Logic Design
Lecture 8 Registers and Counters

EE2280, Fall 2017

Jenny Yi-Chun Liu

jennyliu@gapp.nthu.edu.tw

Outline

- Digital systems and information
- Boolean algebra
- Verilog introduction
- Combinational logic design
- Combinational building blocks
- Arithmetic functions
- Sequential logic
- Datapath sequential logic
- Memory

Chapter Outline

- Registers
- Counters
- Control and Datapath Partitioning

Registers (1/2)

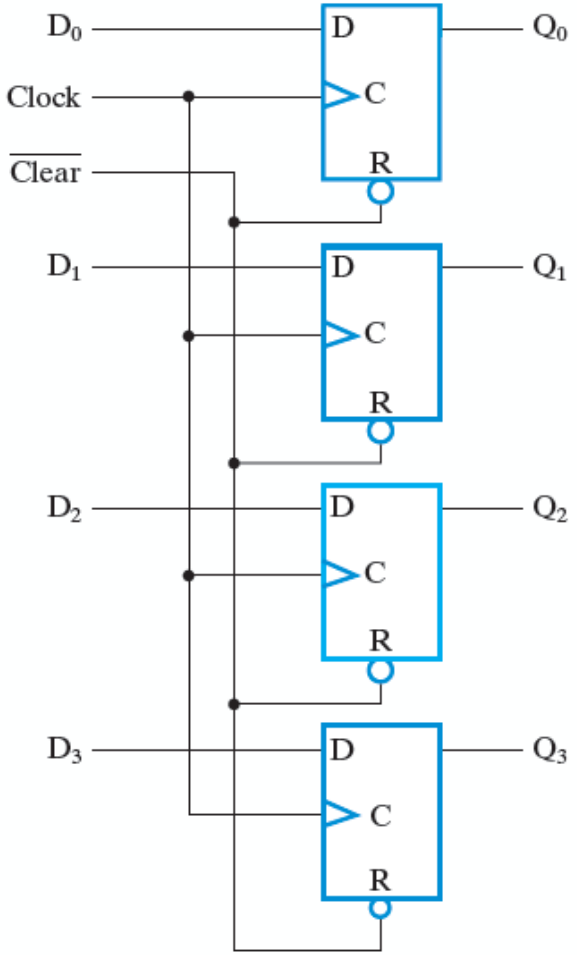
- Register: a collection of binary storage elements

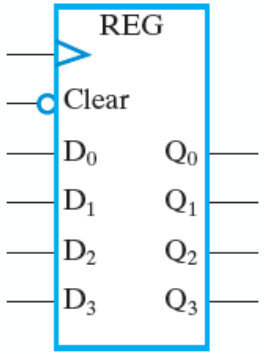
- An n-bit register stores n bit binary information.
- Registers are commonly used to perform data storage and processing.
- A register may have combinational gates to control when and how the new information is transferred into the register.
- Transfer new information into a register is called loading the register.
- Counter: a register that goes through a predetermined sequence of states.

Register (2/2)

- A register can be viewed as a bitwise extension of a flip-flop.
  - The simplest storage component with n inputs, n outputs, and a clock signal.
  - All flip-flops are driven by a common clock signal.

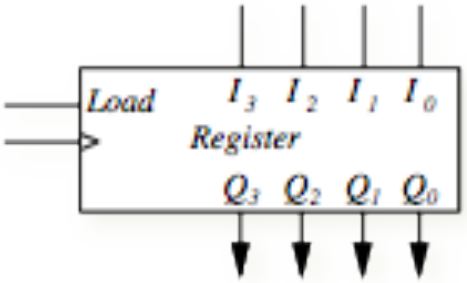
4-bit register



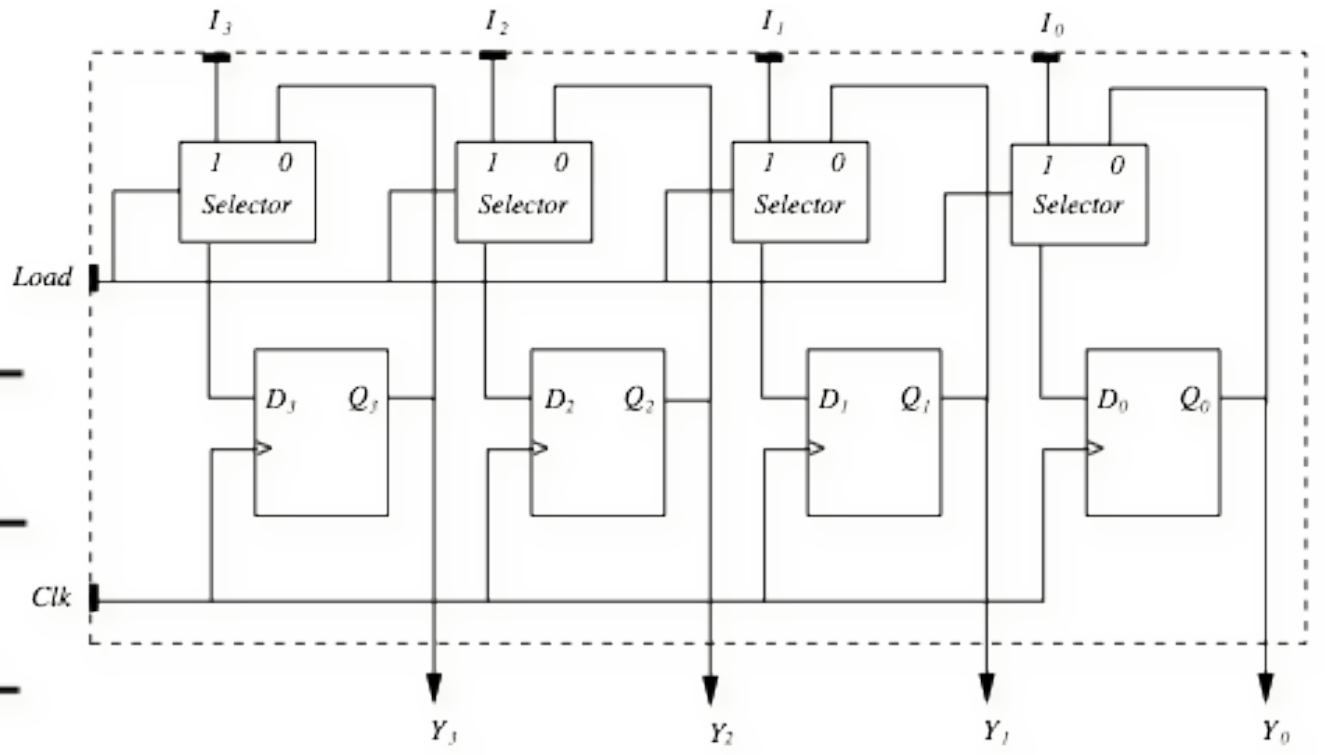


Register with Parallel Load

- Load: control when the data is transferred into a register, and how long it will be stored.



Present state Load	Next state Q_3 Q_2 Q_1 Q_0
0	No change
1	I_3 I_2 I_1 I_0



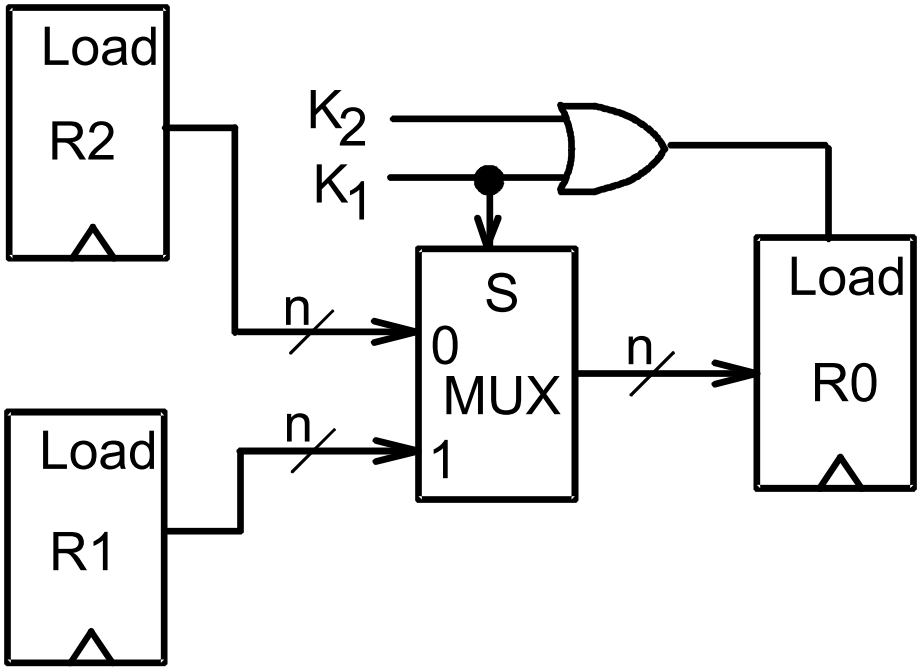
Verilog: 4-Bit Register with Parallel Load

```
module Reg_w_Paraload(clk, rst_n, load, i, y);
input clk;
input rst_n;
input load;
input [3:0] i;
output reg [3:0] y;
reg [3:0] d;

// sequential state DFFs
always @(posedge clk or negedge rst_n)
  if (~rst_n)
    y <= 4'b0;
  else
    y <= d;

// next state function
always @*
  if (load)
    d = i;
  else
    d = y;

endmodule
```

Present state *Load*	*Next state* Q_3 Q_2 Q_1 Q_0
0	*No change*
1	I_3 I_2 I_1 I_0

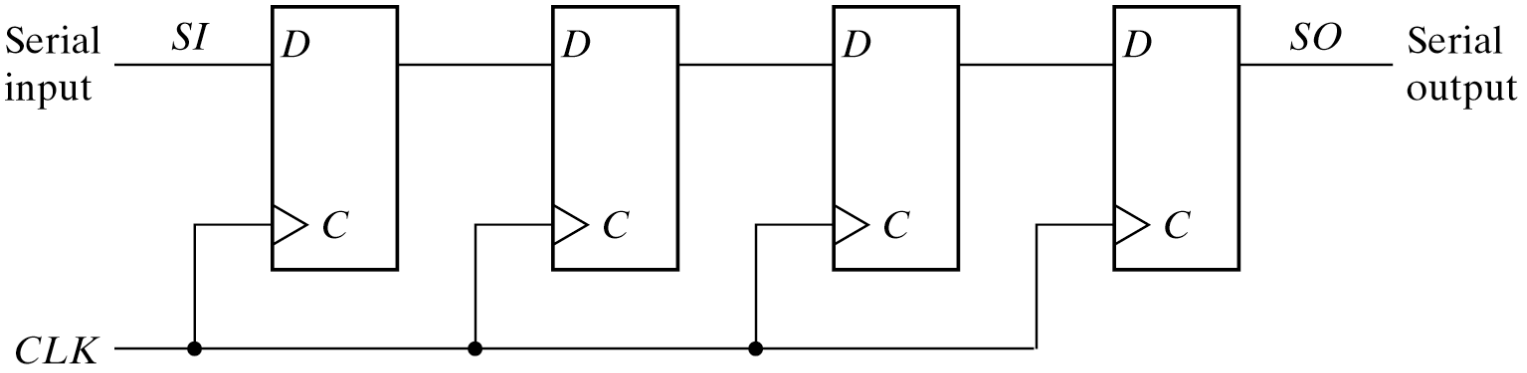
Multiplexer-Based Transfers

- Register may receive data from many different sources at different times.



Shift Registers

- Shift registers move data within the register toward its MSB or LSB position. It shifts one bit at a time.
- In the simplest case, the shift register is simply a set of D flip-flops connected in a row like this:

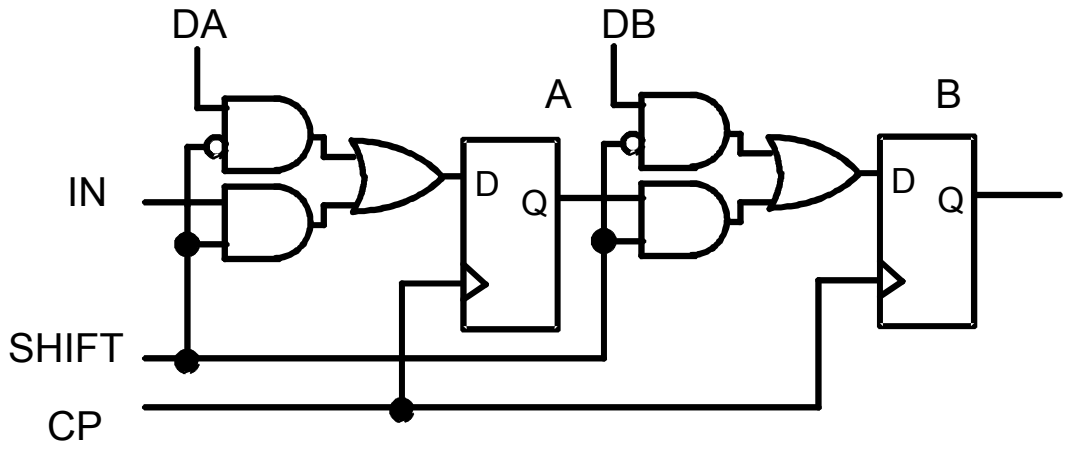


- Data output, SO, is often called
- The vector (A, B, C, SO) is called
- Generally, input and output can be serial or parallel.

Shifter Types

- Logical shifter: shift the number to the left or right and fills empty spots with 0's.
- Arithmetic shifter: same as logical shifter but on right shift fills empty MSBs with the sign bit (sign extension).
- Barrel shifter: rotate numbers in a circle such that empty spots are filled with the bits shifted off the other end.

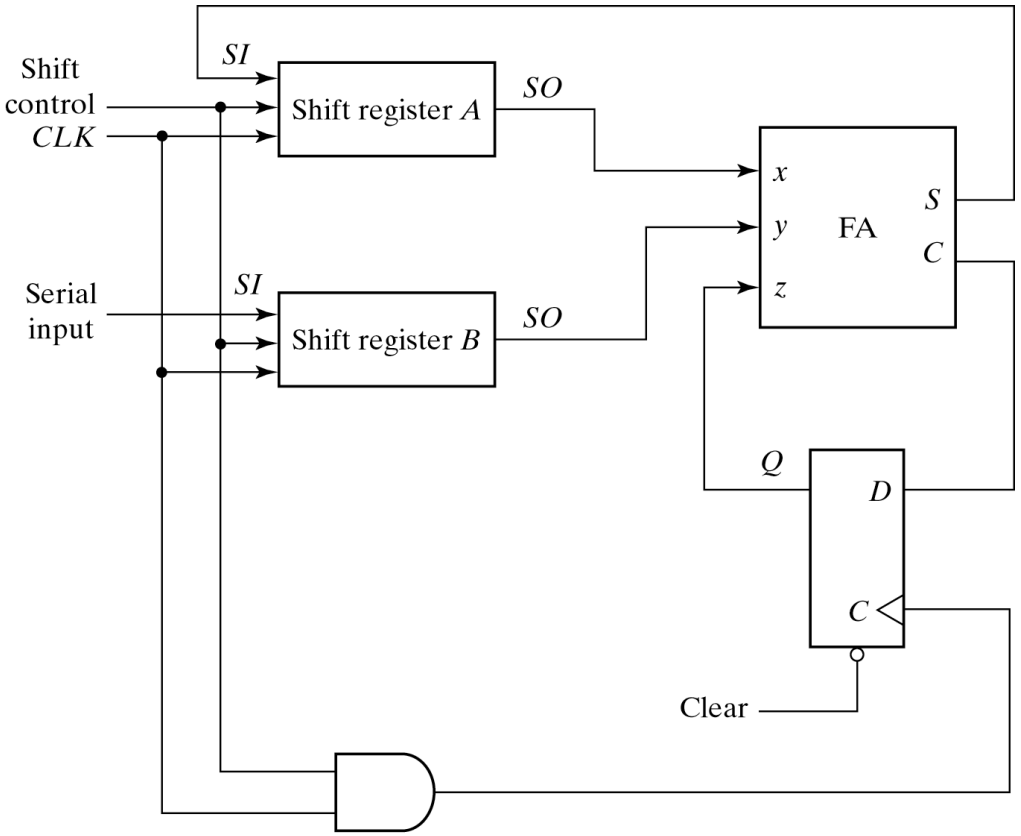
Parallel Load Shift Registers



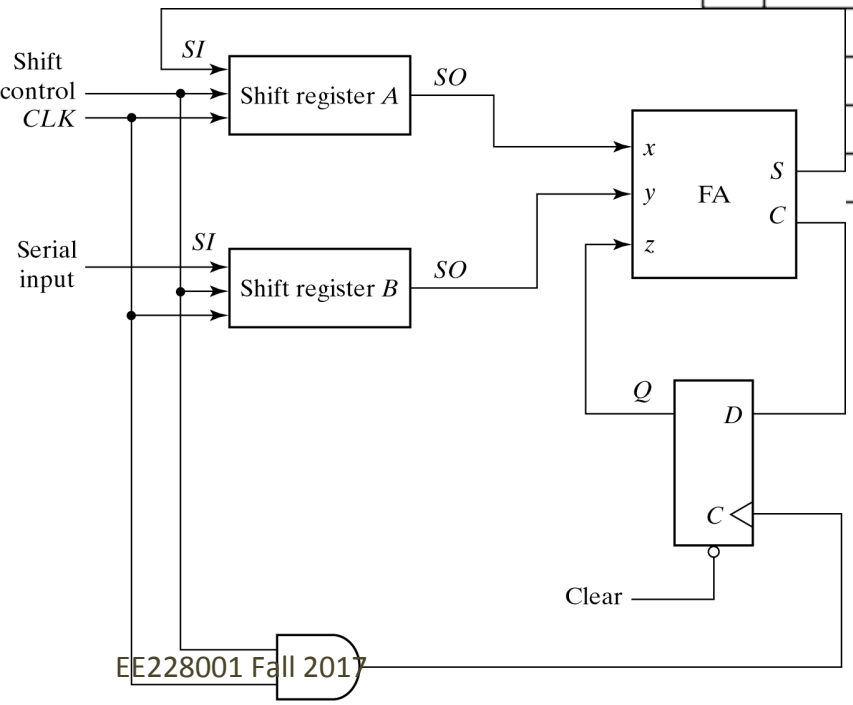
- Add a MUX between each shift register.
- Data can be shifted or loaded.
- When SHIFT = 0,

Serial Adder using DFFs

- Initially, augend in register A and addend in register B.
- Shift control enables the triggering of clock and 1-bit addition of two operands from LSB to MSB.
- A new sum bit is transferred to shift register A.
- A carry-out is transferred to Q.
- Finally, shift control is disabled and the sum is stored in shift register A.

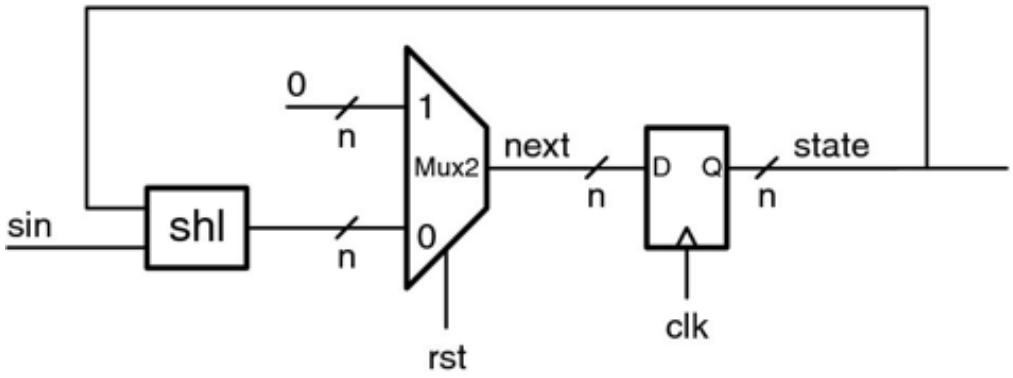


t	SI (B)	B3	B2	B1	B0	SI(A),s	A3	A2	A1	A0	x (A0)	y (B0)	z	c	s
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
3	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
4	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0
5	0	0	1	0	1	1	0	0	0	0	0	1	0	0	1
6	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0
7	0	1	0	0	1	1	0	1	0	0	0	1	0	0	1
8	1	0	1	0	0	0	1	0	1	0	0	0	0	0	0
9	0	1	0	1	0	1	0	1	0	1	1	0	0	0	1
10	0	0	1	0	1	1	1	0	1	0	0	1	0	0	1
11	0	0	0	1	0	1	1	1	0	1	1	0	0	0	1
12	0	0	0	0	1	1	1	1	1	0	0	1	0	0	1
							1	1	1	1	1				
								1	1	1	1				
									1	1	1				
										1	1				



Verilog: Shift Register

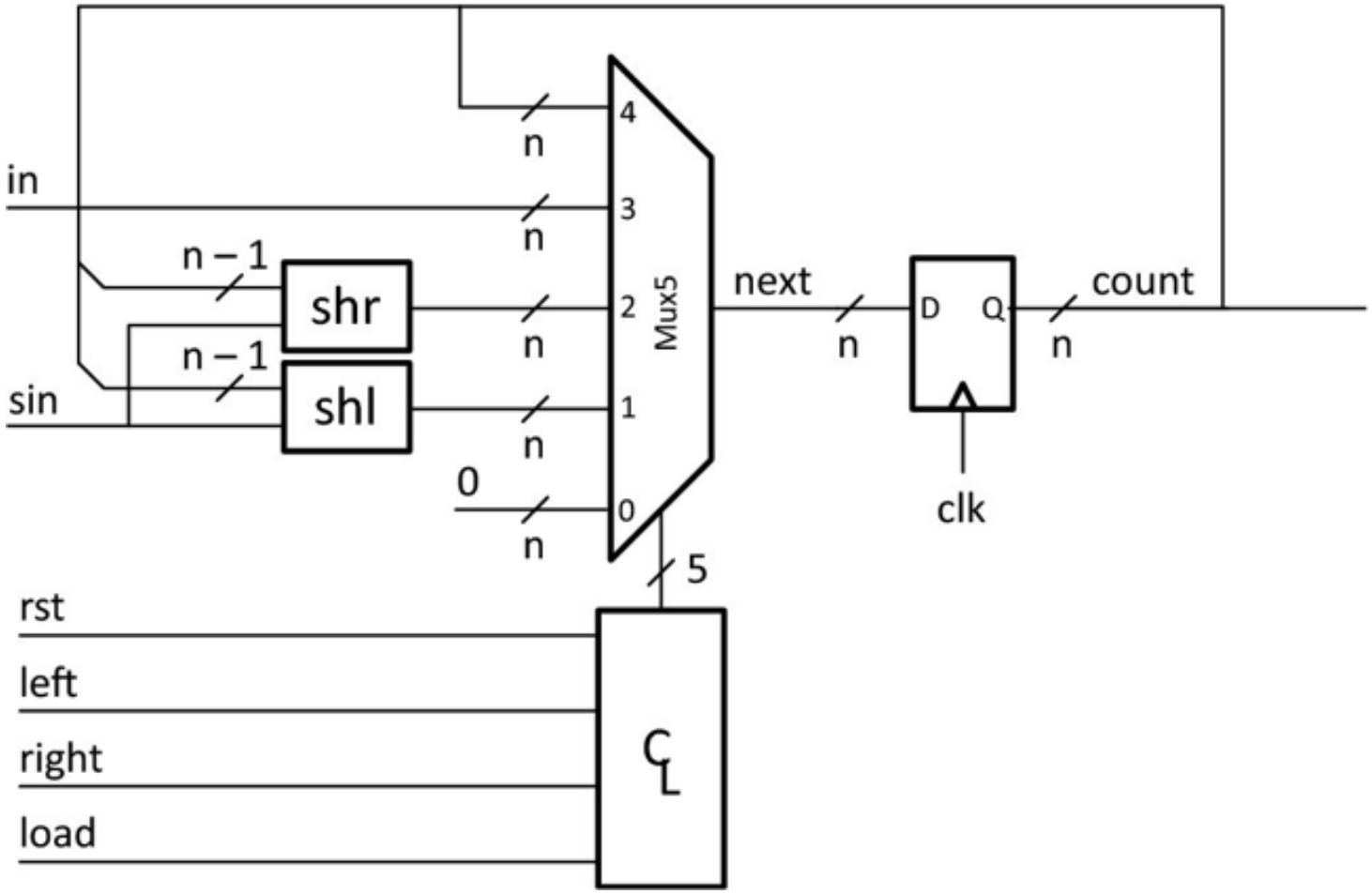
```
module Shifter_Register1(clk, rst, sin, state);
parameter n=4;
input clk;
input rst;
input sin;
output [n-1:0] state;

wire [n-1:0] next = rst ? {n{1'b0}} :
{state[n-2:0], sin};

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

endmodule
```



Verilog: Shift Register (Better Version)

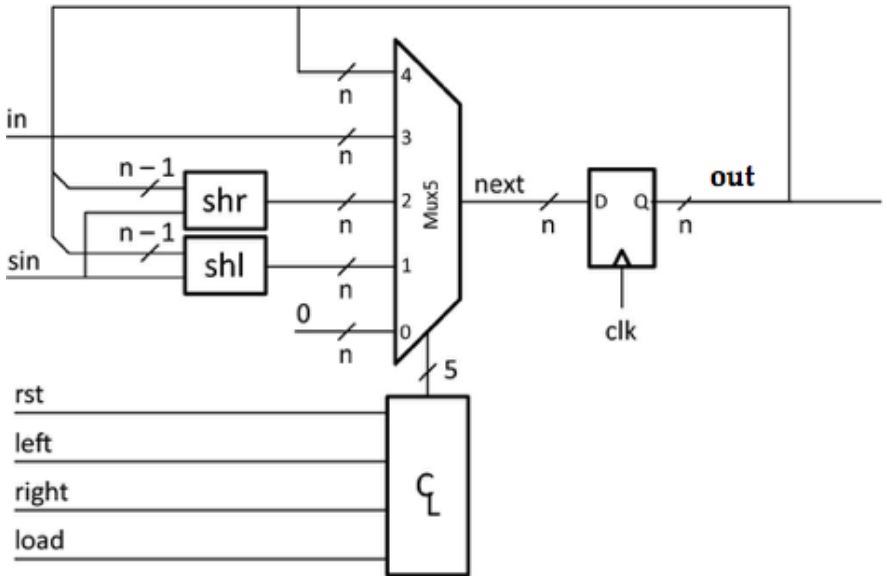

```
module Shifter_Register2(clk, rst, sin, state);
parameter n=4;
input clk;
input rst;
input sin;
output [n-1:0] state;

wire [n-1:0] next = {state[n-2:0], sin};

always @(posedge clk)
  if (rst)
    state <= n'b0;
  else
    state <= next;

endmodule
```

Left/Right/Load Shift Register



Verilog: Left/Right/Load Shift Register

```
module LRL_Shifter_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(clk), .in(next), .out(out));

always @*
  casex({rst,left,right,load})
    4'b1xxx: next = n'b0;                  //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

endmodule
```



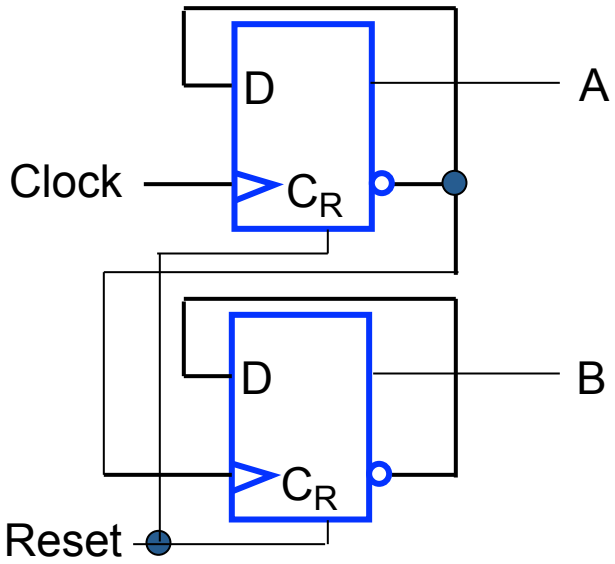
Counters

Counters

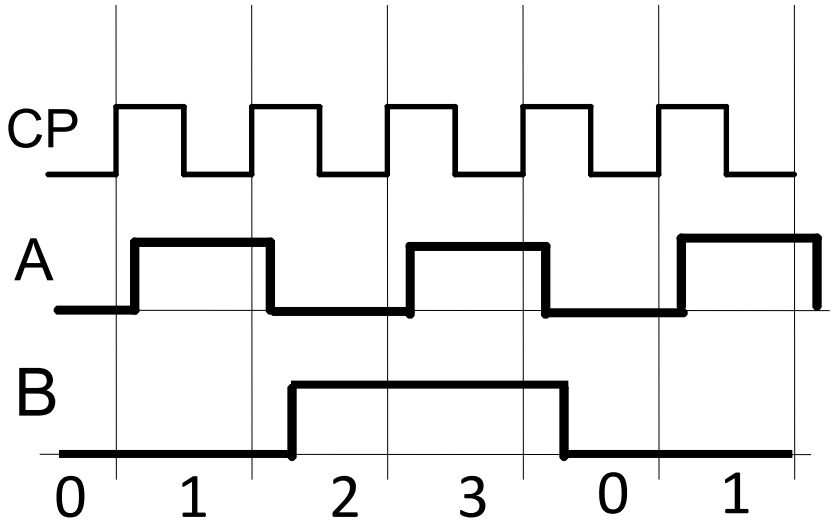
- Counters are sequential circuits which "count" through a specific state sequence.

- Two distinct types are in common usage:
  - Ripple Counters
    - The output transition of flip-flop serves as a source for triggering other flip-flops
  - Synchronous Counters
    - The clock inputs of all flip-flops receive a common clock

Ripple Counter

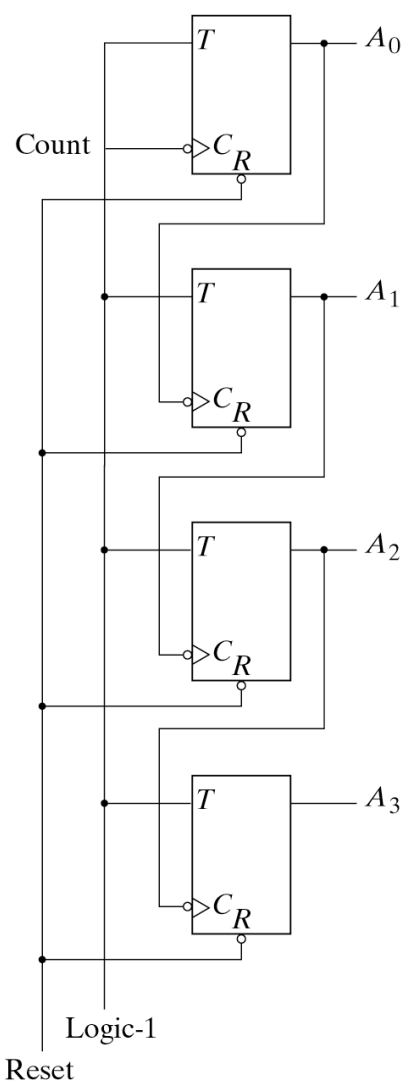


- Clock only connects to the LSB FF.
- At positive edge,
- The clock input of DFF_B is

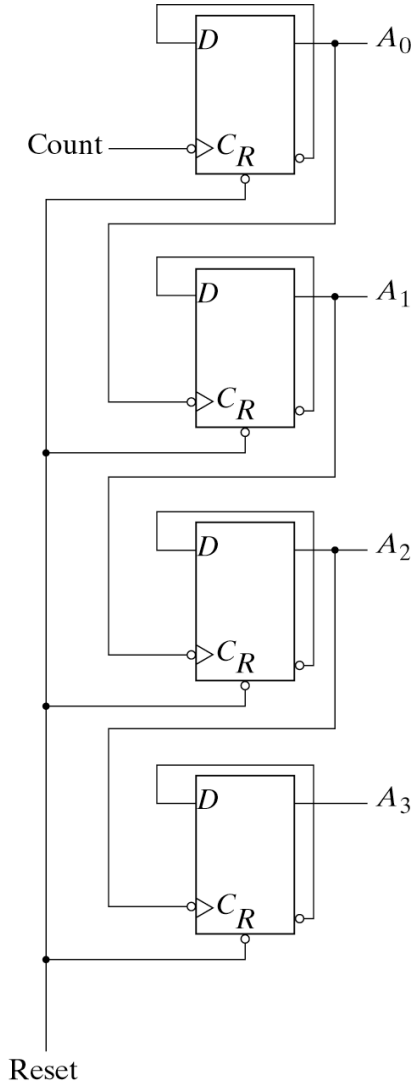


2-Bit Downward Ripple Counter Example

4-Bit Ripple Counter (1/3)



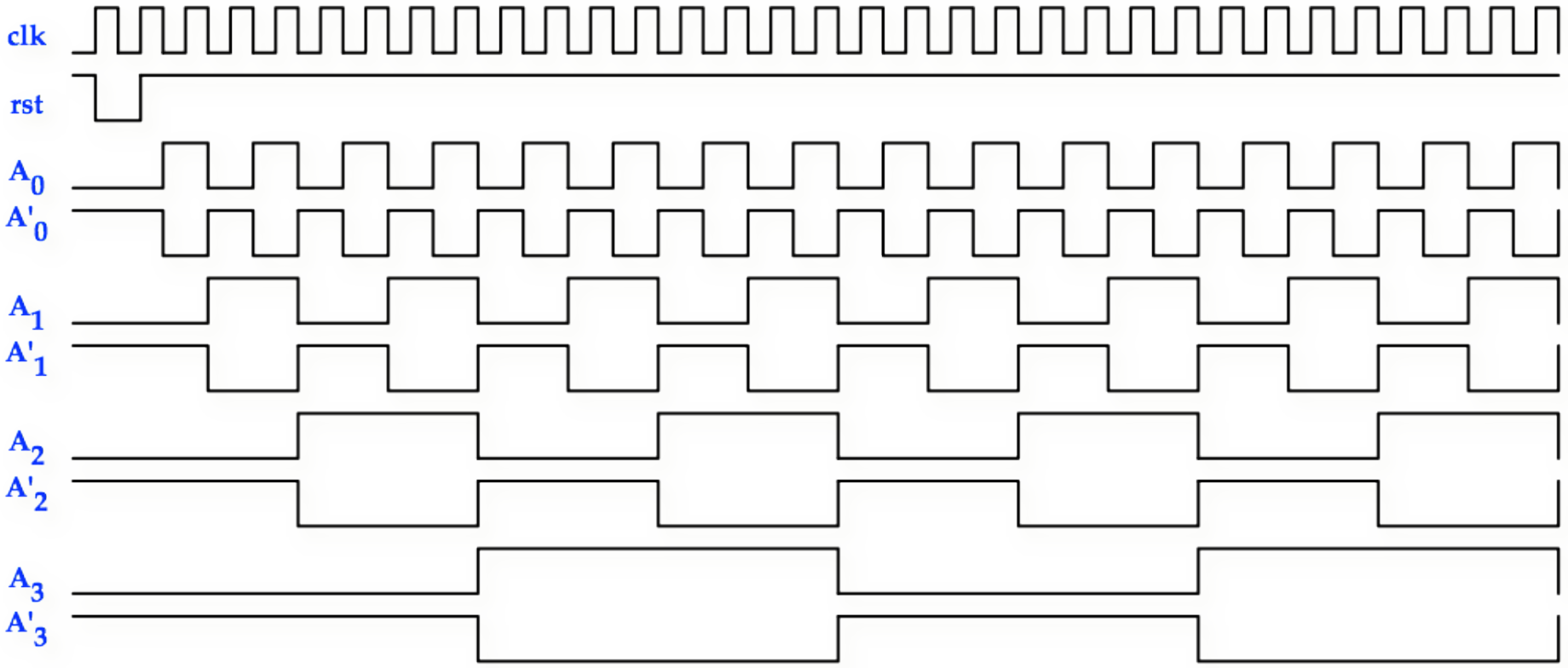
(a) With T flip-flops



(b) With D flip-flops

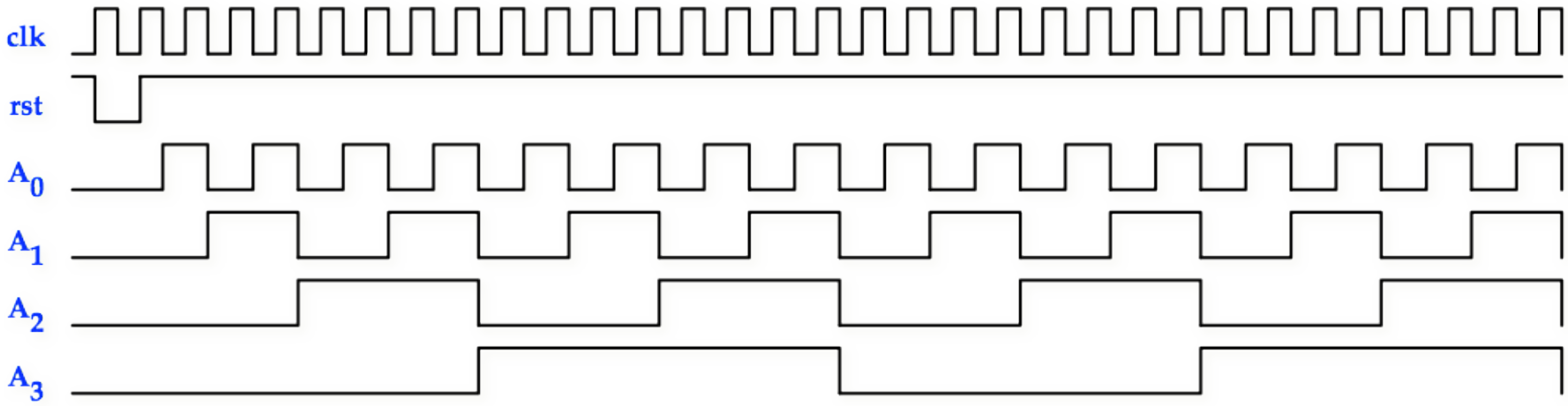
4-Bit Ripple Counter (2/3)

- Use DFFs



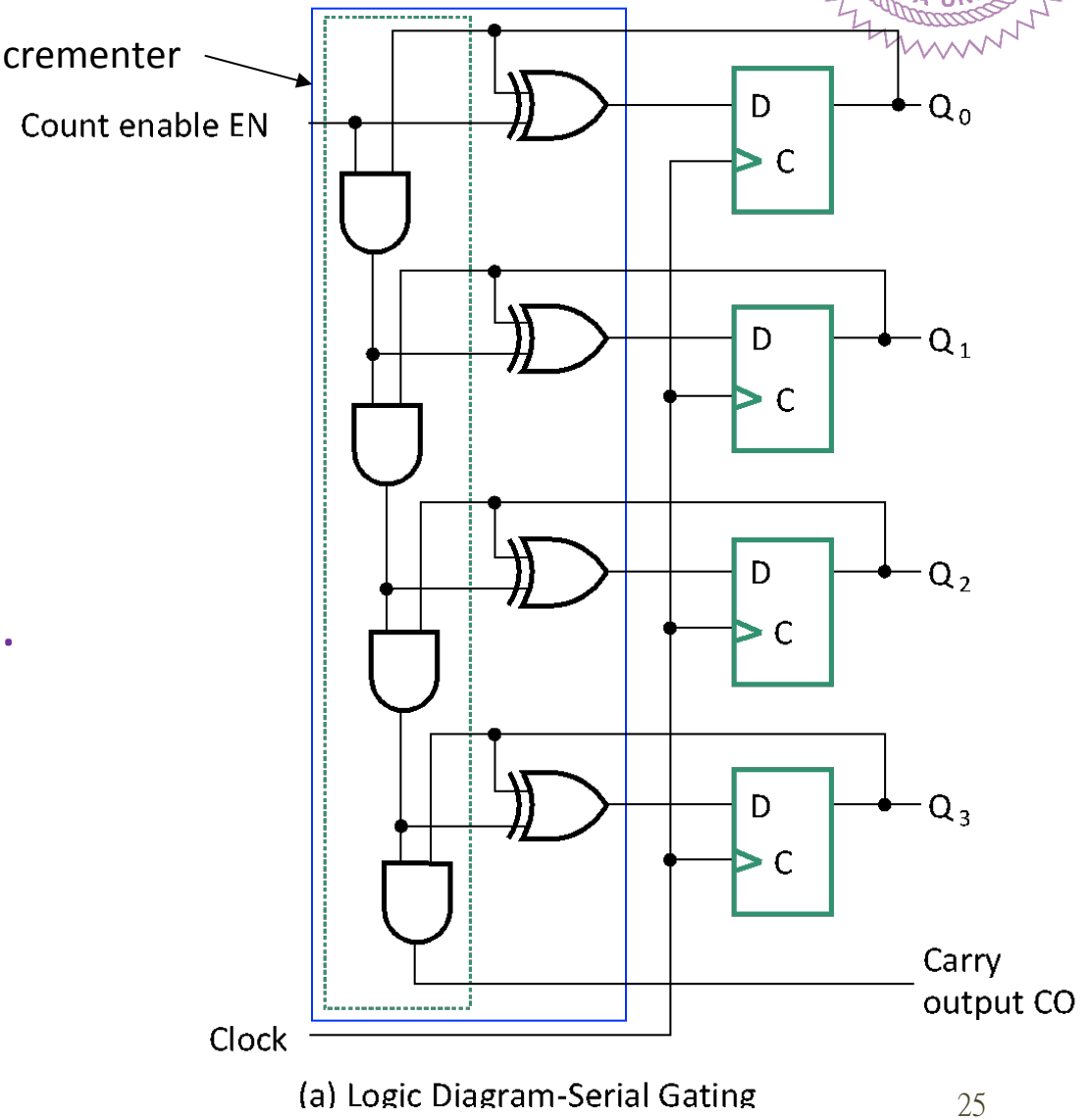
4-Bit Ripple Counter (3/3)

- Down counter with TFFs
  - Replace active low clock with active high clock



Synchronous Counters (1/2)

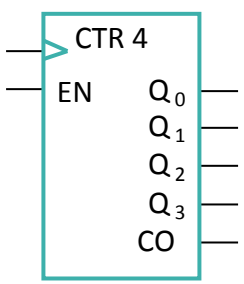
- Clock is applied to all FFs.
- Formed by incrementer and DFFs.
- AND chain causes complement of a bit if all bits toward LSB from it equal 1.
- XOR complements each bit.
- Count enable (EN)



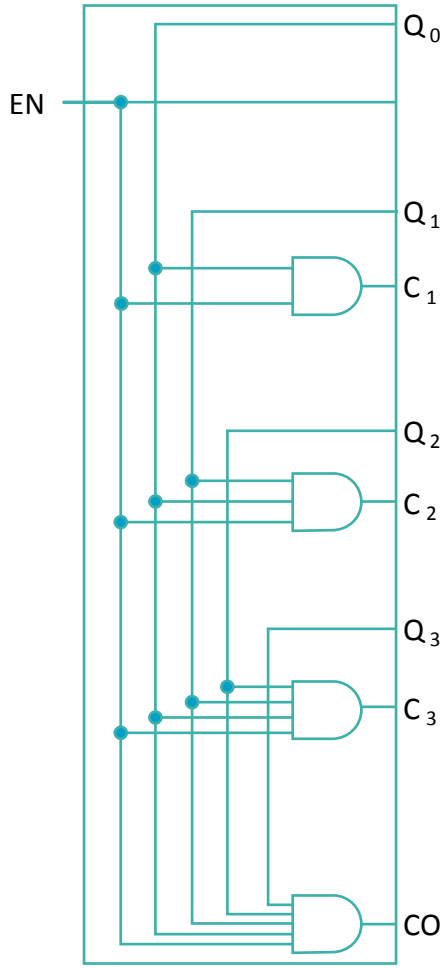
(a) Logic Diagram-Serial Gating

Synchronous Counters (2/2)

- Replace AND carry chain with ANDs in parallel.
- EN applied at each bit.
- Parallel gating reduce path delay.



Symbol



Logic Diagram-Parallel Gating

Up/Down/Load (UDL) Counter

- Counter function includes
  - Up counting
  - Down counting
  - Load arbitrary value
  - Hold the value
- Assume {up, down, load} is one-hot, rst has the highest priority.
  - rst: next=0
  - !rst&up: next=out+1
  - !rst&down: next=out-1
  - !rst&load: next=in
  - else: next=out

UDL Counter State Table

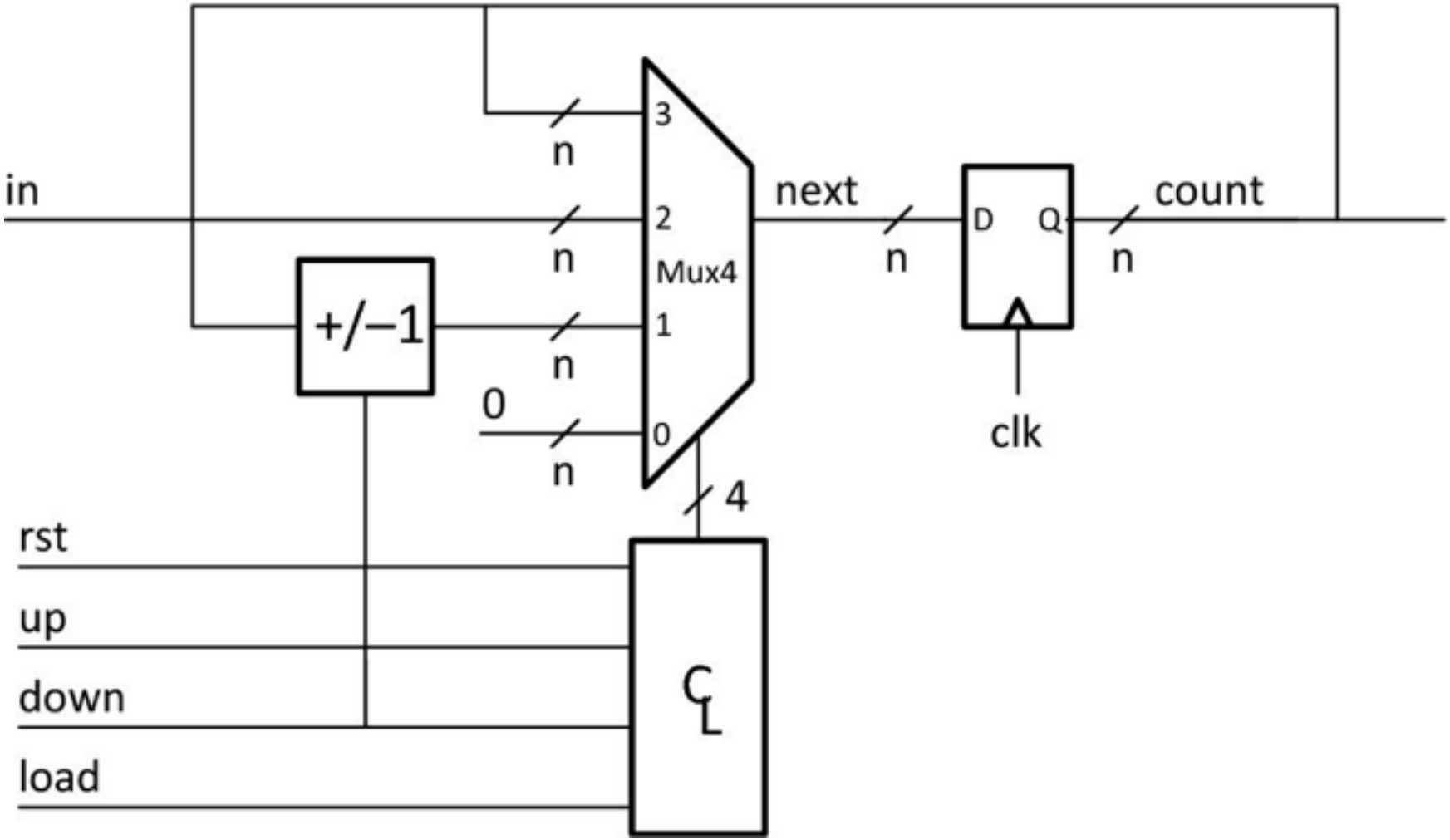
original state table

State	Next State			
	rst=1	up=1	down=1	load=1
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 state table

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

UDL Counter Schematic



Verilog: 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;
reg [n-1:0] next;

DFF #(n) count(.clk(clk), .in(next), .out(out));
always @*
  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;
    default: next = out;
  endcase

endmodule
```

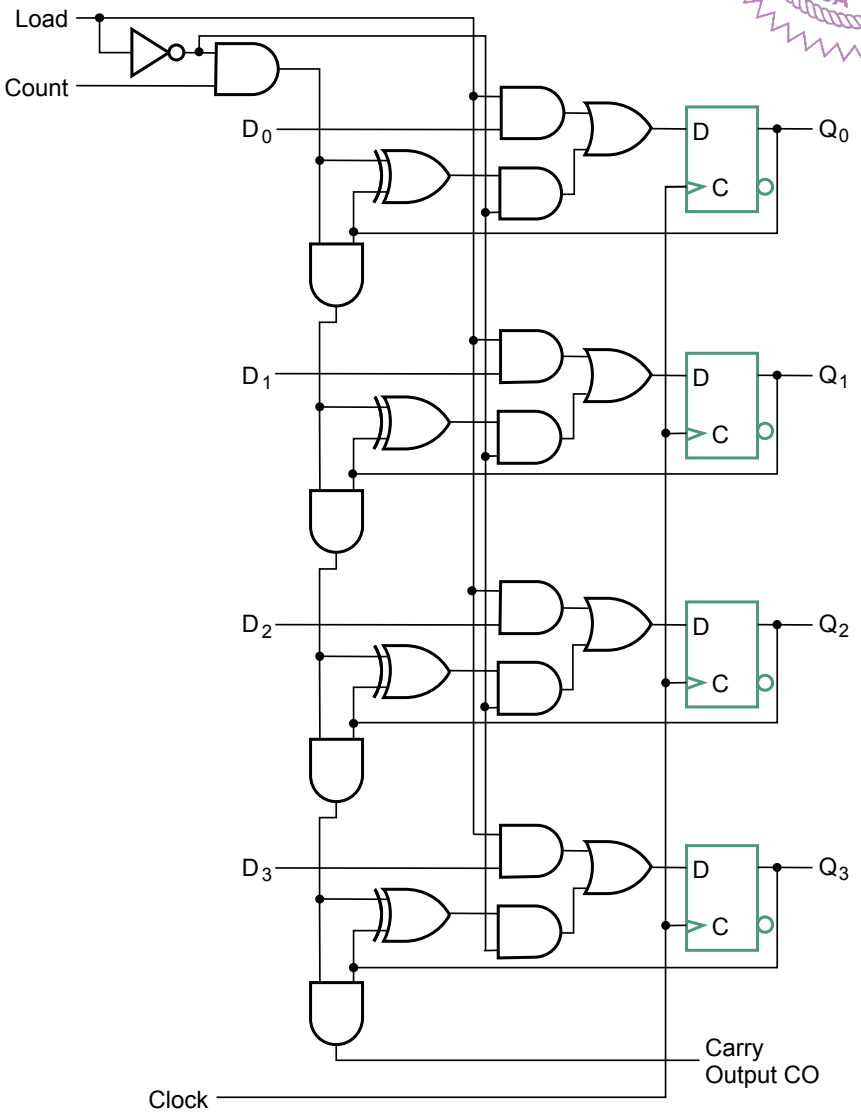
Verilog: UDL Counter (Better Version)

```
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] outpm1;
reg [n-1:0] next;

always @(posedge clk)
 if (rst)
   out <= {n{1'b0}};
 else
   out <= next;
```

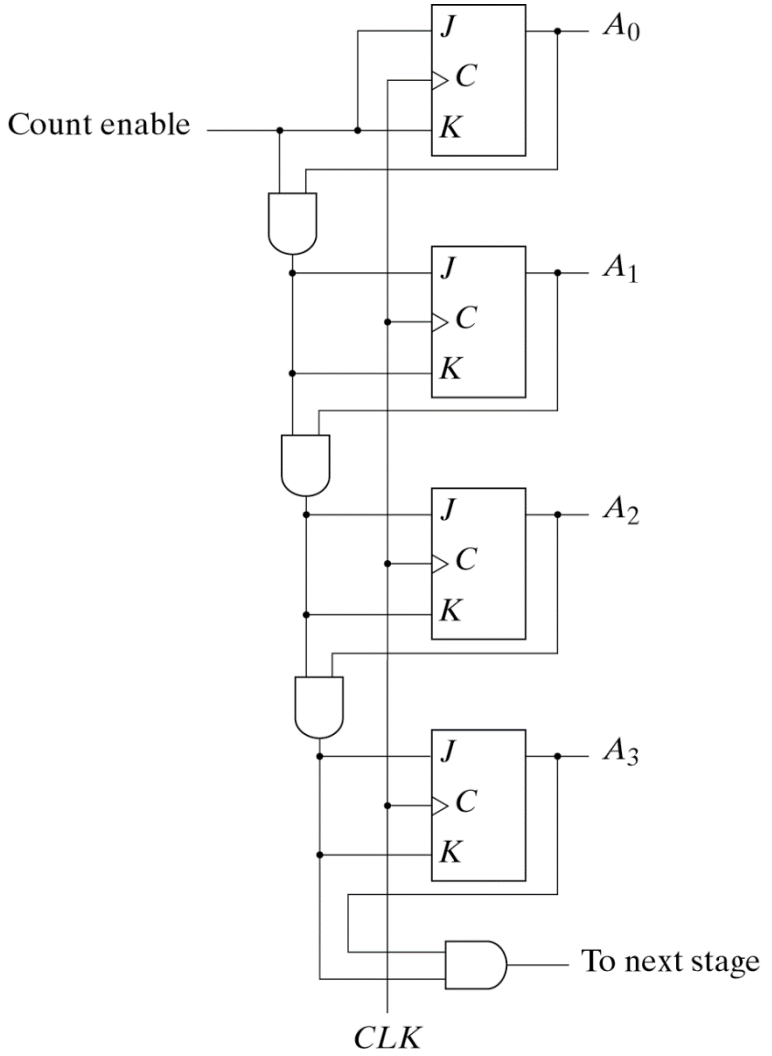
```
always @*
 casex({up, down, load})
   3'b100: next = out + 1'b1;
   3'b010: next = out - 1'b1;
   3'b001: next = in;
   default: next = out;
 endcase

endmodule
```

Counter with Parallel Load

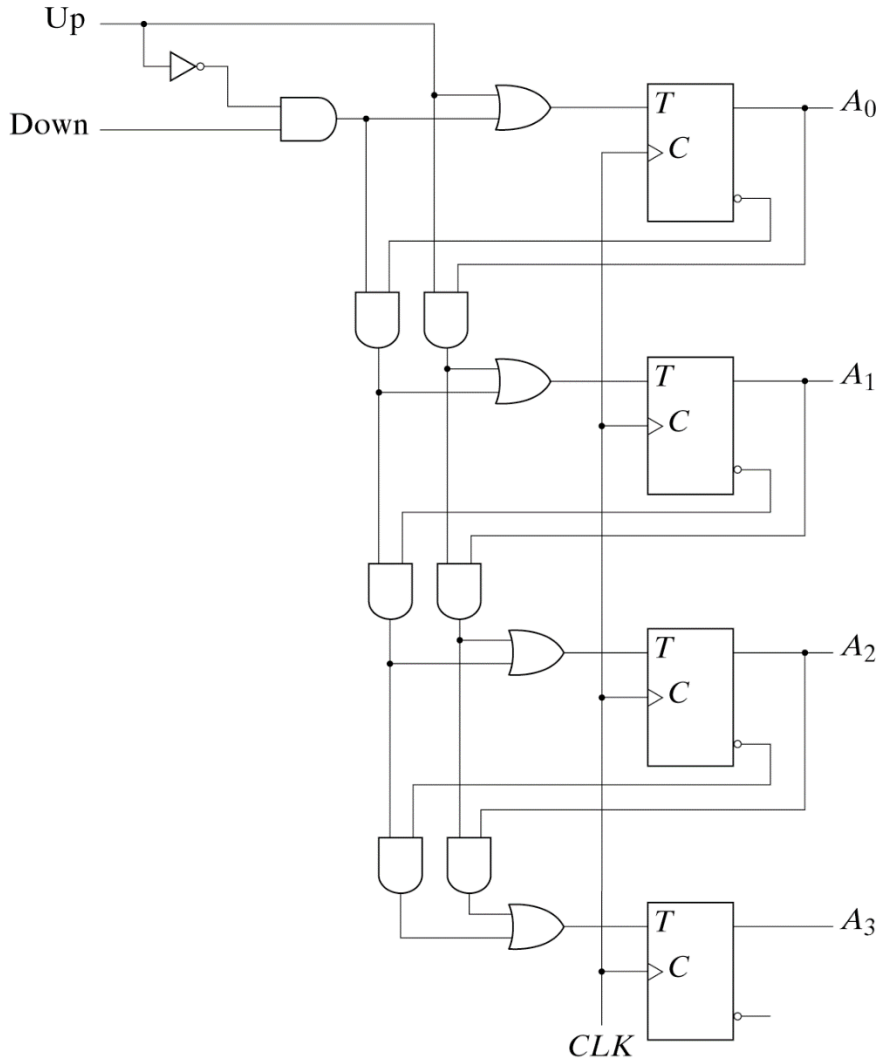
- Add path for input data.
  - Enabled for Load = 1
- Add logic to
  - Disable count logic for Load = 1
  - Disable feedback from outputs for Load = 1
  - Enable count logic for Load = 0 and Count = 1
- The resulting function table:



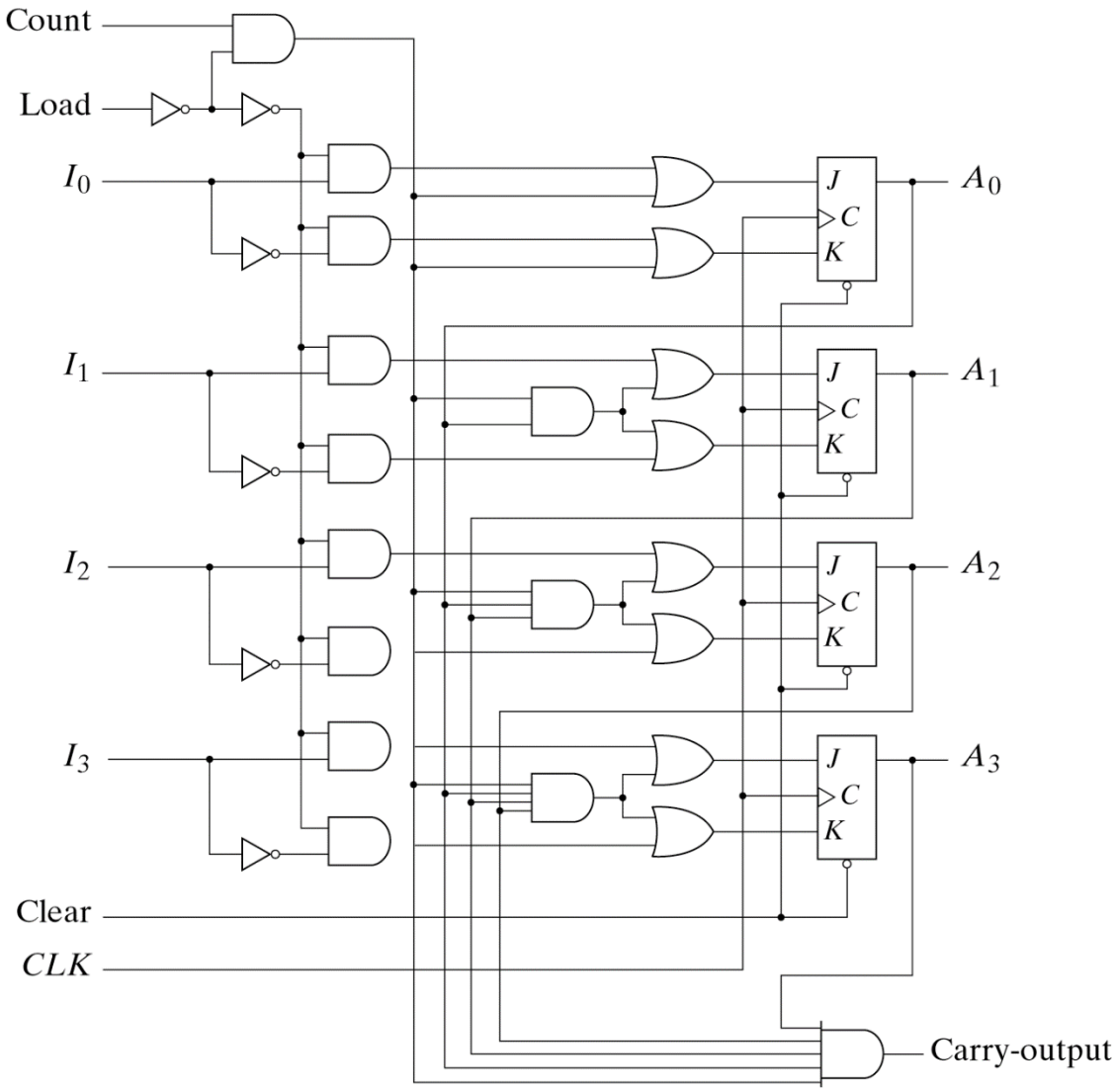
4-Bit Synchronous Binary Counter (1/2)



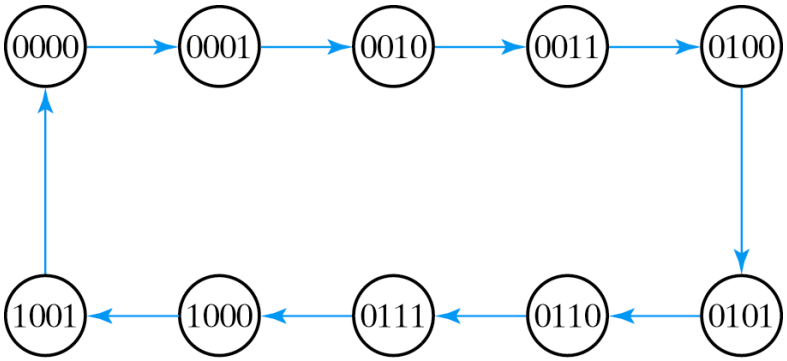
4-Bit Synchronous Binary Counter (2/2)



4-Bit Binary Counter with Parallel Load



BCD Counter (1/2)



Present State				Next State			
Q8	Q4	Q2	Q1	Q8	Q4	Q2	Q1
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0

BCD Counter (2/2)

Ring Counter

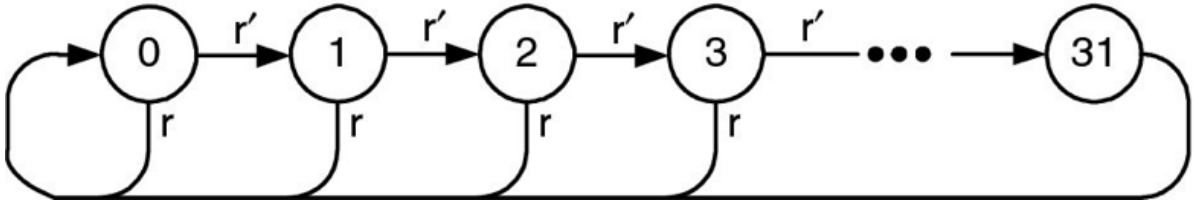
- A circular shift register with only one flip-flop being set at any particular time, all others are cleared. (initial value 1000...000)
- The single bit is shifted from one flip-flop to the next to produce the sequence of timing signals

A_2	A_2	A_1	A_0
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1
1	0	0	0

Johnson Counter

5-Bit Counter

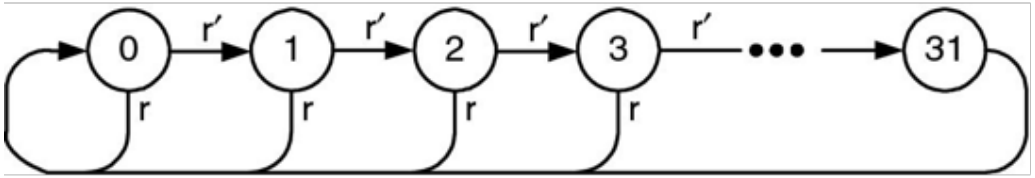
```
module Counter1(clk, rst, out);
input clk, rst;
output [4:0] out;
reg [4:0] next;

DFF #(5) cnt(.clk(clk), .in(next), .out(out));

always @*
  casex({rst,out})
    6'b1xxxxx: next = 5'b0;
    6'd0: next = 5'd1;
    6'd1: next = 5'd2;
    6'd2: next = 5'd3;
    6'd3: next = 5'd4;
    6'd4: next = 5'd5;
    6'd5: next = 5'd6;
    6'd6: next = 5'd7;
    …
    6'd30: next = 5'd31;
    6'd31: next = 5'd0;
    default: next = 5'd0;
  endcase
endmodule
```



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

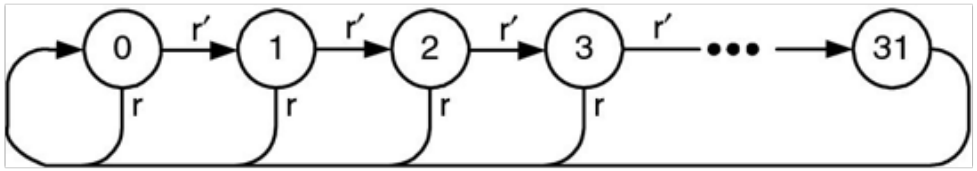
Datapath Implementation

- You can describe the next-state function with a table or you can compactly describe it by an expression: next = r ? 0 : out +1



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

Verilog: Counter




```
module Counter2(clk, rst, out);
parameter n=5;
input clk, rst;
output [n-1:0] out;

wire [n-1:0] next = rst ? 0 : out + 1'b1;

//DFF #(n) cnt(.clk(clk), .in(next), .out(out));
always @(posedge clk)
  out <= next;

endmodule
```

state table

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

Make table symbolic

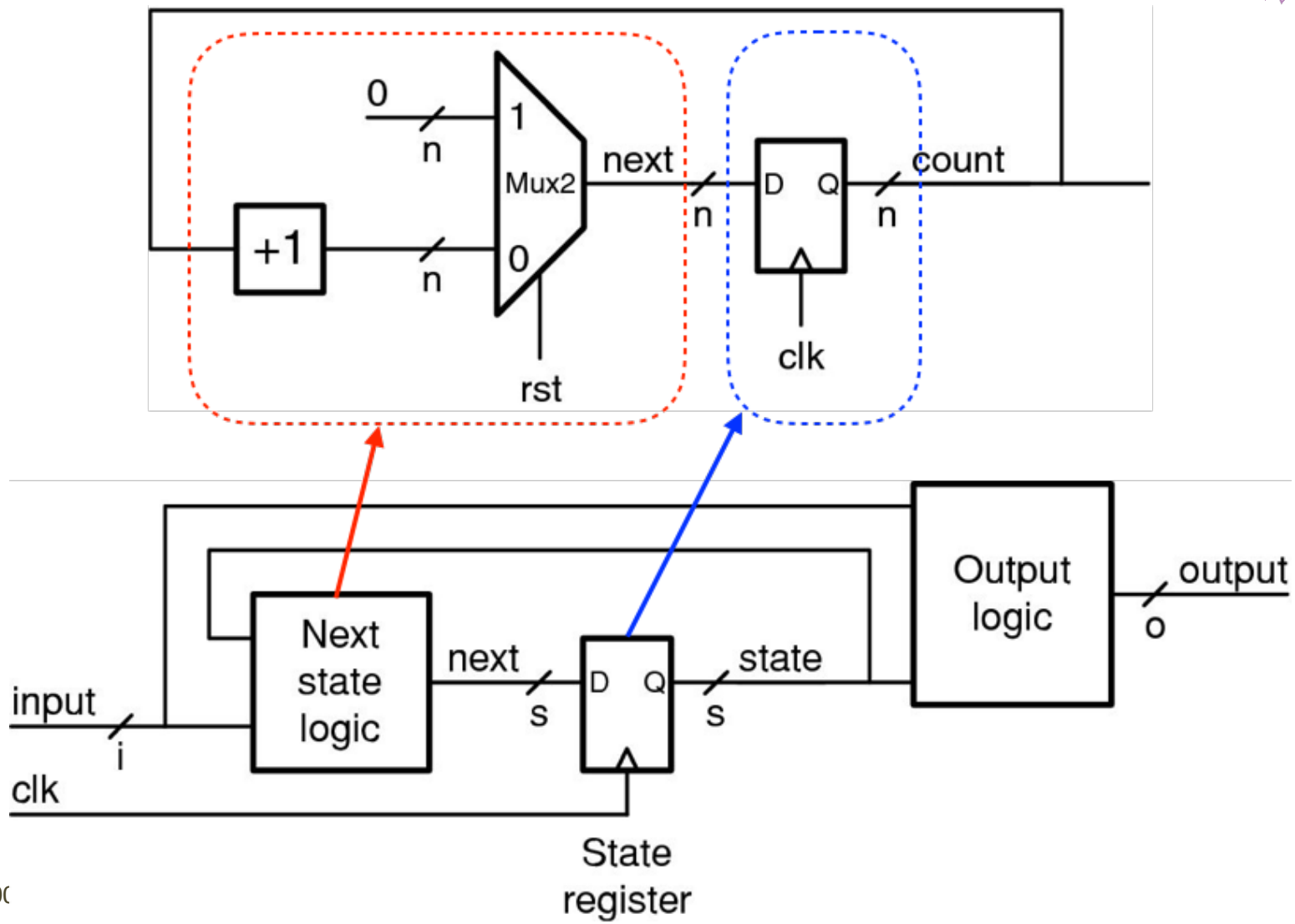
State	Next State	
	~rst	rst
a	a+1	0

symbolic state table

Verilog: Counter (Better Version)


```
module Counter3(clk, rst, out);
parameter n=5;
input clk, rst;
output [n-1:0] out;

wire [n-1:0] next = out + 1'b1;

always @(posedge clk)
  if (rst)
    out <= n'b0;
  else
    out <= next;

endmodule
```

Schematic of 5-Bit Simple Counter



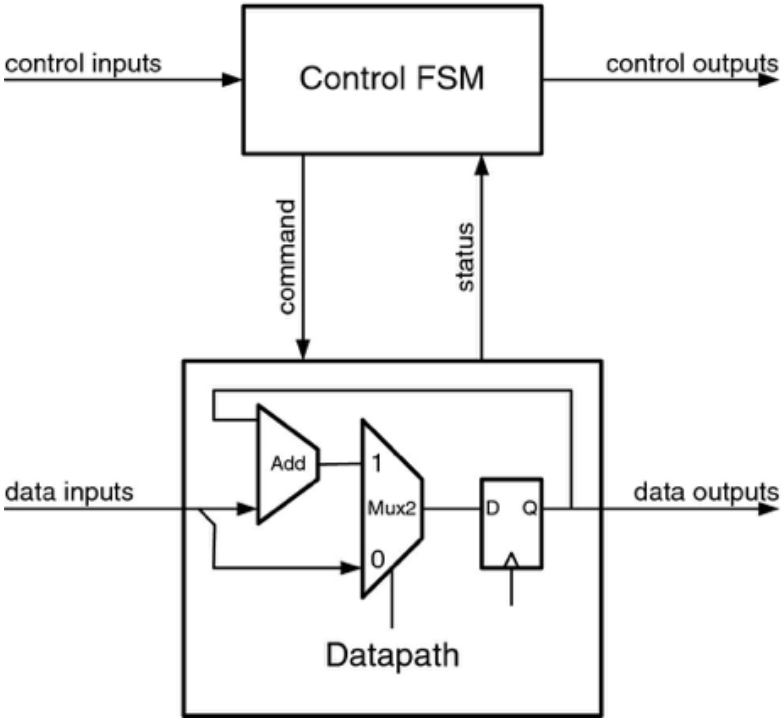
Counters with Unused States

- n flops => 2^n states
- Unused states
  - States that are not used in specifying the FSM, may be treated as don't-care conditions or may be assigned specific next states.
- Self-correcting counters
  - Ensure that when a circuit enter one of its unused states, it eventually goes into one of the valid states after one or more clock pulses so that it can resume normal operation.
    - Analyze the circuit to determine the next state from an unused state after it is designed.

Control and Data Partitioning

Datapath and Control Partitioning

- Datapath: main part of the system determined by a function.
- Control: behavior control determined by state diagram and state table.



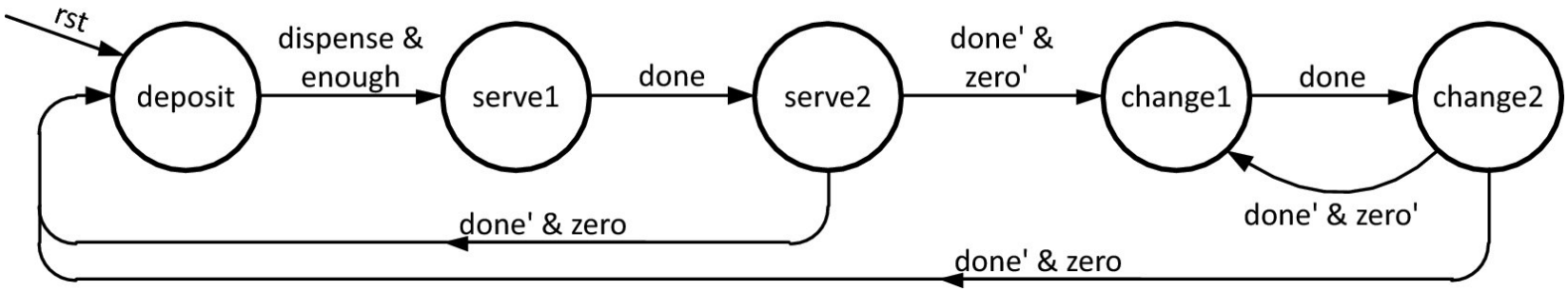
Design Example: Vending Machine

- Function
  - Receives coins (nickel, dime, quarter) and accumulates sum.
  - When dispense button is pressed, serves a drink if enough coins have been deposited.
  - Then returns changes — one nickel a time.
- Partition the tasks
  - Datapath: keep track of amount owed user.
  - Control: keep track of sequence — deposit, serve, change.

Vending Machine (Detailed Behavior)

- Accepting price
- Whenever a coin is deposited
  - A one-cycle pulse on one of nickel, dime, or quarter
  - Adding amount
- After sufficient coins
  - Assert enough
- After enough is asserted, user presses a dispense button
  - Assert serve for exactly one cycle to serve the soft drink
  - Wait until done is asserted
  - Wait for done to go low
  - Return change if any (when zero !=0), one nickel a time
    - Assert change for exactly one cycle
    - Wait for done is asserted
    - Wait done to go low
    - Return the next nickel (when zero !=0), or go to initial state when no changes is needed (zero ==0)

Vending Machine State Diagram

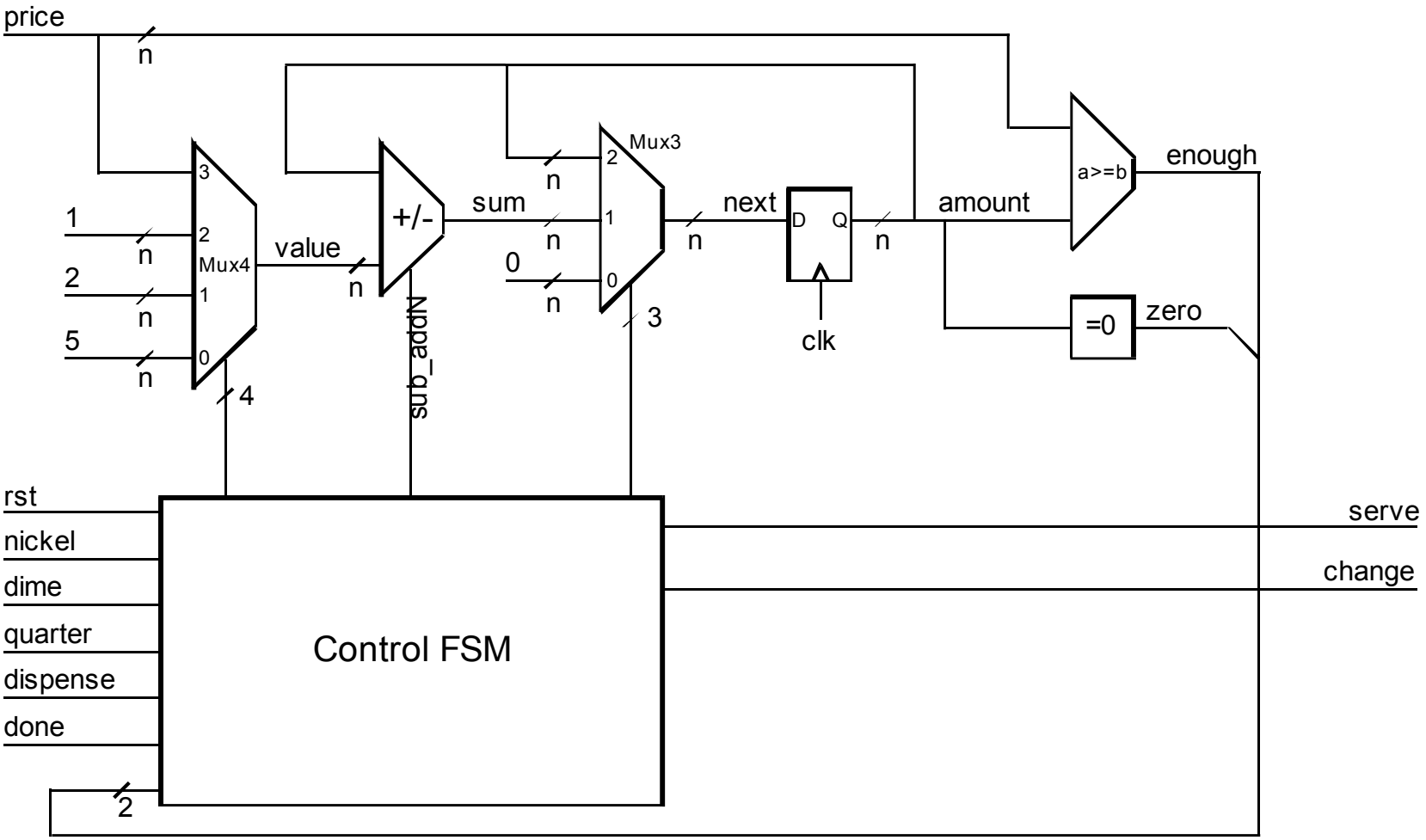


- Control inputs
  - rst, nickel, dime, quarter, dispense, done
- Data inputs
  - price
- Control outputs
  - serve, change
- Status signals
  - enough, zero

Vending Machine Data State

- The amount of money owed to user (unit: nickel)
  - Reset: ***amount*** is 0
  - Deposit a coin: ***amount*** <- ***amount*** + value
  - Serve a drink: ***amount*** <- ***amount*** - value
  - Return the nickel of change: ***amount*** <- ***amount*** -1
  - Otherwise: no change in ***amount*** (hold)
- Data input: ***price***
- State output:
  - enough: ***amount*** > ***price***
  - zero: ***amount*** == 0

Vending Machine Block Diagram



Signals of Control FSM and Datapath

- Control inputs
  - rst, nickel, dime, quarter, dispense, done
- Data input
  - price
- Control outputs
  - serve, change
- Status signals
  - enough, zero
- Commands
  - selval, sub, seltext

Verilog: Vending Machine (Global Define)

```
//-----------------------------------------
// define state assignment – binary
//-----------------------------------------
`define SWIDTH  3
`define DEPOSIT 3'b000
`define SERVE1  3'b001
`define SERVE2  3'b011
`define CHANGE1 3'b010
`define CHANGE2 3'b100
`define DWIDTH  4
`define NICKEL  4'h01
`define DIME    4'h02
`define QUARTER 4'h05
`define PRICE   4'h0b
```

Verilog: Vending Machine (Top Module)

```
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
```

Verilog: Vending Machine (Control Module) (1/3)


```
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;  // one-hot select signal
 output [2:0] selnext; // one-hot select signal
 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 ;
```

Verilog: Vending Machine (Control Module) (2/3)


```
// 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 & first);
 wire [2:0] selnext = {~(selv | rst), ~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);
```

Verilog: Vending Machine (Control Module) (3/3)

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

 // next state logic
 always @*
  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
```

Verilog: Vending Machine (Datapath) (1/2)


```
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
```

Verilog: Vending Machine (Datapath) (2/2)


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

 // select next state from 0, sum, or hold
 Mux3 #(n) nsmux(amout, sum, {n{1'b0}}, 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(price, `QUARTER, `DIME, `NICKEL, selval, value) ;

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

Simulation Waveforms