7.9 Memory elements: SRAMs and DRAMs

(Original section[1])

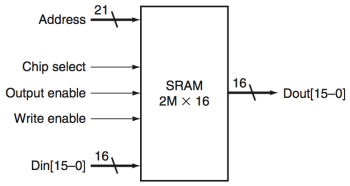
Registers and register files provide the basic building blocks for small memories, but larger amounts of memory are built using either *SRAMs* (*static random access memories*) or DRAMs (dynamic random access memories). We first discuss SRAMs, which are somewhat simpler, and then turn to DRAMs.

Static random access memory (***SRAM***): A memory where data is stored statically (as in flip-flops) rather than dynamically (as in DRAM). SRAMs are faster than DRAMs, but less dense and more expensive per bit.

SRAMs

SRAMs are simply integrated circuits that are memory arrays with (usually) a single access port that can provide either a read or a write. SRAMs have a fixed access time to any datum, though the read and write access characteristics often differ. An SRAM chip has a specific configuration in terms of the number of addressable locations, as well as the width of each addressable location. For example, a 4M × 8 SRAM provides 4M entries, each of which is 8 bits wide. Thus it will have 22 address lines (since $4M = 2^{22}$), an 8-bit data output line, and an 8-bit single data input line. As with ROMs, the number of addressable locations is often called the *height*, with the number of bits per unit called the *width*. For a variety of technical reasons, the newest and fastest SRAMs are typically available in narrow configurations: × 1 and × 4. The figure below shows the input and output signals for a 2M × 16 SRAM.

Figure 7.9.1: A 2M X 16 SRAM showing the 21 address lines (2M = 2^21) and 16 data inputs, the 3 control lines, and the 16 data outputs (COD Figure A.9.1).



To initiate a read or write access, the Chip select signal must be made active. For reads, we must also activate the Output enable signal that controls whether or not the datum selected by the address is actually driven on the pins. The Output enable is useful for connecting multiple memories to a single-output bus and using Output enable to determine which memory drives the bus. The SRAM read access time is usually specified as the delay from the time that Output enable is true and the address lines are valid until the time that the data are on the output lines. Typical read access times for SRAMs in 2004 varied from about 2—4 ns for the fastest CMOS parts, which tend to be somewhat smaller and narrower, to 8—20 ns for the typical largest parts, which in 2004 had more than 32 million bits of data. The demand for low-power SRAMs for consumer products and digital appliances has grown greatly in the past 5 years; these SRAMs have much lower stand-by and access power, but usually are 5—10 times slower. Most recently, synchronous SRAMs—similar to the synchronous DRAMs, which we discuss in the next section—have also been developed.

For writes, we must supply the data to be written and the address, as well as signals to cause the write to occur. When both the Write enable and Chip select are true, the data on the data input lines are written into the cell specified by the address. There are setup-time and hold-time requirements for the address and data lines, just as there were for D flip-flops and latches. In addition, the Write enable signal is not a clock edge but a pulse with a minimum width requirement. The time to complete a write is specified by the combination of the setup times, the hold times, and the Write enable pulse width.

Large SRAMs cannot be built in the same way we build a register file because, unlike a register file where a 32-to-1 multiplexor might be practical, the 64K-to-1 multiplexor that would be needed for a 64K × 1 SRAM is totally impractical. Rather than use a giant multiplexor, large memories are implemented with a shared output line, called a *bit line*, which multiple memory cells in the memory array can assert. To allow multiple sources to drive a single line, a *three-state-buffer* (or *tristate buffer*) is used. A three-state buffer has two inputs—a data signal and an Output enable—and a single output, which is in one of three states: asserted, deasserted, or high impedance. The output of a tristate buffer is equal to the data input signal, either asserted or deasserted, if the Output enable is asserted, and is otherwise in a *high-impedance state* that allows another three-state buffer whose Output enable is asserted to determine the value of a shared output.

The figure below shows a set of three-state buffers wired to form a multiplexor with a decoded input. It is critical that the Output enable at most one of the three-state buffers be asserted; otherwise, the three-state buffers may try to set the output line differently. By using three-state buffers in the individual cells of the SRAM, each cell that corresponds to a particular output can share the same output line. The use of a set of distributed three-state buffers is a more efficient implementation than a large centralized multiplexor. The three-state buffers are incorporated into the flip-flops that form the basic cells of the SRAM. COD Figure A.9.3 (The basic structure of a 4 X 2 SRAM ...) shows how a small 4 × 2 SRAM might be built, using D latches with an input called Enable that controls the three-state output.

Figure 7.9.2: Four three-state buffers are used to form a multiplexor (COD Figure A.9.2).

Only one of the four Select inputs can be asserted. A three-state buffer with a deasserted Output enable has a high-impedance output that allows a three-state buffer whose Output enable is asserted to drive the shared output line.

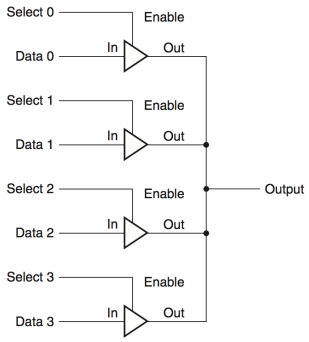
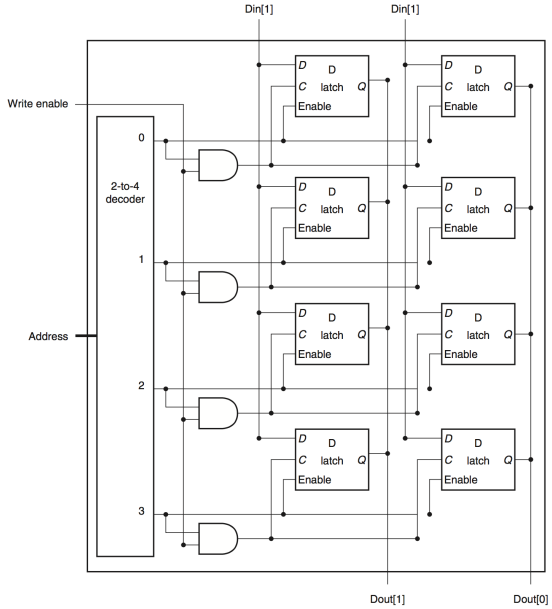


Figure 7.9.3: The basic structure of a 4 X 2 SRAM consists of a decoder that selects which pair of cells to activate (COD Figure A.9.3).

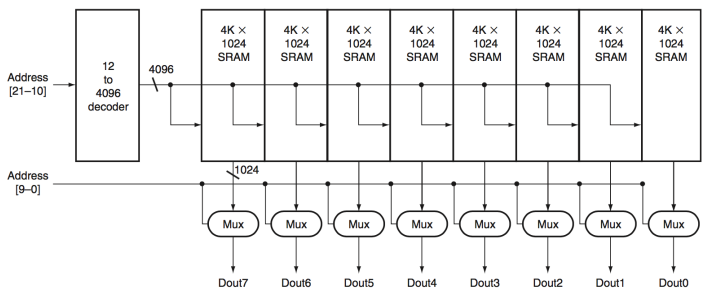
The activated cells use a three-state output connected to the vertical bit lines that supply the requested data. The address that selects the cell is sent on one of a set of horizontal address lines, called word lines. For simplicity, the Output enable and Chip select signals have been omitted, but they could easily be added with a few AND gates.



The design in the figure above eliminates the need for an enormous multiplexor; however, it still requires a very large decoder and a correspondingly large number of word lines. For example, in a 4M × 8 SRAM, we would need a 22-to-4M decoder and 4M word lines (which are the lines used to enable the individual flip-flops)! To circumvent this problem, large memories are organized as rectangular arrays and use a two-step decoding process. The figure below shows how a 4M × 8 SRAM might be organized internally using a two-step decode. As we will see, the two-level decoding process is quite important in understanding how DRAMs operate.

Figure 7.9.4: Typical organization of a 4M X 8 SRAM as an array of 4K X 1024 arrays (COD Figure A.9.4).

The first decoder generates the addresses for eight 4K X 1024 arrays; then a set of multiplexors is used to select 1 bit from each 1024-bit-wide array. This is a much easier design than a single-level decode that would need either an enormous decoder or a gigantic multiplexor. In practice, a modern SRAM of this size would probably use an even larger number of blocks, each somewhat smaller.



Recently we have seen the development of both synchronous SRAMs (SSRAMs) and synchronous DRAMs (SDRAMs). The key capability provided by synchronous RAMs is the ability to transfer a *burst* of data from a series of sequential addresses within an array or row. The burst is defined by a starting address, supplied in the usual fashion, and a burst length. The speed advantage of synchronous RAMs comes from the ability to transfer the bits in the burst without having to specify additional address bits. Instead, a clock is used to transfer the successive bits in the burst. The elimination of the need to specify the address for the transfers within the burst significantly improves the rate for transferring the block of data. Because of this capability, synchronous SRAMs and DRAMs are rapidly becoming the RAMs of choice for building memory systems in computers. We discuss the use of synchronous DRAMs in a memory system in more detail in the next section and in COD Chapter 5 (Large and fast: Exploiting memory hierarchy).

DRAMs

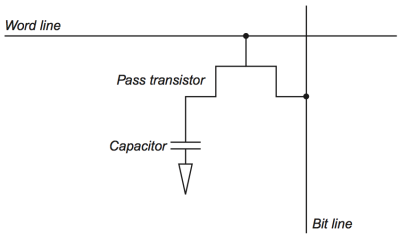
In a static RAM (SRAM), the value stored in a cell is kept on a pair of inverting gates, and as long as power is applied, the value can be kept indefinitely. In a dynamic RAM (DRAM), the value kept in a cell is stored as a charge in a capacitor. A single transistor is then used to access this stored charge, either to read the value or to overwrite the charge stored there. Because DRAMs use only a single transistor per bit of storage, they are much denser and cheaper per bit. By comparison, SRAMs require four to six transistors per bit. Because DRAMs store the charge on a capacitor, it cannot be kept indefinitely and must periodically be *refreshed*. That is why this memory structure is called *dynamic*, as opposed to the static storage in a SRAM cell.

To refresh the cell, we merely read its contents and write it back. The charge can be kept for several milliseconds, which might correspond to close to a million clock cycles. Today, single-chip memory controllers often handle the refresh function independently of the processor. If every bit had to be read out of the DRAM and then written back individually, with large DRAMs containing multiple megabytes, we would constantly be refreshing the DRAM, leaving no time for accessing it. Fortunately, DRAMs also use a two-level decoding structure, and this allows us to refresh an entire row (which shares a word line) with a read cycle followed immediately by a write cycle. Typically, refresh operations consume 1% to 2% of the active cycles of the DRAM, leaving the remaining 98% to 99% of the cycles available for reading and writing data.

Elaboration

How does a DRAM read and write the signal stored in a cell? The transistor inside the cell is a switch, called a pass transistor, that allows the value stored on the capacitor to be accessed for either reading or writing. The figure below shows how the single-transistor cell looks. The pass transistor acts like a switch: when the signal on the word line is asserted, the switch is closed, connecting the capacitor to the bit line. If the operation is a write, then the value to be written is placed on the bit line. If the value is a 1, the capacitor will be charged. If the value is a 0, then the capacitor will be discharged. Reading is slightly more complex, since the DRAM must detect a very small charge stored in the capacitor. Before activating the word line for a read, the bit line is charged to the voltage that is halfway between the low and high voltage. Then, by activating the word line, the charge on the capacitor is read out onto the bit line. This causes the bit line to move slightly toward the high or low direction, and this change is detected with a sense amplifier, which can detect small changes in voltage.

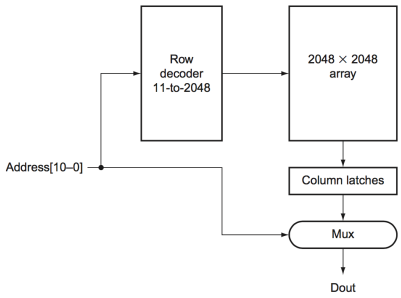
Figure 7.9.5: A single-transistor DRAM cell contains a capacitor that stores the cell contents and a transistor used to access the cell (COD Figure A.9.5).



DRAMs use a two-level decoder consisting of a *row access* followed by a *column access*, as shown in the figure below. The row access chooses one of a number of rows and activates the corresponding word line. The contents of all the columns in the active row are then stored in a set of latches. The column access then selects the data from the column latches. To save pins and reduce the package cost, the same address lines are used for both the row and column address; a pair of signals called RAS (*Row Access Strobe*) and CAS (*Column Access Strobe*) are used to signal the DRAM that either a row or column address is being supplied. Refresh is performed by simply reading the columns into the column latches and then writing the same values back. Thus, an entire row is refreshed in one cycle. The two-level addressing scheme, combined with the internal circuitry, makes DRAM access times much longer (by a factor of 5—10) than SRAM access times. In 2004, typical DRAM access times ranged from 45 to 65 ns; 256 Mbit DRAMs are in full production, and the first customer samples of 1 GB DRAMs became available in the first quarter of 2004. The much lower cost per bit makes DRAM the choice for main memory, while the faster access time makes SRAM the choice for caches.

Figure 7.9.6: A 4M X 1 DRAM is built with a 2048 X 2048 array (COD Figure A.9.6).

The row access uses 11 bits to select a row, which is then latched in 2048 1-bit latches. A multiplexor chooses the output bit from these 2048 latches. The RAS and CAS signals control whether the address lines are sent to the row decoder or column multiplexor.



You might observe that a 64M × 4 DRAM actually accesses 8K bits on every row access and then throws away all but four of those during a column access. DRAM designers have used the internal structure of the DRAM as a way to provide higher bandwidth out of a DRAM. This is done by allowing the column address to change without changing the row address, resulting in an access to other bits in the column latches. To make this process faster and more precise, the address inputs were clocked, leading to the dominant form of DRAM in use today: synchronous DRAM or SDRAM.

Since about 1999, SDRAMs have been the memory chip of choice for most cache-based main memory systems. SDRAMs provide fast access to a series of bits within a row by sequentially transferring all the bits in a burst under the control of a clock signal. In 2004, DDRRAMs (Double Data Rate RAMs), which are called double data rate because they transfer data on both the rising and falling edge of an externally supplied clock, were the most heavily used form of SDRAMs. As we discuss in COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy), these high-speed transfers can be used to boost the bandwidth available out of main memory to match the needs of the processor and caches.

Error correction

Because of the potential for data corruption in large memories, most computer systems use some sort of error-checking code to detect possible corruption of data. One simple code that is heavily used is a *parity code*. In a parity code the number of 1s in a word is counted; the word has odd parity if the number of 1s is odd and even otherwise. When a word is written into memory, the parity bit is also written (1 for odd, 0 for even). Then, when the word is read out, the parity bit is read and checked. If the parity of the memory word and the stored parity bit do not match, an error has occurred.

A 1-bit parity scheme can detect at most 1 bit of error in a data item; if there are 2 bits of error, then a 1-bit parity scheme will not detect any errors, since the parity will match the data with two errors. (Actually, a 1-bit parity scheme can detect any odd number of errors; however, the probability of having three errors is much lower than the probability of having two, so, in practice, a 1-bit parity code is limited to detecting a single bit of error.) Of course, a parity code cannot tell which bit in a data item is in error.

A 1-bit parity scheme is an *error detection code*; there are also *error correction codes* (ECC) that will detect and allow correction of an error. For large main memories, many systems use a code that allows the detection of up to 2 bits of error and the correction of a single bit of error. These codes work by using more bits to encode the data; for example, the typical codes used for main memories require 7 or 8 bits for every 128 bits of data.

Error detection code: A code that enables the detection of an error in data, but not the precise location and, hence, correction of the error.

Elaboration

A 1-bit parity code is a distance-2 code, which means that if we look at the data plus the parity bit, no 1-bit change is sufficient to generate another legal combination of the data plus parity. For example, if we change a bit in the data, the parity will be wrong, and vice versa. Of course, if we change 2 bits (any 2 data bits or 1 data bit and the parity bit), the parity will match the data and the error cannot be detected. Hence, there is a distance of two between legal combinations of parity and data.

To detect more than one error or correct an error, we need a distance-3 code, which has the property that any legal combination of the bits in the error correction code and the data has at least 3 bits differing from any other combination. Suppose we have such a code and we have one error in the data. In that case, the code plus data will be one bit away from a legal combination, and we can correct the data to that legal combination. If we have two errors, we can recognize that there is an error, but we cannot correct the errors. Let's look at an example. Here are the data words and a distance-3 error correction code for a 4-bit data item.

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

To see how this works, let's choose a data word, say 0110, whose error correction code is 011. Here are the four 1-bit error possibilities for this data: 1110, 0010, 0100, and 0111. Now look at the data item with the same code (011), which is the entry with the value 0001. If the error correction decoder received one of the four possible data words with an error, it would have to choose between correcting to 0110 or 0001. While these four words with error have only 1 bit changed from the correct pattern of 0110, they each have 2 bits that are different from the alternate correction of 0001. Hence, the error correction mechanism can easily choose to correct to 0110, since a single error is a much higher probability. To see that two errors can be detected, simply notice that all the combinations with 2 bits changed have a different code. The one reuse of the same code is with 3 bits different, but if we correct a 2-bit error, we will correct to the wrong value, since the decoder will assume that only a single error has occurred. If we want to correct 1-bit errors and detect, but not erroneously correct, 2-bit errors, we need a distance-4 code.

Although we distinguished between the code and data in our explanation, in truth, an error correction code treats the combination of code and data as a single word in a larger code (7 bits in this example). Thus, it deals with errors in the code bits in the same fashion as errors in the data bits.

While the above example requires n — 1 bits for n bits of data, the number of bits required grows slowly, so that for a distance-3 code, a 64-bit word needs 7 bits and a 128-bit word needs 8. This type of code is called a Hamming code, after R. Hamming, who described a method for creating such codes.

(*1) This section is in original form.