



INGENJÖRSHÖGSKOLAN  
HÖGSKOLAN I JÖNKÖPING

# Logic optimization using SIS

Laboratory 1

in course “Logic synthesis”

2001-version

Written by Tomas Bengtsson and Shashi Kumar

---

<b>1. Introduction</b>	<b>4</b>
1.1. Benchmark Circuits	4
1.2. Objectives	5
1.3. Documents	5
<b>2. Lab-instructions</b>	<b>6</b>
2.1. Task 1 Design of combinational logic	6
2.1.1. Preparations	6
2.1.2. Two level optimization	6
2.1.3. Information about BCD to 7-segment decoder	6
2.1.4. Multi level optimization	6
2.1.5. Theory about multiplier	7
2.2. Task 2 Design of sequential logic	8
2.2.1. Preparations	8
2.2.2. Lab task	8
2.2.3. Theory about pattern-recognizer	9
2.3. Task 3 Optimization of benchmarks	9
2.3.1. Preparations	9
2.3.2. Benchmark files	9
2.3.3. Two level optimization of combinatorial benchmarks	9
2.3.4. Multi level optimization of combinatorial benchmarks	9
2.3.5. Optimization of sequential benchmarks	10
<b>3. SIS and the computer-environment</b>	<b>10</b>
3.1. How to start up your UNIX account	10
3.2. Computers that can be used	10
3.3. How to login via "Telnet"	11
3.4. How to login via ReflexionX	12
3.5. How to adjust the system	14
3.6. How to transport files between Windows and Unix.	16
3.7. Handle SIS	17
3.7.1. Start and stop SIS	17
3.7.2. Use "SIS"	17
3.8. Small guide to Unix-commands	17
<b>4. File-formats for combinational logic</b>	<b>18</b>
4.1. "BLIF"-format for combinational logic	18
4.1.1. Example Full-adder	18
4.1.2. Comments in "BLIF"-format	19
4.1.3. The command "write_blif"	19
4.2. Hierarchical descriptions in "BLIF"-format	20

---

<b>4.3. File-format “PLA”</b>	<b>21</b>
4.3.1. The command “write_pla”	22
4.3.2. pla-format for functions with don’t cares	22
<b>4.4. The “EQN”-format</b>	<b>23</b>
4.4.1. The command ”write_eqn”	23
<b>5. File-format for sequential logic</b>	<b>24</b>
<b>5.1. “KISS2”-format, a subpart of “BLIF” to describe an FSM</b>	<b>24</b>
<b>6. SIS Commands</b>	<b>25</b>
<b>6.1. Input Commands</b>	<b>25</b>
<b>6.2. Processing Commands</b>	<b>25</b>
6.2.1. Two Level Optimization Commands	26
6.2.2. Multilevel Optimization Commands	27
6.2.3. FSM optimization commands	33
<b>6.3. Scripts</b>	<b>38</b>
6.3.1. Using a script to simplify work	38
6.3.2. A Script of SIS commands for Multi-Level Logic Optimization	39
<b>6.4. Output Commands</b>	<b>39</b>
<b>6.5. Status Checking Commands</b>	<b>40</b>
<b>6.6. Miscellaneous Commands</b>	<b>42</b>

---

Hint: There’s an English-Swedish dictionary on

<http://www-lexikon.nada.kth.se/skolverket/swe-eng.html>

## 1. Introduction

Before a system is implemented in hardware, it is required to get the structure of the system in terms of hardware components. Logic Synthesis generates the circuit in terms of gates and flip-flops. Generally, Logic Synthesis has cost minimization (in terms of number of gates or transistors) as its main objective. But sometimes, it is required to implement the circuit so that it has minimum delay or fastest clock. Before generating the final implementation, optimizing transformations are applied to the system representation so as to get an equivalent representation that leads to implementation meeting the desired objectives. If the size of the system is large then these techniques cannot be manually applied and computer tools are required to do this job.

SIS is a tool from University of Berkeley, California, which incorporates a set of Logic Optimization techniques. It has techniques for optimization and implementation of both Combinational Circuits (Boolean Functions) and Sequential Circuits (Finite State Machines). SIS uses special formats for representation of Boolean functions, combinational circuits and Finite State Machines (FSMs).

You will learn about the theory of techniques for Logic Synthesis in the lecture classes. The purpose of laboratory exercises is to get hands-on experience in using the SIS tool for Logic Synthesis.

### 1.1. Benchmark Circuits

To get experience with SIS tool, or CAD tools in general, we need to use a large number of example circuits. Generating these circuits is a time consuming activity. Many CAD tool designers and researchers have collected many such examples and designs and they have made them available on the Internet for other people to use. Such design examples are available at various levels of design and for various purposes. Such collections of design examples are called benchmark circuits. A large variety of combinational and sequential benchmark circuits are available for learning and experimenting with logic synthesis tools. The benchmark circuits include adders and multipliers of various types; encoders and decoders; controllers for various applications etc.

Benchmarks circuits are also used for comparing and evaluating the performance of various CAD tools or CAD algorithms.

## 1.2. Objectives

The objectives of the first laboratory are:

1. Learn formats for representation of combinational and sequential circuits.
  - a. Combinational Circuit: BLIF, PLA, EQUATION formats
  - b. Sequential Circuits: KISS
2. Learn SIS commands for combinational and sequential circuit optimization
3. Design and optimize a combinational circuit for a BCD to 7-segment display decoder using SIS
  - a. Describe the circuit in PLA format
  - b. Optimize the circuit for two-level (PLA) implementation
4. Design and optimize a combinational circuit for 4 bit multiplier using SIS
  - a. Describe the circuit in BLIF format
  - c. Optimize the circuit for multi-level optimization
5. Design and optimize a sequential circuit
  - a. Describe the FSM using KISS format
  - b. Optimize using SIS commands
6. Practice the use of SIS tool to optimize three combinational benchmark circuits and three sequential benchmark circuits.
7. Estimate the advantage obtained by the use of SIS optimization tool in different types of designs.

## 1.3. Documents

A form is provided from this laboratory in which you should fill in your results. One sheet of paper written by Arne Ståhl about how to open the UNIX account will also be provided, see section “3.1 How to start up your UNIX account” in this manual.

There is also a home page, which contains the lab manuals and some related documents. Its address is “[http://hem.hj.se/~beto/courses/logic\\_optimization/main.html](http://hem.hj.se/~beto/courses/logic_optimization/main.html)”. There you can find among other things, a link to a document about SIS and one document about BLIF format. In Pingpong you can find those documents converted to pdf-format.

## 2. Lab-instructions

### 2.1. Task 1 Design of combinational logic

#### 2.1.1. Preparations

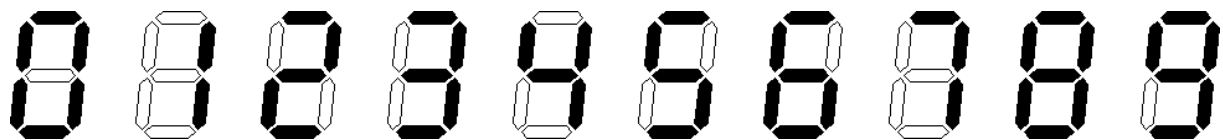
In this task you should design a BCD to 7-segment display decoder. Before you come to the lab you should have described two versions of it in PLA-format, see part “2.1.3 Information about BCD to 7-segment decoder” in this manual. Don’t try to optimize it by hand when you are writing the PLA-format, just write it in the way that feels most simple for you. Then in the lab, the tool will help you to optimize.

You should also design a 4-bit unsigned multiplier and describe it in “BLIF”-format. It should be a pure combinational multiplier without memory-elements. Before you come to the lab you should have thought out how this multiplier should be built and also written this in “BLIF”-format. A 4-bit multiplier has two 4-bit numbers as input. As output it has as many bits as needed to be able to represent every possible product.

#### 2.1.2. Two level optimization

Use “SIS” to optimize your two versions of the BCD to 7-segment decoder. Make two level optimization and fill in the required data in the hand in form.

#### 2.1.3. Information about BCD to 7-segment decoder



A BCD to 7-segment decoder is a combinational circuit with four inputs and seven outputs. Its purpose is to convert a BCD number to information to every segment telling if it should be on or off. In some decoders if the inputs are between 1010 and 1111 the decoder outputs values making the display showing the corresponding hexadecimal letters. When this is not needed it is however better to let the outputs be don’t cares, to make the logic smaller. In this lab you should make the outputs don’t cares for inputs 1010 and higher. Then you should make two version of the decoder, one where the don’t cares is forced to zero and one where they are treated as don’t cares.

#### 2.1.4. Multi level optimization

First you should validate that your multiplier works, in other words check that the function is correct. Use the “SIS”-command “simulate” which is described in part “6.6 Miscellaneous Commands” in this document. (You don’t have to simulate your BCD to 7-segment decoder in this way.)

Use “SIS” to optimize your 4-bit multiplier. Use a sequence of multi level optimization operations so you get less than 167 literals. Read in your multiplier again and optimize it with rugged script. Compare the result from your optimization with the result rugged script gives. Write in the “hand-in” form what is required.

Next task is to convert the multiplier so it gets only two levels of logic. Use the command “*reduce\_depth*” to achieve this. Fill in the “hand-in” form how many literals you get.

## 2.1.5. Theory about multiplier

Here is an example of multiplying two binary numbers, which shows the principal of multiplying. Assume that 1011 and 1101 should be multiplied. A simplifying observation is that when two binary digits, which only can have value “zero” and “one”, are multiplied, it is similar to the and-function.

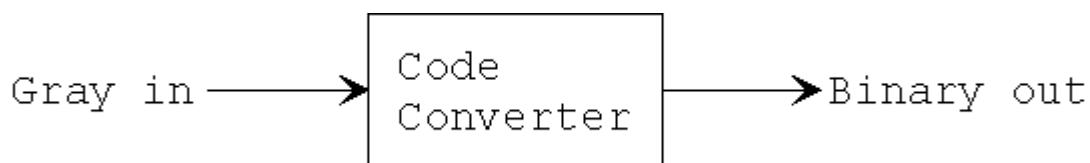
$  \begin{array}{r}  1\ 0\ 1\ 1 \\  * 1\ 1\ 0\ 1 \\  \hline  \end{array}  $	Write the calculation like this.
$  \begin{array}{r}  1\ 0\ 1\ 1 \\  * 1\ 1\ 0\ 1 \\  \hline  1\ 0\ 1\ 1  \end{array}  $	Take the least significant digit, the most right, in the number written on the lower line and multiply it with the upper number.
$  \begin{array}{r}  1\ 0\ 1\ 1 \\  * 1\ 1\ 0\ 1 \\  \hline  1\ 0\ 1\ 1 \\  0\ 0\ 0\ 0 \\  1\ 0\ 1\ 1 \\  1\ 0\ 1\ 1  \end{array}  $	Continue and multiply the number on the upper line with every digit on the lower line and write the products below and adjusted to the left which means multiplied with 10, 100 and so on.
$  \begin{array}{r}  1\ 0\ 1\ 1 \\  * 1\ 1\ 0\ 1 \\  \hline  1\ 0\ 1\ 1 \\  0\ 0\ 0\ 0 \\  1\ 0\ 1\ 1 \\  + 1\ 0\ 1\ 1 \\  \hline  1\ 0\ 0\ 0\ 1\ 1\ 1\ 1  \end{array}  $	Add those numbers that you’ve got now with respect to their position.

When designing the multiplier, don’t forget to think about where carries can come out from adders and how they should be connected to get the multiplier work.

## 2.2. Task 2 Design of sequential logic

### 2.2.1. Preparations

In this task you should design a four bit sequential circuit for Gray-code to binary code conversion. The Gray code is sent into the code converter bet serially as well as the binary output is sent bit serially from the code converter. In both input and output the bits are starting with most significant bit and ending with the least significant bit. There is no pause between data words coming into the code converter. In other words one clock cycle after the least significant bit is put into the code converter, the most significant bit of the subsequent word is enters the converter.



If this converter should work in reality it is important to force it to a specific state when it starts up. Anyway in this lab we don't bother about how to get there but just assuming that we can get there in some way.

The properties of the Gray code is in a way such that the conversion described above can be done without any delay from input to output in terms of clock cycles. That is only when bits comes starting with most significant bit and ending with least significant bit.

Before the lab, you should design a FSM that makes a converter as described above. It should not have any delay from input to output in terms of clock cycles. Observe that you should not try to make the number of states in the state machine small. This work you should let the SIS-tool do when you are at the lab. You should also have made a description of this FSM in "KISS2"-format.

In the "hand-in" form you should show how the state diagram look like before optimization.

### 2.2.2. Lab task

Use "SIS" to minimize number of states in the FSM. Then let "SIS" assign the coding of the states. Try both to assign for minimal number of flip-flops and for one-hot encoding. Try to minimize the logic in the FSM in both cases. Write down required data in the "hand-in" form.



### 2.2.3. Theory about pattern-recognizer

Gray code	Binary code
0000	0000
0001	0001
0011	0010
0010	0011
0110	0100
0111	0101
0101	0110
0100	0111
1100	1000
1101	1001
1111	1010
1110	1011
1010	1100
1011	1101
1001	1110
1000	1111

The table above shows a conversion table between four bit Gray code and four bit binary code. The Gray code is used in some devices where it is required that only one bit differs between adjacent code words.

## 2.3. Task 3 Optimization of benchmarks

### 2.3.1. Preparations

Read about benchmarks in part 1.1 *Benchmark Circuits*.

### 2.3.2. Benchmark files

In this task you should optimize some benchmarks with “SIS”. In directory “/home/beto/public/logic\_synthesis/benchmarks” the benchmarks are stored. Make appropriate directories in your home directory and copy the benchmarks to them.

The benchmarks are also possible to download from Internet at address:  
<http://www-cad.eecs.berkeley.edu/Software/software.html>

Some of the benchmarks may not work so in this case just try another one.

### 2.3.3. Two level optimization of combinatorial benchmarks

You should select three of the benchmarks of combinational circuits described in pla-format and do two-level optimization of them. Fill in the number of product terms before and after minimization in the hand in form.

### 2.3.4. Multi level optimization of combinatorial benchmarks

You should select three of the benchmarks of combinational circuits described in blif-format. Try to use the same sequence of commands you used for optimizing the multiplier. Optimize them also with help of rugged-script. Fill in the required information in the hand in form.

### **2.3.5. Optimization of sequential benchmarks**

Select three of the sequential benchmarks. Use “SIS” to minimize number of states. Then let “SIS” assign the coding of the states. Try both to assign for minimal number of flip-flops and for one-hot encoding. Try to minimize the logic in the FSM in both cases. Write down required data in the hand in form.

## **3. SIS and the computer-environment**

The software “SIS” has to be run under Unix. It is a text-based software so using “Telnet” will be suitable but if you can use “ReflexionX” it will probably look better.

### **3.1. How to start up your UNIX account**

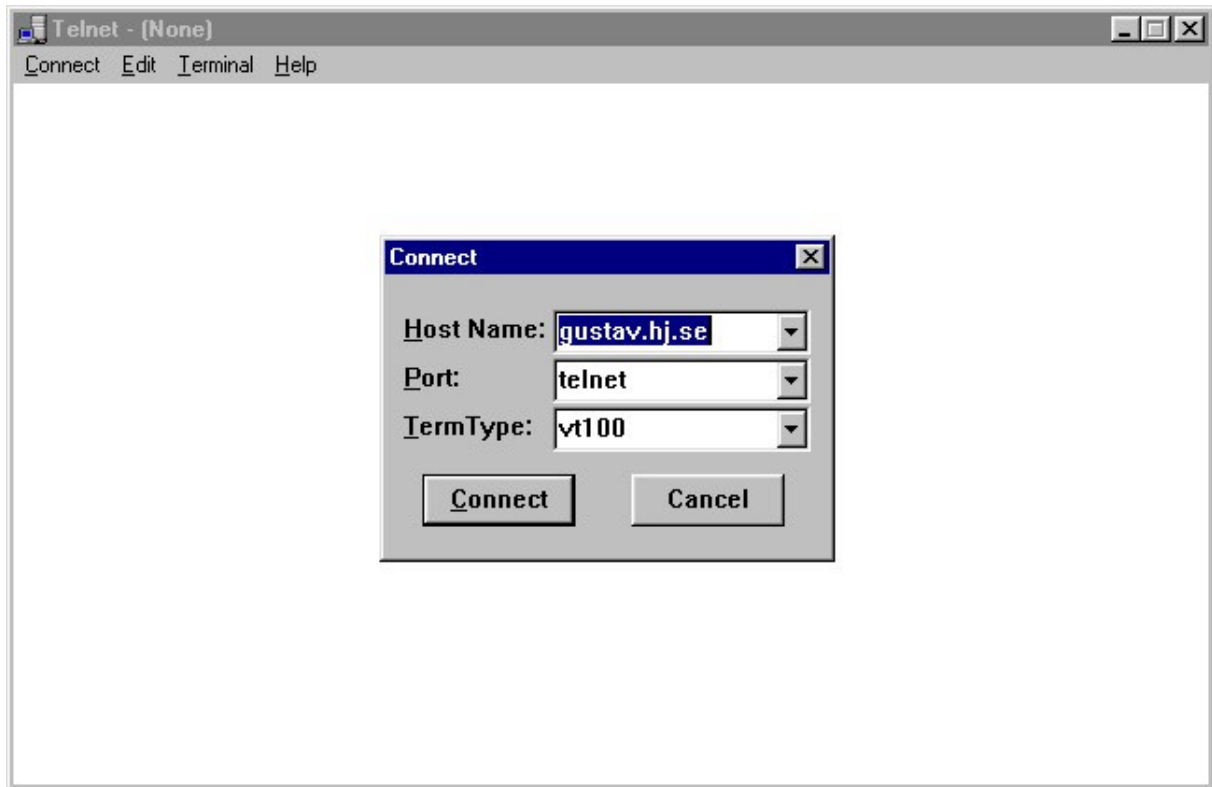
To get you UNIX work you first have to telnet to “junic”. For instructions about this see the one page document “SYNKRONISERA LÖSENORD (NISCLIENT)” made by Arne Ståhl. The username is the same as in Novell network and the password is the password you got from the beginning when you get your computer account to the Novell network. If you have forgotten you can ask the lab assistant.

### **3.2. Computers that can be used**

In the text below it is assumed that you make remote-login to the computer “gustav.hj.se”. You should replace this in the description below with “kronblom.hj.se” because the computer “gustav.hj.se” is not so good.

### 3.3. How to login via “Telnet”

To start “Telnet” choose “Run” from “Windows” start-menu. Key in “telnet” and press *return*. Select “Connect” -> “Remote system...”. Then fill in the up-popping window as the picture below shows, click on “Connect” and follow the instructions.

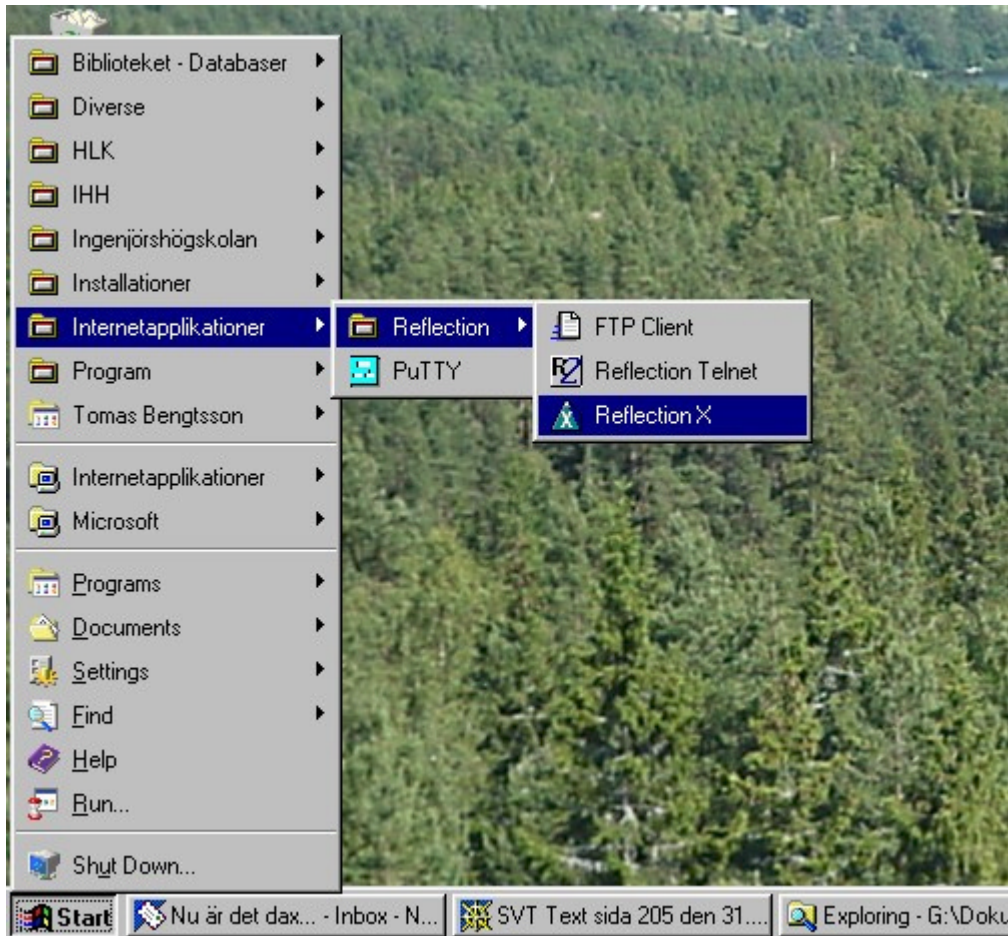


You will hopefully now get into a Unix-environment.

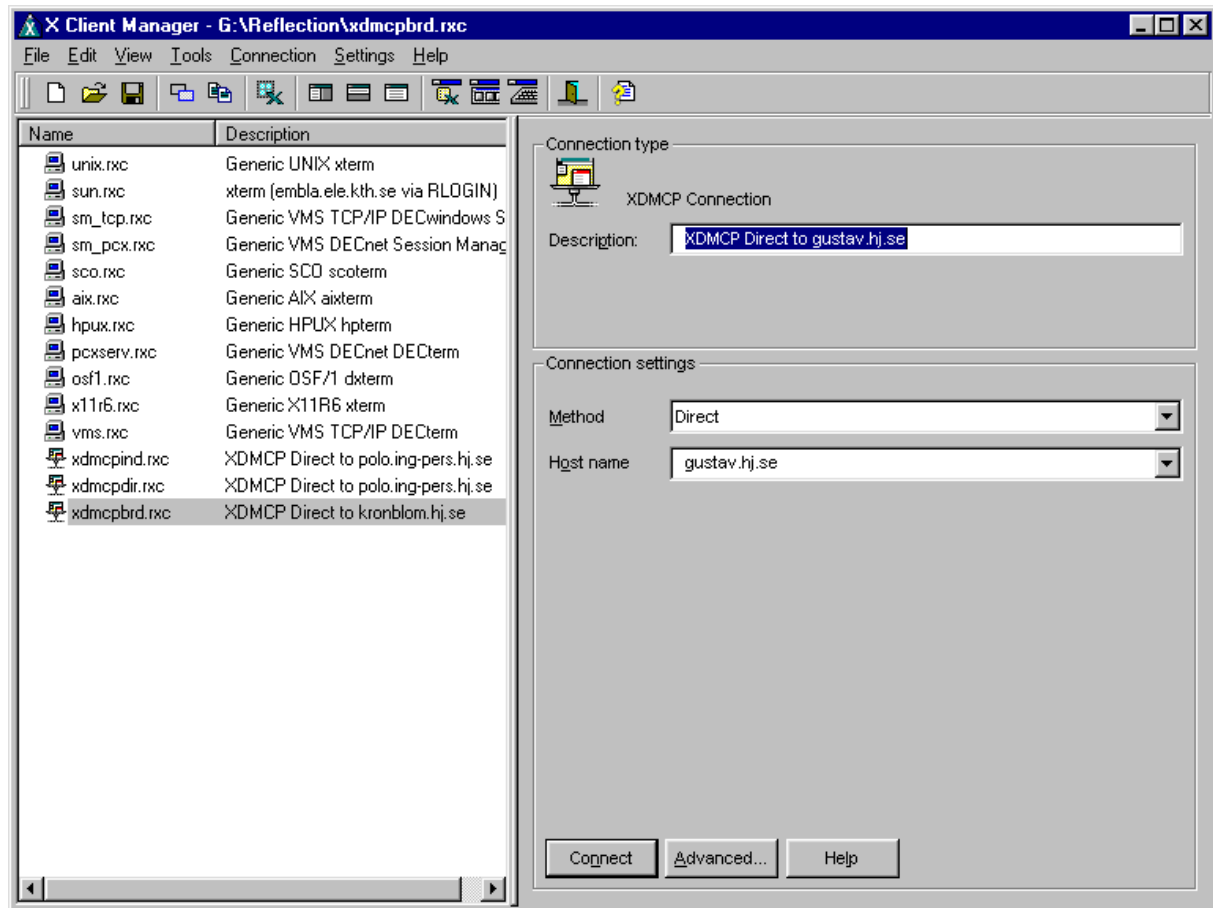
### 3.4. How to login via ReflexionX

The picture below shows how to start “ReflexionX”. It maybe looks different on some computers. Sometimes this program may hang the PC.

If you are using a computer with Windows98 the resolution of the screen maybe adjusted lower than 1024\*768 pixels. If this is the case, you get a better-looking UNIX environment if you change to 1024\*768 pixels before you start ReflexionX. If you don’t know how to do this in Windows98 you can ask the lab assistant.

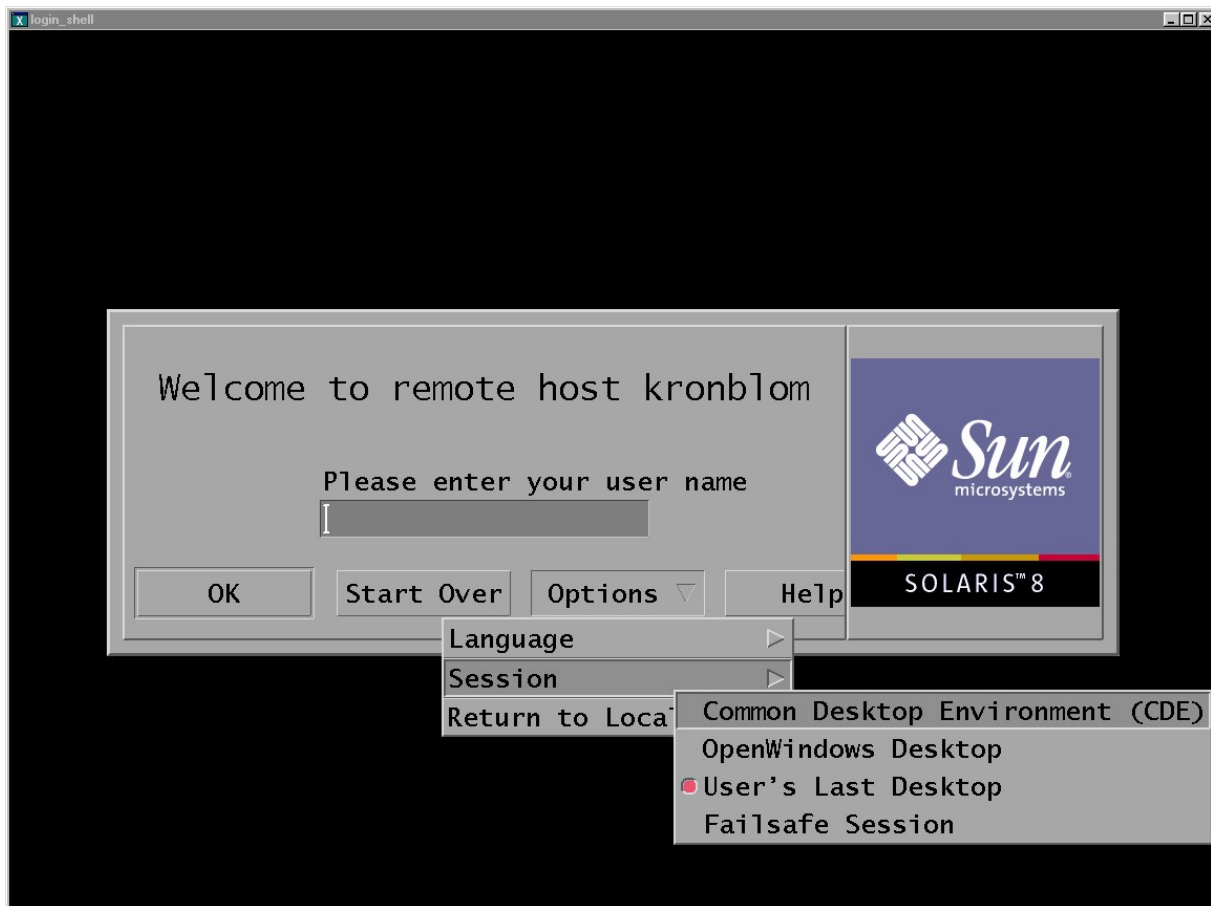


When this program is started it looks as below



Click on some of the lines starting with *XDMCP* in the left. Fill in the settings in the square *Connection settings* in the right lower corner. Then click on *connect*. Maybe you have to experiment and try some other settings in ReflectionX to get in.

You will come to a login screen. Before you log in you can choose between two different window-system from a menu, “Open Windows” and “Common Desktop Environment (CDE)”. We recommend you to use “Common Desktop Environment (CDE)” because we have found it difficult to handle “Open Windows” when logging in via “ReflectionX”. Normally if you are logging in you get into the window-system you used last time. The picture below shows how to select “Common Desktop Environment (CDE)”.



When you have logged in to the Unix, right-click somewhere and you will get a menu. To get a terminal-window choose *tools* and then *terminal*.

## 3.5. How to adjust the system

When you are coming in to the Unix-system, you will probably come into “C-shell” which does not have some good features in our system. This is the case if the prompt is a “\$”-sign.

To get a more user-friendly system you can run a program that adjusts the environment. This you should run only once. Do this by typing: *(Press return after every line!)*

```
cd /usr/sw/
./install_modules.sh
```

When you have done this you can get into a better shell by typing *bash* but before that it is good to make file, which makes the path to “SIS” and some other good things, when you start *bash*. You can do that by coping a file which already exists. Do it by typing the following command:

```
cd  
cp /home/beto/public/.bashrc .
```

You can now give the command:

```
bash
```

If everything has worked you can now run the editor *emacs* by typing:

```
emacs
```

If you have logged in via “ReflexionX” you can put the symbol “&” after if you want to start *emacs* as an independent window, otherwise you are not able to use the terminal you started *emacs* from until you exit *emacs*.

If you want whit background instead of white you can type:

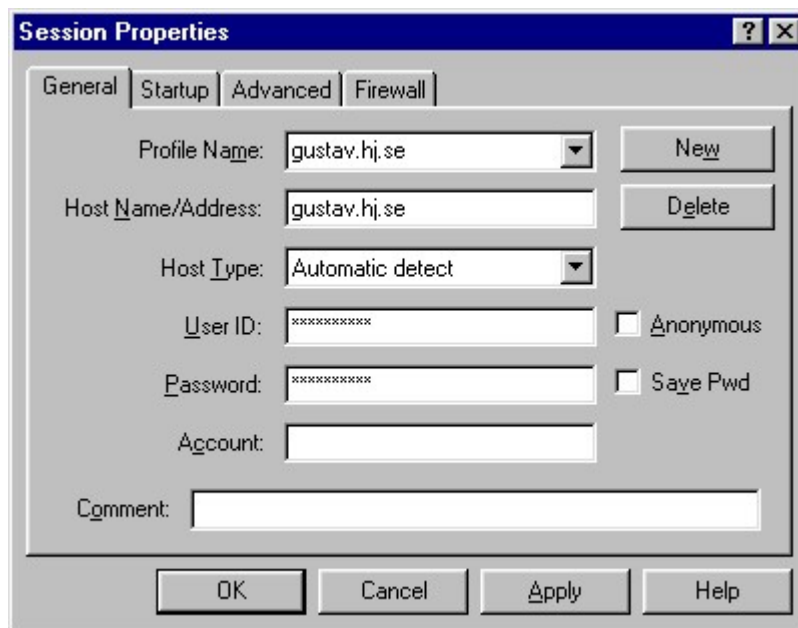
```
emacs -bg white
```

### 3.6. How to transport files between Windows and Unix.

If you want to transport files between windows and UNIX, for example descriptions of circuits you have done, you can use an ftp-software. Start this software from Windows start menu by choosing:

Start -> Internetapplikationer -> Ws\_ftp -> WS\_FTP95 LE

Observe that there are two “Internetapplikationer” and you should choose the lower one to find this. You should then fill in the box as shown in the figure below but change the stars to your Unix-account user-id and password.



When you’ve done this and click on “OK” you’ll get a window, which shows your windows-files to the left and unix-files to the right. If you want to copy a file from one system to another you just mark it and click on the arrow-button in the middle pointing in the direction in which the copying should be done.



### 3.7. Handle SIS

#### 3.7.1. Start and stop SIS

To start “SIS” you should type “sis” in the Unix-prompt and then you’ll get in if the settings described in part 3.5 has worked out. To exit “SIS”, write the command quit.

#### 3.7.2. Use “SIS”

The “SIS” user-interface is a command-based software. This means that you have a command prompt in which you have to write commands to make things happen.

The normal backspace and arrow key does not work in SIS. The two following key combinations could be used for correcting mistakes.

```
Ctrl - h  backspace
Ctrl - w  delete last word
```

There is not much help included in the SIS software but you can at least get a list of available commands by giving command *help*.

If you read in a file that have some errors in the format SIS may get into a state that makes it behave strange. In this case quit SIS and start it again. There could be other similar strange behavior by other mistake commands.

### 3.8. Small guide to Unix-commands

Here follows a list with some Unix-commands. Observe that Unix is sensitive to if you use capital letters or small letters.

UNIX-command	Explanation or corresponding DOS-command
ls	dir /w
ls -o	dir
pwd	Check in which directory you are
cd	cd
cd ..	cd.. (Notice: You need a space between cd and the dots in unix)
mv	ren
rm	del
mkdir	md
rmdir	rm
cp	copy
/	\ (This is not a command, but observe that the slash in unix has the other orientation comparing to dos.)

One useful thing in Unix is when you are typing in a file-name you can write the beginning and then press *tab*. Then the operating system automatically appends the rest of the letters in the file name (if there is no ambiguity in the file name). For example if you want to use a file “my\_circuit.blif” you can just type “my” and press tab.

## 4. File-formats for combinational logic

### 4.1. “BLIF”-format for combinational logic

The BLIF (Berkeley Logic Interchange Format) is a format for describing combinational circuits as a network of nodes. Each node is a single output function and is described as a truth table. The truth table has entries for only those input combinations for which the output is “1”. A bar “-” can be used as a “don’t-care” on an input.

#### 4.1.1. Example Full-adder

A full-adder has three inputs, let us call them “i1”, “i2” and “cin” where “cin” means carry-in. It has two outputs “sum” and “cout”. The truth table for this is:

i1	i2	cin	sum	cout
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

In BLIF-format this can be described as shown below. The orders in which the signals are written on the line starting with “.names” are the order the ones and zeros in the table are interpreted.

```
.inputs i1 i2 cin
.outputs sum cout

.names i1 i2 cin sum
001 1
010 1
100 1
111 1

.names i1 i2 cin cout
110 1
101 1
011 1
111 1

.end
```

We can also use the don't-care-character “-” and then it can be written as follows, which are equivalent with the code on the previous page.

```
.inputs i1 i2 cin
.outputs sum cout

.names i1 i2 cin sum
001 1
010 1
100 1
111 1

.names i1 i2 cin cout
11- 1
1-1 1
-11 1

.end
```

#### 4.1.2. Comments in “BLIF”-format

Comments can be put in to a file in “BLIF”-format. A comment starts with “#” and last to the end of line.

#### 4.1.3. The command “write\_blif”

The command “write\_blif” writes the description of the circuit in “BLIF”-format. There is one more line, “.model”, which only tells the name of the circuit and it will automatically be the file-name if nothing else is specified.

```
sis> write_blif
.model fa.blif
.inputs i1 i2 cin
.outputs sum cout
.names i1 i2 cin sum
111 1
001 1
010 1
100 1
.names i1 i2 cin cout
111 1
011 1
101 1
110 1
.end
```

## 4.2. Hierarchical descriptions in “BLIF”-format

When having larger systems it’s sometimes makes easy to be able to describe systems in a hierarchical manner. To illustrate this a four-bit adder is used which is design with four full-adders. The code below shows how it can be described in “BLIF”-format.

```
.inputs a3 a2 a1 a0 b3 b2 b1 b0
.outputs s3 s2 s1 s0 cout

.subckt fa a=a0 b=b0 cin=zero    sum=s0 cout=cout0
.subckt fa a=a1 b=b1 cin=cout0  sum=s1 cout=cout1
.subckt fa a=a2 b=b2 cin=cout1  sum=s2 cout=cout2
.subckt fa a=a3 b=b3 cin=cout2  sum=s3 cout=cout

.names zero

.end

.model fa

.inputs a b cin
.outputs sum cout

.names a b cin sum
001 1
010 1
100 1
111 1

.names a b cin cout
11- 1
1-1 1
-11 1

.end
```

A description of a sub-cell starts with “.model” followed by a name on the sub-cell. It ends with “.end”.

Before the first “.end” the top-level logic is described. A sub-cell is added as an instance by first writing the key word “.subckt”. After that the name of the instance should be written. Then a description follows that tells which signal in the top-block should be connected to which in the sub-cell. The signal-name to the left of the “=”-sign is the name in the sub-cell and the signal-name to the right is the name in the top-block.

Multi-level hierarchy is possible to use in “BLIF”-format.

The table below shows how to force a signal to “one” or “zero”.

.names s	Assign constant value “0” to the signal “s”.
.names s	Assign constant value “1” to the signal “s”.
1	

## 4.3. File-format “PLA”

The “PLA”-format is quite similar to the “BLIF”-format. The example below is a description of the full-adder described in section “4.1.1 Example Full-adder”.

```
.i 3
.o 2

.ilb i1 i2 cin
.ob sum cout

100 10
010 10
001 10
111 10
011 01
101 01
110 01
111 01

.e
```

```
.i 3
.o 2

.ilb i1 i2 cin
.ob sum cout

100 10
010 10
001 10
-11 01
1-1 01
11- 01
111 11

.e
```

The first two rows “i” and “o” describes how many inputs and outputs the circuit has. The two following rows, “.ilb” and “.ob”, defines the names of the inputs and outputs. The definition of names is not needed.

The “ones” and “zeros” in the table is the description of the logical function. The digits to the left are the inputs and the digits to the right are the outputs. A row in the table means that for the specified input-combination the outputs marked with “1” in the output-column should be “one”. The “zeros” in the output-column have another meaning. A “zero” there for an output, in a row, means that this row does not affect the function of that output. This can be a little misleading if you don’t know it. All combinations where an output has not been declared to be “one”, the output becomes “zero”.

It’s also possible to use “-“ in the inputs to represent “don’t-cares”. If a combination of input should set more than one output to “one”, it can be described in one row. The example to the right in the squares above shows the same function, full-adder, as to the left, but described using don’t cares.

## 4.3.1. The command “write\_pla”

The command “write\_pla” writes the description of the circuit in “PLA”-format. There is one more line, “.p”, which only tells how many product-terms are there in the representation.

```
sis> write_pla
.i 3
.o 2
.ilb i1 i2 cin
.ob sum cout
.p 8
111 10
001 10
010 10
100 10
111 01
011 01
101 01
110 01
.e
```

## 4.3.2. pla-format for functions with don't cares

A function that has “don't cares” is a function that for some combination of inputs, the output doesn't care. It is possible to use pla-format to describe this type of functions. If you want to specify that a combination of input is a “don't care” you do as when you specify that it is one but instead of writing “1” in the output part you write “-” or “2”.

The example below shows a function and its corresponding pla-file.

	$x_1$	0	1
$x_2$			
0		0	1
1		1	-

```
.i 2
.o 1
.ilb x1 x2
.ob out
01 1
10 1
11 -
.e
```

## 4.4. The “EQN”-format

The “EQN”-format stands for equation-format. This is a way to describe a logical network using Boolean equations. The example below shows how the full-adder can be described in the “EQN”-format.

```
sum = i1*i2!*cin + !i1*i2!*cin + !i1*!i2*cin + i1*i2*cin;
cout = i1*i2!*cin + i1*!i2*cin + !i1*i2*cin + i1*i2*cin;
```

Observe that every equation should be ended by “;”. The operators are:

!	Inverse
*	And
+	Or

It’s also possible to write equation with intermediate nodes. SIS computes in that case which nodes are outputs and inputs to the system.

```
sum = i1*i2!*cin + !i1*i2!*cin + !i1*!i2*cin + node*cin;
cout = node*!cin + i1*!i2*cin + !i1*i2*cin + node*cin;
node = i1*i2;
```

### 4.4.1. The command “write\_eqn”

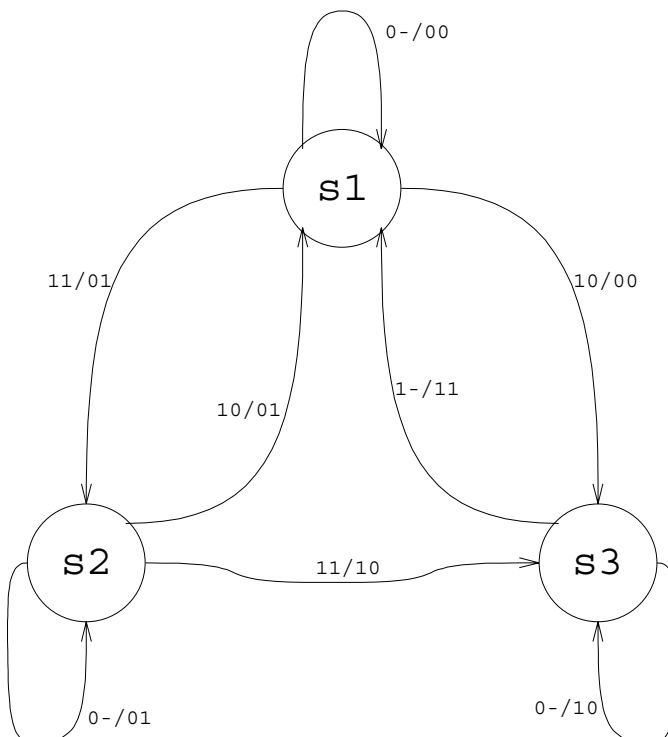
This command writes the system on equation-form. There are two lines in the beginning describing which nodes are inputs and which are outputs.

```
sis> write_eqn
INORDER = i1 i2 cin;
OUTORDER = sum cout;
node = i1*i2;
sum = i1*i2!*cin + !i1*i2!*cin + !i1*!i2*cin + cin*node;
cout = i1*!i2*cin + !i1*i2*cin + !cin*node + cin*node;
```

## 5. File-format for sequential logic

### 5.1. “KISS2”-format, a subpart of “BLIF” to describe an FSM

The format “KISS2” is a sub format of “BLIF”. It is used to describe finite state-machines. Because it is a sub format to “BLIF” you can get some useful information with help of the command “write\_blif”. It’s described in a way independent of the encoding of the states. The example below helps to describe the format.



```

.start_kiss
.i 2
.o 2
.r s1
0- s1 s1 00
10 s1 s3 00
11 s1 s2 01
0- s2 s2 01
10 s2 s1 01
11 s2 s3 10
0- s3 s3 10
1- s3 s1 11
.end_kiss
.end
  
```

```

.i 2 (number of inputs)
.o 2 (number of outputs)
.r s1 (defines start-state, useful during simulation)
  
```

The rows after that describes:

*inputs current\_state next\_state outputs*

So the second line in this section,

```
10 s1 s3 00
```

means that in state *s1* when the input is *10* next state will be *s3* and the outputs *00*.



## 6. SIS Commands

### 6.1. Input Commands

All commands starting with “read” (for example “read\_kiss”) fall in this category. One of these commands must be run to convey the input specification to the circuit. A file name containing the input specification must be specified along with this type of commands. You can learn more about various formats to specify a circuit from a document downloadable from <http://www.bdd-portal.org/docu/blif/blif.html>.

#### **read\_blif**

Read the circuit, which is described in BLIF format. For example, if the circuit in blif format is available in a file **ckt.blif**, the command will be:

```
read_blif    ckt.blif
```

There are similar commands to read the specification of the circuit in other formats.

#### **read\_astg**

Read the specification given in Asynchronous State Transition Graph format.

#### **read\_eqn**

Read the specification given in Equation format.

#### **read\_kiss**

Read the specification given in KISS format.

#### **read\_pla**

Read the specification given in PLA format.

### 6.2. Processing Commands

These commands can be run only after input is conveyed to SIS through one of the input commands. If input is in state machine type, which is in KISS2 or ASTG format, first a circuit is to be generated and then optimization commands can be run. However, if input is already a circuit specification, only optimization commands need to run. (For circuit generation, commands like ‘state\_minimize’ have to be used. For optimization, commands like ‘full\_simplify’ are used).

## 6.2.1. Two Level Optimization Commands

### espresso

There is one “espresso” command that could be run from the UNIX prompt outside SIS. There is also one that works inside SIS. Unfortunately they behave differently. The one inside SIS divides multiple output functions into many single output functions and optimize them separately. The “espresso” command in the UNIX-prompt can however optimize a multiple output function so it takes advantage from common implicants between the outputs. In this lab you should use “espresso” at the UNIX prompt outside SIS.

The espresso command is used for generating minimum cost circuit for “two level implementation”. The command minimizes the number of product terms needed in “two level optimization”. The output of “espresso” is an irredundant prime cover, often minimum in cardinality.

For example, consider the following circuit in PLA format. It has six product terms.

```
bash-2.04$ cat example.pla
.i 3
.o 2
.ilb a b c
.ob f1 f2
100 10
101 10
111 10
011 01
010 01
111 01
.e
bash-2.04$
```

After using the Espresso command, the minimized circuit will be

```
bash-2.04$ espresso example.pla
.i 3
.o 2
.ilb a b c
.ob f1 f2
.p 3
111 11
10- 10
01- 01
.e
bash-2.04$
```

Notice the number of product terms after “two level” optimization has been reduced from six to three.

If you want the result in a file instead on the screen you can add a “>” symbol and then the name of the file. The box below shows an example.

```
bash-2.04$ espresso example.pla > name_of_out_file
```

### 6.2.2. Multilevel Optimization Commands

#### **decomp**

This command decomposes an internal vertex into more than one internal node. Many times it leads to reduction in number of literals. But this command, in general, leads to increase in delay of the network.

Example : Consider a circuit in equation format:

```
sis>write_eqn
INORDER = b e c d a;
OUTORDER = v;
v = !e*a + !c*d + b*d + e!*c + b*e;
sis>
```

After decomposition using decomp command:

```
sis> decomp
sis> write_eqn
INORDER = b e c d a;
OUTORDER = v;
v = [2]*[1] + !e*a;
[1] = d + e;
[2] = !c + b;
sis>
```

## eliminate k

It removes internal vertices from the network if its removal will not increase the number of literals by more than  $k$ . The variable corresponding to the vertex is replaced by the corresponding expression in all its occurrences. This command is reverse of **decomp**. This command helps in reducing the delay of the network.

Let the starting network in Equation format be:

```
sis> write_eqn
INORDER = b e c d a;
OUTORDER = v;
v = [4]*[3] + !e*a;
[3] = d + e;
[4] = !c + b;
sis>
```

After applying **eliminate 2** command, the new network will be:

```
sis> write_eqn
INORDER = b e c d a;
OUTORDER = v;
v = d*[4] + e*[4] + !e*a;
[4] = !c + b;
sis>
```

Notice that node [3] has been eliminated.

---

## **simplify and full\_simplify**

These command are used to simplify the specification of each of the node in the network.

Initial network:

```
sis> write_eqn
INORDER = a b c;
OUTORDER = v w;
v = a*!b*c + !a*!c + a*b;
w = !a*!b + a;
sis>
```

After using **simplify** command

```
sis> write_eqn
INORDER = a b c;
OUTORDER = v w;
v = b*w + !a*!c + a*c;
w = !b + a;
sis>
```

Sometimes **full\_simplify** may lead to better results than **simplify**.

## **invert**

Implement the ‘inverse’ of the node. Many times it simplifies the specification of network.

Consider the following function:

```
sis> write_eqn
INORDER = a c d e g b;
OUTORDER = f;
F = g*b + e*b + d*b + c*b + a*g + a*e + a*d + a*c;
sis>
```

By using **invert f** command, we get:

```
sis> write_eqn
INORDER = a c d e g b;
OUTORDER = f;
[0] = !c*!d*!e*!g + !a*!b;
f = ![0];
sis>
```

## **invert\_io**

Use complemented input variables or produce a complemented output.

---

**fx**

This command extracts common sub-expressions among the nodes and rewrites the nodes of the network in terms of common sub-expressions. The following example illustrates the use of **fx** command.

```
sis> read_eqn book_example.eqn
sis> write_eqn
INORDER = a b c d e;
OUTORDER = w x y z;
w = a*!e + !c*d + b*d + !a*d;
x = d*e + c*e + !b + !a;
y = b*d + a*d + b*c + a*c + e;
z = c + b + a;
sis> fx
sis> write_eqn
INORDER = a b c d e;
OUTORDER = w x y z;
w = a*!e + !c*d + b*d + !a*d;
x = e*[4] + !b + !a;
y = [4]*[5] + e;
z = [5] + c;
[4] = d + c;
[5] = b + a;
sis>
```

## resub

The command **resub** try to substitute expression corresponding to one node in expressions corresponding to other nodes. The purpose is to reuse the expression of a node as sub expression in some other nodes. This step is expected to reduce the number of literals in the circuit.

For example, consider the following network in equation format.

```
sis> write_eqn
INORDER = a c d b;
OUTORDER = u v;
u = c*b + a*d + a!*c;
v = d + !c;
sis>
```

After using **resub** command, the network will be

```
sis> write_eqn
INORDER = a c d b;
OUTORDER = u v;
u = a*v + c*b;
v = d + !c;
sis>
```

Notice that node “u” has been rewritten in terms of “v”.

## sweep

Sweep command eliminates all single input vertices and those with a constant value.

Consider the following network:

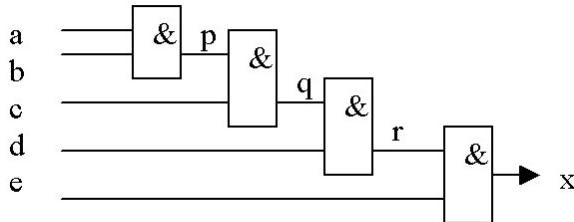
```
sis> write_eqn
INORDER = a c d b;
OUTORDER = u v;
u = a*v + c*b;
v = [2];
[2] = d + !c;
sis>
```

After applying sweep command, the new network will be

```
sis> write_eqn
INORDER = a c d b;
OUTORDER = u v;
u = a*v + c*b;
v = d + !c;
sis>
```

## reduce\_depth

This command is used to control the delay of the network during multi-level logic optimization. The following example illustrates the use of this command.



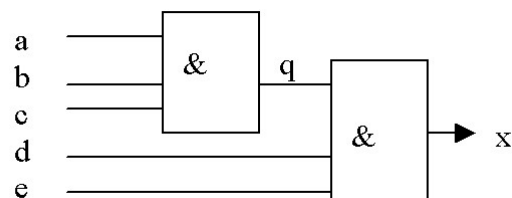
```

sis> read_blif reduce_depth_example.blif
sis> write_blif
.model rd.blif
.inputs a b c d e
.outputs x
.names e r x
11 1
.names a b p
11 1
.names c p q
11 1
.names d q r
11 1
.end
sis> print_level
0: a e d c b
1: p
2: q
3: r
4: {x}
  
```

The network has depth of 4 levels. The depth can be reduced to 2-levels by using the command “*reduce\_depth -d k*”, where *k* gives the number of levels in the reduced network.

```

sis> reduce_depth -d 2
sis> print_level
0: b e a d c
1: q
2: {x}
sis> write_blif
.model rd.blif
.inputs a b c d e
.outputs x
.names d e q x
111 1
.names a b c q
111 1
.end
  
```





## Rugged script

Rugged-script is not a command but a script containing a sequence of multilevel optimization commands. Read more about this in section “6.3.2 A Script of SIS commands for Multi-Level Logic Optimization”.

### 6.2.3. FSM optimization commands

#### **state\_minimize**

Minimizes the number of states in a given FSM. Consider the following FSM that has one input , 3 outputs and 6 states.

```
sis> write_kiss
.i 1
.o 3
.p 12
.s 6
.r s0
0 s0 s0 000
1 s0 s1 000
0 s1 s1 001
1 s1 s2 001
0 s2 s2 010
1 s2 s3 010
0 s3 s3 011
1 s3 s4 011
0 s4 s4 100
1 s4 s5 100
0 s5 s4 100
1 s5 s0 100
sis>
```

After using **state\_minimize** command, the FSM has only 5 states.

```
sis> state_minimize
Running stamina, written by June Rho, University of Colorado at Boulder
Number of states in original machine : 6
Number of states in minimized machine : 5
sis> write_kiss
.i 1
.o 3
.p 10
.s 5
.r S1
0 S0 S0 100
1 S0 S1 100
0 S1 S1 000
1 S1 S2 000
0 S2 S2 001
1 S2 S3 001
0 S3 S3 010
1 S3 S4 010
0 S4 S4 011
1 S4 S0 011
sis>
```

### one-hot

Assign one-hot codes to states of the FSM. If there are  $n$  states in an FSM, states will be given  $n$  bit codes, such that each of the states has exactly one 1 in their codes. For an FSM with four states, the following codes will be used:

```
0001
0010
0100
1000
```

Number of latches required will be equal to the number of states. Remember that the minimum number of latches required to implement an FSM with  $n$  states is  $\lceil \log_2 n \rceil$ . One-hot encoding sometimes leads to smaller combinational logic for FSMs, which have small number of states.

The example below shows this command used on the previous example, which was used for **state\_minimize** command.

```
sis> one_hot
sis> write_eqn
Warning: only combinational portion is being written.
INORDER = IN_0 LatchOut_v1 LatchOut_v2 LatchOut_v3 LatchOut_v4 LatchOut_v5;
OUTORDER = [76] [78] [80] [82] [84] OUT_0 OUT_1 OUT_2;
[65] = !IN_0*LatchOut_v2;
[66] = IN_0*LatchOut_v2;
```

```
[67] = !IN_0*LatchOut_v1;
[68] = IN_0*LatchOut_v1;
[69] = !IN_0*LatchOut_v4;
[70] = IN_0*LatchOut_v4;
[71] = !IN_0*LatchOut_v3;
[72] = IN_0*LatchOut_v3;
[73] = IN_0*LatchOut_v5;
[74] = !IN_0*LatchOut_v5;
[75] = ![67]*![73];
[76] = ![75];
[77] = ![65]*![68];
[78] = ![77];
[79] = ![66]*![71];
[80] = ![79];
[81] = ![69]*![72];
[82] = ![81];
[83] = ![70]*![74];
[84] = ![83];
[85] = ![67]*![68];
OUT_0 = ![85];
[87] = ![69]*![70]*![73]*![74];
OUT_1 = ![87];
[89] = ![71]*![72]*![73]*![74];
OUT_2 = ![89];

Don't care:
INORDER = IN_0 LatchOut_v1 LatchOut_v2 LatchOut_v3 LatchOut_v4 LatchOut_v5;
OUTORDER = LatchIn_v6.0 LatchIn_v6.1 LatchIn_v6.2 LatchIn_v6.3 LatchIn_v6.4
OUT_0 OUT_1 OUT_2;
LatchIn_v6.0 = !LatchOut_v1*!LatchOut_v2*!LatchOut_v3*!LatchOut_v4*!
LatchOut_v5 + LatchOut_v4*LatchOut_v5 + LatchOut_v3*LatchOut_v5 +
LatchOut_v2*
LatchOut_v5 + LatchOut_v1*LatchOut_v5 + LatchOut_v3*LatchOut_v4 +
LatchOut_v2*
LatchOut_v4 + LatchOut_v1*LatchOut_v4 + LatchOut_v2*LatchOut_v3 +
LatchOut_v1*
LatchOut_v3 + LatchOut_v1*LatchOut_v2;
LatchIn_v6.1 = !LatchOut_v1*!LatchOut_v2*!LatchOut_v3*!LatchOut_v4*!
LatchOut_v5 + LatchOut_v4*LatchOut_v5 + LatchOut_v3*LatchOut_v5 +
LatchOut_v2*
LatchOut_v5 + LatchOut_v1*LatchOut_v5 + LatchOut_v3*LatchOut_v4 +
LatchOut_v2*
LatchOut_v4 + LatchOut_v1*LatchOut_v4 + LatchOut_v2*LatchOut_v3 +
LatchOut_v1*
LatchOut_v3 + LatchOut_v1*LatchOut_v2;
LatchIn_v6.2 = !LatchOut_v1*!LatchOut_v2*!LatchOut_v3*!LatchOut_v4*!
LatchOut_v5 + LatchOut_v4*LatchOut_v5 + LatchOut_v3*LatchOut_v5 +
LatchOut_v2*
LatchOut_v5 + LatchOut_v1*LatchOut_v5 + LatchOut_v3*LatchOut_v4 +
LatchOut_v2*
LatchOut_v4 + LatchOut_v1*LatchOut_v4 + LatchOut_v2*LatchOut_v3 +
LatchOut_v1*
LatchOut_v3 + LatchOut_v1*LatchOut_v2;
LatchIn_v6.3 = !LatchOut_v1*!LatchOut_v2*!LatchOut_v3*!LatchOut_v4*!
LatchOut_v5 + LatchOut_v4*LatchOut_v5 + LatchOut_v3*LatchOut_v5 +
LatchOut_v2*
LatchOut_v5 + LatchOut_v1*LatchOut_v5 + LatchOut_v3*LatchOut_v4 +
LatchOut_v2*
```

```
LatchOut_v4 + LatchOut_v1*LatchOut_v4 + LatchOut_v2*LatchOut_v3 +  
LatchOut_v1*  
LatchOut_v3 + LatchOut_v1*LatchOut_v2;  
LatchIn_v6.4 = !LatchOut_v1*!LatchOut_v2*!LatchOut_v3*!LatchOut_v4*!  
LatchOut_v5 + LatchOut_v4*LatchOut_v5 + LatchOut_v3*LatchOut_v5 +  
LatchOut_v2*  
LatchOut_v5 + LatchOut_v1*LatchOut_v5 + LatchOut_v3*LatchOut_v4 +  
LatchOut_v2*  
LatchOut_v4 + LatchOut_v1*LatchOut_v4 + LatchOut_v2*LatchOut_v3 +  
LatchOut_v1*  
LatchOut_v3 + LatchOut_v1*LatchOut_v2;  
OUT_0 = !LatchOut_v1*!LatchOut_v2*!LatchOut_v3*!LatchOut_v4*!LatchOut_v5 +  
LatchOut_v4*LatchOut_v5 + LatchOut_v3*LatchOut_v5 + LatchOut_v2*LatchOut_v5  
+  
LatchOut_v1*LatchOut_v5 + LatchOut_v3*LatchOut_v4 + LatchOut_v2*LatchOut_v4  
+  
LatchOut_v1*LatchOut_v4 + LatchOut_v2*LatchOut_v3 + LatchOut_v1*LatchOut_v3  
+  
LatchOut_v1*LatchOut_v2;  
OUT_1 = !LatchOut_v1*!LatchOut_v2*!LatchOut_v3*!LatchOut_v4*!LatchOut_v5 +  
LatchOut_v4*LatchOut_v5 + LatchOut_v3*LatchOut_v5 + LatchOut_v2*LatchOut_v5  
+  
LatchOut_v1*LatchOut_v5 + LatchOut_v3*LatchOut_v4 + LatchOut_v2*LatchOut_v4  
+  
LatchOut_v1*LatchOut_v4 + LatchOut_v2*LatchOut_v3 + LatchOut_v1*LatchOut_v3  
+  
LatchOut_v1*LatchOut_v2;  
OUT_2 = !LatchOut_v1*!LatchOut_v2*!LatchOut_v3*!LatchOut_v4*!LatchOut_v5 +  
LatchOut_v4*LatchOut_v5 + LatchOut_v3*LatchOut_v5 + LatchOut_v2*LatchOut_v5  
+  
LatchOut_v1*LatchOut_v5 + LatchOut_v3*LatchOut_v4 + LatchOut_v2*LatchOut_v4  
+  
LatchOut_v1*LatchOut_v4 + LatchOut_v2*LatchOut_v3 + LatchOut_v1*LatchOut_v3  
+  
LatchOut_v1*LatchOut_v2;
```

## state\_assign

Assign codes to the FSM states so the combinational circuit for multi-level implementation gets minimized.

We use the same minimized FSM, which was used for illustrating state\_minimize command, and apply **state\_assign** command. Notice the names of inputs and outputs have been changed by the tool (NOVA). Also notice that this state assignment is using minimum number of bits(3) for encoding states. Therefore, it requires three latches.

```
sis> state_assign
Running nova, written by Tiziano Villa, UC Berkeley
Warning: network 'SISKAAa29998', node "v0" does not fanout
sis> write_eqn
Warning: only combinational portion is being written.
INORDER = IN_0 LatchOut_v1 LatchOut_v2 LatchOut_v3;
OUTORDER = v4.0 v4.1 v4.2 OUT_0 OUT_1 OUT_2;
OUT_0 = LatchOut_v2*LatchOut_v3;
OUT_1 = !LatchOut_v2*LatchOut_v3;
OUT_2 = !IN_0*LatchOut_v2*!LatchOut_v3 + IN_0*LatchOut_v2*!LatchOut_v3 +
LatchOut_v1*!LatchOut_v2;
v4.0 = !IN_0*LatchOut_v2*!LatchOut_v3 + !IN_0*LatchOut_v2*LatchOut_v3 +
LatchOut_v1*!LatchOut_v2 + IN_0*!LatchOut_v2;
v4.1 = IN_0*!LatchOut_v2*!LatchOut_v3 + !IN_0*LatchOut_v2*!LatchOut_v3 +
!IN_0*
LatchOut_v2*LatchOut_v3 + IN_0*LatchOut_v1*!LatchOut_v2;
v4.2 = IN_0*LatchOut_v2*!LatchOut_v3 + !IN_0*LatchOut_v2*LatchOut_v3 + !
LatchOut_v2*LatchOut_v3;

Don't care:
INORDER = IN_0 LatchOut_v1 LatchOut_v2 LatchOut_v3;
OUTORDER = LatchIn_[58] LatchIn_[59] LatchIn_[60] OUT_0 OUT_1 OUT_2;
LatchIn_[58] = LatchOut_v1*!LatchOut_v2*!LatchOut_v3 + !LatchOut_v1*
LatchOut_v2;
LatchIn_[59] = LatchOut_v1*!LatchOut_v2*!LatchOut_v3 + !LatchOut_v1*
LatchOut_v2;
LatchIn_[60] = LatchOut_v1*!LatchOut_v2*!LatchOut_v3 + !LatchOut_v1*
LatchOut_v2;
OUT_0 = LatchOut_v1*!LatchOut_v2*!LatchOut_v3 + !LatchOut_v1*LatchOut_v2;
OUT_1 = LatchOut_v1*!LatchOut_v2*!LatchOut_v3 + !LatchOut_v1*LatchOut_v2;
OUT_2 = LatchOut_v1*!LatchOut_v2*!LatchOut_v3 + !LatchOut_v1*LatchOut_v2;
```

---

## retime

Add more latches, or re-position the latches, to reduce the clock period required by the FSM. This command is generally used to speed up the circuit by adding more latches.

Try using retime command after state assignment. Since we have not really decided the actual components, we need to give option `-n` with retime command. Therefore, use the command **retime -n** for reducing the clock period.

```
sis> retime -n
Lower bound on the cycle time = 2.20
Retiming will minimize the cycle time
RETIME: Initial clk = 2.20 , Desired clk = 2.20
Circuit meets specification
```

## 6.3. Scripts

### 6.3.1. Using a script to simplify work

If you want to run a sequence of command in SIS you can use a script. You can write a simple text-file containing the commands you want to execute. For example if you want to run the commands *sweep* followed by *fx* and *sweep*, then you can write a text file with the following contents.

```
sweep
fx
sweep
```

You can then run this sequence of commands by entering the following.

```
sis> source filename
```

More information about script is written in the next section.

### 6.3.2. A Script of SIS commands for Multi-Level Logic Optimization

It is quite difficult to decide what sequence of SIS commands will lead to the required amount of optimization. Experienced designers at Univ. of California at Berkeley have worked out a sequence of SIS commands that seems to give good results for a large variety of circuits. This sequence of commands is available in the file “/home/beto/public/logic\_synthesis/rugged”

The “rugged” sequence of commands can be executed by the following SIS command.

```
sis> source rugged
```

#### Sequence “rugged”

```
sweep; eliminate -1
simplify -m nocomp
eliminate -1
sweep;
eliminate 5
simplify -m nocomp
resub -a
fx
resub -a; sweep
eliminate -1; sweep
full_simplify -m nocomp
```

Notice that it is possible to give many SIS commands in the same line by separating them using a semi-colon(;).

### 6.4. Output Commands

These commands can be used to write the results after processing. The results can be stored in a file. These commands mostly start with the word “write”, e.g. “write\_blif”.

#### write\_blif

Write the current network in BLIF format. You need to provide the file where the network is to be written. If no file is provided then it is written on the screen.

For example,

```
sis> write_blif temp.blif
```

will write the current network into the file “temp.blif” in the current directory.

**write\_blif** command will write the current network in BLIF format on the screen.

Similarly, other write-commands write the network in the required format.

```
write_astg
write_eqn
write_kiss
write_bdnet
write_slif
write_pla
```

## 6.5. Status Checking Commands

These commands can be run any time after first input command to see status of the optimisation. Many of these commands starts with a word 'print', e.g. 'print\_stats').

### **print\_stats**

This command prints the current status of the network. This command is useful to check the improvement achieved after using some optimization commands. The following example illustrates the use of this command.

```
sis> read_eqn book_example.eqn
sis> print_stats
book_example.eqn      pi= 5    po= 4    nodes= 4      latches= 0
lits(sop)= 26
sis> decomp
sis> print_stats
book_example.eqn      pi= 5    po= 4    nodes= 8      latches= 0
lits(sop)= 23
sis>
```

**pi** and **po** give the number of primary inputs and primary outputs; **nodes** gives sum of the number of internal nodes and output nodes in the network; **latches** give the number of latches used to implement the system; **lits(sop)** gives the number of literals in the representation of the network. Note that in the above example, number of nodes goes up but number literals are reduced by the use of "decomp" command.



## print\_level

This command gives the level of all nodes in the network. The following example illustrates the use of this command. The nodes at the line starting with “0:” show the level 0 nodes. Those nodes are the inputs. The following lines shows the signals after the first, second etc., level of logic.

```
sis> read_eqn book_example.eqn
sis> write_eqn
INORDER = a b c d e;
OUTORDER = w x y z;
w = a!*e + !c*d + b*d + !a*d;
x = d*e + c*e + !b + !a;
y = b*d + a*d + b*c + a*c + e;
z = c + b + a;
sis> print_level
  0: e d c b a
  1: {y} {x} {z} {w}
sis> decomp
sis> write_eqn
INORDER = a b c d e;
OUTORDER = w x y z;
w = d*[4] + a!*e;
x = e*[5] + !b + !a;
y = [7]*[6] + e;
z = c + b + a;
[4] = !c + b + !a;
[5] = d + c;
[6] = d + c;
[7] = b + a;
sis> print_level
  0: b a e c d
  1: {z} [4] [5] [6] [7]
  2: {w} {x} {y}
sis>
```

Observe that initially the network had all the input nodes at level 0 and output nodes were at level 1. After optimization using **decomp** command, the network now has two levels and some internal nodes are introduced. Number of levels in a network decides its delay.

## Other print commands

Similarly, other print commands print useful information about the network.

```
print
print_io
print_kernel
print_factor
print_delay
print_gate
```

## 6.6. Miscellaneous Commands

Some commands do different jobs than the above like technology mapping. Most of the technology mapping commands ends up with a word ‘map’, e.g. ‘cutmap’. We will learn about these commands in our next exercise. Here we learn only one command called **simulate**.

### **simulate**

This command is used for simulating a network. We illustrate by simulating the following network.

```
sis> write_eqn
INORDER = a b;
OUTORDER = v;
v = a*!b + !a*b;
```

Suppose we want to simulate this circuit (XOR gate) for input patterns 0 1 and 1 1, then the following commands will do the job. Simulate command can be used to simulate FSMs also. Therefore, for a combinational network it will not specify any next state.

```
sis> simulate 0 1

Network simulation:
Outputs: 1
Next state:
sis> simulate 1 1

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