JOINT INSTITUTE
交大密西根学院

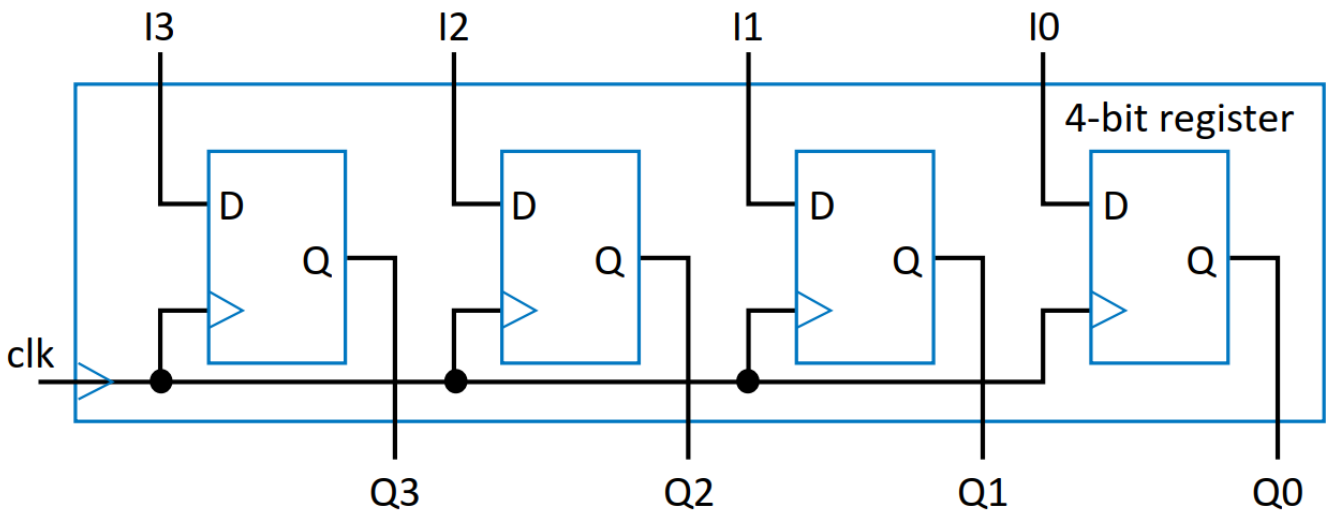
ECE2700J SU24 Final RC

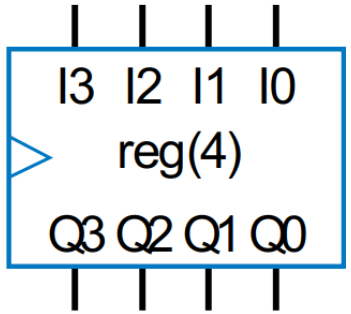
C9-13

Wenyue Li
8/1/2024

Register & Shifter

Basic register

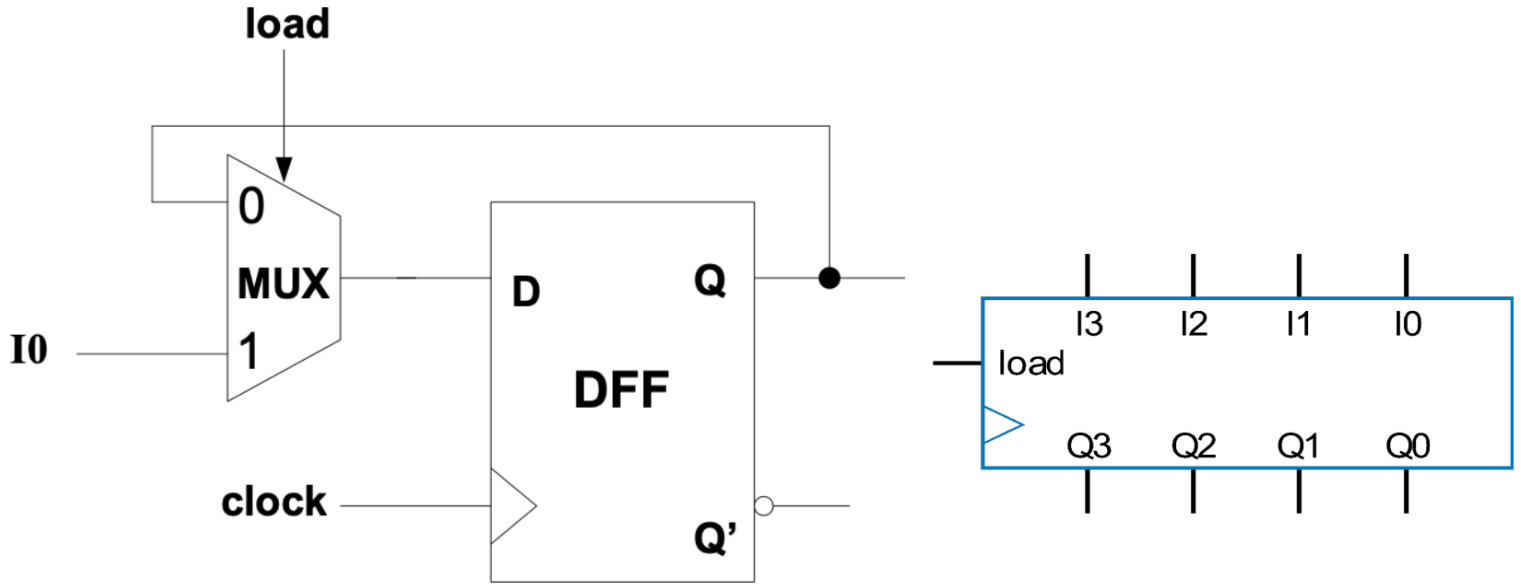




- Can store data, very common in datapath
- Basic register: Loaded every cycle
- Basic Register & DFF

Register with parallel load

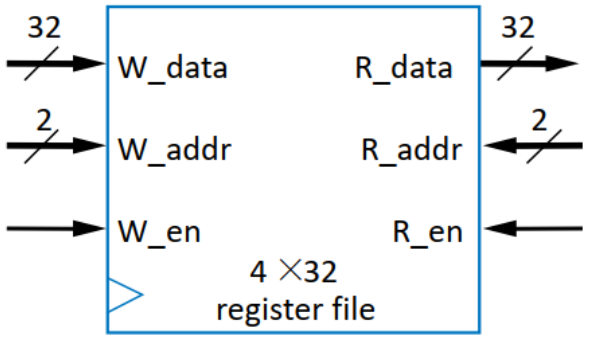
- Add 2x1 mux to each flip-flop
- Register's load input selects mux input to pass
  - Either existing flip-flop value, or new value to load

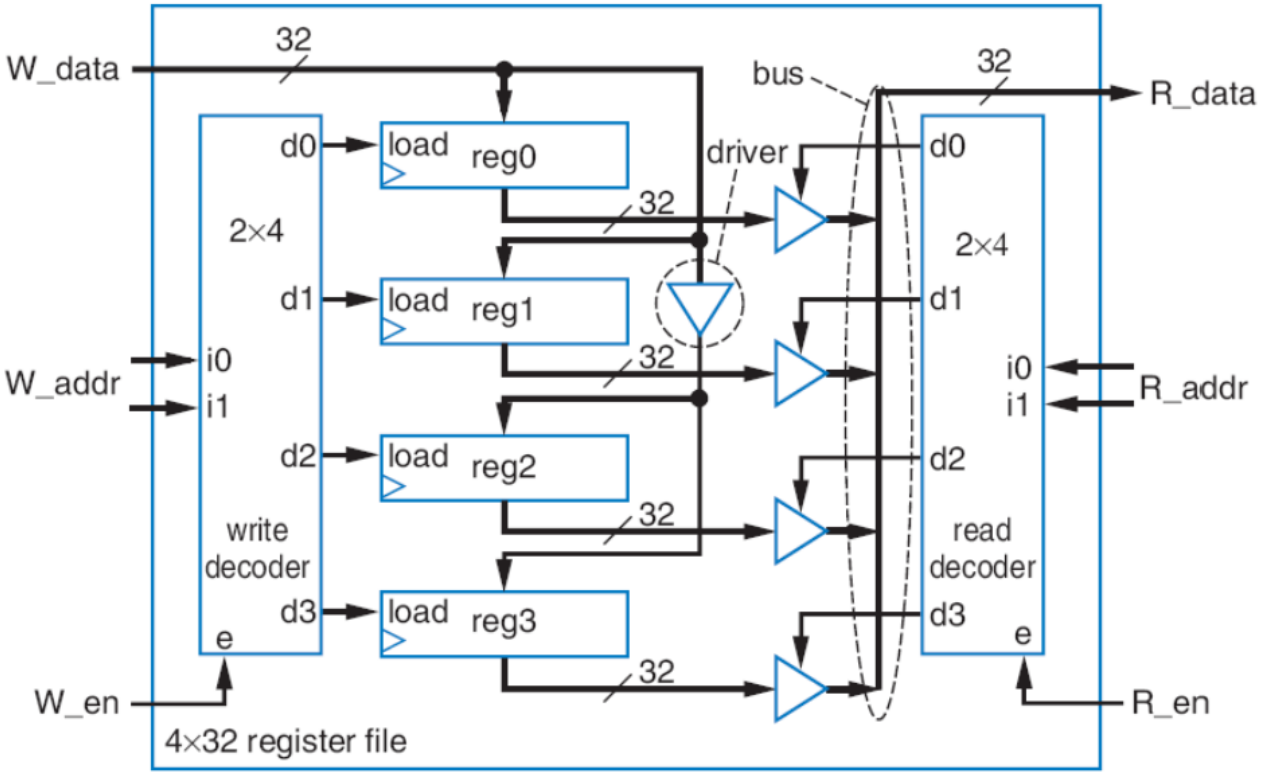


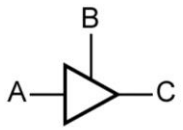
Synchronous active high Load

Register & Shifter

Register file







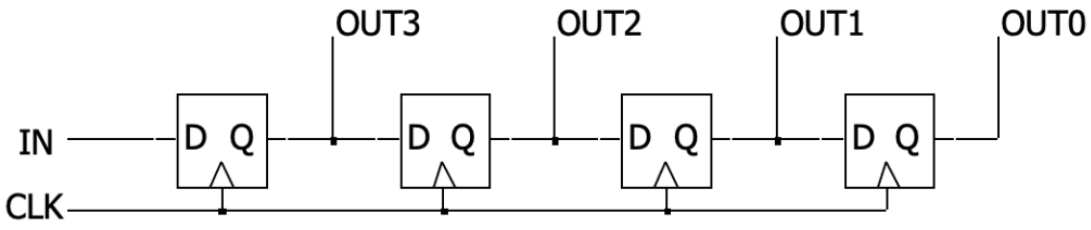
B	A	C
0	0	Z
0	1	Z
1	0	0
1	1	1

- A buffer is a one directional transmission logic device
  - somewhat like a NOT gate without complementing the binary value
  - amplify the driving capability of a signal
  - insert delay
  - Protect input from output

Register & Shifter

Shifter register

- One type of register
  - Stores binary data
  - Stored data can be shifted right (MSB >> LSM) or left (MSB << LSB)
  - Example: Shift right per clock edge
- Implementation:
  - Connect Q output of one flip flop to the D input of the next flip flop
  - 4-bit shift register

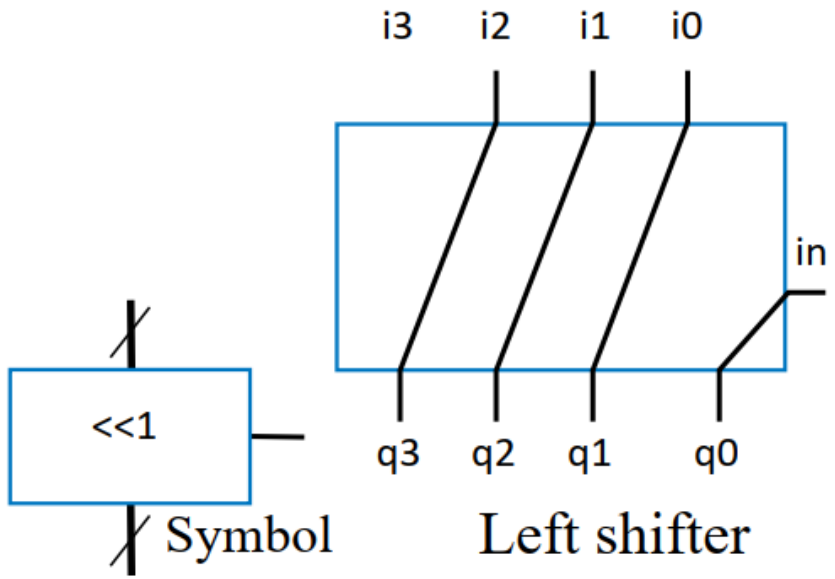


	IN	OUT(3:0)
Initial value:	0	0110
rising edge:	0	0011
rising edge:	0	0001
rising edge:	0	0000
rising edge:	1	1000
rising edge:	0	0100

Register & Shifter

Shifter

- Combinational Datapath component
- Shifting (e.g., left shifting 0011 yields 0110) useful for:
  - Manipulating bits
  - Shift left once is same as multiplying by 2 (0011 (3) becomes 0110 (6))
  - Shift right once same as dividing by 2



Register & Shifter

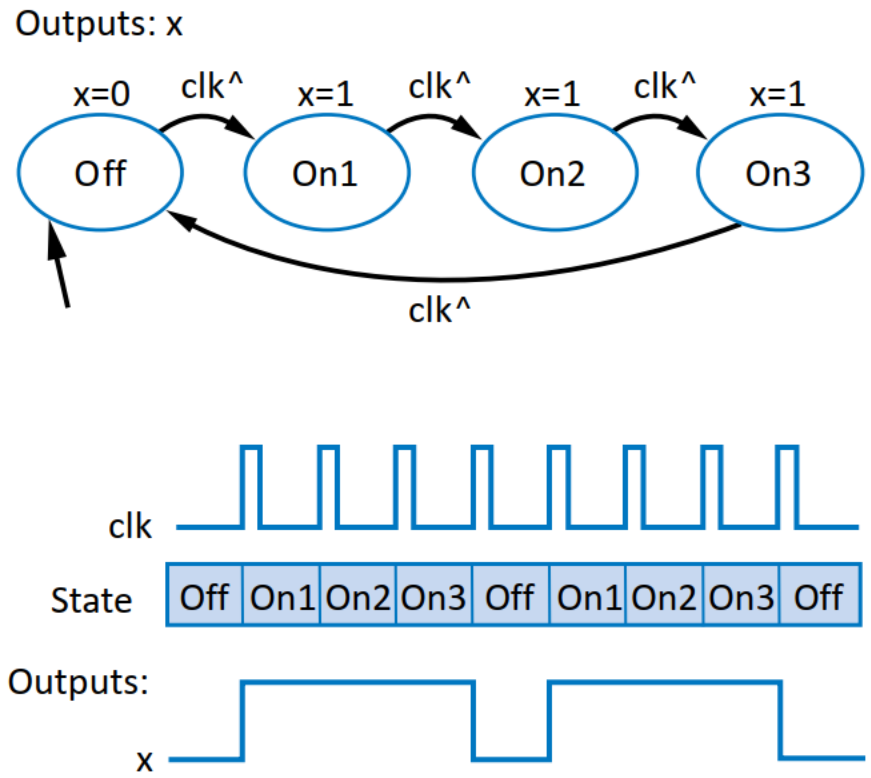
Shifter

[EX1] Create a circuit that approximately computes $P = 0.875 \times Q$ using only shifters and adders. **Roll down** the result to integers. Use wider internal components and wires as necessary to prevent internal overflow.

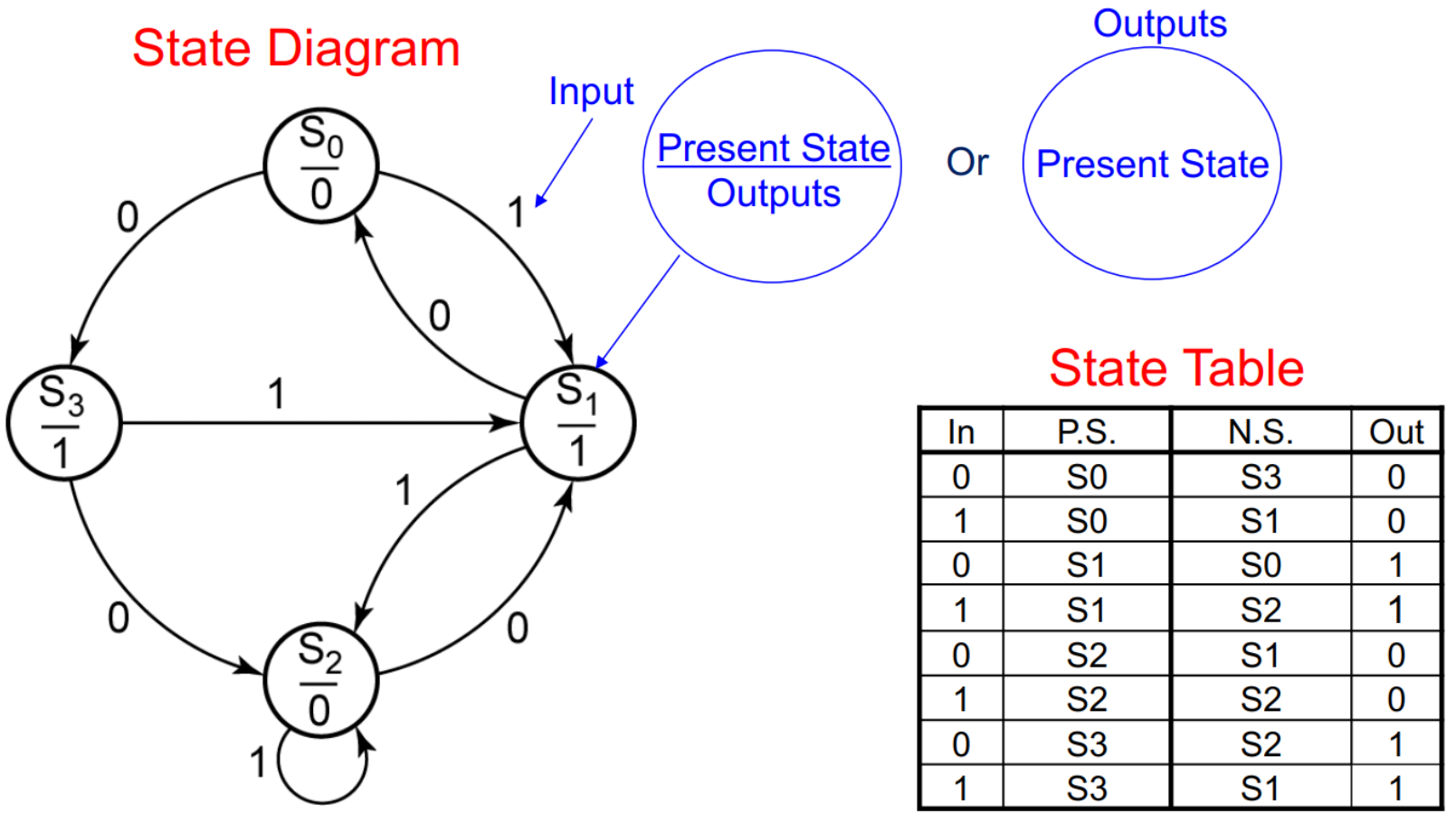
FSM

Definition

- A way to **design** a sequential circuit by **describing desired behavior** of the sequential circuit
- Consists of a set of states, transitions between states, and maybe inputs and outputs
  - ***present state***: currently happening
  - ***next state***: next to happen



State Diagram and State Table

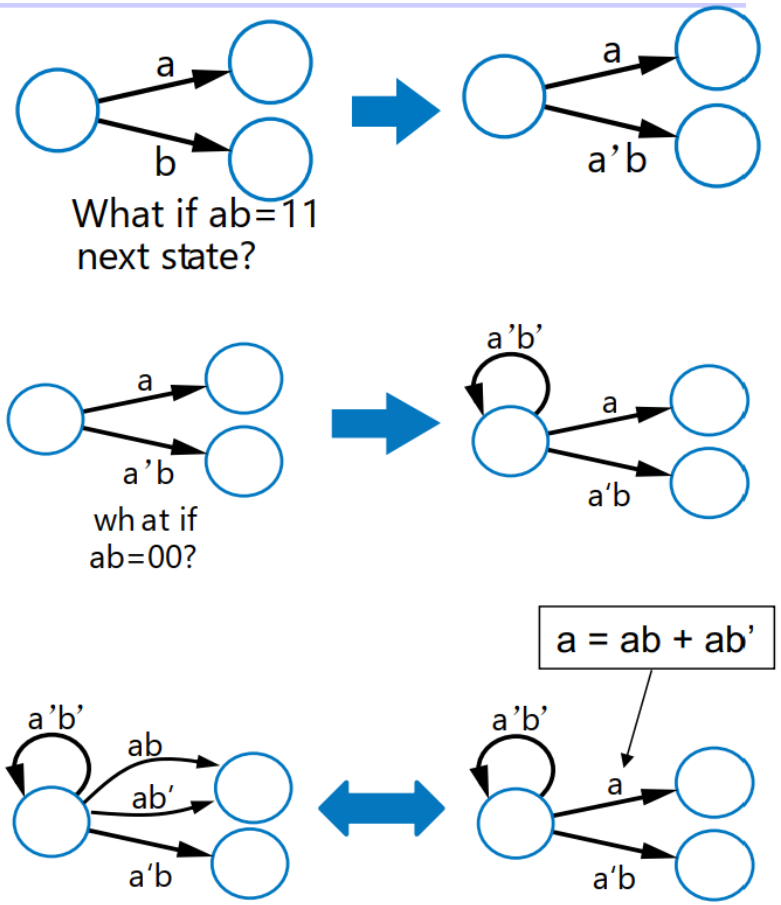


State Table

In	P.S.	N.S.	Out
0	S0	S3	0
1	S0	S1	0
0	S1	S0	1
1	S1	S2	1
0	S2	S1	0
1	S2	S2	0
0	S3	S2	1
1	S3	S1	1

Common State Transition Property

- *Only* one condition should be true, among all transitions leaving a state
- *One* condition must be true
  - For any input combination
- All conditions must be considered when leaving a state



FSM Exercise

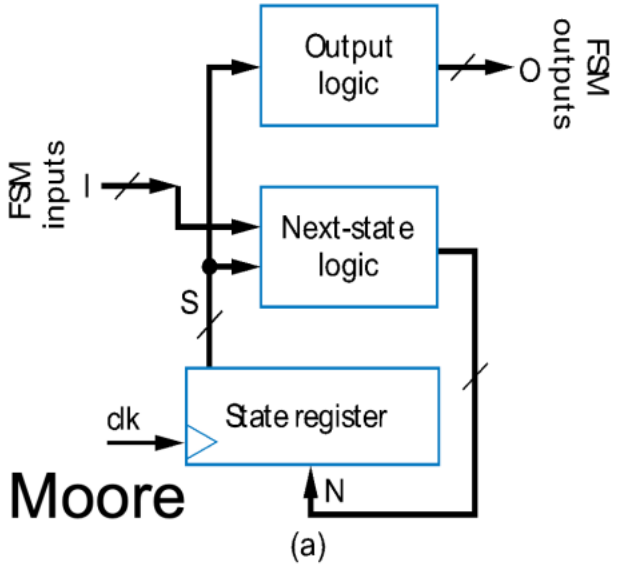
HW6_4. Draw a state diagram for an FSM that has an input X and an output Y. Whenever X changes from 0 to 1, Y should become 1 for two clock cycles and then return to 0, even if X is still 1.

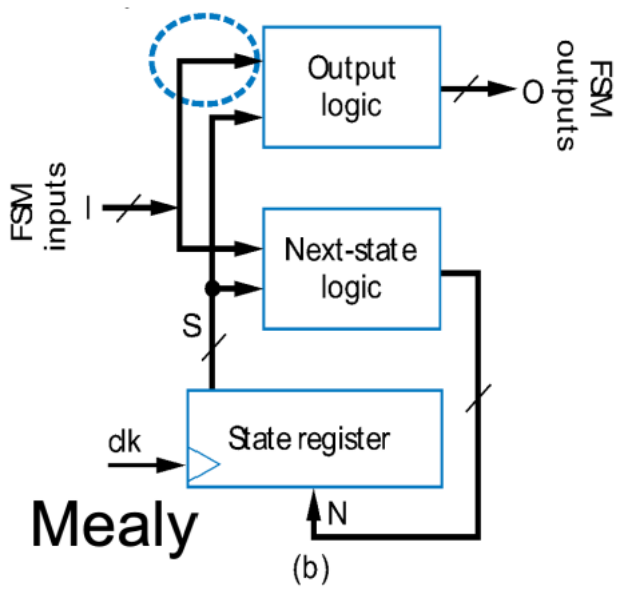
Moore vs. Mealy FSMs

– Output logic

• Depends on present state only – Moore FSM

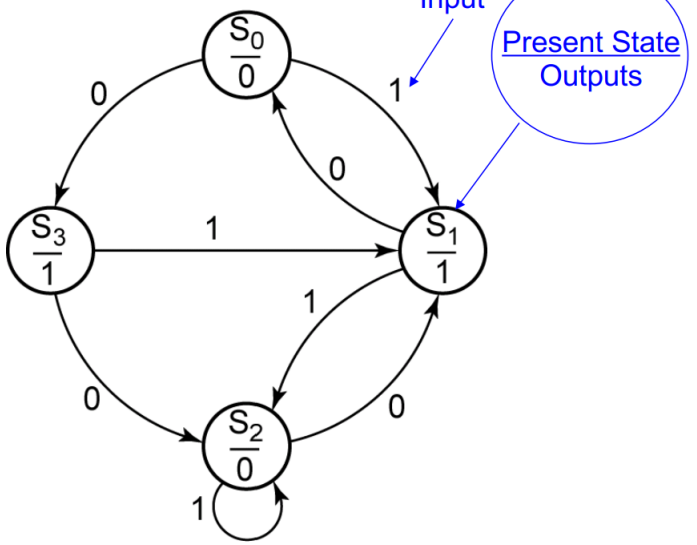
• Depends on present state and FSM inputs – Mealy FSM



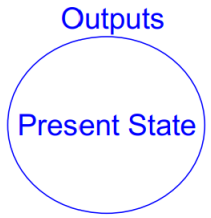


Moore vs Mealy FSMs

State Diagram



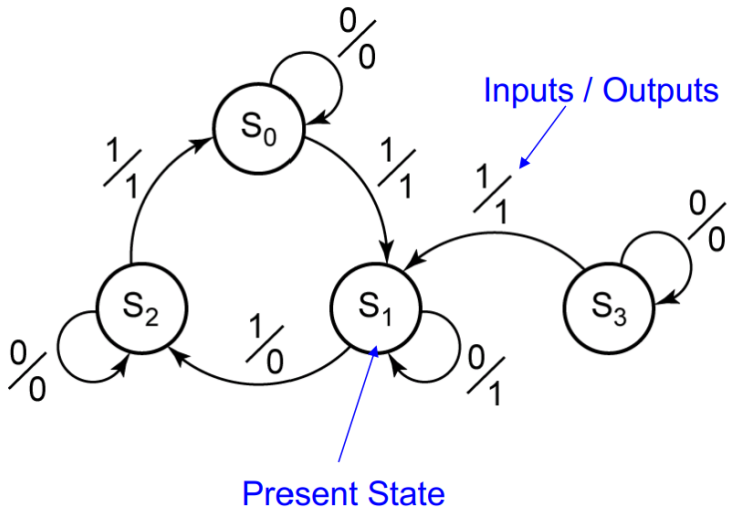
Or



State Table

In	P.S.	N.S.	Out
0	S0	S3	0
1	S0	S1	
0	S1	S0	1
1	S1	S2	
0	S2	S1	0
1	S2	S2	
0	S3	S2	1
1	S3	S1	

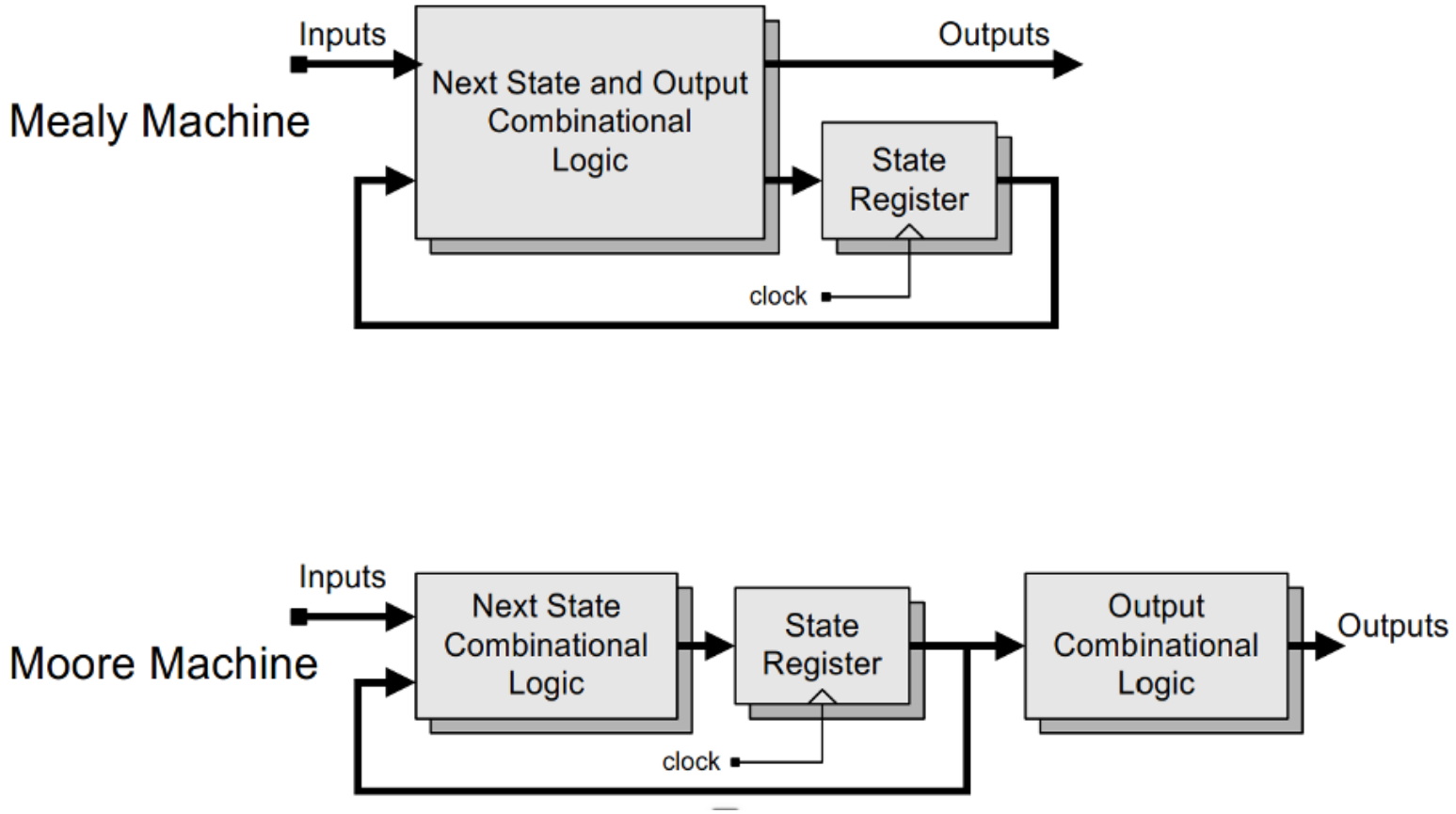
State Diagram



State Table

In	P.S.	N.S.	Out
0	S0	S0	0
1	S0	S1	1
0	S1	S1	1
1	S1	S2	0
0	S2	S2	0
1	S2	S0	1
0	S3	S3	0
1	S3	S1	1

Standard architecture of FSMs



Moore vs Mealy FSMs

- Output
  - Mealy: depends on both inputs and presents
  - Moore: doesn't depend on inputs
- State Diagram
  - Mealy: less states -> potentially less number of flip-flops
  - Moore: more states than Mealy -> possibly bigger circuit
- Speed of output response to the inputs
  - Mealy: quick, as soon as input changes
  - Moore: as long as one clock cycle delay
- TIMING ISSUE
  - Mealy: asynchronous, may cause serious problem
  - Moore: synchronous, more stable

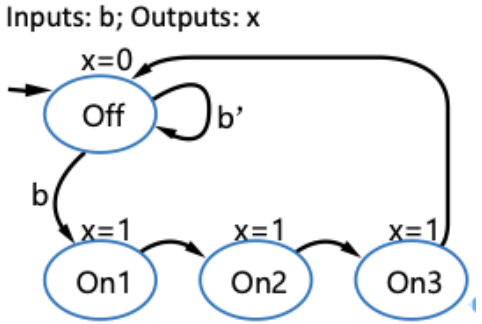
Standard FSM

- Five step FSM design process

	Step	Description
Step 1	*Capture the FSM*	Create an FSM that describes the desired behavior of the controller.
Step 2	*Create the architecture*	Create the standard architecture by using a state register of appropriate width, and combinational logic with inputs being the state register bits and the FSM inputs and outputs being the next state bits and the FSM outputs.
Step 3	*Encode the states*	Assign a unique binary number to each state. Each binary number representing a state is known as an ***encoding.*** Any encoding will do as long as each state has a unique encoding.
Step 4	*Create the state table*	Create a truth table for the combinational logic such that the logic will generate the correct FSM outputs and next state signals. Ordering the inputs with state bits first makes this truth table describe the state behavior, so the table is a state table.
Step 5	*Implement the combinational logic*	Implement the combinational logic using any method.

FSM design Exercise

HW7_5. Compare the logic size (number of gate inputs) and the delay (length of critical path) of a minimum binary encoding and a one-hot encoding of the following FSM.

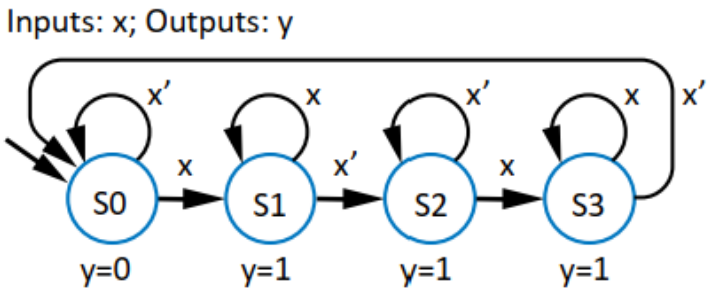


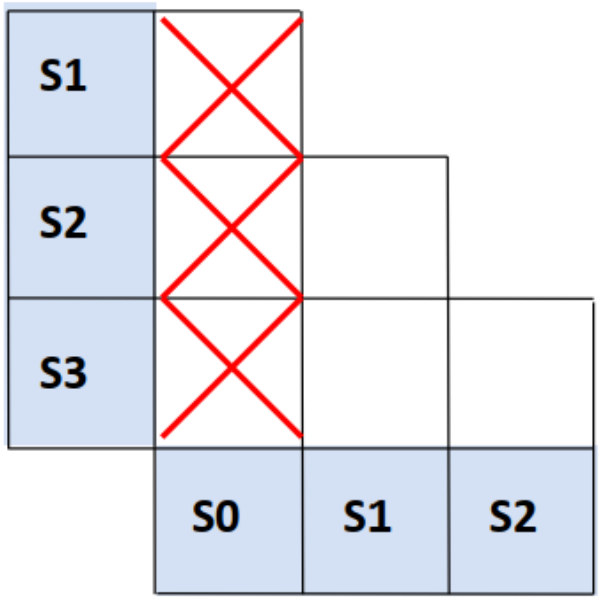
FSM Optimization

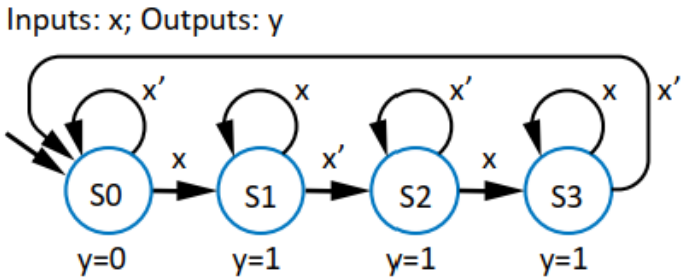
State reduction with Implication tables

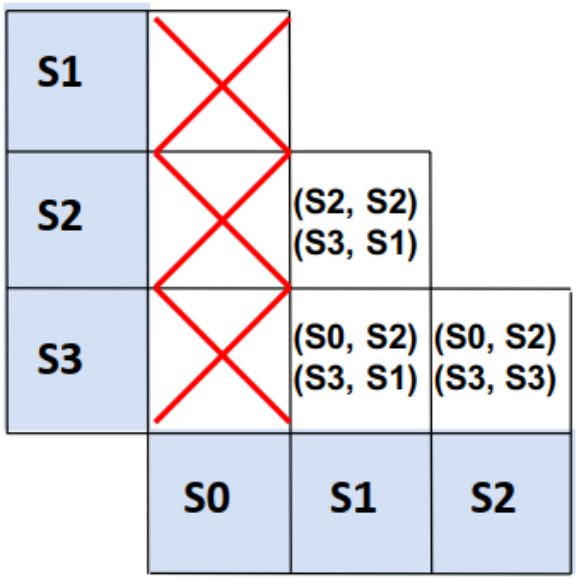
	Step	Description
1	*Mark state pairs having different outputs as nonequivalent*	States having different outputs obviously cannot be equivalent.
2	*For each unmarked state pair, write the next state pairs for the same input values*	
3	*For each unmarked state pair, mark state pairs having nonequivalent next-state pairs as nonequivalent. Repeat this step until no change occurs, or until all states are marked.*	States with nonequivalent next states for the same input values can't be equivalent. Each time through this step is called a ***pass***.
4	*Merge remaining state pairs*	Remaining state pairs must be equivalent.

FSM Optimization



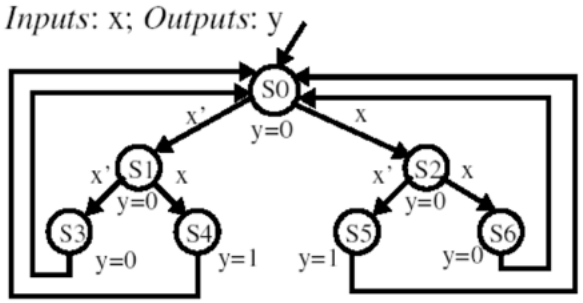






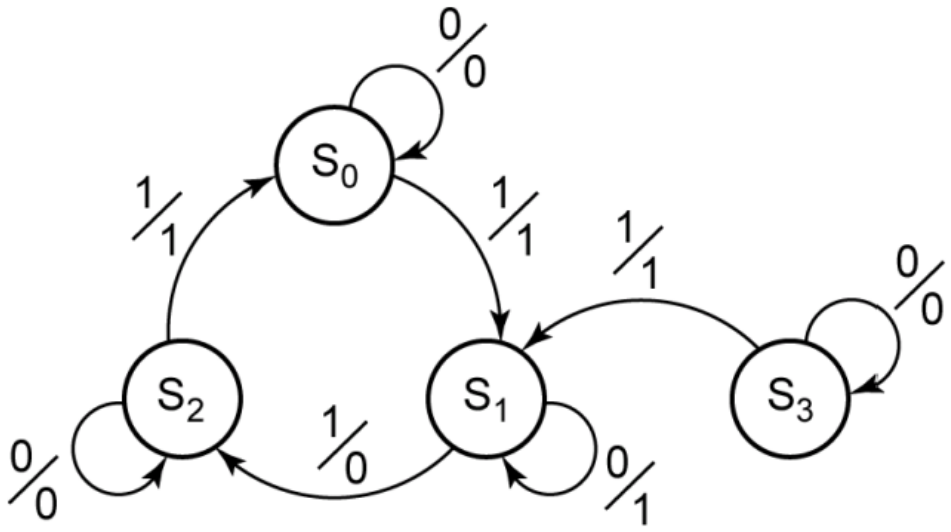
FSM Optimization Exercise

HW7_6. Reduce the number of states for the following FSM by using an implication table.



FSM Optimization of Mealy FSM

- Example:



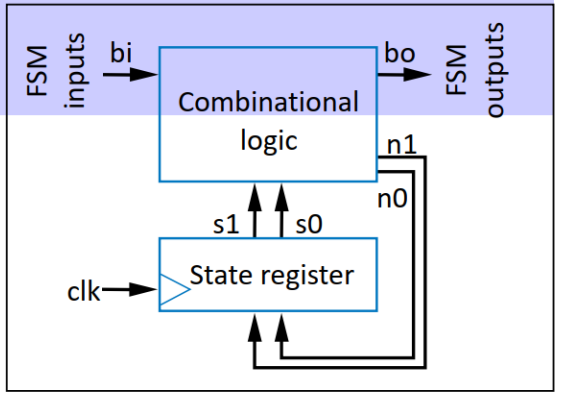
- Should have both next state pairs and output pairs in a cell for comparison

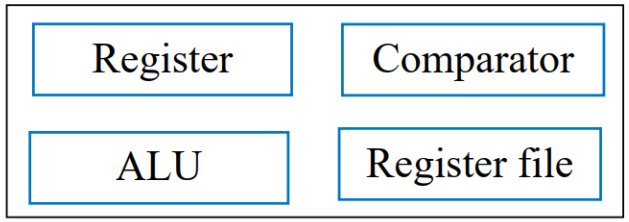
S1	Out: (0, 1) (1, 0) NS: (S0, S1) (S1, S2)		
S2	Out: (0, 0) (1, 1) NS: (S0, S2) (S1, S0)	Out: (1, 0) (0, 1) NS: (S1, S2) (S2, S0)	
S3	Out: (0, 0) (1, 1) NS: (S0, S3) (S1, S1)	Out: (1, 0) (0, 1) NS: (S1, S3) (S2, S1)	Out: (0, 0) (1, 1) NS: (S2, S3) (S0, S1)
	S0	S1	S2

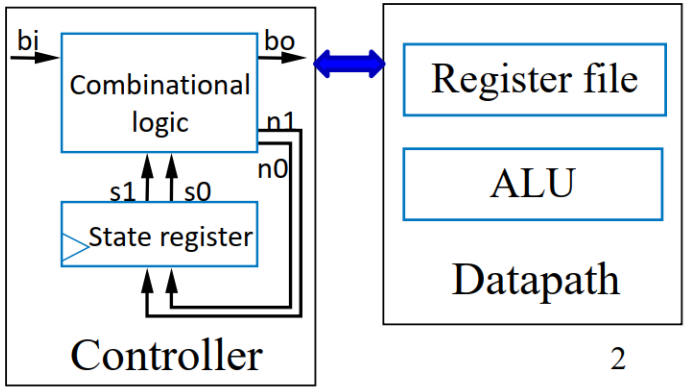
RTL Design

Introduction

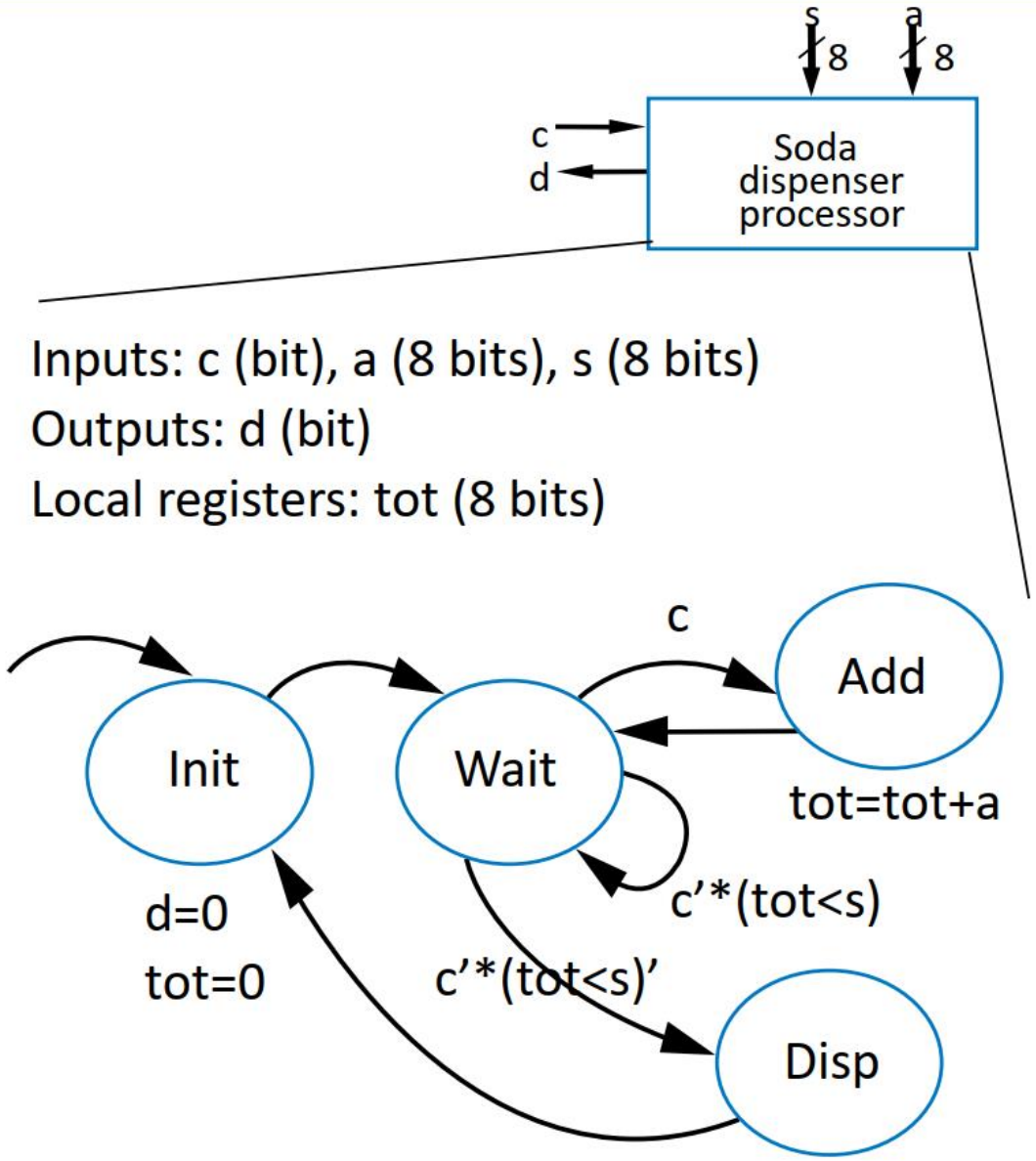
- Controllers (FSM)
  - Describes behavior of circuits
  - Takes inputs, generates outputs
  - Implemented with state register and combinational logic
- Datapath components
  - Operations on data
  - Path that data flows through
  - Places data is stored
- Digital Device
  - Datapath processes data according to control signals produced by FSM
  - To implement an algorithm
  - Design on Register Transfer Level







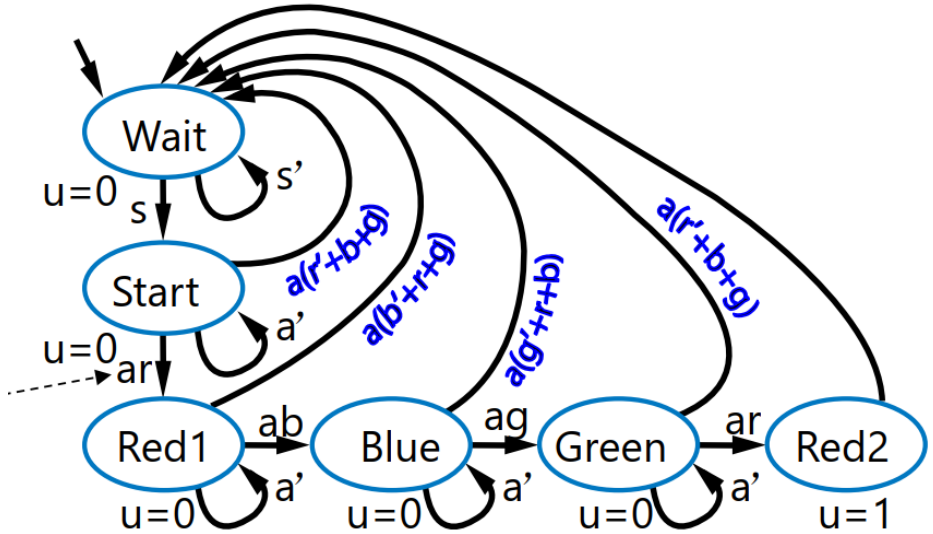
High level state machine (HLSM)



HLSM:

- Data types beyond just bits
- Arithmetic operations in states

HLSM VS. FSM



JOINT INSTITUTE 交大密西根学院

RTL Design Method

	Step	Description
Step 1	*Capture a high-level state machine*	Describe the system's desired behavior as a high-level state machine. The state machine consists of states and transitions. The state machine is "high-level" because the transition conditions and the state actions are more than just Boolean operations on bit inputs and outputs.
Step 2	*Create a datapath*	Create a datapath to carry out the data operations of the high-level state machine.
Step 3	*Connect the datapath to a controller*	Connect the datapath to a controller block. Connect external Boolean inputs and outputs to the controller block.
Step 4	*Derive the controller's FSM*	Convert the high-level state machine to a finite-state machine (FSM) for the controller, by replacing data operations with setting and reading of control signals to and from the datapath.

RTL Design Method

HW8_2. Use the RTL design process to design a reaction timer system that measures the time elapsed between the illumination of a light and the pressing of a button by a user. The reaction timer has three inputs, a clock input *clk*, a reset input *rst*, and a button input *B*. it has three outputs, a light enable output *len*, a 12-bit reaction time output *rtime*, and a *slow* output indicating that the user was not fast enough. The reaction timer works as follows. On reset, the reaction timer waits for 10 seconds before illuminating the light by setting *len* to 1. The reaction timer then measures the length of a time in milliseconds before the user presses the button *B*, outputting the time as a 12-bit binary number on *rtime*. If the user did not press the button within 2 seconds (2000 milliseconds), the reaction timer will set the output *slow* to 1 and output 2000 on *rtime*. Assume that the clock input has a frequency of 1 kHz.

Arithmetic Components

Arithmetic-Logic Unit: ALU

- ***ALU***: Component that can perform any of various arithmetic (add, subtract, increment, etc.) and logic (AND, OR, etc.) operations, based on control inputs
- Key component in computer

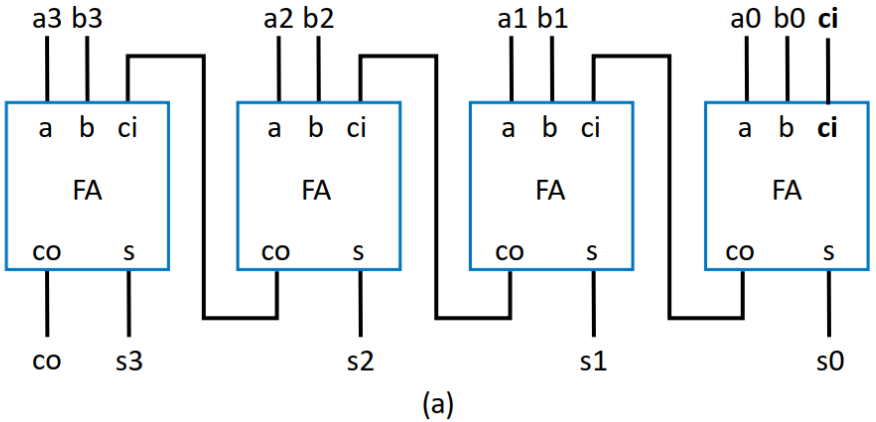
TABLE 4.2 Desired calculator operations

Inputs			Operation	Sample output if A=00001111, B=00000101
x	y	z		
0	0	0	S = A + B	S=00010100
0	0	1	S = A - B	S=00001010
0	1	0	S = A + 1	S=00010000
0	1	1	S = A	S=00001111
1	0	0	S = A AND B (bitwise AND)	S=00000101
1	0	1	S = A OR B (bitwise OR)	S=00001111
1	1	0	S = A XOR B (bitwise XOR)	S=00001010
1	1	1	S = NOT A (bitwise complement)	S=11110000

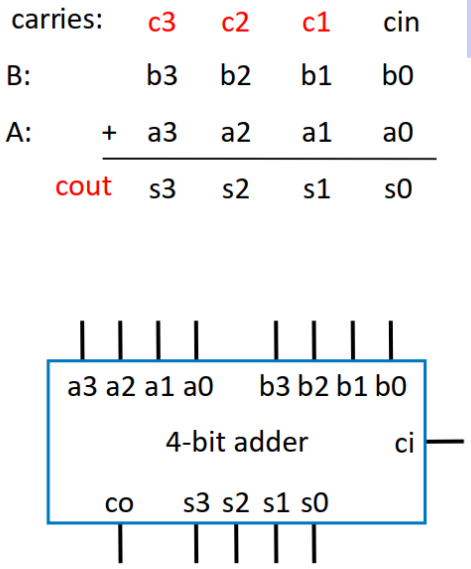
Arithmetic Components

Carry-Ripple Adder

- 4-bit adder: Adds two 4-bit numbers, generates 5-bit output
  - 5-bit output can be considered 4-bit "sum" plus 1-bit "carry out"
- Can easily build any size adder

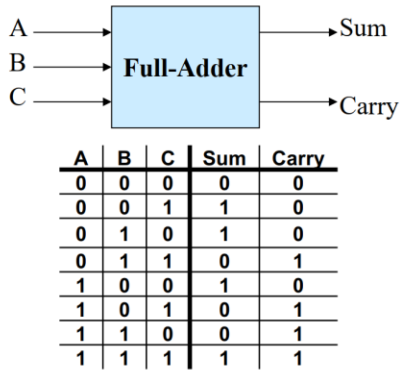


(a)



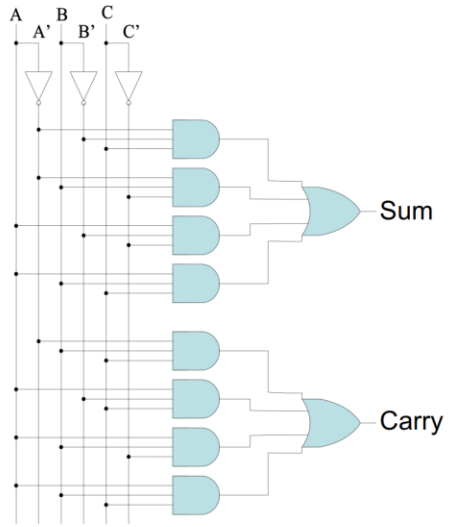
(b)

Full Adder



A	B	C	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Sum = A'B'C + A'BC' + AB'C' + ABC
= Σ m(1, 2, 4, 7)
Carry = A'BC + AB'C + ABC' + ABC
= Σ m(3, 5, 6, 7)

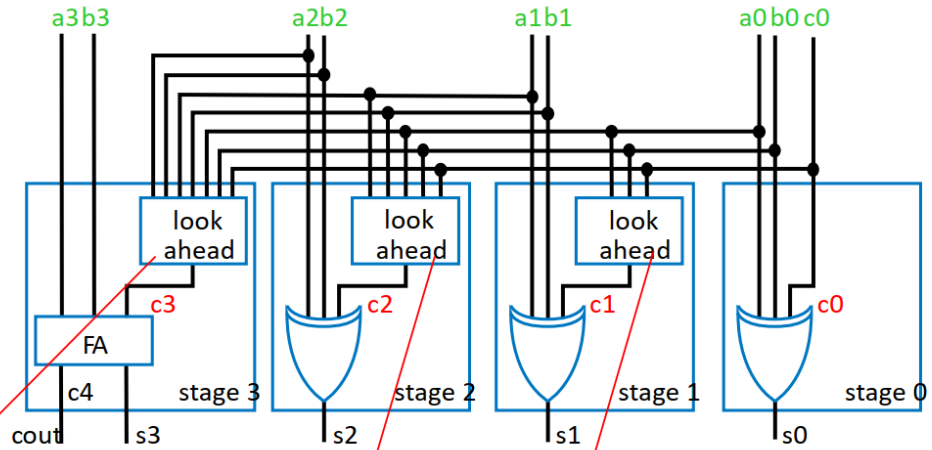


Notice the circuit draw convention

Arithmetic Components

Carry lookahead adder

- Carry lookahead logic
  - No waiting for ripple
  - 2-layer SOP logic
- Problem
  - Equations get too big
  - Not efficient
  - Need a better form of lookahead



c1 = b0c0 + a0c0 + a0b0

c2 = b1b0c0 + b1a0c0 + b1a0b0 + a1b0c0 + a1a0c0 + a1a0b0 + a1b1

c3 = b2b1b0c0 + b2b1a0c0 + b2b1a0b0 + b2a1b0c0 + b2a1a0c0 + b2a1a0b0 + b2a1b1 +
a2b1b0c0 + a2b1a0c0 + a2b1a0b0 + a2a1b0c0 + a2a1a0c0 + a2a1a0b0 + a2a1b1 + a2b2

Huge number of gates

Arithmetic Components

Carry lookahead adder

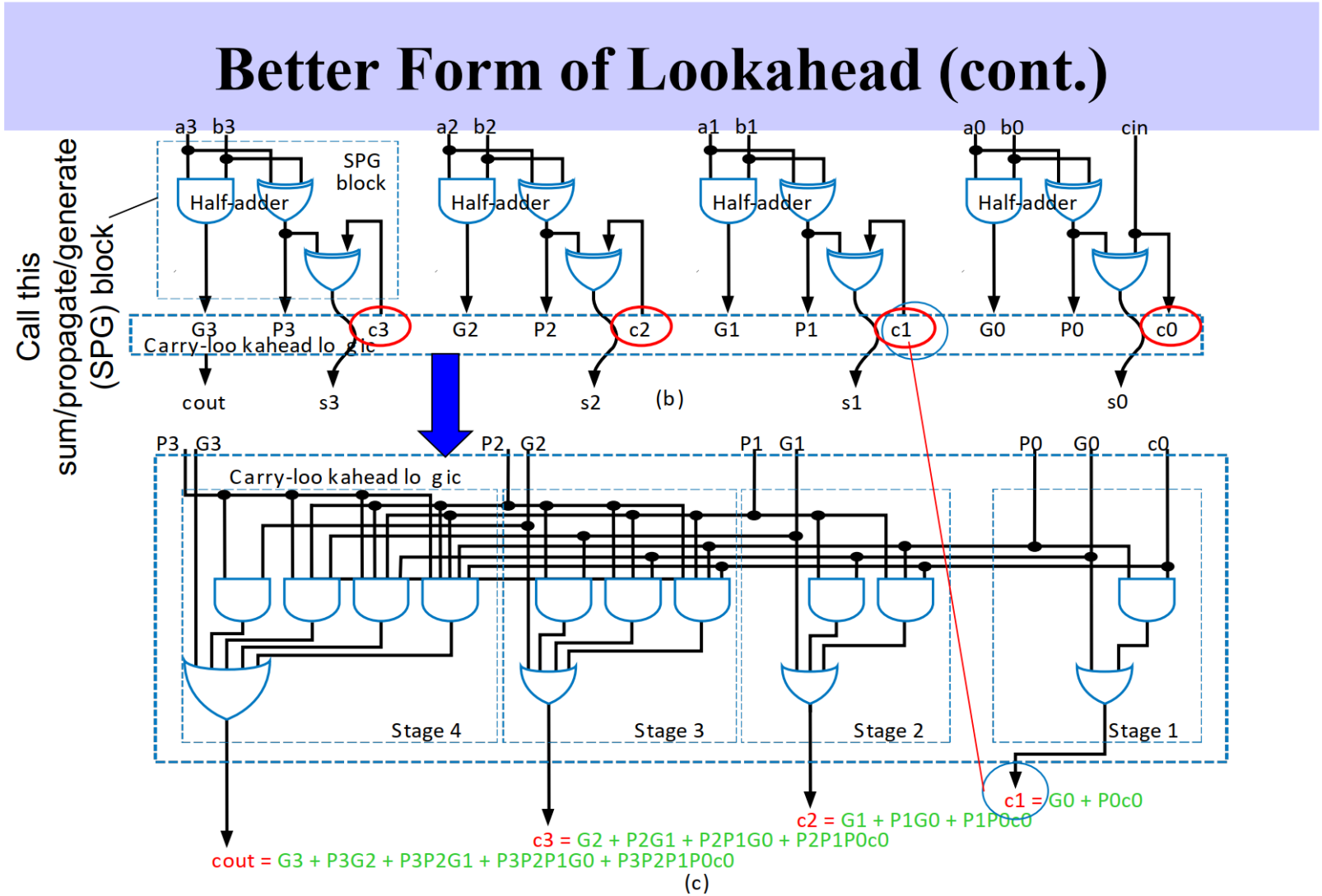
- Recall Full Adder, another equation for carry

  Carry = ab + (a ⊕ b)c

  Logic statement: Carry out is **generated** when a=1 and b=1, or **propagated** from carry in when a != b.
- Define two terms
  - ***Propagate***: P = a ⊕ b
  - ***Generate***: G = ab
- Compute lookahead carries from *P* and *G* terms, *not from external inputs*
  - Cout = G + Pc
    - c1 = a0b0 + (a0⊕b0)c0 = G0 + P0c0
    - c2 = a1b1 + (a1⊕b1)c1 = G1 + P1c1
    - c3 = a2b2 + (a2⊕b2)c2 = G2 + P2c2

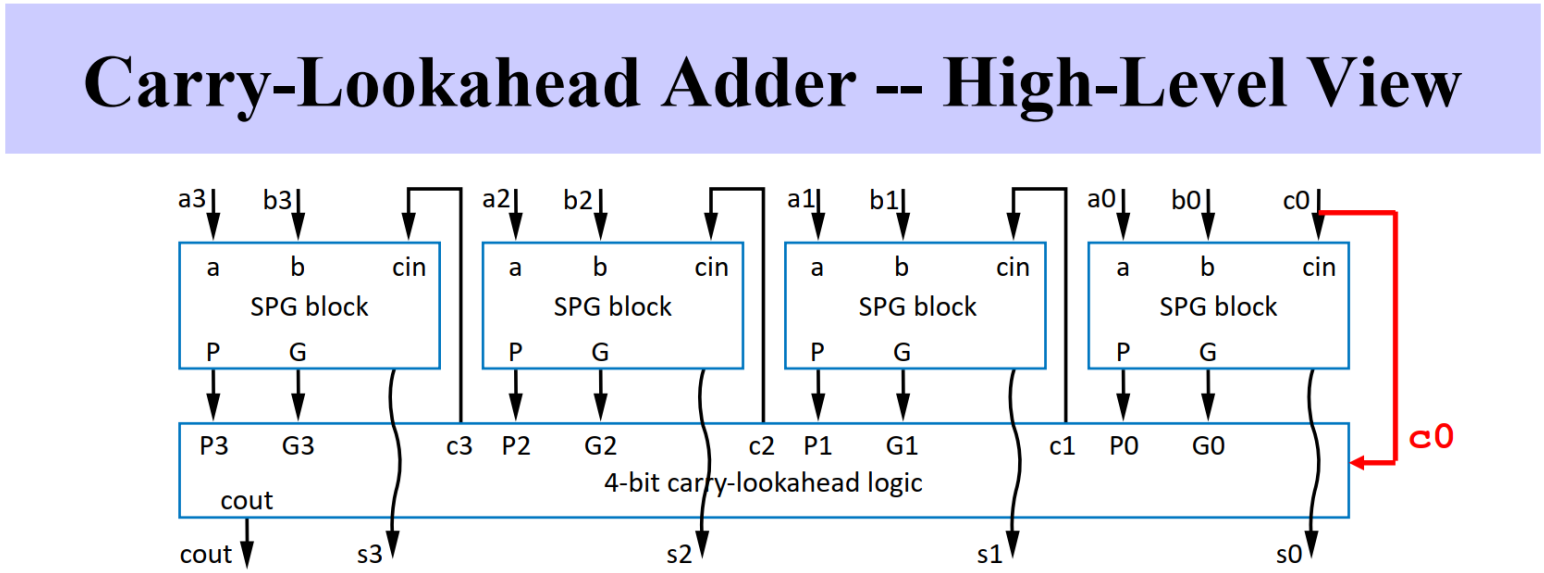
Arithmetic Components

Carry lookahead adder



Arithmetic Components

Carry lookahead adder

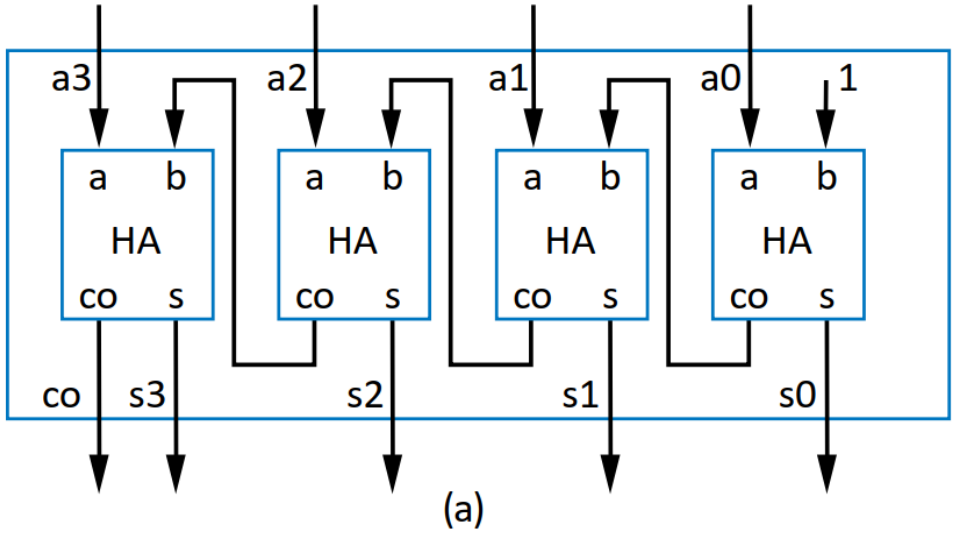


- Fast -- only 4 gate level delays
  - Each stage has SPG block with 2 gate levels
  - Carry-lookahead logic quickly computes the carry from the propagate and generate bits using 2 gate levels inside
- Reasonable number of gates
- Nice balance between intuitive lookahead and pure combinational logic

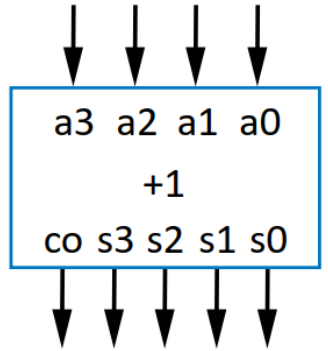
Arithmetic Components

Incrementer

Inputs				Outputs				
a3	a2	a1	a0	c0	s3	s2	s1	s0
0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	1	0
0	0	1	0	0	0	0	1	1
0	0	1	1	0	0	1	0	0
0	1	0	0	0	0	1	0	1
0	1	0	1	0	0	1	1	0
0	1	1	0	0	0	1	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	0	1	0	0	1
1	0	0	1	0	1	0	1	0
1	0	1	0	0	1	0	1	1
1	0	1	1	0	1	1	0	0
1	1	0	0	0	1	1	0	1
1	1	0	1	0	1	1	1	0
1	1	1	0	0	1	1	1	1
1	1	1	1	1	0	0	0	0



(a)



(b)

Arithmetic Components

Multiplier

- Can build multiplier that mimics multiplication by hand
  - Notice that multiplying multiplicand by 1 is same as ANDing with 1

```
  0110   (the top number is called the multiplicand)
  0011   (the bottom number is called the multiplier)
  ----   (each row below is called a partial product)
  0110   (because the rightmost bit of the multiplier is 1, and 0110*1=0110)
 0110    (because the second bit of the multiplier is 1, and 0110*1=0110)
0000     (because the third bit of the multiplier is 0, and 0110*0=0000)
+0000    (because the leftmost bit of the multiplier is 0, and 0110*0=0000)
--------
00010010 (the product is the sum of all the partial products: 18, which is 6*3)
```

Arithmetic Components

Multiplier