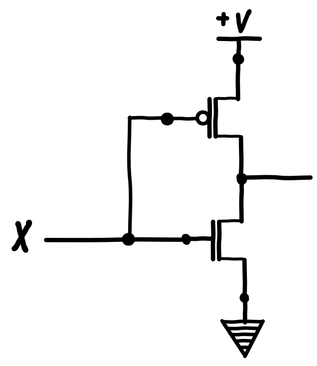
**120-final-studyGuide**



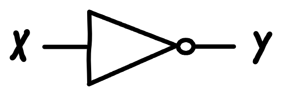
A CMOS circuit consists of two connected complementary sub-circuits:

1) P-type transistors connected to logic 1
2) N-type transistors connected to logic 0.

P-type transistors conduct when their input is logic 0.
N-type transistors conduct when their input is logic 1.

At left is a CMOS circuit for Y == NOT( X ).

logic symbol for NOT



A 2-input eXclusive-OR (XOR) is defined by this truth table:

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

where A and B are inputs, and Y is the output. A 3-input XOR is the functional composition of two 2-input XORs:

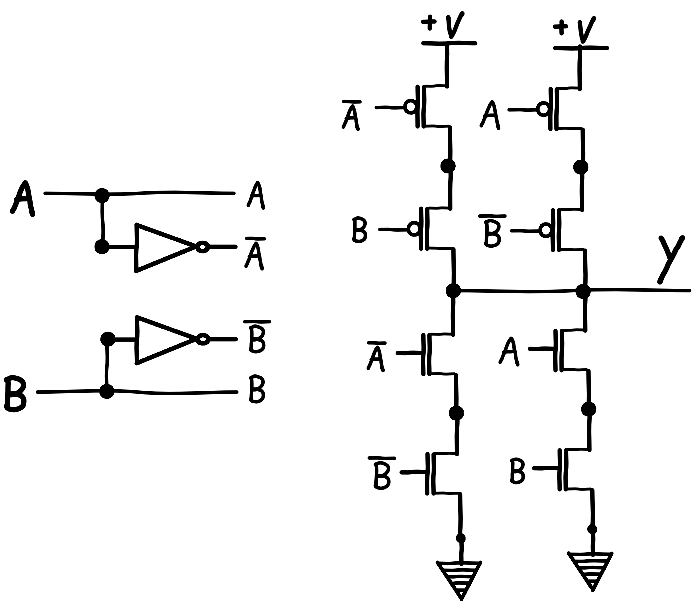
XOR( A, B, C ) == XOR( XOR( A, B ) , C )

There are two interpretations of XOR:

1) NOT-EQUAL( A, B )
   == 1 if A is not the same as B;

2) BIT-FLIP( A, B )
   == NOT( A ) if B == 1;
   == A if B == 0.

CMOS circuit for Y == XOR( A, B )



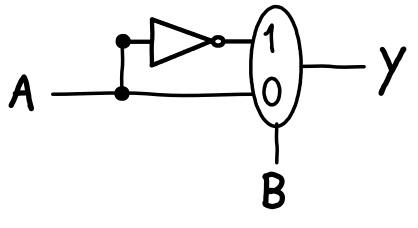
This circuit corresponds to the NOT-EQUAL interpretation:

In the P-transistor circuit, if A != B, then one of the paths will connect Y to +V (logic 1).

In the N-transistor circuit, if A == B, then one of the paths will connect Y to ground (logic 0).

This logic circuit corresponds to the BIT-FLIP interpretation:

B selects A or NOT(A).



There are two interesting classes of Boolean functions:

1) Minterm functions output a 1 for some one particular input, and output 0 for all others.
   That is, the output column of a minterm function's truth table has exactly one 1.
   If we OR minterms together, the output column of the resulting function's truth table will be 1 for any row where any of the minterms has a 1.

2) Maxterm functions output a 0 for some one particular input, and output 1 for all others.
   That is, the output column of a maxterm function's truth table has exactly one 0.
   If we AND maxterms together, the output column of the resulting function's truth table will be 0 for any row where any of the maxterms has a 0.

For instance, this is a 2-input minterm function:

| A | B | $m_1$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

This is another 2-input minterm function:

| A | B | $m_2$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

This the OR of the two minterm functions:

| A | B | $m_1$ OR $m_2$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

For instance, this is a 2-input maxterm function:

| A | B | $M_0$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

This is another 2-input maxterm function:

| A | B | $M_3$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

This the AND of the two maxterm functions:

| A | B | $M_0$ AND $M_3$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

We can express any function as an OR of minterms: for each row that has a 1, pick the minterm that has a 1 for that row, then OR the selected minterms.

We can express any function as an AND of maxterms: for each row that has a 0, pick the maxterm that has a 0 for that row, then AND the selected maxterms.

Minterms are labeled according to which input produces a 1. The input is treated as a binary number: for input ( A, B ) == ( 0, 0 ) the label is 0; for input ( 0, 1 ) the label is 1; for ( 1, 0 ) the label is 2; for ( 1, 1 ) the label is 3. The corresponding minterms are: $m_0$ , $m_1$ , $m_2$ , $m_3$ .

Maxterms are labeled according to which input produces a 0: $M_0$ , $M_1$ , $M_2$ , $M_3$ .

Aside from the fact that any function can be expressed as an OR of minterms (DNF, Disjunctive Normal Form) or as an AND of maxterms (CNF, Conjunctive Normal Form), minterms and maxterms are interesting because it is easy to build logic circuits for them.

For a minterm, we can build a circuit by expressing what **must be true** for the minterm to output a 1. That is, each input bit must conform to a particular value. For example, if the variable A must have the value 0, the expression is NOT(A), while if it must be 1 the expression is (A).

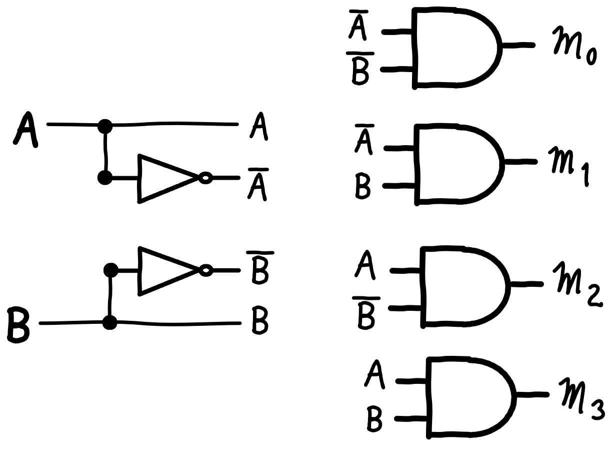
Further, it must also **be true that** every variable has the correct value; so, we AND the expressions for all the variables together. The result is an expression that is only true when all the conditions are met. This expression is the minterm.

For example, for $m_2$ it must be that ((A) AND NOT(B)) because $m_2$ outputs 1 exactly when both A is 1 and B is 0. For any other condition of the variables A and B, $m_2$ outputs a 0.

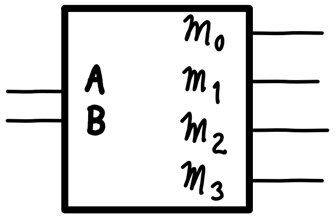
For maxterms, we set the conditions according to what must be **false**. For instance, for $M_2$ , the output must be 0 when all of the variables have the correct values. That could also be said, $M_2$ must output 1 if any of the variables has an incorrect value. So, if A must be 1, the logical expression stating that A has the wrong value is NOT(A). And if B must be 0, (B) is true when B has the wrong value. We OR this together to express that if any of them have the wrong value, $M_2$ outputs a 1:
( NOT(A)  OR  (B) ). And, of course, if none of them have the wrong value, $M_2$ outputs 0, saying in effect, It is false that any of the variables has an incorrect value.

A DECODER is a complete collection of minterms, one output wire per minterm. All the minterms share the same input, which is the input to the DECODER. Exactly one of the DECODER's outputs will be 1 for any particular input.

logic circuit for a 2-bit input DECODER

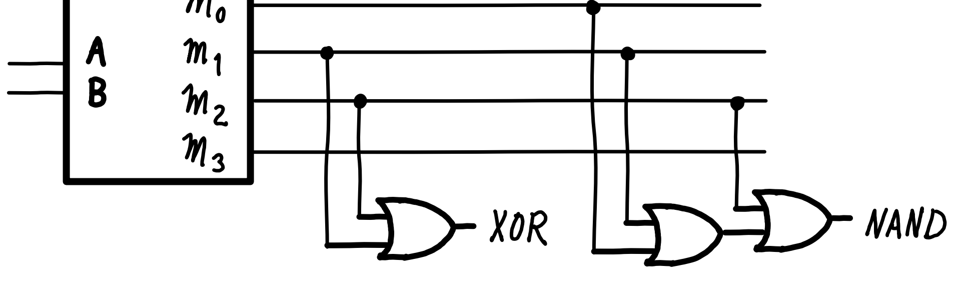


logic symbol for 2-bit DECODER



We can implement multiple functions with a single DECODER by OR'ing its minterm outputs.

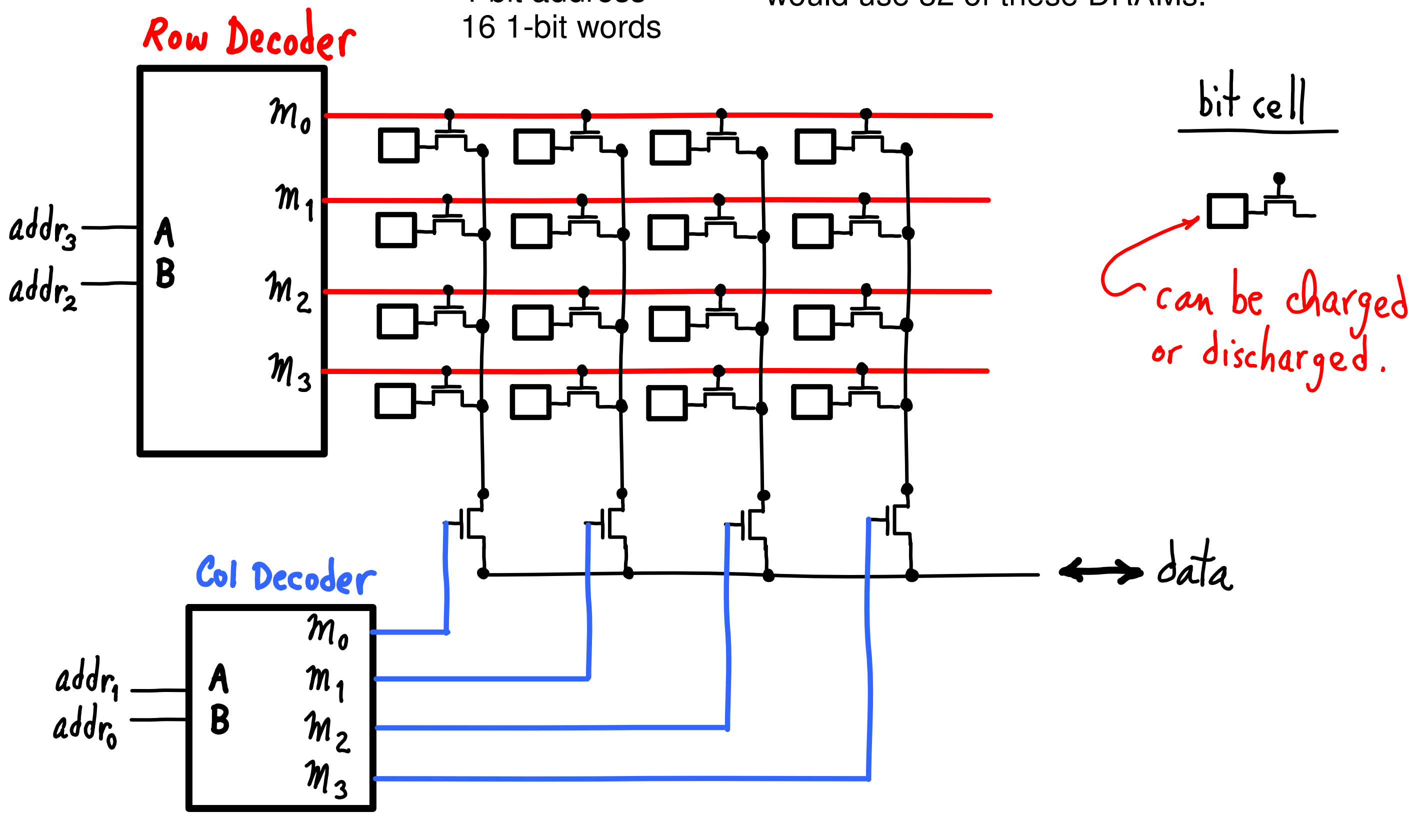
Here, we implement XOR and NAND. Of course, there is a cheaper way to implement NAND.



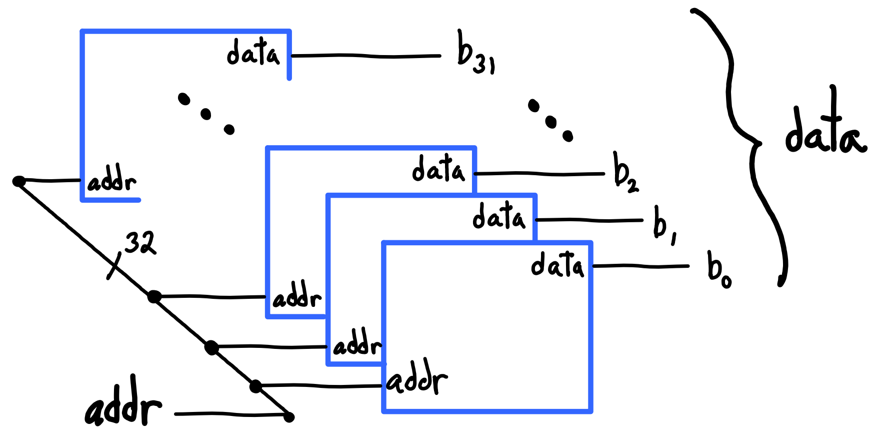
Memory chips are implemented using decoders. One way to keep the delay down is to lay out the bits in a 2-d array; that is, in rows and columns. The row decoder selects an entire row of bits. The column decoder selects which bit in the row to read or write. The transistors allow the bit cells to be charged to +V or discharged to 0. The bit cells are very small, and do not hold their charge for long. A memory controller has to recharge the bits at regular time intervals, called "refresh cycles".

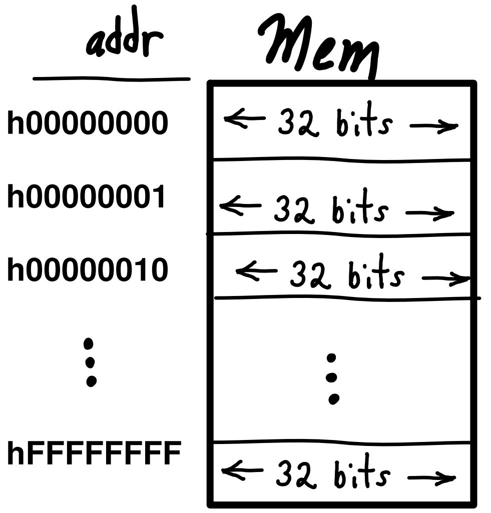
A 16-bit DRAM:
4-bit address
16 1-bit words

A 16-word memory w/ 32-bit words would use 32 of these DRAMs.

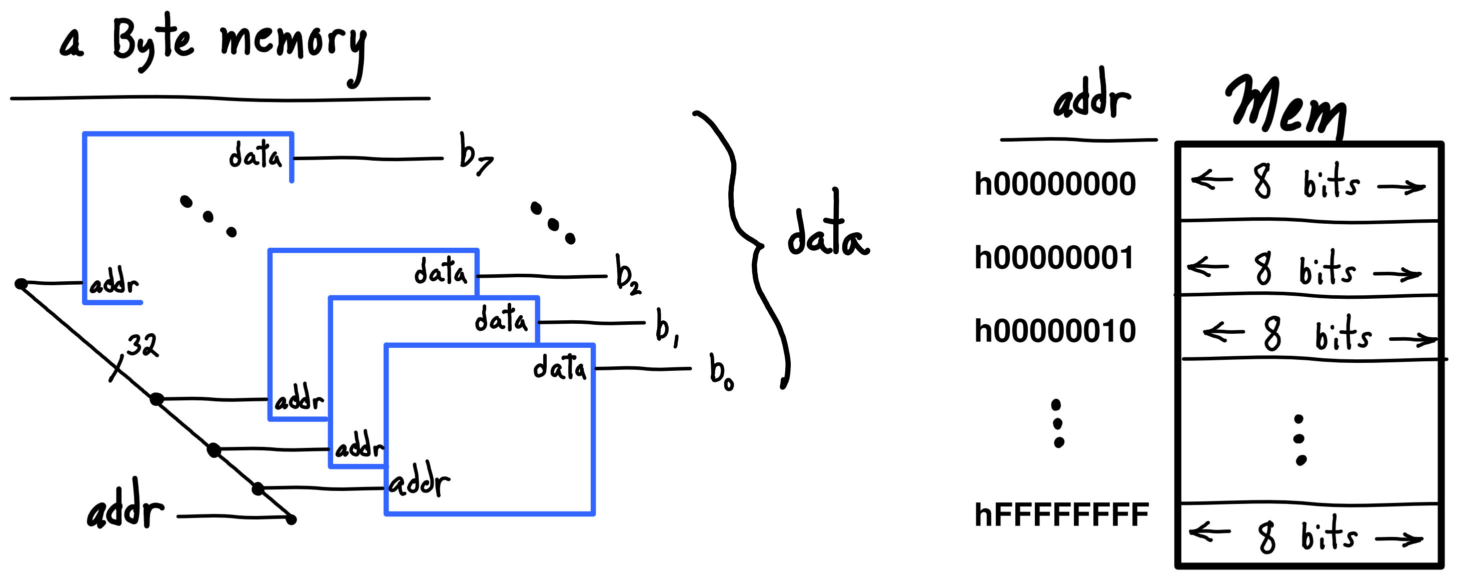


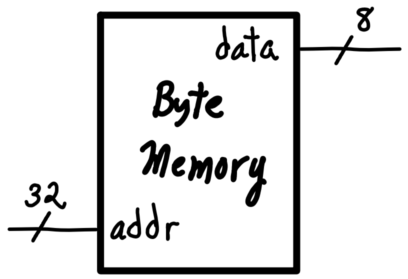
The addressability of a memory organization has to do with memory word size versus how many bits one advances through the memory by adding +1 to a memory address. Suppose we have 32-bit addresses and 32-bit words. A single DRAM chip could hold 1-G bit. Each bit is individually addressed as shown above. Using 32 of these would give us 1-G of 32-bit words. The 32 DRAMs would all share the same address input, and if we added +1 to the address, we would get the next 32-bit word in memory.



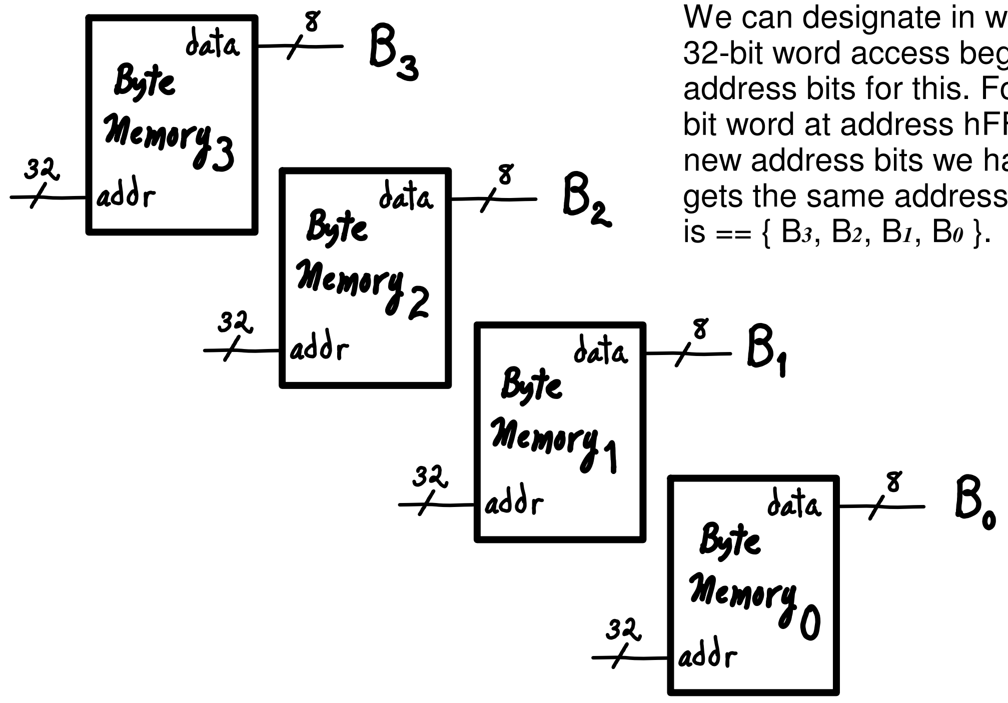


We could instead group our 32 DRAM chips into sub-words of 8-bits each (bytes). Each sub-group would act as a separate memory with 1-G of 8-bit words.





Using four of these, we can form a memory with 32-bit words, and 32-bit addresses.



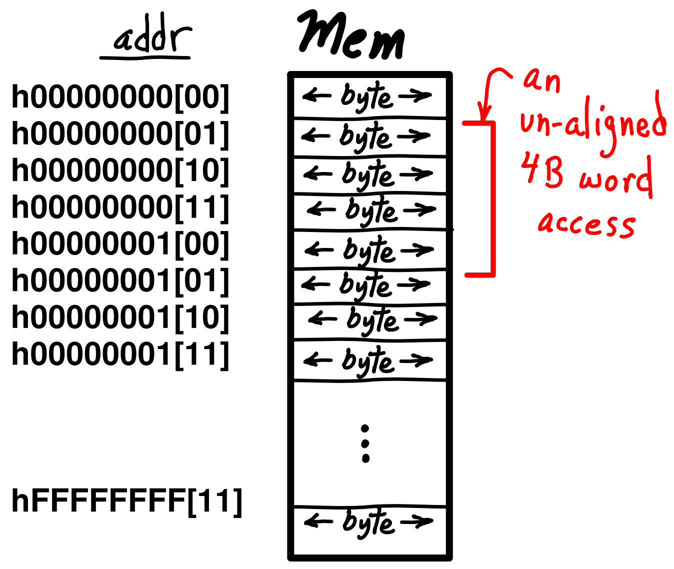
Each Byte-Memory has an independent address input. We can designate in which of the four Byte-Memories a 32-bit word access begins. We need two additional address bits for this. For instance, we can fetch the 32-bit word at address hFFFF5678[00] ( "[00]" indicates the new address bits we have added). Each Byte-Memory gets the same address, hFFFF5678, and the 32-bit word is == { $B_3$, $B_2$, $B_1$, $B_0$ }.

Suppose we instead want to fetch a word starting in Byte-Memory-1. The address would be hFFFF5678[01]. All the Byte-Memories would get the address hFFFF5678, except Byte-Memory-0 would get the address hFFFF5679.

The 32-bit word fetched would be == { $B_0$, $B_3$, $B_2$, $B_1$ }.

We can now think of the entire memory as a sequence of bytes. An access fetches (or stores) four bytes at a time. Accesses that do not have the low-order address bits == [00] are "un-aligned".

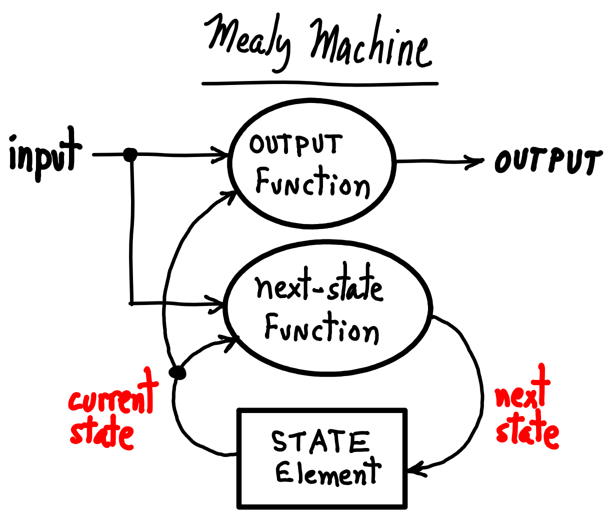
Adding +1 to the low-order bit advances the access one byte in memory: byte-addressable.

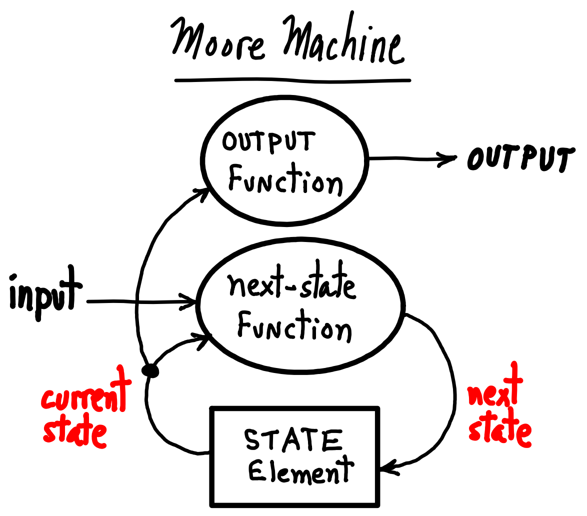


An implementation of a finite-state machine consists of four parts:

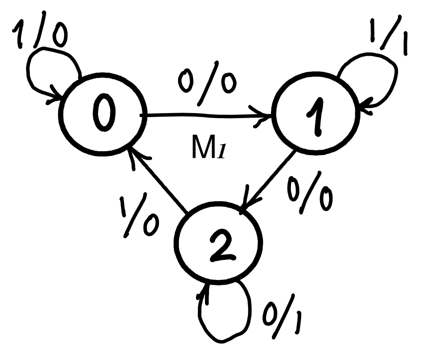
1) an input
2) a state element
3) a functional element
4) an output

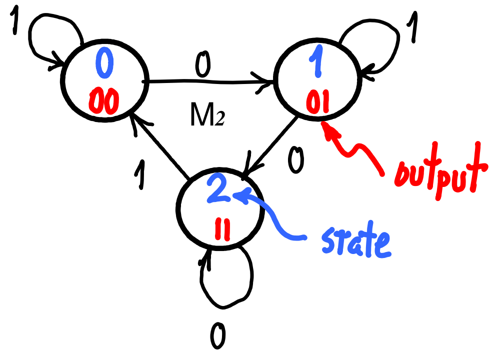
The state element implements the idea of the machine having a finite "current state". The functional part implements two functions: the next-state function and the output function. In a Mealy Machine, both depend on the input and the current state. In a Moore Machine, the output only depends on the current state.





examples





To implement a FSM, we first need to encode the states (and input symbols, if needed).

**State Encoding for M*1***

| State | Code |
|---|---|
| 0 | 00 |
| 1 | 01 |
| 2 | 10 |

**State Encoding for M*2***

| State | Code |
|---|---|
| 0 | 00 |
| 1 | 01 |
| 2 | 10 |

We next get their next-state and output functions.

**Functions for M*1***

| State | IN | Next-State | Output |
|---|---|---|---|
| 00 | 0 | 01 | 0 |
| 00 | 1 | 00 | 0 |
| 01 | 0 | 10 | 0 |
| 01 | 1 | 01 | 1 |
| 10 | 0 | 10 | 1 |
| 10 | 1 | 00 | 0 |

**Functions for M*2***

| State | IN | Next-State | Output |
|---|---|---|---|
| 00 | 0 | 01 | 00 |
| 00 | 1 | 00 | 00 |
| 01 | 0 | 10 | 01 |
| 01 | 1 | 01 | 01 |
| 10 | 0 | 10 | 11 |
| 10 | 1 | 00 | 11 |

We use minterm or maxterms to build the functions for each bit of state or output.
Notation: "-" is NOT, "*" is AND, "+" is OR.

**Logic for M*1***

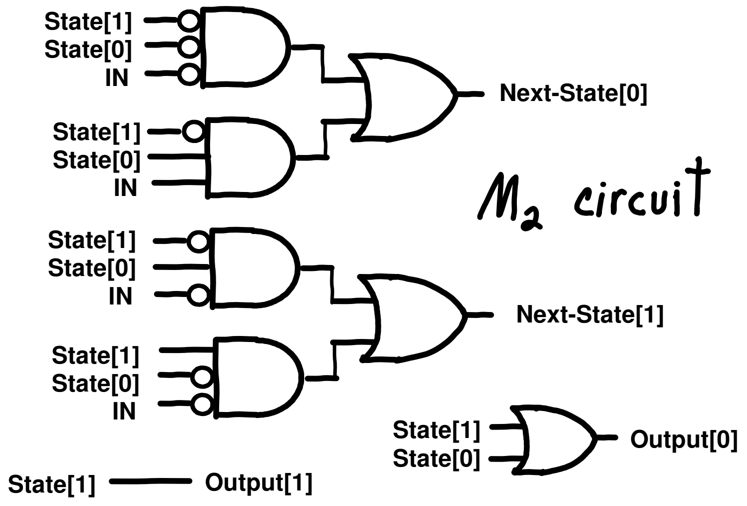
```
Next-State[0]  == (  State[1] * -State[0] * -IN )  +
                  ( -State[1] *  State[0]  * IN )

Next-State[1]  == ( -State[1] *  State[0] * -IN ) +
                  (  State[1] * -State[0] * -IN)

Output == ( -State[1] *  State[0] *  IN ) +
          (  State[1] * -State[0] * -IN )
```

**Logic for M*2***

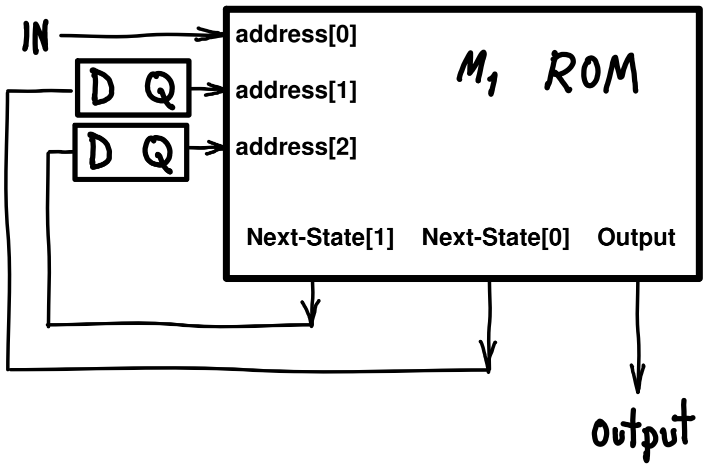
```
Next-State[0]  == ( -State[1] * -State[0] * -IN )  +
                  ( -State[1] *  State[0]  * IN )

Next-State[1]  == ( -State[1] *  State[0] * -IN ) +
                  (  State[1] * -State[0] * -IN)

Output[0] == ( State[1] + State[0] )
Output[1] == ( State[1] )
```

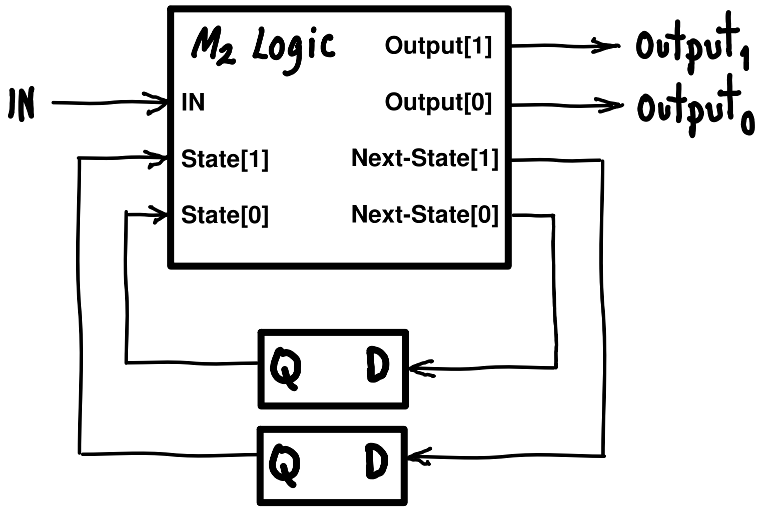
**ROM for M*1* Logic**

| address | Next-State[1] | Next-State[0] | Output |
|---|---|---|---|
| 000 | 0 | 1 | 0 |
| 001 | 0 | 0 | 0 |
| 010 | 1 | 0 | 0 |
| 011 | 0 | 1 | 1 |
| 100 | 1 | 0 | 1 |
| 101 | 0 | 0 | 0 |
| 110 | 0 | 0 | 0 |
| 111 | 0 | 0 | 0 |

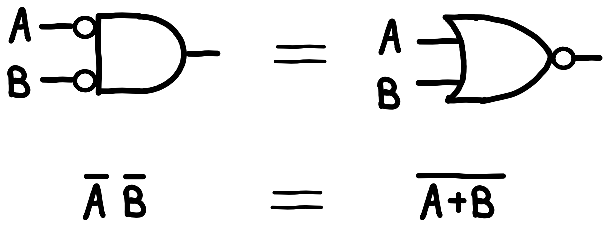


Finally, we use the ROM or the logic circuits above and connect them to our state elements (D-FFs).

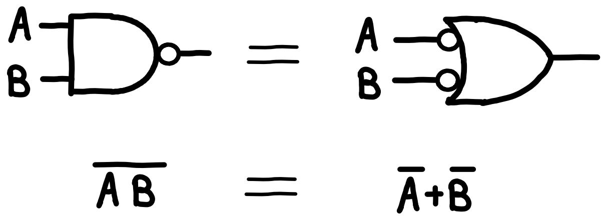




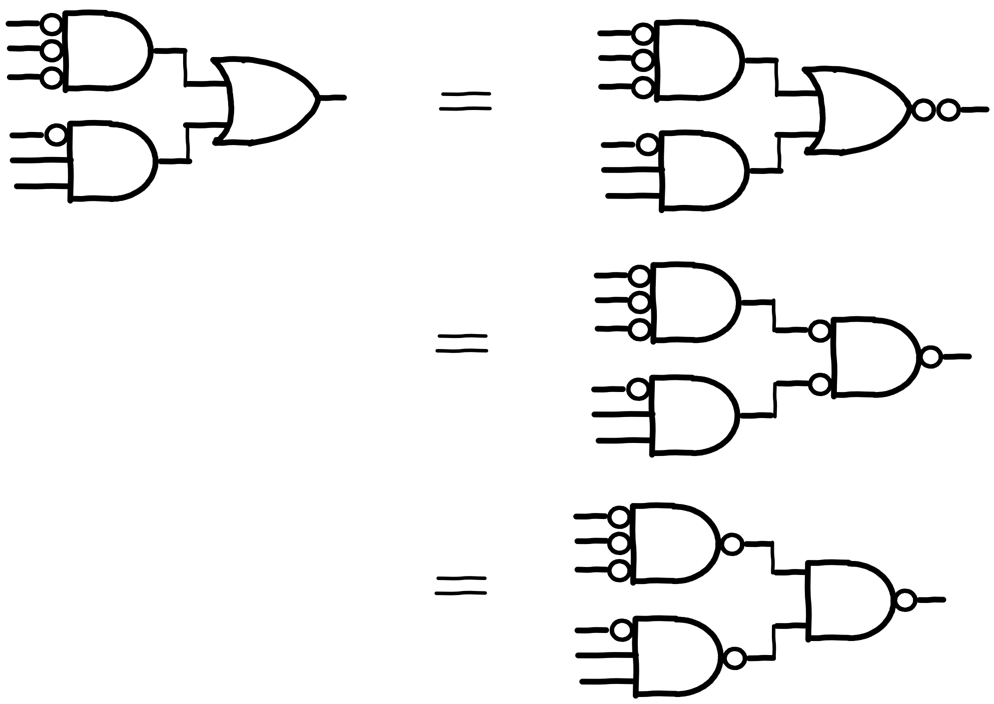
We can simplify our logic circuits by using DeMorgan's Laws and other identities of Boolean Logic.



$\bar{A}\,\bar{B} = \overline{A+B}$



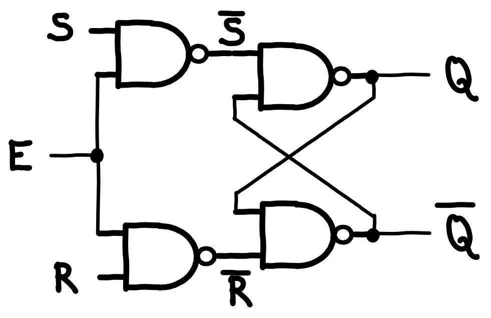
$\overline{A\,B} = \bar{A}+\bar{B}$

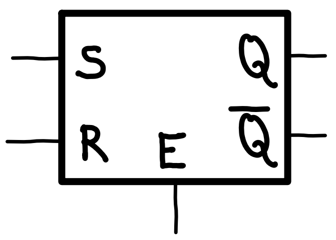


all NANDs

State elements are implemented as R-S latches.

| S | R | Q | next-Q |
|---|---|---|---|
| **1** | **1** | **0** | **0 stable** |
| **1** | **1** | **1** | **1 stable** |
| 1 | **0** | 0 | **0 reset** |
| 1 | **0** | 1 | **0 reset** |
| **0** | 1 | 0 | **1 set** |
| **0** | 1 | 1 | **1 set** |
| 0 | 0 | 0 | ? |
| 0 | 0 | 1 | ? |





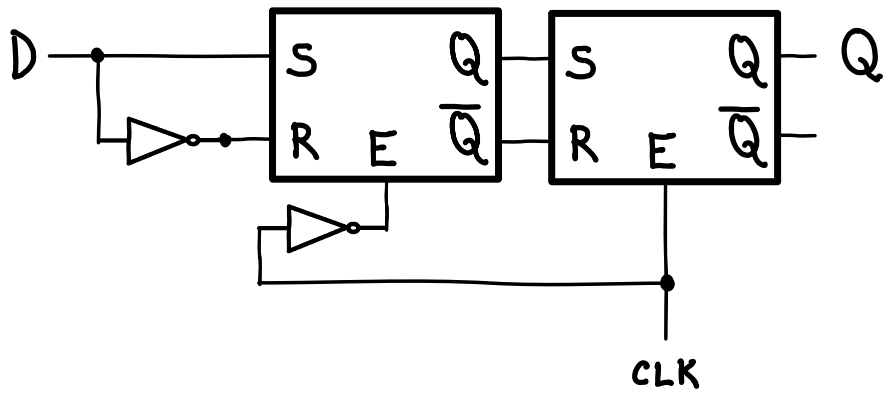
A positive-edge triggered D-flipflop stores one bit of data or state.

There is never a continuous signal path from D to Q. This prevents feedback from changing the state element before we want it to change.

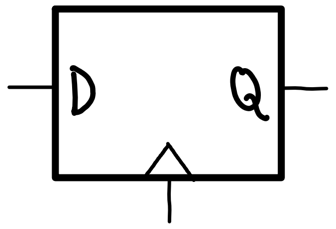
The output, Q, changes when the clock signal transitions from 0 to 1. We say it changes state on the positive edge of the clock.

We can also make it so the state changes on the negative clock edgd.

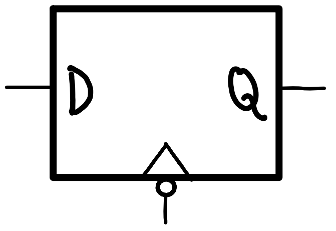
Often, we want to control whether or not the Flip-Flop will get written into on the next clock tick. For that, we add a write-enable.



Positive-edge Triggered D-Flip-Flop



Negative-edge Triggered D-Flip-Flop



Positive-edge Triggered D-Flip-Flop with write-enable