



西安交通大学
XI'AN JIAOTONG UNIVERSITY



Chapter 1

The Basics of Logic Design

Gates, Truth Tables, and Logic Equations

- The electronics inside a modern computer are digital.
- Digital electronics operate with only two voltage levels of interest: a high voltage and a low voltage.

Gates, Truth Tables, and Logic Equations

- Logic blocks are categorized as one of two types, depending on whether they contain memory.
 - Blocks without memory are called combinational; the output of a combinational block depends only on the current input.
 - In blocks with memory, the outputs can depend on both the inputs and the value stored in memory, which is called the state of the logic block.

Truth Tables

- A combinational logic block is normally given as a truth table. For a logic block with n inputs, there are 2^n entries in the truth table.

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

Logic Equations

- The OR operator is written as $+$, as in $A + B$. The result of an OR operator is 1 if either of the variables is 1. The OR operation is also called a logical sum, since its result is 1 if either operand is 1.
- The AND operator is written as \cdot , as in $A \cdot B$. The result of an AND operator is 1 only if both inputs are 1. The AND operator is also called logical product, since its result is 1 only if both operands are 1.
- The unary operator NOT is written as $!A$. The result of a NOT operator is 1 only if the input is 0.

Gates

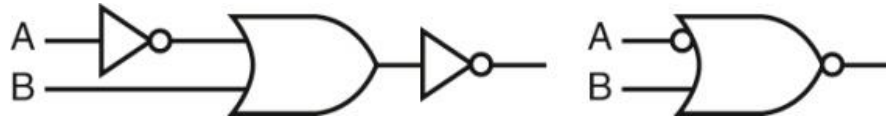
- Logic blocks are built from gates that implement basic logic functions.
 - An AND gate implements the AND function, and an OR gate implements the OR function.
 - An AND or an OR gate can have multiple inputs, with the output equal to the AND or OR of all the inputs.
 - The logical function NOT is implemented with an inverter that always has a single input.



Standard drawing for an AND gate, OR gate, and an inverter, shown from left to right.

Gates

- Rather than draw inverters explicitly, a common practice is to add “bubbles” to the inputs or outputs of a gate to cause the logic value on that input line or output line to be inverted.

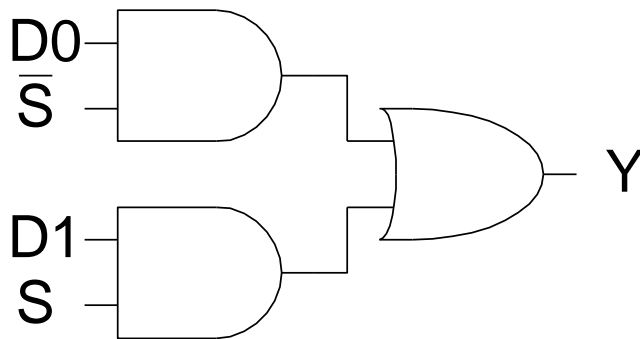


Logic gate implementation of $A + B$ using explicit inverters on the left and bubbled inputs and outputs on the right. This logic function can be simplified to $A B$ or in Verilog, $A \& \sim B$.

Gates

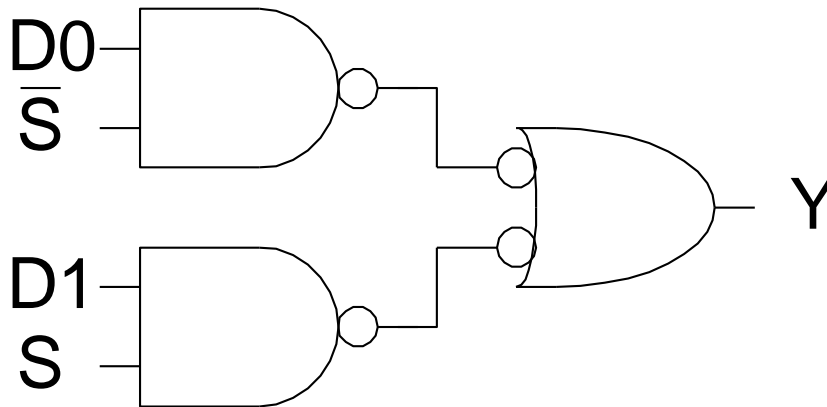
```
module mux(input s, d0, d1,  
           output y);  
  
    assign y = s ? d1 : d0;  
endmodule
```

1) 用AND, OR, 和 NOT 门画出以上语句描述的设计.



Gates

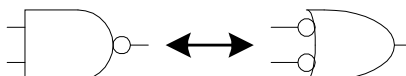
2) 用NAND, NOR, 和 NOT门再次画出前一设计. 假定 $\sim S$ 可得到.

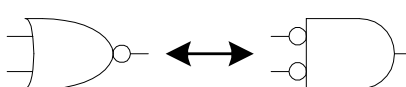


Bubble Pushing (推气泡)

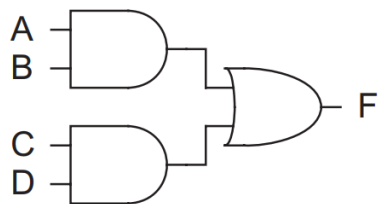
- 从 AND / OR 门网络开始
- 可转为等价的 NAND / NOR + 反相器
- 拓展到简单逻辑

- DeMorgan's 定理

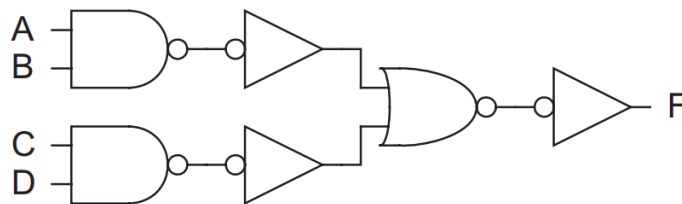
$$!(AB) = !A + !B$$


$$!(A+B) = !A !B$$


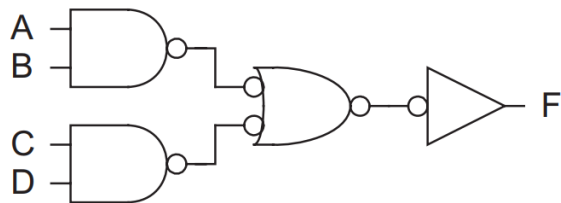
- 采用与非门和或非门的电路计算 $Y = AB + CD$



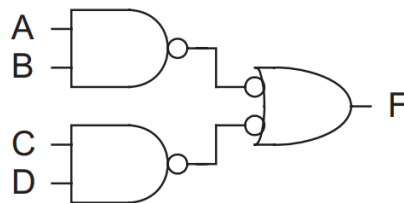
(a)



(b)



(c)



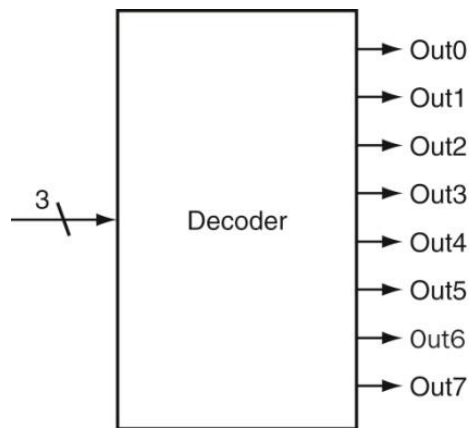
(d)



Combinational Logic

Decoders

- The most common type of decoder has an n -bit input and 2^n outputs, where only one output is asserted for each input combination.
- If the value of the input is i , then Out_i will be true and all other outputs will be false.



a. A 3-bit decoder

Inputs			Outputs							
12	11	10	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

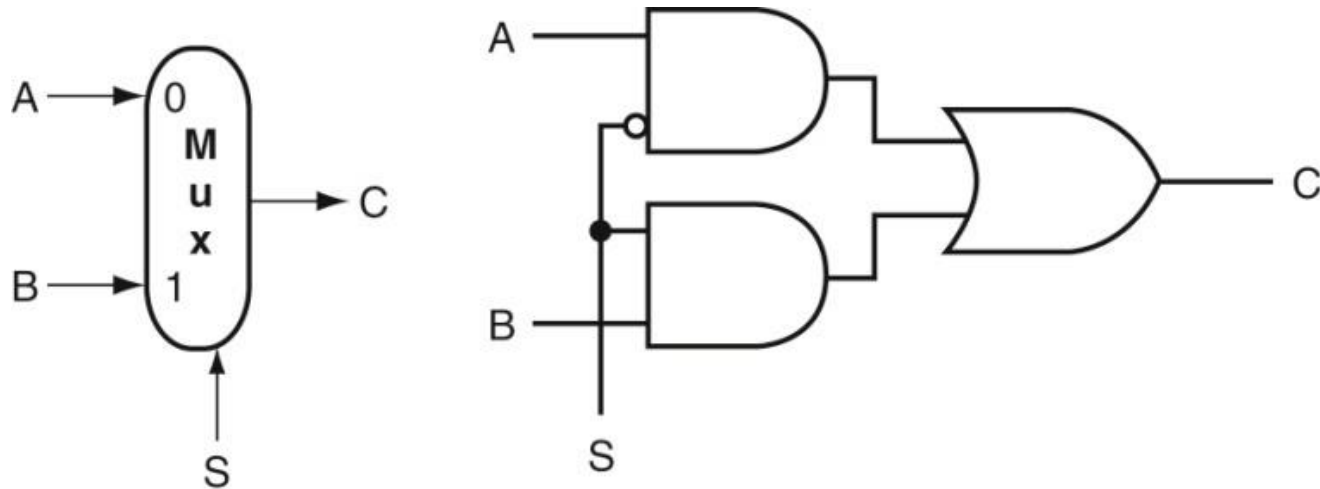
b. The truth table for a 3-bit decoder

A 3-bit decoder has three inputs, called 12, 11, and 10, and $2^3 = 8$ outputs, called Out0 to Out7. Only the output corresponding to the binary value of the input is true, as shown in the truth table. The label 3 on the input to the decoder says that the input signal is 3 bits wide.

Combinational Logic

■ Multiplexors

- A multiplexor might more properly be called a selector, since its output is one of the inputs that is selected by a control.



A two-input multiplexor on the left and its implementation with gates on the right. The multiplexor has two data inputs (*A* and *B*), which are labeled 0 and 1, and one selector input (*S*), as well as an output *C*. Implementing multiplexors in Verilog requires a little more work, especially when they are wider than two inputs. We show how to do this beginning on page A-23.

Combinational Logic

- Two-Level Logic and PLAs
 - Any logic function can be implemented with only AND, OR, and NOT functions.
 - In fact, any logic function can be written in a canonical form, where every input is either a true or complemented variable and there are only two levels of gates—one being AND and the other OR—with a possible inversion on the final output.
 - Such a representation is called a two-level representation, and there are two forms, called sum of products and product of sums.
 - A sum-of-products representation is a logical sum (OR) of products (terms using the AND operator); a product of sums is just the opposite.

Combinational Logic

- Two-Level Logic and PLAs
 - We had two equations for the output E:

$$E = ((A \cdot B) + (A \cdot C) + (B \cdot C)) \cdot \overline{(A \cdot B \cdot C)}$$

$$E = (A \cdot B \cdot \bar{C}) + (A \cdot C \cdot \bar{B}) + (B \cdot C \cdot \bar{A})$$

- This second equation is in a sum-of-products form: it has two levels of logic and the only inversions are on individual variables.
- The first equation has three levels of logic.

Combinational Logic

- Two-Level Logic and PLAs
 - Show the sum-of-products representation for the following truth table for D.

Inputs		Outputs	
A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

- Thus, we can write the function for D as the sum of these terms:

$$D = (\bar{A} \cdot \bar{B} \cdot C) + (\bar{A} \cdot B \cdot \bar{C}) + (A \cdot \bar{B} \cdot \bar{C}) + (A \cdot B \cdot C)$$

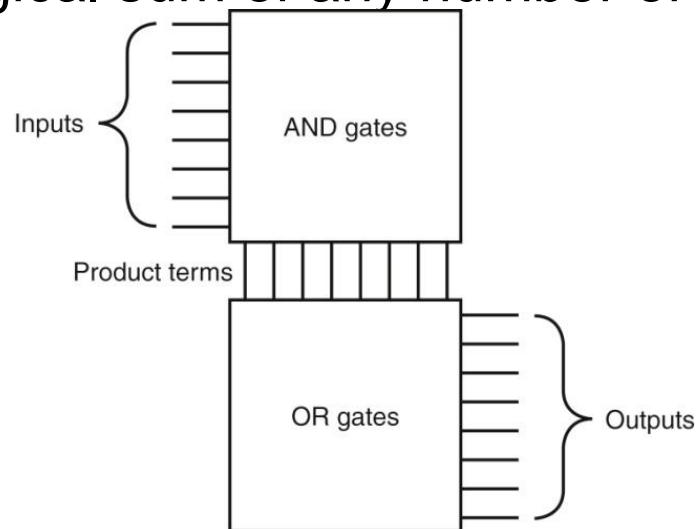
Combinational Logic

- Two-Level Logic and PLAs
 - We can use this relationship between a truth table and a two-level representation to generate a gate-level implementation of any set of logic functions.
 - The sum-of-products representation corresponds to a common structured-logic implementation called a programmable logic array (PLA).
 - A PLA has a set of inputs and corresponding input complements (which can be implemented with a set of inverters), and two stages of logic.

Combinational Logic

■ Two-Level Logic and PLAs

- The first stage is an array of AND gates that form a set of product terms (sometimes called minterms); each product term can consist of any of the inputs or their complements.
- The second stage is an array of OR gates, each of which forms a logical sum of any number of the product terms.



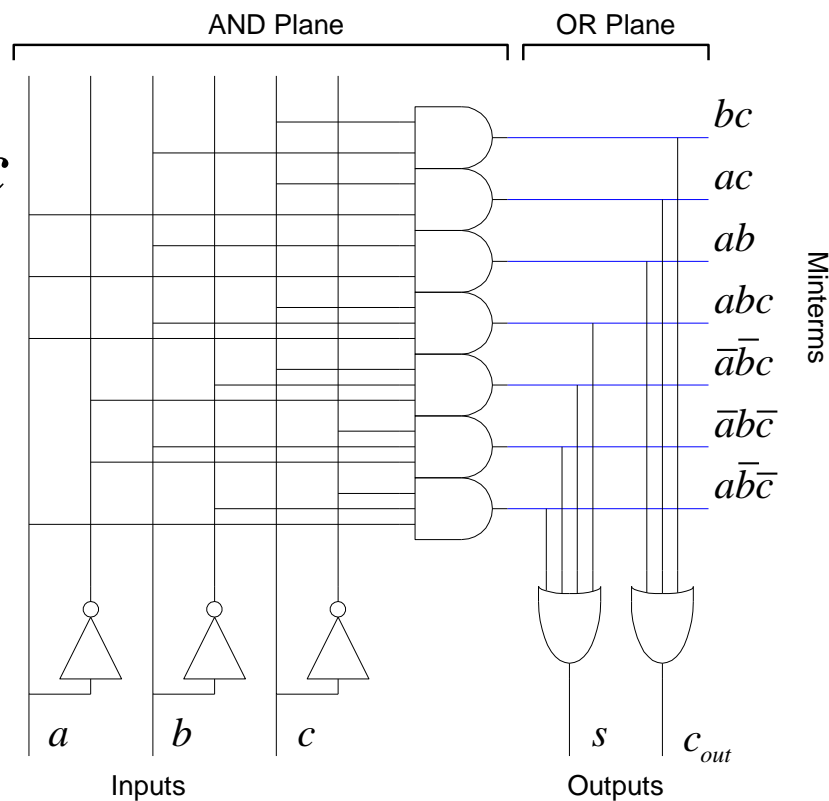
The basic form of a PLA consists of an array of AND gates followed by an array of OR gates. Each entry in the AND gate array is a product term consisting of any number of inputs or inverted inputs. Each entry in the OR gate array is a sum term consisting of any number of these product terms.

Combinational Logic

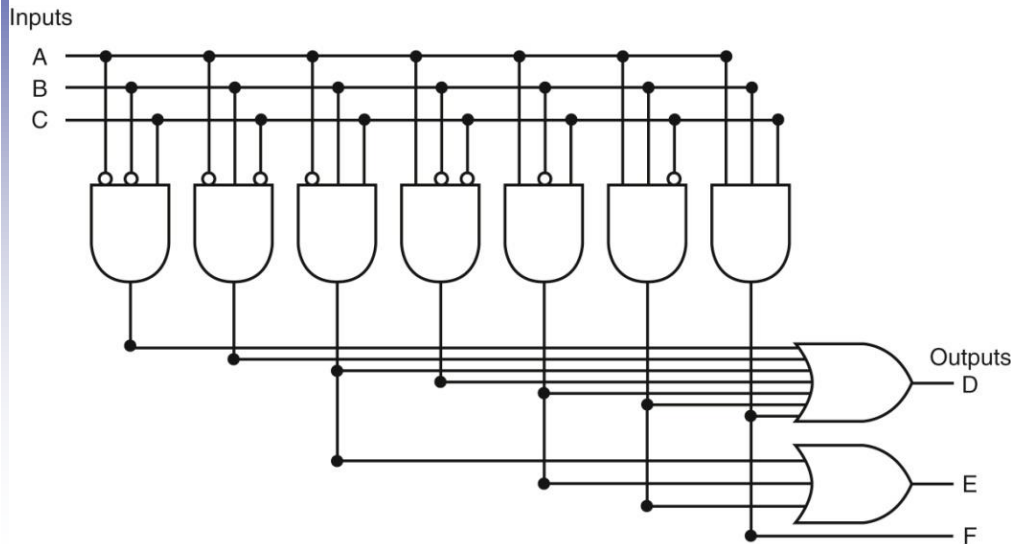
■ Ex: Full Adder

$$s = a\bar{b}\bar{c} + \bar{a}b\bar{c} + \bar{a}\bar{b}c + abc$$

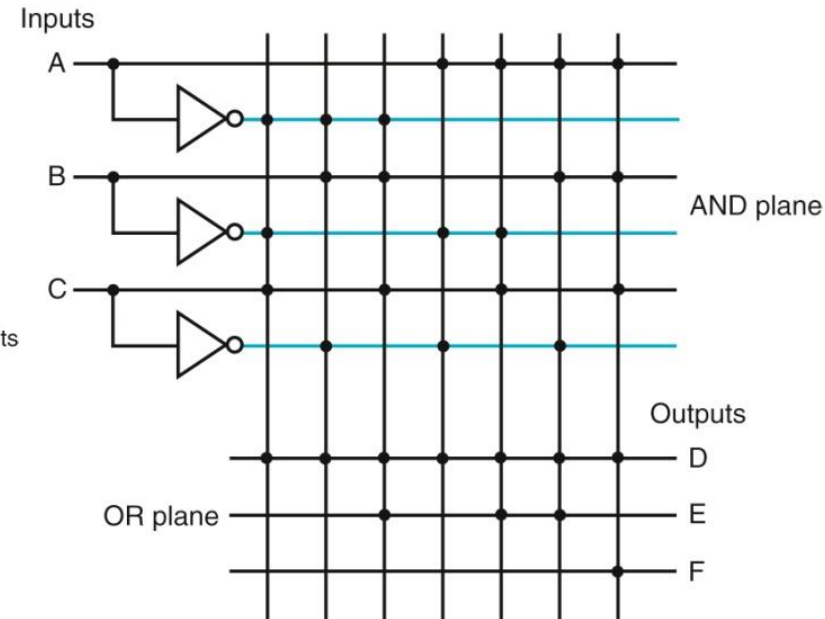
$$c_{out} = ab + bc + ac$$



Combinational Logic



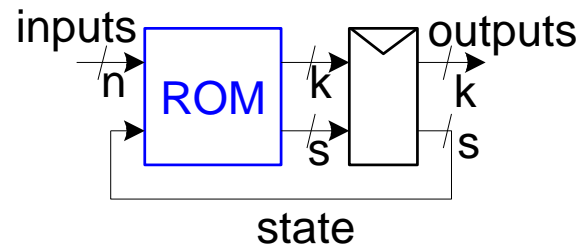
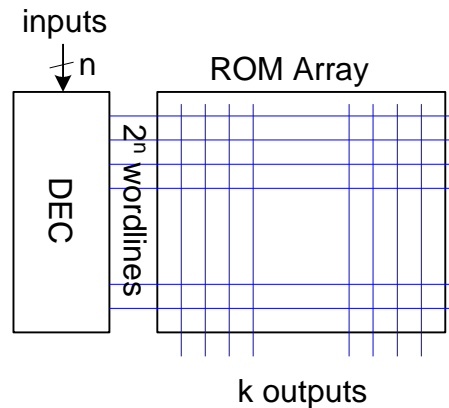
The PLA for implementing the logic function described in the example.



A PLA drawn using dots to indicate the components of the product terms and sum terms in the array. Rather than use inverters on the gates, usually all the inputs are run the width of the AND plane in both true and complement forms. A dot in the AND plane indicates that the input, or its inverse, occurs in the product term. A dot in the OR plane indicates that the corresponding product term appears in the corresponding output.

Combinational Logic

- 把ROM用作包含真值表的查找表
 - n 输入, k 输出 需要 2^n words x k bits
 - 改变功能比较简单 – 重编程 ROM
- 有限状态机
 - n 输入, k 输出, s bits of state
 - Build with 2^{n+s} x $(k+s)$ bit ROM and $(k+s)$ bit reg



PLAs vs. ROMs

- PLA的OR平面类似 ROM阵列
- PLA的AND平面类似 ROM译码器
- PLAs 比ROMs 更灵活
 - No need to have 2^n rows for n inputs
 - Only generate the minterms that are needed
 - Take advantage of logic simplification

Example: RoboAnt

Let's build an Ant

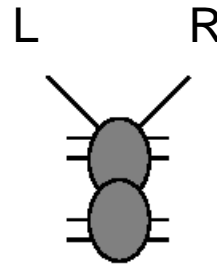
Sensors: Antennae

(L,R) – 1 when in contact

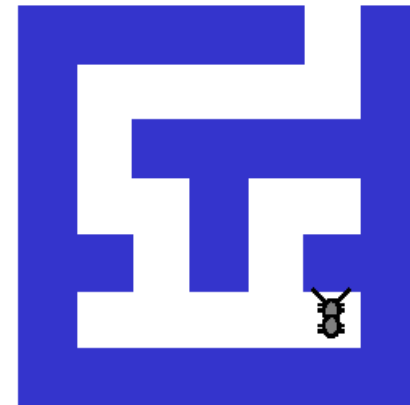
Actuators: Legs

Forward step F

Ten degree turns TL, TR



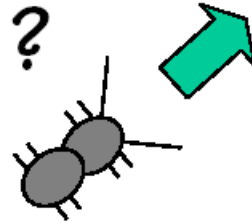
Goal: make our ant smart enough to get out of a maze



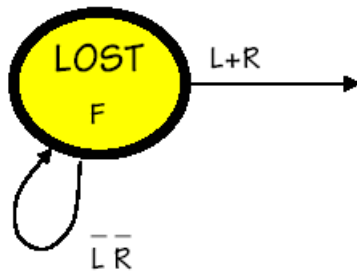
Strategy: keep right antenna on wall

(RoboAnt adapted from MIT 6.004 2002 OpenCourseWare by Ward and Terman)

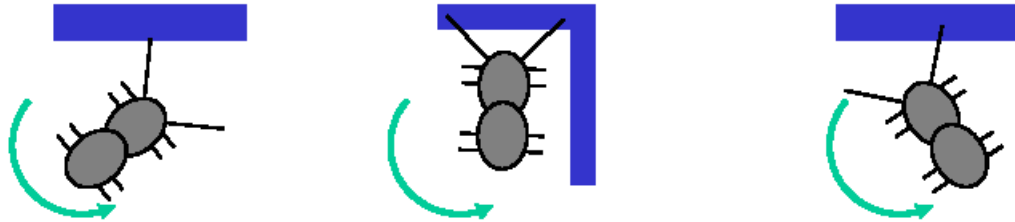
Lost in space



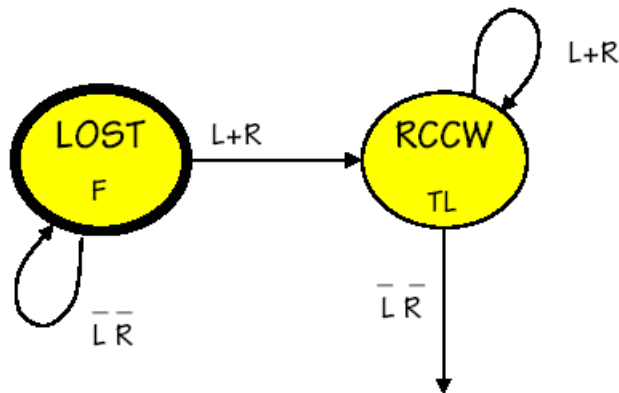
- Action: go forward until we hit something
 - Initial state



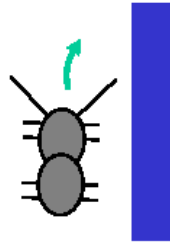
Bonk!!!



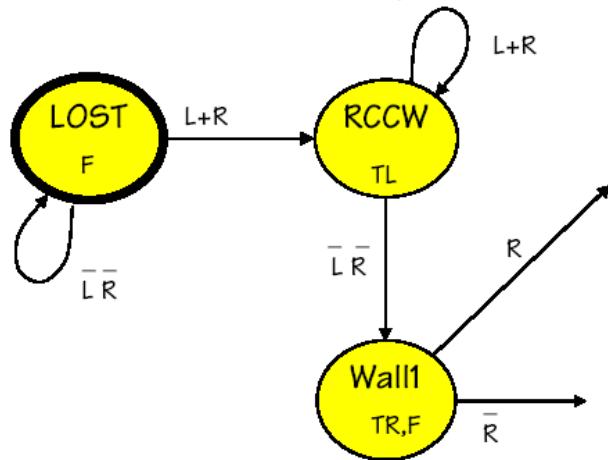
- Action: turn left (rotate counterclockwise)
 - Until we don't touch anymore



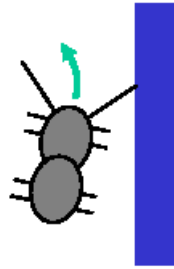
A little to the right



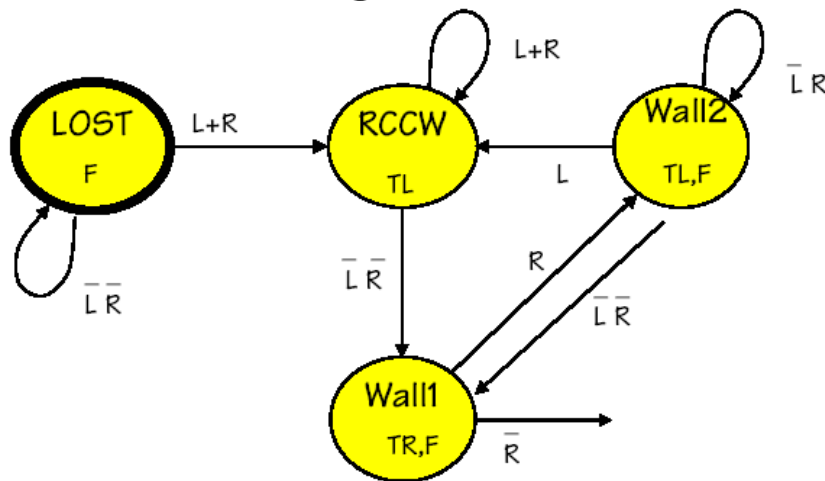
- Action: step forward and turn right a little
 - Looking for wall



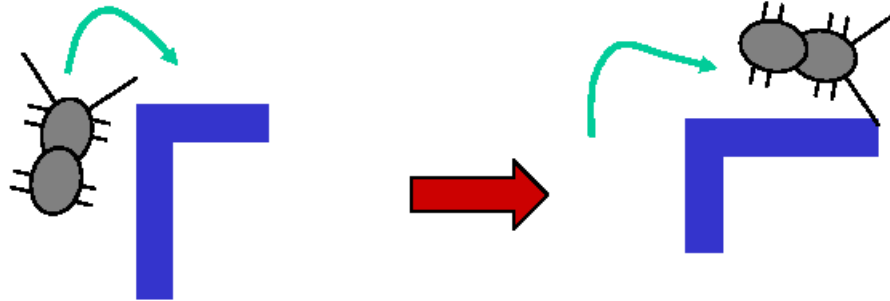
Then a little to the left



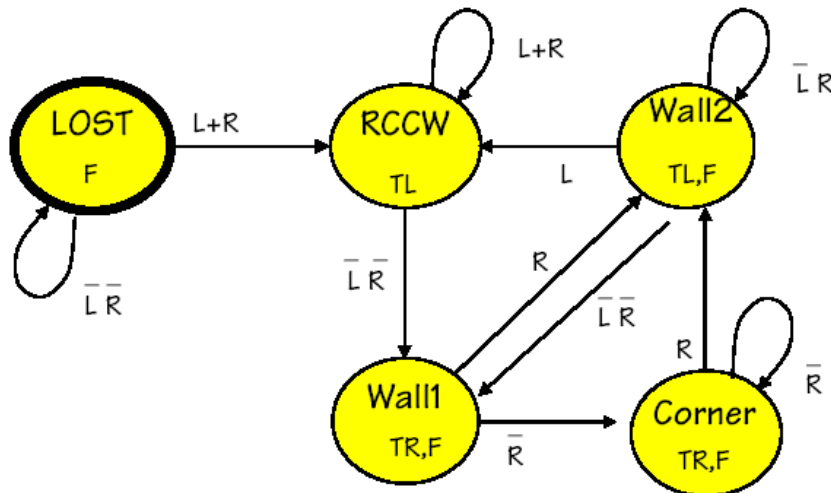
- Action: step and turn left a little, until not touching



Whoops – a corner!

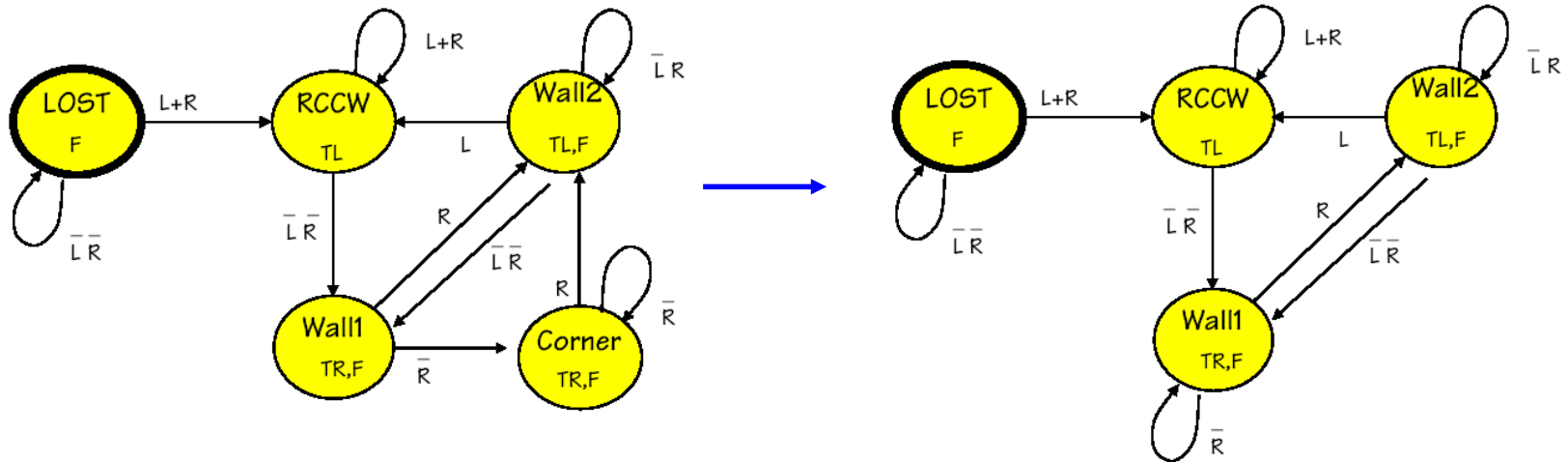


- Action: step and turn right until hitting next wall



Simplification

- Merge equivalent states where possible

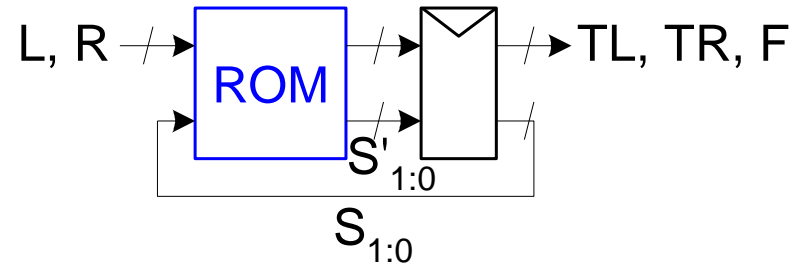
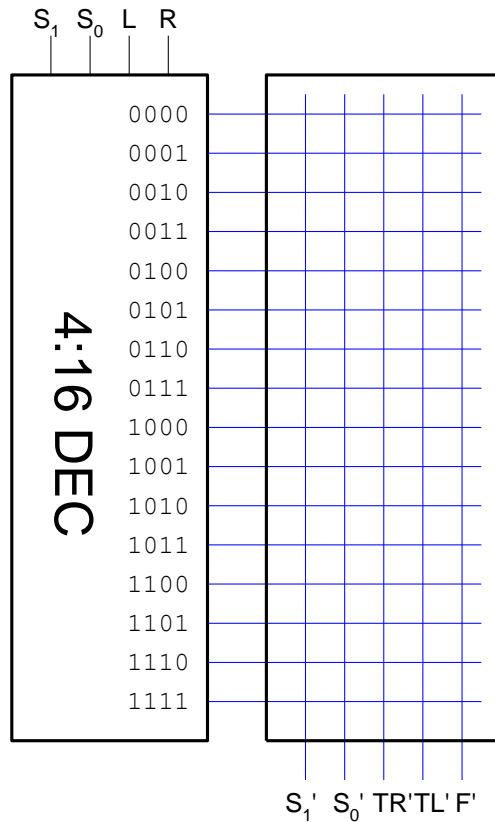


State Transition Table

	$S_{1:0}$	L	R	$S_{1:0}'$	TR	TL	F
Lost	00	0	0	00	0	0	1
	00	1	X	01	0	0	1
	00	0	1	01	0	0	1
RCCW	01	1	X	01	0	1	0
	01	0	1	01	0	1	0
	01	0	0	10	0	1	0
Wall1	10	X	0	10	1	0	1
	10	X	1	11	1	0	1
Wall2	11	1	X	01	0	1	1
	11	0	0	10	0	1	1
	11	0	1	11	0	1	1

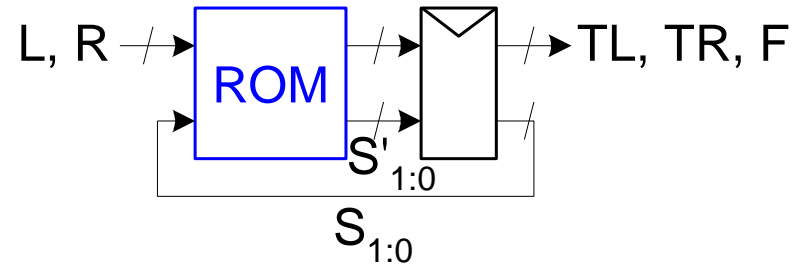
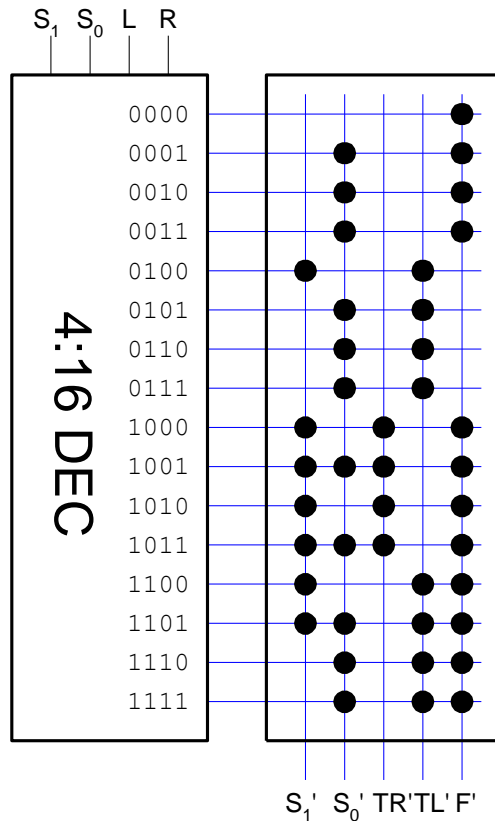
ROM Implementation

■ 16-word x 5 bit ROM



ROM Implementation

■ 16-word x 5 bit ROM



例: RoboAnt PLA

- 将转态转换表转为逻辑方程

$S_{1:0}$	L	R	$S_{1:0}'$	TR	TL	F
00	0	0	00	0	0	1
00	1	X	01	0	0	1
00	0	1	01	0	0	1
01	1	X	01	0	1	0
01	0	1	01	0	1	0
01	0	0	10	0	1	0
10	X	0	10	1	0	1
10	X	1	11	1	0	1
11	1	X	01	0	1	1
11	0	0	10	0	1	1
11	0	1	11	0	1	1

$$\begin{array}{c}
 S1' \\
 \begin{array}{ccccc}
 & & S_1 S_0 & & \\
 & & 00 & 01 & 11 & 10 \\
 00 & 0 & 1 & 1 & 1 & \\
 LR & 01 & 0 & 0 & 1 & 1 \\
 & 11 & 0 & 0 & 0 & 1 \\
 & 10 & 0 & 0 & 0 & 1
 \end{array} \\
 S_1' = S_1 \overline{S_0} + \overline{L} S_1 + \overline{L} R S_0
 \end{array}$$

$$\begin{array}{c}
 S0' \\
 \begin{array}{ccccc}
 & & S_1 S_0 & & \\
 & & 00 & 01 & 11 & 10 \\
 00 & 0 & 0 & 0 & 0 & 0 \\
 LR & 01 & 1 & 1 & 1 & 1 \\
 & 11 & 1 & 1 & 1 & 1 \\
 & 10 & 1 & 1 & 1 & 0
 \end{array} \\
 S_0' = R + \overline{L} S_1 + L S_0 \\
 TR = S_1 \overline{S_0} \\
 TL = S_0 \\
 F = S_1 + \overline{S_0}
 \end{array}$$



RoboAnt 点图

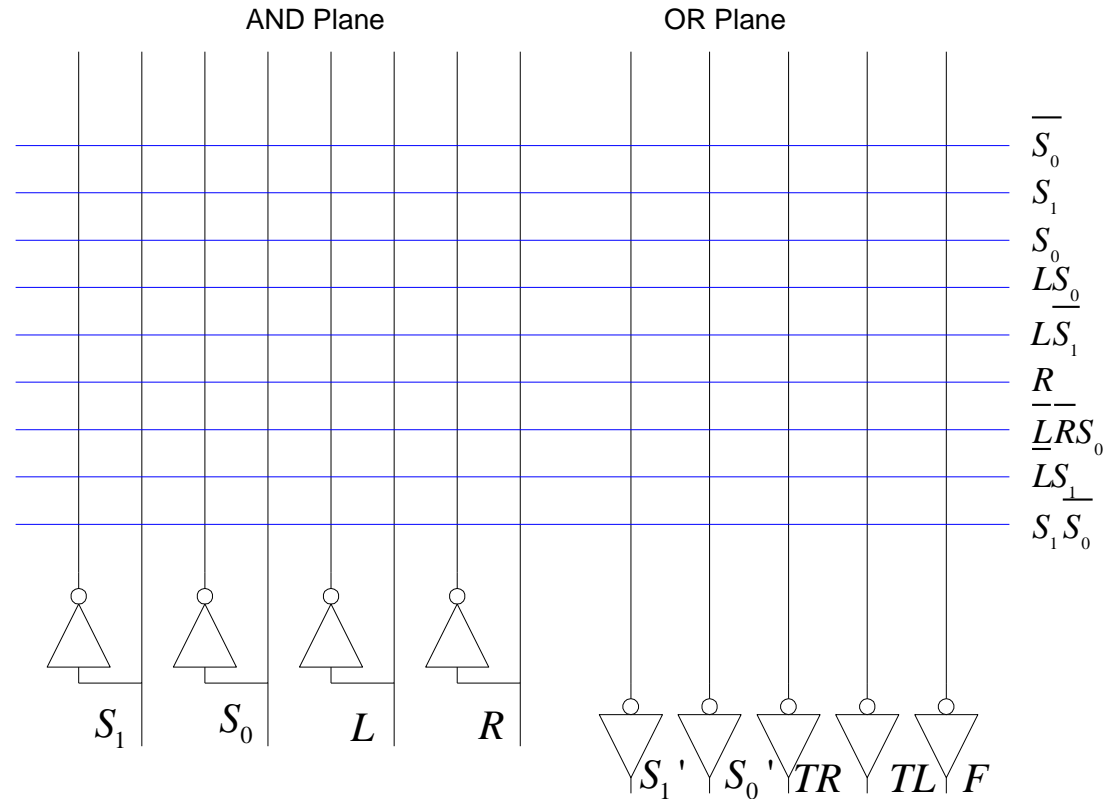
$$S1' = S_1 \overline{S_0} + \overline{L} S_1 + \overline{L} \overline{R} S_0$$

$$S0' = R + L \overline{S_1} + L S_0$$

$$TR = S_1 \overline{S_0}$$

$$TL = S_0$$

$$F = S_1 + \overline{S_0}$$



Combinational Logic

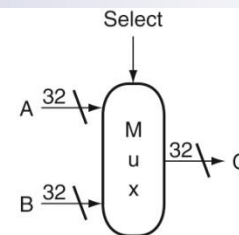
■ Don't Cares

- There are two types of don't cares: output don't cares and input don't cares.
- Output don't cares arise when we don't care about the value of an output for some input combination. They appear as Xs in the output portion of a truth table. When an output is a don't care for some input combination, the designer or logic optimization program is free to make the output true or false for that input combination.
- Input don't cares arise when an output depends on only some of the inputs, and they are also shown as Xs, though in the input portion of the truth table.

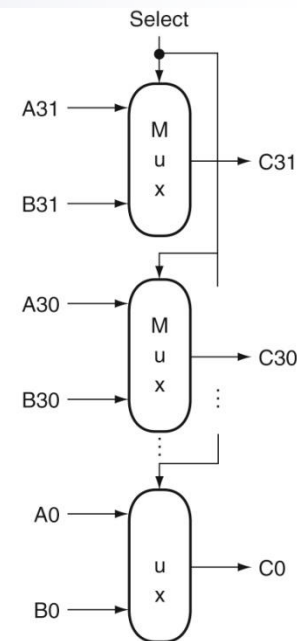
Combinational Logic

■ Arrays of Logic Elements

- Many of the combinational operations to be performed on data have to be done to an entire word (64 bits) of data.
- In the RISC-V instruction set, the result of an instruction that is written into a register can come from one of two sources.
- A multiplexor is used to choose which of the two buses (each 32 bits wide) will be written into the Result register.
- The 1-bit multiplexor, which we showed earlier, will need to be replicated 32 times.



a. A 32-bit wide 2-to-1 multiplexor



b. The 32-bit wide multiplexor is actually an array of 32 1-bit multiplexors

A multiplexor is arrayed 64 times to perform a selection between two 64-bit inputs. Note that there is still only one data selection signal used for all 32 1-bit multiplexors.

Hardware Description Language

- Datatypes and Operators in Verilog
 - A wire specifies a combinational signal.
 - A reg (register) holds a value, which can vary with time. A reg need not necessarily correspond to an actual register in an implementation.
 - A register or wire, named X, that is 32 bits wide is declared as an array:
 - reg [31:0] X or wire [31:0] X
 - An array of registers is used for a structure like a register file or memory. Thus, the declaration
 - reg [31:0] registerfile[0:31]

Hardware Description Language

- Datatypes and Operators in Verilog
- Verilog provides the full set of unary and binary operators from C, including the arithmetic operators (+, -, *, /), the logical operators (&, |, ~), the comparison operators (==, !=, >, <, <=, >=), the shift operators (<<, >>), and C's conditional operator (?), which is used in the form condition ? expr1 :expr2 and returns expr1 if the condition is true and expr2 if it is false).
- Verilog adds a set of unary logic reduction operators (&, |, ^) that yield a single bit by applying the logical operator to all the bits of an operand. For example, &A returns the value obtained by ANDing all the bits of A together, and ^A returns the reduction obtained by using exclusive OR on all the bits of A.

Hardware Description Language

■ Structure of a Verilog Program

- A Verilog program is structured as a set of modules, which may represent anything from a collection of logic gates to a complete system. A module specifies its input and output ports, which describe the incoming and outgoing connections of a module. A module may also declare additional variables. The body of a module consists of:
 - ✓ initial constructs, which can initialize reg variables
 - ✓ Continuous assignments, which define only combinational logic
 - ✓ always constructs, which can define either sequential or combinational logic
 - ✓ Instances of other modules, which are used to implement the module being defined

Hardware Description Language

- A continuous assignment, which is indicated with the keyword `assign`, acts like a combinational logic function: the output is continuously assigned the value, and a change in the input values is reflected immediately in the output value. Wires may only be assigned values with continuous assignments. Using continuous assignments, we can define a module that implements a half-adder.

```
module half_adder (A,B,Sum,Carry);  
    input A,B; //two 1-bit inputs  
    output Sum, Carry; //two 1-bit outputs  
    assign Sum = A ^ B; //sum is A xor B  
    assign Carry = A & B; //Carry is A and B  
endmodule
```

A Verilog module that defines a half-adder using continuous assignments.

Hardware Description Language

- An always block specifies an optional list of signals on which the block is sensitive (in a list starting with @). The always block is re-evaluated if any of the listed signals changes value; if the list is omitted, the always block is constantly reevaluated.

```
module Mult4to1 (In1,In2,In3,In4,Sel,Out);  
    input [31:0] In1, In2, In3, In4; //four 32-bit inputs  
    input [1:0] Sel; //selector signal  
    output reg [31:0] Out; //32-bit output  
    always @(In1, In2, In3, In4, Sel)  
    case (Sel) // a 4->1 multiplexor  
        0: Out <= In1;  
        1: Out <= In2;  
        2: Out <= In3;  
        default: Out <= In4;  
    endcase  
endmodule
```

A Verilog definition of a 4-to-1 multiplexor with 32-bit inputs, using a case statement. The case statement acts like a C switch statement, except that in Verilog only the code associated with the selected case is executed (as if each case state had a break at the end) and there is no fall-through to the next statement.

Hardware Description Language

- The below shows a definition of a RISC-V ALU, which uses a case statement.

```
module RISCVALU (ALUctl, A, B, ALUOut, Zero);
    input [3:0] ALUctl;
    input [31:0] A,B;
    output reg [31:0] ALUOut;
    output Zero;
    assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0; goes anywhere
    always @(ALUctl, A, B) //reevaluate if these change
        case (ALUctl)
            0: ALUOut <= A & B;
            1: ALUOut <= A | B;
            2: ALUOut <= A + B;
            6: ALUOut <= A - B;
            7: ALUOut <= A < B ? 1:0;
            12: ALUOut <= ~(A | B); // result is nor
            default: ALUOut <= 0; //default to 0, should not happen;
        endcase
    endmodule
```

A Verilog behavioral definition of a RISC-V ALU. This could be synthesized using a module library containing basic arithmetic and logical operations.