2.4 Logic Minimization and Karnaugh Maps

As we found above, given a truth table, it is always possible to write down a correct logic expression simply by forming an `OR` of the `AND`s of all input variables for which the output is true ($Q = 1$). However, for an arbitrary truth table such a procedure could produce a very lengthy and cumbersome expression which might be needlessly inefficient to implement with gates.

There are several methods for simplification of Boolean logic expressions. The process is usually called "logic minimization", and the goal is to form a result which is efficient. Two methods we will discuss are algebraic minimization and Karnaugh maps. For very complicated problems the former method can be done using special software analysis programs. Karnaugh maps are also limited to problems with up to 4 binary inputs.

Let's start with a simple example. The table below gives an arbitrary truth table involving 2 logic inputs.

Table 1: Example of simple arbitrary truth table.

A	B	Q
0	0	1
0	1	1
1	0	0
1	1	1

There are two overall stategies:

1. Write down an expression directly from the truth table. Use Boolean algebra, if desired, to simplify.
2. Use Karnaugh mapping ("K-map"). This is only applicable if there are ≤ 4 inputs.

In our example above, we can use two different ways of writin down a result directly from the truth table. We can write down all `TRUE` terms and `OR` the result. This gives

$$Q = \bar{A}\bar{B} + \bar{A}B + AB$$

While correct, without further simplification this expression would involve 3 2-input `AND` gates, 2 inverters, and 1 3-input `OR` gate.

Alternatively, one can write down an expression for all of the `FALSE` states of the truth table. This is simpler in this case:

$$\overline{Q} = A\bar{B} \quad \rightarrow Q = \overline{A\bar{B}} = \bar{A} + B$$

where the last step results from Eqn. 3. Presumably, the two expressions can be found to be equivalent with some algebra. Certainly, the 2nd is simpler, and involves only an inverter and one 2-input `OR` gate.

Finally, one can try a K-map solution. The first step is to write out the truth table in the form below, with the input states the headings of rows and columns of a table, and the corresponding outputs within, as shown below.

Table 2: K-map of truth table.

$A\backslash B$	0	1
0	1	1
1	0	1

The steps/rules are as follows:

1. Form the 2-dimensional table as above. Combine 2 inputs in a "gray code" way – see 2nd example below.
2. Form groups of 1's and circle them; the groups are rectangular and must have sides of length $2^n \times 2^m$, where n and m are integers $0, 1, 2, \ldots$.
3. The groups can overlap.
4. Write down an expression of the inputs for each group.
5. `OR` together these expressions. That's it.
6. Groups can wrap across table edges.
7. As before, one can alternatively form groups of 0's to give a solution for $\overline{Q}$.
8. The bigger the groups one can form, the better (simpler) the result.
9. There are usually many alternative solutions, all equivalent, some better than others depending upon what one is trying to optimize.

Here is one way of doing it:

$A\backslash B$	0	1
0	1	1
1	0	1

The two groups we have drawn are $\bar{A}$ and B. So the solution (as before) is:

$$Q = \bar{A} + B$$

2.4.1 K-map Example 2

Let's use this to determine which 3-bit numbers are prime. (This is a homework problem.) We assume that $0, 1, 2$ are *not* prime. We will let our input number have digits $a_2a_1a_0$. Here is the truth table:

Here is the corresponding K-map and a solution.

Note that where two inputs are combined in a row or column that their progression follows gray code, that is only one bit changes at a time. The solution shown above is:

$$Q = a_1a_0 + a_2a_0 = a_0(a_1 + a_2)$$

Table 3: 3-digit prime finder.

Decimal	a_2	a_1	a_0	Q
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

Table 4: K-map of truth table.

$a_2 \backslash a_1\, a_0$	00	01	11	10
0	0	0	1	0
1	0	1	1	0

2.4.2 K-map Example 3: Full Adder

In this example we will outline how to build a digital *full adder*. It is called "full" because it will include a "carry-in" bit and a "carry-out" bit. The carry bits will allow a succession of 1-bit full adders to be used to add binary numbers of arbitrary length. (A *half adder* includes only one carry bit.)

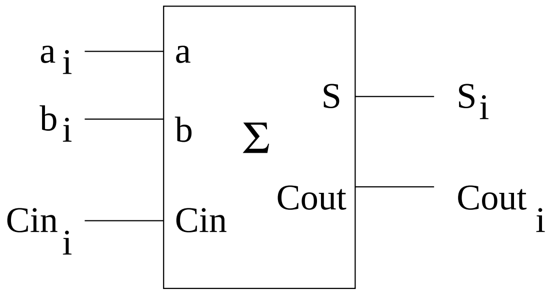

Figure 7: Block schematic of full adder. (We name our adder the "Σ chip").

The scheme for the full adder is outlined in Fig. 7. Imagine that we are adding two n-bit binary numbers. Let the inputs a_i and b_i be the i-th bits of the two numbers. The carry in bit $C\text{in}_i$ represents any carry from the sum of the neighboring less significant bits at position $i-1$. That is, $C\text{in}_i = 1$ if $a_{i-1} = b_{i-1} = 1$, and is 0 otherwise. The sum S_i at position i is therefore the sum of a_i, b_i, and $C\text{in}_i$. (Note that this is an arithmetic sum, *not* a Boolean `OR`.) A carry for this sum sets the carry out bit, $C\text{out}_i = 1$, which then can be applied to the sum of the $i+1$ bits. The truth table is given below.

$C\text{in}_i$	a_i	b_i	S_i	$C\text{out}_i$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

With $C\text{in}_i = 0$, we see that the output sum S_i is just given by the `XOR` operation, $a_i \oplus b_i$. And with $C\text{in}_i = 1$, then $S_i = \overline{a_i \oplus b_i}$. Perhaps the simplest way to express this relationship is the following:

$$S_i = C\text{in}_i \oplus (a_i \oplus b_i)$$

To determine a relatively simple expression for $C\text{out}_i$, we will use a K-map:

$C\text{in}_i \backslash a_i b_i$	00	01	11	10
0	0	0	1	0
1	0	1	1	1

This yields

$$Cout_i = a_i b_i + Cin_i a_i + Cin_i b_i = a_i b_i + Cin_i(a_i + b_i)$$

which in hardware would be 2 2-input `OR` gates and 2 2-input `AND` gates.

As stated above, the carry bits allow our adder to be expanded to add any number of bits. As an example, a 4-bit adder circuit is depicted in Fig. 8. The sum can be 5 bits, where the MSB is formed by the final carry out. (Sometimes this is referred to as an "overflow" bit.)

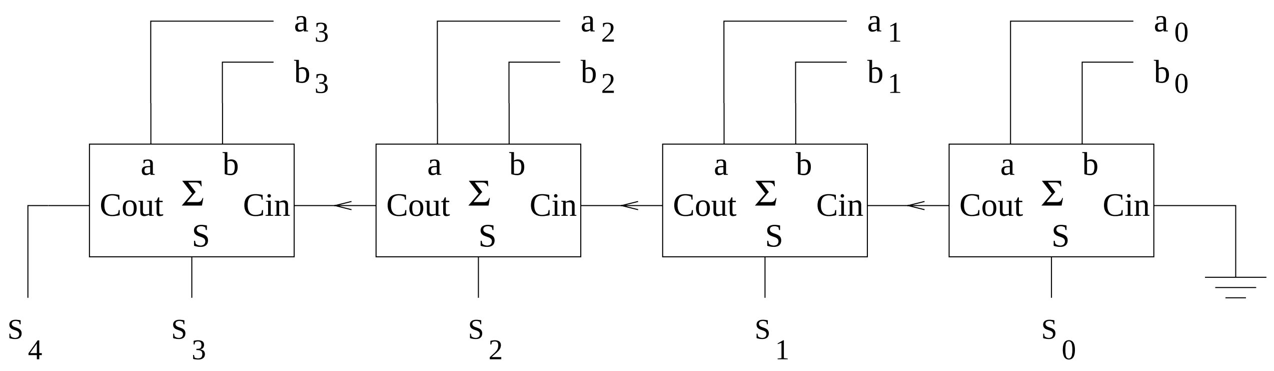

Figure 8: Expansion of 1-bit full adder to make a 4-bit adder.

2.4.3 Making a Multiplier from an Adder

In class we will discuss how to use our full adder (the "Σ chip") to make a multiplier.

2.5 Multiplexing

A multiplexer (MUX) is a device which selects one of many inputs to a single output. The selection is done by using an input *address*. Hence, a MUX can take many data bits and put them, one at a time, on a single output data line in a particular sequence. This is an example of transforming *parallel* data to *serial* data. A demultiplexer (DEMUX) performs the inverse operation, taking one input and sending it to one of many possible outputs. Again the output line is selected using an address.

A MUX-DEMUX pair can be used to convert data to serial form for transmission, thus reducing the number of required transmission lines. The address bits are shared by the MUX and DEMUX at each end. If n data bits are to be transmitted, then after multiplexing, the number of separate lines required is $\log_2 n + 1$, compared to n without the conversion to serial. Hence for large n the saving can be substantial. In Lab 2, you will build such a system.

Multiplexers consist of two functionally separate components, a *decoder* and some switches or gates. The decoder interprets the input address to select a single data bit. We use the example of a 4-bit MUX in the following section to illustrate how this works.

2.5.1 A 4-bit MUX Design

We wish to design a 4-bit multiplexer. The block diagram is given in Fig. 9. There are 4 input data bits D_0–D_3, 2 input address bits A_0 and A_1, one serial output data bit Q, and an (optional) enable bit E which is used for expansion (discussed later). First we will design the decoder.

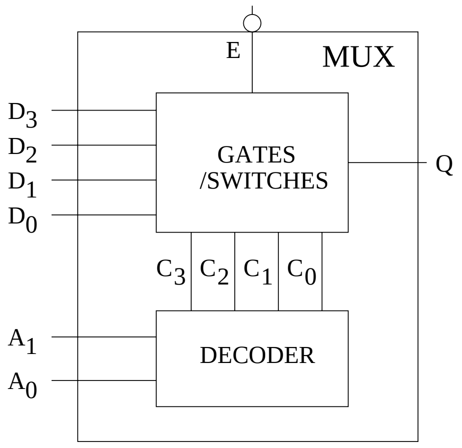

Figure 9: Block diagram of 4-bit MUX.

We need m address bits to specify 2^m data bits. So in our example, we have 2 address bits. The truth table for our decoder is straightforward:

A_1	A_0	C_0	C_1	C_2	C_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

The implementation of the truth table with standard gates is also straightforward, as given in Fig. 10.

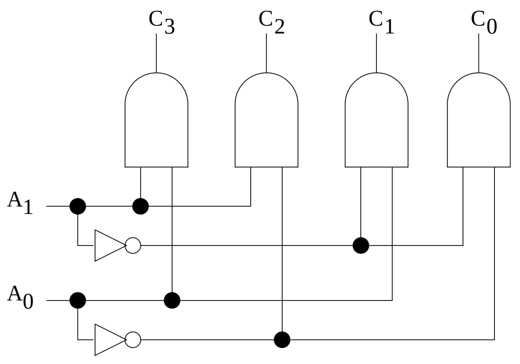

Figure 10: Decoder for the 4-bit MUX.

For the "gates/switches" part of the MUX, the design depends upon whether the input data lines carry digital or analog signals. We will discuss the analog possibility later. The digital case is the usual and simplest case. Here, the data routing can be accomplished simply by forming 2-input ANDs of the decoder outputs with the corresponding data input, and then forming an OR of these terms. Explicitly,

$$Q = C_0D_0 + C_1D_1 + C_2D_2 + C_3D_3$$

Finally, if an ENABLE line E is included, it is simply ANDed with the righthand side of this expression. This can be used to switch the entire MUX IC off/on, and is useful for expansion to more bits. as we shall see.