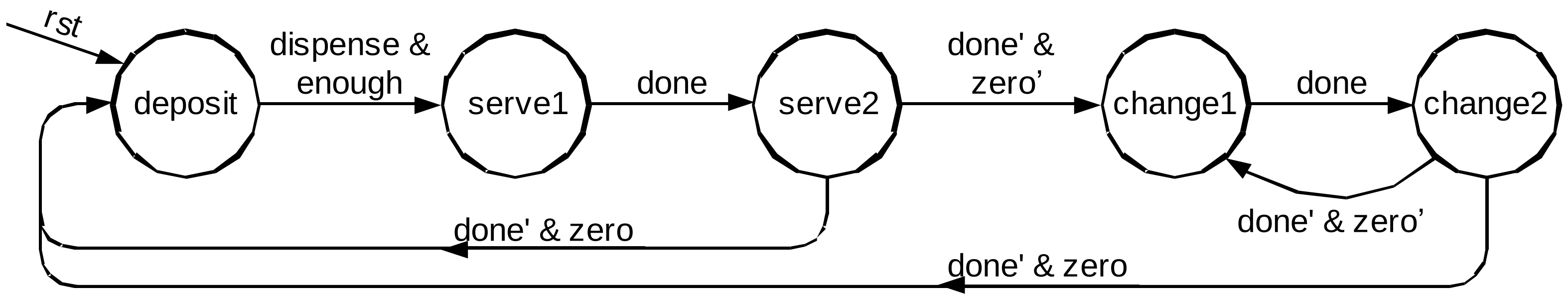


- In this diagram, edges from a state to itself are omitted. If the conditions on all edges leading out of the current state are not satisfied, the FSM stays in that state.

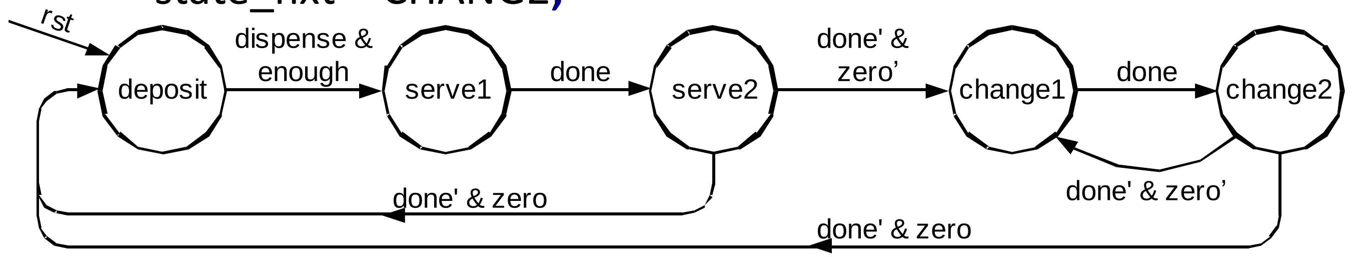
```
module VendingMachine(clk, rst, nickel, dime, quarter, dispense, done, price,
                     serve, change) ;
 parameter n = `DWIDTH ;
 input clk, rst, nickel, dime, quarter, dispense, done ;
 input [n-1:0] price ;
 output serve, change ;

 wire serve, change;
 //internal signals
 wire enough, zero;

 //variable declaration
 logic [2:0] value;
 logic [n-1:0] amount;

 assign value = nickel ? 1 : (dime ? 2 : 5));
 assign zero = amount == 0;
 assign enough = amount >= price;

always_ff @(posedge clk or posedge rst)
  if(rst) amount <= 0;
  else amount <= amount_nxt;
```

- Data state:
  - amount of money 目前投了多少錢 (in unit of nickels): *amount*
- 影響amount的運算
  - Reset: amount =0;
  - Deposit a coin: amount = amount + value
    - Value = 1, 2, 5 for a nickel, dime, or quarter
  - Return one nickel of change: amount = amount -1
  - Otherwise: no change

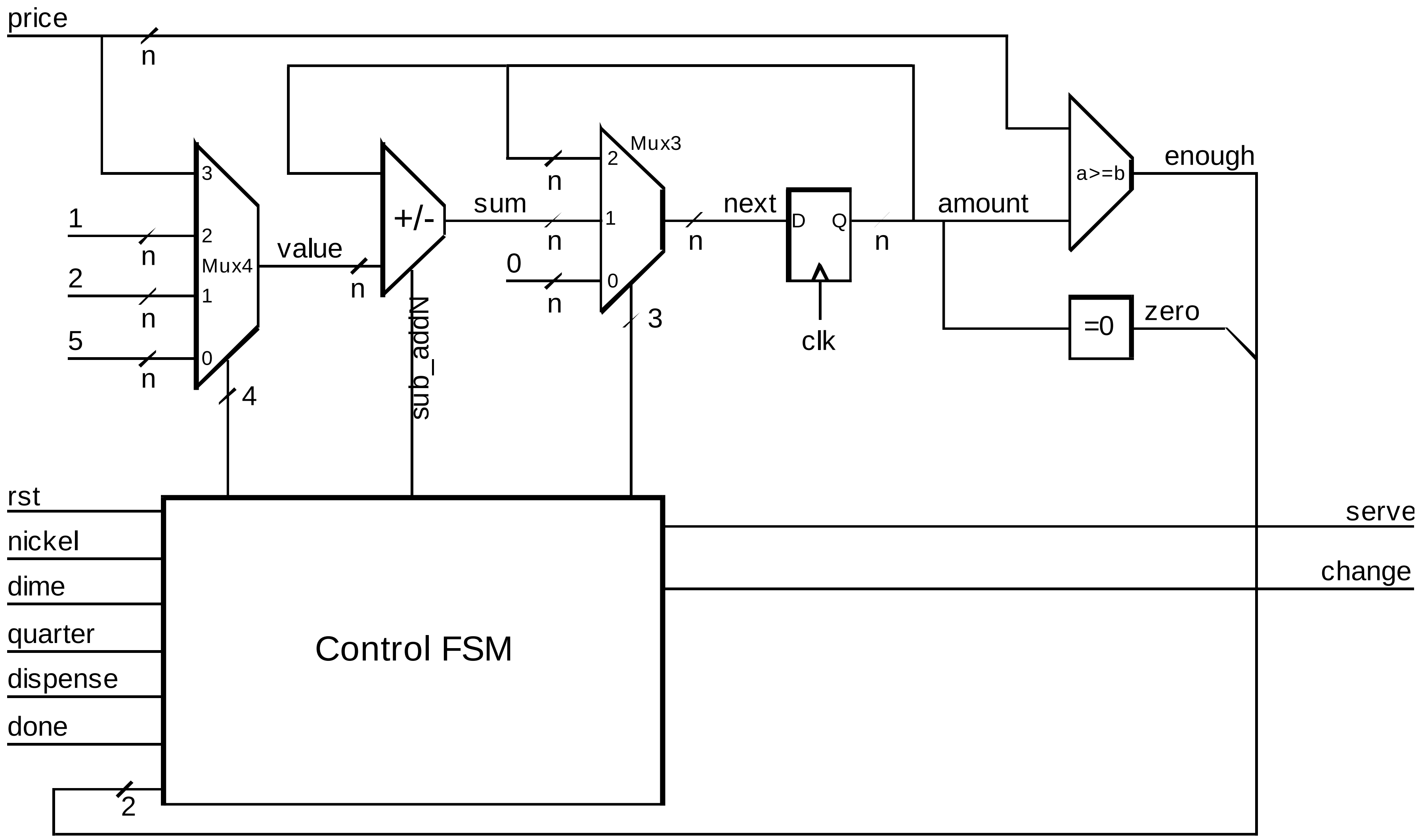
```
//state machine
 parameter [2:0]    DEPOSIT = 0,
                    SERVE1  = 1,
                    SERVE2  = 2,
                    CHANG1  = 3,
                    CHANG2  = 4;
 reg [2:0] state, state_nxt;

 always_ff @(posedge clk or posedge rst)
   if(rst == 1) state <= DEPOSIT;
              else state <= state_nxt;
```



```
always_comb begin
        amount_nxt = amount;
            state_nxt = DEPOSIT;
                serve = 0;
              change = 0;
  case(state)
            DEPOSIT: begin
                                 if(dispense & enough) state_nxt = SERVE1;
                                 else                  state_nxt = DEPOSIT;

                                 amount_nxt = amount + value;
                                 if(dispense & enough) serve = 1; //when from DEPOSIT to SERVE1
                     end
            SERVE1:  begin
                                          if(done) state_nxt = SERVE2;
                                          else     state_nxt = SERVE1;

                                          amount_nxt = amount - price;
                      end
            SERVE2:  begin
                                          if(~done & zero)         state_nxt = DEPOSIT;
                                          else if(~done & ~zero) state_nxt = CHANG1;
                                          else                          state_nxt = SERVE2;

                                          if(~done & ~zero) change = 1; //when from SERVE2 to CHANG1

                               end
            CHANG1:  begin
                                          if(done) state_nxt = CHANG2;
                                          else      state_nxt = CHANG1;

                                          amount_nxt = amount - 1;

                               end
            CHANG2:  begin
                                          if(~done & ~zero)      state_nxt = CHANG1;
                                          else if(~done & zero) state_nxt = DEPOSIT;
                                          else                         state_nxt = CHANG2;

                               end
            endcase
        end
endmodule
```



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 ;

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

//define data path
  VendingMachineData #(n) vmd(clk, selval, selnext, sub, price, enough, zero) ;
endmodule
```

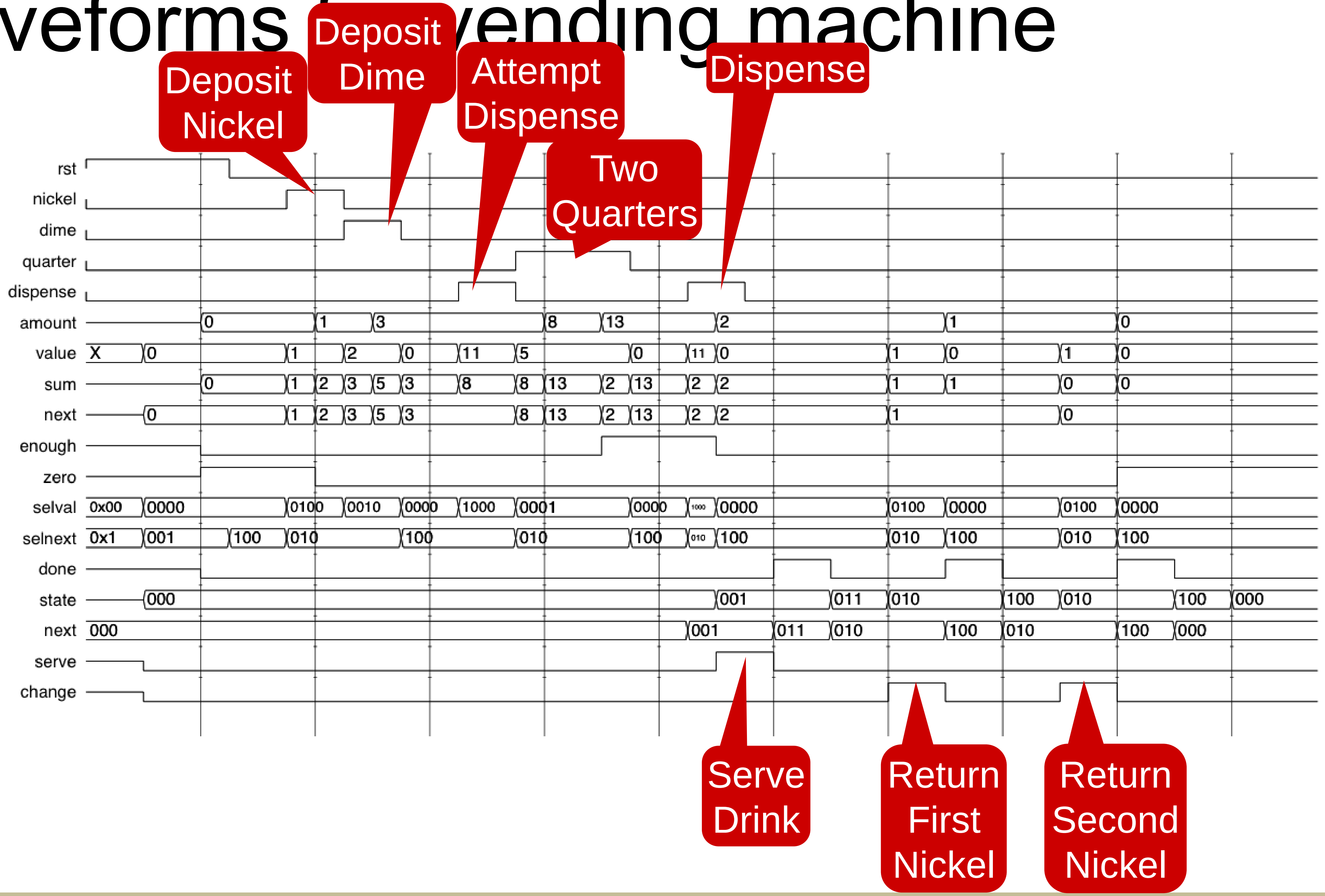
Another coding style: in two modules

```
//----------------------------------------------------------------------
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 change1
  wire serve1 = (state == `SERVE1) ;
  wire change1 = (state == `CHANGE1) ;
  wire serve = serve1 & first ;
  wire change = change1 & 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 change1
  wire nfirst = !(serve1 | change1) ;
  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

  // 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
    - next = rst ? 0 : (inc ? next + 1 : 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
- Lead to an arithmetic and logic unit (ALU) in processor designs