Lecture 11: Datapath Elements

Based on material prepared by prof. Visvesh S. Sathe

Acknowledgements

All class materials (lectures, assignments, etc.) based on material prepared by Prof. Visvesh S. Sathe, and reproduced with his permission

Visvesh S. Sathe
Associate Professor
Georgia Institute of Technology
https://psylab.ece.gatech.edu

UW (2013-2022)
GaTech (2022-present)

Datapath Elements

- Covered Datapaths
  - Multiplexers (MUXs)
  - Zero (or one) detect
  - Counters
  - Priority Encoders
  - Comparators
  - Shifters
  - Adders (Covered later)
  - Crossbar Switches
  - Dividers
  - Multipliers (Covered later)
- Not an exhaustive list
  - Representative of common datapath structures and objectives

Binary numbers and two's complement

3 bit (0 - 1)

1 1 0

$1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$

$1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1$

$= 6$

325 decimal (0 - 9)

$= 3 \cdot 100 + 2 \cdot 10 + 5 \cdot 1$

$= 3 \cdot 10^2 + 2 \cdot 10^1 + 5 \cdot 10^0$

$$\begin{array}{r} {\scriptstyle +1}\ 1\,1\,0 \\ 0\,1\,1 \\ \hline 1\,0\,0\,1 \end{array}$$

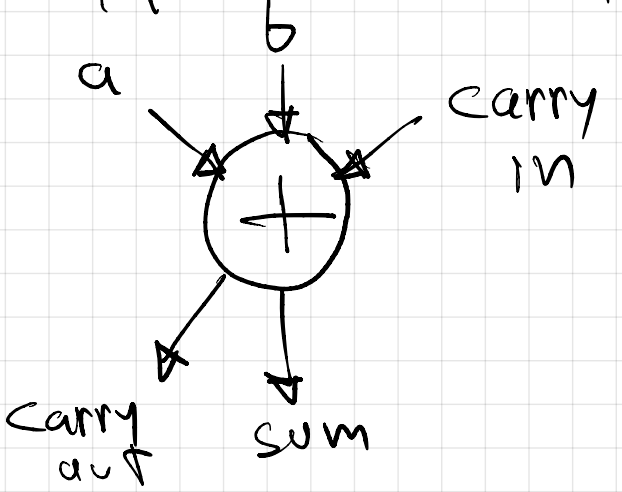
$$\begin{array}{r} 6 \\ 3 \\ \hline 9 \end{array}$$

$$\begin{array}{r} 0 \\ +0 \\ \hline 0 \end{array} \qquad \begin{array}{r} 0 \\ +1 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ \times 6 \\ \hline 1 \end{array}$$

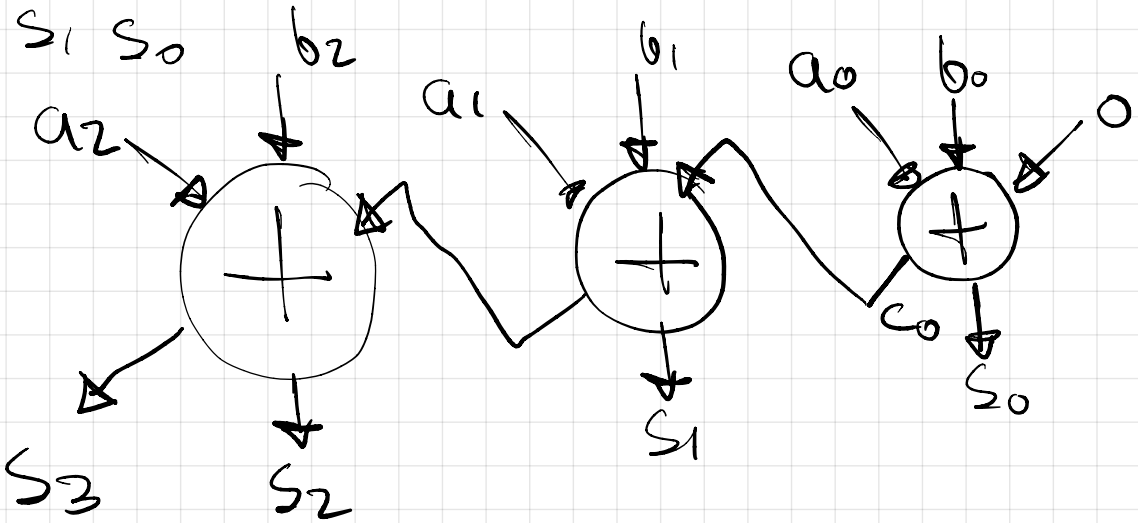
$$\begin{array}{r} 1 \\ \times 1 \\ \hline 10 \end{array}$$

$$\begin{array}{r} 1\,1\,1 \\ 1\,1\,1 \\ \hline 1\,1\,1\,0 \end{array} \qquad \begin{array}{r} 7 \\ \times 7 \\ \hline 49 \end{array}$$

$$\begin{array}{r} s_0 \\ s_1 \\ \hline 1\,a_1 \end{array}$$



$$\begin{array}{r} a_2\ a_1\ a_0 \\ b_2\ b_1\ b_0 \\ \hline s_3\ s_2\ s_1\ s_0 \end{array}$$



How do you do subtractions?

$$\begin{array}{r} a_2\ a_1\ a_0 \\ b_2\ b_1\ b_0 \\ \hline s_3\ s_2\ s_1\ s_0 \end{array}$$

s_3: discard

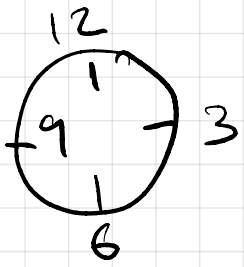
~~0 0 0~~ 8 → 0

~~0 0 1~~ 9 → 1

~~0 1 0~~ 10 → 2

n bits

modulo 2^n arithmetic



$$\begin{array}{r} 1\ 1\ 1 \\ 1\ 1\ 1 \\ \hline \cancel{1}\ 1\ 1\ 0 \end{array} \qquad \begin{array}{r} 7 \\ 7 \\ \hline \cancel{14} \end{array} \quad 6$$

$$\begin{array}{r} 1\ 0\ 1 \\ 0\ 1\ 0 \\ \hline 1\ 1\ 1 \end{array}$$

$$2^{N-1} + 2^{N-2} + \cdots + 2^2 + 2^1 + 2^0 = 2^N - 1$$

N bits:

$$\begin{array}{r} a_{N-1}\ a_{N-2}\ \cdots\ a_2\ a_1\ a_0 \\ +\ \overline{a_{N-1}}\ \overline{a_{N-2}}\ \cdots\ \overline{a_2}\ \overline{a_1}\ \overline{a_0} \\ \hline 1\ 1\ \cdots\ 1\ 1\ 1 \end{array} = 2^N - 1$$

$$\begin{array}{r} +\ 0\ 0\ \cdots\ 0\ 0\ 1 \\ \hline 1\ 0\ 0\ \cdots\ 0\ 0\ 0 \end{array}$$

N bits

$= 0$

$$a = a_{N-1}\, a_{N-2} \cdots a_2\, a_1\, a_0$$

$$\bar{a} = \bar{a}_{N-1}\, \bar{a}_{N-2} \cdots \bar{a}_2\, \bar{a}_1\, \bar{a}_0$$

$$\Rightarrow \quad a + \bar{a} + 1 = 0$$

N-bits arithmetic (modulo 2^N arithmetic)

$$b - a = b + (-a)$$

How do you define "$-a$"

$$a + (-a) = 0 \Rightarrow -a = \bar{a} + 1$$

$$b - a = b + (-a) = b + \bar{a} + 1$$

Read about 2's complement

3 a_2 a_1 a_0

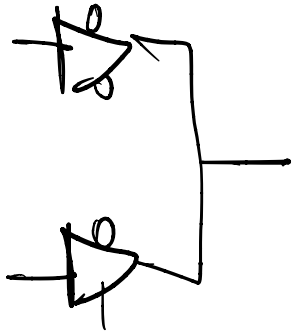
Unsigned: $a_2 \cdot 2^2 + a_1 2^1 + a_0 2^0$

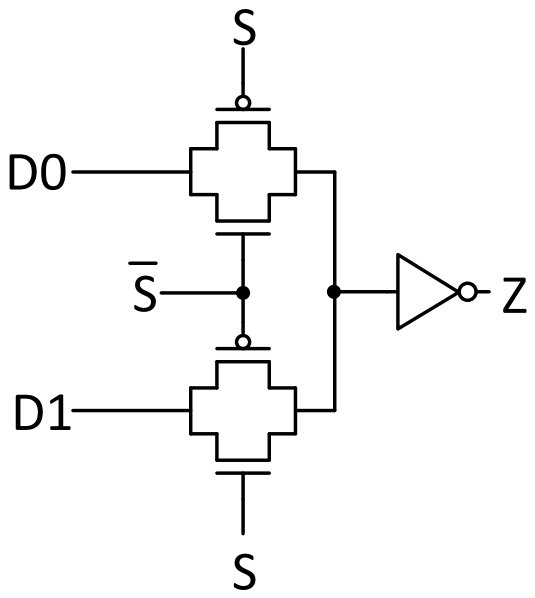
signed (2's comp.) $-a_2 2^2 + a_1 2^1 + a_0 2^0$

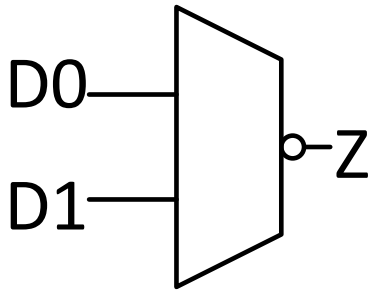
Binary representation	Unsigned	Signed
0 0 0	0	0
0 0 1	1	1
0 1 0	2	2
0 1 1	3	3
1 0 0	4	-4
1 0 1	5	-3
1 1 0	6	-2
1 1 1	7	-1

Quick overview of adders (more to come later)

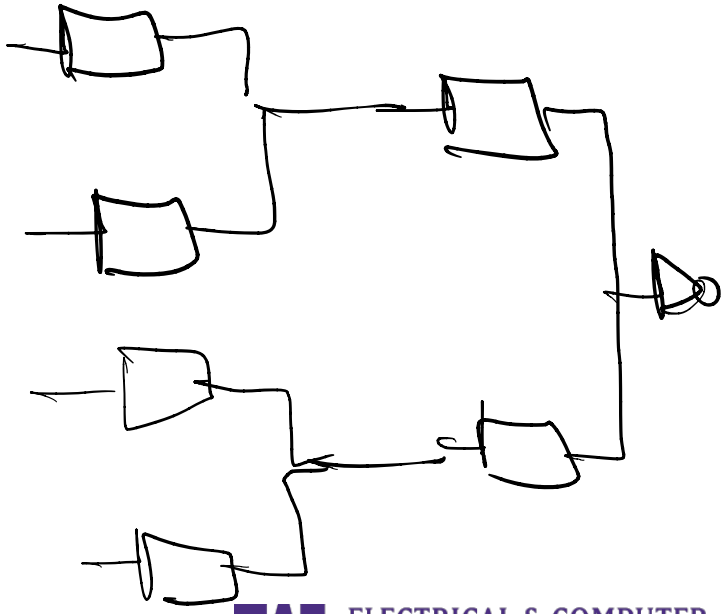
Multiplexers



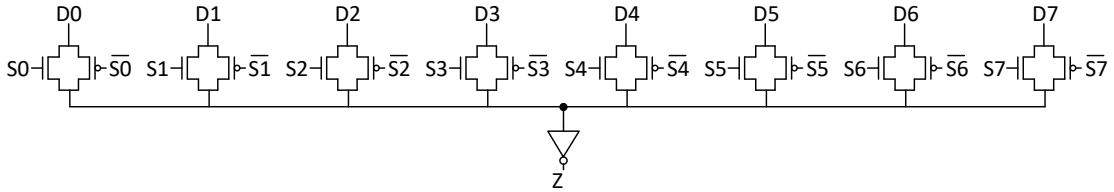


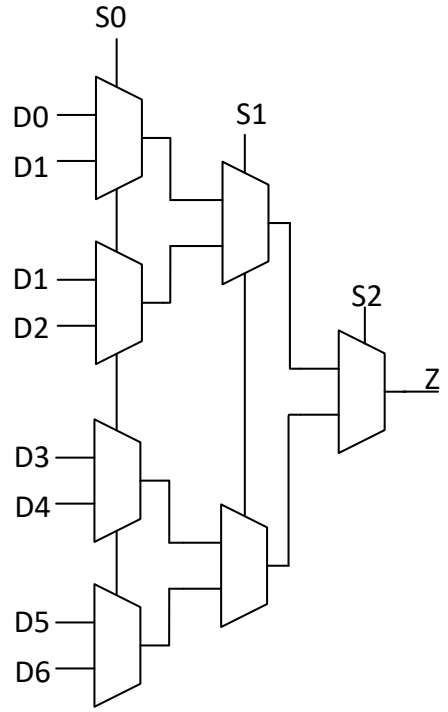


- Transmission gate mux commonly used
- Wide mux-like structures common in several datapaths
  - Shift/Rotate functionality
  - Issuing instructions from a microprocessor instruction queue
  - Permutation blocks (Cryptography)



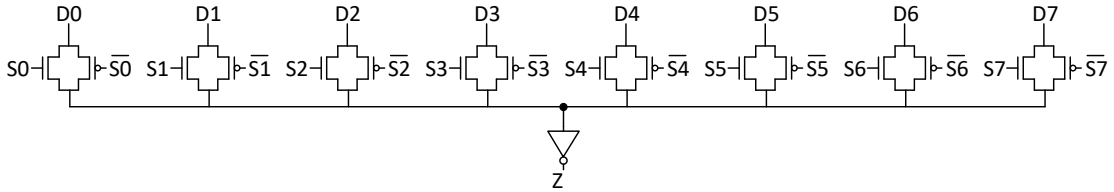
Wide Mux Structures



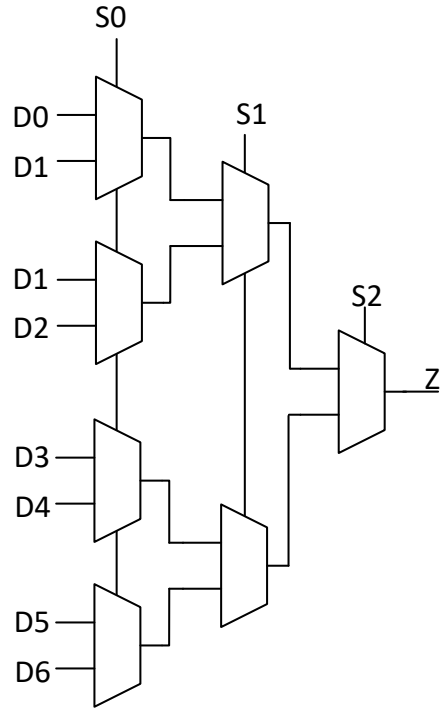


- E.g. 8-stage Mux
  - Manage trade-off between number of stages and loading (per stage)
    - Wide, mux structure : Low number of stages, high loading
    - Mux tree structure: Larger number of stages, smaller load per stage
- Sweet-spot depends on
  - Process technology (Wire cap, self-loading factor)
  - Availability of slack (e.g. relative arrival time) among select bits vs. data bits
  - Loading

Wide Mux Structures

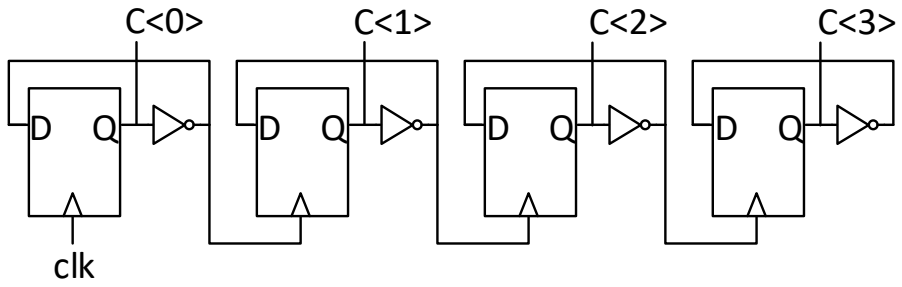


If S and D all arrive together, what is likely a fast mux-delay approach



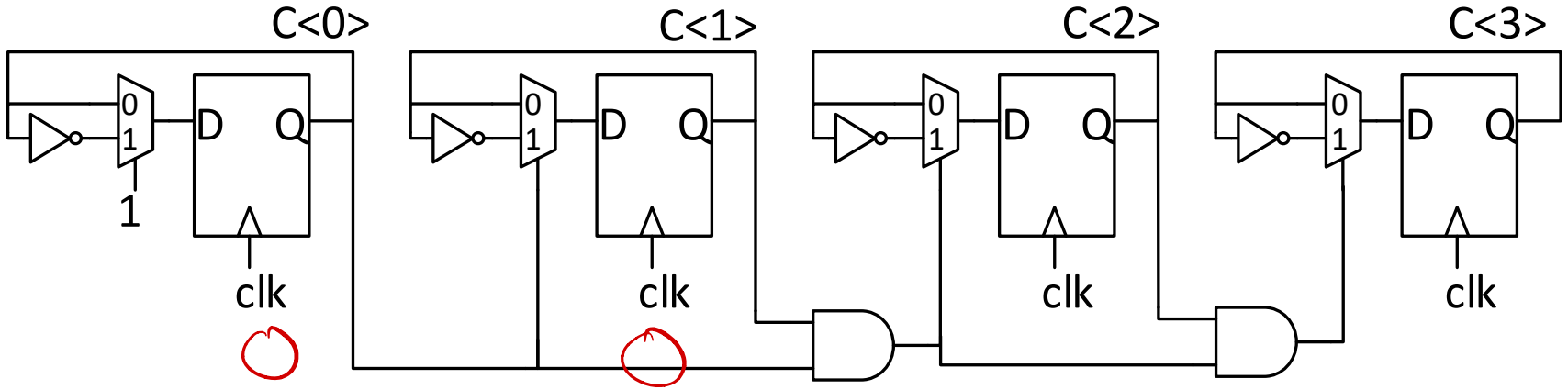
- E.g. 8-stage Mux
  - Manage trade-off between number of stages and loading (per stage)
    - Wide, mux structure : Low number of stages, high loading
    - Mux tree structure: Larger number of stages, smaller load per stage
- Sweet-spot depends on
  - Process technology (Wire cap, self-loading factor)
  - Availability of slack (e.g. relative arrival time) among select bits vs. data bits
  - Loading

Counters



- Increment or Decrement stored value in a register every clock cycle
  - Program Counters
  - State Machines
  - Timers
  - PLLs
- Common features
  - Programmable (Load initial count value)
  - Enable
  - Resettable
  - Terminal Count

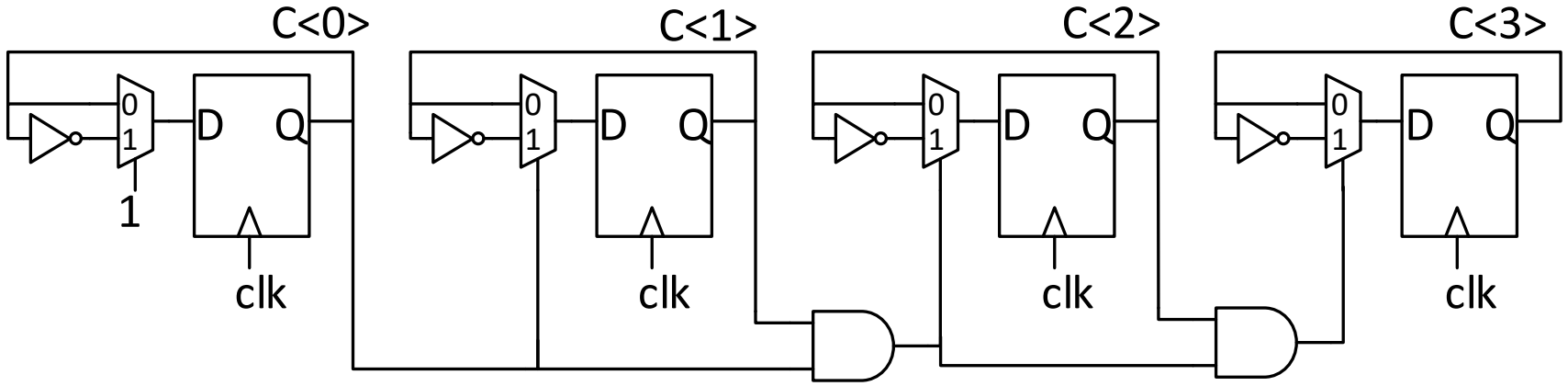
Basic Synchronous Counter



- Relies on observation that bits transition every time all LSBs are 1
  - Use AND operation to detect if all LSBs are 1
  - If so, select the "inverted-Q" path
- Outputs synchronous with clock
- Long count values are challenging

Index	Code
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

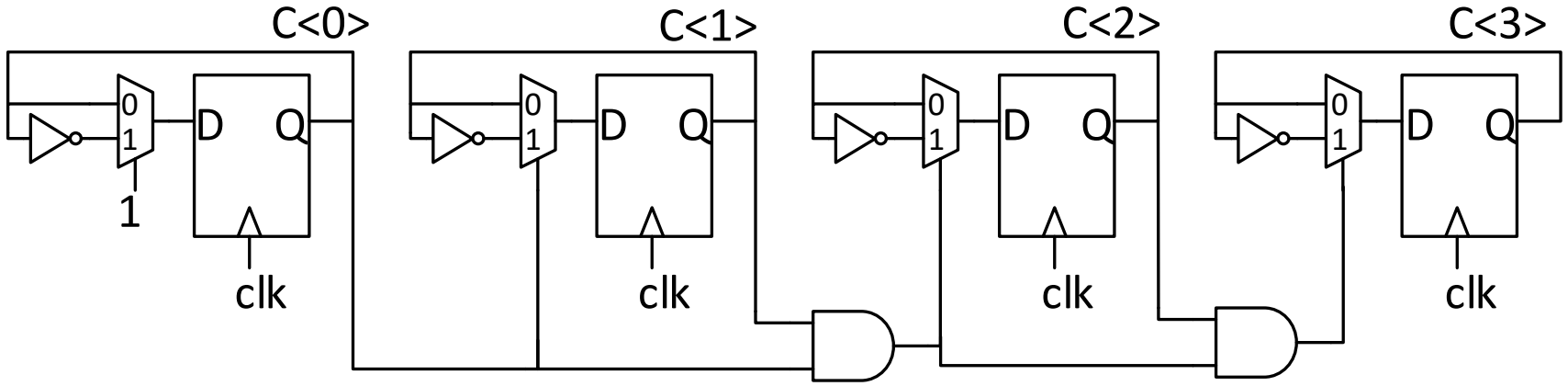
Basic Synchronous Counter



- Relies on observation that bits transition every time all LSBs are 1
  - Use AND operation to detect if all LSBs are 1
  - If so, select the "inverted-Q" path
- Outputs synchronous with clock
- Long count values are challenging

Index	Code
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

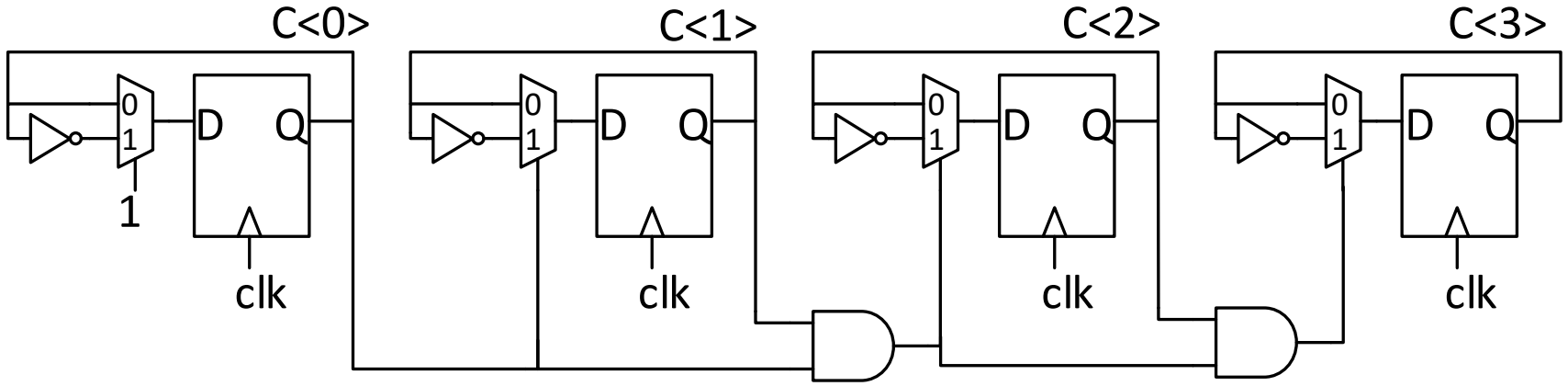
Basic Synchronous Counter



- Relies on observation that bits transition every time all LSBs are 1
  - Use AND operation to detect if all LSBs are 1
  - If so, select the "inverted-Q" path
- Outputs synchronous with clock
- Long count values are challenging

Index	Code
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

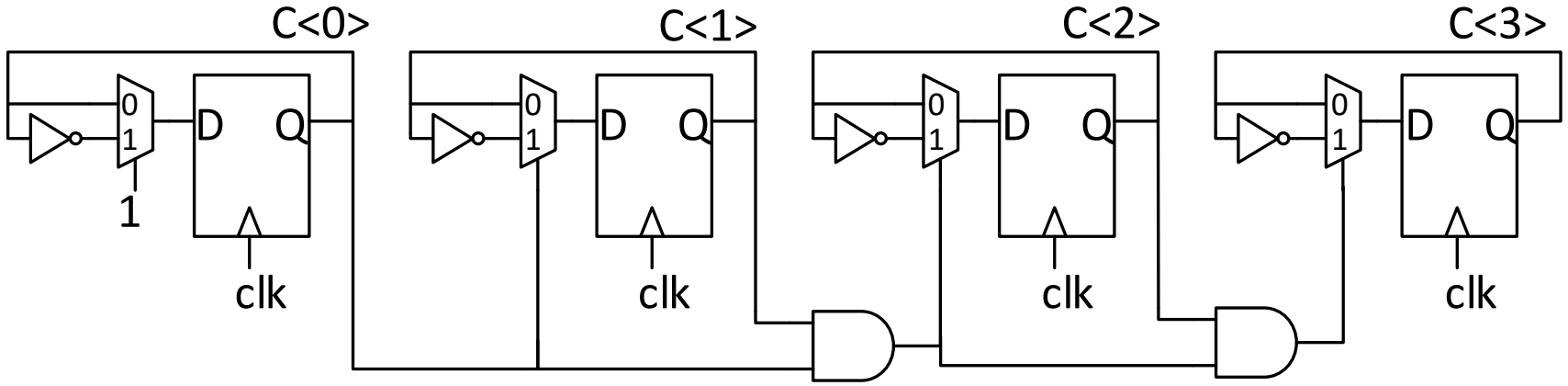
Basic Synchronous Counter



- Relies on observation that bits transition every time all LSBs are 1
  - Use AND operation to detect if all LSBs are 1
  - If so, select the "inverted-Q" path
- Outputs synchronous with clock
- Long count values are challenging

Index	Code
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

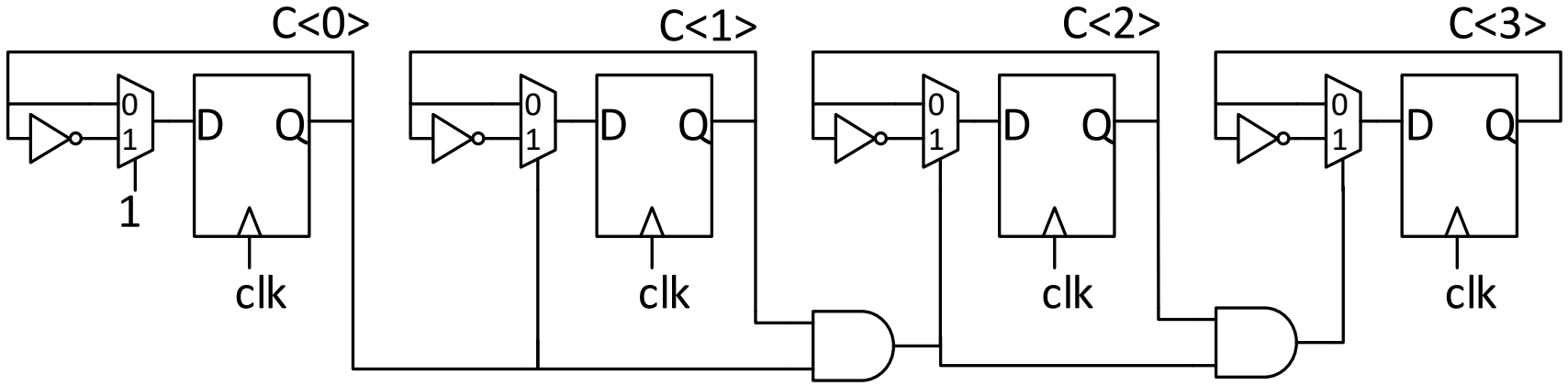
Basic Synchronous Counter



- Relies on observation that bits transition every time all LSBs are 1
  - Use AND operation to detect if all LSBs are 1
  - If so, select the "inverted-Q" path
- Outputs synchronous with clock
- Long count values are challenging

Index	Code
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

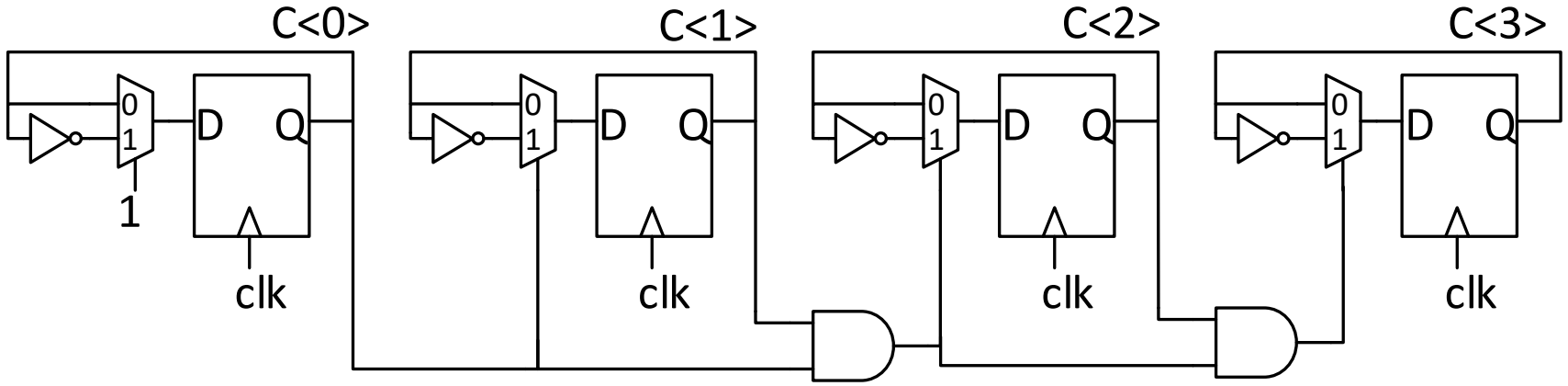
Basic Synchronous Counter



- Relies on observation that bits transition every time all LSBs are 1
  - Use AND operation to detect if all LSBs are 1
  - If so, select the "inverted-Q" path
- Outputs synchronous with clock
- Long count values are challenging

Index	Code
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

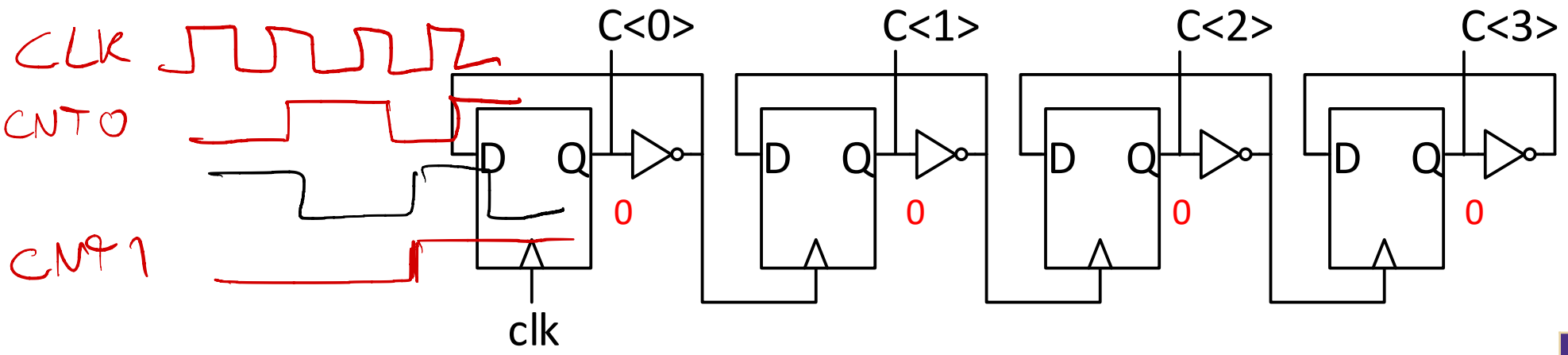
Basic Synchronous Counter



- Relies on observation that bits transition every time all LSBs are 1
  - Use AND operation to detect if all LSBs are 1
  - If so, select the "inverted-Q" path
- Outputs synchronous with clock
- Long count values are challenging

Index	Code
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

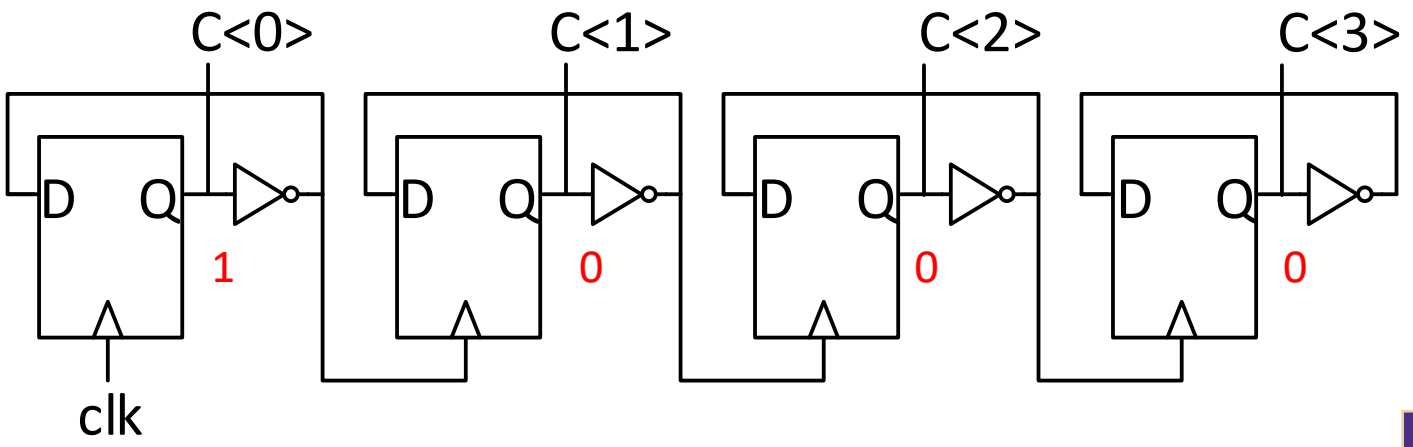
Counters (Asynchronous)



- Simplest counter to build
- Governing rule: Toggle a bit if less significant bit transitions to 0
- Capable of performing very high frequency counts
- Limited usage case due to asynchronous behaviour

Index	Code
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

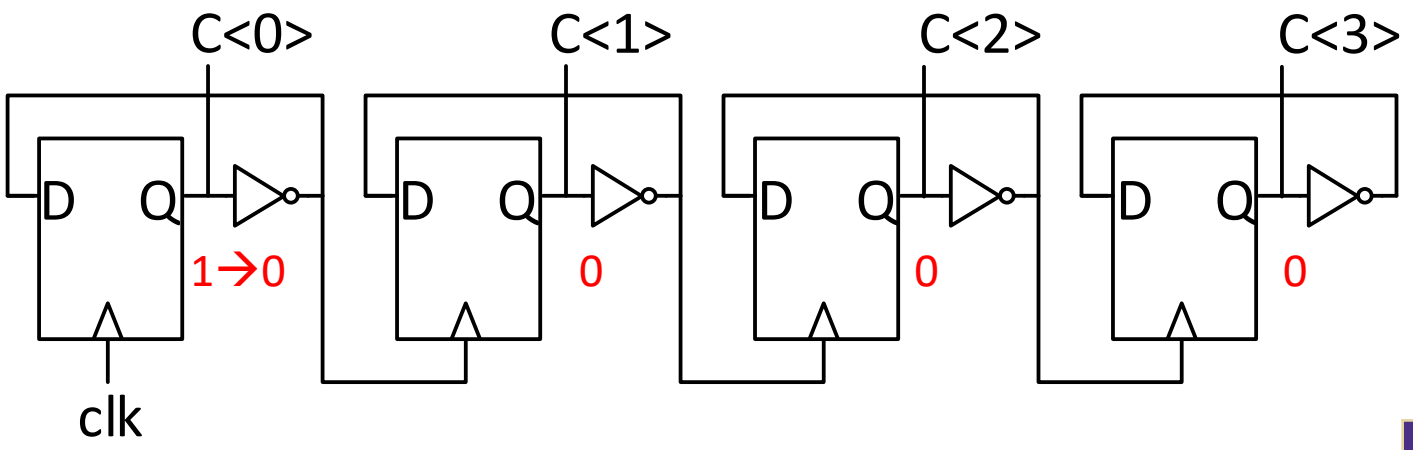
Counters (Asynchronous)



- Simplest counter to build
- Governing rule: Toggle a bit if less significant bit transitions to 0
- Capable of performing very high frequency counts
- Limited usage case due to asynchronous behaviour

Index	Code
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

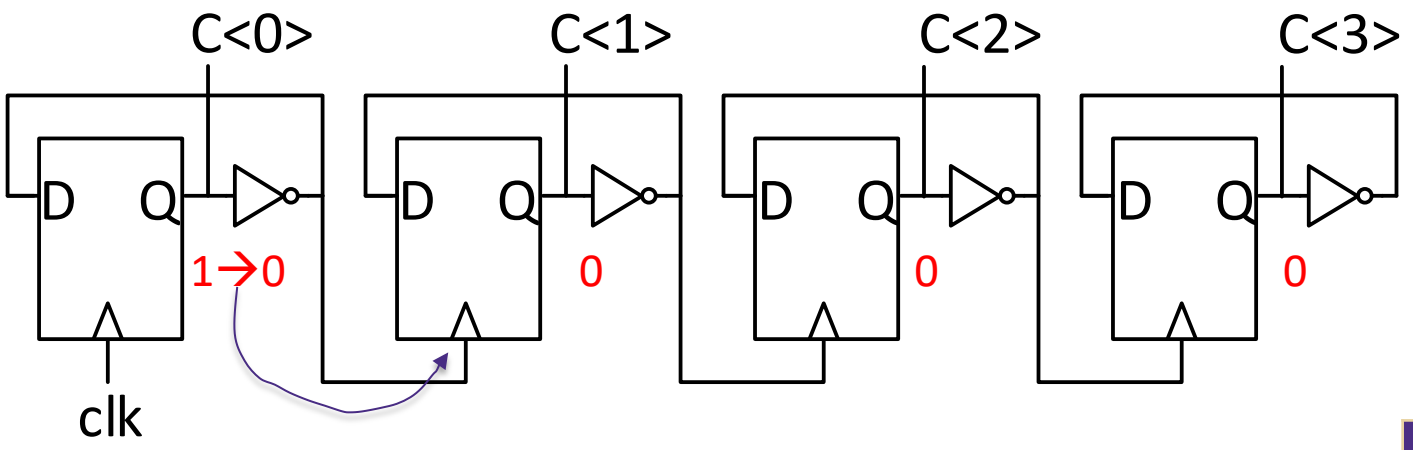
Counters (Asynchronous)



- Simplest counter to build
- Governing rule: Toggle a bit if less significant bit transitions to 0
- Capable of performing very high frequency counts
- Limited usage case due to asynchronous behaviour

Index	Code
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

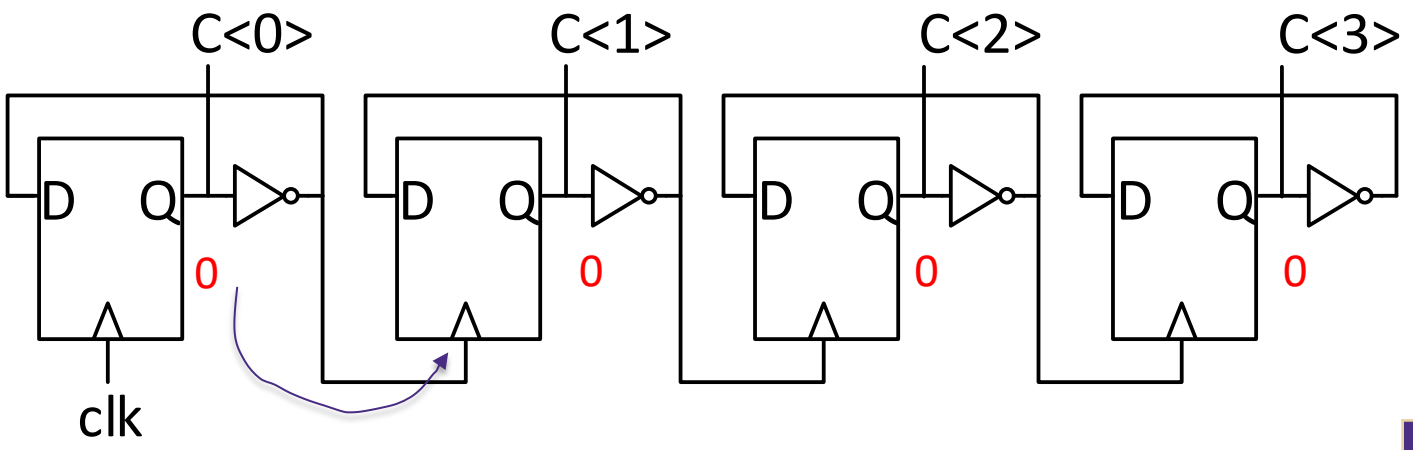
Counters (Asynchronous)



- Simplest counter to build
- Governing rule: Toggle a bit if less significant bit transitions to 0
- Capable of performing very high frequency counts
- Limited usage case due to asynchronous behaviour

Index	Code
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

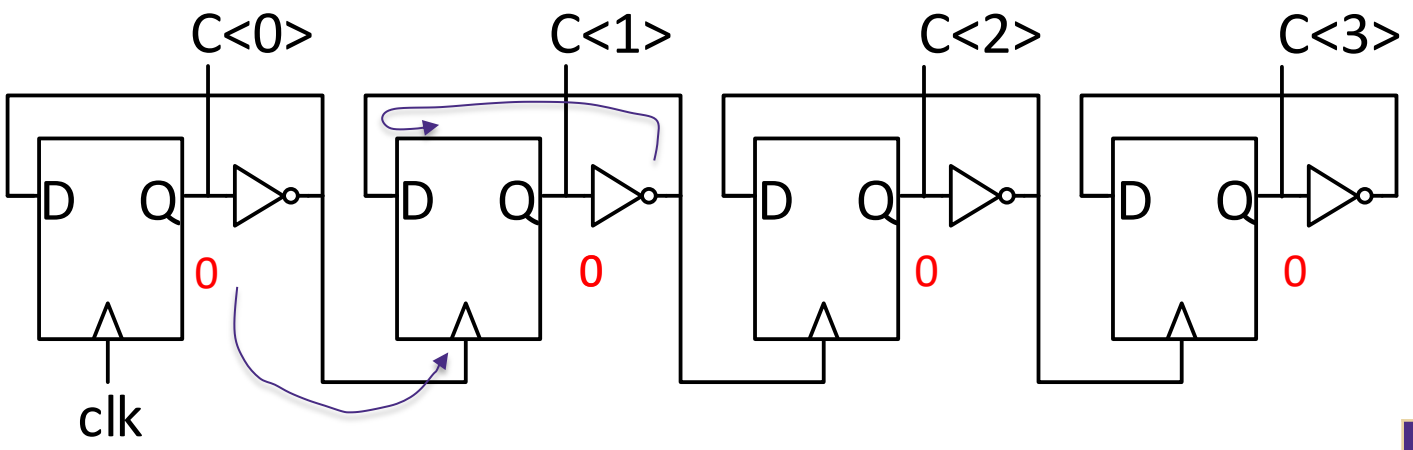
Counters (Asynchronous)



- Simplest counter to build
- Governing rule: Toggle a bit if less significant bit transitions to 0
- Capable of performing very high frequency counts
- Limited usage case due to asynchronous behaviour

Index	Code
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

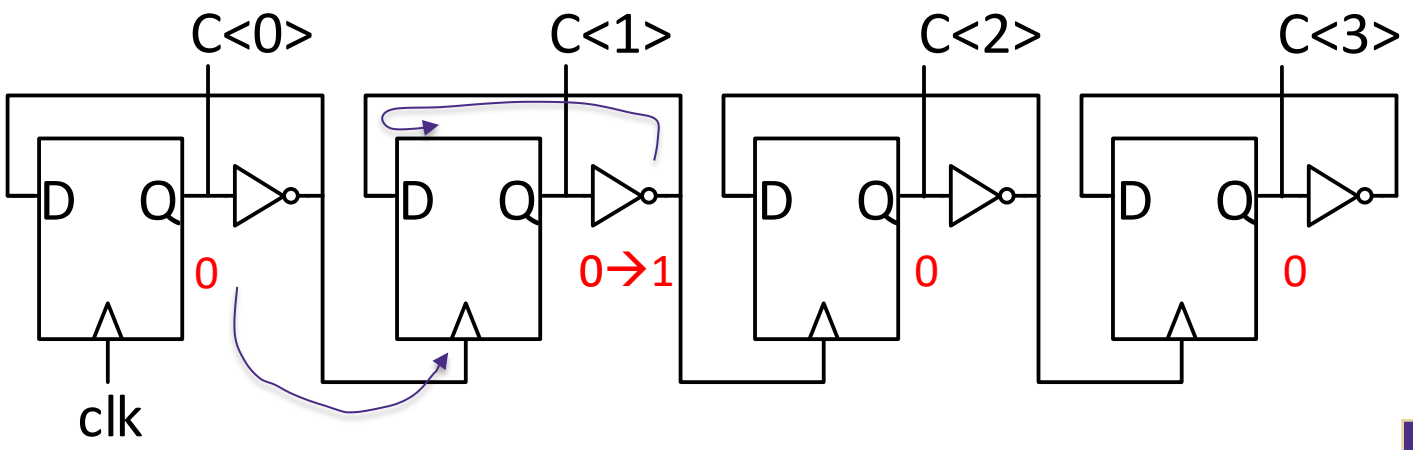
Counters (Asynchronous)



- Simplest counter to build
- Governing rule: Toggle a bit if less significant bit transitions to 0
- Capable of performing very high frequency counts
- Limited usage case due to asynchronous behaviour

Index	Code
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

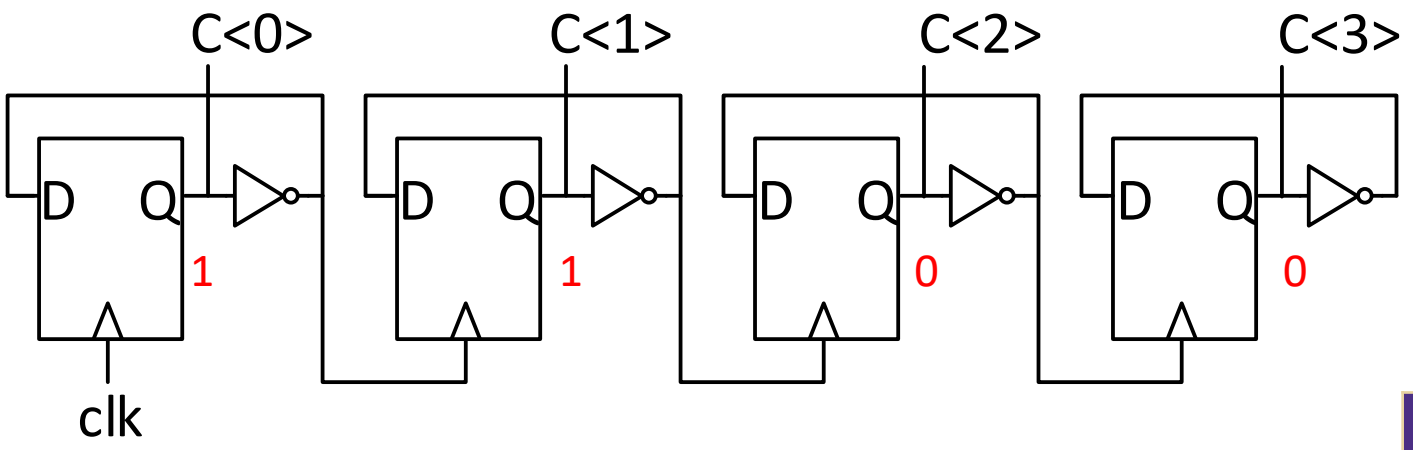
Counters (Asynchronous)



- Simplest counter to build
- Governing rule: Toggle a bit if less significant bit transitions to 0
- Capable of performing very high frequency counts
- Limited usage case due to asynchronous behaviour

Index	Code
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

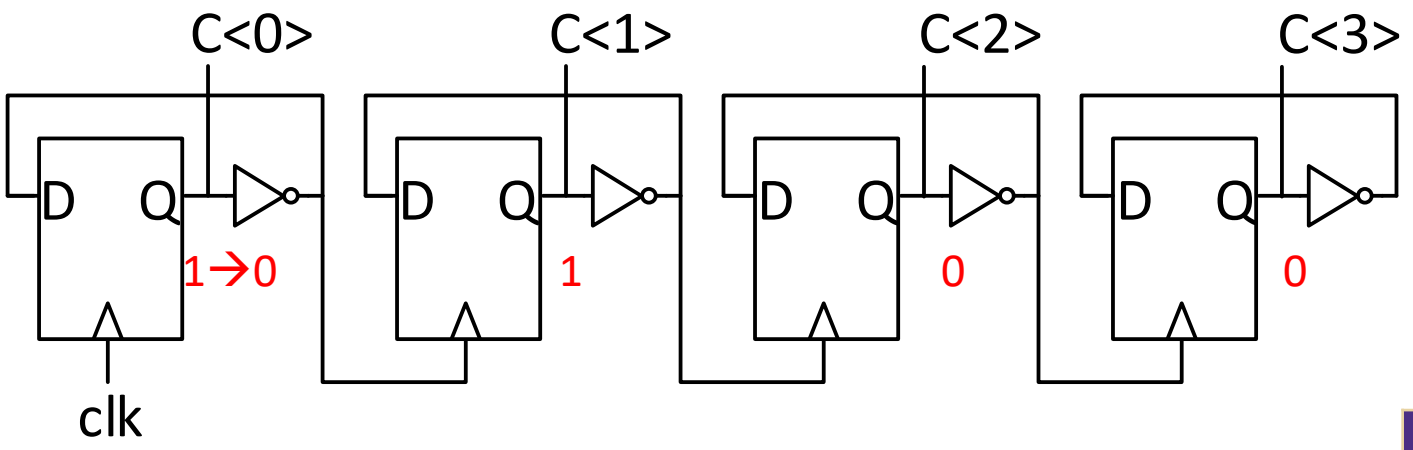
Counters (Asynchronous)



- Simplest counter to build
- Governing rule: Toggle a bit if less significant bit transitions to 0
- Capable of performing very high frequency counts
- Limited usage case due to asynchronous behaviour

Index	Code
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

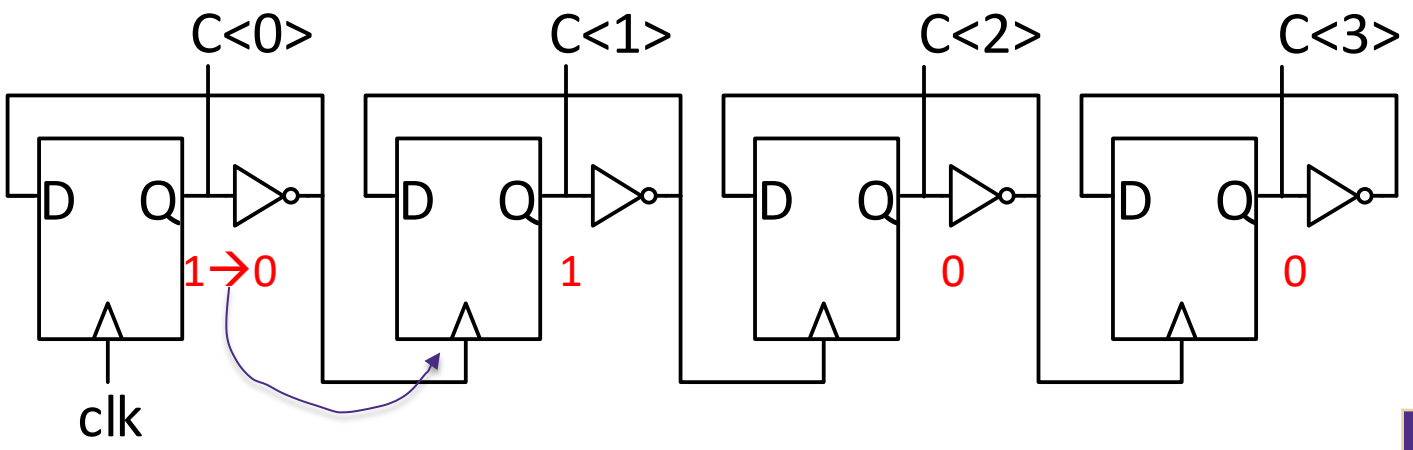
Counters (Asynchronous)



- Simplest counter to build
- Governing rule: Toggle a bit if less significant bit transitions to 0
- Capable of performing very high frequency counts
- Limited usage case due to asynchronous behavior

Index	Code
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

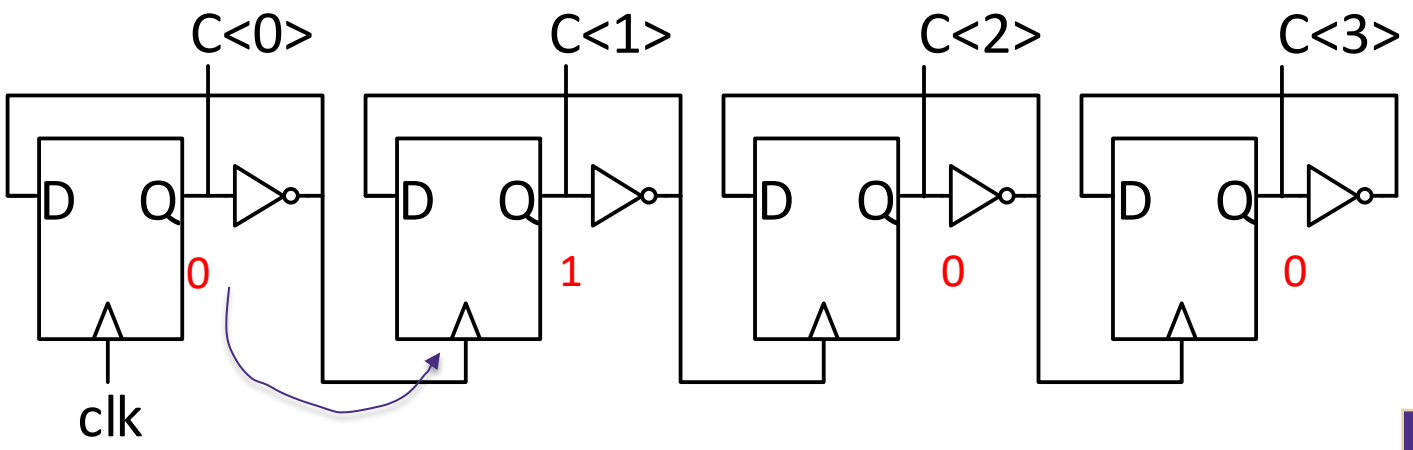
Counters (Asynchronous)



- Simplest counter to build
- Governing rule: Toggle a bit if less significant bit transitions to 0
- Capable of performing very high frequency counts
- Limited usage case due to asynchronous behavior

Index	Code
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

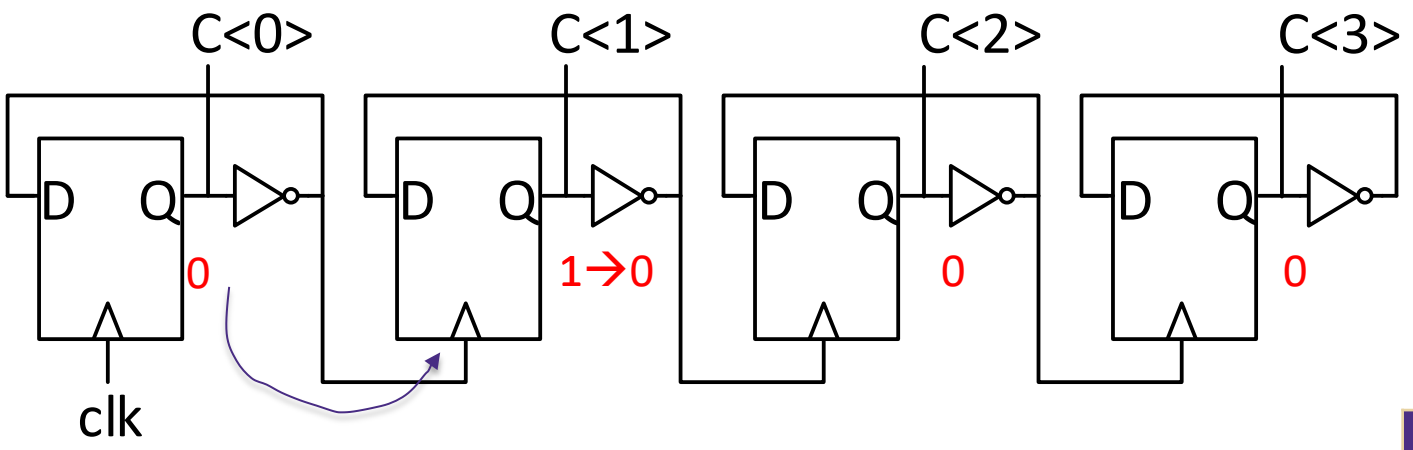
Counters (Asynchronous)



- Simplest counter to build
- Governing rule: Toggle a bit if less significant bit transitions to 0
- Capable of performing very high frequency counts
- Limited usage case due to asynchronous behavior

Index	Code
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

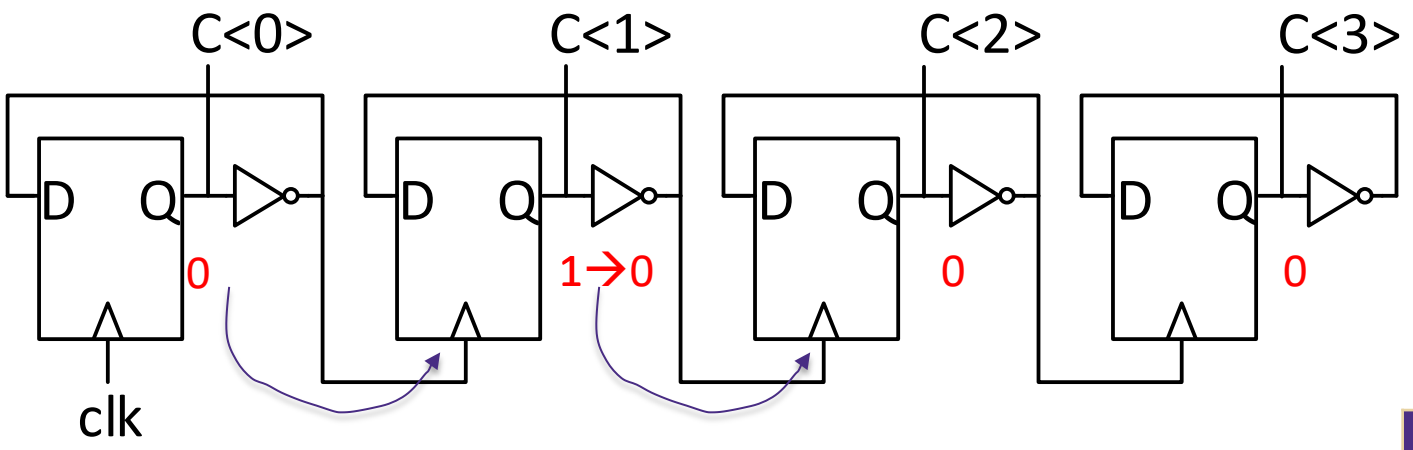
Counters (Asynchronous)



- Simplest counter to build
- Governing rule: Toggle a bit if less significant bit transitions to 0
- Capable of performing very high frequency counts
- Limited usage case due to asynchronous behavior

Index	Code
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

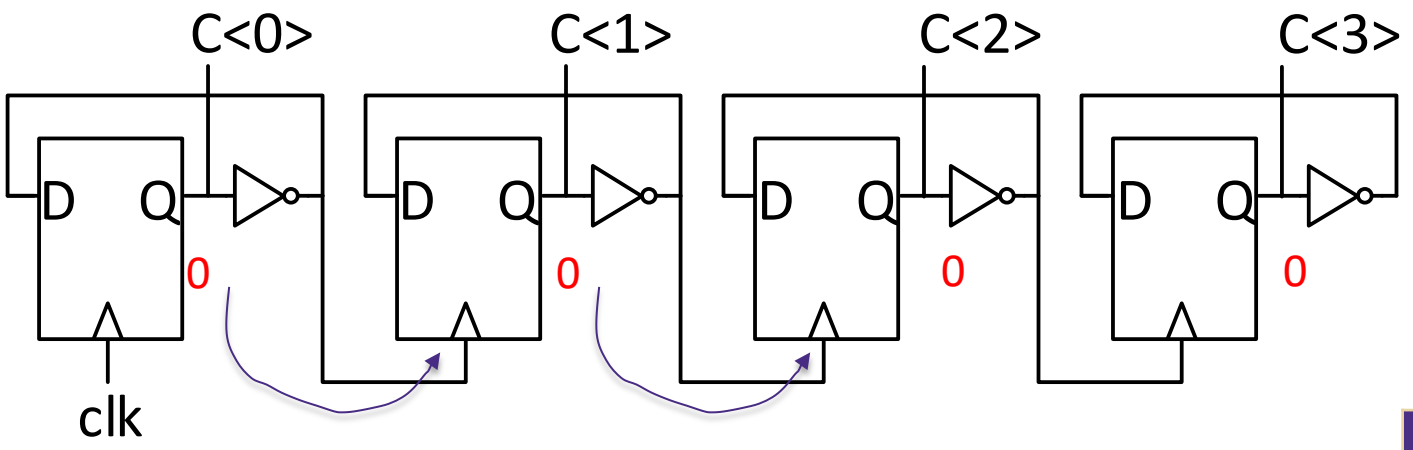
Counters (Asynchronous)



- Simplest counter to build
- Governing rule: Toggle a bit if less significant bit transitions to 0
- Capable of performing very high frequency counts
- Limited usage case due to asynchronous behavior

Index	Code
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

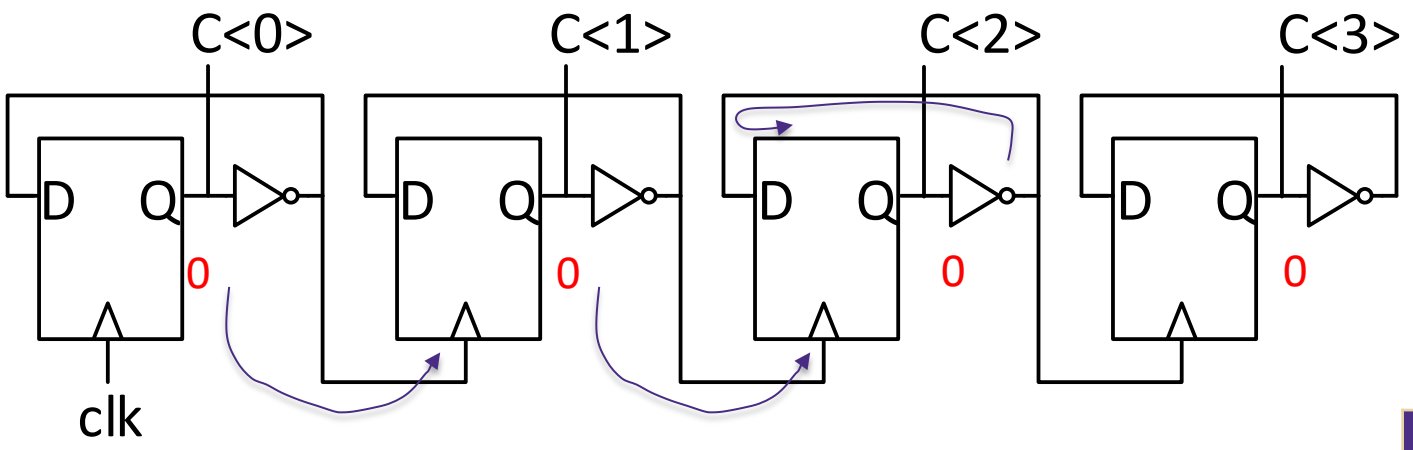
Counters (Asynchronous)



- Simplest counter to build
- Governing rule: Toggle a bit if less significant bit transitions to 0
- Capable of performing very high frequency counts
- Limited usage case due to asynchronous behavior

Index	Code
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

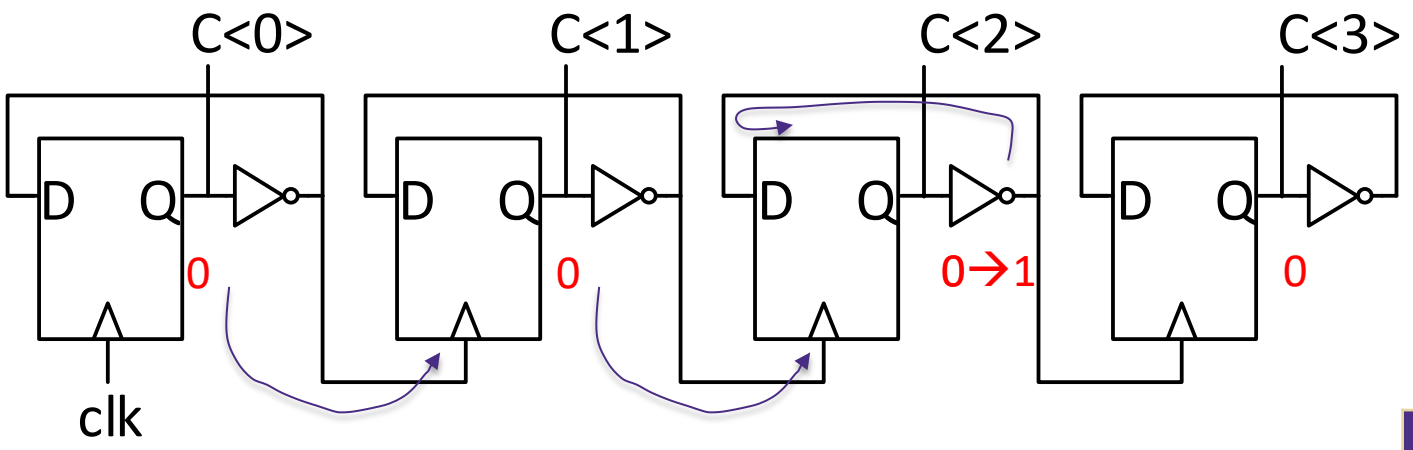
Counters (Asynchronous)



- Simplest counter to build
- Governing rule: Toggle a bit if less significant bit transitions to 0
- Capable of performing very high frequency counts
- Limited usage case due to asynchronous behavior

Index	Code
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

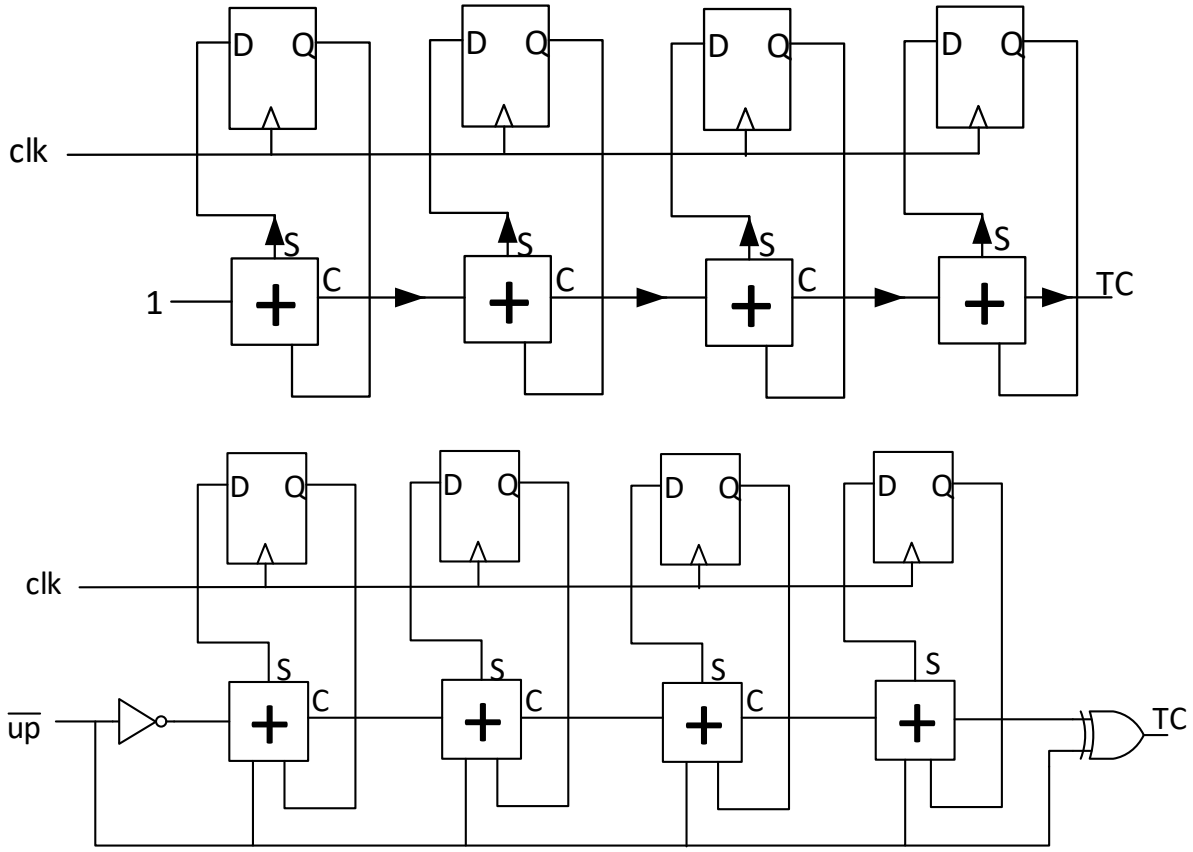
Counters (Asynchronous)



- Simplest counter to build
- Governing rule: Toggle a bit if less significant bit transitions to 0
- Capable of performing very high frequency counts
- Limited usage case due to asynchronous behavior

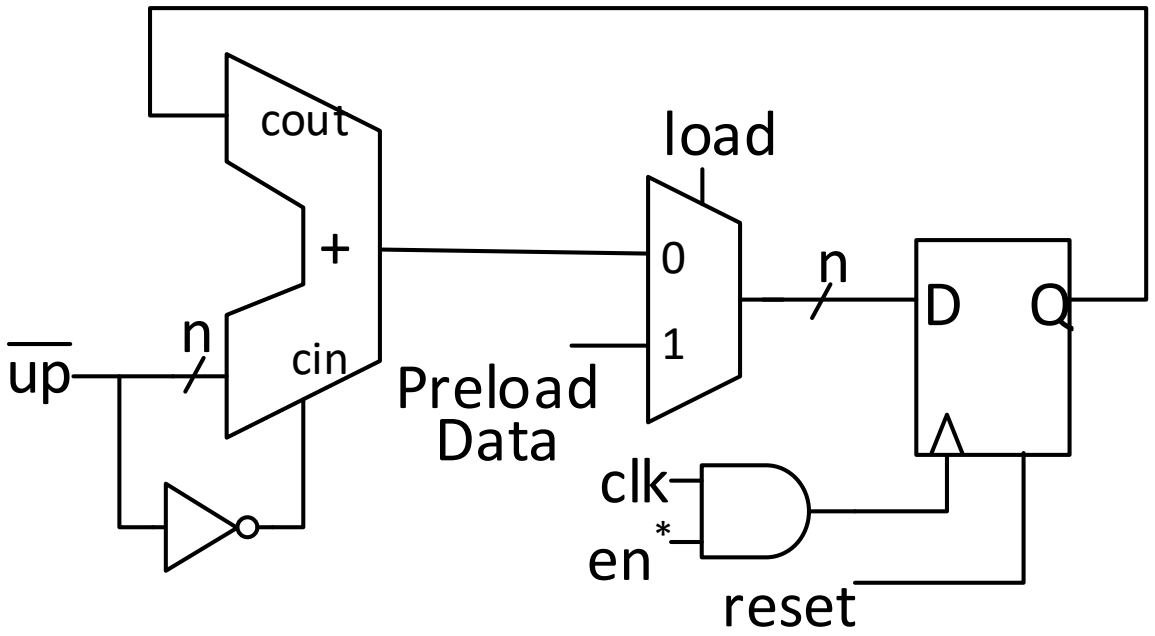
Index	Code
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Counters (Up/Down)



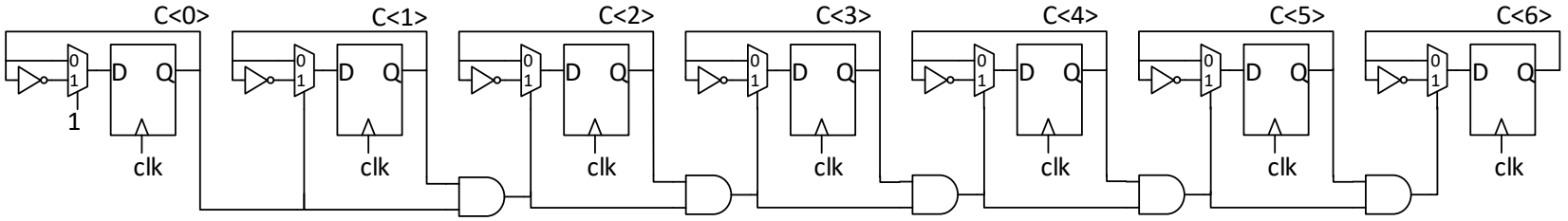
- Some applications (DLLs) call for an up/down counter
  - Control signal UP provides count direction
  - UP=1 ➔increment count
  - UP=0 ➔ decrement count

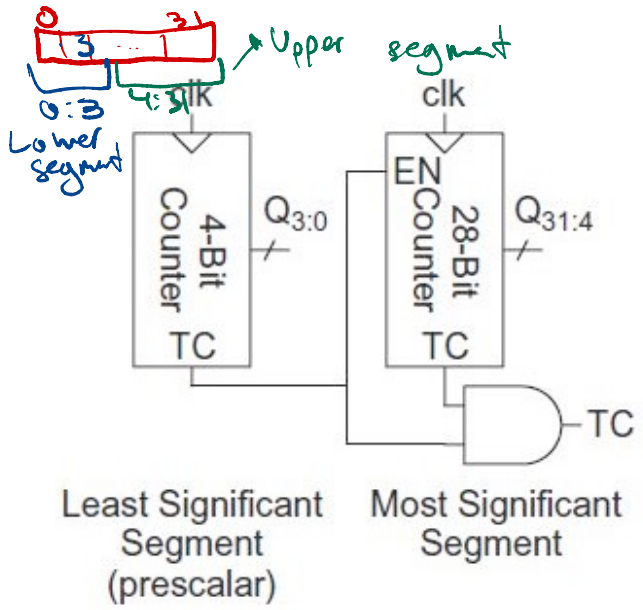
Putting it Together..



- Up/Down count, reset, pre-load, enable* functionality
- Pre-load counter value through Mux onto n-bit register
- Adder performs either +1 or -1 (in 2's complement)

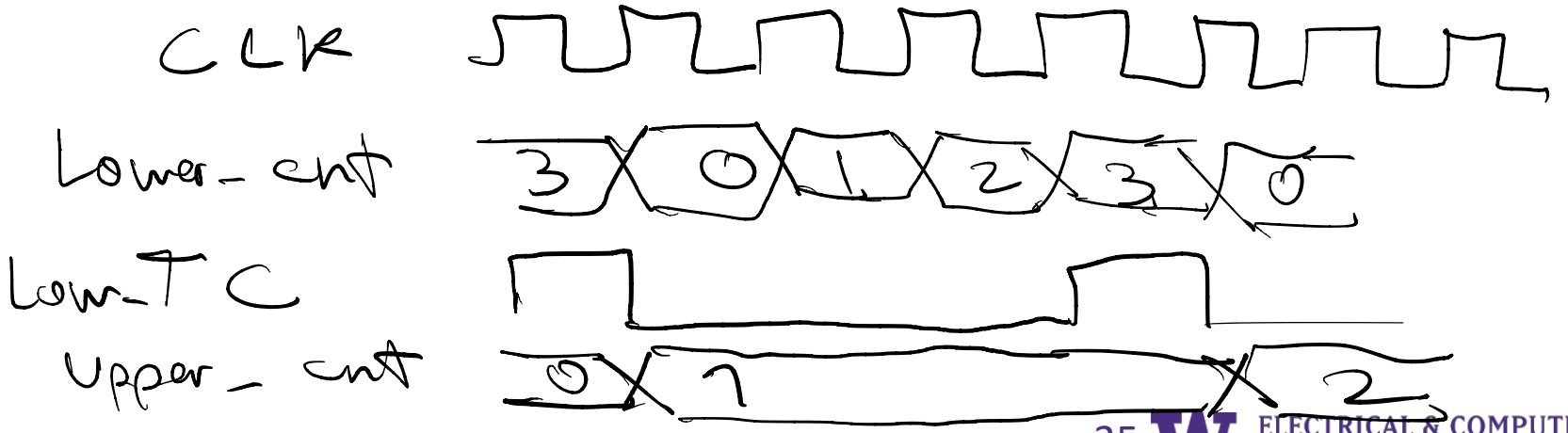
Counters (Fast Counters)



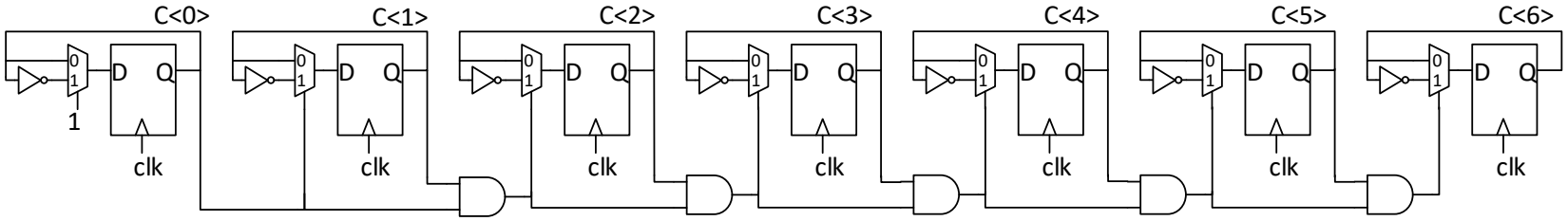


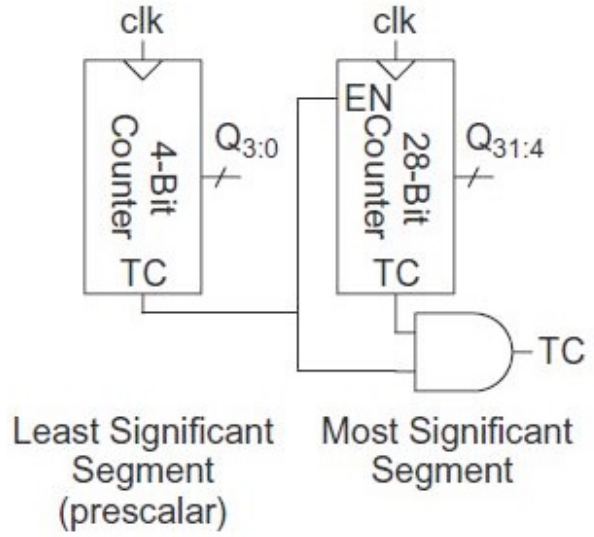
Source:W&H

- Long synchronous counters are challenging due to long **prefix** chain of 1s (Long and-gate chain)



Counters (Fast Counters)

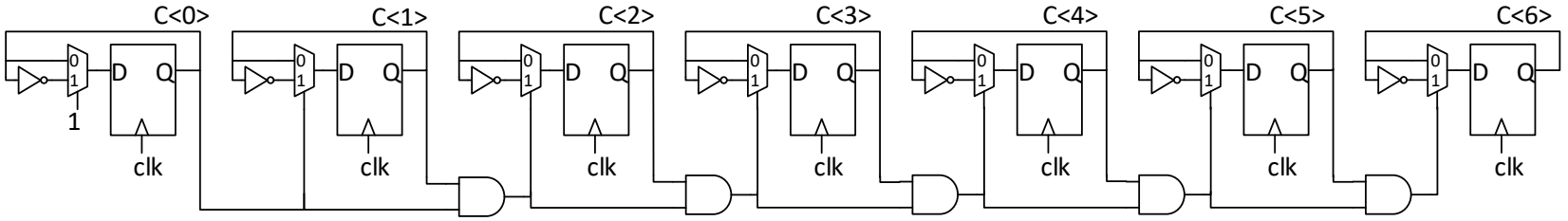


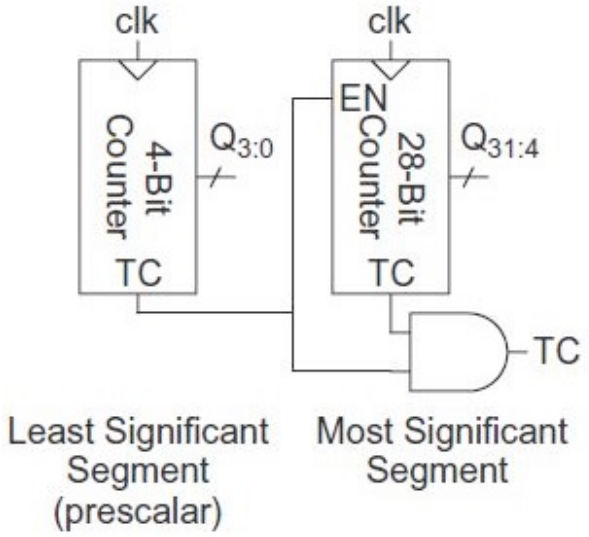


Source:W&H

- Long synchronous counters are challenging due to long **prefix** chain of 1s (Long and-gate chain)
- Build a 2-stage counters
- Keep the first stage small enough to meet cycle-time
- Use Terminal Count signal to enable higher order counter

Counters (Fast Counters)

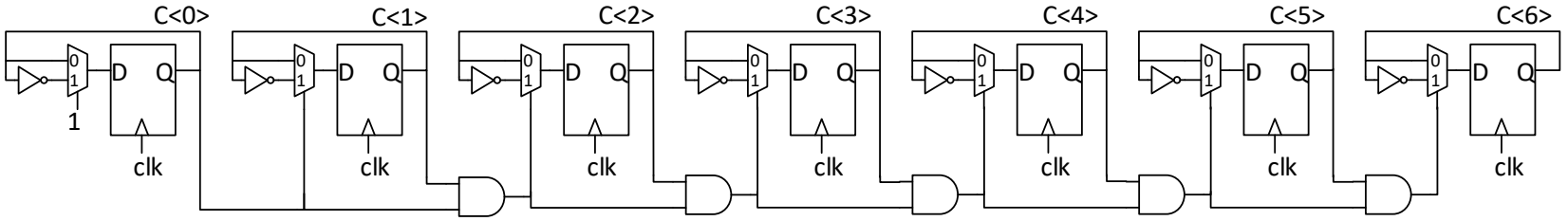


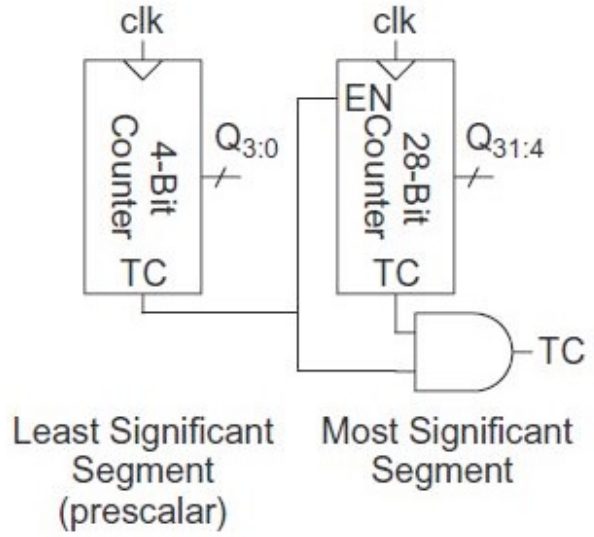


Source:W&H

- Long synchronous counters are challenging due to long **prefix** chain of 1s (Long and-gate chain)
- Build a 2-stage counters
- Keep the first stage small enough to meet cycle-time
- Use Terminal Count signal to enable higher order counter

Counters (Fast Counters)



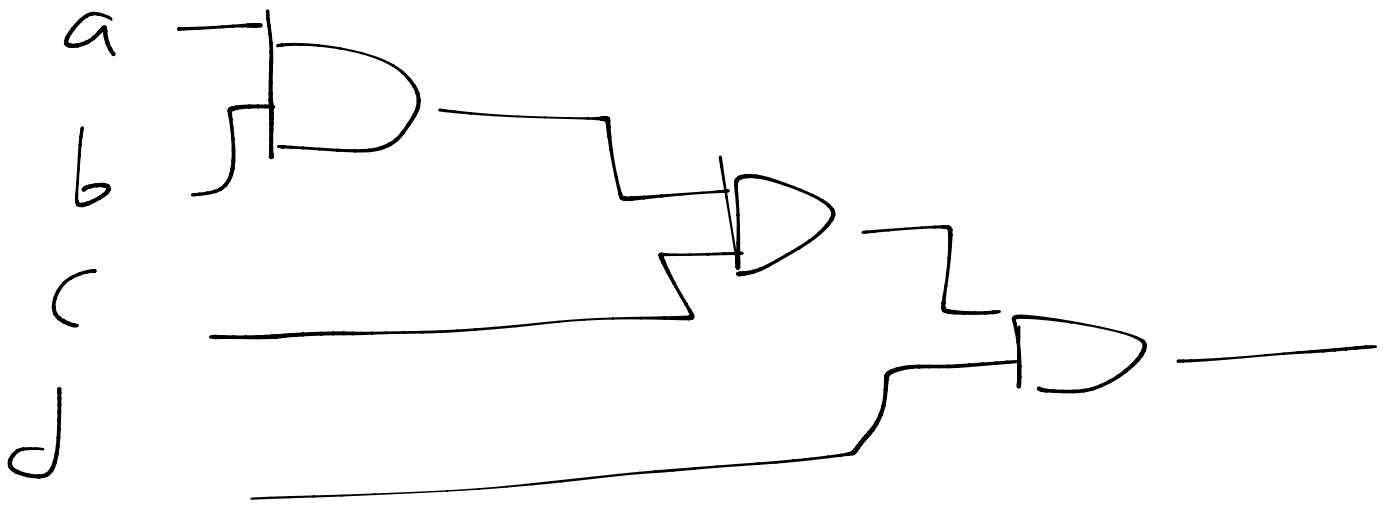


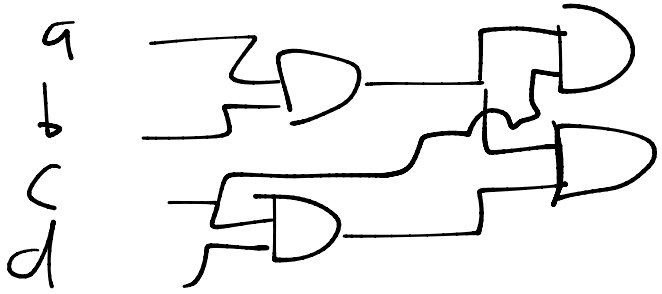
Source:W&H

- Long synchronous counters are challenging due to long **prefix** chain of 1s (Long and-gate chain)
- Build a 2-stage counters
- Keep the first stage small enough to meet cycle-time
- Use Terminal Count signal to enable higher order counter
  - Why does this circumvent the delay problem?

Segway Exercise/Discussion

- Counter speed is limited by the critical path of either the carry chain of my counter (If built using adder), or by a sequence of AND gates



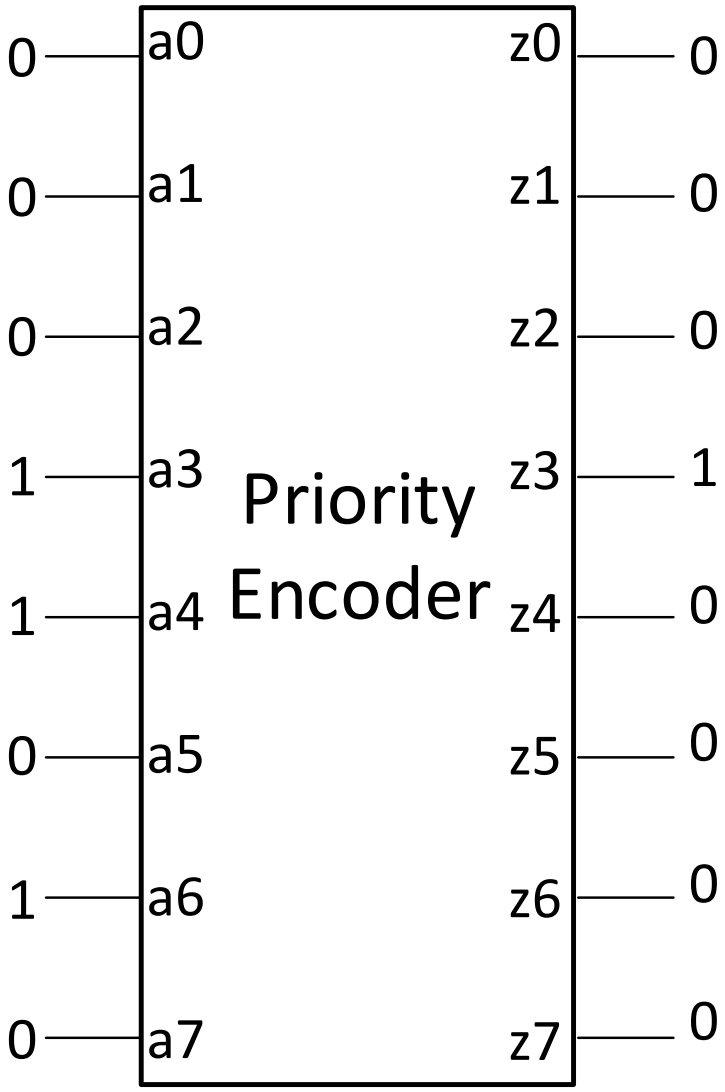


Segway Exercise/Discussion

- Counter speed is limited by the critical path of either the carry chain of my counter (If built using adder), or by a sequence of AND gates
  - How can I speed this up further?
  - How can I exploit associativity

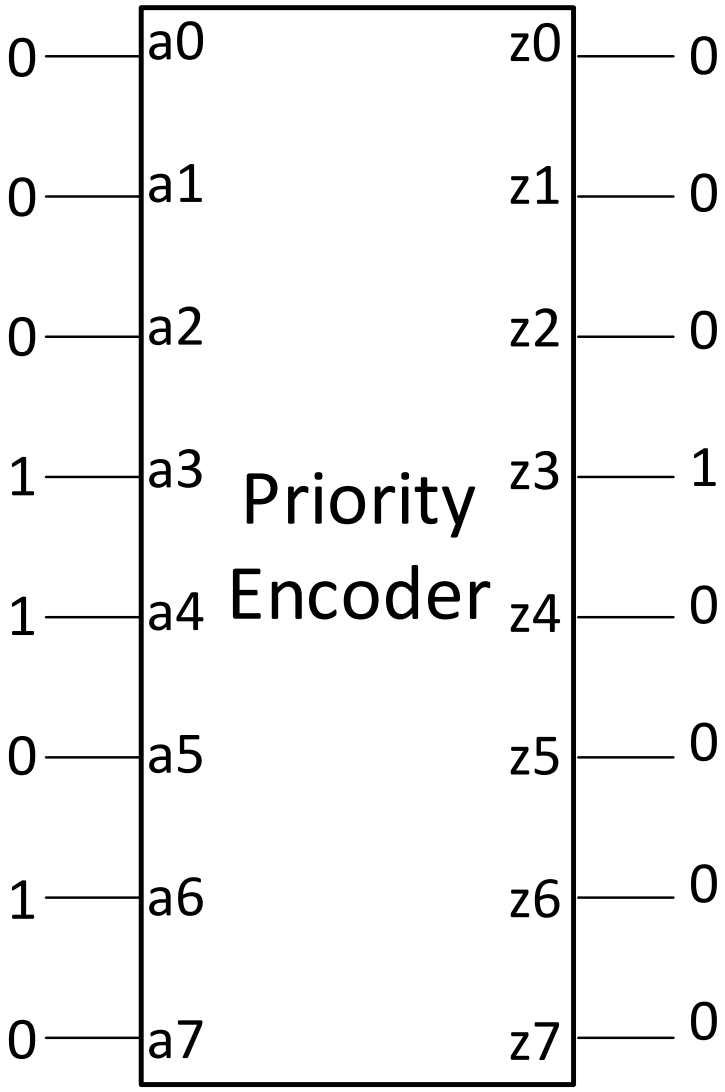
Priority Encoders

- Priority encoders
  - Used for “picking-out” items based on priority
    - Instruction selection in a processor
    - Interrupt controller
  - Output lowest index with a “1” as its input

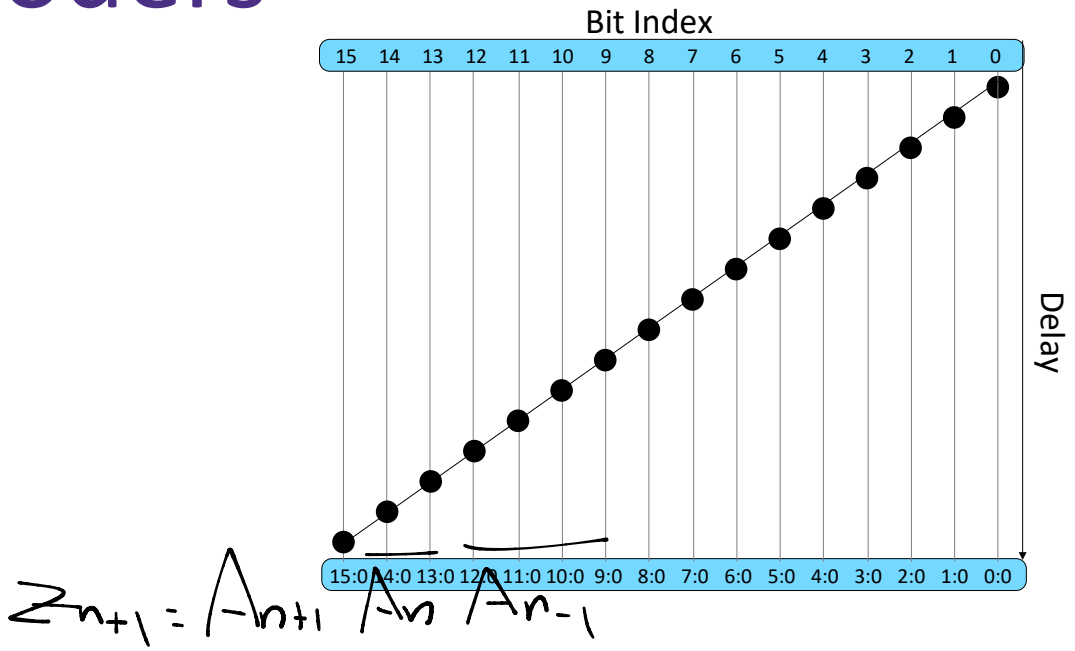


Priority Encoders

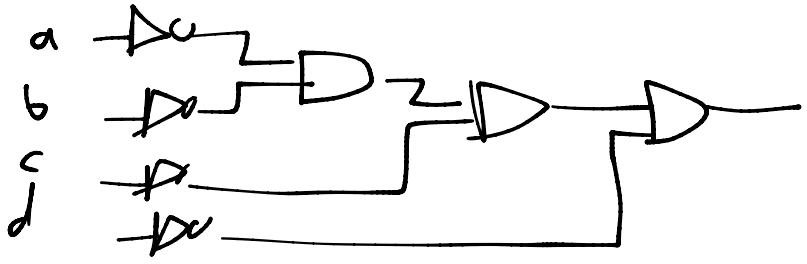
- Priority encoders
  - Used for “picking-out” items based on priority
    - Instruction selection in a processor
    - Interrupt controller
  - Output lowest index with a “1” as its input
    - Equivalent to outputting a 1 if every less significant bit has a “0” input (i.e. it has an all-0 ***prefix***



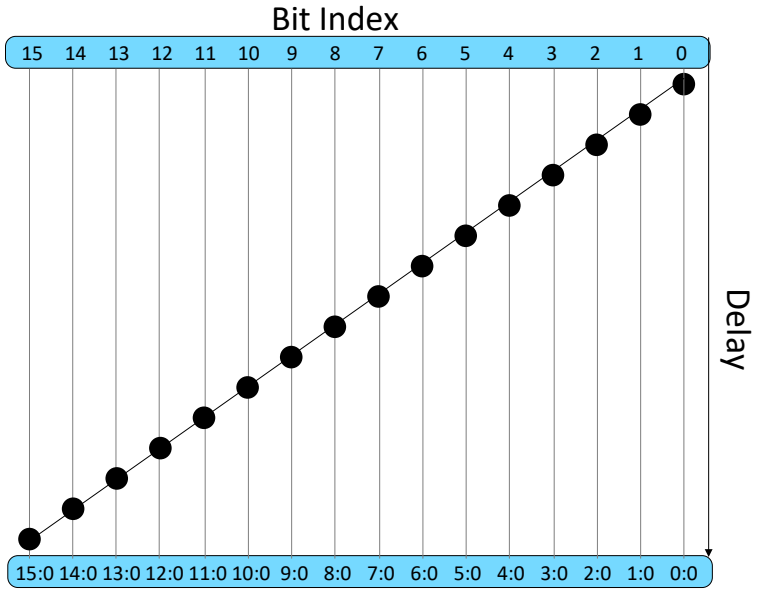
Priority Encoders



- Governing Logic: $Z_n = A_n \cdot \overline{A_{n-1}} \cdot \overline{A_{n-2}} \cdot \overline{A_{n-3}} \ldots \cdot \overline{A_1} \cdot \overline{A_0}$
  - Output depends on previous input bits, specifically, on its prefix

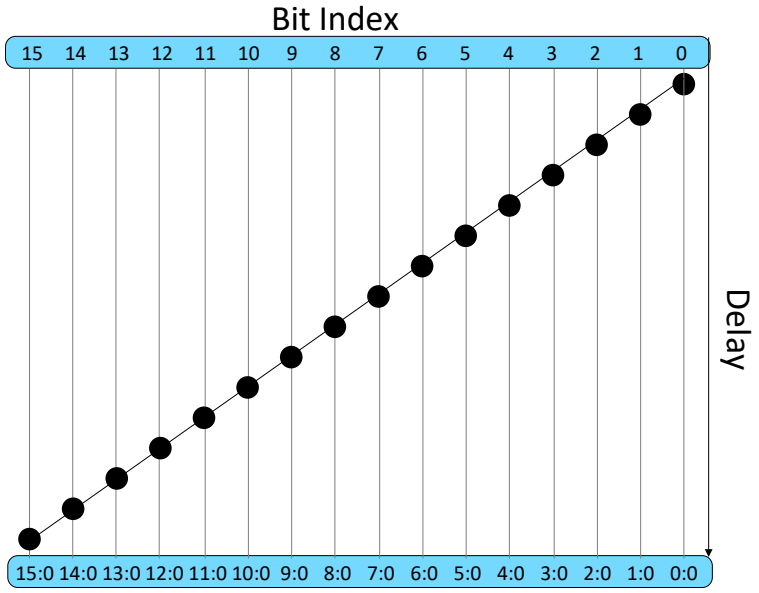


Priority Encoders



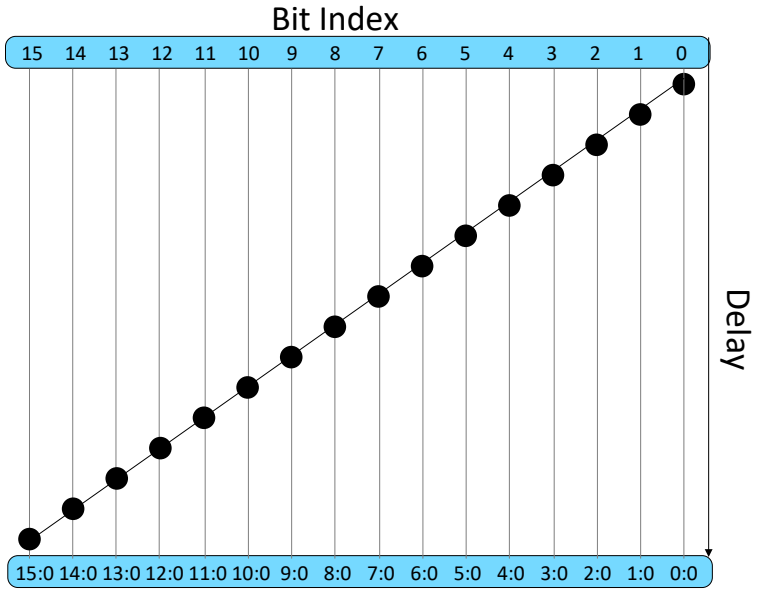
- Governing Logic: $Z_n = A_n \cdot \overline{A_{n-1}} \cdot \overline{A_{n-2}} \cdot \overline{A_{n-3}} \ldots \cdot \overline{A_1} \cdot \overline{A_0}$
  - Output depends on previous input bits, specifically, on its prefix
  - Sequential computation of bit-wise AND is too slow (Sound Familiar?)

Priority Encoders



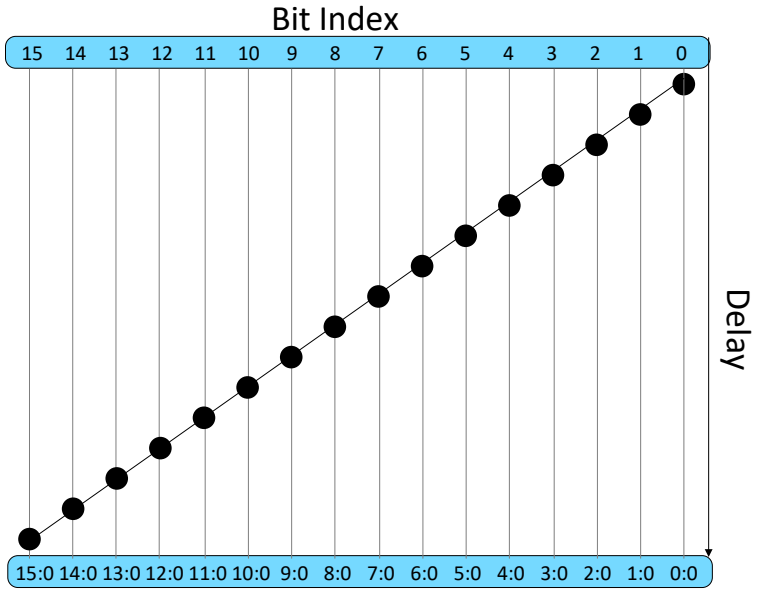
- Governing Logic: $Z_n = A_n \cdot \overline{A_{n-1}} \cdot \overline{A_{n-2}} \cdot \overline{A_{n-3}} \ldots \cdot \overline{A_1} \cdot \overline{A_0}$
  - Output depends on previous input bits, specifically, on its prefix
  - Sequential computation of bit-wise AND is too slow (Sound Familiar?)
  - Define a "Group-vacancy" $V_{i:j} = \overline{A_i} \cdot \overline{A_{i-1}} \ldots \cdot \overline{A_{j+1}} \cdot \overline{A_j}$

Priority Encoders



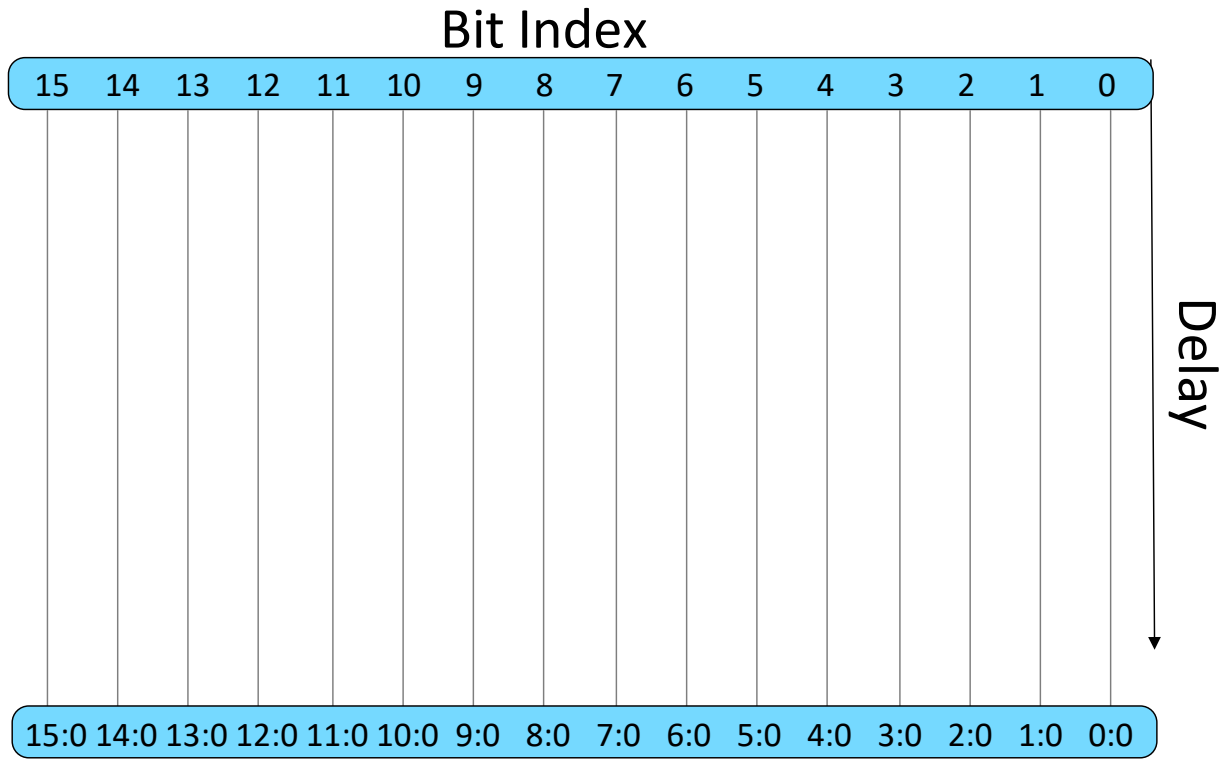
- Governing Logic: $Z_n = A_n \cdot \overline{A_{n-1}} \cdot \overline{A_{n-2}} \cdot \overline{A_{n-3}} \ldots \cdot \overline{A_1} \cdot \overline{A_0}$
  - Output depends on previous input bits, specifically, on its prefix
  - Sequential computation of bit-wise AND is too slow (Sound Familiar?)
  - Define a "Group-vacancy" $V_{i:j} = \overline{A_i} \cdot \overline{A_{i-1}} \ldots \cdot \overline{A_{j+1}} \cdot \overline{A_j}$
  - AND is an associative function: Implement prefix computation in parallel!

Priority Encoders



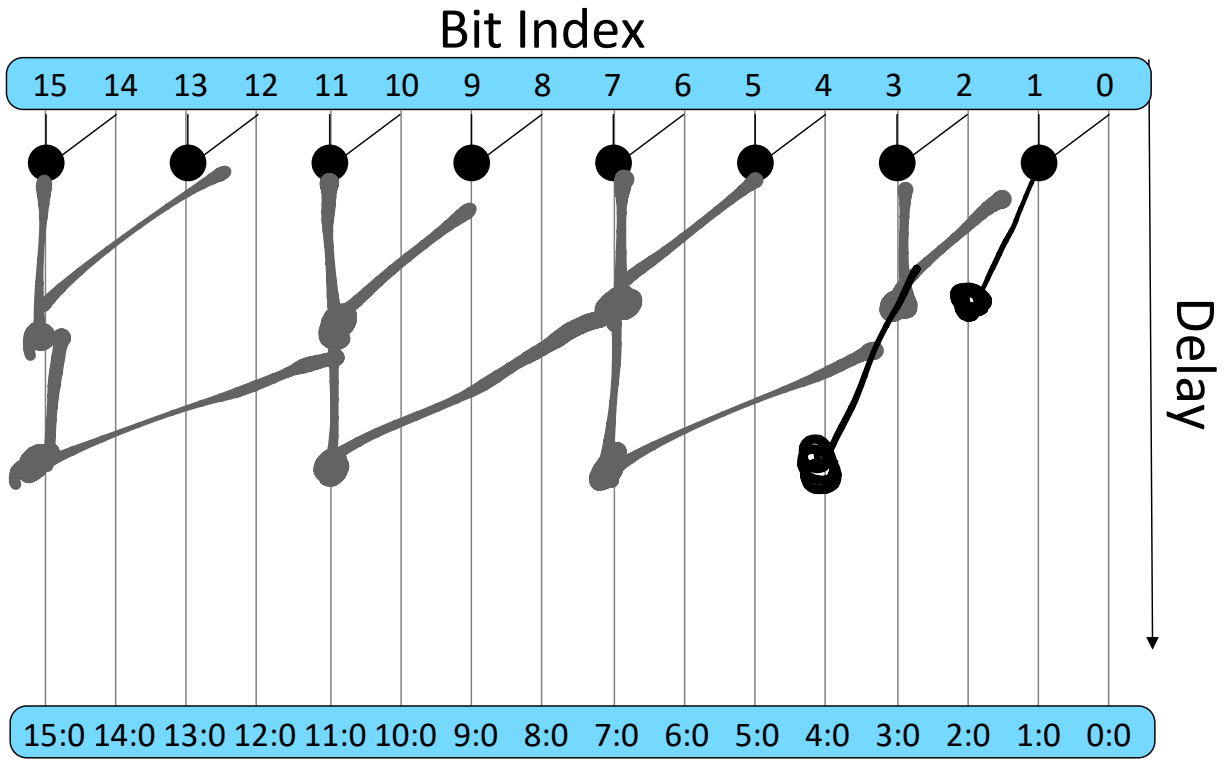
- Governing Logic: $Z_n = A_n \cdot \overline{A_{n-1}} \cdot \overline{A_{n-2}} \cdot \overline{A_{n-3}} \ldots \cdot \overline{A_1} \cdot \overline{A_0}$
  - Output depends on previous input bits, specifically, on its prefix
  - Sequential computation of bit-wise AND is too slow (Sound Familiar?)
  - Define a "Group-vacancy" $V_{i:j} = \overline{A_i} \cdot \overline{A_{i-1}} \ldots \cdot \overline{A_{j+1}} \cdot \overline{A_j}$
  - AND is an associative function: Implement prefix computation in parallel!
  - Associative → Resulting properties apply
    - $V_{i:j} = V_{i:k} \cdot V_{k:j}$
    - $Z_n = A_n \cdot V_{n-1:0}$

Parallel Prefix Computation



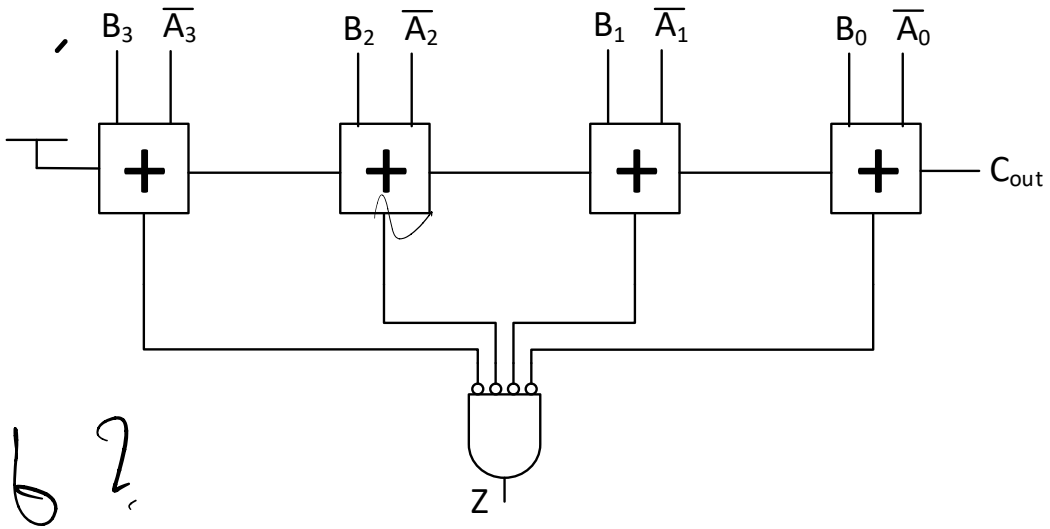
- Exploiting Parallelism

Parallel Prefix Computation



- Exploiting Parallelism

Comparators (Unsigned)



Check	Evaluation
A==B	Z
A>B	C
A<B	C

is $a > b$?

$a - b \geq 0$?

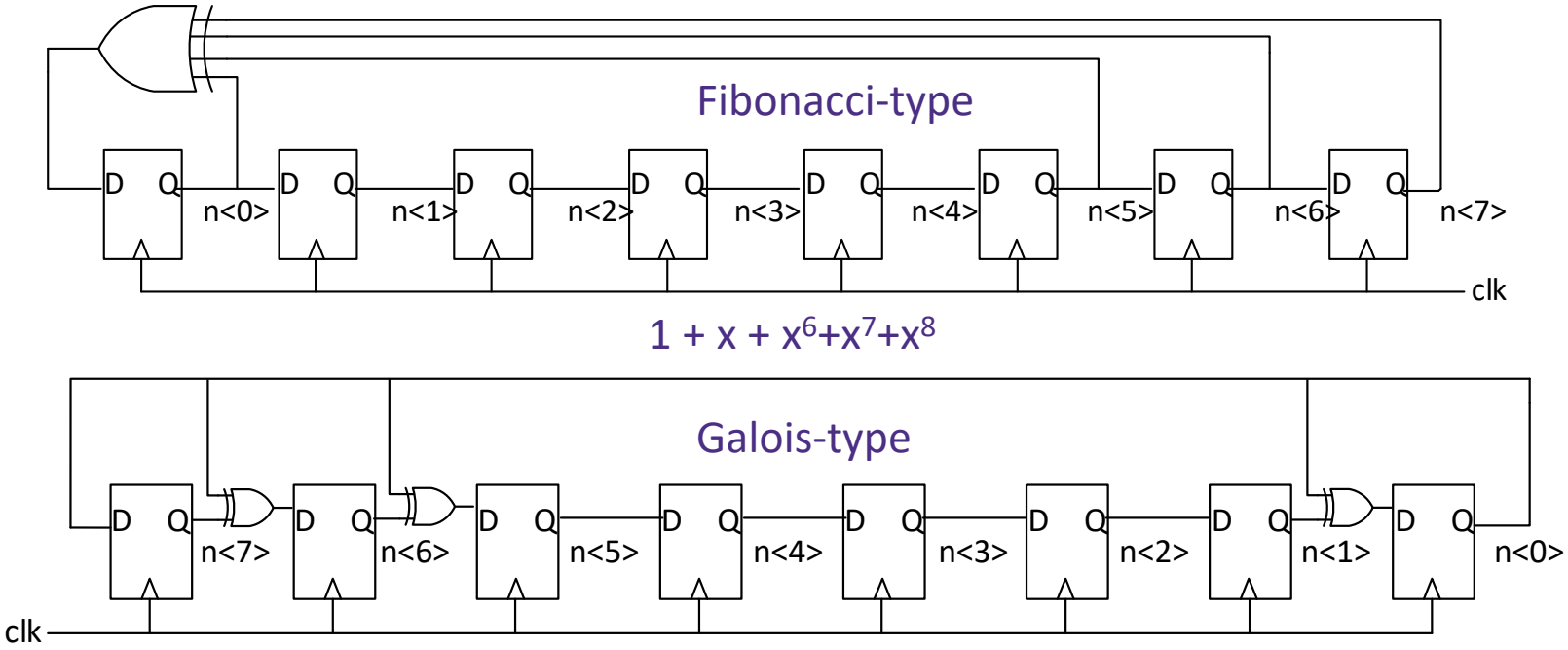
$b - a < 0$?

$a - b = a + \bar{b} + 1$

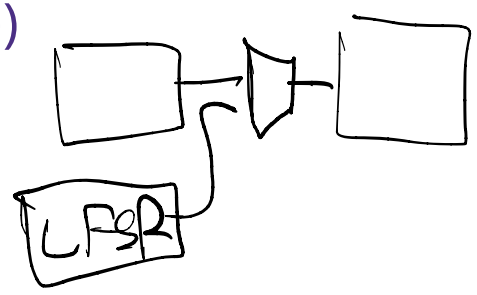
$b - a = \bar{a} + b + 1$

- Perform subtraction followed by 0-detection
- Signed Comparators are similar, but require handling the overflow case
- How do you check which input is larger?

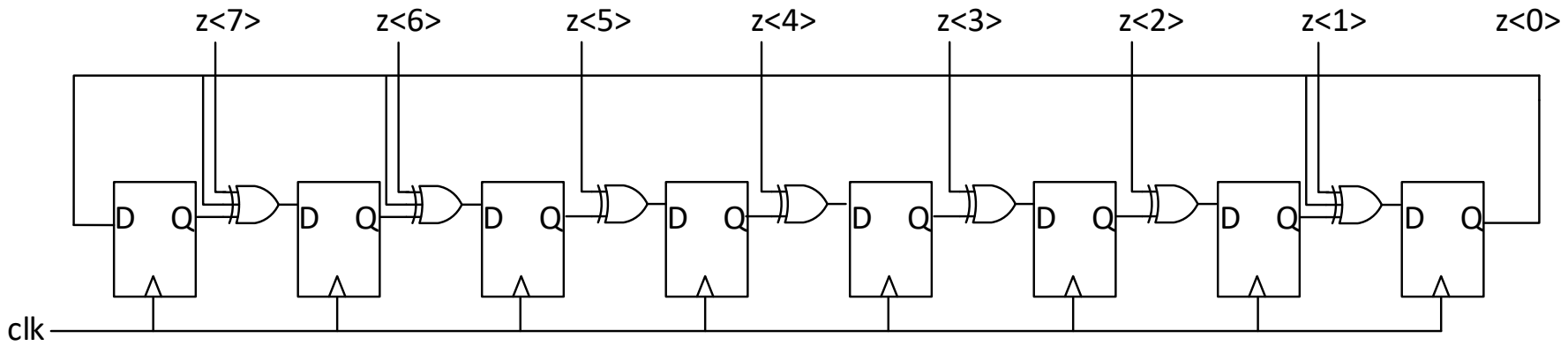
LFSR

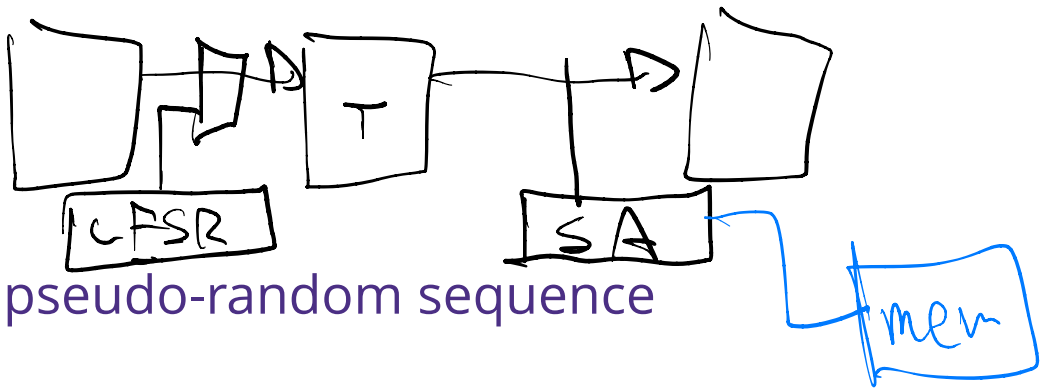


- Capable of generating a pseudo-random number sequence
  - E.g. n-bit LFSR will go through all 2^n possibilities in a pre-determined sequence
  - Uses a generator (primitive polynomial) to produce all entries of a finite field
  - Implementations fall under 2 types (Fibonacci-type and Galois-type)
  - Primitive polynomial tables readily available for n-bit LFSRs
- Used in Built-In Self Test (BIST)
  - Pseudo-ransom “data” generation for Datapath test
  - Compression of output data

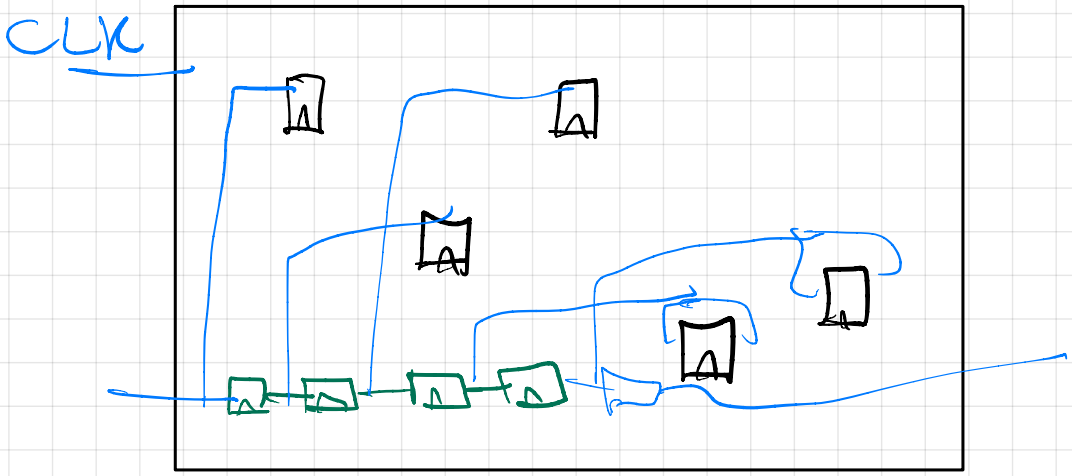


Signature Analyzer

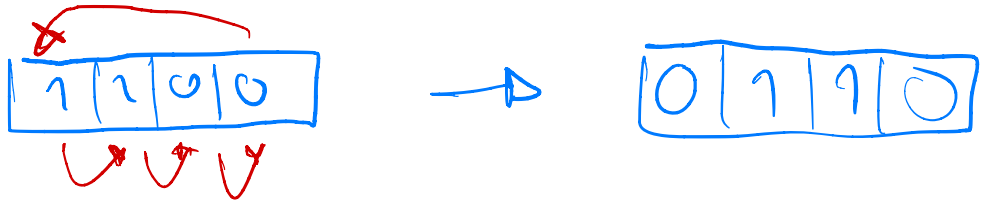




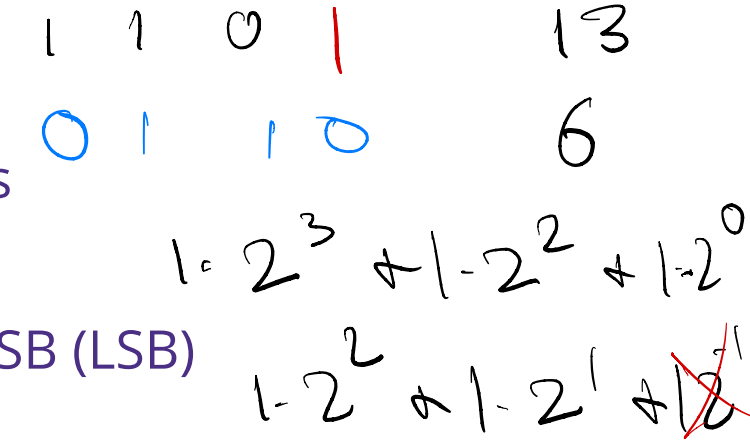
- Basic LFSR structure
  - Incorporate XOR at each input
  - Data output from a digital block “mixes” into pseudo-random sequence
  - Run test for many cycles, observe signature
  - Single-bit output error will result in a significantly different signature
  - If signatures match, it is ***very*** likely that there was no error in any computation


CLK

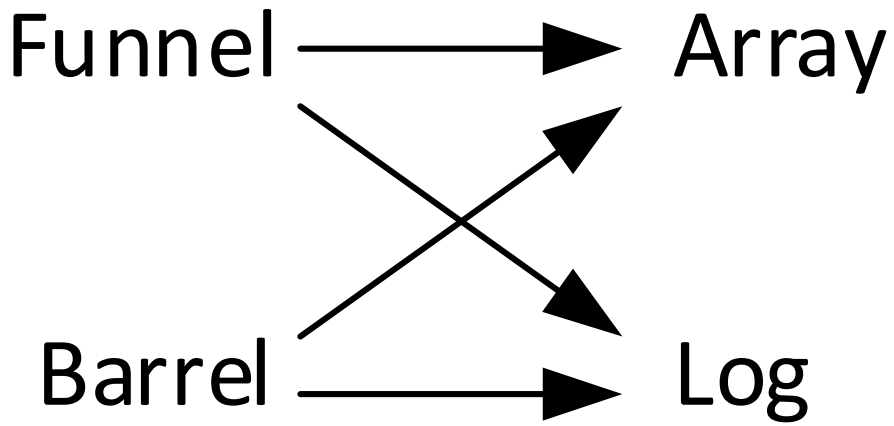
Shifters



- Efficient VLSI implementation
- Rotate/Shift (Logical and Arithmetic) frequency used in microprocessors (e.g. R1 << 3)
- Efficient means of binary multiplication and division
  - Processors without a multiplier rely on shift and add operations
- Shift operations
  - Rotate: ROR (ROL) shifts with LSB (MSB) wrapping around to MSB (LSB)
    - 6'b100110 ROR 1 = 6'b010011, 6'b100110 ROL 3 = 6'b110100
  - Logical Shift: LSL (LSR) shifts bits to the left (right) inserting 0's in the LSB(MSB)
    - 6'b100110 LSL 2 = 6'b011000, 6'b100110 LSR 3 = 6'b000100
  - Arithmetic Shift
    - ASL same as LSL : 6'b100110 ASL 2 = 6'b011000
    - ASR same as LSR **except** MSB bits filled by sign extension : 6'b100110 ASR 3 = 6'b111100

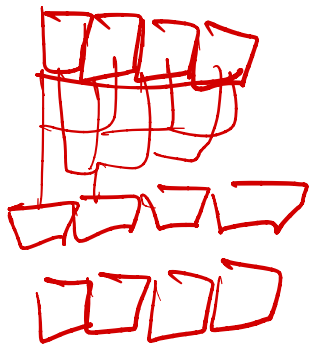


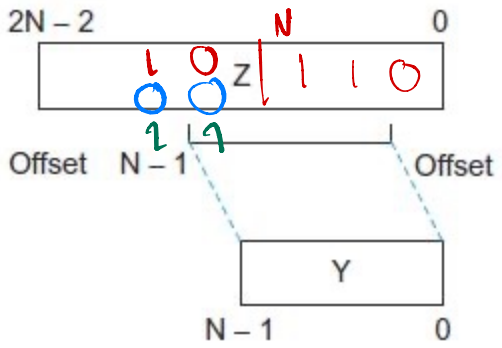
Shifter Design

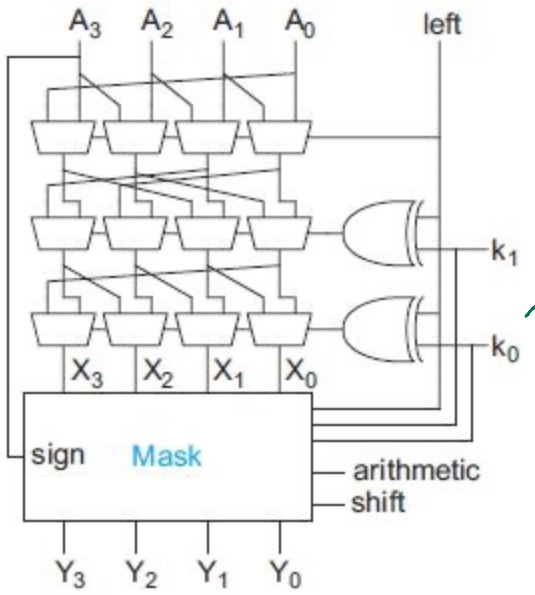


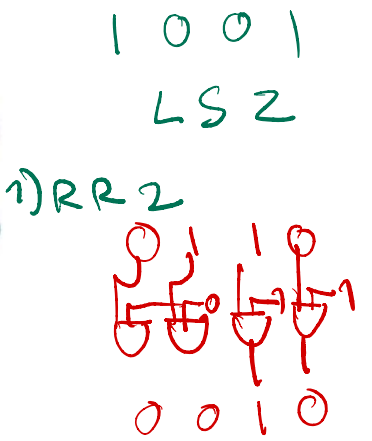
- Two main conceptual implementations
  - Funnel
  - Barrel
- Two main physical implementations
  - Array
  - Log

Funnel and Barrel shifters



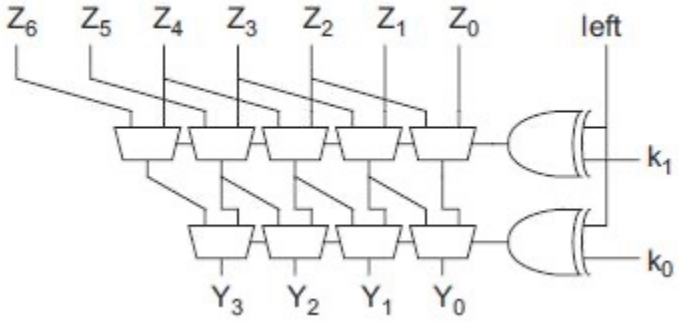


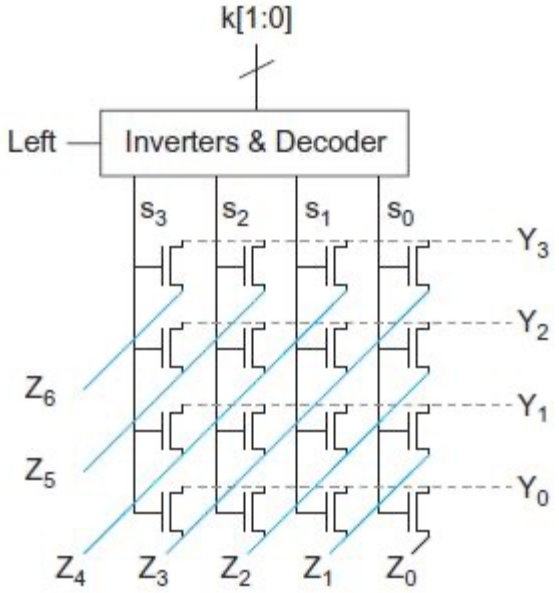




- Funnel Shifters
  - Operate on a 2N-1 bit set of data. Prepare data-set depending on operation
  - Select N-1 bits from this broader range (Funnel down from 2N-1 to N-1 bits)
- Barrel Shifters
  - System naturally implements a “rotate-right”
  - Shift-right performed by rotating right, and “masking off” msbs by 1 or 0
  - Rotate-left by RL performed by rotating right by 2^n-RL = ~RL + 1
  - Shift-left performed by Rotate-left followed by masking off LSB bits

Log and Array Shifters





- Log shifters
  - Perform an n-bit shift as n successive shifts
  - First shift by 0 or 2^n based on n-th bit. Continue down to shift by 0 or 1
  - Better suited to large designs (Log in the number of stages required)
- Array shifters
  - Compact and handy for smaller cells
  - Essentially perform a wide mux operation
  - Be careful of the wiring load impact!

Shifter Architectures

- Funnel Shifter
  - Create a sequence $Z_{2n-2:0}$ based on input $A_{n-1:0}$
  - Select n-bits from Z with an offset k forming the LSB of the output Y
  - Performs right shift/rotate naturally
  - Left shift/rotate : Rotate or shift right by 2's complement

- Barrel Shifter Architecture
  - Perform right-rotate (Handle left-shift using 2's complement)
  - Mask output bits if operation is a shift

Shifter Architectures

- Funnel Shifter
  - Create a sequence $Z_{2n-2:0}$ based on input $A_{n-1:0}$
  - Select n-bits from Z with an offset k forming the LSB of the output Y
  - Performs right shift/rotate naturally
  - Left shift/rotate : Rotate or shift right by 2's complement
  - "Funnel shifter architecture"

- Barrel Shifter Architecture
  - Perform right-rotate (Handle left-shift using 2's complement)
  - Mask output bits if operation is a shift

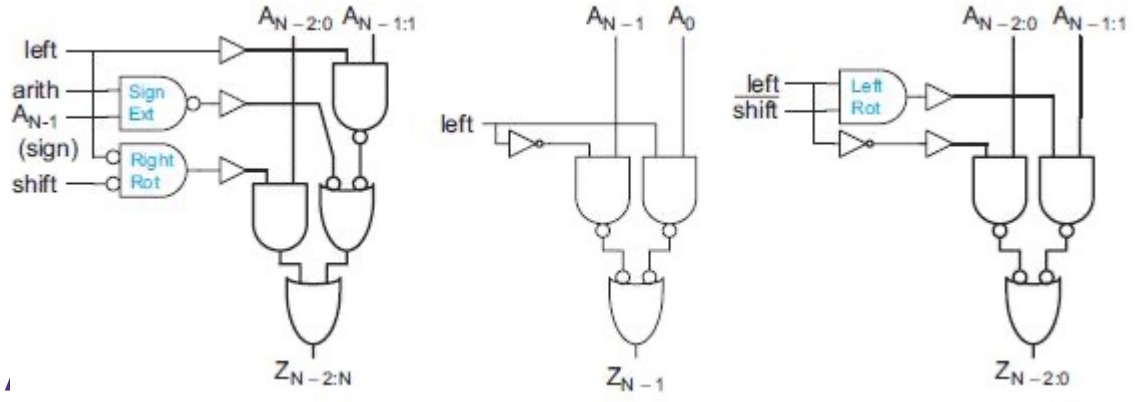
Shifter Architectures

- Funnel Shifter
  - Create a sequence $Z_{2n-2:0}$ based on input $A_{n-1:0}$
  - Select n-bits from Z with an offset k forming the LSB of the output Y
  - Performs right shift/rotate naturally
  - Left shift/rotate : Rotate or shift right by 2's complement
  - "Funnel shifter architecture"

- Barrel Shifter Architecture
  - Perform right-rotate (Handle left-shift using 2's complement)
  - Mask output bits if operation is a shift
  - "Barrel Shifter architecture"

Funnel Shifters

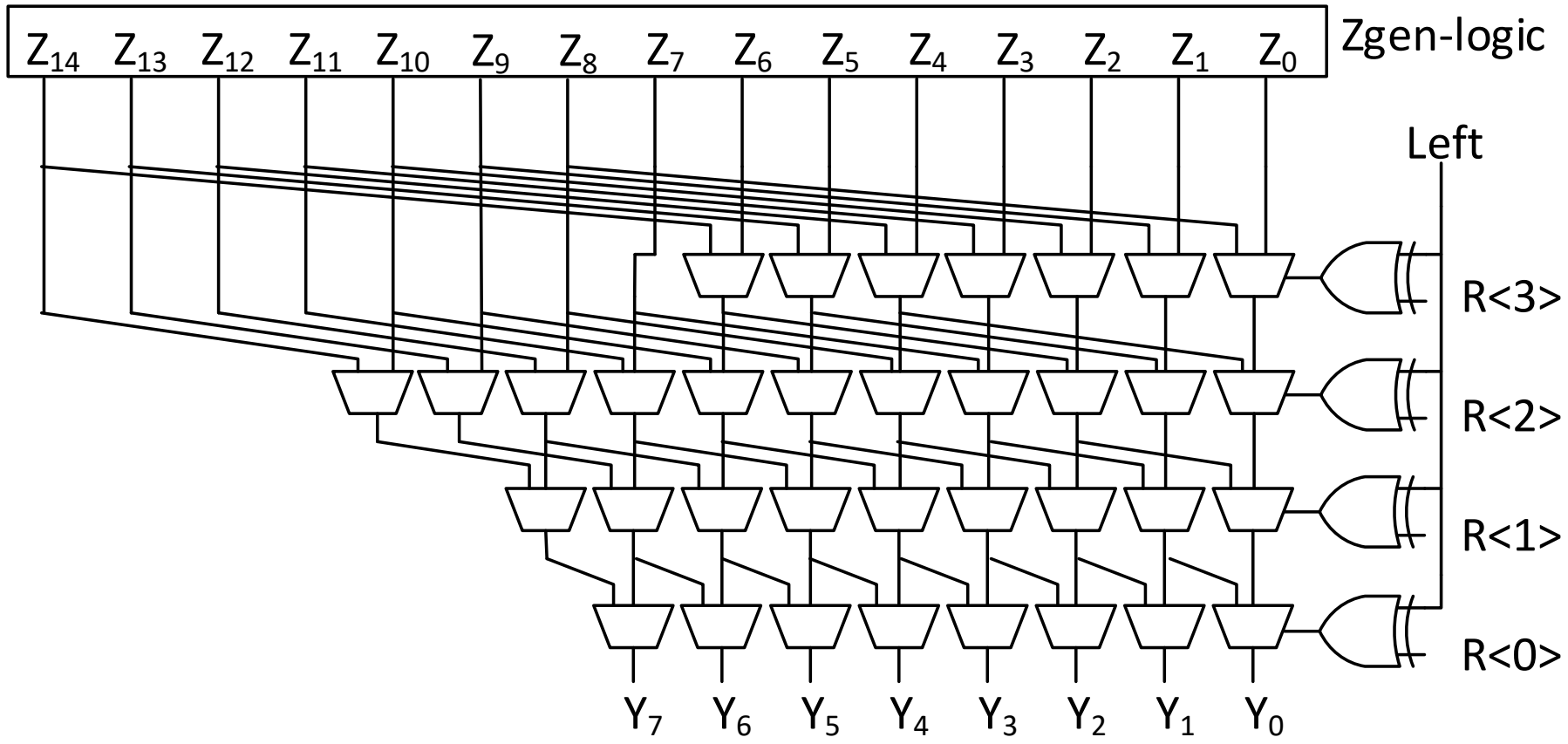
Operation	$Z_{2N-2:N}$	Z_{N-1}	$Z_{N-2:0}$	Offset
Logical R	0	X<N-1>	X<N-2:0>	K
Arithmetic R	X<N-1>	X<N-1>	X<N-2:0>	K
Rotate R	X<N-2:0>	X<N-1>	X<N-2:0>	K
Logical L	X<N-1:1>	X<0>	0	~K
Arithmetic L	X<N-1:1>	X<0>	0	~K
Rotate L	X<N-1:1>	X<0>	X<N-1:1>	~K



- Selection of Z-bits based on L/R, A
- Log-based approach more amenable to larger shift values
- Use of wider-mux for improved delay

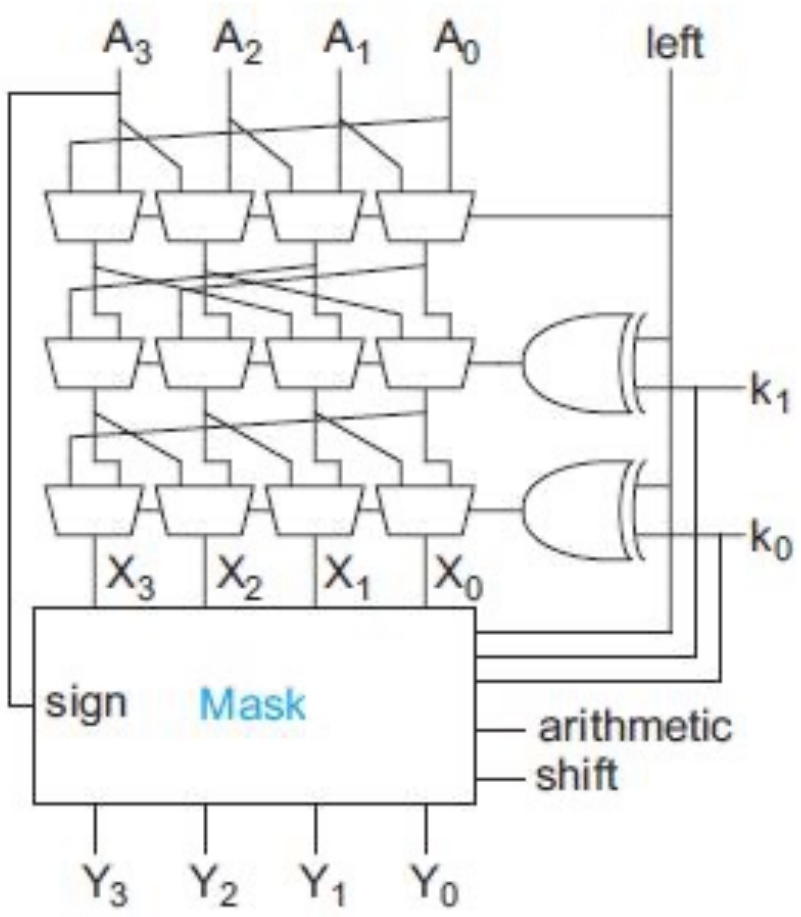
Source: W&H

Log-Based Funnel Shifter Example



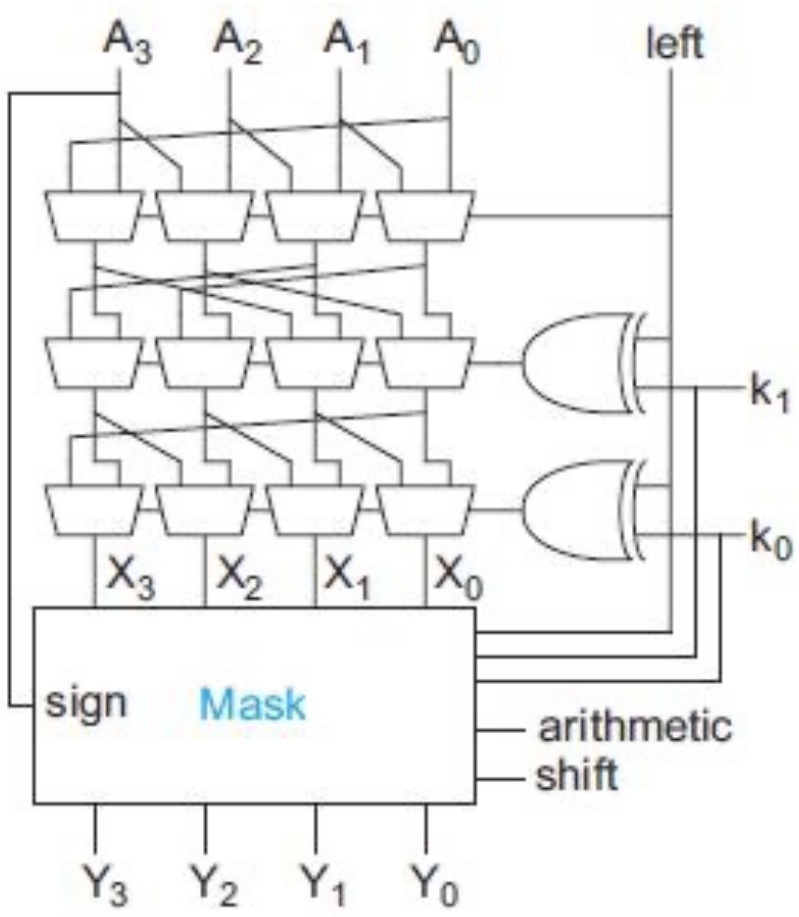
- Log-based approach more amenable to larger shift values
- Use of wider-mux for improved delay

Barrel Shifter Example



Taken from Weste & Harris

Barrel Shifter Example



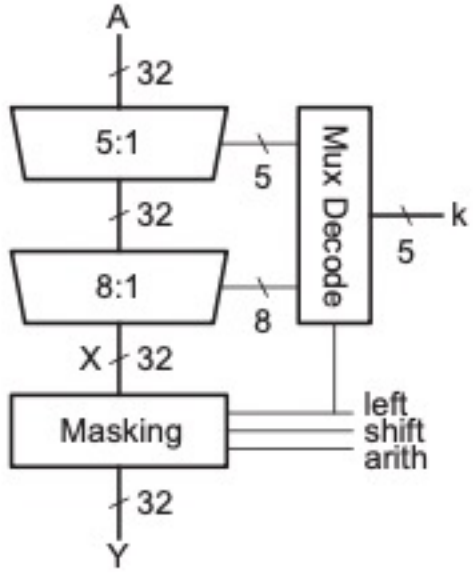


FIGURE 11.69 32-bit logarithmic barrel shifter

"The first stage rotates right by 0, 1, 2, 3, or 4 bits to handle a prerotate of 1 bit and a fine rotate of up to 3 bits combined into one stage. The second stage rotates right by 0, 4, 8, 12, 16, 20, 24, or 28 bits. "

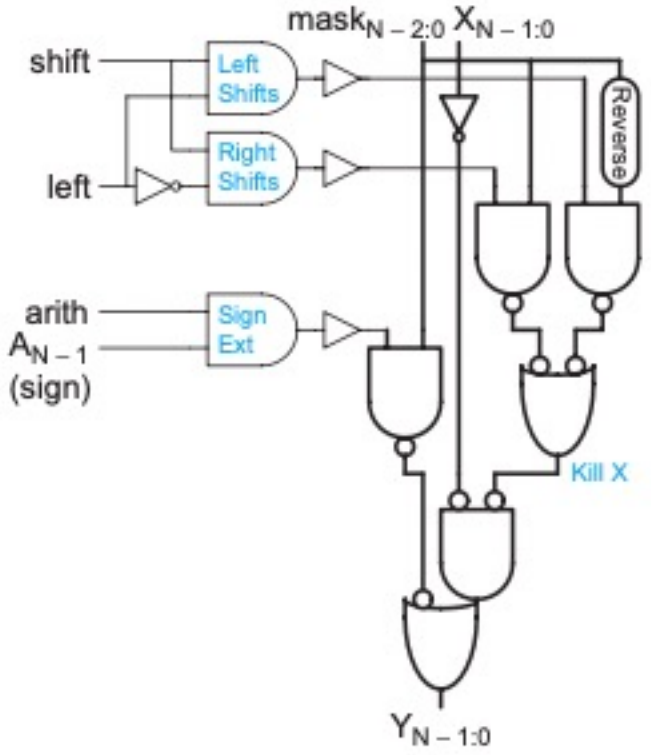


FIGURE 11.70 Barrel shifter masking logic

Taken from Weste & Harris

Questions/Assignments

- Funnel shifter based on a left shift..
- Array barrel shifts
- Kogge-stone priority encoder
- Absolute difference engine

Reading Assignment

- Required: Familiarity with 2's complement. Search online for sources. See me if help is needed here.
- Required W&H
  - 11.5 – 11.5.2
  - 11.3
  - 11.4
  - 11.8
- Extra Reading W&H
  - 11.7