SIENA*college*
Computer Science

Computer Science 220
Assembly Language & Comp. Architecture
Siena College
Fall 2011

Topic Notes: Building Memory

We'll next see how we can use flip-flop devices to construct memory.

Buffers

We've seen and used inverters.

Recall that we can put two inverters in series to construct a *buffer*.

We represent a buffer as a triangular symbol (like an inverter) but with no circle at the end.

Its effect is to slow down and boost the signal strength of the current in the circuit.

Recall that large fan-out when wiring physical circuits can be problematic if there is not enough current to drive subsequent gates. A buffer can take care of that.

We also may need to delay the propagation of a value through a circuit to deliver it to the desired place at the same time as some other value. Buffers can come in handy for this.

Tristate Buffers

So far, we have thought almost exclusively about each wire or gate output as having a value 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.

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

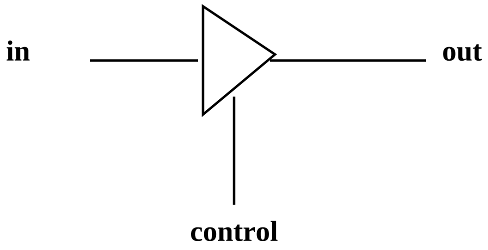
You can think of the wire as transmitting some value. A 0 value means the wire like a pipe spilling out 0's, a 1 value is like a pipe spilling out 1's.

Or think of it 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. Sometimes, this is just what we want.

To accomplish this, we next look at a device called a *tristate buffer*.

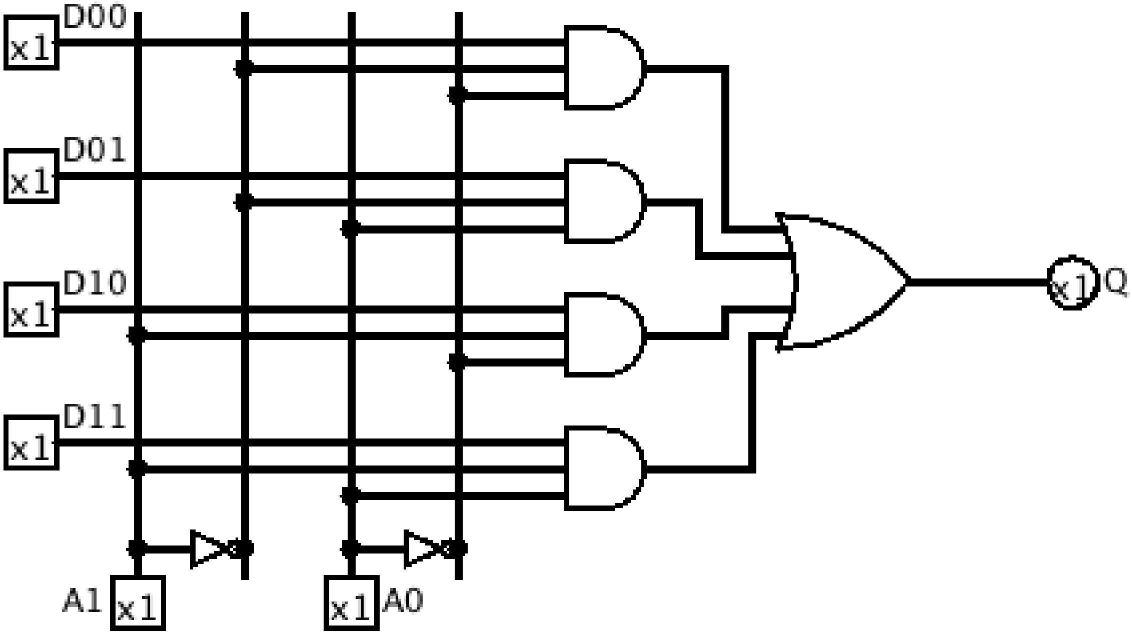


When the control line is high, the device connects the output to the input (with some amplifying and delay properties), like a buffer.

When control is low, it behaves like a broken wire – physically disconnected. This is often referred to as a *high impedence* state.

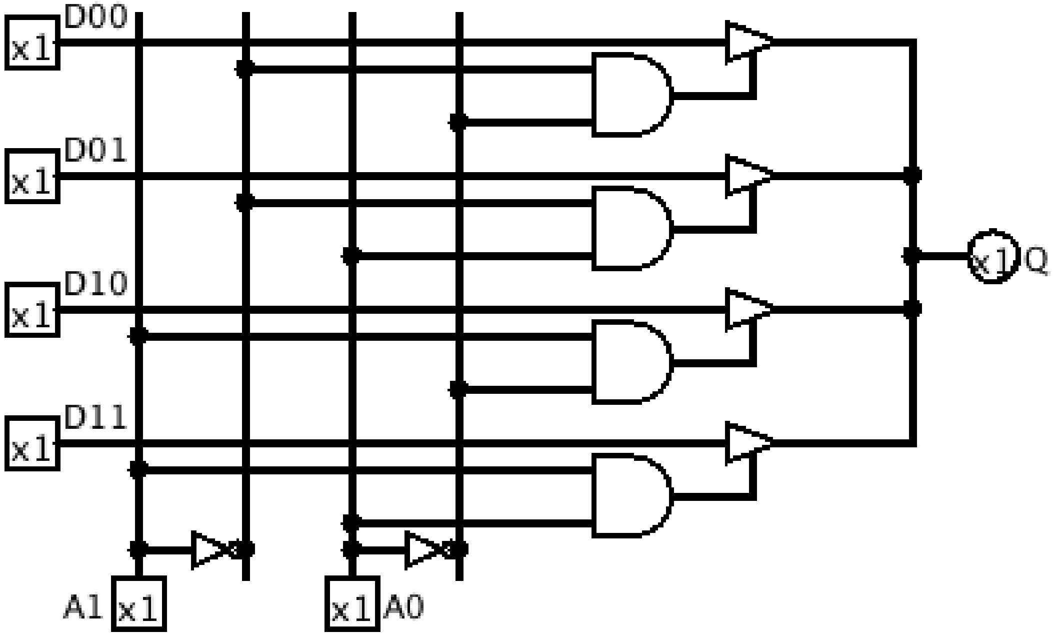
We will not be concerned about how to construct such a device – they're available to us as building blocks.

As an example of where this device can be useful, suppose 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:



Logisim Circuit: `~jteresco/shared/cs220/examples/logisim/4to1mux.circ`

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.



Logisim Circuit:
`~jteresco/shared/cs220/examples/logisim/4to1muxtristate.circ`

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

Note: the Logisim device is the *controlled buffer* (or, if we want to invert as part of the device, the *controlled inverter*).

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.

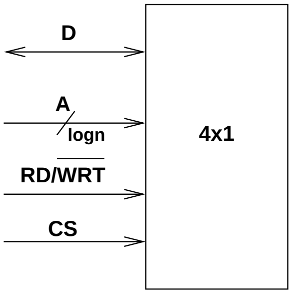
Activating multiple tristate buffers resulting in multiple bus drivers can cause problems. At best, we will have conflicting signals and get incorrect behavior. At worst, we would sendi signals backwards through our gates, resulting in potential damage to the components.

Building Memory Devices

We often measure the sizes of memory using terms like "kilobyte", "megabyte", "and gigabyte".

Our goal is to see how to construct memory devices at that scale.

But...we'll start much smaller. We first build a device that can store 4 bits of memory:

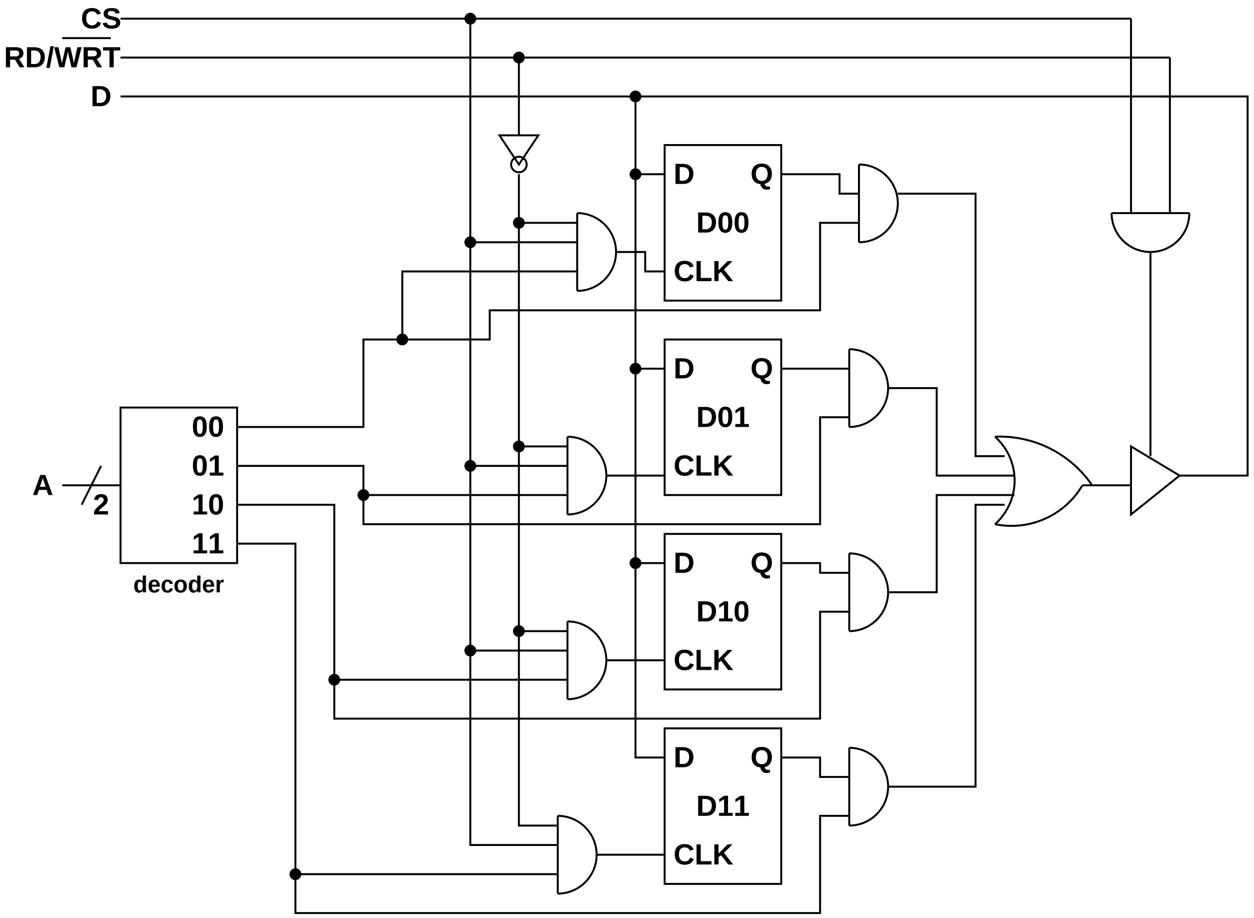


Let's look at what each input and output means here.

1. 4×1: 4 is the number of units of memory, 1 is the *addressability*, the smallest number of bits we can address. "4 memory locations of 1 bit each."
2. D is a bidirectional data bus:
   - when we want to store a value in the memory, we put that value on D as input to the memory circuit
   - when we want to retrieve a value from memory, its value is put on D as output from the memory circuit
   - 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
   - this requires $\log_2(n)$ address lines
   - the value on A specifies which bit(s) of the memory are storing a copy of what is on the data bus (a memory write) or are placing their value for output onto the data bus (a memory read)
4. $RD/\overline{WRT}$ 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? In practice, this is often an *active low* control, meaning the chip is selected when the input is 0, disabled when the input is 1. To keep things simple in our current discussion, we will assume an active high chip select.

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, the AND gate at the output is low, meaning the tristate buffer prevents any signal from the circuit from being written to the data bus.

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".

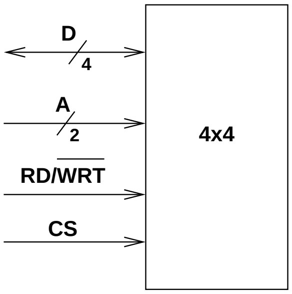
7. When CS is high and $RD/\overline{WRT}$ is low, we assume someone else is "driving the bus" and is putting the values on the bus that we wish to store in one of our flip-flops.

We have a 4-bit memory!

4×4 Memory

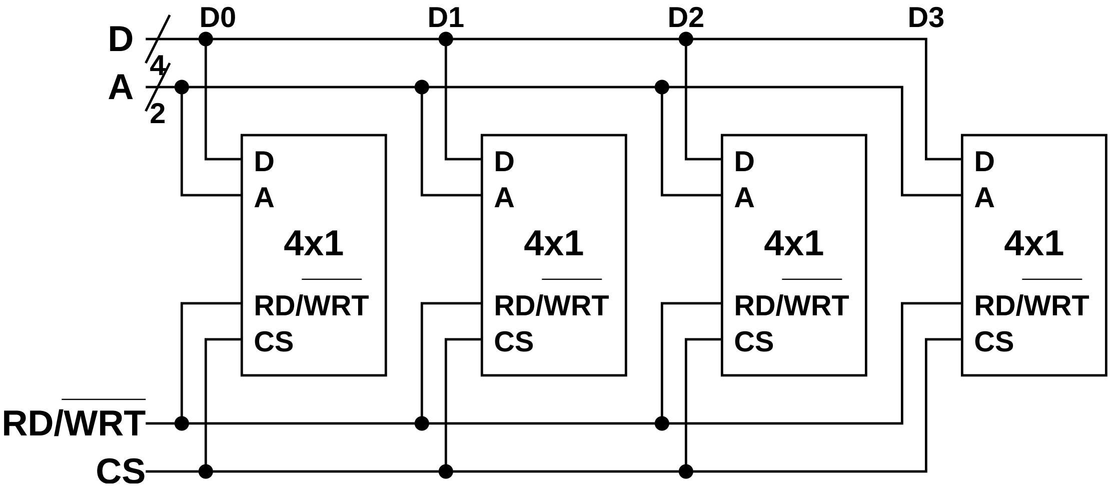
We normally don't think about storing values in memory just one bit at a time.

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



Here, we have the same 2 address lines, a $RD/\overline{WRT}$ 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.

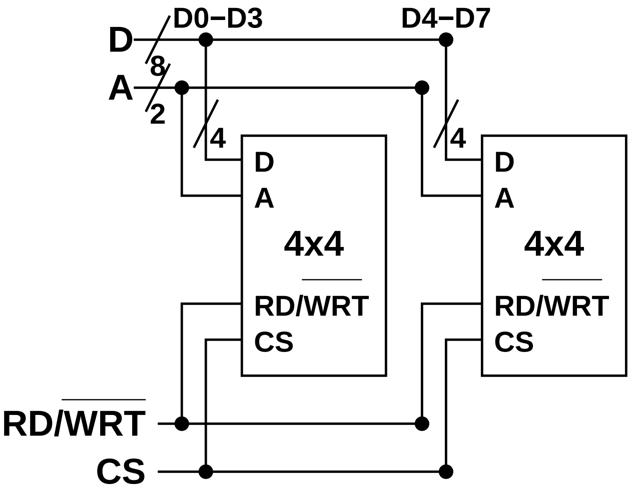
4×8 Memory

So far, we've gone from the D-type flip flip, (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:

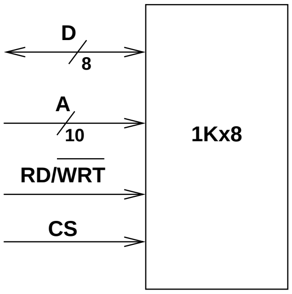

D
8
A
2
D0−D3
D4−D7
4
4
D
A
4x4
RD/$\overline{\text{WRT}}$
CS
D
A
4x4
RD/$\overline{\text{WRT}}$
CS
RD/$\overline{\text{WRT}}$
CS

That's 4 bytes of memory.

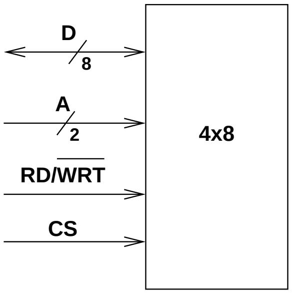
1KB memory

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.

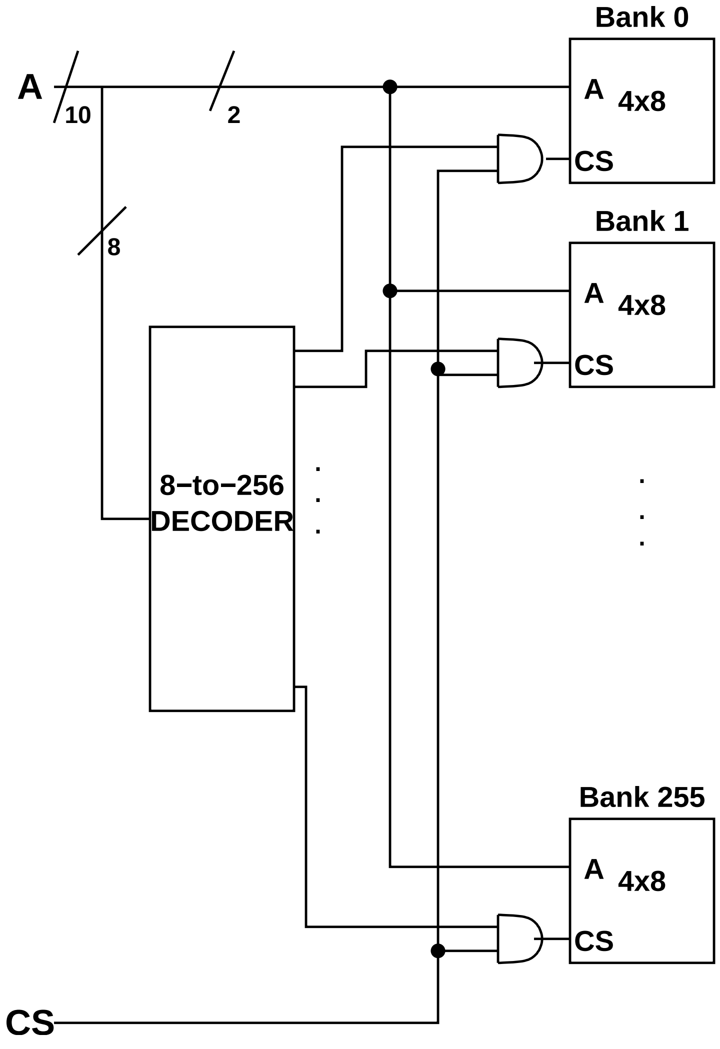
If we think of our address lines as bits $A_9A_8...A_0$, two main possibilities come to mind for how to organize things:

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

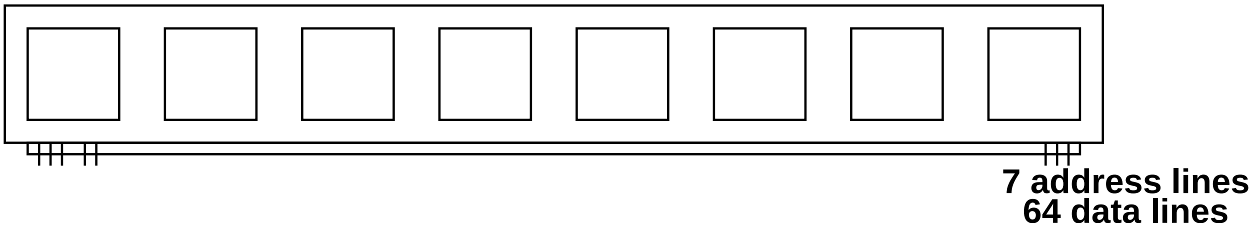
Each possible configuration has advangates and disadvantages.

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:

$A = A_9 A_8 ... A_0 = 0000000001$

The 7 high bits of the address $A_{9...3} = 0000000$ 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*. This is a memory circuit that can still give the correct answer even if a bit has mistakenly been changed. This could happen from a bad gate in a flip-flop, for example.

In order to do any kind of error correction, we need to build in some redundancy.

We can detect a single-bit memory error by adding a *parity* bit. The parity bit is set to tell whether the original value in memory has an odd or even number of bits set to 1. If we later read the value from memory and the parity bit no longer accurately reflects the odd/evenness of the number of bits set to 1, we know something has gone wrong.

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 is has even paity. 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 phenomenon - I have had Unix systems crash with a kernel error that a memory parity error was detected. The idea is that since we know there was a memory error but we don’t know how to fix it, it’s better to halt the system rather than risk producing incorrect results.

To both detect and fix an error, we will need to store more extra bits.

We consider here an 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 use 12 bits to represent an 8-bit value. We number the bits 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	1000	0111	0110	0101	0100	0011	0010	0001

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	__	1	1	1	__	0	__	__
1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001

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	0	1	1	1	1	0	1	0
1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001

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	0	1	1	1	1	0	1	0
1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001

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.