



Topic Notes: Building Memory

We'll next look at how we can use the devices we've been looking at to construct memory.

Tristate Buffer

We've seen and used inverters.

We also mentioned when we first saw inverters that we can put two together in series and have a *buffer*.

We can draw one of these as a triangular symbol but with no circle at the end.

What use is that? It will slow down but also boost the current in the circuit.

Recall that large fan-out can cause trouble with not enough current to drive subsequent gates. A buffer can take care of that.

We think about a wire or an output as having values of 0 or 1. It can be easy to think about the value 0 as the absence of a value, but that's not true.

Connected to ground or to a low gate output is different than not connected.

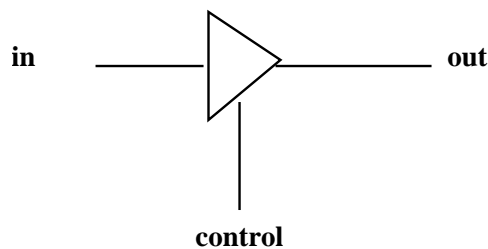
You can think of the wire as transmitting some value. A 0 value means the wire like a pipe spilling out 0's, a 1 value is like a pipe spilling out 1's.

Or like a pipe with hot water flowing vs. a pipe with cold water flowing.

But a disconnected wire or a pipe with no water source isn't spilling out anything.

So even in our circuits so far, there could be a wire or a gate not connected to anything.

Now, we'll look at a new device called a *tristate buffer*.



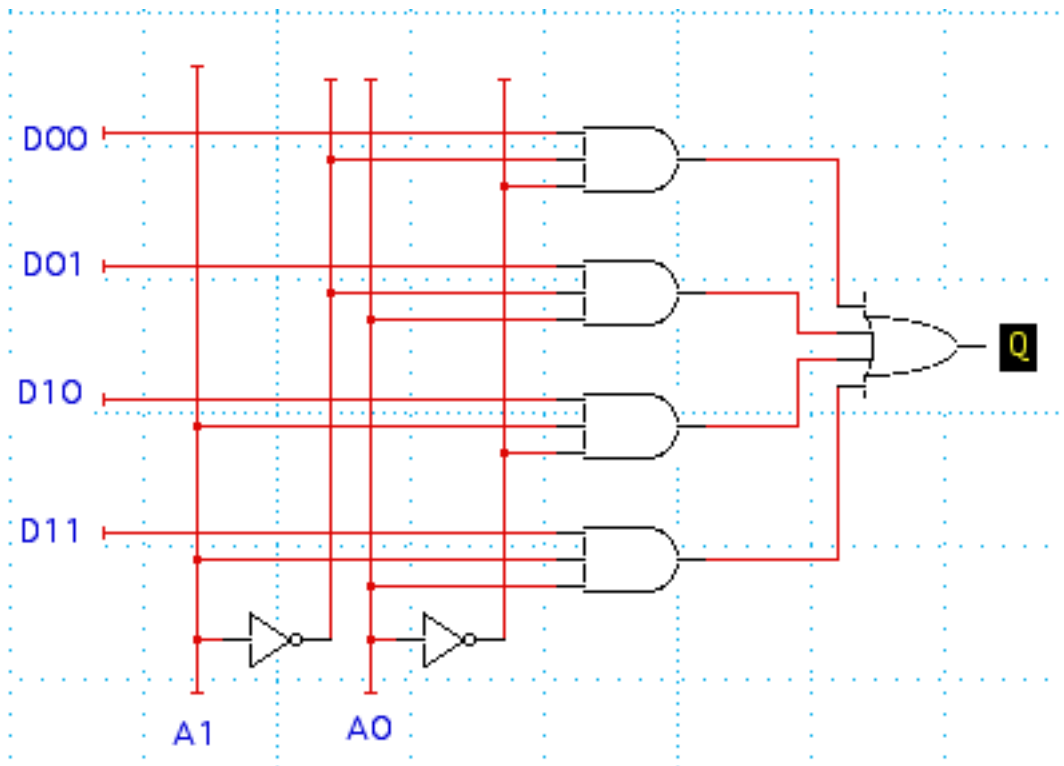
When the control line is high, we have a wire (with some amplifying properties) like a buffer.

When control is low, it looks like a broken wire – physically disconnected.

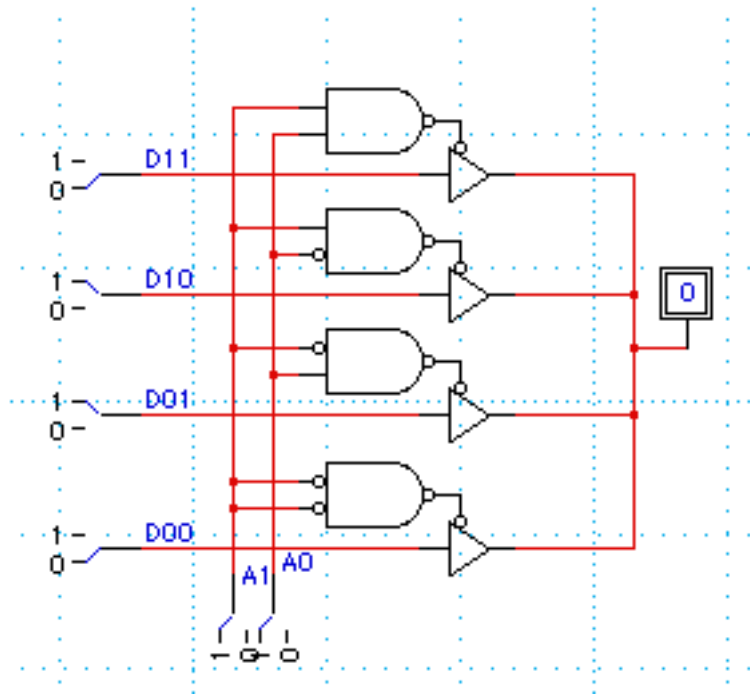
Don't worry how it's built for now.

What use is a device like this?

Well, if we have a point in a circuit where we know that exactly one of a number of outputs will be high, as in a MUX:



We couldn't just tie the outputs of the ANDs together because we can't feed back the output of some gates through the output lines of others. We had to feed them through an OR gate to the output.



Since only one of the tristate buffers will be connected, we can safely do what looks like a “wired or”.

Driving the Bus

We can use this idea to allow one of multiple inputs to be placed onto a wire for transmission somewhere else.

Such a wire is called a *bus*.

A bus must have exactly 0 or 1 “drivers”.

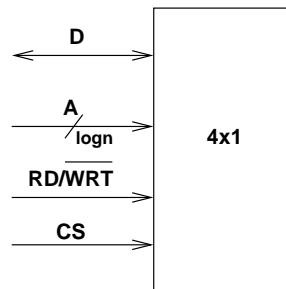
This will be useful when we want to connect up our CPUs to a bank of memory.

In that case, tristate buffers can be used to determine whether the bus is being used to send data from the CPU to memory, from the memory to the CPU, or neither. Both can’t be done at once.

If we activated multiple tristate buffers and had multiple bus drivers, we can make some toasters (sending the signal backwards through our gates).

Building Memory Devices

Suppose we wish to build a device that holds 4 bits of memory.

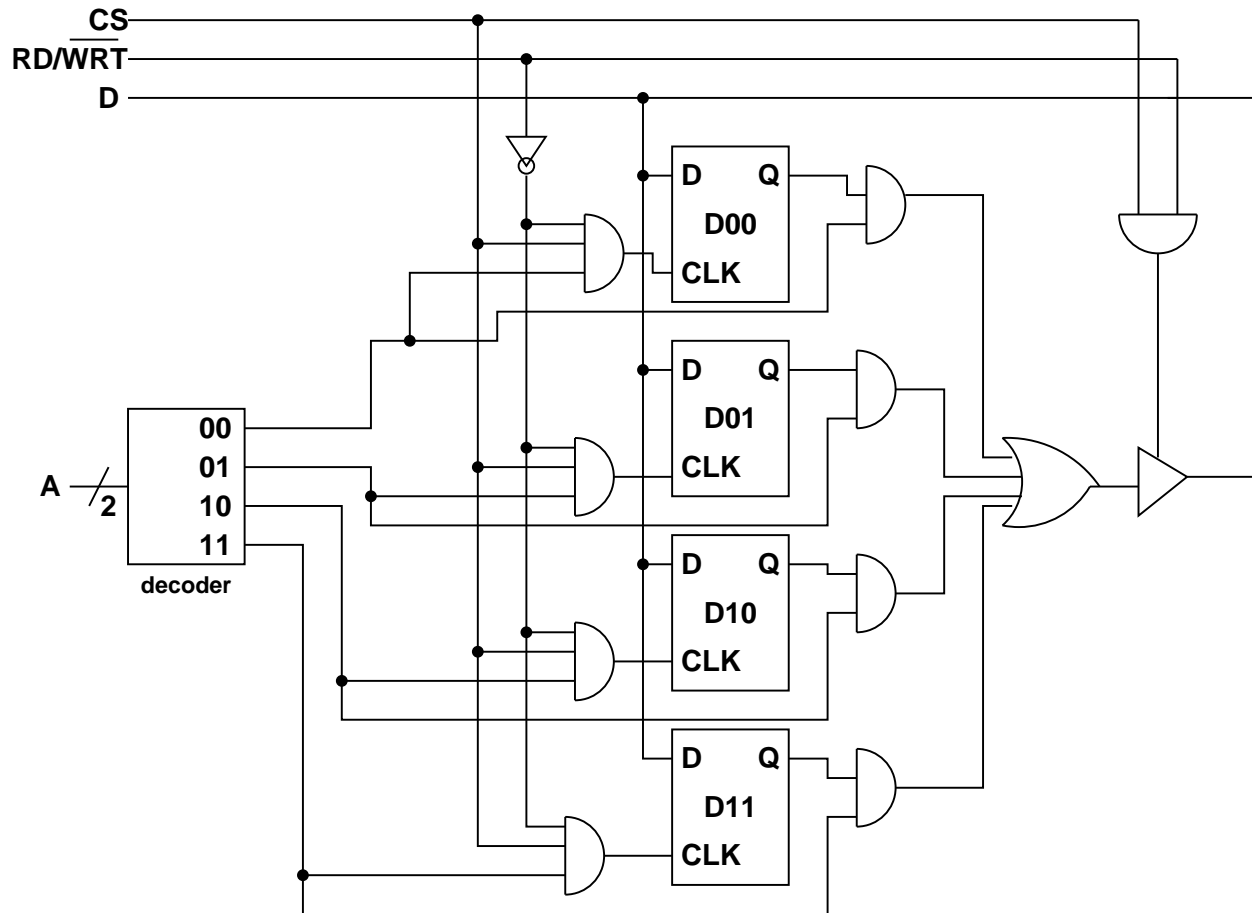


What does all this mean?

1. 4×1 : 4 is the number of units of memory, 1 is the *addressability*, the smallest number of bits we can address.
2. D is a bidirectional data bus:
 - when we want to store a value in the memory, we put that value on D
 - when we want to retrieve a value from memory, its value is put on D
 - D consists of a number of wires equal to the addressability of the memory
3. A is the address lines
 - need to be able to select from among all n units of memory
 - so we need $\log_2(n)$ address lines
 - specifies which bits are being read from or written to the data bus
4. $\overline{RD/WRT}$ selects whether the chip should be reading or writing – “who’s driving the bus?”
5. CS is chip select. Are we doing anything with this chip right now?

How can we build this out of components we have seen?

We use 4 D-type flip-flops and connect them up to our inputs, as appropriate.



How it works:

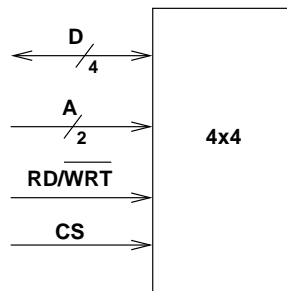
1. Correct address line is set high by the decoder. This will disable 3 of the 4 *CLK* inputs and 3 of the 4 AND gates that mask *Q* outputs of the flip-flops.
2. Data line is fed as input, unconditionally, to all flip-flops.
3. When *CS* (“chip select”) is low, all 4 input AND gates are low, inhibiting any *CLK* input. Also, and the AND gate at the output is low, disconnecting the tristate buffer.
4. When RD/\overline{WRT} is high, an input to all 4 input AND gates is low, so *CLK* is inhibited.
5. When RD/\overline{WRT} is low, the AND gate controlling the tristate buffer is low, meaning the output is not being sent to *D*.
6. When both *CS* and RD/\overline{WRT} are high, the output being sampled from one of the D flip-flops is being passed through to *D*. One of the tri-state buffers is “driving the bus”.
7. When *CS* is high and RD/\overline{WRT} is low, we assume someone else is “driving the bus” and is putting the values on the bus that we wish to store in one of our flip-flops.

We have a 4-bit memory!

4×4 Memory

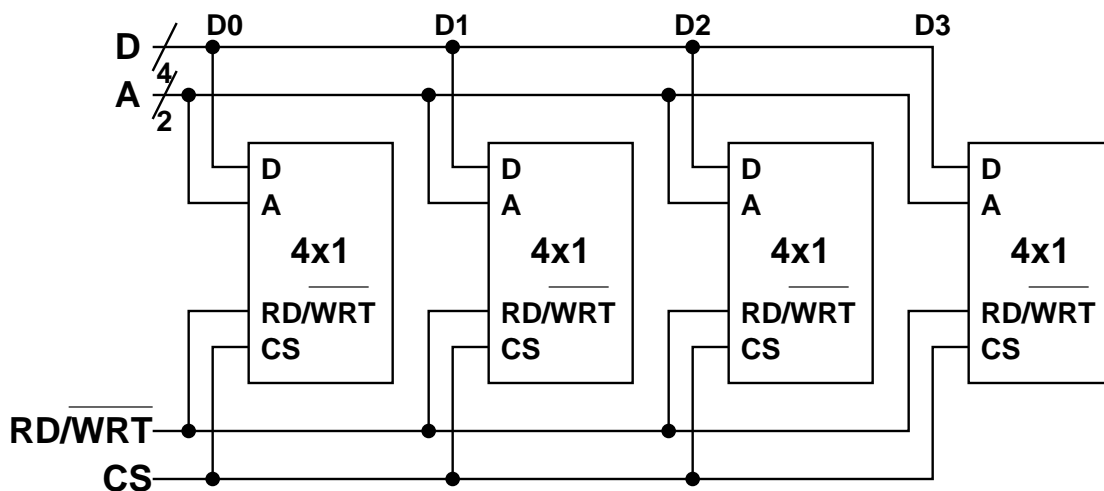
We normally don't think about storing our memory just one bit at a time.

How about building a device that can store 4 4-bit values?



Here, we have the same 2 address lines, a $\overline{RD/WRT}$ line and a chip select, but instead of a single data line, we have 4 data lines.

We will build our 4×4 device out of 4 4×1 devices:



Note that the data bus D is actually 4 wires, and we only connect one of the 4 to each of the 4×1's.

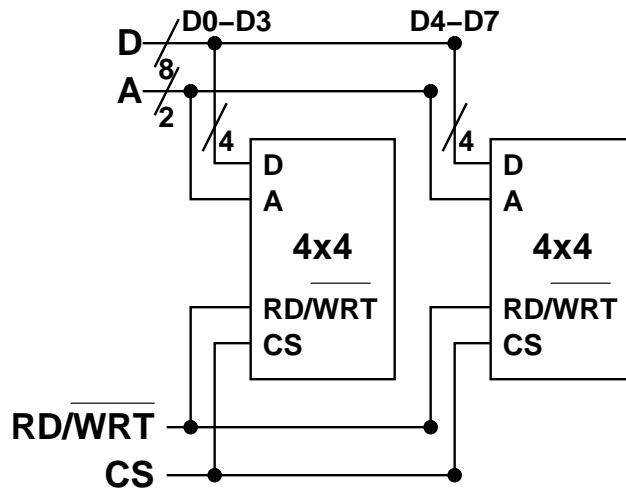
4×8 Memory

So far, we've gone from the D-type flip flop, (which we can think of as a 1×1 memory device) to a 4×1 by adding 2 address lines, and using a decoder to have those address lines activate one of 4 1×1 devices.

Then we went from 4×1 to 4×4 by having 4 data lines, each of which goes to one of the 4×1 devices.

We could do the same thing for a 4×8 device – 8 data lines, each of which is fed into a separate 4×1 device.

But if we already have 4×4 devices, we can expand to 4×8 :

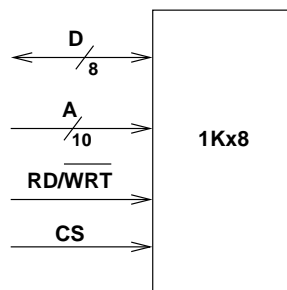


That's 4 bytes of memory.

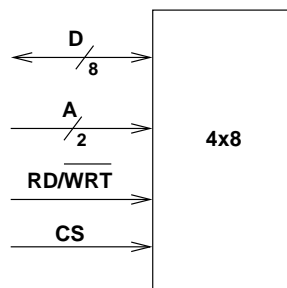
1KB memory

Now let's consider how we might build a kilobyte of memory out of our 4×8 devices.

The device we're trying to build will look like this:



We'll need 256 4×8 devices, each of which looks like this:



We refer to each of these 4×8 devices as a *bank* of memory.

Of our 10 address lines, 8 are used to select among the 256 banks and the other 2 select among the 4 bytes on the bank chosen.

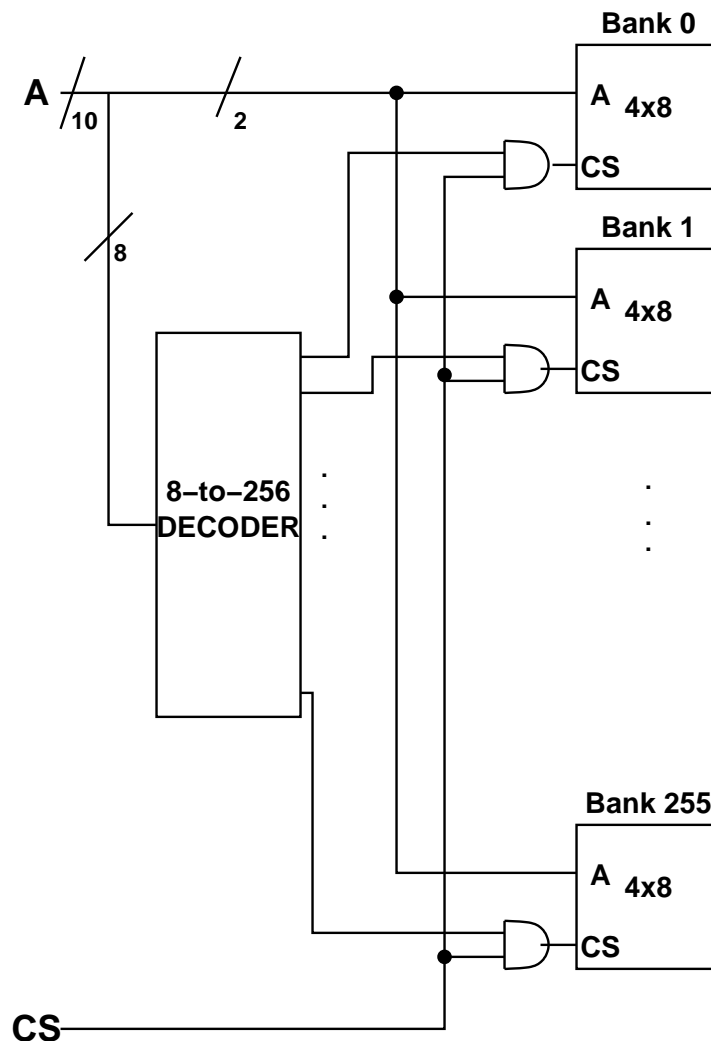
If we think of our address lines as bits $A_9 A_8 \dots A_0$, two main possibilities come to mind for how to organize things:

- Option 1: A_1 and A_0 select the byte within a bank and $A_9 \dots A_2$ select the bank.
- Option 2: A_9 and A_8 select the byte within a bank and $A_7 \dots A_0$ select the bank.

In either case, we want all 8 data lines wired to each bank, we want the RD/\overline{WRT} wired to each bank.

Our two lines to select a byte within a bank are wired to the address lines of the bank.

The other 8 address lines are passed through a decoder, and the decoder outputs are ANDed with the CS inputs of the whole circuit then wired to the CS inputs of each bank:



Which bytes are stored on which chips with the two layout options?

Bank	Option 1	Option 2
0	0–3	0, 256, 512, 768
1	4–7	1, 257, 513, 769
2	8–11	2, 258, 514, 770
...
254	1016–1019	254, 510, 766, 1022
255	1020–1023	255, 511, 767, 1023

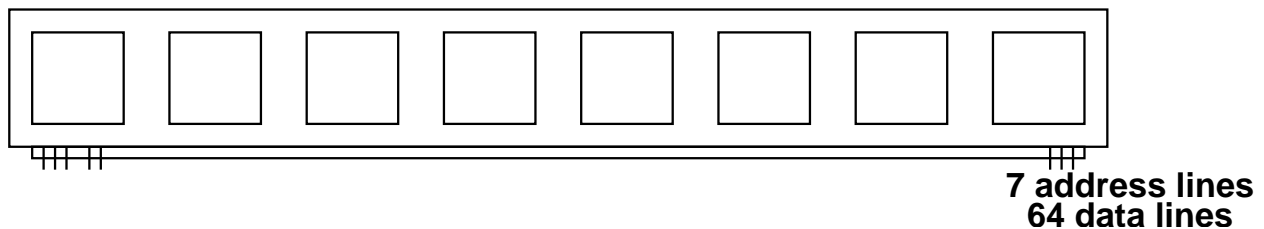
Each possible configuration has advantages and disadvantages.

Option 1 means we can lose a chip and still have large chunks of contiguous memory available.

Option 2 has advantages for chip layout.

SIMM Layout

You have probably seen and maybe used memory chips of the SIMM (or, *single in-line memory module*) type.



Suppose we have 1 KB of memory of 8-bit bytes.

This requires 10 bits for full addressing, 8 data lines.

But this is not normally how it would be set up.

More likely, you would find 64 data lines and only 7 address lines.

What does this mean?

It's really an 8-byte addressable memory. We load/store memory in chunks of 8 bytes at a time, so there are only 128 addressable chunks.

When memory is requested, say address 1:

$$A = A_9 A_8 \dots A_0 = 0000000001$$

The 7 high bits of the address $A_9 \dots A_3 = 0000000$ are sent to the SIMM on the address lines.

We get back bytes 0–7, even though we only wanted 1. It's up to the CPU, which still has the full address (including $A_2 A_1 A_0$) to pick out the byte it's interested in.

To access location $517 = 1000000101$, we request 8-byte chunk 64, and take byte 5 of those that come back.

This memory is organized using “Option 2” from our discussion about how to arrange memory among multiple banks.

This may at first seem wasteful, but the memory chip can get all 8 bytes easily and more wires in and out means we can transfer more memory more quickly.

Plus.. there’s a good chance that any memory access will be followed by additional memory access to nearby locations (think – local variables, an array, sequential program text).

This *locality* is a natural feature of most programs.

All machines today have memory that is addressable in some chunk larger than one byte.

The decisions about how we break this down have ripple effects throughout the architecture. We will soon see this in much detail.

Error Detection and Correction

You have probably heard about error correcting memory.

If we want to do this, we need to build in some redundancy.

We can detect a single-bit memory error by adding a *parity* bit.

For example, if we have an 8-bit data value and we want to maintain even parity, we add a 9th bit that makes the total number of bits that are set an even number:

byte	even parity bit
00000000	0
00000001	1
01110100	0
11000111	1
11111111	0

From this, we can check, each time we retrieve a byte, that it has even parity. If not, we know that something is wrong.

But that’s all it tells us. Since we don’t know which of the 9 bits are wrong, we can’t fix it.

Some of you may have had computers that crash with a Blue Screen of Death saying that a memory parity error was detected.

This is not just a Windows thing - I have had Unix systems crash with a kernel error that a memory parity error was detected.

To fix it, we need more extra bits.

Here’s one error correction scheme that can fix a single bit error but at the expense of 4 extra bits for each byte of memory (50% overhead).

We have 12 bits used to represent the 8-bit value. We number them 12-1 and use the ones whose numbers are powers of 2 as parity bits:

12	11	10	9	8	7	6	5	4	3	2	1
1100	1011	1010	1001	<u>1000</u>	0111	0110	0101	<u>0100</u>	0011	<u>0010</u>	<u>0001</u>

We use the parity bits as follows:

1. Position 1 stores the even parity of odd-numbered bits
2. Position 2 stores the even parity of bits whose number has the 2's bit set
3. Position 4 stores the even parity of bits whose number has the 4's bit set
4. Position 8 stores the even parity of bits whose number has the 8's bit set

So to store the value $94 = 01011110$ we first fill in the data bits:

0	1	0	1	<u> </u>	1	1	1	<u> </u>	0	<u> </u>	<u> </u>
1100	1011	1010	1001	<u>1000</u>	0111	0110	0101	<u>0100</u>	0011	<u>0010</u>	<u>0001</u>

Position 1 stores the even parity of the bits at 3, 5, 7, 9, 11. 4 of those are set to 1, so we set that bit to 0.

Position 2 stores the even parity of the bits at 3, 6, 7, 10, 11. 3 of those are set to 1, so we set that bit to 1.

Position 4 stores the even parity of the bits at 5, 6, 7, 12. 3 of those are set to 1, so we set that bit to 1.

Position 8 stores the even parity of the bits at 9, 10, 11, 12. 2 of these are set to 1, so we set that bit to 0.

0	1	0	1	<u>0</u>	1	1	1	<u>1</u>	0	<u>1</u>	<u>0</u>
1100	1011	1010	1001	<u>1000</u>	0111	0110	0101	<u>0100</u>	0011	<u>0010</u>	<u>0001</u>

When we retrieve a value from memory, we can make sure it's OK by computing the 4 parity bits and comparing to the stored parity bits.

If they all match, we're OK.

If there's any mismatch, we know there's an error.

Let's introduce an error into our stored value. We'll change the third bit to a 1.

0	1	1	1	<u>0</u>	1	1	1	<u>1</u>	0	<u>1</u>	<u>0</u>
1100	1011	1010	1001	<u>1000</u>	0111	0110	0101	<u>0100</u>	0011	<u>0010</u>	<u>0001</u>

So we recompute the parity bits on the value we retrieved, and compare to the stored bits:

bit	computed	stored	match
P_1	0	0	0
P_2	0	1	1
P_4	1	1	0
P_8	1	0	1

We can quickly detect the mismatches in P_2 and P_8 (hey! XOR!)

This means that the bit at position 1010 has an error and must be flipped. Convenient!

Think about how this might be implemented

- the memory itself doesn't even need to know
- we can drop in some XORs to generate parity bits to be stored in memory
- we XOR again to regenerate parity bits for retrieved values
- still more XOR to do correction

This works even if the parity bit is the one that has an error. It just ends up "fixing" the parity bit.

It does not work for 2-bit errors.