DDCO

UNIT-3

CLASS NOTES

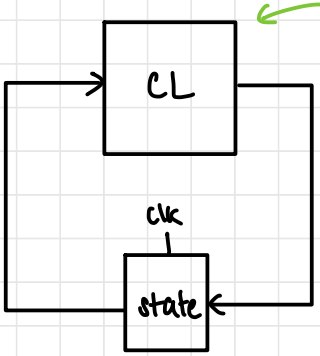
feedback/corrections: vibha@pesu.pes.edu

Vibha Masti

FINITE STATE MACHINES

continuation

- Mathematical foundation for sequential logic circuits
- Consists of two blocks — CL and state block



only combinational logic elements

each arrow represents wires

can contain any memory elements

in this course, only synchronous, seq. logic circuits

specifically, D-flip flops

- Any sequential logic circuit (from a counter to a complex microprocessor) can be represented as an FSM
- Fundamental concept in CSE
- In AFLL, nodes and edges, but to implement, this
- This diagram lacks input & output
- Two types — Mealy type and Moore type FSMs

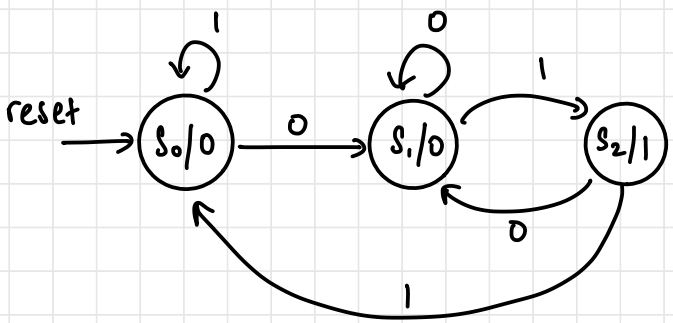
Question 1

A snail crawls down a paper tape with 1's and 0's. The snail smiles whenever the last 2 digits it has crawled over are 01. Design Moore and Mealy machines.

Moore

(input ↛ output, output & state)

output = 1 → smile
0 → no smile



eg: 10010

- as long as we are in a state, o/p is constant

State Transition Table

Current State	Input	Next State	output
S_0	0	S_1	0
S_0	1	S_0	0
S_1	0	S_1	0
S_1	1	S_2	0
S_2	0	S_1	1
S_2	1	S_0	1

o/p depends only on current state

Binary Encoding Method

S_0 — 00
S_1 — 01
S_2 — 10

} states, do not confuse with flip flop names

Current State	Input	Next State	output
00	0	01	0
00	1	00	0
01	0	01	0
01	1	10	0
10	0	01	1
10	1	00	1

Logic

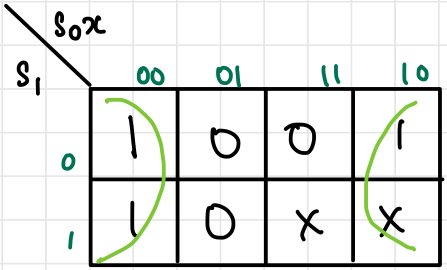
S_1	S_0	x	S_1'	S_0'	y
0	0	0	0	1	0
0	0	1	0	0	0
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	0	1	1
1	0	1	0	0	1

S_1'

S_1 \ S_0x	00	01	11	10
0	0	0	1	0
1	0	0	X	X

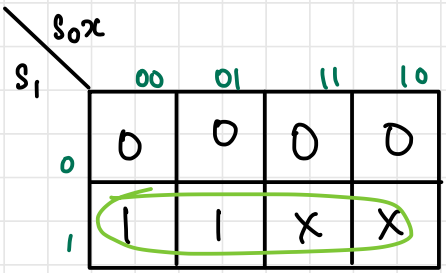
$S_1' = S_0 x$

S_0'

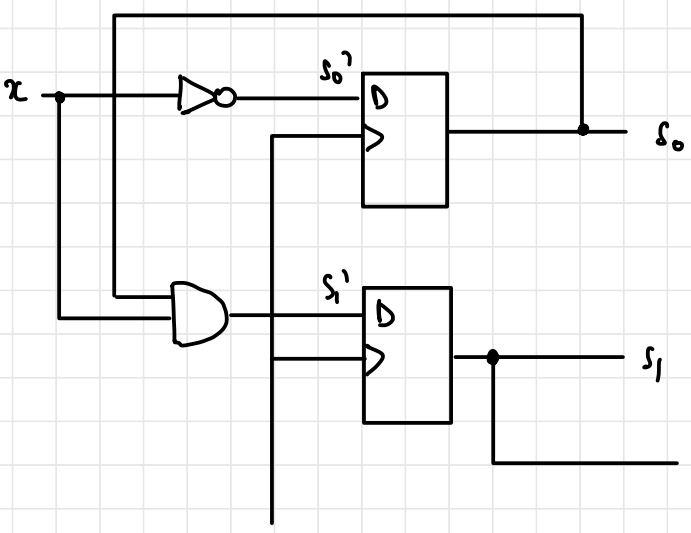


$S_0' = \bar{x}$

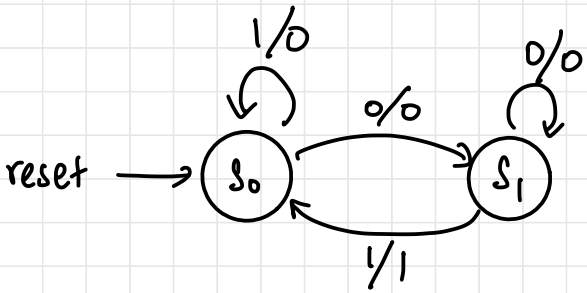
y



$y = S_1$



Mealy



Current State	Input	Next State	output
0	0	1	0
0	1	0	0
1	0	1	0
1	1	0	1

s	x	s'	y
0	0	1	0
0	1	0	0
1	0	1	0
1	1	0	1

$s' = s\bar{x} + s\bar{x} = \bar{x}$

$y = sx$

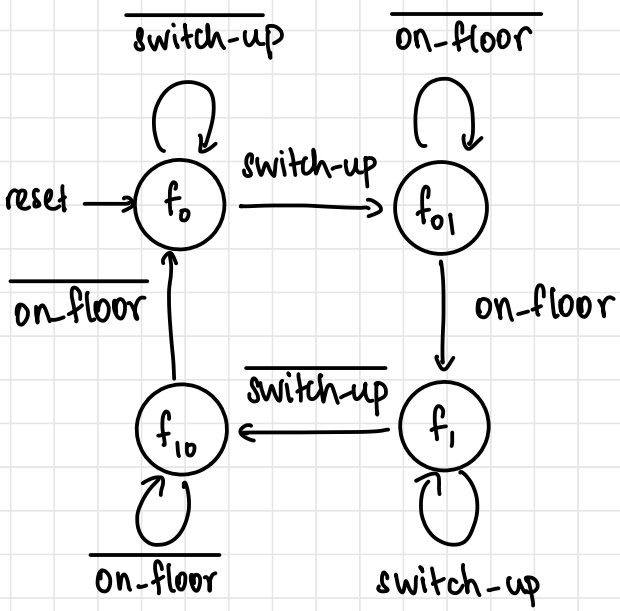
ONE HOT ENCODING

- Same number of flip flops as states
- Binary encoding: $\log_2$(States) flip flops

Question 2

Implement the lift problem (Moore & Mealy) with one hot encoding

Moore



State Encoding Table

State	Encoding
f_0	0001
f_{01}	0010
f_1	0100
f_{10}	1000

Output Encoding Tables

- on-ground

Meaning	Encoding
Lift on ground floor	1
Lift anywhere else	0

- on-first

Meaning	Encoding
Lift on first floor	1
Lift anywhere else	0

- lift-up

Meaning	Encoding
Lift going from ground to first floor	1
Lift anywhere else	0

- lift_down

Meaning	Encoding
Lift going from first to ground floor	1
Lift anywhere else	0

State Transition Table

Current State				Inputs		Next State			
s_3	s_2	s_1	s_0	*switch_up*	*on_floor*	s_3'	s_2'	s_1'	s_0'
0	0	0	1	0	0	0	0	0	1
0	0	0	1	0	1	0	0	0	1
0	0	0	1	1	0	0	0	1	0
0	0	0	1	1	1	0	0	1	0
0	0	1	0	0	0	0	0	1	0
0	0	1	0	0	1	0	1	0	0
0	0	1	0	1	0	0	0	1	0
0	0	1	0	1	1	0	1	0	0
0	1	0	0	0	0	1	0	0	0
0	1	0	0	0	1	1	0	0	0
0	1	0	0	1	0	0	1	0	0
0	1	0	0	1	1	0	1	0	0
1	0	0	0	0	0	1	0	0	0
1	0	0	0	0	1	0	0	0	1
1	0	0	0	1	0	1	0	0	0
1	0	0	0	1	1	0	0	0	1

- Not using K-maps (6 inputs)
- We have only learnt 4 variable K-maps
- Can be minimised using Boolean identities

S_3' (SOP)

$\overline{S_3}\,S_2\,\overline{S_1}\,\overline{S_0}\ \overline{\text{switch\_up}}\ \overline{\text{on\_floor}}\ +$
$\overline{S_3}\,S_2\,\overline{S_1}\,\overline{S_0}\ \overline{\text{switch\_up}}\ \text{on\_floor}\ +$

$S_3\,\overline{S_2}\,\overline{S_1}\,\overline{S_0}\ \overline{\text{switch\_up}} + \overline{\text{on\_floor}} +$
$S_3\,\overline{S_2}\,\overline{S_1}\,\overline{S_0}\ \text{switch\_up} + \overline{\text{on\_floor}} +$

(Boolean identities)

$= \overline{S_3}\,S_2\,\overline{S_1}\,\overline{S_0}\ \overline{\text{switch\_up}} + S_3\,\overline{S_2}\,\overline{S_1}\,\overline{S_0}\ \overline{\text{on\_floor}}$

We know through one-hot encoding that at any given time, only one of the 4 inputs is high (1)

$= S_2\ \overline{\text{switch\_up}} + S_3\ \overline{\text{on\_floor}}$ → much simpler than Binary encoding

$S_2' = S_1\ \text{on\_floor} + S_2\ \text{switch\_up}$

$S_3' = S_0\ \text{switch\_up} + S_1\ \overline{\text{on\_floor}}$

$S_4' = S_0\ \overline{\text{switch\_up}} + S_0\ \text{on\_floor}$

Output Table

State				Outputs			
s_3	s_2	s_1	s_0	on_ground	on_first	lift_up	lift_down
0	0	0	1	1	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	0	1	0	0
1	0	0	0	0	0	0	1

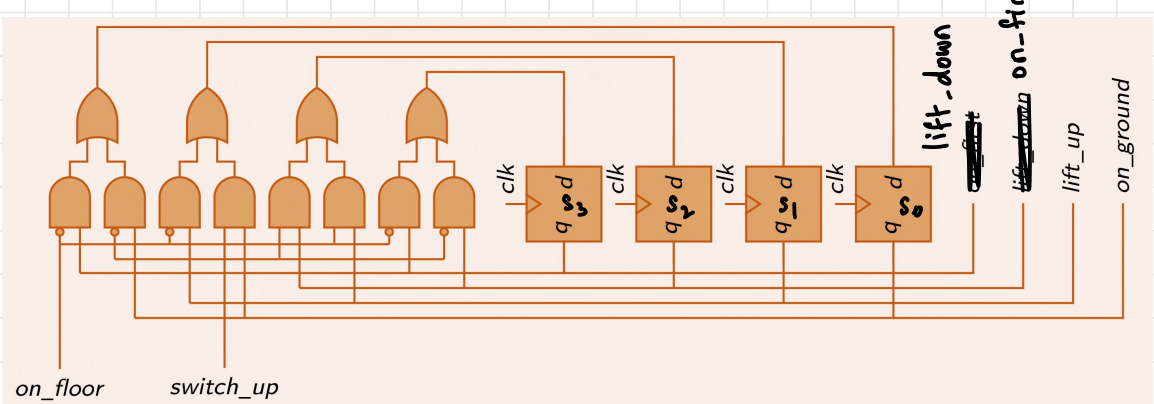
on_ground = s_0

on_first = s_2

lift_up = s_1

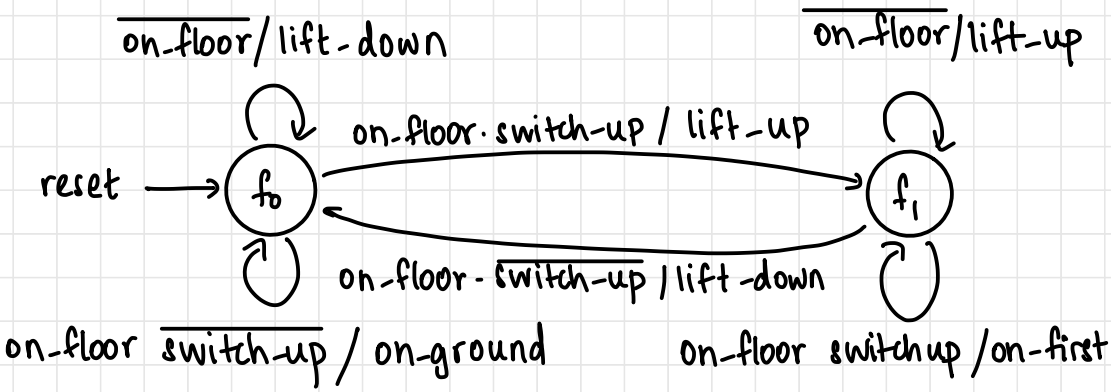
lift_down = s_3

Logic Circuit



- critical path delay for next state logic is reduced
- output logic simplified

Mealy



State Encoding Table

State	Encoding $(S_1 S_0)$
f_0	01
f_1	10

Output Encoding Tables

- on_ground

Meaning	Encoding
Lift on ground floor	1
Lift anywhere else	0

- on_first

Meaning	Encoding
Lift on first floor	1
Lift anywhere else	0

- lift_up

Meaning	Encoding
Lift going from ground to first floor	1
Lift anywhere else	0

- lift_down

Meaning	Encoding
Lift going from first to ground floor	1
Lift anywhere else	0

State Transition Table + Output

Current state		Input		Next state		Output			
S_1	S_0	on_floor	switch_up	S_1	S_0	on_ground	on_first	lift_up	lift_down
0	1	0	0	0	1	0	0	0	1
0	1	0	1	0	1	0	0	0	1
0	1	1	0	0	1	1	0	0	0
0	1	1	1	1	0	0	0	1	0
1	0	0	0	1	0	0	0	1	0
1	0	0	1	1	0	0	0	1	0
1	0	1	0	0	1	0	0	0	1
1	0	1	1	1	0	0	1	0	0

outputs

$$\text{on\_ground} = S_0 \cdot \text{on\_floor} \cdot \overline{\text{switch\_up}}$$
$$\text{on\_first} = S_1 \cdot \text{on\_floor} \cdot \text{switch\_up}$$
$$\text{lift\_up} = S_1 \cdot \overline{\text{on\_floor}} + S_0 \cdot \text{on\_floor} \cdot \overline{\text{switch\_up}}$$
$$\text{lift\_down} = S_0 \cdot \overline{\text{on\_floor}} + S_1 \cdot \text{on\_floor} \cdot \overline{\text{switch\_up}}$$

One Hot Encoding Method

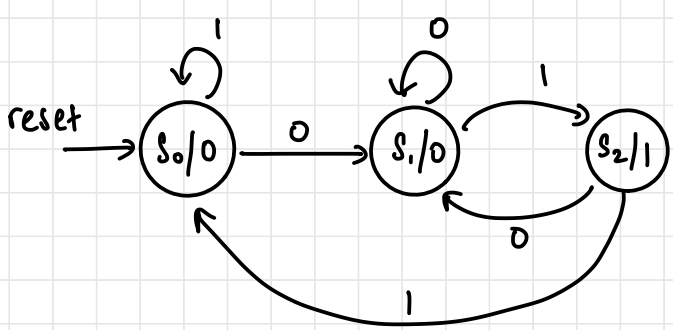
- no. of bits = no. of states
- $S_0 \rightarrow 001$
  $S_1 \rightarrow 010$
  $S_2 \rightarrow 100$
  ] at any given time, 1 bit is hot

Question 3

A snail crawls down a paper tape with 1's and 0's. The snail smiles whenever the last 2 digits it has crawled over are 01. Design Moore and Mealy machines. Use one hot encoding.

Moore

S_2	S_1	S_0	x	S_2'	S_1'	S_0'	y
0	0	1	0	0	1	0	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	1	0	0	0
1	0	0	0	0	1	0	1
1	0	0	1	0	0	1	1



eg: 10010

$$S_2' = \bar{S}_2 S_1 \bar{S}_0 x$$
$$= S_1 x$$

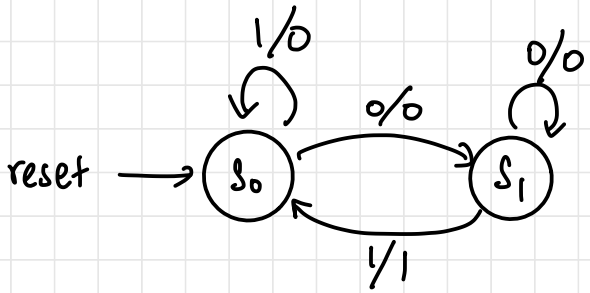
$$S_1' = \bar{S}_2 \bar{S}_1 S_0 x + \bar{S}_2 S_1 \bar{S}_0 x + S_2 \bar{S}_1 \bar{S}_0 \bar{x}$$
$$= S_0 x + S_1 x + S_2 \bar{x}$$

$$S_0' = \bar{S}_2 \bar{S}_1 S_0 x + S_2 \bar{S}_1 \bar{S}_0 x$$
$$= S_0 x + S_2 x$$

$$y = S_2 \bar{S}_1 \bar{S}_0 \bar{x} + S_2 \bar{S}_1 \bar{S}_0 x$$
$$= S_2$$

lower critical path delay

Mealy



$S_0 \rightarrow 01$
$S_1 \rightarrow 10$

S_1	S_0	x	S_1'	S_0'	y
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	0	1	1

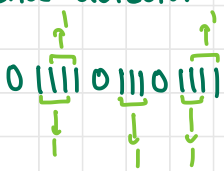
$$S_1' = \bar{S}_1 S_0 \bar{x} + S_1 \bar{S}_0 \bar{x}$$
$$= S_0 \bar{x} + S_1 \bar{x}$$

$$S_0' = \bar{S}_1 S_0 x + S_1 \bar{S}_0 x$$
$$= S_0 x + S_1 x$$

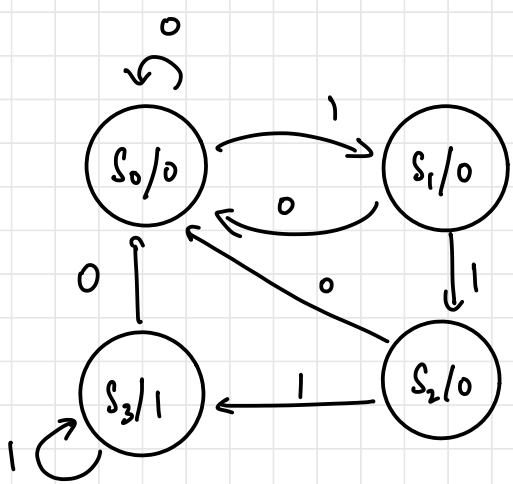
$$y = S_1 \bar{S}_0 x$$
$$= S_1 x$$

Question 4

Sequence detector → 111



Moore



S_1	S_0	x	S_1'	S_0'	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	1	1	1	1

S_1'

S_0 \ S_1x	00	01	11	10
0	0	0	1	0
1	0	1	1	0

$S_1' = S_1x + S_0x$

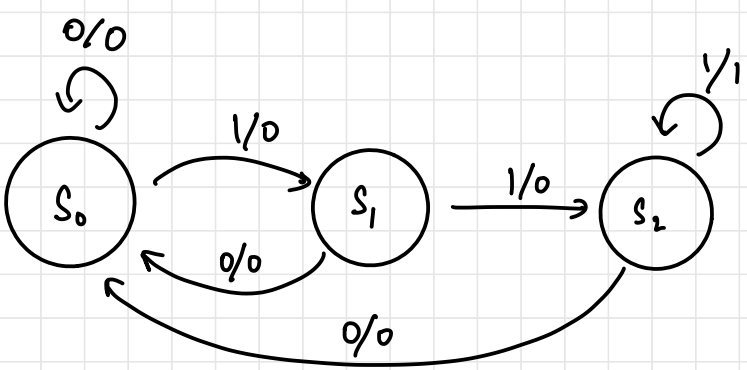
S_0'

S_0 \ S_1x	00	01	11	10
0	0	1	0	0
1	0	1	1	0

$S_1' = \bar{S}_1x + S_0x$

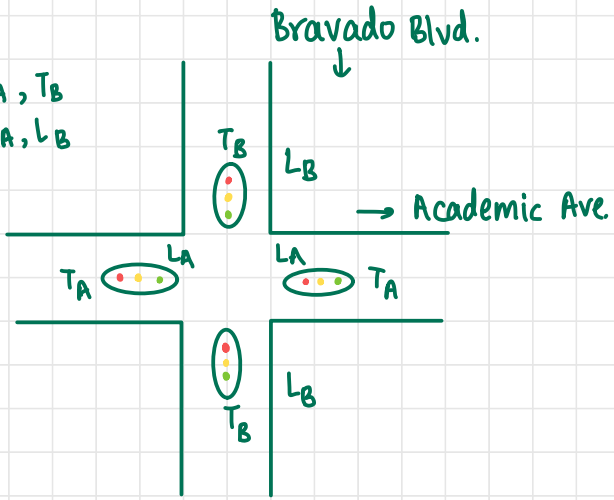
$y = S_1S_0$

Mealy



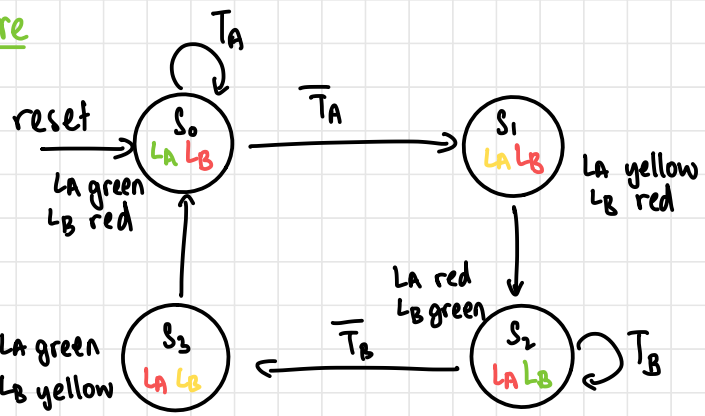
Question 5

Traffic light controller (FSM)



- Two sensors: T_A, T_B
- Two lights: L_A, L_B
- Clock: every 5 seconds
- Reset button for known initial state (green – academic ave, red – bravado blvd.)
- L_A / L_B are outputs (red, yellow, green) based on inputs from T_A / T_B
- While T_A is 1, L_A remains green
- When T_A becomes 0, L_A becomes red and L_B is green
- Starts collecting data from T_B, not T_A.

Moore



State Encoding Table

State	Encoding
S_0	00
S_1	01
S_2	10
S_3	11

State Transition Table

Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S_1'	S_0'
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

Output Encoding Table

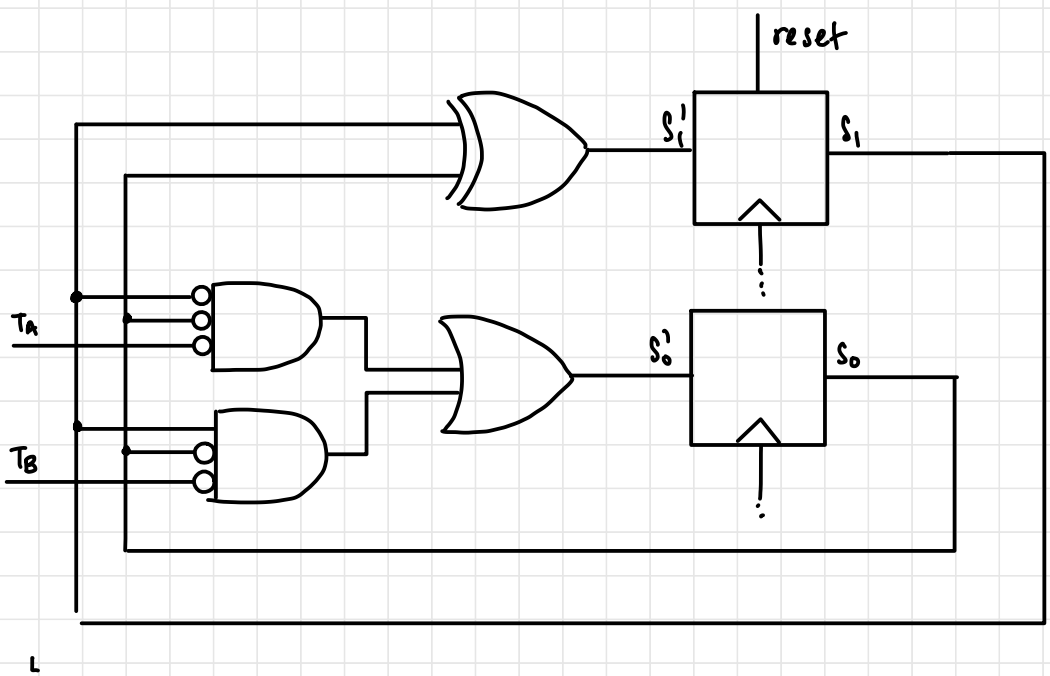
Output	Encoding
green	00
yellow	01
red	10

Output Table

Current state		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

$$L_{A1} = S_1$$
$$L_{A0} = \overline{S_1} S_0$$

$$L_{B1} = \overline{S_1}$$
$$L_{B0} = S_1 S_0$$

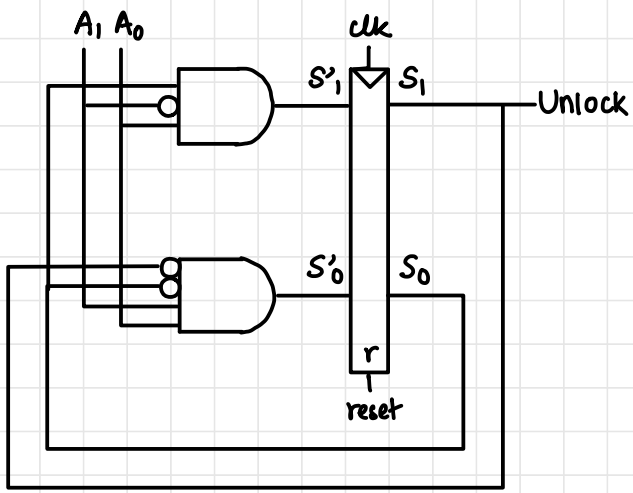


REVERSE ENGINEERING

- Given a logic ciccuit, determine FSM

Question 6

Reverse engineer the following ciccuit



Moore type

Input states : A_0 & A_1

Current states: S_1 & S_0

Next states: S_1' & S_0'

Output: unlock

] inputs (16 rows)

Next State Logic

Current State		Input		Next State	
S_1	S_0	A_1	A_0	S_1'	S_0'
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	1	0
1	1	1	0	0	0
1	1	1	1	0	0

- Next state never reaches 11
- We can therefore eliminate the current state 11 rows
- Whenever current state is 10, next state is always

Current State		Input		Next State	
S_1	S_0	A_1	A_0	S_1'	S_0'
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	x	x	0	0

$$S_1' = \bar{S}_1 S_0 \bar{A}_1 A_0$$

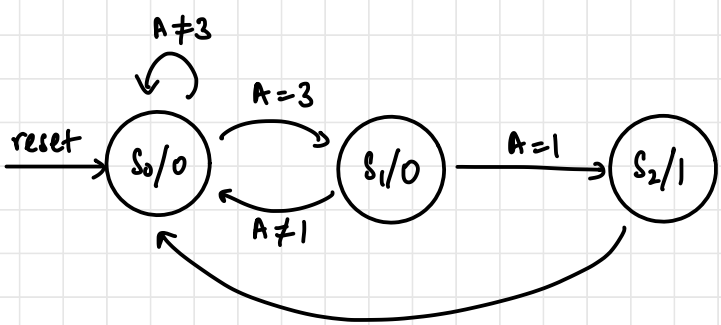
$$S_0' = \bar{S}_1 \bar{S}_0 A_1 A_0$$

$$\text{unlock} = S_1$$

State Encoding and Output

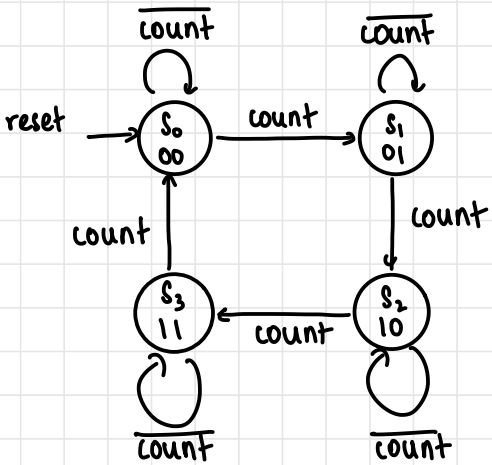
State	S_1	S_0	Output
S_0	0	0	0
S_1	0	1	0
S_2	1	0	1

FSM



COUNTERS

- Clock period of 2 GHz frequency is 0.5 ns
- Measure / keep time based on clock cycles
- Counters are Moore type FSMs having nodes arranged in a circle
- No inputs required



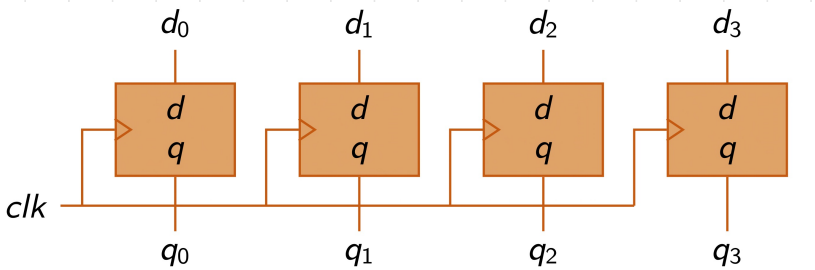
INCREMENTER COUNTER

n-Bit Counter

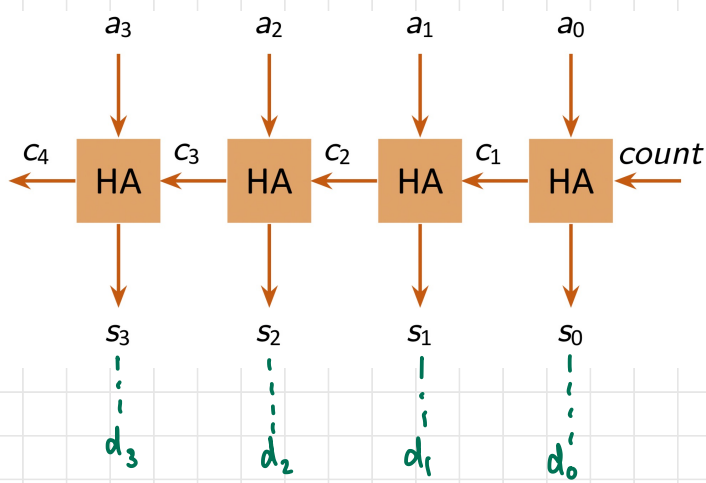
- From 0 to $2^n - 1$
- Every clock cycle we need to store the current clock value and then increment it every clock cycle.

Storing n bits

- n d-flip flops with a common clock
- n-bit register

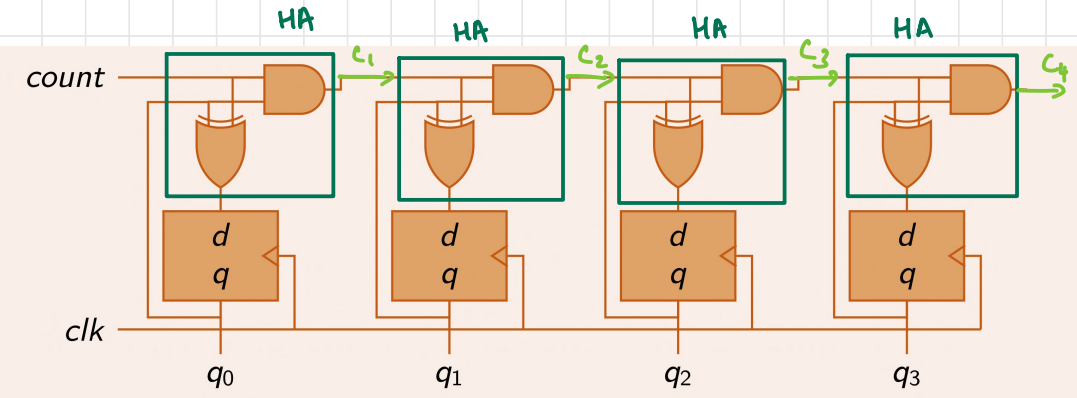


n-bit incrementer



when count = 0, same output

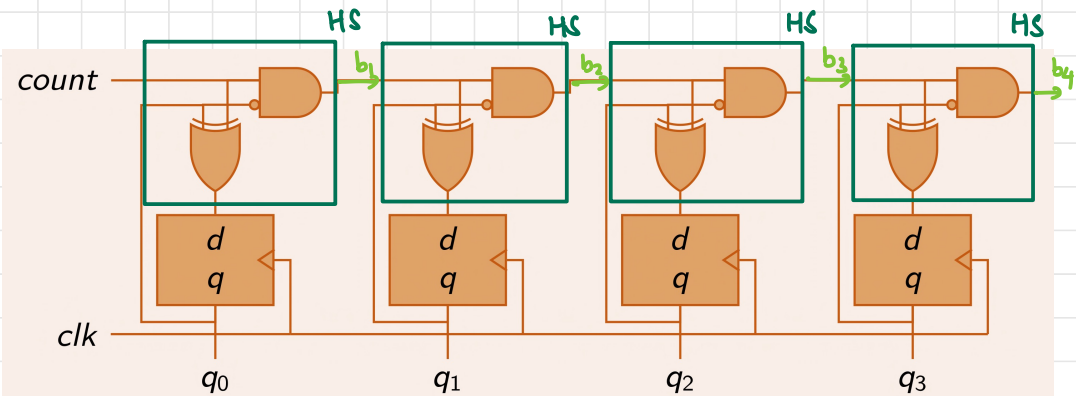
4-bit incrementer counter



- only stores incremented value when clk = ↑

DECREMENTOR COUNTER

4-bit decrementer counter

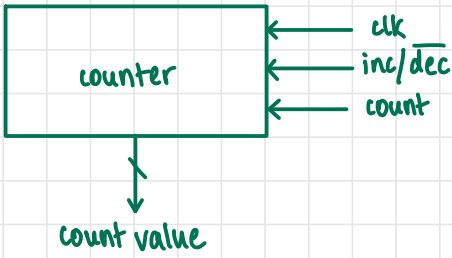


Applications

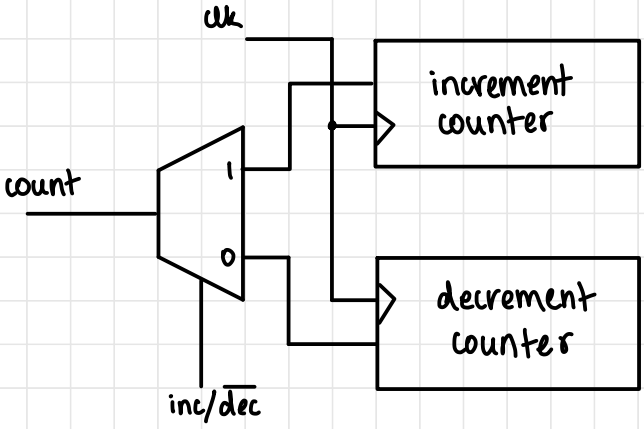
1. Iterations in logic design (n×n-bit shift-add multiplier requires n iterations)
2. Interrupt timers (schedulers in OS)
3. Software timeouts (loading webpage)

Question 7

Construct a combined incrementor / decrementer counter

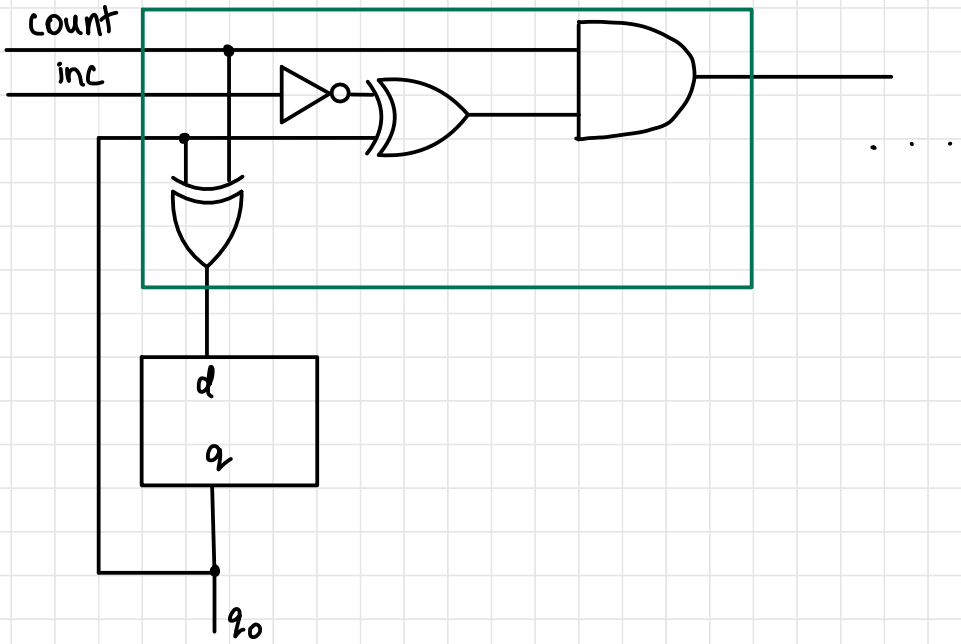


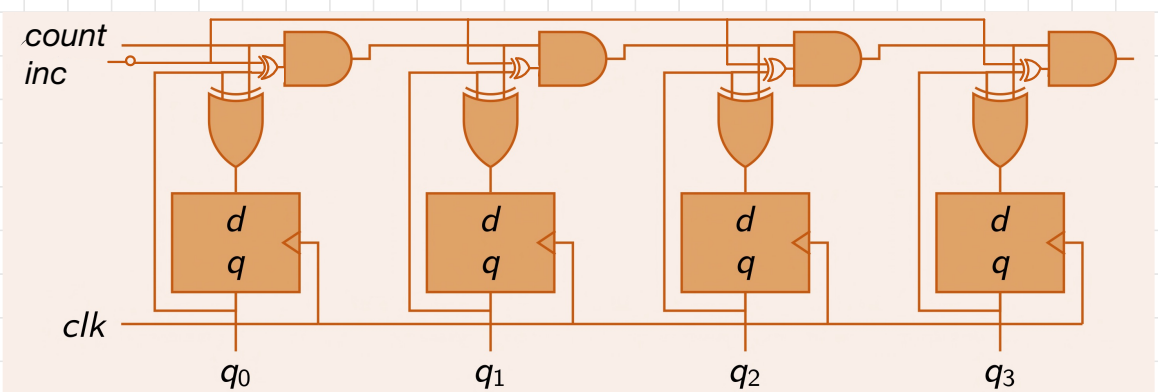
1. Approach: use a mux to either increment or decrement



2. Different approach

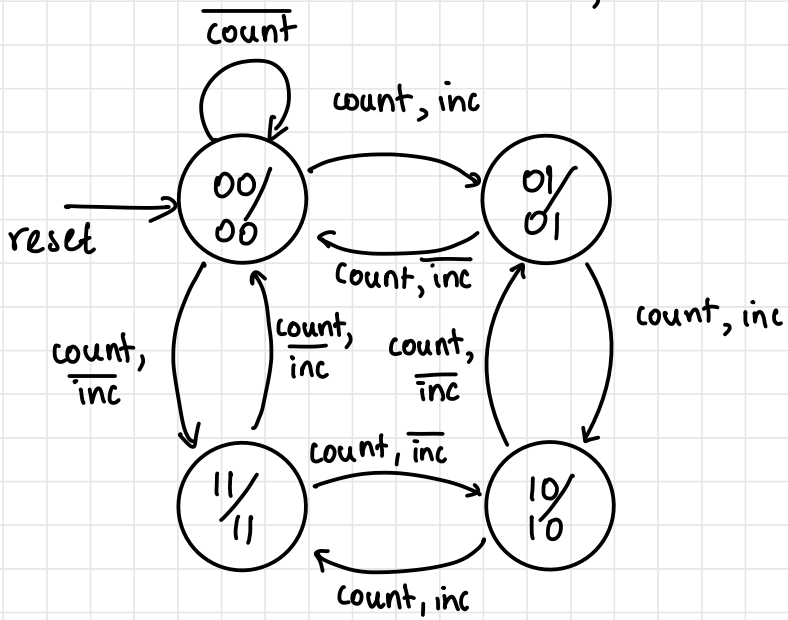
Combined HA/HS





3. Using FSMs

if inc = 1, up
inc = 0, down



State Transition Table

Current State		Inputs		Next State		Output	
$S_1(t)$	$S_0(t)$	count	inc	$S_1(t+1)$	$S_0(t+1)$	Z_1	Z_0
0	0	0	X	0	0	0	0
0	0	1	0	1	1	0	0
0	0	1	1	0	1	0	0
0	1	0	X	0	1	0	1
0	1	1	0	0	0	0	1
0	1	1	1	1	0	0	1
1	0	0	X	1	0	1	0
1	0	1	0	0	1	1	0
1	0	1	1	1	1	1	0
1	1	0	X	1	1	1	1
1	1	1	0	1	0	1	1
1	1	1	1	0	0	1	1

Output

$$z_0 = s_0(t)$$
$$z_1 = s_1(t)$$

Next State

$s_1(t)\,s_0(t)$ \ count inc	00	01	11	10
00	0	0	0	1
01	0	0	1	0
11	1	1	0	1
10	1	1	1	0

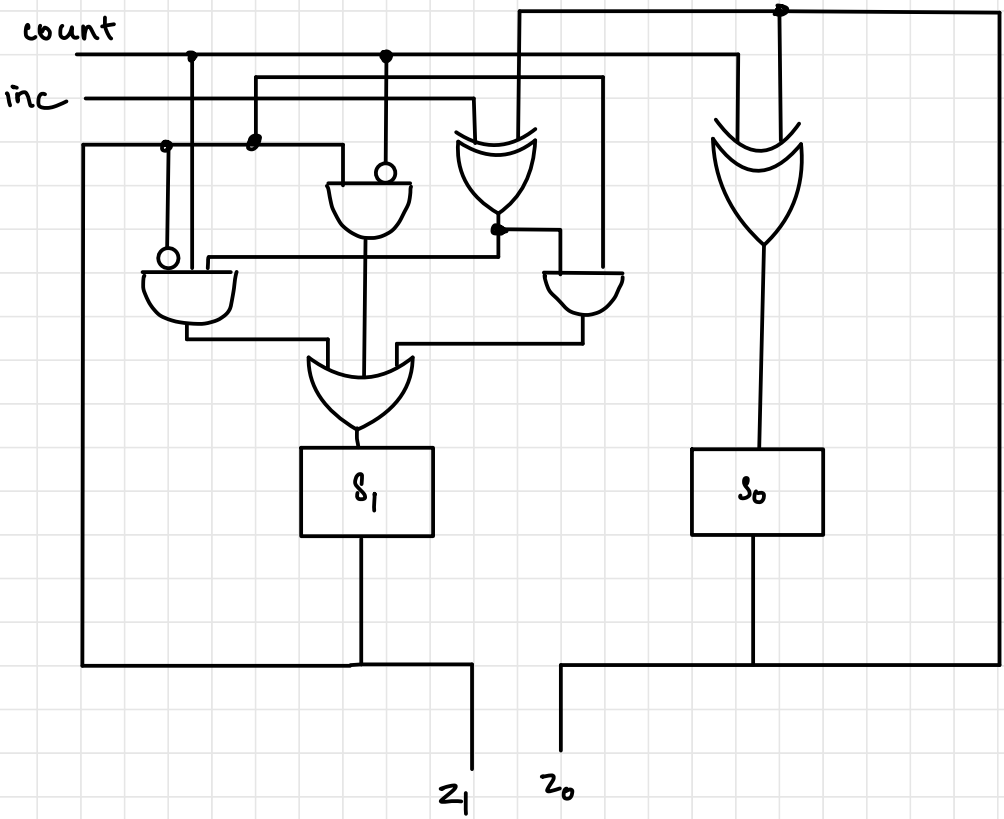
$$s_1(t+1) = \overline{count}\, s_1(t) + s_1(t)\, s_0(t)\, \overline{inc} + s_1(t)\, \overline{s_0(t)}\, inc + \overline{s_1(t)}\, s_0(t)\, count\, inc + \overline{s_1(t)}\, \overline{s_0(t)}\, count\, \overline{inc}$$

K-map

$s_1(t)\,s_0(t)$ \ count inc	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	1	1	0	0
10	0	0	1	1

$$s_0(t+1) = = \overline{count}\, s_0(t) + \overline{s_0(t)}\, count$$

Logic circuit



$$S_1(t+1) = \overline{count}\, S_1(t) + S_1(t) S_0(t) \overline{inc} + S_1(t) \overline{S_0(t)}\, inc + \overline{S_1(t)} S_0(t)\, count\, inc + \overline{S_1(t)}\, \overline{S_0(t)}\, count\, \overline{inc}$$

$$= \overline{count}\, S_1(t) + S_1(t) [S_0(t) \oplus inc] + \overline{S_1(t)}\, count\, (S_0(t) \oplus inc)$$

$$=$$

$$S_0(t+1) = = \overline{count}\, S_0(t) + \overline{S_0(t)}\, count = count \oplus S_0(t)$$

ARBITRARY MODULUS COUNTER

- 0 to k-1
- select $2^{n-1} < k < 2^n$
- start with an n-bit (modulus 2^n) incrementing counter
- ability to detect when the count value has reached k-1
- ability to reset the value to 0 when k-1 is reached

Approach

- D-flip flops with reset
- And gate for k-1
- Some inputs may need to be inverted (minterm)

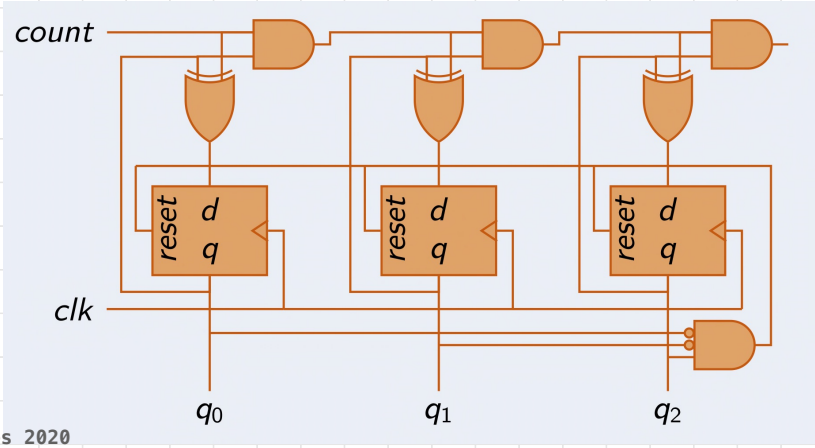
Question 8

k=5 (0-4, 0-4 ...) counter

Count sequence: 000, 001, 010, 011, 100, 000 . . .

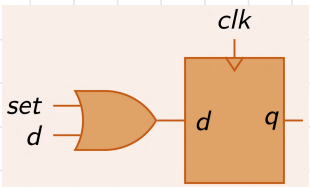
$$2^2 < 5 < 2^3$$

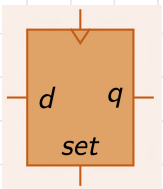
when $q_2q_1q_0 = 100$, reset signal



Settable Flip Flop

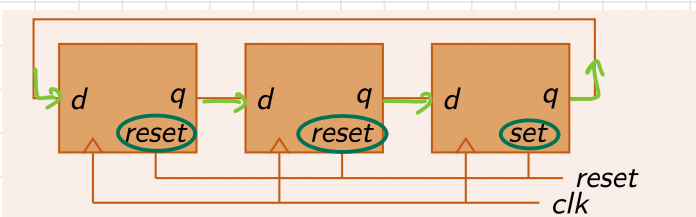
- when set = 1, value stored is 1
- else as usual
- used in ring counters





Ring Counter

- if reset signal applied, all d flip flops become 0 and no counting happens.
- one settable flip flop used with reset signal
- n-modulo counter

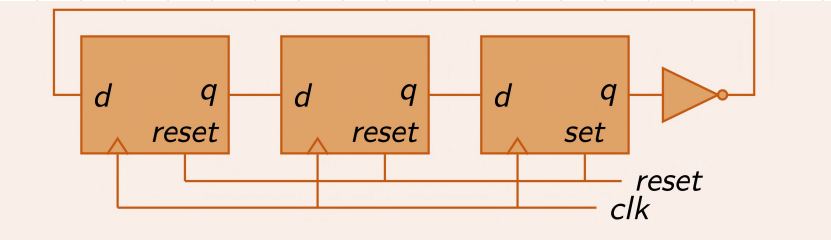


- initial value: 001
- next clock cycle: 100
- next clock cycle: 010
- next clock cycle: 001

- Also called one-hot counter (just like one-hot encoding)

- Seems less efficient in terms of space, but faster clock speeds attainable

n-Bit Johnson Counter

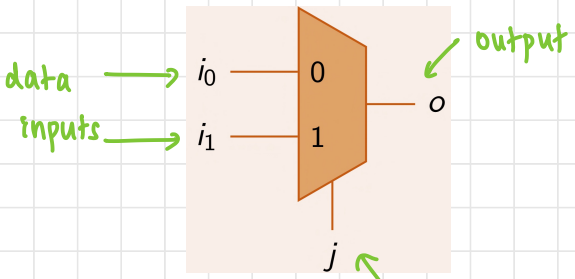


- initial value: 001
- next cycle : 000
  100
  110
  111
  011
  001
  000

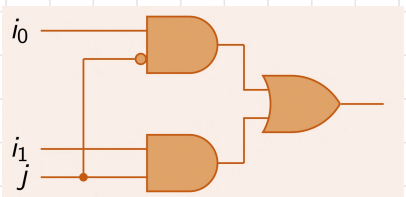
1
0
4
6
7
3
1
0

Demultiplexers

MUX

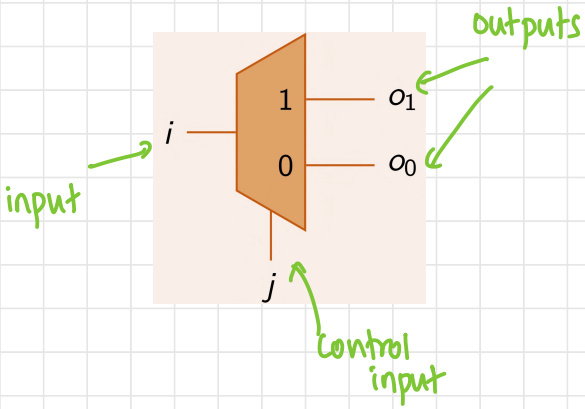


Logic circuit

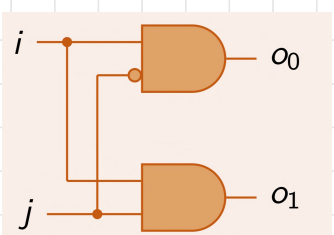


$o = \bar{j} i_0 + j i_1$

DEMUX



Logic circuit

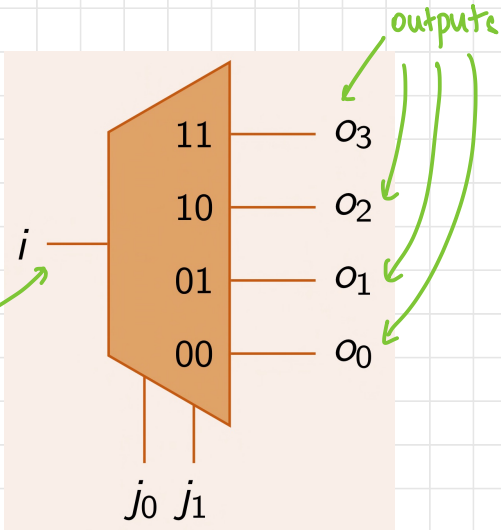


$o_0 = \bar{j} i$

$o_1 = j i$

- data direction opposite

1:4 Demux



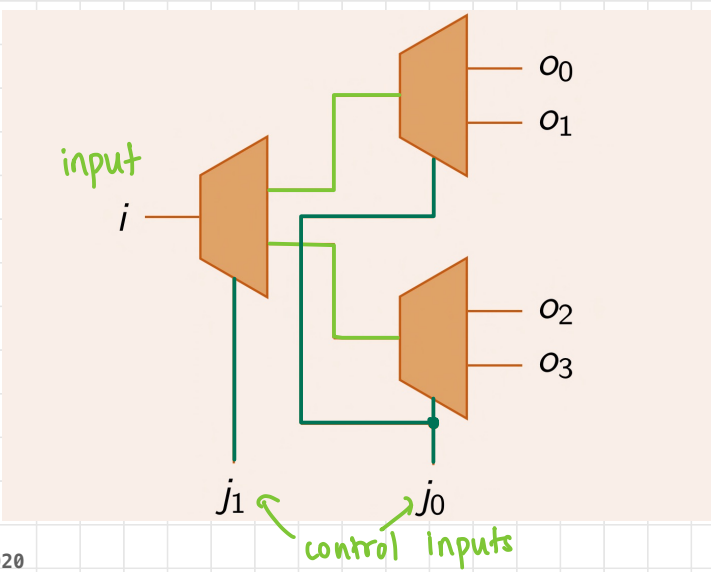
$o_3 = j_1 j_0 i$

$o_2 = j_1 \bar{j}_0 i$

$o_1 = \bar{j}_1 j_0 i$

$o_0 = \bar{j}_1 \bar{j}_0 i$

1:4 demux using 1:2 demuxes



1:n demux

- no. of outputs = n
- no. of control inputs = $\lceil \log_2(n) \rceil$ ← ceiling
- no. of inputs = 1

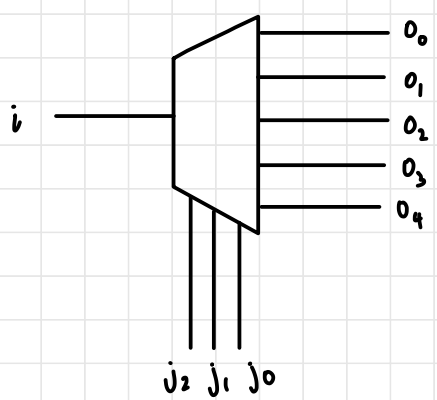
- similar to mux where no. of inputs = n, no. of control inputs = $\lceil \log_2(n) \rceil$, no. of outputs = 1

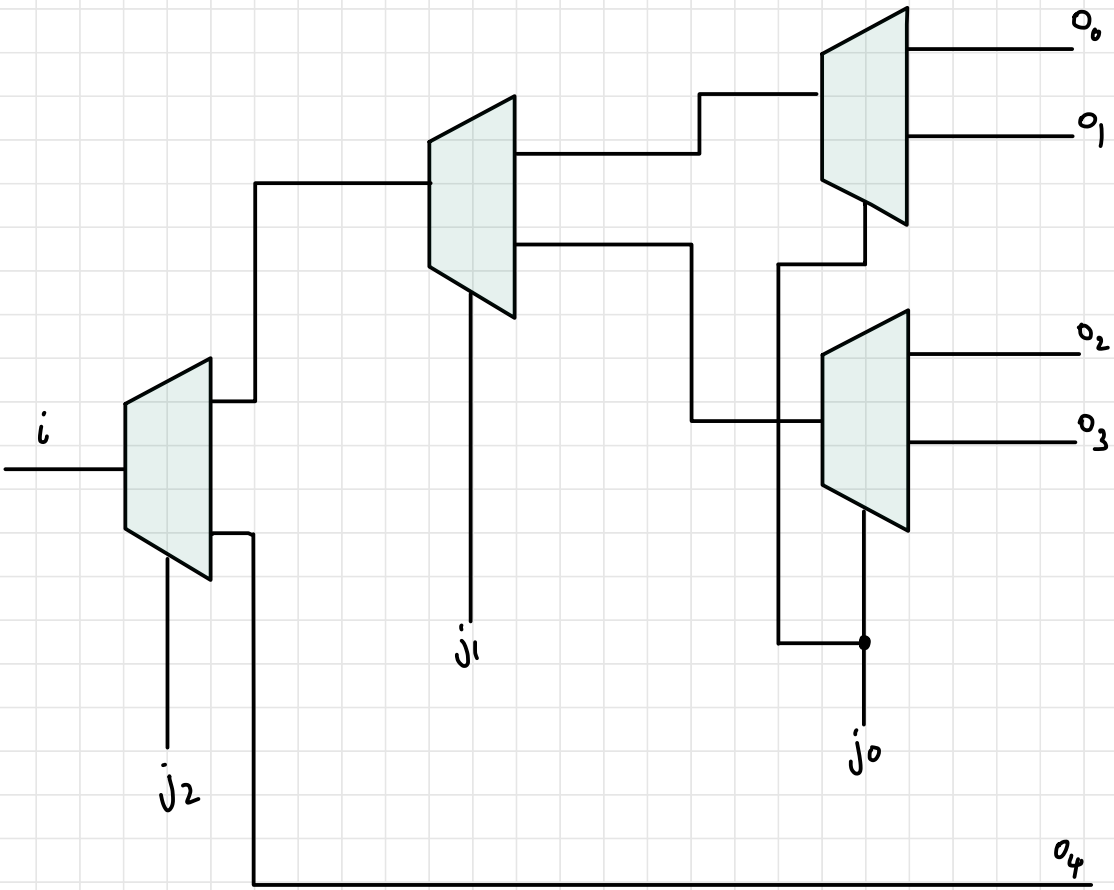
Question 9

Construct a 1:5 demux using
(a) 1:2 demuxes
(b) AND, OR and NOT gates

(a)

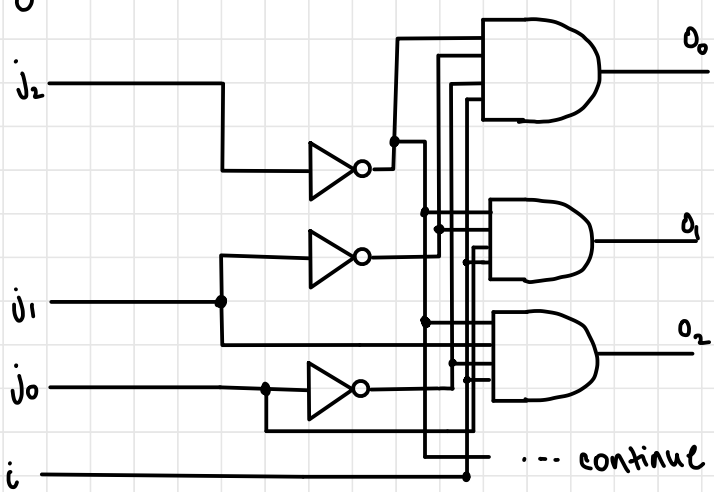
$$O_0 = \bar{j}_2 \bar{j}_1 \bar{j}_0 i$$
$$O_1 = \bar{j}_2 \bar{j}_1 j_0 i$$
$$O_2 = \bar{j}_2 j_1 \bar{j}_0 i$$
$$O_3 = \bar{j}_2 j_1 j_0 i$$
$$O_4 = j_2$$





(b) AND, OR and NOT gates

$$O_0 = \bar{j}_2 \bar{j}_1 \bar{j}_0 i$$
$$O_1 = \bar{j}_2 \bar{j}_1 j_0 i$$
$$O_2 = \bar{j}_2 j_1 \bar{j}_0 i$$
$$O_3 = \bar{j}_2 j_1 j_0 i$$
$$O_4 = \bar{j}_2$$



Memory Arrays

- high-level arrays : name + index (software)
- 2 operations : read array location, write at location
- implementation in hardware)

Read

```
x = a[i];
```

operation performed at RHS

input = index

Write

```
a[i] = x;
```

input = index

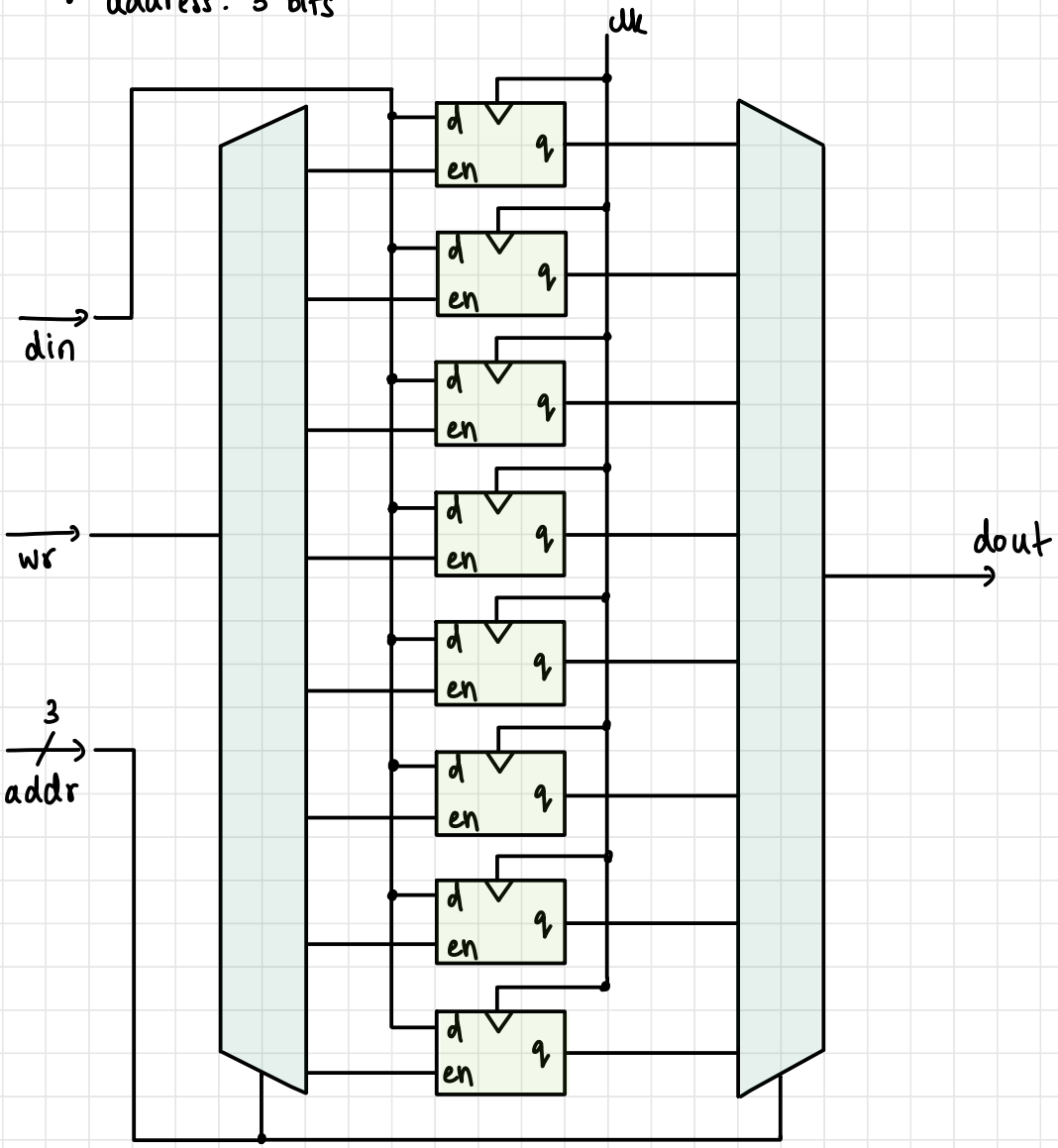
input = data

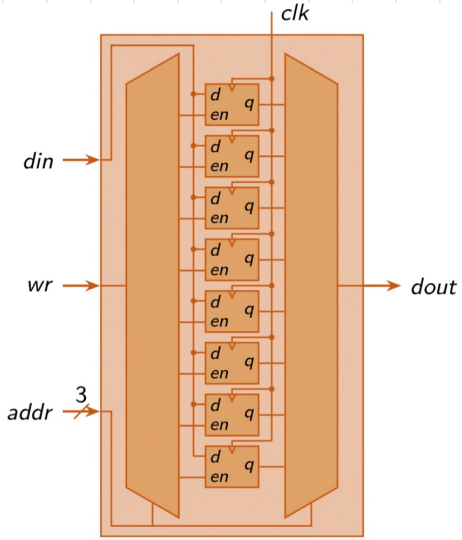
- index : software, addr : hardware (address)
- size of index (software) → 32 bit int (eg)
- size of index (hardware) → 8 bit (1 byte) ≤ 256
  → 9 bit → ≤ 512
- generally, for n memory locations, $\lceil \log_2 n \rceil$ bits for address

Simplest Memory Array

8 memory locations, 1-bit storage

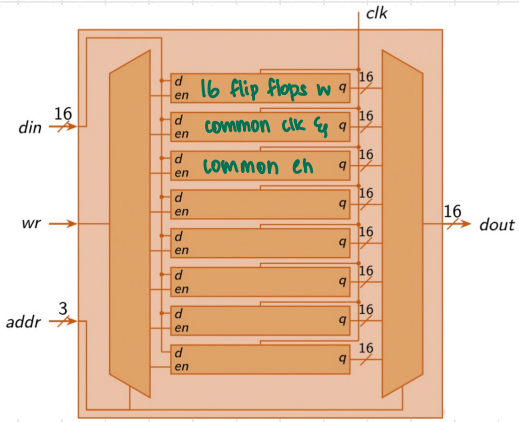
- 8 d-flip flops with enable used to store values at 8 different locations
- clock inputs are common
- address: 3 bits





- called random access memory

8 memory locations, 16-bit words
8 word x 16 bit
8 x 16



Memory Port

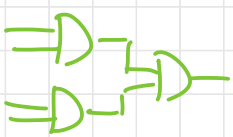
- set of signals that provide read/write access
- one read/write port: addr, wr, din and dout

Carry-Lookahead & Prefix Adder

- Evaluate performance by estimating area and time requirements

For 2-input AND/OR/XOR gates
- Area of each AND, OR, XOR gate estimated to be a_g
- Propagation delay of every 2-input gate estimated to be t_g

For k-input AND/OR/XOR gates
- Area = $(k-1)\,a_g$
- Propagation delay = $\lceil \log_2 k \rceil\, t_g$ → tree design, like MUX



Ripple Carry Adder Area and Time

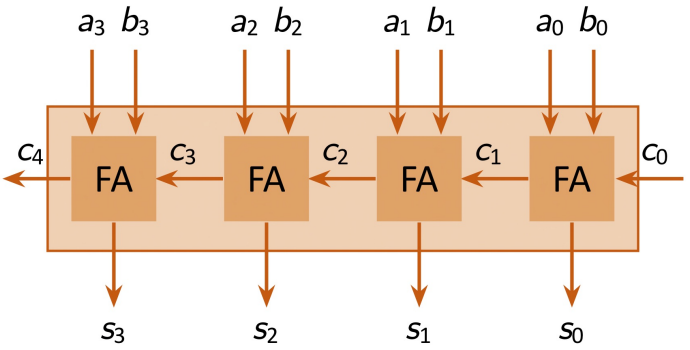
- Sum: 1 3-input gate
- Carry: 5 2-input gates

Area requirements
- n-bit ripple carry adder occupies $7n\,a_g$ area

Time requirements
- Propagation delay from c_0 to c_{n-1}
- signal passes through three gates in each of the n-1 stages
- delay = $3(n-1)\,t_g$

- Sum computation: $2t_g$ time required for 3 input XOR gate

- n-bit ripple carrier takes $(3n-1)\,t_g$ time



- To reduce delay, carry-lookahead & prefix adders
- Computation time increases linearly with no. of gates (very slow)
- Speed can be improved by eliminating the carry chain

Compute Carry Values Directly from a_i & b_i

$$c_1 = a_0b_0 + a_0c_0 + b_0c_0$$
$$= a_0b_0 + (a_0+b_0)c_0$$

$$c_2 = a_1b_1 + (a_1+b_1)c_1$$
$$= a_1b_1 + (a_1+b_1)(a_0b_0 + (a_0+b_0)c_0)$$
$$= a_1b_1 + (a_1+b_1)a_0b_0 + (a_1+b_1)(a_0+b_0)c_0$$

Let us make the following substitution to simplify

$$g_i = a_ib_i$$
$$p_i = a_i+b_i$$

$c_1 = g_0 + p_0 c_0$ (1, 1, 1)] 1 = 4

$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0 = 10$

$c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0 = 18$

$c_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0 = 28$

Why p & g?

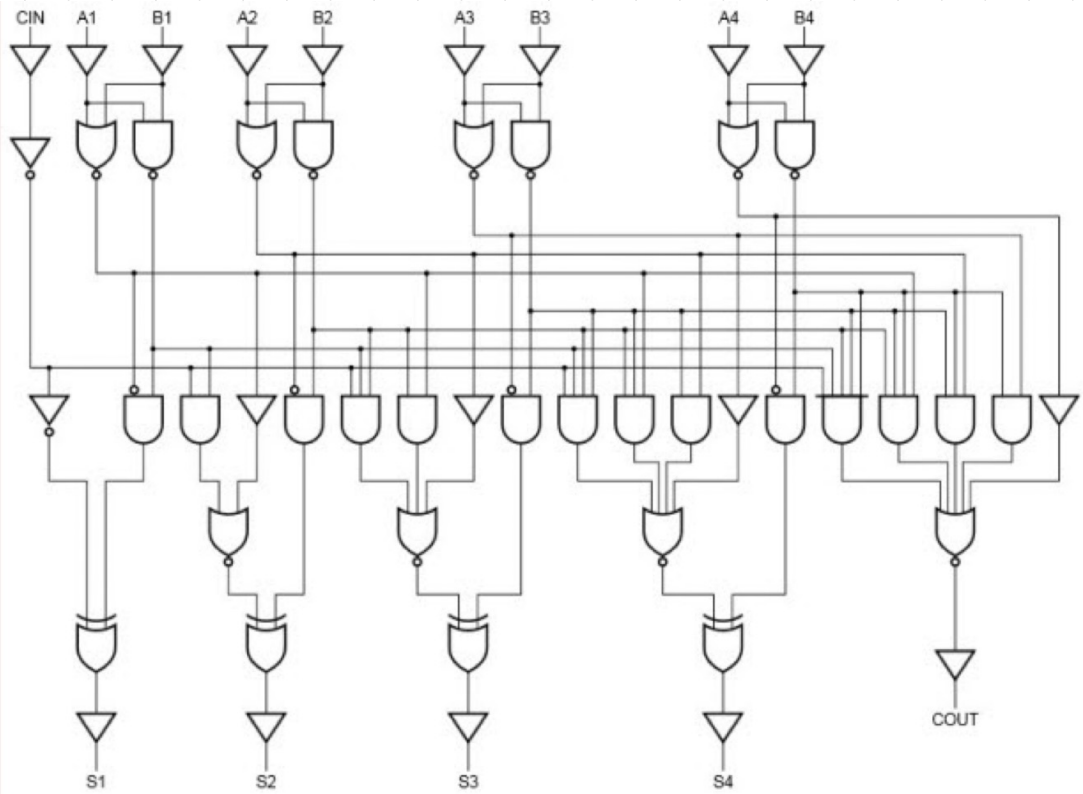
Consider c_3

- $g_2 = a_2 b_2$
- if $a_2 = 1$ & $b_2 = 1$, c_3 is guaranteed to be high
- ∴ g stands for **generate**

- $p_2 = a_2 + b_2$
- if carry is generated at 2 & propagated to 3, c_3 is high
- if carry is generated at 1 & propagated to 2 & then 3, c_3 is high
- ∴ p stands for **propagate**

Circuit Diagram

- bubbles & inverters can be ignored
- here, 1-4 & not 0-3



- complexity increases from 1 to 4
- from left to right, complexity increases very rapidly

Performance Estimate

- for c_1, 4 gates required — 4
  for c_2, 10 gates total — 6
  for c_3, 18 gates total — 8
  for c_4, 28 gates total — 10

- for c_i, no of gates required is $2i+2$ greater than gates required for c_{i-1}

- So $i(i+3)$ gates required from c_1 to c_i — n^2+3n for c_1 to c_n

- Each three input XOR gate counts as 2 gates

- $s_1 \longrightarrow$ 2 XOR
  $s_2 \longrightarrow$ 2 XOR
  $s_3 \longrightarrow$ 2 XOR
  $s_4 \longrightarrow$ 2 XOR
  ] 2n

- n-bit carry lookahead adder would require n^2+5n gates

Time Estimate

$$c_1 = g_0 + p_i c_0$$
$$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$$
$$c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$$
$$c_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$$

Critical path delay depends on

1) p_i and g_i computation

- t_g time required for 2-input AND / OR gate

2) Carry computation delay

- time required for c_i depends on i
- longest delay: c_{n-1} term
- delay for minterm $p_{n-2}\, p_{n-3} \cdots p_0 c_0$ is longest (last minterm)
- time required $\lceil \log_2(n) \rceil t_g$ for n-input AND gate
- delay for the OR of all minterms requires $\lceil \log_2(n) \rceil t_g$ time

3) Sum computation

- $2t_g$ time for 3-input XOR gate

Total delay

$$2\lceil \log_2(n) \rceil t_g + 3t_g$$

Performance Comparison

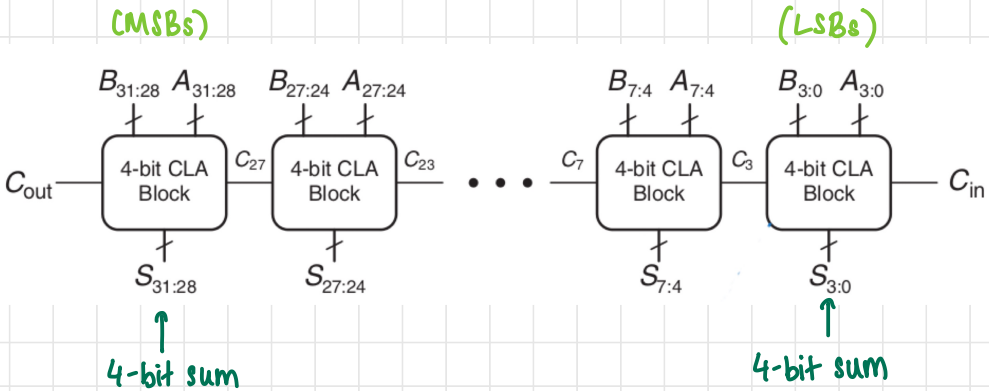
	Area	Time
Ripple carry	$7na_g$	$(3n-1)t_g$
Carry-lookahead	$(n^2+5n)a_g$	$2\lceil \log_2(n) \rceil t_g + 3t_g$

- time increases as $\log_2 n$, which is much faster than a ripple carry adder
- area increases as n^2 which is difficult to scale

Hybrid Approach Solution

- split adder into a number of blocks
- use carry lookahead technique to add bits in each block
- blocks combined together using ripple carry technique

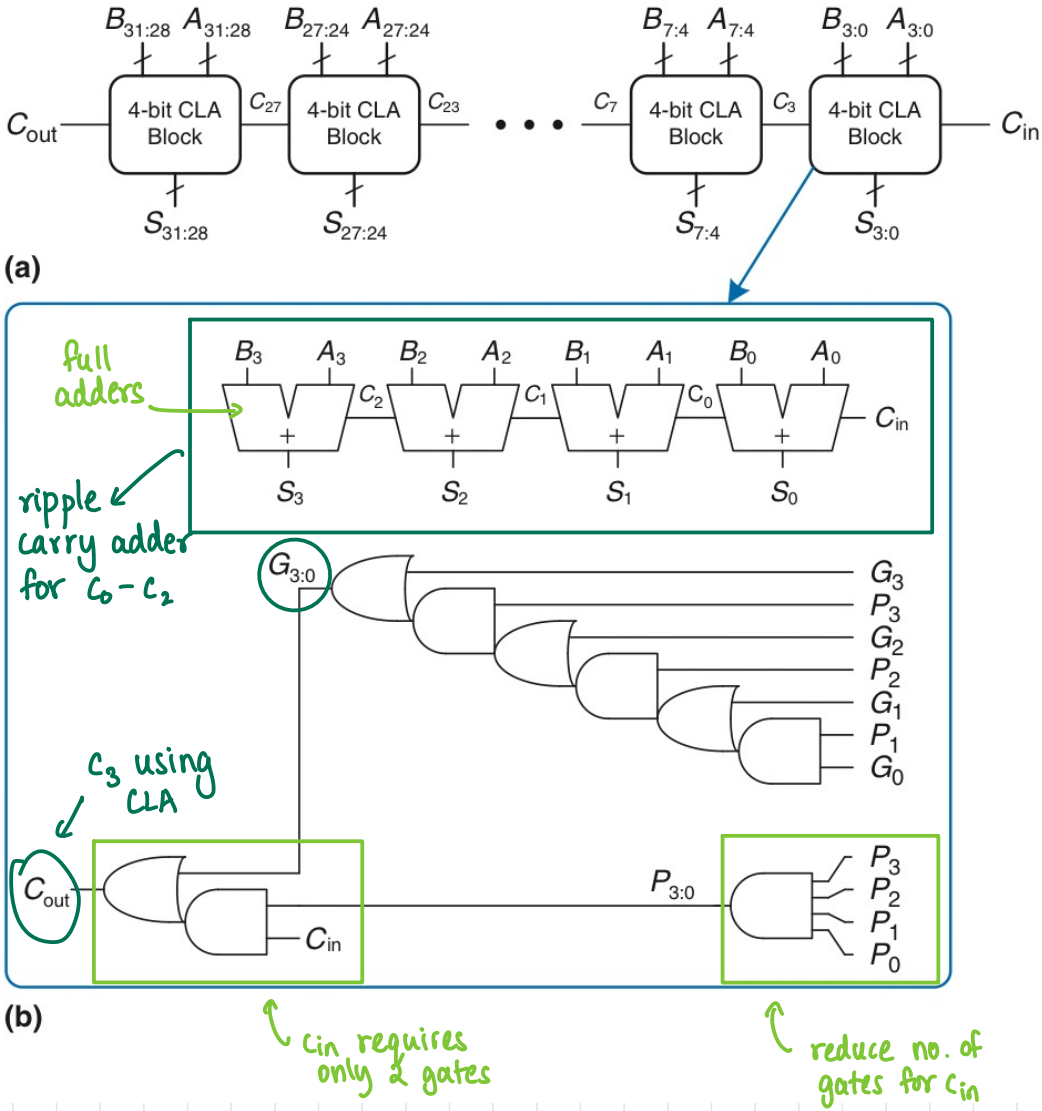
- 32-bit adders use 4-bit blocks



- Better speed than ripple carry adder and better space than carry-lookahead

Further Optimisation

- Critical path: a_0, b_0, c_{in} to s_{31}
- Therefore, $c_3, c_7, \ldots, c_{27}$ need to be computed quickly
- Not essential to compute s_0 to s_{30} quickly
- Can use ripple carry inside each block to compute sum outputs (save space)
- Use CLA only for $c_3, c_7, \ldots c_{27}$ in each block



$$C_{out} = g_3 + p_3g_2 + p_2p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_{in}$$

$$= g_3 + p_3(g_2 + p_2g_1 + p_2p_1g_0) + p_3p_2p_1p_0c_{in}$$

$$= g_3 + p_3(g_2 + p_2(g_1 + p_1g_0)) + p_3p_2p_1p_0c_{in}$$

c_{in} takes longest to come from previous block

can be computed ahead of time

Critical Path Delay

- Three parts
  1) compute all p's and g's
  2) carry needs to propagate from c_0 to c_{27}
  3) compute sum s_{31}

1) Time taken to compute various p & g values

all t_{pg}'s at once

- Compute p_i & g_i $(0 \le i \le 4)$ in each block in time $\boxed{t_{pg}}$
- Compute $g_{3:0}$ in each block in time $\boxed{t_{pg\text{-}block}}$ ← computed parallelly

$$g_i = a_i b_i \qquad p_i = a_i + b_i$$

2) Time taken for carry to propagate from c_0 to c_{27}

- In each block, c_{in} propagates through one AND & one OR block in time $t_{AND\text{-}OR}$
- Since carry propagates in the above manner through first seven blocks, time required is $\boxed{7t_{AND\text{-}OR}}$ $(0, 3, \ldots 27)$

3) Time taken to compute s_{31}

- Once c_{27} is available, needs to propagate through four full adders, each of which takes time $\boxed{t_{FA}}$
- Time required is $4t_{FA}$

Critical Path Delay

$$\boxed{t_{CLA} = t_{pg} + t_{pg\text{-}block} + 7t_{AND\text{-}OR} + 4t_{FA}}$$

Generalising

- N-bit adder using K-bit blocks
- $\frac{N}{K}$ blocks each of size K used

$$t_{CLA} = t_{pg} + t_{pg\text{-}block} + \left(\frac{N}{K} - 1\right) t_{AND\text{-}OR} + K\, t_{FA}$$

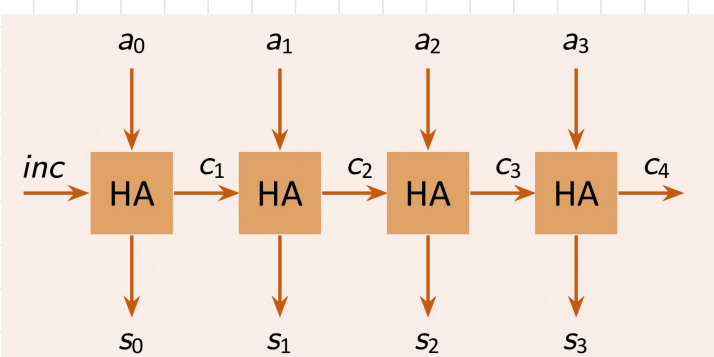
Parallel Prefix Adder

- Requires less area than CLA
- Faster than RCA

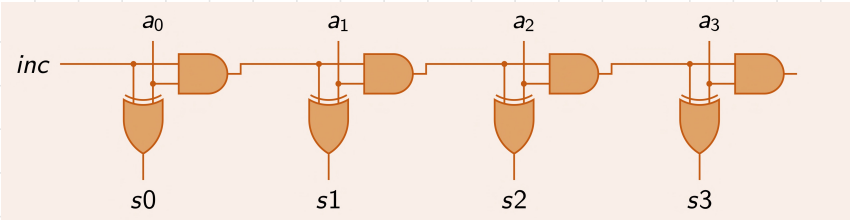
Parallel Prefix Incrementor

- Simpler than adder
- In order to understand

- Ripple carry incrementor

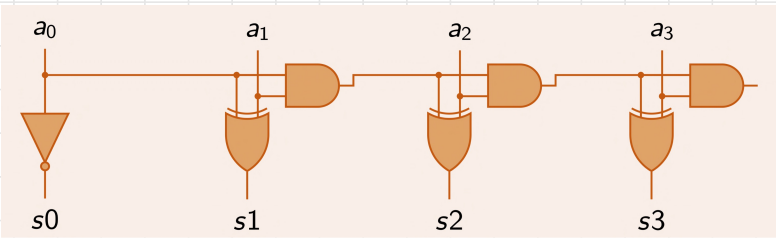


Using Gates



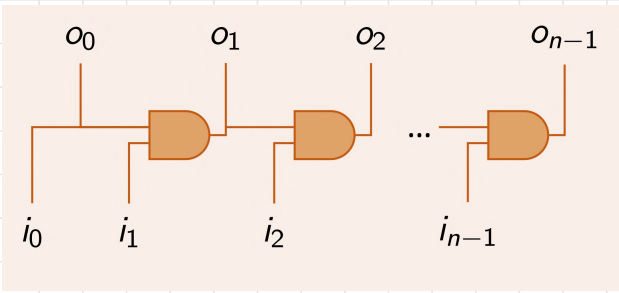
Without inc signal

- inc always 1



Incrementer Carry Chain

- only carry chain
- assuming 2-input gate takes time t_g and area a_g
- carry chain: $(n-1)\,a_g$ and $(n-1)\,t_g$



- Given n inputs $i_0, i_1, i_2, \ldots i_{n-1}$ and n outputs $o_0, o_1, o_2 \ldots o_{n-1}$

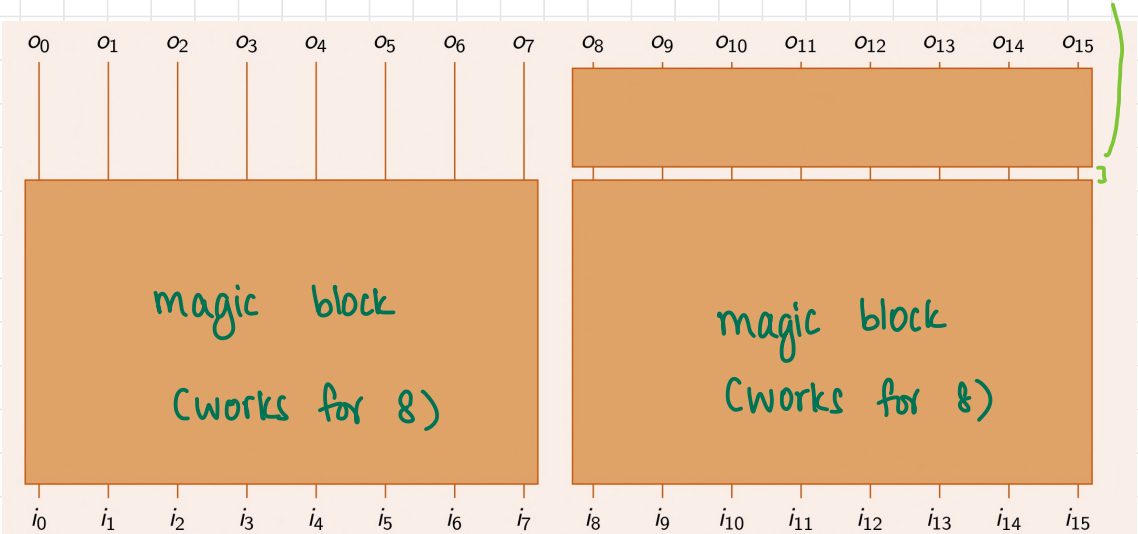
$$o_0 = i_0$$
$$o_1 = i_0 i_1$$
$$o_2 = i_0 i_1 i_2$$
$$\vdots$$
$$o_{n-1} = i_0 i_1 i_2 \ldots i_{n-1}$$

- Each output statement computed based on inputs so far (prefix of input sequence) called **prefix problem**

Parallel Prefix Computation of Incrementer Carry Chain

- 16-inputs

1. Assume it works for 8 inputs
   - Extend to 16 bits
   - Solve using recursion



- In-between wire at position 8

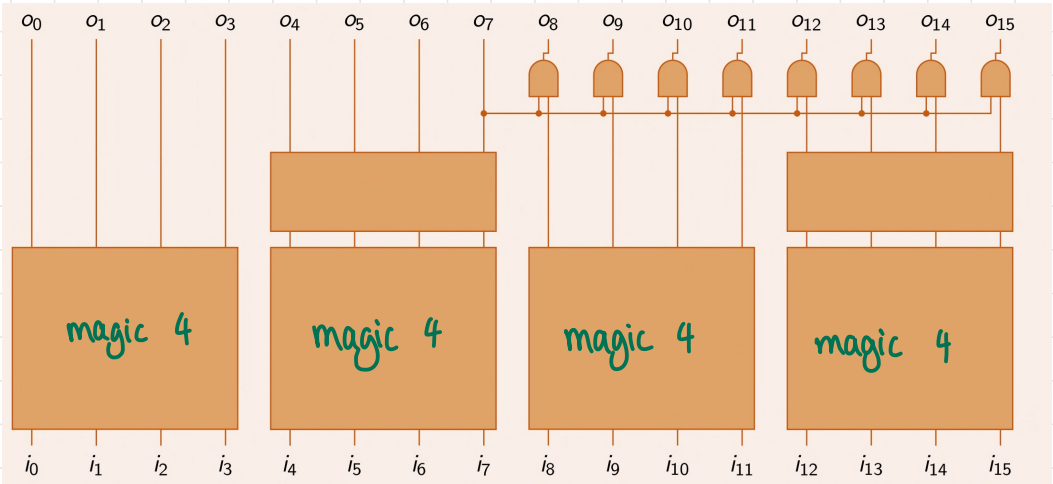
$$o_8 = i_0 i_1 i_2 \cdots i_8 \quad \text{(by definition)}$$

$$o_7 = i_0 i_1 i_2 \cdots i_7$$

$$\therefore \; o_8 = o_7 \, i_8$$

- We AND all the in-between wires with o_7

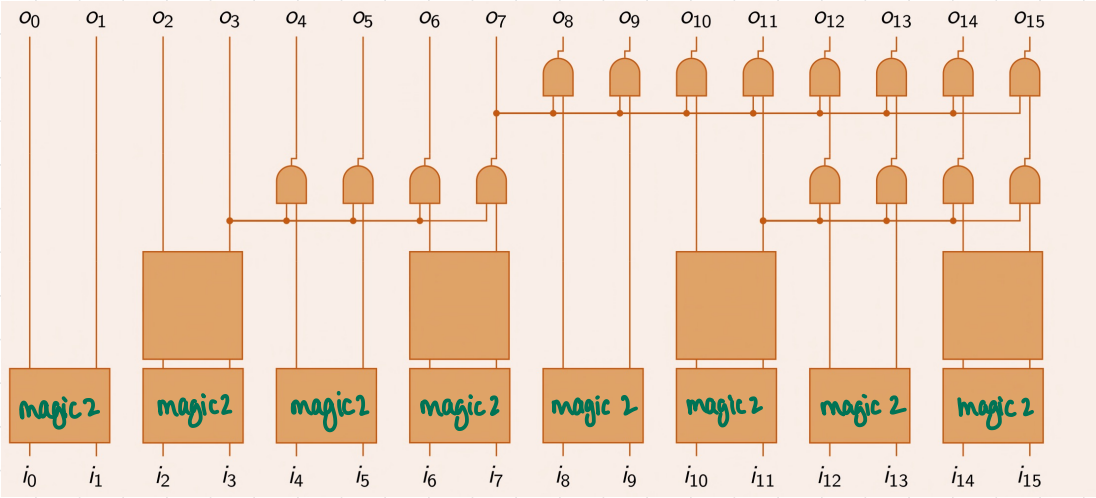
2. Assume it works for 4 inputs



$$o_4 = o_3 \, i_4 \quad \text{(by definition)}$$

- Perform the same AND operation

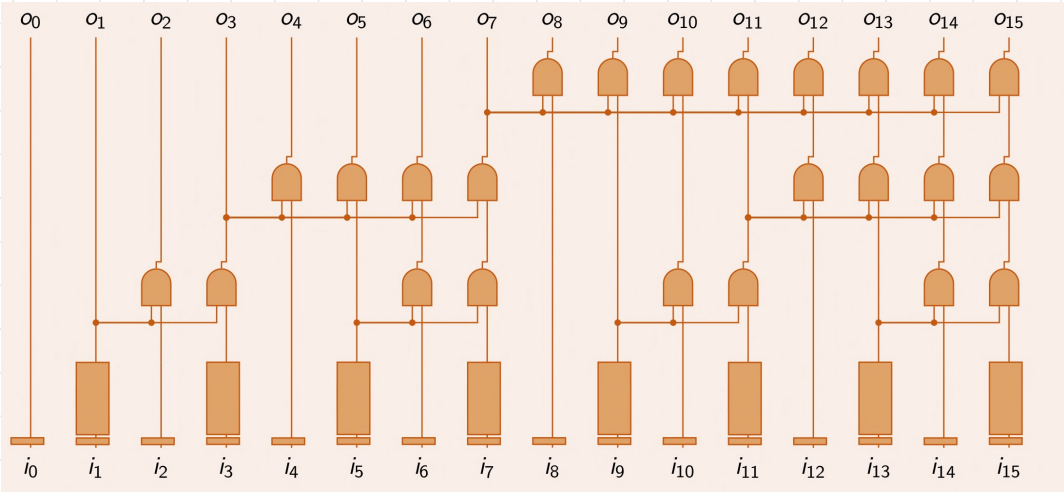
3. Assume it works for 2 inputs



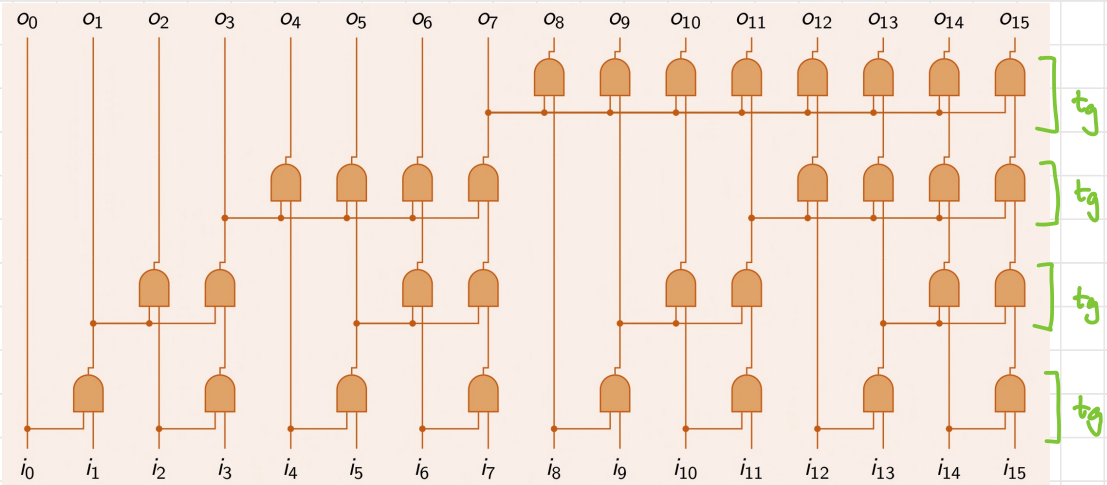
$o_2 = o_1 i_2$ (by definition)

- Perform AND operation again

4. Assume it works for 1 input



5. For 1-input, $o_0 = i_0$



- Due to parallel computation, only $4t_g$ time required
- o_3 is not computed using o_2, but it is computed with the results of $i_0 i_1$ (o_1) and $i_2 i_3$
- Reduces ripple dependencies (time)
- Area: $4 \times 8 = 32$ AND gates (each row has 8 gates)

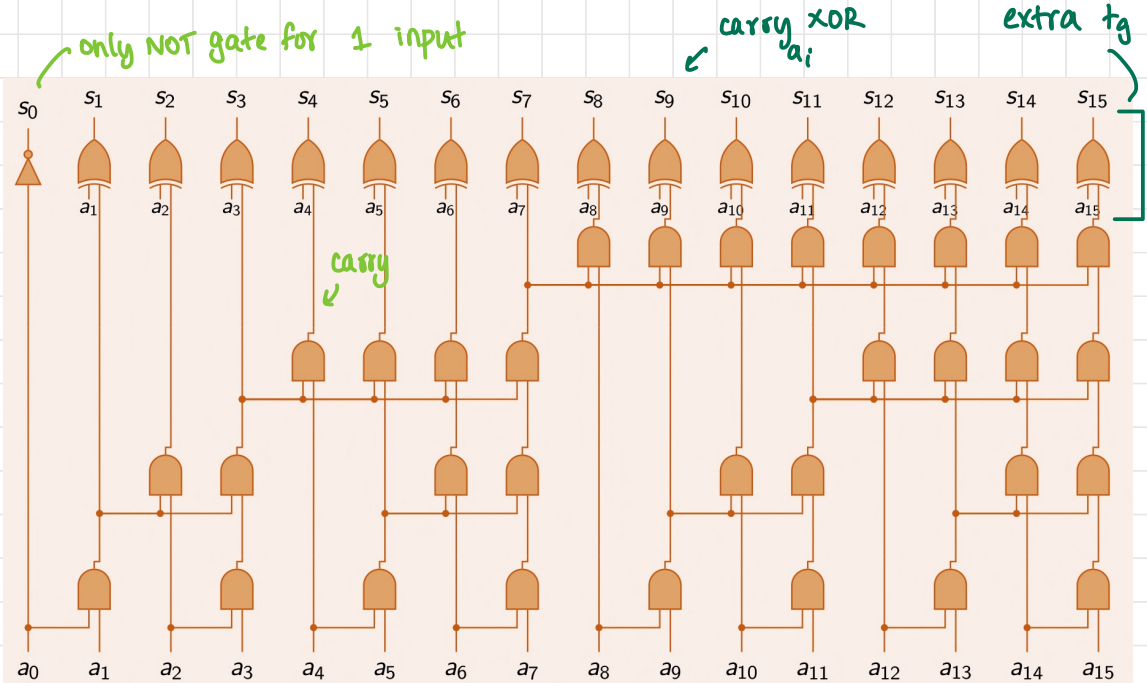
Generalising

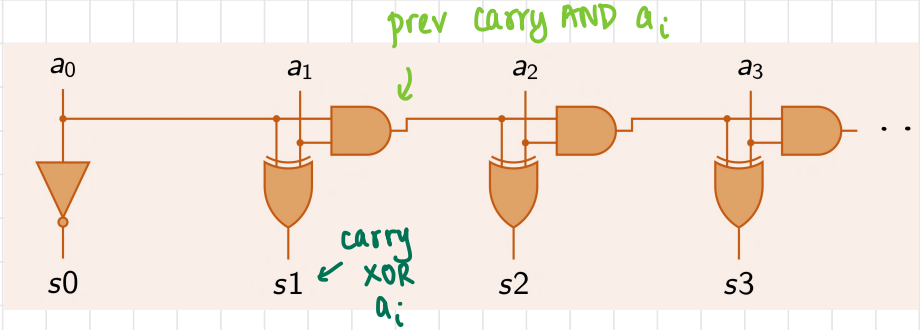
- $n/2$ AND gates per row
- $\log_2 n$ rows

Area $= (n/2)(\log_2 n)\, a_g$
Time $= (\log_2 n)\, t_g$

↙ ceil for non-powers

Parallel Prefix Incrementor





	Area	Time
Ripple carry	$2(n-1)a_g$	$(n-1)t_g$
Parallel prefix	$\frac{n}{2}(\log_2 n)a_g + (n-1)a_g$	$(\log_2 n + 1)t_g$

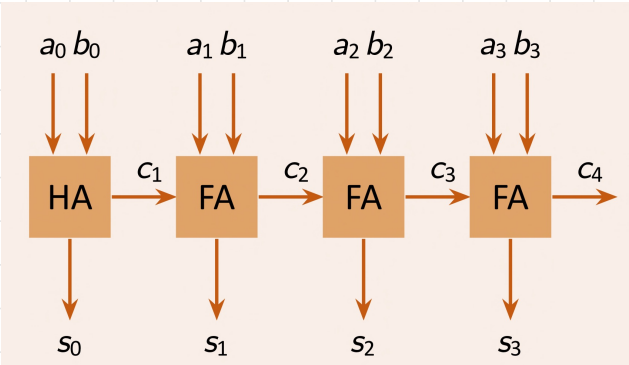
- Area less than n^2 of CLA

Parallel Prefix Applicability

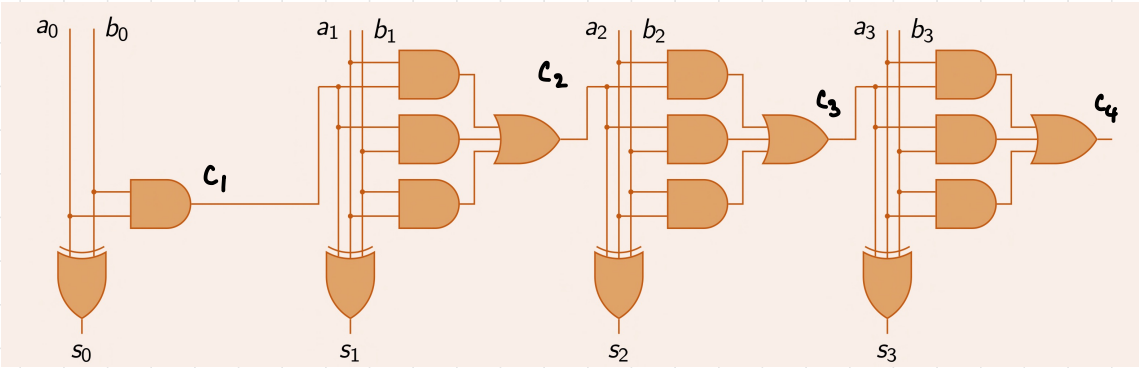
- Only works for associative functions

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c$$

4-BIT RIPPLE CARRY ADDER



Gates



$$s_i = a_i \oplus b_i \oplus c_i$$

$$c_{i+1} = a_i b_i + b_i c_i + c_i a_i = a_i b_i + c_i (a_i + b_i)$$

- is it associative?

Generate & Propagate

g_i = carry generated in position i

$g_i = a_i b_i$

p_i = carry propagated in position i

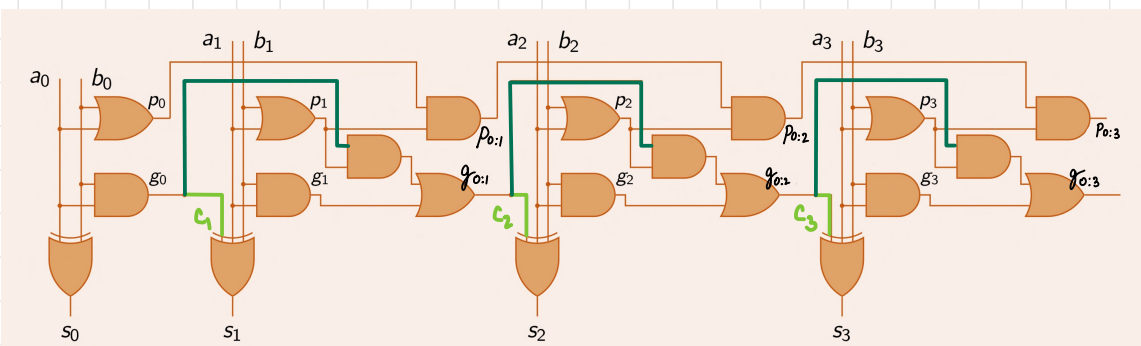
$p_i = a_i + b_i$

$c_{i+1} = g_i + p_i c_i$

$g_{0:i+1} = g_i + p_i g_{0:i}$

$p_{0:i+1} = p_i p_{0:i}$

$c_{i+1} = g_{0:i+1}$

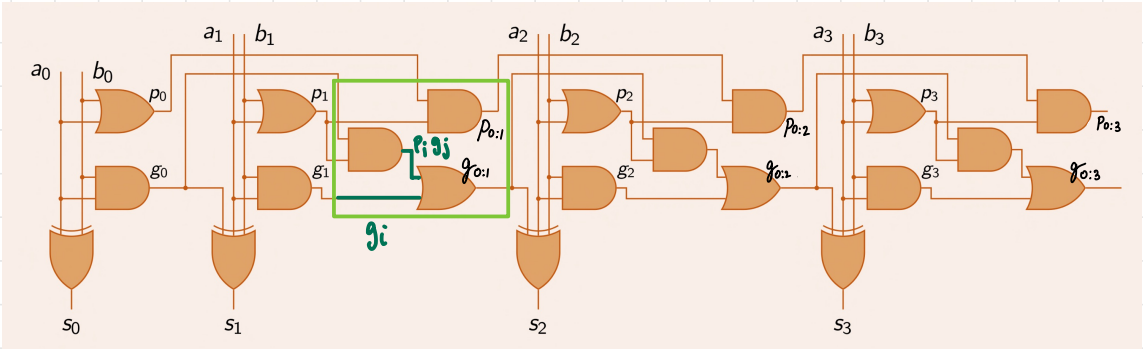


Associative Operation

times operator

$$(p_i, g_i) \otimes (p_j, g_j) = (p_i p_j, g_i + p_i g_j)$$

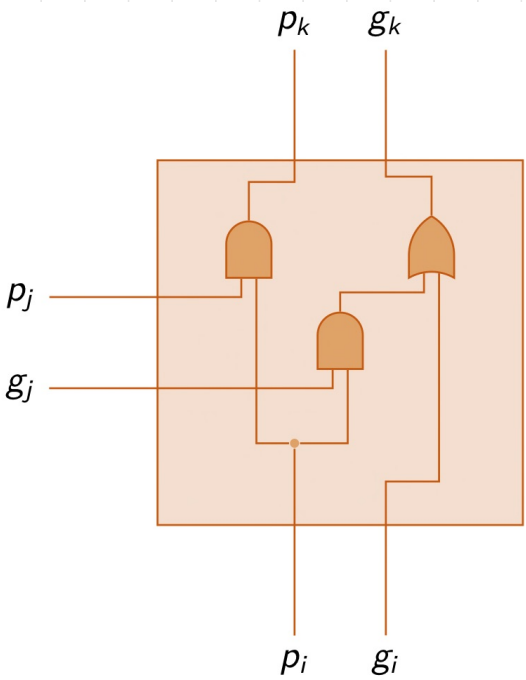
for i=1, j=0



$$(p_i g_i) \otimes ((p_j, g_j) \otimes (p_k, g_k))$$

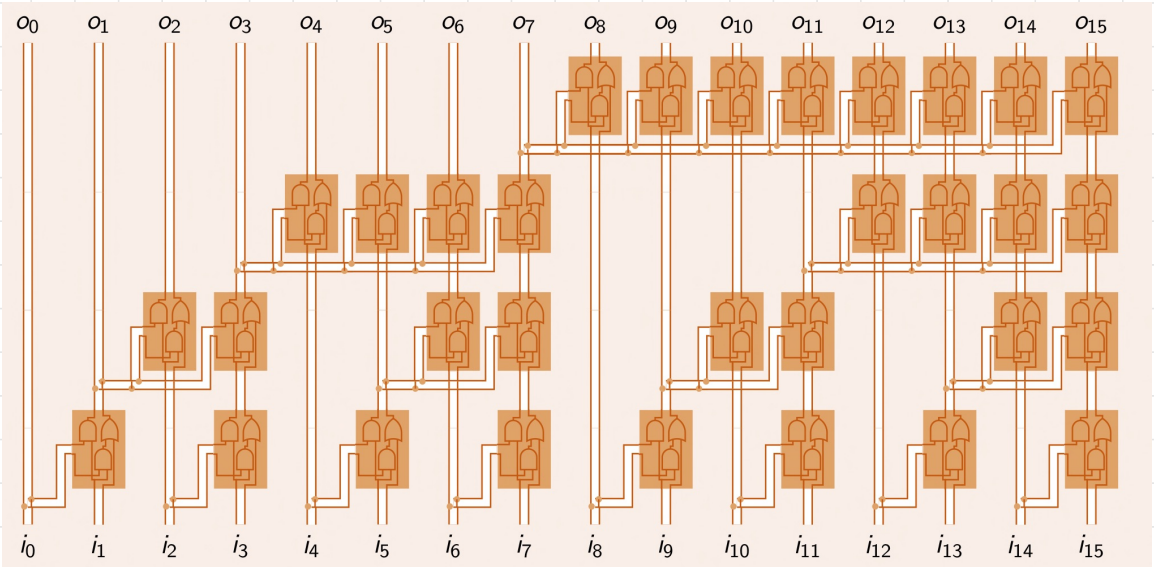
$$((p_i g_i) \otimes (p_j, g_j)) \otimes (p_k, g_k)$$

Single cell for $\otimes$ computation



Parallel Prefix for ⊗

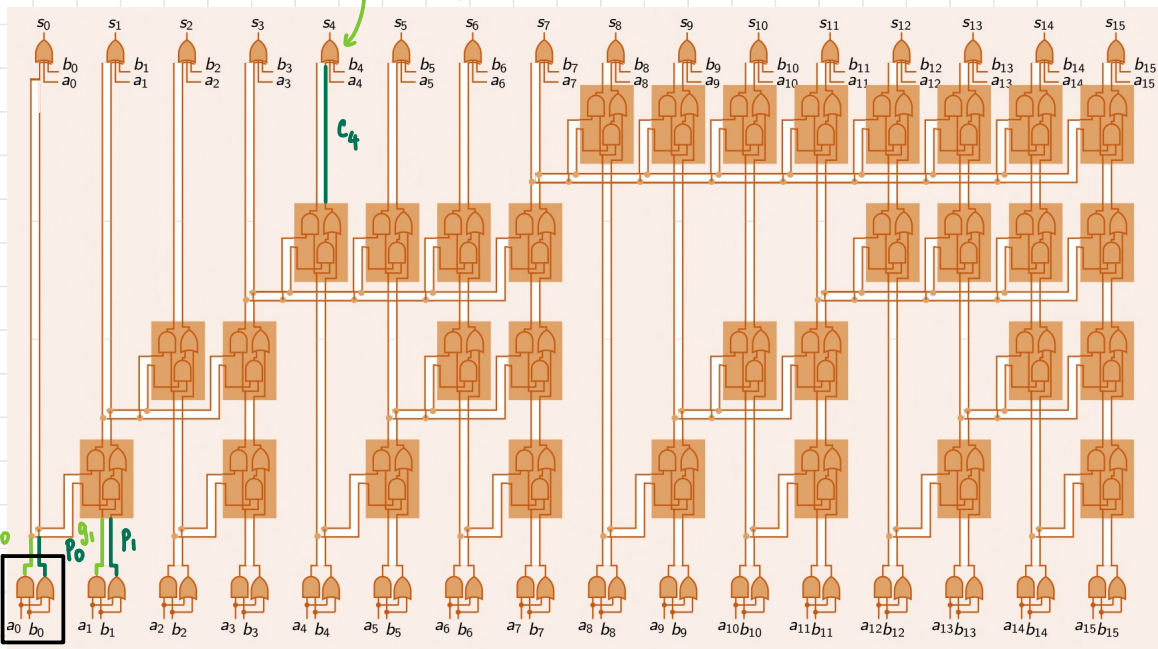
2 inputs & 2 outputs



Parallel Prefix Adder

XOR gate

(many errors in slide diagram)



computes p_i & g_i

Critical Path Delay

- Computation of p_i and g_i : t_{pg}
- $p_{i:j}$ and $g_{i:j}$ computation: t_{pg_prefix}
- s_i computation: t_{XOR}

$$t_{PA} = t_{pg} + (\log_2 N)\, t_{pg\_prefix} + t_{XOR}$$

Question 10

Compare time delay of 32 bit CLA (4-bit blocks) and 32 bit Ripple carry adder

$t_{FA} = 300\,ps \quad t_{2\text{-input}} = 10$

$t_{RCA} = 300 \times 32 = 9.6\,ns$

t_{CLA}

$$t_{CLA} = t_{pg} + t_{pg\_block} + \left(\frac{N}{K} - 1\right) t_{AND\_OR} + K\, t_{FA}$$

$g_0, p_0 \ldots g_3, p_3 \rightarrow$ can be parallel

$= t_{pg}$ time $= 100\,ps$

t_{pg_block} ($g_{0:3}$ ← 6 gates, $g_{4:7} \ldots$) $= 6\times$ 2-input delay

$= 6 \times 100 = 600ps$

$t_{AND\_OR} = 2 \times 100 = 200ps$

$t_{CLA} = 100 + 600 + 7 \times 200 + 4 \times 300$

$= 3.3\,ns$

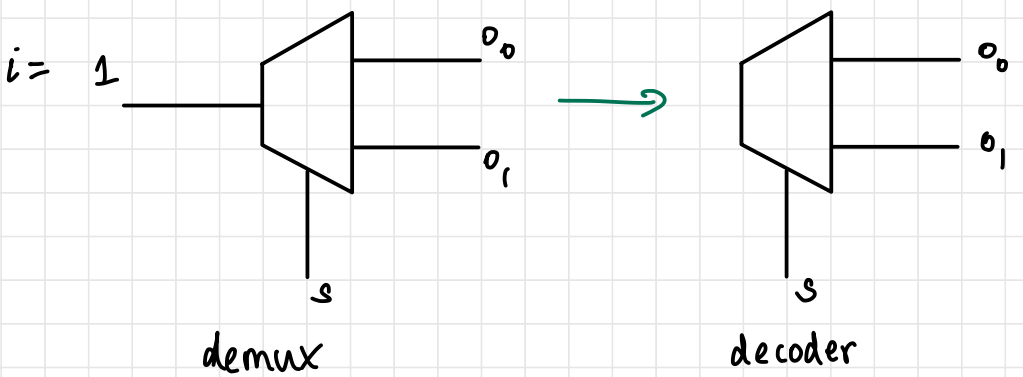
Question 11

Symmetric Boolean function

only combination of 1's & 0's matters, not permutation

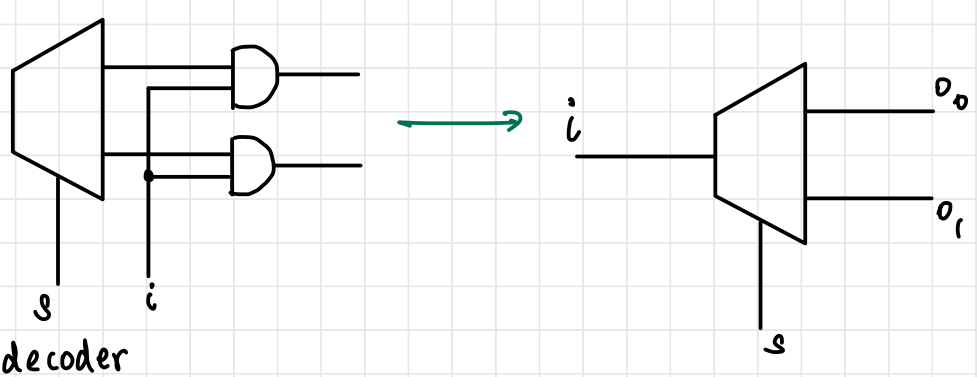
Question 12

Construct decoder from demux



Question 13

Construct demux from decoder



Question 14

Using 2's complement representation, 8-bit

(a) -0 to 255
(b) -128 to 127
(c) -128 to 128
(d) -127 to 127

Question 15

Using unsigned representation, 10-bit number can represent numbers from

0 - 1023

Question 16

How many 2-input gates are required for constructing 2:1 MUX?

3 gates (3 Nand gates)