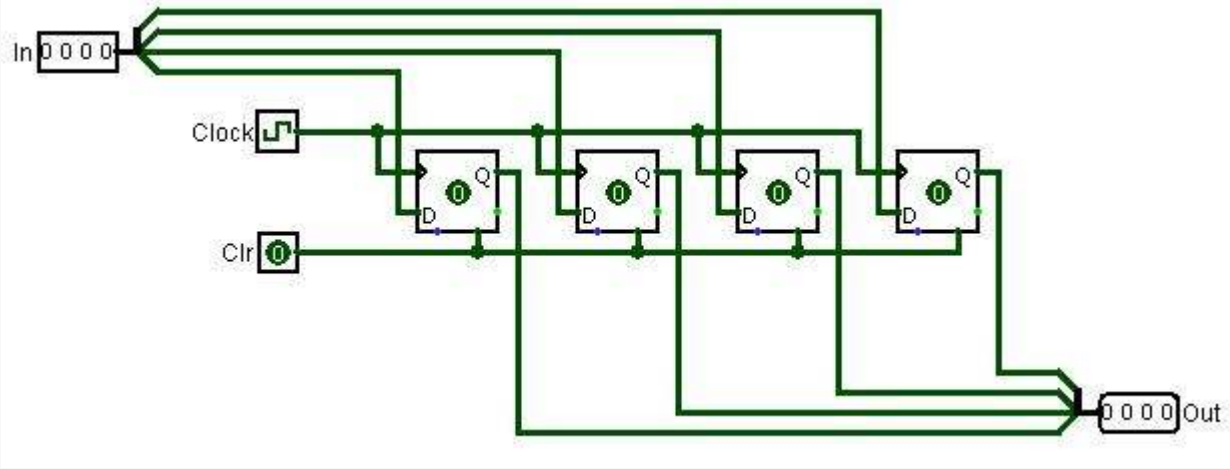
4-Bit Register

Built using D flip-flops:



Clock input controls when input is "written" to the individual flip-flops.

However, the design above isn't quite what we want…

QTP What's wrong with this?

How can we fix it?

Register File

A *register file* is a collection of *k* registers (a sequential logic block) that can be read and written by specifying a register number that determines which register is to be accessed.

The interface should minimally include:

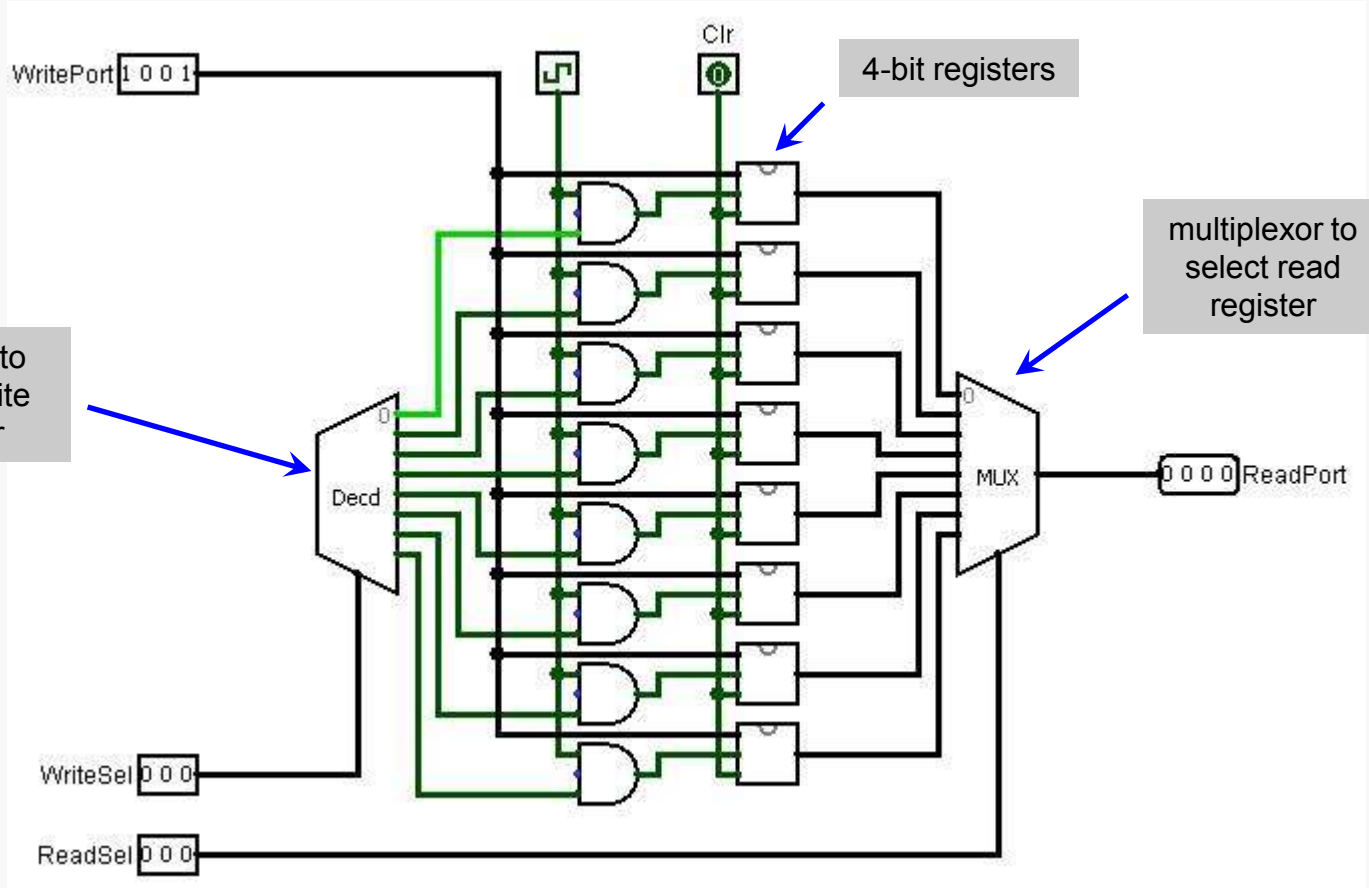
- an *n*-bit input to import data for writing (a *write port*)
- an *n*-bit output to export read data (a *read port*)
- a log(*k*)-bit input to specify the register number
- control bit(s) to enable/disable read/write operations
- a control bit to clear all the registers, asynchronously
- a clock signal

Some designs may provide multiple read or write ports, and additional features.

For MIPS, it is convenient to have two read ports and one write port. Why?

A File of 4-Bit Registers

Aggregating a collection of 4-bit registers, and providing the appropriate register selection and data input/output interface:



Random Access Memory

Random access memory (RAM) is an array of memory elements.

Static RAM (SRAM):

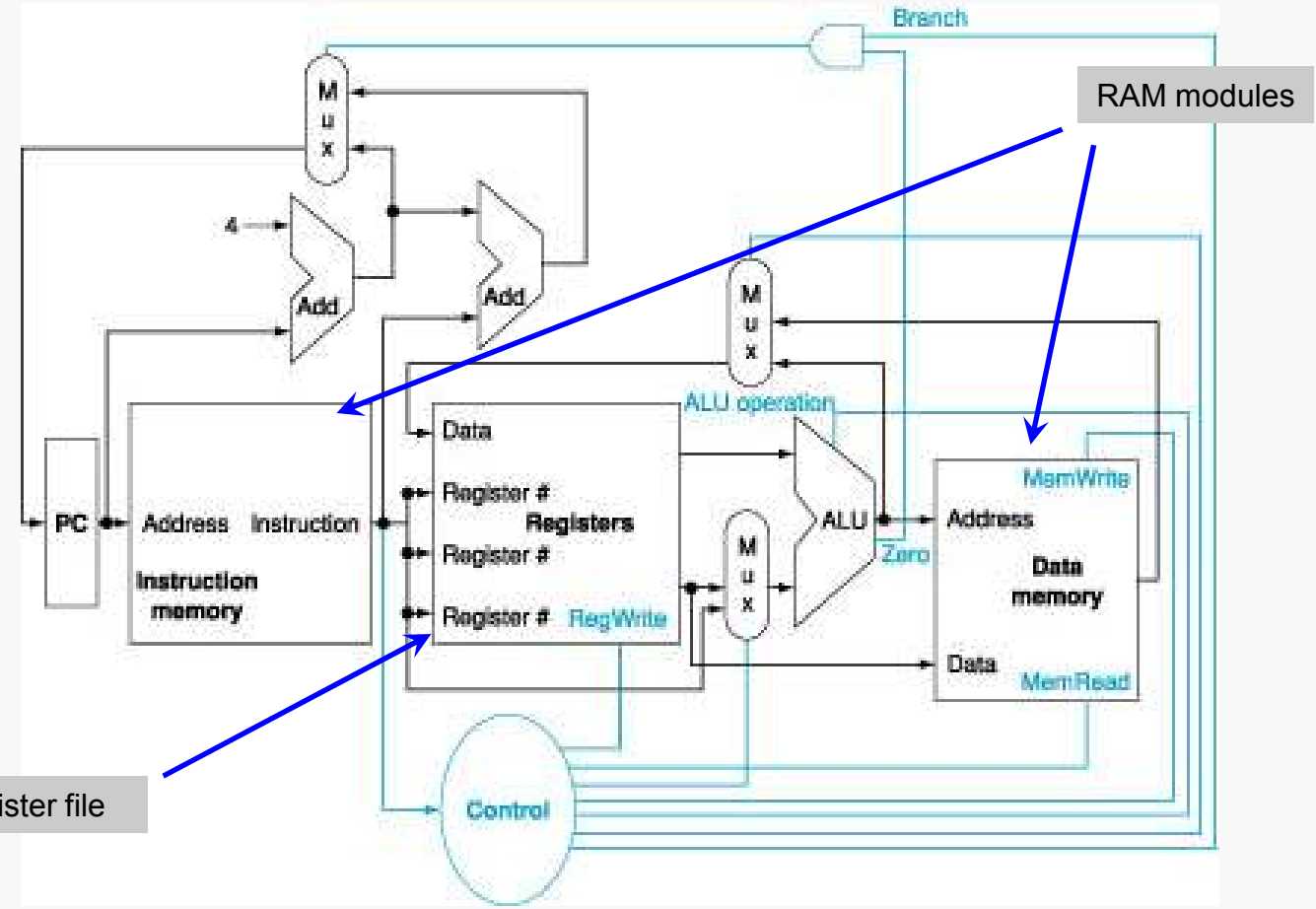
- bits are stored as flip-flops (typically 4 or more transistors per bit)
- hence "static" in the sense that the value is preserved as long as power is supplied
- somewhat complex circuit per bit, so not terribly dense on chip
- typically used for cache memory

Dynamic RAM (DRAM):

- bits are stored as a charge in a capacitor
- hence "dynamic" since periodic refreshes are needed to maintain stored values
- single transistor needed per data bit, so very dense in comparison to SRAM
- much cheaper per bit than SRAM
- much slower access time than SRAM
- typically used for main memory

Basic MIPS Implementation

Here's an updated view of the basic architecture needed to implement a subset of the MIPS environment:

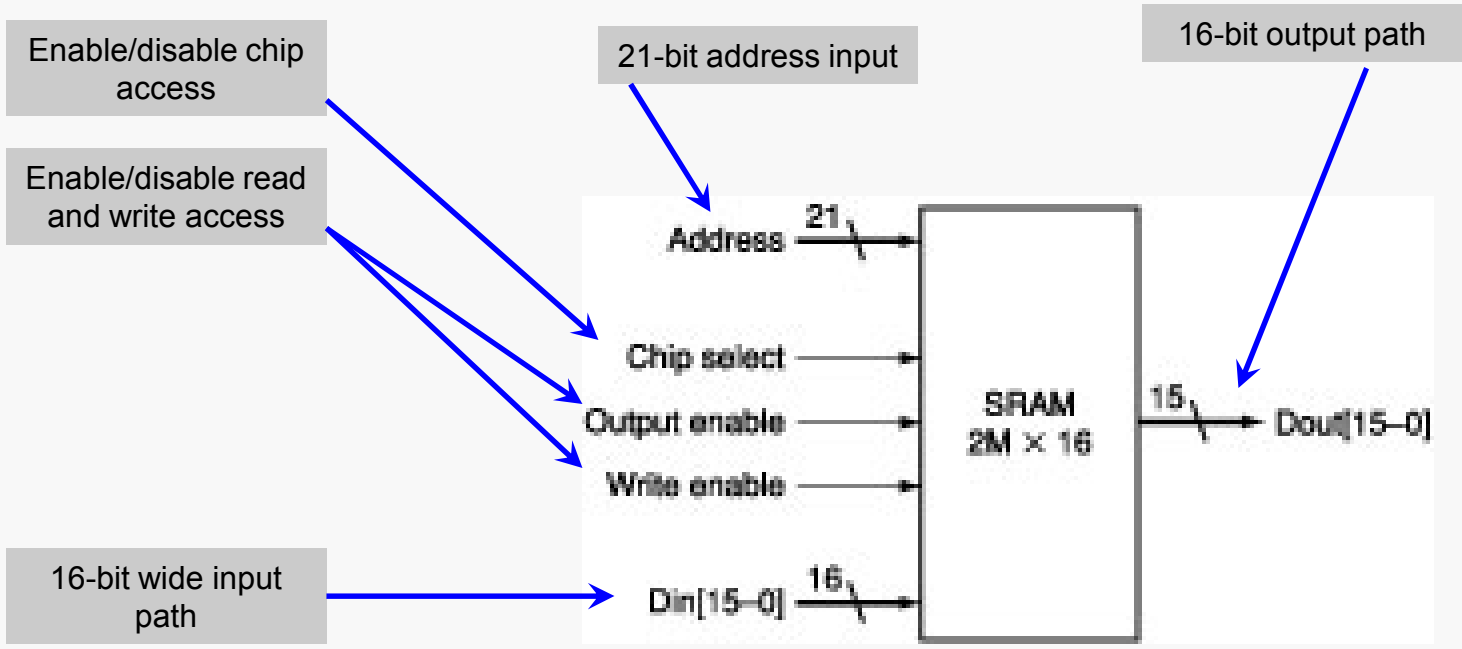


SRAMs

Configuration specified by the # of addressable locations (# of rows or height) and the # of bits stored in each location (width).

Consider a 4M x 8 SRAM:

- 4M locations, each storing 8 bits
- 22 address bits to specify the location for a read/write
- 8-bit data output line and 8-bit data input line



SRAM Performance

read access time

- the delay from the time the Output enable is true and the address lines are valid until the time the data is on the output lines.
- typical read access times might be from 2-4 ns to 8-20 ns, or considerably greater for low-power versions developed for consumer products.

write access time

- set-up and hold-time requirements for both the address and data lines
- write-enable signal is actually a pulse of some minimum width, rather than a clock edge
- write access time includes all of these

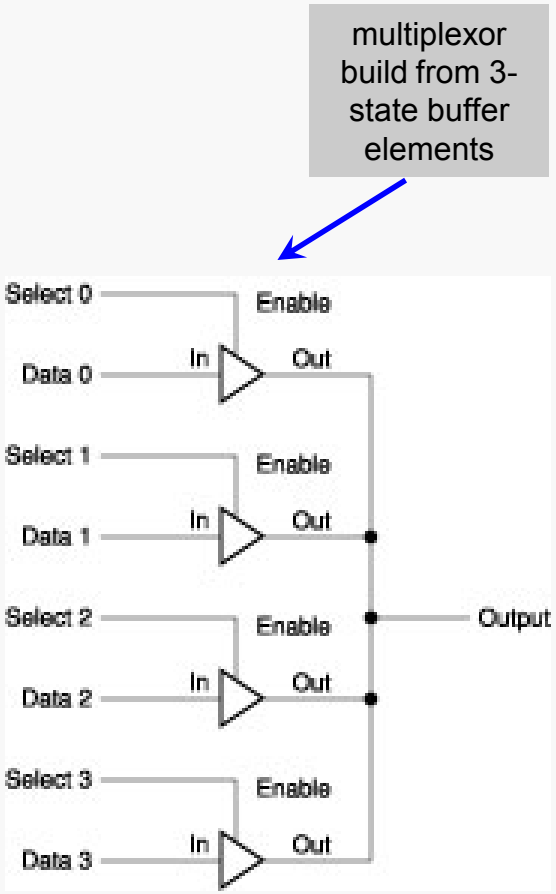
SRAM Implementation

Although the SRAM is conceptually similar to a register file:

- impractical to use same design due to the unreasonable size of the multiplexors that would be needed
- design is based on a *three-state buffer*

If output enable is 1, then the buffer's output equals its input data signal.

If output enable is 0, then the buffer's output is in a high-impedance state that effectively disables its effect on the bit line to which it is connected.

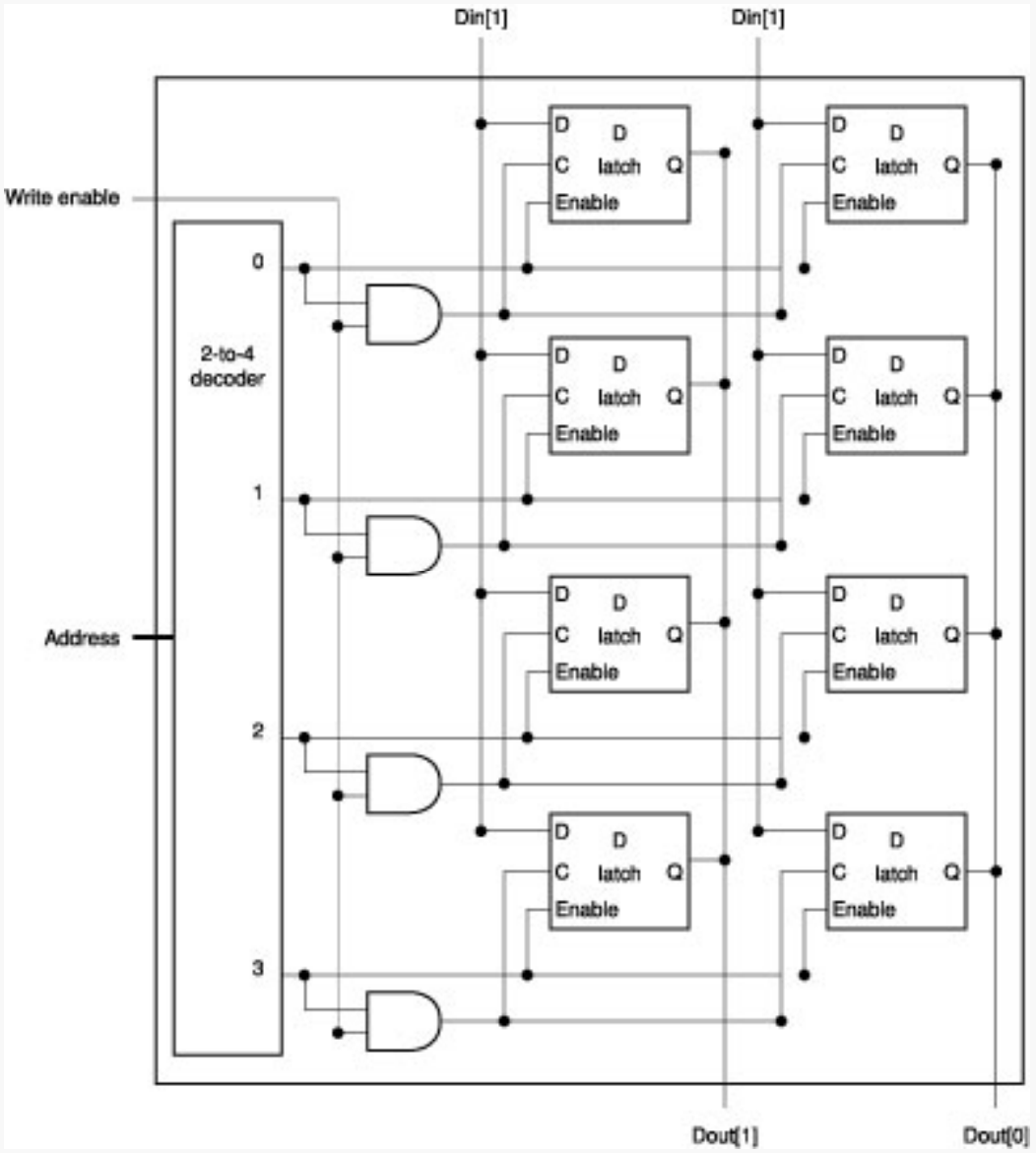


SRAM Implementation

At right is a conceptual representation of a 4x2 SRAM unit built from D latches that incorporate 3-state buffers.

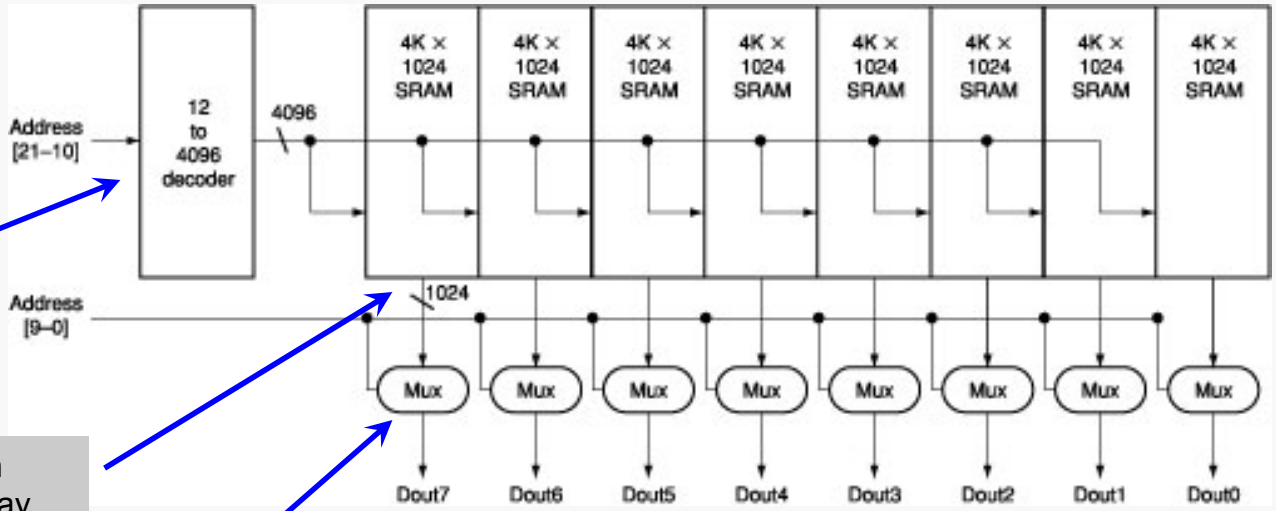
For simplicity, the chip select and output enable signals have been omitted.

Although this eliminates the need for a multiplexor, the decoder that IS required will become excessively large if we scale this up to a useful capacity.



SRAM Implementation

A 4Mx8 SRAM as an array of 4Kx1024 arrays:



decoder generates addresses for the 4096 rows of each of the 8 subarrays

each subarray outputs a row of 1024 bits

bank of 10-bit multiplexors select one bit from each of the subarrays

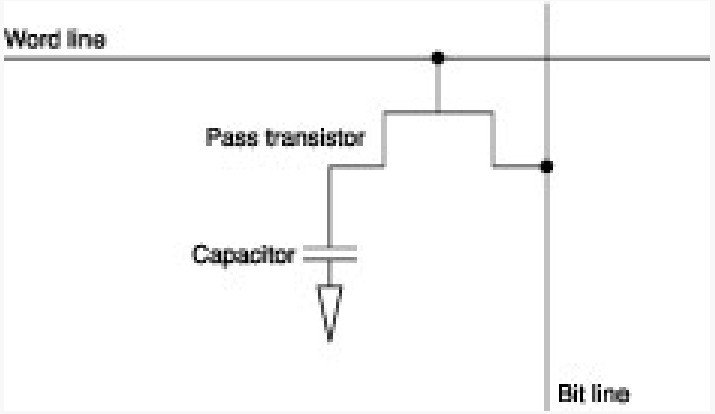
This requires neither a huge multiplexor nor a huge decoder.

A practical version might use a larger number of smaller subarrays. How would that affect the dimensions of the decoder and multiplexors that would be needed?

DRAM Implementation

Each bit is stored as a charge on a capacitor.

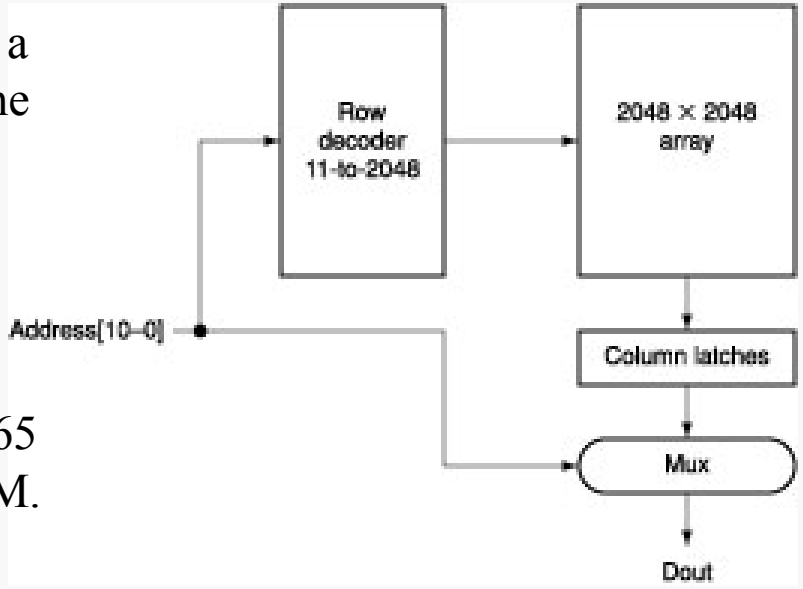
Periodic refreshes are necessary and typically require 1-2% of the cycles of a DRAM module.



Access uses a 2-level decoding scheme; a *row access* selects and transfers a row of values to a row of latches; a *column access* then selects the desired data from the latches.

Refreshing uses the column latches.

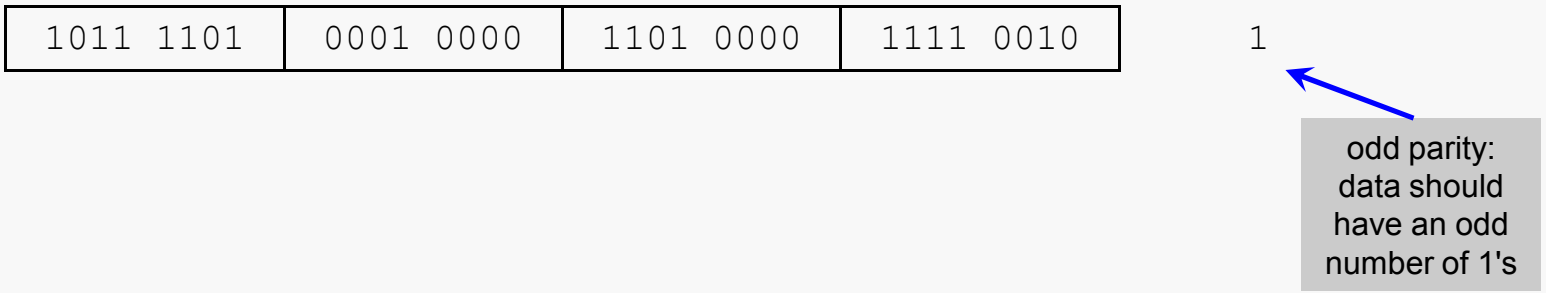
DRAM access times typically range from 45-65 ns, about 5-10 times slower than typical SRAM.



Error Detection

Error detecting codes enable the detection of errors in data, but do not determine the precise location of the error.

- store a few extra state bits per data word to indicate a necessary condition for the data to be correct
- if data state does not conform to the state bits, then something is wrong
- e.g., represent the correct *parity* (# of 1's) of the data word
- 1-bit parity codes fail if 2 bits are wrong…



A 1-bit parity code is a *distance-2 code*, in the sense that at least 2 bits must be changed (among the data and parity bits) produce an incorrect but legal pattern. In other words, any two legal patterns are separated by a distance of at least 2.

Error Correction

Error correcting codes provide sufficient information to locate and correct some data errors.

- must use more bits for state representation, e.g. 6 bits for every 32-bit data word
- may indicate the existence of errors if up to k bits are wrong
- may indicate how to correct the error if up to l bits are wrong, where $l < k$
- c code bits and n data bits → $2^c >= n + c + 1$

We must have at least a *distance-3* code to accomplish this.

Given such a code, if we have a data word + error code sequence X that has 1 incorrect bit, then there will be a unique valid data word + error code sequence Y that is a distance of 1 from X, and we can correct the error by replacing X with Y.

Error Correction

A *distance-3* code is also knows as a single-error correcting, double-error detecting or *SECDED* code.

If X has 2 incorrect bits, then we will replace X with an incorrect (but valid) sequence.

We cannot both detect 2-bit errors and correct 1-bit errors with a distance-3 code.

But, hopefully flipped bits will be a rare occurrence and so sequences with two or more flipped bits will have a negligible probability.

Hamming Codes

Richard Hamming described a method for generating minimum-length error-correcting codes. Here is the (7,4) Hamming code for 4-bit words:

Data bits	Check bits
0000	000
0001	011
0010	101
0011	110
0100	110
0101	101
0110	011
0111	000
1000	111
1001	100
1010	010
1011	001
1100	001
1101	010
1110	100
1111	111

Say we had the data word 0100 and check bits 011.

The two valid data words that match that check bit pattern would be 0001 and 0110.

The latter would correspond to a single-bit error in the data word, so we would choose that as the correction.

Note that if the error was in the check bits, we'd have to assume the data word was correct (or else we have an uncorrectable 2-bit error or worse). In that case, the check bits would have to be 1 bit distance from 110, which they are not.

Hamming Code Details

Hamming codes use extra parity bits, each reflecting the correct parity for a different subset of the bits of the code word. Parity bits are stored in positions corresponding to powers of 2 (positions 1, 2, 4, 8, etc.). The encoded data bits are stored in the remaining positions.

The parity bits are defined as follows:

- position 1: check 1 bit, skip 1 bit, check 1 bit, skip 1 bit, …
- position 2: check 2 bits, skip 2 bits, …

…

- position 2^k: check 2^k bits, skip 2^k bits, …

Consider the data byte: `10011010`

Expand to allow room for the parity bits: `_ _ 1 _ 001_1010`

Now compute the parity bits as defined above…

Hamming Code Details

We have the expanded sequence: _ _ 1 _ 0 0 1 _ 1 0 1 0

The parity bit in position 1 (first bit) would depend on the parity of the bits in positions 1, 3, 5, 7, etc:

_ _ **1** _ **0** 0 **1** _ **1** 0 **1** 0

Those bits have even parity, so we have: **0** _ 1 _ 0 0 1 _ 1 0 1 0

The parity bit in position 2 would depend on bits in positions 2, 3, 6, 7, etc:

0 _ **1** _ 0 **0** **1** _ 1 **0** **1** 0

Those bits have odd parity, so we have: **0** **1** 1 _ 0 0 1 _ 1 0 1 0

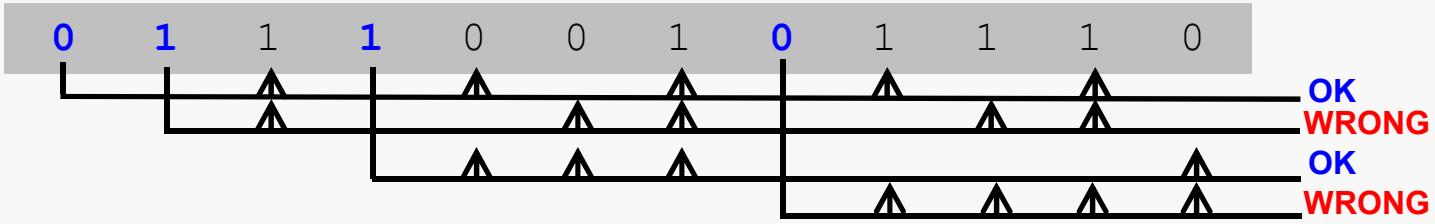
Continuing, we obtain the encoded string: **0** **1** 1 **1** 0 0 1 **0** 1 0 1 0

Hamming Code Correction

Suppose we receive the string: **0 1** 1 **1** 0 0 1 **0** 1 1 1 0

How can we determine whether it's correct? Check the parity bits and see which, if any are incorrect. If they are all correct, we must assume the string is correct. Of course, it might contain so many errors that we can't even detect their occurrence, but in that case we have a communication channel that's so noisy that we cannot use it reliably.

Checking the parity bits above:



So, what does that tell us, aside from that the string is incorrect? Well, if we assume there's no more than one incorrect bit, we can say that because the incorrect parity bits are in positions 2 and 8, the incorrect bit must be in position 10.