We have seen how to create a n-bit register from D flip flops.

We will now consider combining these registers to form a multibyte memory.

Our motivation is to illustrate conceptually how memory could be constructed. In practice, the design and construction of commercially available memory is highly specialized and complex: many innovative engineering techniques are employed to increase access speed and to minimize physical size.

Nevertheless, the memory we construct here is conceptually equivalent in every way to its commercial equivalent.

When we first encountered Memory in CS1111/CS5020 we saw that two types of technology were used:

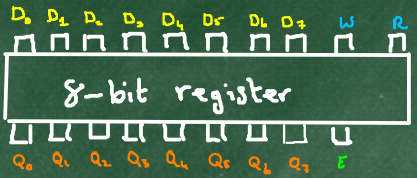
Dynamic Random Access Memory (DRAM)
—for main memory

and Static Random Access Memory (SRAM)
—for Cache memory

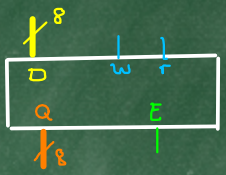
Each with their associated advantages and disadvantages.

The memory we're constructing here is akin to SRAM.

First, let's recall our abstraction of an 8-bit register from Lecture 19:

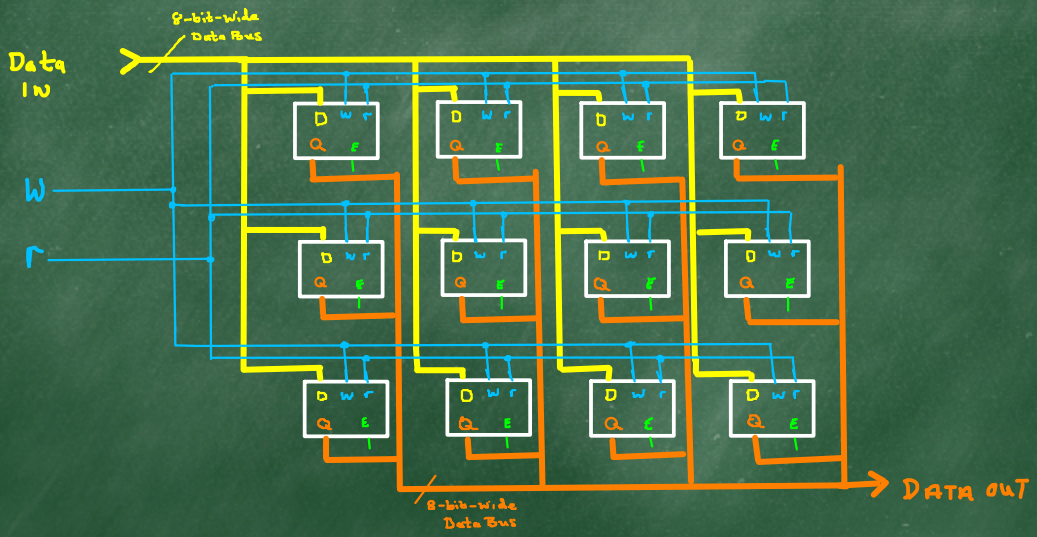


and let's represent it graphically as:



Here, the thick lines associated with D and Q represent a group of 8 lines. We call these collections, **busses**.

Now, let's connect 12 of these 8-bit registers to create a 12 byte memory:



(i) All W inputs connected together.
(ii) All r inputs connected together.
(iii) All D_i inputs connected together.
(iv) All Q_i inputs connected together.

- This arrangement could be extended to make bigger memories.

- Each 8-bit register represents a memory location, capable of storing a single byte.

The connections specified above ensures that each register is presented with the same inputs:

- Each 'sees' the same values on the D bus
- Each is in the same mode (w or r) at the same time
- Each is connected to the Q bus and so can potential place their data onto it simultaneously.

Obviously, such an arrangement results in chaos! To eliminate this chaos, we can use the register enable inputs in such a way that only one register is enabled at a time.

This process is known as addressing.

How can we do this?

We could begin by numbering each memory location.

We note that we have 12.

To count to 12 in binary we need at least 4 bits.

In fact, with 4-bits we could go as far as 15 (including 0).

We will call the number associated with each location, its address.

Therefore, in our example, we have the addresses:

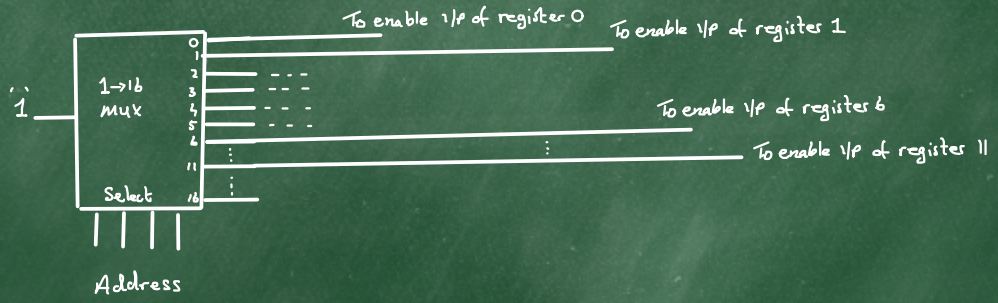
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011

We would like to have a circuit with 12 outputs.

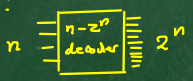
For each input address we want only 1 corresponding output to be = 1 and all of the others to be = 0

	outputs											
Inputs	0	1	2	3	4	5	6	7	8	9	10	11
0000	1	0	0	0	0	0	0	0	0	0	0	0
0001	0	1	0	0	0	0	0	0	0	0	0	0
0010	0	0	1	0	0	0	0	0	0	0	0	0
0011	0	0	0	1	0	0	0	0	0	0	0	0
0100	0	0	0	0	1	0	0	0	0	0	0	0
0101	0	0	0	0	0	1	0	0	0	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
1010	0	0	0	0	0	0	0	0	0	0	1	0
1011	0	0	0	0	0	0	0	0	0	0	0	1

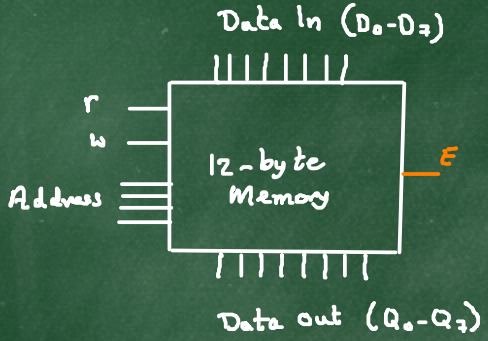
Luckily, we have a circuit for that: the multiplexer:



Note: there is a chip called a decoder that places a 1 on one of its 2^n output corresponding to the value on its n-inputs:

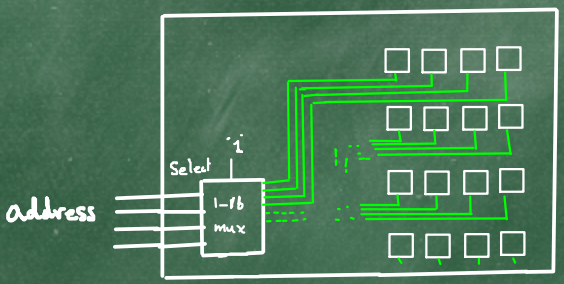


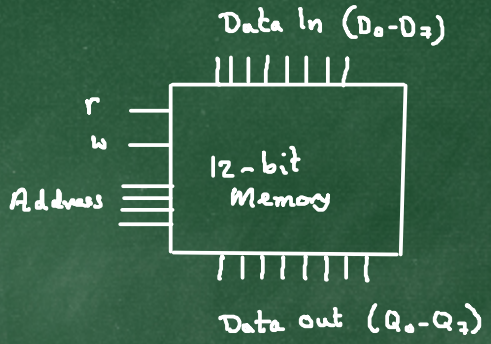
Let's now abstract further to create a 12-byte memory:



In practice the Data In and the Data out pin are not separate – a single set of pins carry the data in/out depending on read/write mode

Peeking inside to see the addressing:





In practice the Data In and the Data out pin are not separate – a single set of pins carry the data in/out depending on read/write mode

What about the address?

In our simple memory module we will address each bytes individually using the enable line of its respective register.

By turning only one register on, at a time, in this fashion will give it exclusive access to the data In and data out buses.

In our simple memory, we could use a 1-16-Mux to enable the appropriate register: