Chapter 2

A Quick Tour of Logic Synthesis with the Help of a Simple Example

Our purpose in this chapter is to give a quick overview of logic synthesis from the point of view of the user. We shall see what problems are faced (and solved) by a synthesis program and what general strategy may be applied to their solution. We shall use a very simple circuit as an example. No special effort has been made to make the example realistic. The sole purpose was to come up with something simple enough to be analyzed in detail with limited effort, and yet demonstrating a sufficient number of interesting problems.

In the rest of this chapter, we shall not concern ourselves with the solution of the problems, but rather with their statement. Thus we shall identify some of the particular degrees of freedom that present themselves to a designer in a practical problem. We shall then show how the designer's ingenuity exploits these degrees of freedom to create an efficient design. The sequel of this book is then devoted to showing how, and to what extent, equivalent ingenuity can be embodied in CAD tools that solve the same design problems automatically.

2.1 A Simple Case Conversion Circuit

We consider a simple circuit that will serve us as an example to illustrate the major steps of the synthesis process. The interface of the circuit is described in Figure 2.1.

A stream of alphabetic ASCII characters (a–z, A–Z) is fed to the circuit, one character for each clock cycle. The circuit performs case conversion on the incoming characters and outputs the corresponding sequence. One character is output for each clock cycle. No specification is given for the latency of the circuit. This means we can choose how many clock cycles will be required to process one character, as long as the throughput is one character per clock cycle.

The type of case conversion to be applied is specified by escape sequences[1], which

[1] An escape sequence is a sequence of two or more characters, the first of which is the escape character. Escape sequences are used to augment a character set and are typically used to carry

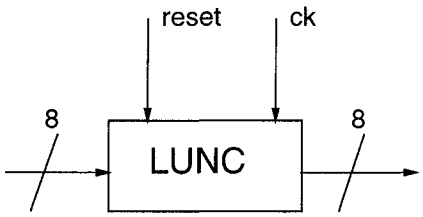

Figure 2.1: Interface of the example circuit.

are interspersed with the text characters. Our circuit recognizes four escape sequences (^[is our representation of the escape character):

- ^[L Lower case;
- ^[U Upper case;
- ^[N No conversion;
- ^[C Change Case.

By joining the four characters we get the name of our circuit: LUNC.

When the circuit is reset, it goes into a state where it passes the input characters unchanged. It remains in that state until it receives an escape sequence other than ^[N .

The behavior of the circuit for escape sequences other than the four listed above and for non-alphabetic characters is don't care, represented by symbol ? (that is, the behavior is unspecified). In addition, the output of the circuit when an escape sequence is input is also ? (that is, left unspecified).

We are allowed to use the freedom resulting from the don't care specification to simplify our design as much as we can. In a more realistic situation specifications may be more complete. However, it is important to make use of such *don't care* information, whenever it is available.

As an example of possible behavior of the circuit, consider the following two streams of characters. The one on the top is the input stream and the other is the output stream. In this example we are assuming that the latency of the circuit is two clock cycles.

```
a b C d E f ^[ U a b C D...
? ? a b C d E  f ? ? A B C D...
```

The question marks indicate don't care outputs. They occur at the beginning, when the first character has not been processed yet, and in response to the escape sequence. After the escape sequence has been processed, the circuit converts all incoming characters to upper case.

commands. Refer to the ASCII table of Figure A.1 in the Appendix for the codes of the characters.

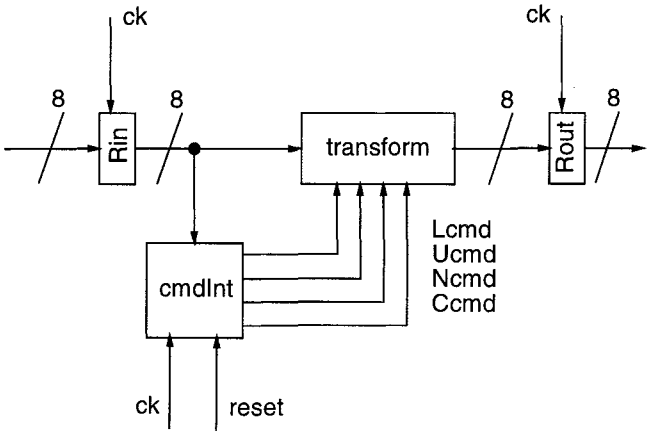

Figure 2.2: Block diagram for LUNC.

2.2 First Refinement

Our first step in the design of the LUNC circuit is to separate the data processing from the control. The result of this first step is shown in the block diagram of Figure 2.2. In this block diagram we can distinguish two main blocks: A command interpreter block (CMDINT), which parses the escape sequences and decides the state of the circuit accordingly, and a transformation block (TRANSFORM), which actually performs the case conversion. There are also an input and an output register, which determine the latency of the circuit. (We assume that there is no register inside the TRANSFORM block.)

The type of decomposition we have applied to our problem is fairly typical in the design of large digital systems. It is customary to divide the control functions from the data processing functions. It is not uncommon that different design methods are then applied to the separate parts. For instance, in a microprocessor, the design of the data path (ALUs and register files) is approached differently from the design of the instruction decoder or the cache controllers. In addition, the design of a complex circuit is carried out by a team. The block diagram is then important in defining the interfaces between the blocks designed by different people.

Defining the top-level block diagram of a circuit goes under the name of *high-level* or *architectural* design. In this course, we shall assume that this phase of the design has been carried out already—either manually or automatically—and we shall concentrate on the succeeding phases. It should be clear, however, that architectural design has a large impact on the outcome of the entire design process.

Before returning to our example, we make another general remark. In this case we have used a graphical representation of our block diagram. We could have used a textual representation as well. We are going to see examples of both in the sequel. It is typical of a real-life design system to support both ways of representation, for each has its own advantages.

Consider for instance the two major blocks of Figure 2.2. The command interpreter communicates with the transformation block by means of four signals (*Lcmd,*

Ucmd, Ncmd, Ccmd). At any time, only one of these signals is active. The active signal indicates the transformation to be applied to the character currently in the input register.

This kind of information is not easily conveyed by a drawing, but can be easily expressed by text. The previous paragraph is an example of *informal* textual description; formal languages can be used instead. These formal languages are similar to programming languages and are called *Hardware Description Languages* (HDLs).

Hardware description languages are used to describe both the structure of a circuit (what parts constitute it and how they are connected) and its behavior (how it reacts to given inputs). The syntax of HDLs is often similar to that of programming languages. For instance, VHDL,[2] one of the most widely used HDLs, is derived from Ada. The semantics of the HDLs, however, differ from those of ordinary programming languages in various respects.

In this book, we shall use a fictitious language, which borrows its syntax from the 'C' language. We shall informally define its semantics as we examine the examples of its use. Our treatment of HDLs in this book is essentially restricted to these brief introductory notes. It is important to realize, though, their relevance and their relationship to automatic logic synthesis. We shall try to emphasize that relationship as we discuss the design of the transform and command interpreter blocks.

2.3 The Transform Block

In the architectural design phase we decided that the TRANSFORM block would be a combinational circuit that, given an alphabetic character, outputs either the character itself or the character obtained by changing its case. The choice is determined by the four control inputs coming from the command interpreter and by the case of the input character.

We can describe the desired function in the following piece of code, that is largely self-explanatory.

```
Procedure TRANSFORM(Rin, Lcmd, Ucmd, Ncmd, Ccmd){
    if (Lcmd)
        { mux = TOLOWER(Rin)}
        else if (Ucmd)
            {mux = TOUPPER(Rin)}
            else if (Ncmd)
                {mux = Rin }
                else if (Ccmd)
                    {mux = CHANGECASE(Rin) }
    return(mux)
}
```

Rin, *Lcmd*, *Ucmd*, *Ncmd*, and *Ccmd* are the inputs and *mux* is the output. All of these are either 1 or 0 on each clock cycle. In a complete description, we would

[2] VHDL stands for VHSIC Hardware Description Language; VHSIC stands for Very High Speed Integrated Circuits and is the name of a research initiative of the US Department of Defense.

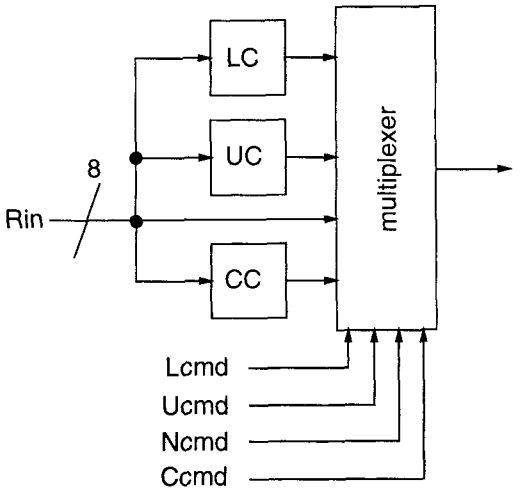

Figure 2.3: Block diagram for the transform block.

also specify how many bits each terminal has. The horizontal data path is marked as an 8-bit line, which is natural, since there are $2^8 = 256$ ASCII characters. The other lines are symbolic in principle, although in this case, as discussed below, all represent just one data bit.

Two remarks are in order here. First, a relatively high-level description like the one afforded by our simple HDL is easy to read and write. It is easier to interpret than a gate-level schematic, because it is more concise, it uses evocative names like TOLOWER, and it describes the behavior rather than the structure. For the same reasons, such a description is easier to write. In order to write it, we do not have to make up our mind on how precisely we are going to implement our circuit.

The second consideration should be suggested by the name of the circuit output. Calling the output MUX suggests that we can translate the **if-then-else** statement into a multiplexer. The translation is portrayed in Figure 2.3. In general, a synthesis program will have a scheme to translate the constructs of its input language into structure (i.e., into the interconnection of registers, multiplexers, adders, gates, and similar building blocks).

The translation scheme is not in general very sophisticated. We shall see that in our simple example, a straightforward translation of the initial description into a circuit yields an implementation that is far from optimal. We do not worry too much, though, because we rely on *optimization* techniques to improve our 'draft' circuit. This approach is followed by commercial and academic synthesis programs alike. One of its advantages is to make the final result independent—to a large extent, if not completely—from the initial description. The user of such a system can therefore save time and concentrate on clarity.

The approach we have just described relies heavily on the effectiveness and efficiency of the optimization techniques. We shall indeed concentrate on these techniques for most of this course. Returning to our LUNC circuit, we have described the TRANSFORM block in terms of simpler functions (TOLOWER, TOUPPER, CHANGECASE). We have to specify them, in order to complete our design. Here we shall only examine

```
Procedure CHANGECASE(Rin) {
    if (ISUC(Rin))
        { res = Rin + 32 }
    else
        {res = Rin − 32 }
    return (res)
}
```

Figure 2.4: Procedure CHANGECASE.

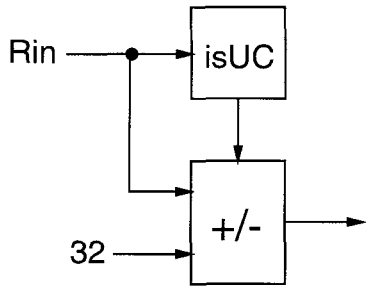

Figure 2.5: Block diagram for the CC block.

the most complex of them (CHANGECASE); the others are similar.

2.3.1 The CC Block

From Figure A.1 we see that, for each letter, the ASCII code for the lowercase character can be obtained from that of the uppercase character by adding 32 (base 10). This suggests the following definition for the CHANGECASE function. In this piece of code, ISUC is a function that says whether the input character is uppercase or lowercase. Since the output of the circuit is *don't care* when the input character is non-alphabetic, we can define ISUC by noting that bit 5 (the third most significant bit) is 0 for all uppercase letters and is 1 for all lowercase letters. Therefore, we can just define ISUC as $Rin[5]$[3]. If we notice that both addition and subtraction can be performed by a single adder/subtracter, then we can come up with the block diagram of Figure 2.5.

The TOLOWER and TOUPPER functions can also be implemented with an adder and a subtracter, respectively. Even though we have not worked out all the details, we can now get an idea of the cost—in terms of gates—of our 'draft' implementation of the TRANSFORM block. We have a total of three 8-bit adder/subtracters and an 8-bit, four-way multiplexer. It is reasonable to assume that about 200 gates are necessary for this implementation.

[3]Question: How could we hide the details of the ASCII code still further? (What if we used EBCDIC instead of ASCII?)

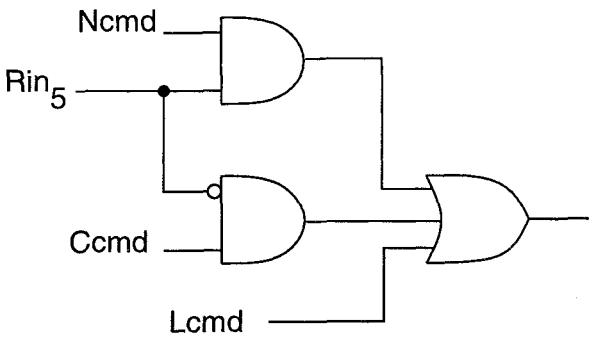

Figure 2.6: Circuit schematic for the optimized transform block.

2.3.2 An Optimized Transform Block

The discussion of the ISUC function may have already suggested to the attentive reader an alternative implementation of the TRANSFORM block. Let us consider the codes for 'a' and 'A.'

```
A = 0x41 = 01000001 = 65
a = 0x61 = 01100001 = 97
```

It is sufficient to flip bit 5 to go from lowercase to uppercase or vice versa. The same is true of all letters in ASCII. A minute's thought will show that the circuit of Figure 2.6—based on this idea—is indeed a correct implementation of the TRANSFORM block. The circuit actually produces only bit 5 of the results. All other bits of the result are identical to the corresponding input bits and therefore are not represented. Suppose $Ccmd = 1$. This implies $Ncmd = Lcmd = 0$. Then, the output is the complement of bit 5 of the input. Similarly, we can analyze the other three cases. Notice that *Ucmd* does not explicitly appear as an input to the circuit of Figure 2.6. Therefore, when $Ucmd = 1$, the output is 0.

This new implementation consists of only three gates. This is a lot less than the two hundred gates we estimated for our first 'draft.' If the optimization phase cannot pick up this slack, then the translation/optimization scheme is in trouble. Fortunately, in this case and in many others, optimization can easily get rid of the extra gates.

On the other hand, most people will agree that the purpose of the circuit of Figure 2.6, taken out of context, is not obvious. Our high-level description is more readable and eventually leads to an equally efficient implementation.

In drawing conclusions from our simple example, it is important to put things in perspective. There are cases where careful manual design is superior to the results of the best synthesis programs. The opposite also occurs, though less frequently. The importance of time-to-market should be always kept in mind when comparing manual design to automatic synthesis. One should also keep in mind that the problem of readability of a circuit description increases considerably with its size. If all design problems where of the complexity of our LUNC circuit, logic synthesis would have probably never evolved. For circuits with millions of transistors, on the other hand, the advantage afforded by logic synthesis may be decisive.

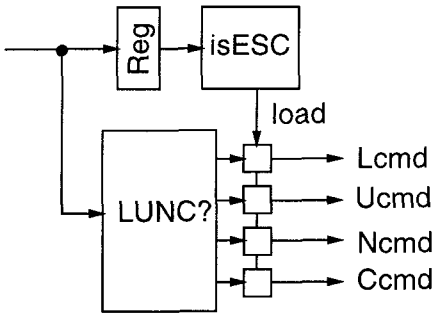

Figure 2.7: Block diagram for the command interpreter.

2.4 The Command Interpreter

Let us consider now the command interpreter. A simple block diagram for it is shown in Figure 2.7. As in the case of the *Transform* block we are initially interested in clarity and simplicity more than in efficiency.

The circuit of Figure 2.7 works by keeping a copy of the previous input character in a register. If the previous character is 'escape,' then the current character is used to determine the new state. The current input character is always decoded, but unless the previous character is 'escape,' the output of the decoder (block LUNC?) is ignored.

The function of the decoder is described by the code of Figure 2.8.

This code can be translated, for instance, in a truth table, from which a circuit can be derived. Notice that the output of the block is *don't care* for any unexpected character. This information is extremely important for the optimization of the circuit.

2.4.1 Checking for Equality

We can translate the test isESC into a test for equality.

```
isESC(Reg) = isSame(Reg, ESC);
```

Checking two values for equality occurs frequently in digital designs. In Figure 2.9 we show the typical template used in the translation phase. Similar templates are used for tests like $x \geq y$.

Notice that this circuit simplifies considerably when one of the two operands is constant.

2.4.2 Optimizing the Command Interpreter

Also for the command interpreter we now see how things can be optimized manually. This will give a target for the optimization phase. Let us consider the codes of the four letters that may appear in an escape sequence.

```
L = 0x4C = 01001100 = 74
U = 0x55 = 01010101 = 85
```

```
Procedure LUNC?(Rin) {
    if (Rin = L) {
        Lcmd = 1
        Ucmd = Ncmd = Ccmd = 1
        break
    }
    else if (Rin = U) {
        Ucmd = 1
        Lcmd = Ncmd = Ccmd = 1
        break
    }
    else if (Rin = N) {
        Ncmd = 1
        Lcmd = Ucmd = Ccmd = 1
        break
    }
    else if (Rin = C) {
        Ccmd = 1
        Lcmd = Ucmd = Ncmd = 1
        break
    }
    else
        Lcmd = Ucmd = Ncmd = Ccmd = Don't Care
    return (Lcmd, Ucmd, Ncmd, Ccmd)
}
```

Figure 2.8: Procedure LUNC?

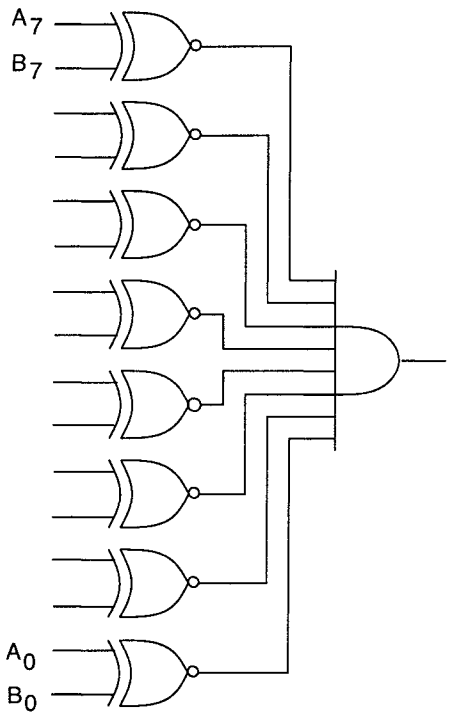

Figure 2.9: Circuit schematic for an equality checker.

```
N = 0x4E = 01001110 = 78
C = 0x43 = 01000011 = 67
```

Notice that the two least significant bits are sufficient to distinguish them. Furthermore, from the discussion of the TRANSFORM block, we know that we do not need to produce *Ucmd*. Therefore, we can implement the command decoder as shown in Figure 2.10.

Notice also that we can considerably reduce the number of flip-flops by a simple device. Instead of storing each input to test whether it is an 'escape' at the next clock cycle, we can test each input character as soon as we see it, and then store (in a single flip-flop) the result of the test. Such a transformation is called a *retiming*

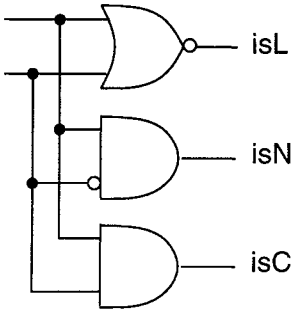

Figure 2.10: Circuit schematic for the optimized command interpreter.

of the circuit. It can be formalized and automated as it is in SIS. For lack of time, however, we shall not study it in this course and we only mention it here.

We can further reduce the number of flip-flops by noting that our command interpreter has only three possible states. Two flip-flops are sufficient to encode three states. In general, eliminating one flip-flop this way may make the combinational logic more complicated. One of the advantages of an automatic synthesis system is to make it possible for a designer to explore several possible solutions in a short time. In our case, though, a simple retiming transformation is sufficient to reduce the number of flip-flops without changing the combinational logic. It is sufficient to latch the inputs to the circuit of Figure 2.10, instead of the outputs. This transformation will make the circuit a little slower, but will also reduce power consumption.

2.5 Technology Mapping

Let us assume that our objective is to produce a netlist that may be used to fabricate a standard-cell chip. Suppose we have chosen a CMOS library. We now have to address the concerns arising from these choices.

Gates in CMOS (and in other technologies) are *negative*. This means that the basic gate is the inverter, rather than the non-inverting buffer. Consequently, NANDs and NORs are cheaper and faster than ANDs and ORs. Practical gates have limited driving capability, so that a gate typically drives four other gates or less. We say that the maximum fanout is four. This number may be further reduced if speed is a primary concern.

Likewise, a restriction is usually applied to the number of inputs to a gate. Even though it is possible in theory to build NAND and NOR gates with very large number of inputs, performance rapidly degrades as the number of inputs increases. Therefore, cell libraries do not usually provide gates with more than four or five inputs. Not all functions with those many inputs will be available either. A five-input NAND gate may be available, but a five-input EXOR may not. One must then make sure that only gates from the library are used in the circuit. All these concerns must be addressed by a synthesis program, as they are addressed by a human designer. This task is called *technology mapping*.

In the translation/optimization scheme we have examined so far, we have assumed that the result of the optimization phase is a technology-independent circuit. Our final scheme is therefore composed of three phases: Translation, optimization and technology mapping (or *techmapping* for short). The division is to some extent arbitrary. By separating optimization from technology mapping we simplify the two tasks and we can develop very powerful techniques for both. On the other hand, in the optimization phase we may have a less than perfect knowledge of the consequences of a given choice. In general the penalty for ignoring technology-specific information during optimization is higher when the target of optimization is high performance or low power. It is lower, but non-null, when area is being optimized.

As a consequence, the boundaries between the technology-independent phase and the technology-dependent phase tend to be blurred in real systems. The division is, however, useful from the didactic standpoint and we shall adopt it.

Suppose our cell library consists of inverters, two-input NAND and NOR gates,

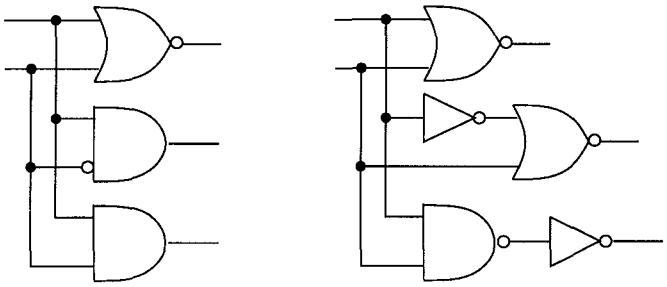

Figure 2.11: Circuit schematic for the technology-mapped decoder of the command interpreter.

and D-type flip flops. Then Figure 2.11 describes a possible mapping for the decoder of the command interpreter.

The choice we made (standard-cell chip in CMOS) is not the only one possible. On the one hand, we may be interested in full-custom design, and therefore in mapping at the transistor level, rather than at the gate level. On the other hand, we may want to implement our circuit as a Field Programmable Gate Array (FPGA). In both cases the mapping problem is different from that encountered with a fixed cell library.

2.6 Problems

1. Describe an 8-bit adder in BLIF format.

   Run SIS on your adder. Read in the circuit (with SIS command `read_blif`) and print out the statistics for it (with command `print_stats`).

   Include in your homework the description of the adder and the result of the `print_stats` command.

   Save your adder, because you will use it in other assignments as a building block. You may want to spend some time to familiarize with SIS. For instance, you may want to try the `simulate` command to verify that your description works as intended. Take a look at the man page. Obviously, at this stage, not everything will be clear: Don't worry. Notice that there is a handy UNIX `alias` command that lists all standard abbreviations. You may use `alias` to create your own abbreviations.

   Solution. Since the description of the standard "ripple-carry" adder is already in the blif documentation—albeit for four-bit numbers—we describe here another type of adder, which is faster. It is known as a carry-bypass adder. We shall have occasion to discuss it later in the course. Notice that it is composed of four blocks, each computing two output bits.

```
#------------------------ cbpadd8.blif ------------------------
# Adds two 8-bit inputs and a carry-in bit. Index 0 signals the
# least significant bit. The result is a 9-bit number. No two's
# complement overflow output is produced.
# The adder is composed of 4 modules, each computing the sum of
# two bits and based on the carry-bypass scheme.
```

```
.model cbpadd8
.inputs cin a0 a1 a2 a3 a4 a5 a6 a7 \
b0 b1 b2 b3 b4 b5 b6 b7
.outputs s0 s1 s2 s3 s4 s5 s6 s7 s8
.subckt cbp2 c0=cin a0=a0  b0=b0  a1=a1  b1=b1  s0=s0  s1=s1  c2=c2
.subckt cbp2 c0=c2  a0=a2  b0=b2  a1=a3  b1=b3  s0=s2  s1=s3  c2=c4
.subckt cbp2 c0=c4  a0=a4  b0=b4  a1=a5  b1=b5  s0=s4  s1=s5  c2=c6
.subckt cbp2 c0=c6  a0=a6  b0=b6  a1=a7  b1=b7  s0=s6  s1=s7  c2=s8
.end

# Two-bit carry bypass adder.

.model cbp2
.inputs c0 a0 b0 a1 b1
.outputs s0 s1 c2
.names a0 b0 g1
10 1
01 1
.names a0 b0 g2
11 1
.names a1 b1 g3
10 1
01 1
.names a1 b1 g4
11 1
.names c0 g1 g5
10 1
01 1
.names c0 g1 g6
11 1
.names g2 g6 g7
1- 1
-1 1
.names g3 g7 g8
10 1
01 1
.names g3 g7 g9
11 1
.names g1 g3 g10
11 1
.names g4 g9 g11
1- 1
-1 1
.names g10 g11 c0 mux
01- 1
1-1 1
.names g5 s0
1 1
.names g8 s1
1 1
.names mux c2
```

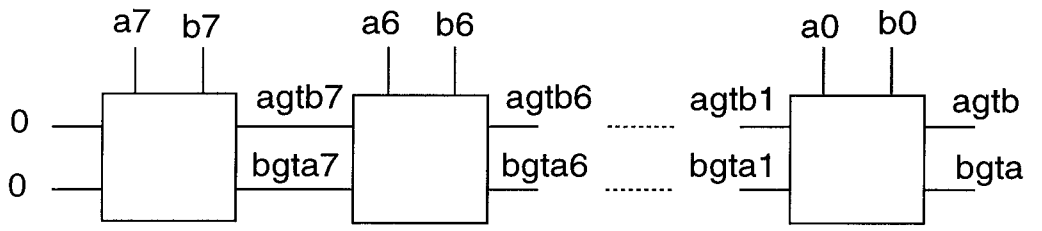

Figure 2.12: Iterative scheme for the 8-bit comparator of Problem 3.

```
1 1
.end
```

The result of running the PS command (an alias for `print_stats -f`), is the following:

```
cbpadd8          pi=17   po= 9   nodes= 51       latches= 0
lits(sop)= 139  lits(fac)= 139
```

□

2. Repeat Problem 1, this time for an 8-bit equality comparator.

3. Describe an 8-bit comparator in BLIF that takes two unsigned integers a and b as inputs and produces two outputs:

   (a) *agtb*: is 1 if and only if $a > b$;

   (b) *bgta*: is 1 if and only if $a < b$.

   Clearly, if $a = b$ both outputs are 0, and it is never the case that the two outputs are 1 simultaneously.

   Use the following "interface" for your comparator.

```
.model cmp8
.inputs a0 a1 a2 a3 a4 a5 a6 a7 b0 b1 b2 b3 b4 b5 b6 b7
.outputs agtb bgta
```

   Design the circuit by replicating a basic cell according to the scheme of Figure 2.12. Verify that your circuit works properly by using SIS to simulate the following pairs of inputs:

   (a) 0 0;

   (b) 2 3;

   (c) 3 2;

   (d) 200 100;

   (e) 5 5.

Create a script file containing the simulation commands and use the `source` command to run them.

Include in your homework your BLIF description, the simulation script, and the output of the simulation produced by SIS. (Use the `set sisout foo` command to redirect your output to file `foo`.)

This problem counts for 10.

Solution. A BLIF file for the comparator is as follows:

```
.model cmp8
.inputs a0 a1 a2 a3 a4 a5 a6 a7 b0 b1 b2 b3 b4 b5 b6 b7
.outputs agtb bgta

.names zero

.subckt comp agtb-1=zero  altb-1=zero  a=a7 b=b7 agtb=agtb7 altb=bgta7
.subckt comp agtb-1=agtb7 altb-1=bgta7 a=a6 b=b6 agtb=agtb6 altb=bgta6
.subckt comp agtb-1=agtb6 altb-1=bgta6 a=a5 b=b5 agtb=agtb5 altb=bgta5
.subckt comp agtb-1=agtb5 altb-1=bgta5 a=a4 b=b4 agtb=agtb4 altb=bgta4
.subckt comp agtb-1=agtb4 altb-1=bgta4 a=a3 b=b3 agtb=agtb3 altb=bgta3
.subckt comp agtb-1=agtb3 altb-1=bgta3 a=a2 b=b2 agtb=agtb2 altb=bgta2
.subckt comp agtb-1=agtb2 altb-1=bgta2 a=a1 b=b1 agtb=agtb1 altb=bgta1
.subckt comp agtb-1=agtb1 altb-1=bgta1 a=a0 b=b0 agtb=agtb  altb=bgta

.end

.model comp
.inputs agtb-1 altb-1 a b
.outputs agtb altb

.names agtb-1 altb-1 a b agtb
1--- 1
0010 1

.names agtb-1 altb-1 a b altb
-1-- 1
0001 1
.end
```

The script file to simulate the comparator is:

```
sim 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0
sim 0 1 0 0 0 0 0 0  1 1 0 0 0 0 0 0
sim 1 1 0 0 0 0 0 0  0 1 0 0 0 0 0 0
sim 0 0 0 1 0 0 1 1  0 0 1 0 0 1 1 0
sim 1 0 1 0 0 0 0 0  1 0 1 0 0 0 0 0
```

When the script file is 'sourced,' the output is:

```
Network simulation:
Outputs: 0 0
```

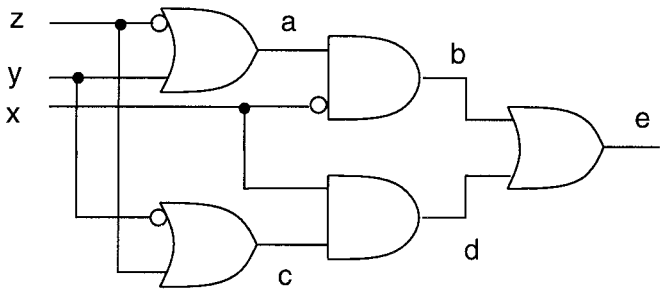

Figure 2.13: Circuit for Problem 4.

```
Next state:

Network simulation:
Outputs: 0 1
Next state:

Network simulation:
Outputs: 1 0
Next state:

Network simulation:
Outputs: 1 0
Next state:

Network simulation:
Outputs: 0 0
Next state:
```

□

4. This problem guides you through a simple example of optimization and technology mapping with SIS. Describe in BLIF the circuit of Figure 2.13. Use one *.names* directive for each gate in the drawing. Do the following.

   (a) Read your file into SIS;

   (b) read in a library for mapping with the command `rlib lib2.genlib`;

   (c) print the statistics of your circuit with `ps`;

   (d) perform techmapping with the command `map`; (ignore the warnings;)

   (e) print the statistics on your mapped network with `pg -s`;

   (f) print the equations of the network with `p`.

   (g) read in your file again;

   (h) simplify the circuit with the command ESPRESSO;

(i) repeat Steps 4c–4f.

Report in your homework the output of SIS for all steps. Include also your BLIF file. By how much does the area decrease when the circuit is optimized? (Use the areas reported by the command `pg -s`. Note that those areas only account for the *active area*; the area taken by the interconnections is not included.)

Solution. This is the `blif` file:

```
.model pb3.16
.inputs x y z
.outputs e
.names y z a
01 0
.names x a b
01 1
.names y z c
10 0
.names x c d
11 1
.names b d e
00 0
.end
```

This is the output of SIS:

```
pb3.16          pi= 3 po= 1 nodes=  5 latches= 0
lits(sop)=  10 lits(fac)=  10
inv2x           :    2 (area=928.00)
nor2            :    1 (area=1392.00)
oai21           :    2 (area=1856.00)
Total: 5 gates, 6960.00 area
     [339] = z'
     a = [339]' y'
     [338] = y'
     [355] = [338]' z' + x'
     {e} = [355]' + a' x'
pb3.16          pi= 3 po= 1 nodes=  5 latches= 0
lits(sop)=  10 lits(fac)=  10
inv2x           :    1 (area=928.00)
nand2           :    1 (area=1392.00)
oai221          :    1 (area=2784.00)
Total: 3 gates, 5104.00 area
     [400] = x'
     [415] = y' + z'
     {e} = [400]' y' + [415]' + x' z'
```

The area decreases by $6960 - 5104 = 1856$ units (in this case square microns). The percentage improvement is about 27%. Note that in both cases (with and without optimization) complex gates from the library are used to map the circuit. □

5. In this problem we use SIS to synthesize the TRANSFORM block of the LUNC circuit. (See Figure 2.3.) The purpose of this problem is twofold. On the one hand, we get to know SIS better. On the other hand, we see that the translation/optimization strategy can actually produce the results we got by manual optimization. Summarizing in a few words, the typical synthesis system translates a high-level description of the circuit into a structure composed of adders, multiplexors, comparators, etc.. This structure is then optimized. In this problem, we describe the initial structure in `blif` format and then use SIS to optimize it.

   Though the optimized transform block is small, the initial description, with several adders and multiplexors, is not so small. Therefore, we use a hierarchical description. How to write hierarchical descriptions in `blif` is described in the `blif` manual.

   To get good results from SIS, it is important that we specify whatever don't care information we have. How to specify external don't cares in `blif` is also described in the `blif` manual.

   Once we have described the TRANSFORM block, we have to optimize it with SIS. We shall use a `script`, i.e., an existing recipe that applies several commands in sequence. Scripts are run with the `source` command. Standard scripts are provided with SIS and can be used directly, without even knowing what they look like. (Though it is not advisable in general.) As all files in the default library directory, they can be referenced directly from within SIS by just giving the last component of their pathnames. One of the script is called `script.rugged`. It is a good idea to look at it and then try it out. You should also read `script.rugged.notes`. Finally, you may want to compare different scripts.

   Once we have optimized the logic, we shall perform technology mapping. We shall use the `lib2.genlib` library for that. The commands that we need are `read_library` and `map`.

   Finally, we shall run the `atpg -r` command to generate test vectors for our optimized circuit. Note we use the `-r` option.

   You may find it useful to check what the `alias` command does for you. There are many handy aliases that are defined for frequently used commands in the standard configuration file.

   Now, the details of how you have to report your results.

   (a) Write separate BLIF files for each block in Figure 2.3. You will also need separate files for components like adders that you will use in separate blocks. Finally, write a master file, where you describe how the blocks are connected. Use the `.search` directive to put everything together. Remember the external don't cares!

   (b) Run SIS. Show the statistics before running `script.rugged` and afterwards. Show the equations after optimization. Draw a schematic of the circuit after optimization and compare it to what might have been obtained manually.

(c) Perform technology mapping. Show the stats, the equations, and the library gates used (the last with the `print_gate` command). Draw a schematic of the mapped circuit.

(d) Run `atpg` and include the untestable faults and the tests generated in your report.

Solution. Listed below are the files comprising the description of the transform block. First we show the top-level description. We use a style that is only partially hierarchical. Indeed, the multiplexer is not describe as a nested block, but rather as a set of gates—one for each output. Notice also the description of the don't cares. It is a requirement of SIS that all don't cares be listed in the top level description—one function for each primary output.

```
#--------------------------- transform.blif -----------------------
.search toLower.blif
.search toUpper.blif
.search changecase.blif

.model transform
.inputs L U N C in0 in1 in2 in3 in4 in5 in6 in7
.outputs o0 o1 o2 o3 o4 o5 o6 o7
.subckt toLower \
        in0=in0 in1=in1 in2=in2 in3=in3 in4=in4 in5=in5 in6=in6 in7=in7 \
        o0=l0 o1=l1 o2=l2 o3=l3 o4=l4 o5=l5 o6=l6 o7=l7
.subckt toUpper \
        in0=in0 in1=in1 in2=in2 in3=in3 in4=in4 in5=in5 in6=in6 in7=in7 \
        o0=u0 o1=u1 o2=u2 o3=u3 o4=u4 o5=u5 o6=u6 o7=u7
.subckt changecase \
        in0=in0 in1=in1 in2=in2 in3=in3 in4=in4 in5=in5 in6=in6 in7=in7 \
        o0=c0 o1=c1 o2=c2 o3=c3 o4=c4 o5=c5 o6=c6 o7=c7
.names L U N C l0 u0 in0 c0 o0
10001--- 1
0100-1-- 1
0010--1- 1
0001---1 1
.names L U N C l1 u1 in1 c1 o1
10001--- 1
0100-1-- 1
0010--1- 1
0001---1 1
.names L U N C l2 u2 in2 c2 o2
10001--- 1
0100-1-- 1
0010--1- 1
0001---1 1
.names L U N C l3 u3 in3 c3 o3
10001--- 1
0100-1-- 1
0010--1- 1
0001---1 1
.names L U N C l4 u4 in4 c4 o4
10001--- 1
```

```
0100-1-- 1
0010--1- 1
0001---1 1
.names L U N C l5 u5 in5 c5 o5
10001--- 1
0100-1-- 1
0010--1- 1
0001---1 1
.names L U N C l6 u6 in6 c6 o6
10001--- 1
0100-1-- 1
0010--1- 1
0001---1 1
.names L U N C l7 u7 in7 c7 o7
10001--- 1
0100-1-- 1
0010--1- 1
0001---1 1
.exdc
.names L U N C o0
11-- 1
1-1- 1
1--1 1
-11- 1
-1-1 1
--11 1
0000 1
.names L U N C o1
11-- 1
1-1- 1
1--1 1
-11- 1
-1-1 1
--11 1
0000 1
.names L U N C o2
11-- 1
1-1- 1
1--1 1
-11- 1
-1-1 1
--11 1
0000 1
.names L U N C o3
11-- 1
1-1- 1
1--1 1
-11- 1
-1-1 1
--11 1
0000 1
.names L U N C o4
```

```
11-- 1
1-1- 1
1--1 1
-11- 1
-1-1 1
--11 1
0000 1
.names L U N C o5
11-- 1
1-1- 1
1--1 1
-11- 1
-1-1 1
--11 1
0000 1
.names L U N C o6
11-- 1
1-1- 1
1--1 1
-11- 1
-1-1 1
--11 1
0000 1
.names L U N C o7
11-- 1
1-1- 1
1--1 1
-11- 1
-1-1 1
--11 1
0000 1
.end

#--------------------------- changecase.blif ----------------------
# Changes case of an alphabetic ASCII character by adding or
# subtracting 32 (decimal). This block does not check for the
# character being non-alphabetic.

.search addsub8.blif

.model changecase
.inputs in0 in1 in2 in3 in4 in5 in6 in7
.outputs o0 o1 o2 o3 o4 o5 o6 o7
.names zero
.names one
1
.subckt addsub8 addsub=in5 \
   a0=in0 a1=in1 a2=in2 a3=in3 a4=in4 a5=in5 a6=in6 a7=in7 \
   b0=zero b1=zero b2=zero b3=zero b4=zero b5=one b6=zero b7=zero \
   s0=o0 s1=o1 s2=o2 s3=o3 s4=o4 s5=o5 s6=o6 s7=o7 s8=dummy
.end

#---------------------------- toLower.blif ----------------------
```

```
# Changes case of an uppercase alphabetic ASCII character by adding 32
# (decimal). Lowercase characters are left unchanged.
# This block does not check for the character being non-alphabetic.

.search adder8.blif

.model toLower
.inputs in0 in1 in2 in3 in4 in5 in6 in7
.outputs o0 o1 o2 o3 o4 o5 o6 o7
.names zero
.names one
1
.subckt adder8 cin=zero \
    a0=in0 a1=in1 a2=in2 a3=in3 a4=in4 a5=in5 a6=in6 a7=in7 \
    b0=zero b1=zero b2=zero b3=zero b4=zero b5=one b6=zero b7=zero \
    s0=k0 s1=k1 s2=k2 s3=k3 s4=k4 s5=k5 s6=k6 s7=k7 s8=dummy
.names in5 in0 k0 o0
11- 1
0-1 1
.names in5 in1 k1 o1
11- 1
0-1 1
.names in5 in2 k2 o2
11- 1
0-1 1
.names in5 in3 k3 o3
11- 1
0-1 1
.names in5 in4 k4 o4
11- 1
0-1 1
.names in5 in5 k5 o5
11- 1
0-1 1
.names in5 in6 k6 o6
11- 1
0-1 1
.names in5 in7 k7 o7
11- 1
0-1 1
.end



#-------------------------- toUpper.blif ----------------------
# Changes case of a lowercase alphabetic ASCII character by
# subtracting 32 (decimal). Uppercase characters are left unchanged.
# This block does not check for the character being non-alphabetic.

.search addsub8.blif

.model toUpper
```

```
.inputs in0 in1 in2 in3 in4 in5 in6 in7
.outputs o0 o1 o2 o3 o4 o5 o6 o7
.names zero
.names one
1
.subckt addsub8 addsub=one \
   a0=in0 a1=in1 a2=in2 a3=in3 a4=in4 a5=in5 a6=in6 a7=in7 \
   b0=zero b1=zero b2=zero b3=zero b4=zero b5=one b6=zero b7=zero \
   s0=k0 s1=k1 s2=k2 s3=k3 s4=k4 s5=k5 s6=k6 s7=k7 s8=dummy
.names in5 in0 k0 o0
01- 1
1-1 1
.names in5 in1 k1 o1
01- 1
1-1 1
.names in5 in2 k2 o2
01- 1
1-1 1
.names in5 in3 k3 o3
01- 1
1-1 1
.names in5 in4 k4 o4
01- 1
1-1 1
.names in5 in5 k5 o5
01- 1
1-1 1
.names in5 in6 k6 o6
01- 1
1-1 1
.names in5 in7 k7 o7
01- 1
1-1 1
.end


#--------------------------- addsub8.blif ----------------------
# Adds/subtracts two 8-bit integers. Index 0 signals the least
# significant bit. Input addsub causes addition when it is 0 and
# subtraction (a-b) when it is 1. The result is a 9-bit number.
# No two's complement overflow output is provided.

.search adder8.blif

.model addsub8
.inputs addsub a0 a1 a2 a3 a4 a5 a6 a7 b0 b1 b2 b3 b4 b5 b6 b7
.outputs s0 s1 s2 s3 s4 s5 s6 s7 s8
.names addsub b0 c0
10 1
01 1
.names addsub b1 c1
10 1
01 1
```

```
.names addsub b2 c2
10 1
01 1
.names addsub b3 c3
10 1
01 1
.names addsub b4 c4
10 1
01 1
.names addsub b5 c5
10 1
01 1
.names addsub b6 c6
10 1
01 1
.names addsub b7 c7
10 1
01 1
.subckt adder8 \
     cin=addsub a0=a0 a1=a1 a2=a2 a3=a3 a4=a4 a5=a5 a6=a6 a7=a7 \
     b0=c0 b1=c1 b2=c2 b3=c3 b4=c4 b5=c5 b6=c6 b7=c7 \
     s0=s0 s1=s1 s2=s2 s3=s3 s4=s4 s5=s5 s6=s6 s7=s7 s8=s8
.end


#--------------------------- adder8.blif ----------------------
# Adds two 8-bit inputs and a carry-in bit. Index 0 signals the least
# significant bit. The result is a 9-bit number. No two's complement
# overflow output is produced.
.model adder8
.inputs cin a0 a1 a2 a3 a4 a5 a6 a7 b0 b1 b2 b3 b4 b5 b6 b7
.outputs s0 s1 s2 s3 s4 s5 s6 s7 s8
.subckt full_adder cin=cin a=a0 b=b0 sum=s0 cout=c0
.subckt full_adder cin=c0 a=a1 b=b1 sum=s1 cout=c1
.subckt full_adder cin=c1 a=a2 b=b2 sum=s2 cout=c2
.subckt full_adder cin=c2 a=a3 b=b3 sum=s3 cout=c3
.subckt full_adder cin=c3 a=a4 b=b4 sum=s4 cout=c4
.subckt full_adder cin=c4 a=a5 b=b5 sum=s5 cout=c5
.subckt full_adder cin=c5 a=a6 b=b6 sum=s6 cout=c6
.subckt full_adder cin=c6 a=a7 b=b7 sum=s7 cout=s8
.end

.model full_adder
.inputs a b cin
.outputs sum cout
.names cin a b sum
001 1
010 1
100 1
111 1
.names cin a b cout
-11 1
1-1 1
```

```
11- 1
.end
```

If these files are read into SIS, the following initial statistics are obtained.

```
transform       pi=12   po= 8   nodes= 94       latches= 0
lits(sop)= 700  lits(fac)= 606
```

After running the script, we get the following stats.

```
transform       pi=12   po= 8   nodes=  9       latches= 0
lits(sop)=  12  lits(fac)=  12
```

These are the equations. As we can see, we obtain the same solution that was obtained manually.

```
{o0} = in0
{o1} = in1
{o2} = in2
{o3} = in3
{o4} = in4
{o5} = C in5' + L + N in5
{o6} = in6
{o7} = in7
[170] = -0-
```

These are the equations after mapping.

```
{o0} = in0
{o1} = in1
{o2} = in2
{o3} = in3
{o4} = in4
[510] = in5'
[165] = C' L' N' + C' L' in5' + L' N' [510]' + L' [510]' in5'
{o5} = [165]'
{o6} = in6
{o7} = in7
```

The library cells used in the mapped circuit are given by the `pg` command.

```
[510]       inv2x           928.00
[165]       aoi221          2784.00
{o5}        inv2x           928.00
```

Finally, running the `atpg -r` command, we get the following output.

```
38 total faults
RTG: covered 35 remaining 3
RTG: covered 1 remaining 2
36 faults covered by RTG
S_A_0: NODE: U  OUTPUT
Redundant
S_A_1: NODE: U  OUTPUT
Redundant
faults: 38      tested: 36      aborted: 0      redundant: 2
```

Notice that the redundant faults correspond to input U, that is not used in the circuit (and hence, it is not observable). The following is the list of patterns that are generated.

```
# atpg test patterns for transform
.inputs  L U N C in0 in1 in2 in3 in4 in5 in6 in7
011000000001
111001010110
010001101110
100000011000
010000111011
010110111010
101001111111
000111000101
001001011111
101001001110
```

□

6. Create a blif file and perform the SIS print_stats and sim (for the input character string of Section 2.1) commands for the overall LUNC circuit.