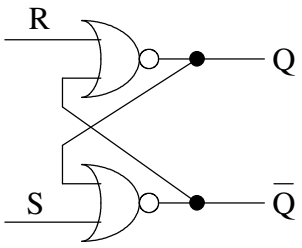
"Memory" Supplement for Section 3.6 of the textbook

The most basic 1-bit memory is the *SR-latch* with consists of two cross-coupled NOR gates.



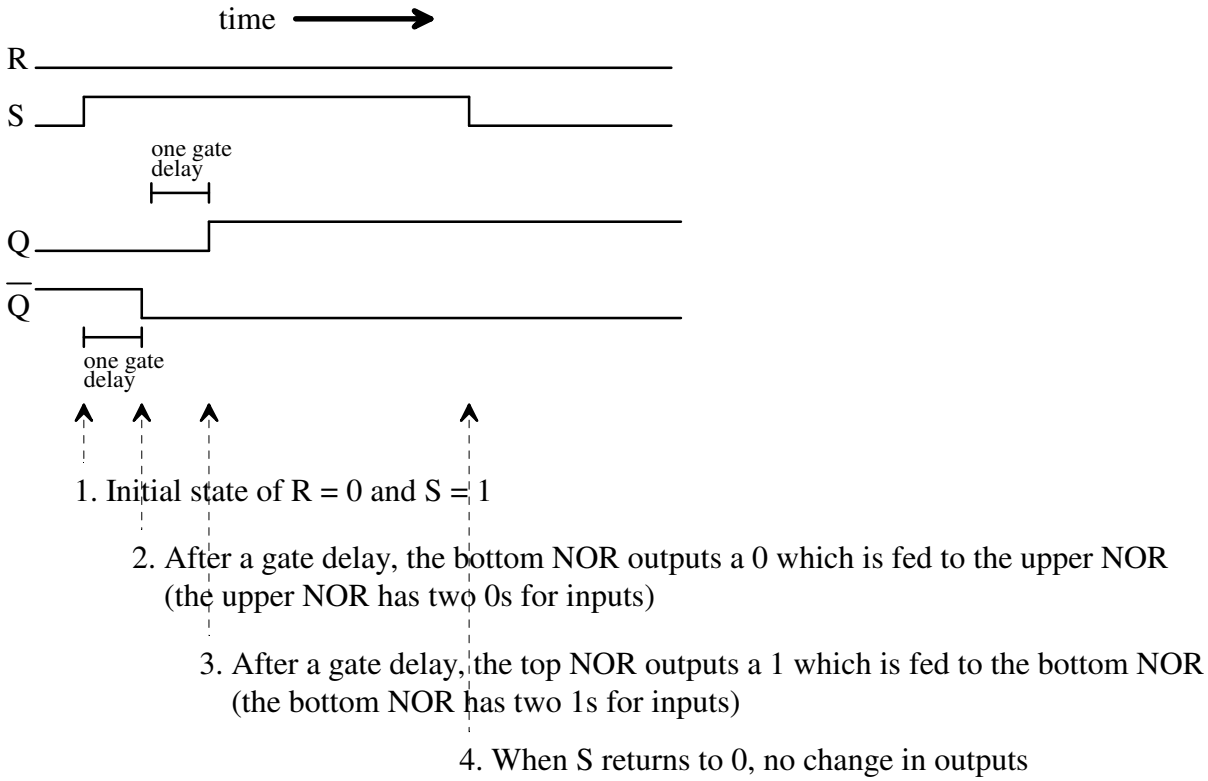
Recall the NOR gate truth table:

A	B	$\overline{(A+B)}$
0	0	1
0	1	0
1	0	0
1	1	0

The "S" stands for Set to remember 1, and the R for Reset to remember 0. For example, if the input values of R = 0 and S = 1 are held constant, then the bottom NOR gate will start outputting 0. This 0 is fed as an input to the upper NOR with the R = 0, so both of the upper NOR gate's inputs are 0. Thus, the upper NOR gate will start outputting a 1 which is fed back to the bottom NOR as an input. Now, the bottom NOR has an input of two 1's, but it continues to output a 0 because that's what NORs do (see truth table).

If the S input's value is now changed back to 0 (both R =0 and S=0), then bottom NOR will continue to output 0 (i.e., $\overline{Q} = 0$) since the top NOR is still outputting a 1. The top NOR has two 0 inputs, so it continues to output 1 (i.e., $Q = 1$). This is a stable state where the SR-latch is remembering a 1.

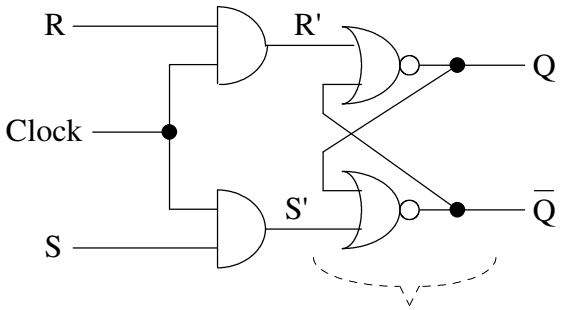
A timing diagram for the above scenario would look like:



The SR-latch is actual symmetric from top-to-bottom, so a similar argument can be made for the inputs of R = 1 and S = 0 causing a stable state of it remembering a 0.

The main problem with the SR-latch is that its output changes are determined by some number of gate delays after its inputs are changed. This makes it difficult for a collection of one-bit memories to work in a lock-step fashion to hold a multiple bit value. It also has the problem that inputs of R = 1 and S = 1 lead to an undefined state.

We can solve these problems in a series of steps. First we'll add a Clock input to prevent the memory from changing states if the Clock = 0. This leads us to a *clocked-SR latch*:



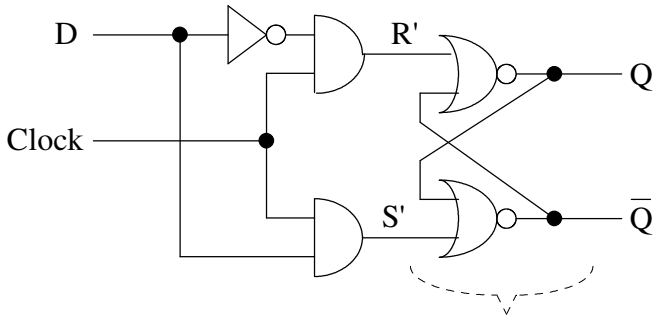
Original SR-latch

When the Clock = 0, both AND gates will output 0s. This guarantees that the original SR-latch has inputs of R'= 0 and S'= 0, so it cannot change its memory state.

When the Clock = 1, the AND gates allow the R and S inputs to be passed through without change. This does not exactly solve either of the previously mentioned problems, but we can at least prevent the *clocked-SR latch* from changing by having the Clock = 0.

To solve the problem of inputs of R = 1 and S = 1 leading to an undefined state. We'll consider a slightly different clocked latch called a *clocked D-latch*. The clocked D-latch remembers the value on the D input when the Clock = 1. If the Clock = 0, then the D input is ignored and the one-bit memory is not changed.

The clocked D-latch circuit is very similar to the clocked SR latch.

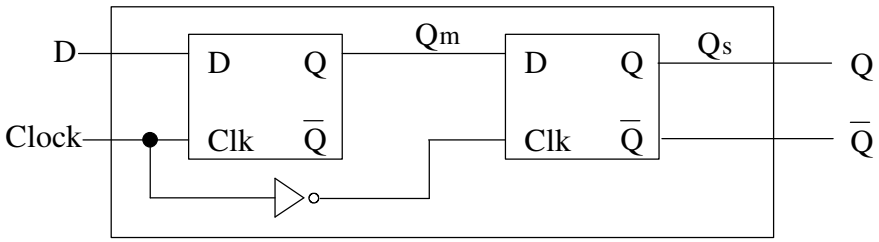


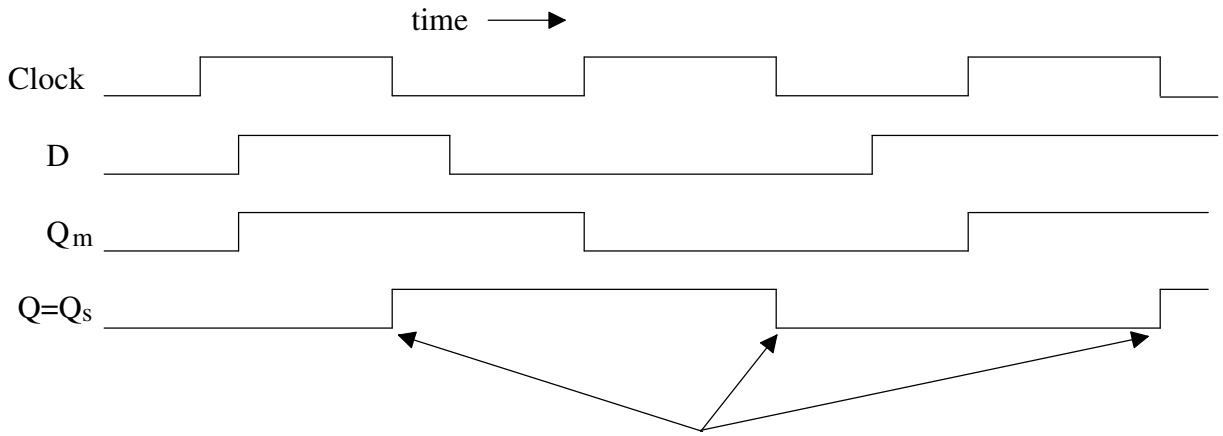
Original SR-latch

To fix the problem of the one-bit memory having its output changing being determined by some number of gate delays after its inputs are changed. We can use two clocked D-latches to build a *Master-Slave D flip-flop* which changes state only on the falling edge of the clock pulse. The below timing diagram shows how the internal clocked D-latches change within the master-slave flip-flop to achieve this.

When the Clock = 1, the master D-latch can change to D's value, but the slave D-latch cannot change since it receives the negated Clock signal of 0. When the Clock = 0, the master D-latch cannot change, but the slave D-latch can change to remember the value being remembered in the master D-latch (Q_m).

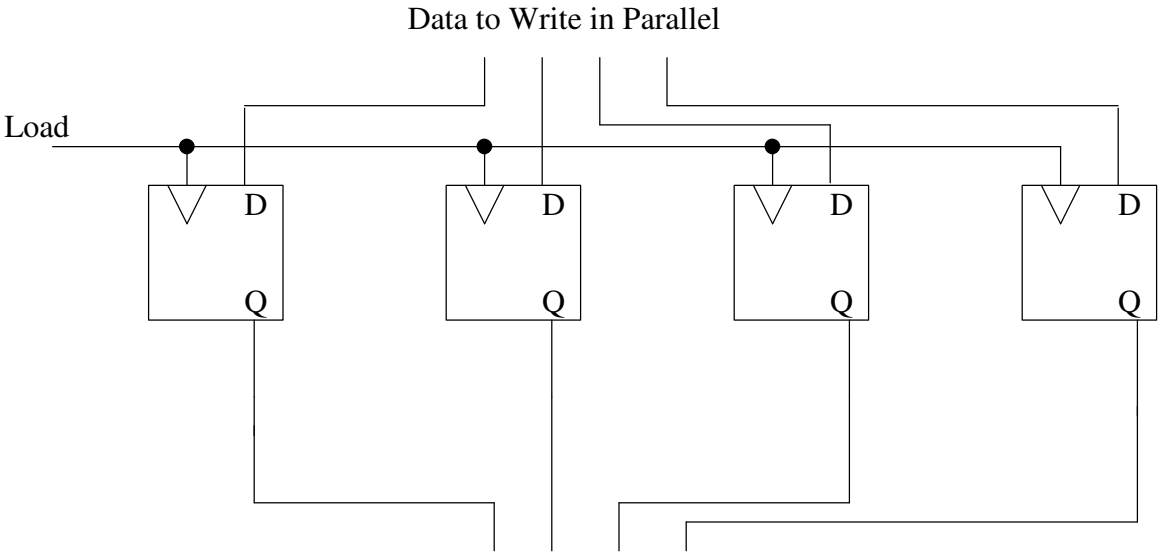
"Memory" Supplement for Section 3.6 of the textbook





The value of Q only changes on the falling-edge of the Clock

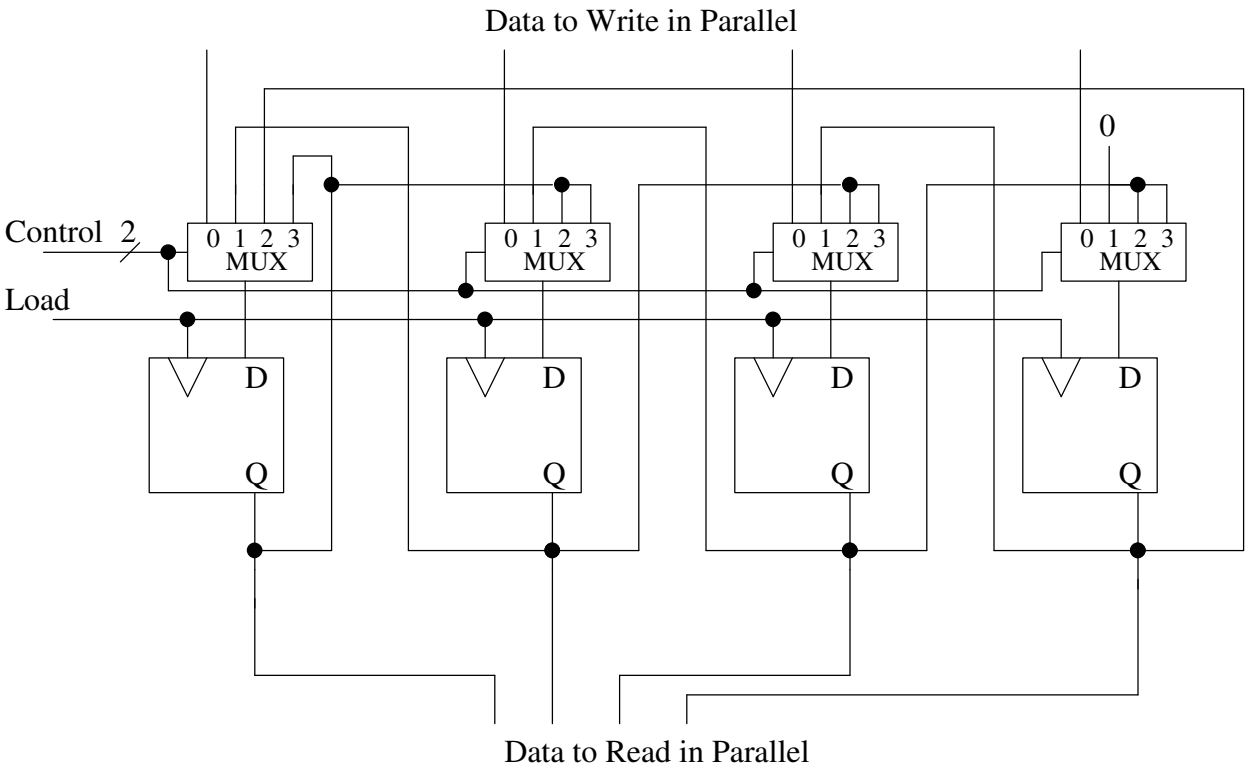
This master-slave D flip-flop allows for a collection of one-bit memories to work in a lock-step fashion to hold a multiple bit value. The value currently contained in a D flip-flop can to read (from the slave's Q) in the same clock pulse as a new value is written to it (into the master's D input). The following block diagram shows how four D flip-flops can be used to load and read 4-bit value in parallel. The "triangle" input of each flip-flop connected to the Load wire is the "Clock" input in the above diagram. This triangle input indicates that a flip-flop is being used instead of a latch.



Another more complex example would be a *shift-register* that allows the value stored in the register to also be shifted by one bit position several ways. The below diagram is a 4-bit shift-register that is able to perform the following operations:

- parallel read/output of all bits (just look at the Q values)
- (Control of 00_2) parallel write/input of all bits
- (Control of 01_2) logical shift left one bit position (value shifted out of most-significant bit is lost and a "0" is shifted into the least-significant bit)
- (Control of 10_2) circular shift right one bit position (value shifted out of least-significant bit is shifted into the most-significant bit)
- (Control of 11_2) arithmetic shift right (sign-extend the most-significant bit)

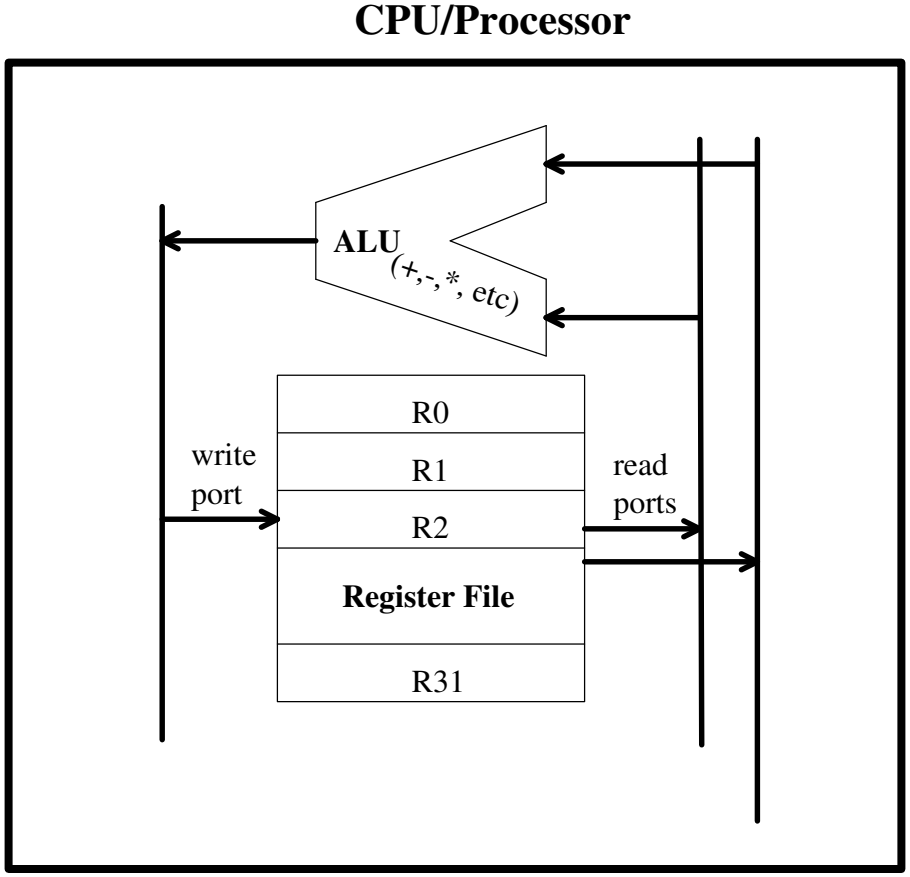
For each D-flip flop, the output of a MUX is used as the D-input. The Control wires control the MUXs to vary the value routed to the D inputs.



Register File:

Computers typically have a collection of registers, called a *register file*, to store data values currently being calculated with by the CPU. Individual registers are numbered (register 0 or R0, register 1 or R1, etc.) and assigned by the compiler or the assembly-language programmer to hold variable values.

Since most arithmetic operations need two operands (e.g., addition: Y + Z), most register files have at least two *read/output ports* and one *write/input port* to accommodate sending two values to the ALU and receiving one result.

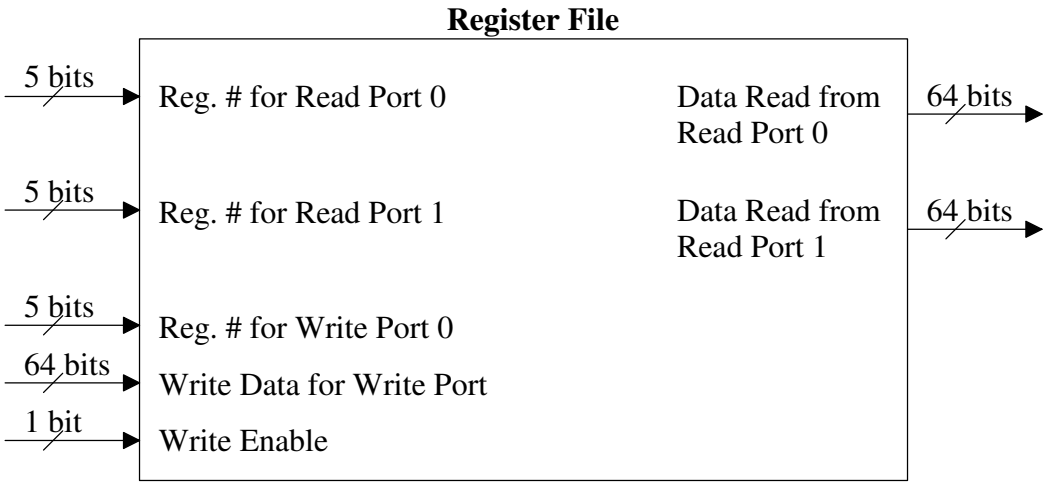


To control each read port we need to be able to specify a register number for the register to be read. The width/number of bits read equals the number of bits per register. For example, 32 register (R0 to R31) need 5-bits to represent this range of values 00000_2 to 11111_2, i.e., 0_{10} to 31_{10}.

To control a write port we need to be able to specify a register number for the register to be written, the data to be written (equal to the number of bits in a register), and a write-enable signal. The write-enable signal indicates if we are writing a register, i.e., indicates if we should ignore the write register # and data.

A register-file block-diagram assuming:

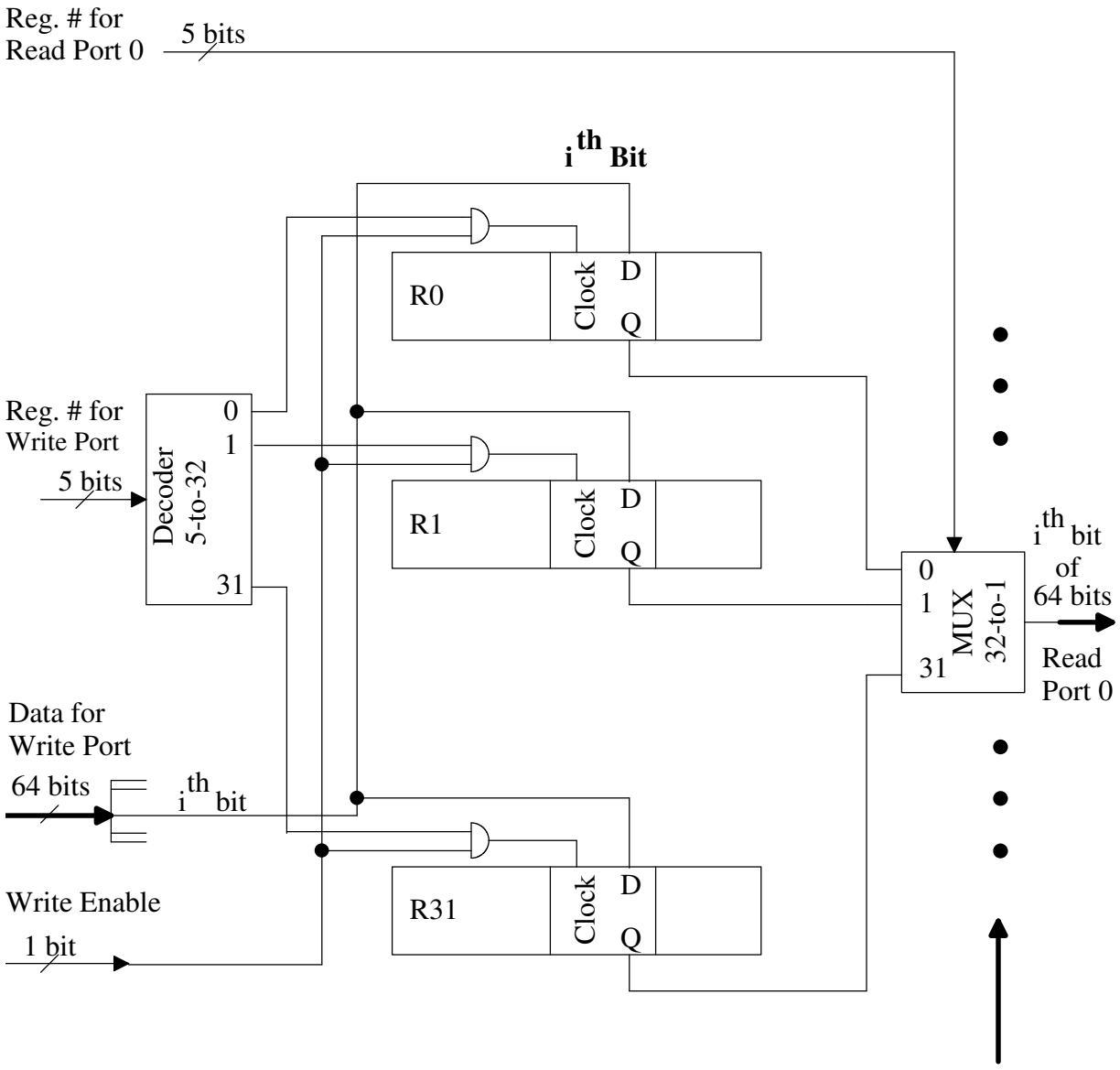
- 32 registers numbered R0 to R31 (00000_2 to 11111_2 in binary)
- 64-bit registers
- two read ports
- one write port



One-bit (i^{th} bit) Slice of a Register File Implementation Using D-latches:
The diagram below shows only a one-bit slice for the i^{th} bit of the register file to simplify the diagram.

The write-port register number (5-bit number from R0 to R31) is input to a decoder to "wake-up" the corresponding register to be written by sending a 1 into the clock signal for register to be written. The decoder sends 0s to all the other registers' clock signals. Notice that the selected register only receives the 1 from the decoder if the write-enable signal is also a 1.

When a register is read at a read-port, the whole register is read in parallel (i.e., all 64-bits at the same time). For example, 64-bit registers requires 64 data wires of output. Each output wire is fed by a multiplexer which uses the register number to be read as its Control to select the bit from the appropriate register. In this example 64 MUXs each with 32-inputs (one for each register) are need per read port. Since we have two read ports, a total of 128 MUXs would be needed.



Note: To write a D flip-flop, we want its reg. # to be specified and the "Write Enable" to be asserted.

Note: 64 MUXs are need per Read Port - one per bit

"Memory" Supplement for Section 3.6 of the textbook

To help clarify the above discussion, a complete (not just a one-bit slice) register file that has:

- 4 registers (numbered R0 to R3, i.e., register numbers 00_2 to 11_2)
- 2-bits per register
- one write-port
- two read-ports

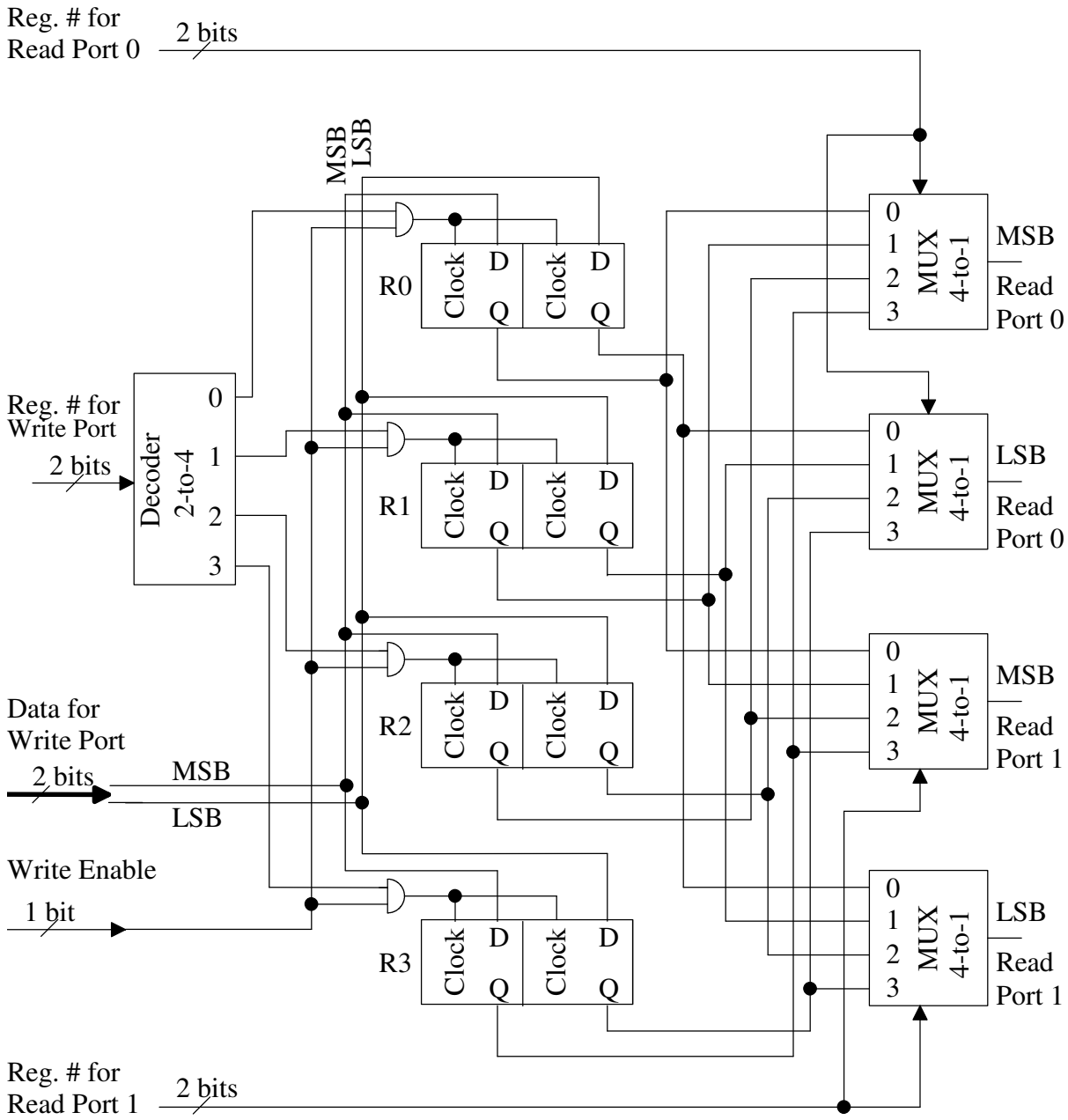


Figure 3.32 on page 143 is a similar to the above picture, except it is for a 4 word x 3 bit *memory*. Memories unlike register files typically have only a single combined read/write port. So only a read or a write can be performed at the same time. That's why Figure 3.32 has only one set of address (S_0 and S_1) lines that act like the register # in the above diagram. Notice that Figure 3.32 shows the AND gates and OR gates need to implement the decoder (along the bottom) and the MUXs (along the right and above the D flip-flops).

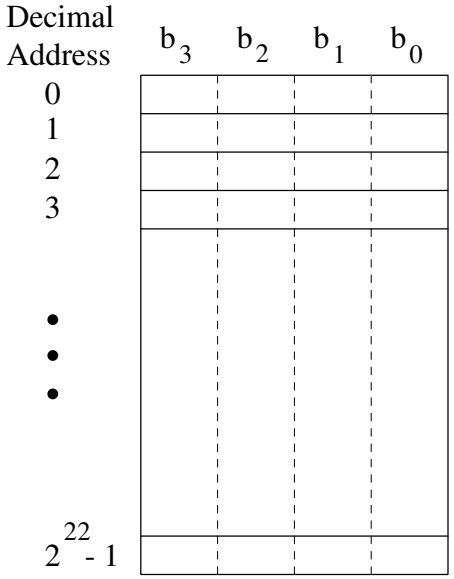
Implementation of Large Memory Chips:

The above register-file design does not scale well for large (realistic) memories for several reasons:
- the number of gates in the address decoder (and MUXs) grows exponentially with the number of bits in the address.
- lots of wires into/out of the memory chip for address, data, and control

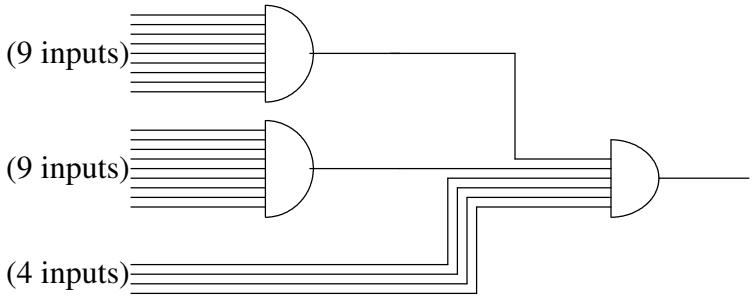
These problems are solved by
- using square-array of bits and decoding the address in two parts (row then column number)
- eliminate MUX's by using tri-state buffers
- single-port RAM memory - data wires shared for reading and writing, and one address specified

Consider a larger memory chip with 4M word x 4-bit words, i.e., a 4M x 4 memory chip. This chip would have a 22-bit addresses since $4M = 2^{22}$.

Conceptually (logically/externally) we view a 4M x 4-bit memory as pictured below with:
- each memory word made up of four bits: $b_3\ b_2\ b_1\ b_0$



When we want to write a word, we supply a 22-bit address and the corresponding new 4-bit word for that row. If we actually implemented this memory using a register-file design, we'd need a 22-to-2^{22} decoder for the write-port. If we think about how a decoder is implemented, each decoder output is fed by an AND gate looking for its corresponding address. Thus, we'd need 2^{22} (or 4 M) 22-input AND gates. Since 22-input AND gates exceed the limit of 9-inputs or less for an AND, we would need to build each 22-input AND gate using 3 smaller ANDs as:



A total of 3 x 2^{22} AND gates are needed to implement just the write-port decoder.

As in Figure 3.32, the read port could use the same decoder as the write port. Still, each bit in the memory requires a 2-input AND gate. Additionally, each of the four output wires is fed by a really BIG 2^{22}-input OR gate. Because of the 9-input limit, each 2^{22}-input OR must be implemented by multiple levels of smaller OR gates. The number of actual OR gates needed to implement each 2^{22}-input OR would be about:

$$\left\lceil \frac{2^{22}}{9} \right\rceil + \left\lceil \frac{2^{22}}{9^2} \right\rceil + \left\lceil \frac{2^{22}}{9^3} \right\rceil + \ldots + 1 = 524{,}289$$

The total for number of gates for the read port would be 4 x 2^{22} ANDs + 4 x 524,289 ORs.

Assuming D-latches to store each bit of the memory, the total number of gates needed for storing all 2^{22} x 4 bits of memory is 2^{22} x 4 x 4 = 2^{26} gates.

Overall, the number of gates in the whole memory chip is about (3 x 2^{22}) + (4 x 2^{22} + 4 x 524,289) + (2^{26}) .

Since the D-latch gates are the only ones that are used for actually storage, the gates associated with the read and write ports can be considered overhead. The percentage of gates used for actual storage (the D-latch gates) of bits is:

$$\frac{2^{26}}{(3\times 2^{22})+(4\times 2^{22}+4\times 524{,}289)+(2^{26})} \text{ x } 100 = 68.1\ \%$$

In practice memory chips do not use SR-latch based 1-bit memories, called *static memory*, because each bit requires multiple gates per bit of storage (e.g., 4 gates per bit for a D-latch). To increase the amount of memory on a chip, *dynamic memory* is used that only requires **one gate per bit** of storage. The one gate used in one bit of dynamic memory acts like a capacitor. You might recall from Physics that a capacitor is a device for storing a charge across two nearby "plates." The electrons from one plate are forces from one plate to the other plate. Because the plates are insulated from each other, the charge is stored. The magnitude of the charge difference between the plates determines whether a "0" is being stored (say a voltage difference of 0 to 1 volts) or a "1" is being stored (say a voltage difference of 2 to 5 volts) in the memory.

Dynamic memory (the "D" in DRAM) can store about four times more bits on a chip than static memory, but is also has some problems:

- reading the value being stored is relatively slow since the volt difference must be sensed to determine the binary value being stored.
- chips are made from semiconductor material, so a dynamic-memory gate storing a "1" with a high voltage difference (say 5 volts) will have electrons leak across the "plates" and eventually be storing a "0" (voltage difference of 0). To solve this problem, dynamic memory chips must be *refreshed* which means that periodically the whole memory must be read and rewritten to boost the "1" values up to a high voltage difference. Usually, the memory refreshing only takes a fraction of a percent of the memory's overall execution time.

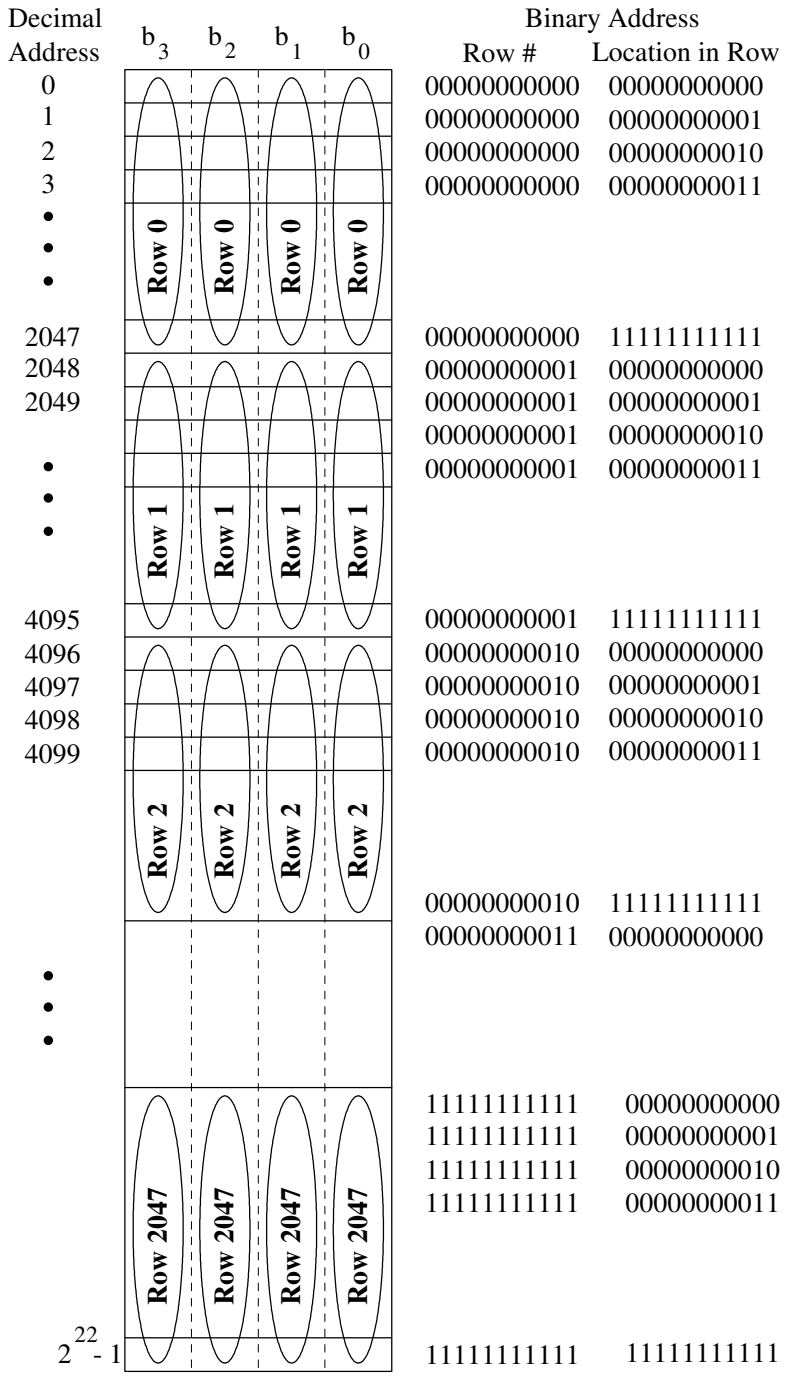
"Memory" Supplement for Section 3.6 of the textbook

If we recalculate the percentage of gates used for actual storage assuming dynamic memory (one gate per bit), we get:

$$\frac{2^{24}}{(3\times 2^{22})+(4\times 2^{22}+4\times 524{,}289)+(2^{24})} \text{ x } 100 = 34.8\ \%$$

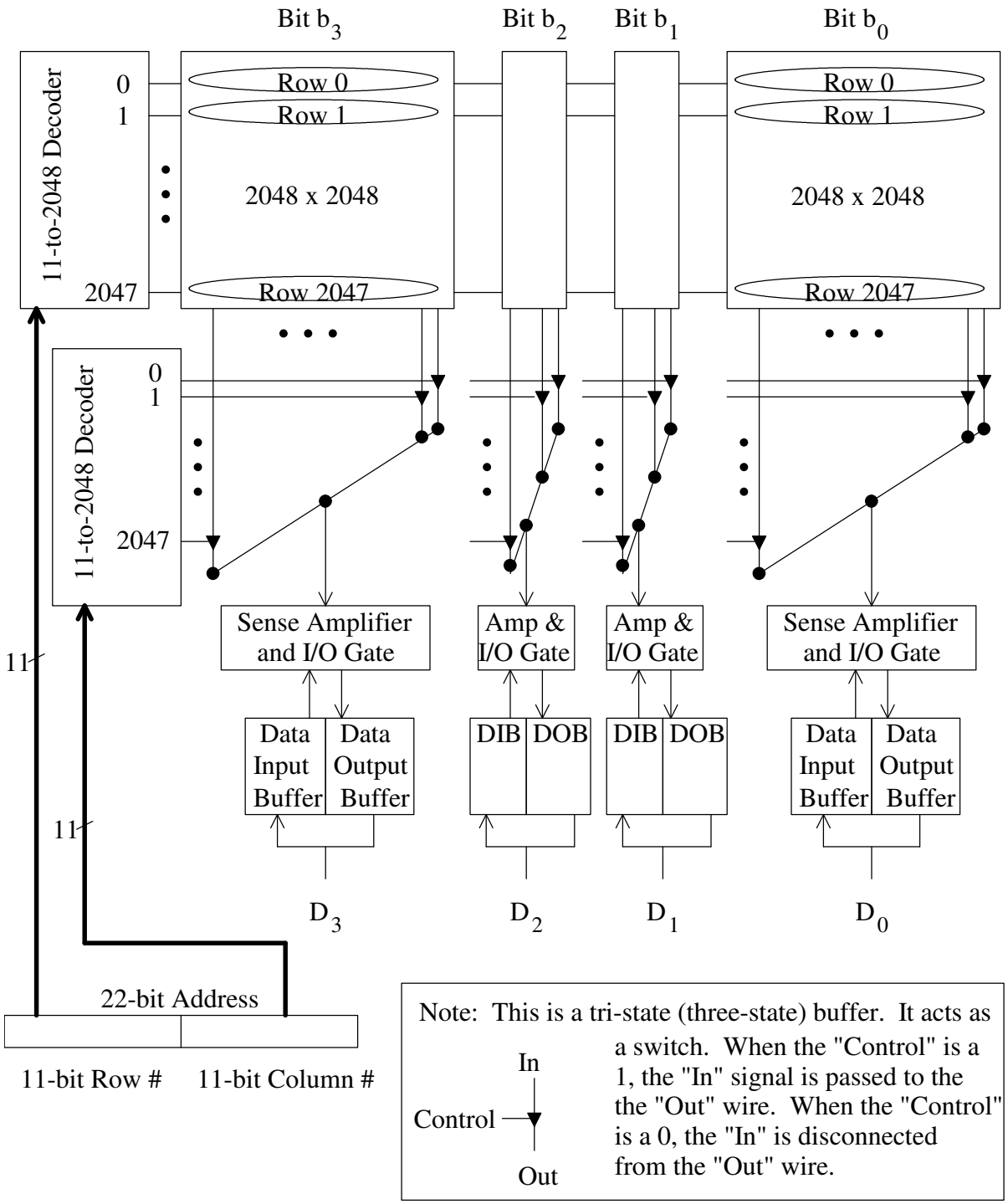
Thus, 65.2 % of the gates on the chip are being "wasted" in overhead for decoding, MUXs, etc.

To help us see how the 4M x 4-bit memory gets mapped to the 2048 x 2048 x 4 memory array on the next page consider splitting memory into 2048 word blocks as shown below. Notice that the most-significant 11 bits of the binary address correspond to the row number, and the least-significant 11 bits of the address to the offset within the row.



"Memory" Supplement for Section 3.6 of the textbook

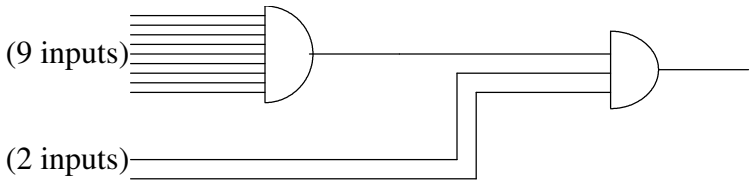
Each bit of a word is split into a separate 2048 x 2048 square memory "array". Each of these square memory arrays is 2048 x 2048 = 2^{11} x 2^{11} = 2^{22} = 4M bits.
The 22-bit address of the 4M memory is split into two 11-bit parts. The upper 11-bits is first used to activate the correct row within the square memory arrays. Of the 2048-bit row that is read from each memory array, we are interested in only one bit. The lower 11-bits of the address specifies the location of the desired bit within the 2048-bit row.



Note: This is a tri-state (three-state) buffer. It acts as a switch. When the "Control" is a 1, the "In" signal is passed to the the "Out" wire. When the "Control" is a 0, the "In" is disconnected from the "Out" wire.

"Memory" Supplement for Section 3.6 of the textbook

For this "square-memory" design of memory, we can recalculate the percentage of gates used for actual storage (the dynamic-memory "capacitor" gates) of bits. We now have two, smaller, 11-to-2^{11} decoders. Each decoder output is fed by an AND-gate looking for its corresponding address. Thus, we'd need 2^{11} (or 2 K) 11-input AND gates. Since 11-input AND gates exceed the limit of 9-inputs or less for an AND, we would need to build each 11-input AND gate using 2 smaller ANDs as:



A total of 2 x 2 x 2^{11} = 2^{13} AND gates are needed to implement both the row and column decoders.

No MUXs are needed in the square-memory design, but we do need 4 x 2^{11} = 2^{13} tri-state buffers.

Assuming dynamic memory to store each bit of the memory, the total number of gates needed for storing all 2^{22} x 4 bits of memory is 2^{22} x 4 = 2^{24} gates.

Overall, the number of gates in the whole memory chip is about $2^{13} + 2^{13} + 2^{24}$.

For the square-memory design, the percentage of gates used for actual storage of bits is:

$$\frac{2^{24}}{2^{13}+2^{13}+2^{24}} \text{ x } 100 = 99.9\ \%$$

Thus, only about 0.1 % of the memory chips capacity is wasted on overhead for decoding, MUX, etc.

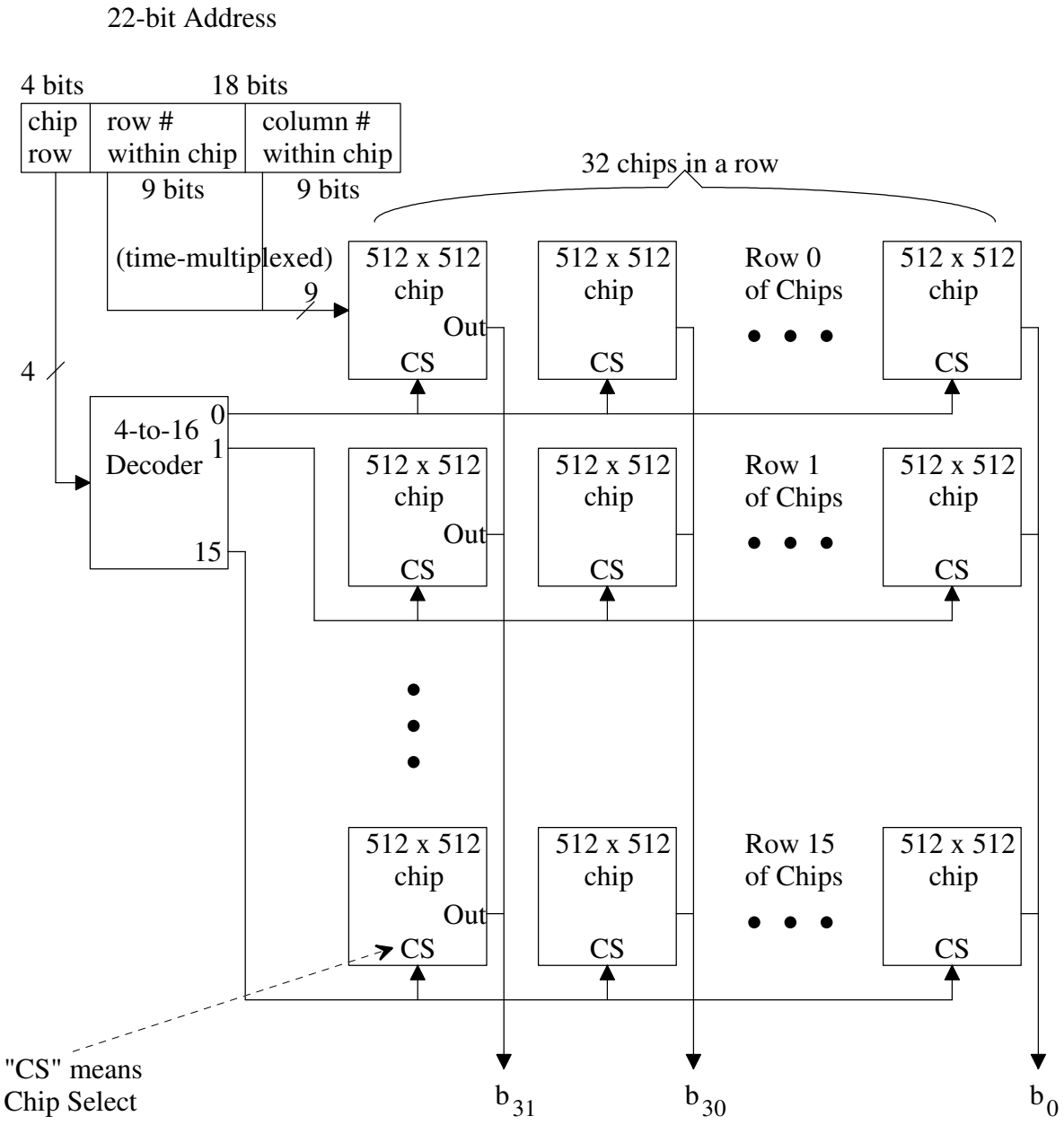
The square-memory design also offers a couple of other important benefits:

- when refreshing the memory, whole rows can be read and rewritten at a time so refreshing is relatively quick
- since whole rows are read from the slow dynamic memory at a time, a block of memory from contiguous memory locations can be read about as fast as a reading a single memory access. Thus, many real DRAM memory chips provide a *fast-page mode* or *burst* feature that allow accessing a block of memory efficiently.

Implementing Large Memory with Smaller Chips

Consider for example, implementing 4M x 32 bits memory with 256KB x 1 bit chips.
The 256KB x 1 chips are implemented as square arrays of 512 x 512.

We will use an two-dimensional array of the 256KB x 1 chips to implement the larger memory.

The number of chips per row would be 32 bits / 1 bit = 32 chips.
The number of chips per column would be 4M / 256K = $2^{22} / 2^{18} = 2^4 = 16$ chips per column.

The 22-bit address of the 4M x 32 bit memory would be split up as follows: