EECS 303: Advanced Digital Logic Design
Lab Two - Espresso and SIS logic minimization

Robert Dick

Assigned: 16 October
Due: 28 October

1 Introduction

In this laboratory assignment, you will use the Espresso and SIS synthesis systems to minimize two-level logic expressions and map a collection of combinational functions into an optimized multi-level network implementation. 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
4. Use SIS to map an optimized multi-level expression into a specific gate library: 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 line in your shell startup file, "`~/.tcshrc`".

```
source /vol/ece303/sis.env
```

If that file can't be written, you can turn on write permissions with the following command:

```
chmod u+w .tcshrc
```

Please glance at the appendices now. They might contain information that will help make the assignment easier to complete. In addition, please look at Section 5 before starting the lab. This section explains the things you will need to hand in for credit.

2 Espresso

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

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

Espresso reads the file provided (or the standard input if no file is specified), performs the minimization, and writes the minimized (but not necessarily minimal) 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, i.e.,

```
espresso [options] [file] > [output file]
```

Espresso allows many options. Most of these are for experts, and you need not understand them.

Input to Espresso can be specified with a truth table. For example, given the following expressions

$$sum = \overline{ain}\,\overline{bin}\,cin + \overline{ain}\,bin\overline{cin} + ain\overline{bin}\,\overline{cin} + ainbincin$$
$$cout = ainbin\overline{cin} + \overline{ain}\,bincin + ainbincin + ain\overline{bin}\,cin$$

the corresponding Espresso input file would be

```
# number of input variables, e.g., ain, bin, cin
.i      3
# number of output functions, e.g., sum, cout
.o      2
# input variable names, separated by spaces
.ilb ain bin cin
# output function names, separated by spaces
.ob     sum cout
# number of non-zero truth table entries
.p      8
# truth table row ain=1, bin=1, cin=1: sum=1, cout=0
111     10
001     10
010     10
100     10
111     01
011     01
101     01
110     01
# end of table
.e
```

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. Note that there is a utility called Eqntott that converts the equation file

```
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;
```

into the PLA truth table file automatically. However, we will use truth table entry in this part of the lab. For many examples, especially if they are specified in truth table or minterm index form, it is easier to type the truth table file

directly rather than use Eqntott. Furthermore, 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 above file will yield the following

```
.i 3
.o 2
.ilb ain bin cin
.ob sum cout
# number of unique reduced product terms
.p 7
100 10
010 10
001 10
111 10
# cout = bc +
-11 01
# ac +
1-1 01
# ab
11- 01
.e
```

Note that I added a few comment lines to the output file to explain it. These lines won't be present in the output from Espresso. However, comment lines from the input file will be included in the output. 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*.

2.1 Espresso 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. 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?

Espresso uses the SOP form internally. However, you can request that it produce a POS output using the "`-epos`" flag. Type

```
espresso --help
```

to get a summary of Espresso's commands.

Repeat the process of finding the minimum sum of products form with the following three four-variable functions:

$$F(A,B,C) = \sum_m (0,4,6,7)$$
$$F(A,B,C,D) = \sum_m (0,2,3,5,6,7,8,10,11,14,15)$$

Now enter the truth table for a two-bit binary adder. Read Chapter 5, Section 2 of the book by Mano and Kime. You'll find a truth table for a (one-bit) full adder on page 204. Derive the truth table for a two-bit adder, described as follows:

a	b	s
0	0	0
0	1	1
0	2	2
0	3	3
1	0	1
1	1	2
1	2	3
1	3	4
2	0	2
2	1	3
2	2	4
2	3	5
3	0	3
3	1	4
3	2	5
3	3	6

a and b are both two-bit numbers, composed of a_1, a_0, b_1, and b_0. The sum, s, is a three-bit number composed of s_2, s_1, and s_0. Derive minimized equations for the sum bits using K-maps. Repeat the process using Espresso. How does Espresso's solution compare with yours? 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 priority encoder, given on Page 154 of Mano and Kime. Generate the minimized two level functions using Espresso. Did you obtain the same results as shown on page 155? If not, can you explain the differences?

3 SIS tutorial

This laboratory will be partially tutorial in nature. SIS is a sequential logic synthesis package developed at UC Berkeley in the early 1990s. After introducing the basic commands supported by SIS, we will use them to obtain a multi-level solution for the (one-bit) full adder circuit from Mano and Kime. Then it will be up to you to use SIS to obtain good multi-level implementations using the commands provided by SIS. Note that SIS contains the optimization techniques in MIS-II, a version of the MIS package mentioned in class. In this lab, we will use SIS for multilevel combinational optimization.

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

```
sis
```

We will use SIS in interactive mode. It can also be used in batch mode.. In response to that SIS will give a prompt:

```
sis>
```

While SIS can read in inputs of circuits in various forms, truth tables, equations. 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 numerous methods of reading in, manipulating, and printing a collection of Boolean functions. At this point, you need only understand a subset of these options to use SIS to good effect. You can get a list of commands by typing

```
sis> help
```

For detailed instructions on a specific command type

```
sis> help [command]
```

For example,

```
sis> help read_pla
```

Here is a list of some of the command full names, and brief descriptions.

Full command name	Brief description
help	SIS on-line help
read_eqn	read equations from a file
read_library	read a gate library
write_eqn	write equations to a file
source	execute commands from a file
print	print sum-of-products
print_factor	print factored form of a node
print_gate	print gate information for a node
print_gate -s	print gate usage summary with area
sweep	eliminate single-input and constant nodes
map	map the current network onto the gates of a library
full_simplify -m nocomp *	perform Espresso minimization of all nodes
decomp	decompose all nodes by making new nodes for divisors

Exit SIS.

```
sis> exit
```

We will 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 will create an equation file for the full adder in the format expected by the Eqntott program. Eqntott can be used to convert from equations to truth tables for use in Espresso. However, SIS can read in equation files directly. We will create a Unix file named "`full-adder.eqn`". Inside the file we type two lines as follows:

```
cout = 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 omit the "`*`" sign and type this as

```
cout = 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 ;
```

However, be sure not to omit the spaces between the ANDed variables. Without them, SIS would treat *abc* as a single multi-character variable name instead of a product.

Save the file and exit your text editor. In the Eqntott input file, equations lines have the following format:

```
[signal] = [expr] ;
```

You can use the following operators within expressions:

Operator symbol	Operator name
( )	grouping
!= (or ^)	exclusive or
==	exclusive nor
! (or ') complement	
& (or * or simple juxtaposition)	boolean and
— (or +)	boolean or

Start SIS and read in the full adder equations.

```
sis
read_eqn full-adder.eqn
```

View the equations in sum-of-products form.

```
print
```

SIS should output the following text:

```
{cout} = 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 *cout* and *sum* are in curly brackets: these have been identified by SIS as output nodes, and are treated in a special way. View the equations in a factored form.

```
print_factor
```

```
{cout} = 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.

We are now ready to instruct SIS to decompose *co* and *sum* into a collection of simpler functions.

```
decomp *
print_factor
```

```
{cout} = a b ci′ + ci [2]
{sum} = ci [4]′ + ci′ [4]
[2] = b [3] + a b′
[3] = a′ + a
[4] = a b′ + a′ b
```

Three new functions have been introduced and are represented at nodes [2], [3], and [4]. You may get slightly different forms depending on your ordering of inputs, however, the result should be functionally equivalent and efficient.

Now that the function is broken down into reusable sub-functions, we need to simplify the sub-functions. One of the strengths of SIS is that it contains Espresso's optimization algorithms within itself. Invoke Espresso on all the nodes of current network

```
full_simplify -m nocomp *
print_factor
```

SIS should produce the following output

```
{cout} = ci [2] + a b
{sum} = ci [4]′ + ci′ [4]
[2] = [4]
[3] = -0-
[4] = a b′ + a′ b
```

Each of the sub-functions is simplified. However, [3] is now just 0, and [2] is redundant. Don't worry about the fact that one would expect [3] to be 1 because it was $\overline{a} + a$ before simplification. It isn't used by other nodes as input after simplification so its value is irrelevant.

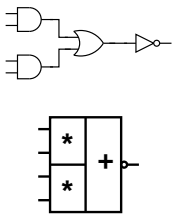
Get rid of these useless sub-functions.

```
sweep
print_factor
```

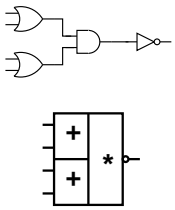
```
{cout} = ci [4] + a b
{sum} = ci [4]' + ci' [4]
[4] = a b' + a' b
```

Now we have a decomposed form, in which each of the nodes has been simplified and redundant nodes have been removed.

It is time to map the functions to gates. Technology mapping takes a set of (usually minimized) logic expressions and implements them in a particular technology, i.e., a particular set of gates. Let's map the full adder to a gate library from Mississippi State University. It includes various simple gates. Each gate is identified by a number (used by SIS), a name, and a brief functional description. In addition to NAND and NOR gates, the library also contains XOR, XNOR, AOI, and OAI gates. An AOI is an and-or-invert gate. An OAI is an or-and-invert gate. The diagrams for the functionality of these gates, as well as acceptable symbols, are given below.



AOI



OAI

Read in the library and map the functions to the gates within it.

```
read_library msu.genlib
map
print_factor
```

You can ignore the warnings "`map`" produces.

```
[348] = a b' + a' b
{cout} = [348] ci + a b
[345] = ci'
[364] = [348]'
{sum} = [345]' [348]' + [364]' ci'
```

It was necessary to add some nodes (gates) to implement them in terms of the gates in the library. Let's see which gate was used for each node.

```
print_gate
```

```
[348]      2310:physical   40.00
{cout}     1970:physical   56.00
[345]      1310:physical   16.00
[364]      1310:physical   16.00
{sum}      1860:physical   40.00
```

This commands prints out the gates used in deriving each of the non-leaf (non-literal) nodes of the networks. The first column gives the node names, the second column gives the gate type identification numbers, and the third column contains the area of the gates.

Get an area summary

```
print_gate -s
```

```
1310:physical   :    2 (area=16.00)
1860:physical   :    1 (area=40.00)
1970:physical   :    1 (area=56.00)
2310:physical   :    1 (area=40.00)
Total: 5 gates, 168.00 area
```

Find out what the delay is for *cout* and *sum*, using the library model

```
print_delay -a cout sum
```

```
{sum}     : arrival=( 4.40  4.40)
{cout}    : arrival=( 3.20  3.20)
```

The outputs will be ready 4.4 ns and 3.2 ns after an input change occurs.

Determine the power consumption using a general delay model from the MSU library.

```
power_estimate -d GENERAL
```

```
Combinational power estimation, with General delay model.
Network: full-adder.eqn, Power = 63.4 uW assuming 20 MHz clock and Vdd = 5V
```

Let's review what we did.

```
read_eqn full-adder.eqn
print
print_factor
decomp *
print_factor
full_simplify -m nocomp *
print_factor
sweep
print_factor
read_library msu.genlib
map
print
print_gate
print_gate -s
print_delay -a cout sum
power_estimate -d GENERAL
```

Reload the full adder, and map it to the MSU library *without* first decomposing or simplifying the expression. Get a gate area summary, output delay information, and a power estimation again. What happened?

Reload the full adder again. Simplify and sweep it without first decomposing it. Then map it to the library and check the gate area, output delay, and power consumption. What happened? How does this compare to the decomposed and simplified case? Howe does it compare to the totally unoptimized case? Is this what you would expect?

Start a new shell and view the MSU library file.

```
less /vol/cad/sis-1.2/sis/sis_lib/msu.genlib
```

You can lookup a gate's function by gate type identification number. For example, gate 1860 is a OAI (or-and-invert) gate. Between "`print`", "`print_gate`", and the library file, you have all the information to draw diagrams for the three implementations of the full adder you used SIS to produce. If you don't know how to draw a particular gate, take a look at Mano and Kime. See Chapter 3 for a number of examples. You can lookup a single gate by typing

```
grep [gate number] /vol/cad/sis-1.2/sis/sis_lib/msu.genlib
```

For a more automated solution, take a look at the "`/vol/ece303/bin/lookup-gate.perl`" script. If you put the output of the "`print_gate`" command into a file called "`gate_use`" and run this command

```
lookup-gate.perl /vol/cad/sis-1.2/sis/sis_lib/msu.genlib gate_use
```

It will tell you the type of gate used for each node in your circuit.

Verify that these implementations implement the full adder correctly.

Reload the full adder and automatically apply a MSU-oriented automatic optimization and mapping script that comes with MIS.

```
source script.msu
```

See what impact this had on area, delay, and power consumption.

End the SIS session

```
quit
```

4 The two-bit binary adder

Use what you have learned to map the two-bit adder described in Section 2 to the MSU library. The implementation should be reasonably efficient in terms of area, delay, and power consumption. You can use additional commands available within SIS. However, keep track of the commands use. It will be useful to use an external script and the "`source`" command if your list of commands is complicated.

5 Deliverables

1. The equations you produced for the two-bit adder using K-maps
2. The equations produced by Espresso for the two-bit adder
3. A sentence or two of explanation
4. The equations you produced for the priority encoder using K-maps
5. The equations produced by Espresso for the priority encoder
6. A sentence or two of explanation
7. SIS gate area, output delay, power consumption, equations, and circuit diagrams for
   (a) Decomposed and node-simplified full adder
   (b) Unsimplified full adder
   (c) Node-simplified full adder
   (d) Script-based minimization of full adder
   (e) Your high-efficiency two-bit adder
8. A list of commands used to simplify and map the two-bit adder
9. A paragraph or two describing your observations about Espresso, SIS, and anything interesting you encountered while doing the lab

A Viewing and editing text files with Unix

You can view text files with the "`less`" command. To get more information about less, type

```
man less
```

There are many text editors available in Unix. Emacs is a big, complicated, and extremely powerful editor tailored to programmers. Vi is small, a little bit cryptic, and also quite powerful. Pico is a small, simple, and easy to learn editor with limited capabilities. If you are going to program a lot in the future, it is probably worth learning emacs or vi. You can get more information about these editors by typing one of the following commands on an ECE Solaris machine:

```
info emacs
  man vi
 man pico
```

You can also see find FAQs on the web using a search engine, e.g., Google.

B Remote login

The programs you will be using can be found in the directory "`/vol/cad/sis-1.2/bin/`" on the SUN workstations in the Wilkinson Lab. If you are on a remote Unix machine, you can determine the IP addresses of a randomly selected machine by looking at the first address given by

```
nslookup solaris.ece.nwu.edu
```

Take a look at the "`/vol/ece/bin/get-sparc`" command if you want to see a cleaner way to get the same effect. SSH to the machine it gives you.

```
ssh [machine]
```

or, if you have access to "`get-sparc`", type

```
ssh `get-sparc`
```

using backtics around "`get-sparc`". Don't try to use rlogin or telnet. They're horribly insecure.