Northwestern University — May 2, 2002
Electrical and Computer Engineering — Handout #4
ECE303: Advanced Logic Design, Spring 02 — Due: May 9
Prof. Hai Zhou

Lab 3 Assignment

1 Introduction

In this laboratory, you will gain experience in using the **ESPRESSO** and **SIS** synthesis systems to minimize two-level logic expressions and map a collection of combinational functions into optimized multi-level Boolean expressions. In particular, you will:

1. Learn how to create inputs and interpret outputs for ESPRESSO;
2. Learn how to use ESPRESSO to minimize single and multiple output functions;
3. Become familiar with the operations provided by SIS for manipulating a Boolean network, such as decomposition, extraction, and factoring;
4. Use SIS to map an optimized multi-level expression into a specific gate library, such as the Mississippi State University library of standard cells.

It is assumed that you are familiar with UNIX, and that you know how to log in, navigate your way around directories, and edit files. Place the following lines in your startup file (seek the help of the teaching assistant or a colleague in the course if none of this makes sense to you):

```
setenv SIS = /vol/cad/sis-1.2
set path = ($SIS/bin $path)
```

The programs you will be using can be found in the directory /vol/cad/sis-1.2/bin in the SUN workstations in the Wilkinson Lab. You can login to the machines using the command

```
rlogin fastlab
```

This will log you into the next available and least loaded SUN machine in the lab.

2 ESPRESSO

2.1 Tutorial

Espresso takes as input a two-level representation of a Boolean function, and produces a minimal equivalent representation. The command line typed at the UNIX prompt looks like the following:

```
espresso [options] [file]
```

Espresso reads the file provided (or standard input if no file is specified), performs the minimization, and writes the minimized result to standard output. If you want to keep the output, for example, to print it out for later reference, you should redirect it to a file. Espresso comes with a rather large choice of options. Most of these are for the expert user, and will not be of concern for us.

Input to espresso can be created by creating a truth table form directly. For example, the expressions for the full adder would be written as:

```
sum = ain bin' cin + ain' bin cin' + ain bin' cin' + ain bin cin;
cout = ain bin cin' + ain' bin cin + ain bin cin + ain bin' cin;
```

Parentheses are optional if you write your expressions in Sum of Products form, since complement has higher precedence than AND, which has higher precedence than OR. However, you may use them to change the default precedence.

The ESPRESSO input file in the PLA format is:

```
.i    3             -- number of input variables, e.g., ain, bin, cin
.o    2             -- number of output functions, e.g., sum, cout
.ilb  ain bin cin   -- input variable names, separated by spaces
.ob   sum cout      -- output function names, separated by spaces
.p    8             -- number of non-zero truth table entries
111   1 0           -- truth table row ain=1, bin=1, cin=1:  sum=1, cout=0
001   1 0
010   1 0
100   1 0
111   0 1
011   0 1
101   0 1
110   0 1
.e                  -- end of table
```

Note that there are some redundant truth table rows in this file (e.g., 111 is a term for sum as well as cout). Do not be concerned. These will be eliminated by ESPRESSO. [There is a utility called `eqntott` which can convert the equation file shown above into the PLA truth table file automatically but we do not want you to use that now. We want you to learn to use the PLA format.] For many examples, especially if you are given them in a truth table or minterm index form, it is easier to type the truth table file directly rather than use eqntott. Further, the truth table file is the only way to express don't care conditions. These are represented by placing a '-' in the truth table entry for the given input conditions and function.

Running espresso on the output of the above file will yield the following:

```
.i    3
.o    2
.na   [filename]
.ilb  ain bin cin
.ob   sum cout
.p    7             -- number of unique reduced product terms
001   1 0
010   1 0
-11   0 1           -- cout = bc +
100   1 0
1-1   0 1           -- ac +
11-   0 1           -- ab
111   1 0
.e
```

A '-' in the truth table index part means that the particular input variable does not participate in that reduced product term. A function's reduced sum of products description is the OR of all product terms which have a 1 in the output column for that function. Espresso could not find a reduced expression for sum, but did eliminate a term and some literals from the expression for cout.

2.2 Experiments

Let us begin with a simple 3-variable function: $f(A,B,C) = \sum m(3,4,5,6,7)$. First, fill in the K-map on the summary sheet, and determine the reduced two-level implementation by hand. Now, using the input format as described above, prepare an input file describing the function. Run Espresso and examine the output truth table it produces. How does the result compare with your hand determined result?

Since ESPRESSO uses the sum of products form internally, it cannot compute the product of sums form directly. However, it is possible to request that the function's complement be produced as output, using the parameter described above. You will need to apply DeMorgan's Law to obtain the result in the desired form. Try it for the function f. What answer do you obtain?

Repeat the process of finding the minimum sum of products form with the following 3, 4 variable functions:

$$f(A,B,C) = \sum m(0,4,6,7) \quad (1)$$

$$f(A,B,C,D) = \sum m(0,2,3,5,6,7,8,10,11,14,15) \quad (2)$$

The examples we have worked with so far have involved a single output function. Now enter the truth table for the two 2-bit binary adder described in page 77 of the Katz book. You can check out the K-maps for the three functions and the reduced equations from page 77-78 of Katz. Repeat the process using ESPRESSO. How does espresso's solution compare with the book? If they are not exactly the same, it may not be because you made a mistake, but rather that espresso found an alternate cover. If they are different, is the complexity of the covers the same, that is, do both solutions use the same number of product terms and literals?

Let us look at an example that contains don't care conditions. Take the truth table for the BCD increment by 1 function, given on Page 78 of the Katz book, generate the minimal two level functions process using espresso. Did you obtain the same results as the book? If not, can you explain the differences?

3 SIS

This laboratory will be partially tutorial in nature. First we will introduce the basic commands supported by SIS and we will use them to obtain a multi-level solution for the full adder circuit with which you should be familiar. Then it will be up to you to use SIS to obtain good multi-level implementations using the commands provided by SIS.

3.1 Tutorial

The command line typed at the UNIX prompt looks like the following:

```
    sis
```

Since sis is normally executed in an interactive mode, we will use that only. It can be used in batch mode also but we will not use those options. In response to that SIS will give a prompt:

```
sis>
```

While SIS can read in inputs of circuits in various forms, truth tables, equations, and BLIF format, the truth table form is the only way to make SIS aware of don't care conditions. These are represented by placing a '-' in the truth table entry for the given input conditions and function.

SIS provides a very large number of operations for reading in, manipulating, and printing out a collection of Boolean functions. Once again, many of these are for the expert user, but several can be used to good effect even by the novice. You can get a list of commands by typing

```
sis> help
```

For detailed instructions on a specific command type

```
sis> help read_pla
```

Here is a list of some of the command full names, and brief descriptions. Look at the SIS tutorial handled out separately for more details.

full command name	brief description
read_eqn	read equations from a file
read_pla	read PLA in ESPRESSO format
read_library	read a gate library
read_blif	read a logic netlist in BLIF format
write_eqn	write equations to a file
write_pla	write PLA to a file in expresso format
write_blif	write netlist in BLIF format
print_stats	print network status
print	print sum-of-products
print_factor	print factored form of a node
print_gate	print gate information for a node
collapse	collapse a level of the network
eliminate	eliminate nodes that don't save enough literals
decomp -g	perform a good decomposition
gcx	general cube extraction
gkx	general kernal extraction
resub	perform resubstitution
factor -g	perform a good factoring
map	map the current network onto the gates of a library
simplify -m espresso	perform an ESPRESSO minimization of a node
tech_decomp	decompose a network onto a constrained AND/OR network

The commands may seem complicated, by actually they are easy to use. Let's run through the multi-level optimization of the full adder. It is a good idea to create your own input file and follow this example step by step by typing the same commands at the keyboard.

First, we create an equation file for the full adder in the format expected by the EQNTOTT program which creates an ESPRESSO file from equations. We will create a UNIX file named `full.adder`. using a text editor like `vi` or `emacs`. Inside the file we type two lines as follows:

```
co = a * b * ci + a * b * ci' + a * b' * ci + a' * b * ci ;
sum = a * b * ci + a * b' * ci' + a' * b * ci' + a' * b' * ci ;
```

We can also type this as:

```
co = a b ci + a b ci' + a b' ci + a' b ci ;
sum = a b ci + a b' ci' + a' b ci' + a' b' ci ;
```

Save this file as "full_adder". The format of the EQNTOTT input file is that the equations are of the form

```
<signal> = <expr>;
```

The formats are:

`( )`	`grouping`
`!= (or ^ )`	`exclusive` OR
`==`	`exclusive` NOR
`!  (or ')`	`complement`
`& (or * or simple juxtaposition)`	`boolean` AND
`| (or + )`	`boolean` OR

Then we invoke sis and read in the network:

```
% sis

sis> read_eqn full.adder
```

To see the equations in sum-of-products form, type the 'p' command:

```
sis> print
{co} = a b ci + a b ci' + a b' ci + a' b ci
{sum} = a b ci + a b' ci' + a' b ci' + a' b' ci
```

Note that `co` and `sum` are in curly brackets: these have been identified by SIS as output nodes, and are treated in a special way. To see the equations in an initial factored form, type 'pf':

```
sis> print_factor
{co} = a b' ci + b (ci (a' + a) + a ci')
{sum} = ci (a' b' + a b) + ci' (a b' + a' b)
```

This is a direct reflection of how SIS is representing the current Boolean network internally. One of the powers of SIS is that it completely subsumes ESPRESSO. To invoke espresso on all the nodes of current network, simply type:

```
sis> simpify *
sis> print
{co} = a b + a ci + b ci
{sum} = a b ci + a b’ ci’ + a’ b ci’ + a’ b’ ci
```

The two level network should look familiar to you. The factored form is not surprising either:

```
sis > print_factor
{co} = ci (b + a) + a b
{sum} = ci (a’ b’ + a b) + ci’ (a b’ + a’ b)
```

We are now ready to instruct sis to decompose {co} and {sum} into a collection of simpler functions:

```
sis> decomp *
sis> print_factor
{co} = a [2] + b ci
{sum} = a’ [3]’ + a [3]
[2] = ci + b
[3] = b’ ci’ + b ci
```

Two new functions have been introduced and are represented by nodes [2] and [3]. NOTE THAT YOU MAY GET SOME SLIGHTLY DIFFERENT OUTPUTS DEPENDING ON YOUR EXACT ORDERING OF INPUTS. BUT THEY WILL HAVE THE SAME MINIMAL FORM.

Now let’s attempt some technology mapping, that is, the mapping of current network onto a pre-specified library of gates. We will use the Mississippi State gate library called “msu.genlib”. It includes various simple gates. Each gate is identified by a number (used by sis), a name, and a brief functional description. Besides the usual NAND/NOR gates, the library also contains some AOI and OAI gates, as well as XOR and XNOR. To use the library, we must first issue a `read_library` command followed by the `map` command.

```
sis> read_library msu.genlib
sis> map
<<ignore the warning messages>>
```

To see the revised multi-level network, type:

```
sis> print_factor
[361] = b’ ci’ + a’
[328] = b’
[329] = ci’
co = [328]’ [329]’ + [361]’
[3] = b ci’ + b’ ci
sum = [3] a’ + [3]’ a
```

(You may see something slightly different from the above, don’t be concerned). A number of new nodes have been introduced during the mapping onto the MSU gates. To understand what is going on, the `print_gates` command will print out the gates used in deriving each of the non-leaf nodes of the networks (the leaf nodes describe input literals):

```
sis> print_gate
[361] 1890:physical 32.00
```

```
[328] 1310:physical 16.00
[329] 1310:physical 16.00
{co} 1890:physical 32.00
[3] 2310:physical 40.00
{sum} 2310:physical 40.00
```

We can now draw a circuit schematic for the solution that sis has obtained. The gates chosen for implementation include two inverters (1310), two XOR gates (2310), and two OR-AND-Inverters (1890). The numbers following the word 'physical' describe the relative size of these gate, so you can see that XORs require considerably more area than inverters: 40 units versus 16! The total area of the implementation is 176 units. The figure at the top of the page depicts the implementation of the full adder in terms of these gate primitives. Gate numbers and internal node numbers are shown. Note that sis has used some AND-OR-INVERT gates.

VERIFY THAT THE ABOVE GATE IMPLEMENTATION INDEED IMPLEMENTS THE FULL ADDER USING THE TRUTH TABLES.

To complete a sis session, just type:

```
sis> quit
```

3.2 Further Experiments with the Full Adder

To illustrate a few more commands, let's see if we can improve on the implementation of the full adder. To start over, re-read the equation file and perform the two level simplification once again:

```
sis> read_eqn full.adder
sis> simplify *
```

To map the equations into an AND/OR network with constrained fan-ins, we can issue the td command. Its parameters are the OR fan-in (-o) and the AND fan-in (-a). In this case we will limit fan-ins to 4 for both kinds of gates:

```
sis> tech_decomp -o 4 -a 4
sis> print
{co} = [848] + [849] + [850]
{sum} = [844] + [845] + [846] + [847]
[844] = a b' ci'
[845] = a' b ci'
[846] = a' b' ci
[847] = a b ci
[848] = b ci
[849] = a ci
[850] = a b
```

The mapping onto 4-input AND and OR gates is visible from the printout of the network. Now let's map the network using the MSU gate library. The output should be somewhat different, since it does not support 4-input OR gates!

```
sis> read_library msu.genlib
sis> map
```

```
<<ignore the warning messages that print out now>>
sis> print
{co} = a b + a ci + b ci
[1207] = a'
[1205] = b'
[1214] = [1205] [1207] ci + a b ci
[844] = [1207]' b' ci'
[845] = [1205]' a' ci'
{sum} = [1214] + [844] + [845]
```

Can you draw the schematic that corresponds to this technology mapping? Issue the print_gates command to see what gates are used to implement the various network nodes. We now introduce a new command: print_map_stats. It summarizes the area and number of gates used in a particular technology mapping:

```
sis> print_map_stats

Total Area = 352.00
Gate Count = 12
Buffer Count = 0
Most Negative Slack = -5.00
Sum of Negative Slack = - 10.00
Number of Critical PO = 2
```

You can compare this final solution to the previous final solution in terms of gates and areas. Which one is superior?

SIS comes with a standard script that has been designed to give reasonably good technology independent solutions. To run the script, just type the following:

```
sis> read_eqn full.adder
sis> source script
sis> print
{co} = a b + a ci + b ci
{sum} = {co}' a + {co}' b + {co}' ci + a b ci
```

This solution yields 12 literals, which compares favorably with 18 literals found before mapping in the example above and the 11 literal solution before that. You can also execute SIS commands in a batch. This is particularly useful when using the standard minimization script. Simply type the command line:

```
sis -t eqn -T eqn -f script < [eqn filename]
```

'-t' indicates that the input file is in equation form while -T tells SIS that the output should be printed in equation form as well. -f gives the name of a command file or script file to be executed. Be careful, however. Sis first looks in the current directory for the specified file. Don't use the file name "script" if you want to use the standard minimization script.

3.3 The Two-Bit Binary Adder

Repeat the process described above, but this time using the two-bit binary adder introduced in Section 2 Proceed as follows:

1. create a pla or equation file for the two bit adder function;
2. invoke sis and read it in via the read_pla or read_eqn commands;
3. simplify the network using the `simplify` command;
4. decompose the network using the `decompose` command;
5. Map the function using the MSU library gates
6. Draw the resulting schematics in order to count gates;
7. Use the `print_map_stats` to get summary information on the gates, and areas.

Use the collapse, eliminate, and tech_decomp commands to obtain a second solution (see more about these commands in the SIS manual section of your tutorial). Obtain a third solution using the SIS standard script. Compare the three solution in terms of the number of gates required for implementation and the number of literals. How do these compare with the final result obtained using ESPRESSO only? Assuming 4-input AND and OR gates are available, how would ESPRESSO's solution compare with SIS solutions in terms of gate count and number of literals?

Once you have the confidence that you can use SIS to obtain reduced gate count implementations, you should be able to use this tool for all your future laboratories and design projects.