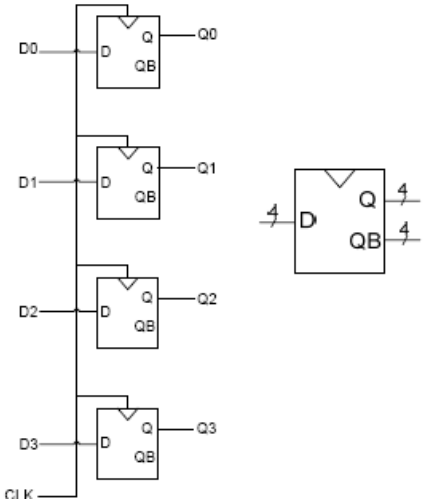
Today

- Memories
- Sequential MSI devices
- Start on combinational logic minimization.

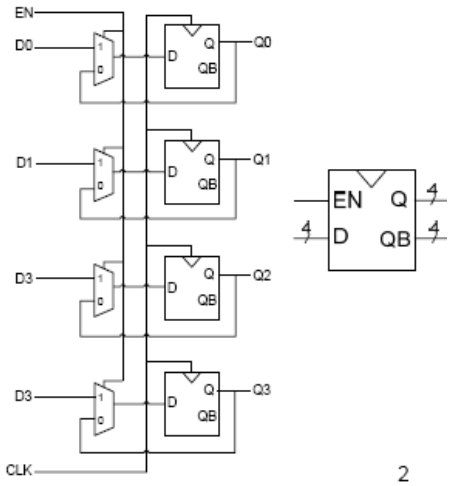
Sequential MSI devices.

Registers are a collection of 2 or more D flip-flops working in parallel.

- A collection of two or more D flip-flops with a common clock is called a **register**
- Used to store a collection of bits



- Adding a mux at the input of each D FF allows for loading new value or storing current value



2

Counters

A simple counter is a device that increments its value every clock cycle. (So 0, 1, 2, etc.). It generally has an enable (to count or stay where it is) and a reset (to go back to zero on the next rising edge.

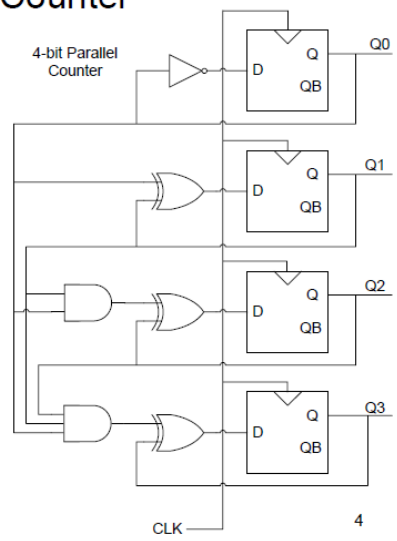
1. Design a 4-bit (count 0 to 15) counter using standard MSI devices.
2. Design a "count from 0 to 9" counter using your 4-bit counter.

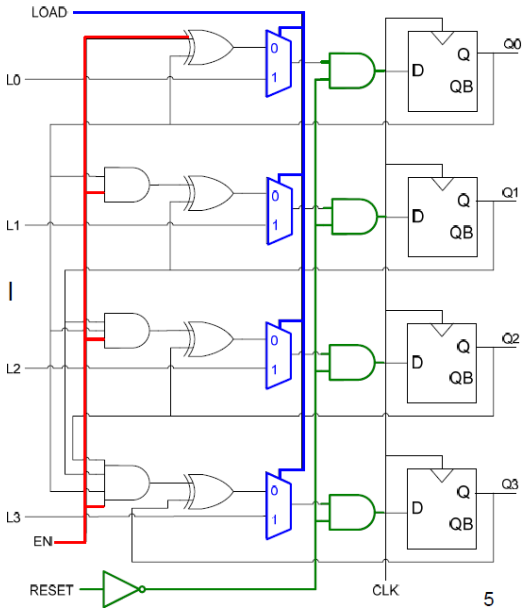
Parallel Counter

- Would like *all* state variables to run on the same clock signal
- Examining the binary code yields an easy implementation:

b_2	b_1	b_0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1
0	0	0

- Under what condition does each bit toggle?
- *When all bits of lesser significance are 1!*
- This is true no matter how many bits in the counter





- How do the above circuits work?

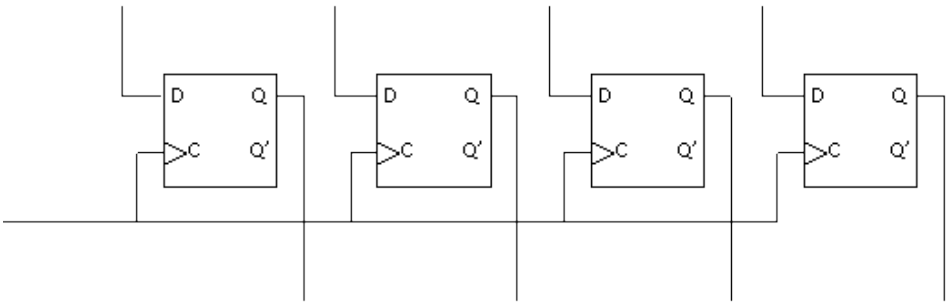
- Can you build a counter out of T flip-flops?

- What are the problems with this “ripple” counter? What’s nice about it?

Terminology

- **Counters** rotate through a set of values. Most count from 0 to N.
  - Some can count both up and down (generally called **up/down** counters)
  - Most have **reset** and **enable**.
  - One question is what to do with a counter that reaches the maximum (or minimum) value.
    - **Saturating** counters stay at that value
    - **Modulo** counters wrap around.
  - We can have **ripple** or **parallel** counters.
    - Ripple is simple, but has some issues.
    - Parallel is more complex, but better behaved.

- **Shift registers** move the data one way or the other.
  - We can have shift registers that shift either way, load or reset all with the aid of counters.
  - Let's design a 4-bit shift register that takes two inputs X[1:0]. If X=3 we shift left, if X=2 we shift right, if X=1 we hold our value, and behavior is undefined if X=0.



Minimization of Combinational Circuits (6.2)

Notes:

- I have posted a set of slides that does this very formally later today. They are a _really_ good set of slides. Look them over.
- The text does a pretty good job here (though not as good/formal as the slides).

One good question is "how do we minimize combinational circuits?" And the answer to that question is more complex than you'd think. Part of the problem is it isn't clear what it means to minimize. Do we mean to use the fewest number of gates? If so, we could end up taking something like the canonical sum-of-products and making a really, really, long path. That would greatly slow the clock.

We *probably* mean to reduce the number of gates as much as possible while keeping the delay bounded by some value. In this class, we'll just look at finding the minimal sum-of-products and product-of-sums forms for a given logic equation. This is called *two-level logic* because the worst-case path through the circuit will consist of exactly two gate delays.

Motivational Example #1

Consider the logic statement F(a,b,c)=ab'c+abc+abc'. You'll notice that we are using three 3-input AND gates and one three input OR gate to implement this logic (let us assume that all literals are available, so a and a' can both be used and don't require a NOT gate). What can you do to reduce the complexity of this circuit while keeping ourselves to two levels of logic?

First notice that ab'c+abc can be combined (using the "combining rule" from the first lecture of the year). We can also

combine abc+abc'. So we get: _____ as a simplified version of this. As you've likely noticed, it isn't always easy to find pairs that can combine, and sometimes those pairs are well hidden inside of various terms.

Motivational Example #2

Consider the following: a'b'c+a'cd'+bcd. It turns out that equation can be reduced to a'c+bcd. How the heck can we figure that out? One trick is to drop back to the canonical sum-of-products form. That gets us:

You might notice that all four the terms that have a'c in them are true. So we can combine those four into a'c. That leaves abcd. It can be combined with a'bcd to make bcd. It turns out we can't combine any more and this point and so we've succeeded!

We got the right answer this way, but it wasn't obvious. Heck, it wasn't obvious that we are done. So the question is, how do we make it obvious? How can we quickly see what terms combine and prove to ourselves that we are done? What we'd like is an easy way to visualize exactly

What we are looking for

Say there are 3 variables: a, b and c. A given term (say abc) can be combined with three other terms (a'bc, ab'c and abc' in this case). We want a way to see those connections quickly.

Karnaugh Maps

Consider ab'c+abc+abc' from above. Let's write the truth table in a rather odd way:

ab/c	00	01	11	10
0	0	0	1	0
1	0	0	1	1

Notice that box which represents abc is adjacent to the three different terms it can be combined with. In this case we can quickly see how the combining rule can be applied. We end up with two product terms: _____ and _____

Now consider

ab/c	00	01	11	10
0	1	1	1	1
1	0	0	0	0

In this case, we can combine all four terms into a single term: _____

ab/c	00	01	11	10
0	1	0	0	1
1	0	0	0	0

And notice we "wrap around" the edges. So this combines to be: _____

Informal Algorithm

What we are going to do is circle groups of "1s" that are rectangles[1] where each side is a power of 2 in length. The first K-map on this page had a 2x1 and a 1x2 rectangle, the second a 1x4 and the third a 1x2. We never circle a rectangle that is part of a larger legal rectangle. We only circle enough rectangles so that every 1 is circled.

ab/c	00	01	11	10
0	0	0	1	1
1	0	1	1	0

Notice that in the K-map above there are three rectangles we could circle, but two of them cover all the "1s".
Let's practice a bit.

ab/c	00	01	11	10
0	1	1	1	1
1	0	0	1	0

_____

ab/c	00	01	11	10
0	1	0	0	1
1	0	1	1	1

_____

ab/c	00	01	11	10
0	1	1	1	1
1	1	1	0	1

_____

[1] Recall a square is special case of a rectangle.

An interesting example

Circle all rectangles:

ab/c	00	01	11	10
0	1	1	1	0
1	1	0	1	1

Answer 1

ab/c	00	01	11	10
0	1	1	1	0
1	1	0	1	1

Answer 2

ab/c	00	01	11	10
0	1	1	1	0
1	1	0	1	1

Terminology (p308-310)

Notice that we are finding sum-of-products solutions.

- Recall that a ***minterm*** is a product term that includes all of the functions variables exactly once.
- The ***on-set*** of a function is the set of minterms that define when the function should evaluate to 1 (the minterms that have a 1 in the truth table.)
  - The ***off-set*** is the set of minterms that evaluate to zero.
- An ***implicant*** of a function is a product term that evaluates to 1 only in places that function evaluates to 1. (The on-set of an implicant of a function is a subset of the on-set of the function.)
  - Graphically, in a K-map an implicant is: _____
- An implicant ***covers*** those minterms that appears in its on-set.
  - What is the on-set of the function F(a,b)=a? _____
  - What minterms does that function cover? _____
- Removing a variable from a term is known as ***expanding*** the term. This is the same as expanding the size of a circle on a K-map.

ab/c	00	01	11	10
0	1	1	0	0
1	1	1	0	0

- ***Prime implicant***:_____
- ***Essential one***[2]: _____
- ***Essential prime implicant***: _____

[2] This term isn't used by our text, they skip from prime implicant directly to essential prime implicant.