SIS Logic Optimisation and Synthesis

Maurizio Palesi

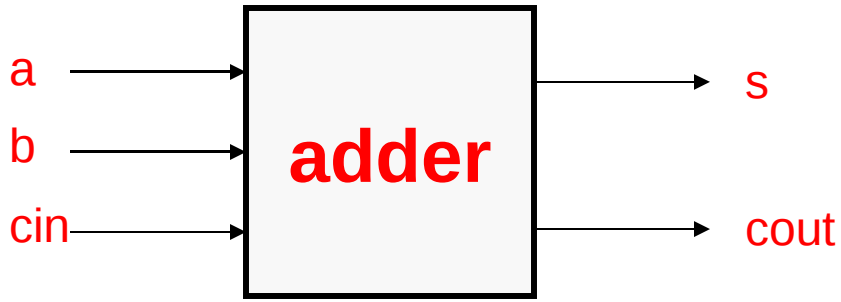
About SIS

- SIS: Sequential Interactive Synthesis
  - ➔Is a software package for logic design developed at the University of California, Berkeley
- SIS can synthesise combinational, synchronous and asynchronous circuits, generating either two-level or multi-level (factorised) equations
- These equations can then be mapped onto a user-defined component library representing gates, flip-flops, standard cells, etc, and these circuits optimised for minimum size, maximum speed, etc.

Example – Full Adder

The BLIF Format

- The file **FA.BLIF**, contains the truth-table for a full adder




```
.model adder
.inputs a b cin
.outputs o cout

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

.names a b cin
cout
011 1
101 1
110 1
111 1

.end
```

Example – Full Adder

The print command

- We invoke SIS and read in the network

```
$ sis
UC Berkeley, SIS 1.3 (compiled May 28 1995)
sis>
sis> read_blif fa.blif
```

- To see the equations in sum-of-products form, type the "p" (print) command:

```
sis> p
{sum} = a b cin + a b' cin' + a' b cin' + a' b' cin
{co} = a b cin + a b cin' + a b' cin + a' b cin
```

Example – Full Adder

The simplify command

- To carry out a two-level minimisation on all the nodes of current network, we will use a version of the "simplify" command - simply type "sim1 *" to simplify all of the output functions

```
sis> sim1 *
sis> p
{sum} = a b cin + a b' cin' + a' b cin' + a' b' cin
{co} = a b + a cin + b cin
```

Example – Full Adder

The print_stats Command

- The "ps" (print_stats) command will show the number of literals required to represent the functions in both sum-of-products and factored forms

```
sis> ps
fa.pla pi= 3 po= 2 nodes= 2 latches= 0
lits(sop)= 18 lits(fac)= 15
```

- The factorised form is:

```
sis> pf
{sum} = cin (a' b' + a b) + cin' (a b' + a' b)
{co} = cin (b + a) + a b
```

Example – Full Adder

The rlib & map Commands

- We will now carry out technology mapping, that is, the mapping of the current network onto a predefined library of gates.
- In this case, we will use the library **ANDOR4.GEN** (found in **SIS_LIB**)
- To use a library, we must first issue a "rlib" command followed by the "map" command.

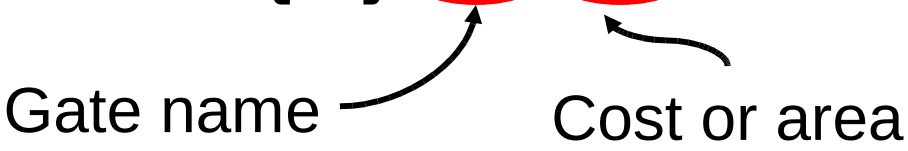
```
sis> rlib andor4.gen
sis> map
sis> pf
[122] = b'
[123] = cin'
[97] = [122] [123]
[98] = b cin
[124] = [98] + [97]
[151] = [124] + a
[152] = [151]'
[96] = [124] a
{sum} = [96] + [152]
[145] = cin + b
[93] = [145] a
[94] = b cin
{co} = [94] + [93]
```

Example – Full Adder

The print_gate Command

- A number of new nodes have been introduced during the mapping onto the library components.
- The "pg" (print_gate) command will print out the gates used in deriving each of the non-leaf nodes of the network (the leaf nodes describe input literals)
- We see that SIS has implemented the circuit using 2-input AND and OR gates and inverters

```
sis> pg
[122] inv 2.00
[123] inv 2.00
[97] and2 3.00
[98] and2 3.00
[124] or2 3.00
[151] or2 3.00
[152] inv 2.00
[96] and2 3.00
{sum} or2 3.00
[145] or2 3.00
[93] and2 3.00
[94] and2 3.00
{co} or2 3.00
```



Example – Full Adder

The print_gate Command

- We can now investigate the effect of adding XOR gates to the library.
- The **AOXOR.GEN** library contains a 2-input XOR gate
- Let we add to the current library using the "rlib -a" command
- We now see that the full-adder has been implemented using the XOR gates, even though they are more expensive, since this has reduced the overall cost and delay of the circuit

```
sis> rlib -a aoxor.gen
sis> map
sis> pf
[124] = b cin' + b' cin
{sum} = [124] a' +
[124]' a
[318] = cin + b
[93] = [318] a
[94] = b cin
{co} = [94] + [93]
sis> pg
[124] xor 5.00
{sum} xor 5.00
[318] or2 3.00
[93] and2 3.00
[94] and2 3.00
{co} or2 3.00
```

Example – Full Adder

The print_gate Comand

- A summary of the gates used is given by the "pgc" command

```
sis> pgc
and2 : 2 (area=3.00)
or2 : 2 (area=3.00)
xor : 2 (area=5.00)
Total: 6 gates, 22.00 area
```

- The circuit propagation delay is given by the “pat” command

```
sis> pat sum co
... using library delay model
{sum} : arrival=( 6.10 6.10)
{co} : arrival=( 4.40 4.40)
```

Algebraic Manipulations

- Normally, logic equations are manipulated in order to reduce the number of literals that they contain, since this is taken as an indicator of the number and size of logic gates that will be required to implement the circuit
- An equation may be ‘factorised’ in order to extract literals common to a number of terms
- The following example shows an input file **F2.EQN** written in the form of an equation

```
z = a b + a c + a d + a e + b c + b d + b e;
```

Algebraic Manipulations

```
sis> re f2.eqn
sis> p
{z} = a b + a c + a d + a e + b c + b d + b e
sis> pf
{z} = a (e + d + c) + b (e + d + c + a)
sis> ps
f2.eqn pi= 5 po= 1 nodes= 1 latches= 0
lits(sop)= 14 lits(fac)= 9
```

Algebraic Manipulations

The factor Comand

- Better quality results are typically obtained using the "factor -good" or "gf" command:

```
sis> gf z
sis> pf
{z} = (b + a) (e + d + c) + a b
sis> ps
f2.eqn pi= 5 po= 1 nodes= 1
latches= 0
lits(sop)= 14 lits(fac)= 7
```

Algebraic Manipulations

The decomp Comand

- Decomposition may be used to break down complex functions into simpler components

```
sis> re f2.eqn
sis> decomp -g
sis> p
{z} = [8] b + [9] a
[9] = [8] + b
[8] = c + d + e
```

Algebraic Manipulations

The eliminate Comand

- The ‘eliminate’ command carries out the reverse of decomposition by replacing a node with the equation it represents, according to the number of times that node is used in the network
- For example, the command "eliminate -1" will remove any nodes which are only used once :

```
sis> el -1
sis> pf
{z} = [8] (b + a) + a b
[8] = e + d + c
```

- In this case, node [9] has been eliminated.
- The eliminate command will tend to reduce the number of levels in a network

Algebraic Manipulations

The factor Comand

- For a multiple-output circuit, the common factors, or ‘divisors’, must be chosen in order to minimise the overall number of literals in the combined equations

```
sis> re f6.eqn
sis> p
{y} = a f + b f + c f + d f + g
{z} = a f + b f + c f + e f + g
sis> gf *
sis> pf
{y} = f (d + c + b + a) + g
{z} = f (e + c + b + a) + g
sis> ps
f6.eqn pi= 7 po= 2 nodes= 2 latches= 0
lits(sop)= 18 lits(fac)= 12
```

- This shows the result of factoring each equation seperately

Algebraic Manipulations

The gkx Comand

- The "gkx" command may be used to extract the largest factor common to both equations

```
sis> gkx
sis> pf
{y} = f ([2] + d) + g
{z} = f ([2] + e) + g
[2] = c + b + a
sis> ps
f6.eqn pi= 7 po= 2 nodes= 3 latches= 0
lits(sop)= 13 lits(fac)= 11
```

- We see that in this instance the use of the common factor gives a result which is less than optimal in each individual case, but gives a better overall result

Technology Mapping

- The process of ‘technology mapping’ describes the conversion of a network described only by Boolean equations into a circuit made up of logic components chosen from a particular device library
- In general, we begin technology mapping with a network containing a minimum number of literals - on the basis that this will generate the simplest logic circuit
  - However, if the component library does not contain a component corresponding to a particular expression in the network, then the given expression will be automatically decomposed
  - For example, if a library contains only simple NAND gates, then the network will be decomposed until it consists only of NAND functions, even though this may increase the number of literals (and logic levels).
  - It is normally possible to implement a design using different combinations of components, and SIS can be directed to choose components on the basis of their speed or their cost, allowing the designer to make a trade-off between these factors

Technology Mapping

- The following example shows the equations for a four-bit ripple-carry adder **`rip4.eqn`** being mapped onto a library of standard cells

```
# 4-bit ripple-carry full adder
s0 = a0 ^ b0 ^ ci0 ;
co0 = ci0 * (a0 ^ b0) + a0 * b0 ;
ci1 = co0 ;
s1 = a1 ^ b1 ^ ci1 ;
co1 = ci1 * (a1 ^ b1) + a1 * b1 ;
ci2 = co1 ;
s2 = a2 ^ b2 ^ ci2 ;
co2 = ci2 * (a2 ^ b2) + a2 * b2 ;
ci3 = co2 ;
s3 = a3 ^ b3 ^ ci3 ;
co3 = ci3 * (a3 ^ b3) + a3 * b3 ;
```

Technology Mapping

- The input file is read, and the equations printed in SOP form

```
sis> re rip4.eqn
sis> p
{s0} = a0 b0 ci0 + a0 b0' ci0' + a0' b0 ci0' + a0' b0' ci0
co0 = a0 b0 + a0 b0' ci0 + a0' b0 ci0
ci1 = co0
{s1} = a1 b1 ci1 + a1 b1' ci1' + a1' b1 ci1' + a1' b1' ci1
co1 = a1 b1 + a1 b1' ci1 + a1' b1 ci1
ci2 = co1
{s2} = a2 b2 ci2 + a2 b2' ci2' + a2' b2 ci2' + a2' b2' ci2
co2 = a2 b2 + a2 b2' ci2 + a2' b2 ci2
ci3 = co2
{s3} = a3 b3 ci3 + a3 b3' ci3' + a3' b3 ci3' + a3' b3' ci3
{co3} = a3 b3 + a3 b3' ci3 + a3' b3 ci3
```

- The "ps" command shows the literal count for these equations as 83 (sop) or 71 (factored)

Technology Mapping

The map Comand

- We now open the library **`lib2.gen`** which contains a set of simple and complex combinational devices representative of a VLSI standard cell library :
  - ➔ `sis> rlib lib2.gen`
- The "pgc" command will be used to display the gates used, and the "pat -p 1" command will show the maximum propagation delay present in the resulting circuit. Since these commands will be used each time the circuit is mapped, the "alias" command is used to define a name, ‘pr’, to represent the required command string
  - ➔ `sis> alias pr "pgc; pat -p 1"`

specifies a minimum-area mapping

```
sis> re rip4.eqn
sis> map -m 0 -AFW
sis> pr
aoi21 : 3 (area=1856.00)
aoi22 : 1 (area=2320.00)
inv1x : 6 (area=928.00)
inv2x : 12 (area=928.00)
inv4x : 2 (area=1392.00)
nand2 : 6 (area=1392.00)
nand3 : 3 (area=1856.00)
oai21 : 9 (area=1856.00)
oai22 : 3 (area=2320.00)
Total: 45 gates, 64960.00 area
... using library delay model
{s3} : arrival=(12.27 12.33)
```

Technology Mapping

- The example is now repeated using different delay-area trade-offs :

Command	*Area*	*Delay*
map -m 0.5 -AFW	45 gates, 64960.00 area	12.33
map -n 1 -AFGW	43 gates, 63104.00 area	11.89
map -m 0	27 gates, 47328.00 area	16.45
map -m 0.5	35 gates, 54752.00 area	14.00
map -m 1	39 gates, 66352.00 area	14.97

- These results show that a range of results may be obtained, with a variation of around 1.7:1 in cost and 1.4:1 in worst-case delay

Technology Mapping

- The results above are obtained by mapping the source equations as given, with no optimisations being carried out. The mappings are now repeated, but with the equations processed by **`SCRIPT.BOO`** giving a literal count of 60 (sop) or 48 (factored).

Command	*Area*	*Delay*
so script.boo map -m 0 -AFW	38 gates, 54288.00 area	10.73
so script.boo map -m 0.5 -AFW	38 gates, 54288.00 area	10.73
so script.boo map -n 1 -AFGW	38 gates, 54288.00 area	11.20
so script.boo map -m 0	25 gates, 41760.00 area	13.60
so script.boo map -m 0.5	29 gates, 45472.00 area	13.20
so script.boo map -m 1	37 gates, 59856.00 area	12.80

- The simplification of the original equations has been reflected in the circuits obtained

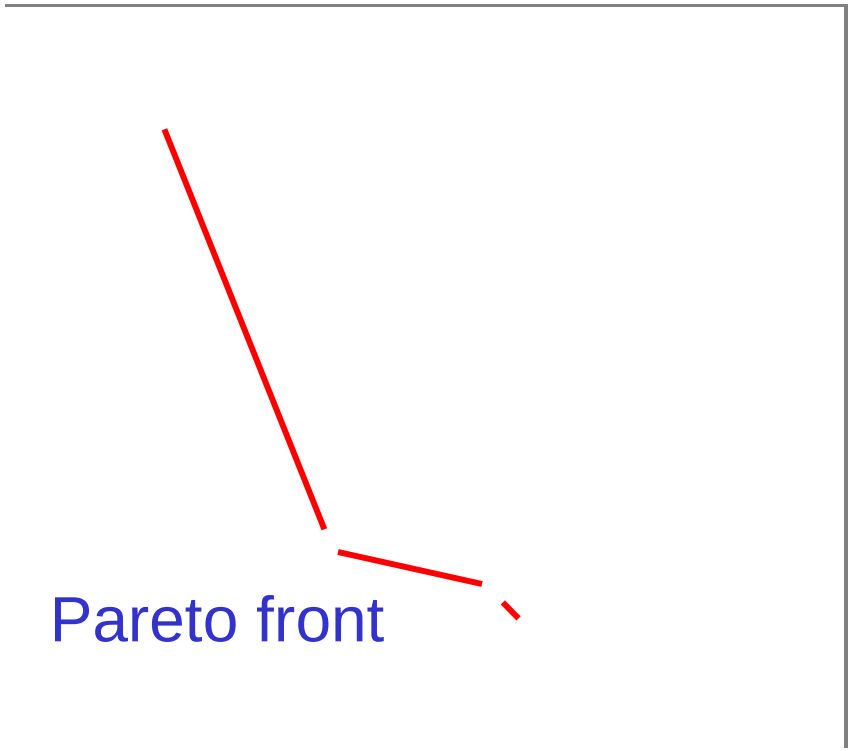
Technology Mapping

- Finally, the simplified equations are collapsed before mapping, in an attempt to reduce the propagation delay irrespective of the effect on area. The literal count increases to 684 (sop) or 134 (factored).

Command	*Area*	*Delay*
so script.boo ; collapse map -m 0 -AFW	80 gates, 126208.00 area	9.06
so script.boo ; collapse map -m 0.5 -AFW	80 gates, 120640.00 area	8.44
so script.boo ; collapse map -n 1 -AFGW	76 gates, 120640.00 area	11.20
so script.boo ; collapse map -m 0	61 gates, 106256.00 area	10.90
so script.boo ; collapse map -m 0.5	67 gates, 110896.00 area	10.50
so script.boo ; collapse map -m 1	71 gates, 125280.00 area	9.80

Technology Mapping

Pareto-front

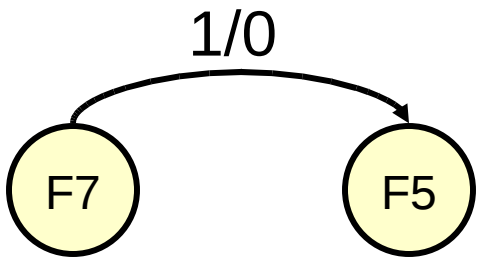


Sequential Synthesis

- This BLIF format file **`seq735.blf`** shows the state table which will be processed

```
.model
.inputs x
.outputs z
.start_kiss
.i 1
.o 1
0 f1 f8 0
1 f1 f3 0
0 f2 f8 0
1 f2 f6 0
0 f3 f8 1
1 f3 f1 0
0 f4 f1 1
1 f4 f9 0
0 f5 f4 0
1 f5 f7 0
0 f6 f4 1
1 f6 f2 0
0 f7 f4 1
1 f7 f5 0
0 f8 f5 1
1 f8 f9 0
- f9 f10 –
0 f10 f1 0
1 f10 f5 0
.end_kiss

.end
```



Sequential Synthesis

The state_minimize Comand

- The file is read in using the "read_blif" ("rl") command, and the "ps" command shows that the design contains 10 states, but that no circuit equations yet exist

```
sis> rl seq735.blf
sis> ps
seq735.blf pi= 1 po= 1 nodes= 1 latches= 0
lits(sop)= 0 lits(fac)= 0 #states(STG)= 10
```

- The "sm" command is now used to carry out state minimisation:

```
sis> sm
Running stamina, written by June Rho, University of
Colorado at Boulder
system(stamina < d:\sis\SISBAAa00057 >
d:\sis\SISCAAa00057)
Number of states in original machine : 10
Number of states in minimized machine : 5
```

Sequential Synthesis

The state_assign Command

- The minimised state table may now be displayed

```
sis> write_kiss
.i 1
.o 1
.p 9
.s 5
.r S1
0 S0 S2 1
1 S0 S1 0
0 S2 S1 1
1 S2 S3 0
0 S1 S2 0
1 S1 S0 0
- S3 S4 –
0 S4 S1 0
1 S4 S1 0
```

- To carry out state assignment, the "jedi" program is used
  - `sis> sa jedi -e c`
- Note that the " `-e c` " option was used to attempt to generate an optimal state assignment.
- Several other options may be used, for example, to generate natural binary (`-e s`) or one-hot (`-e h`) assignments.
- See the Reference Manual for details.

Sequential Synthesis

```
sis> write_blif
.model seq735.blf
.inputs x
.outputs z
.latch [3] LatchOut_v1 1
.latch [4] LatchOut_v2 0
.latch [5] LatchOut_v3 1
.start_kiss
.i 1
.o 1
.p 9
.s 5
.r S1
0 S0 S2 1
1 S0 S1 0
0 S2 S1 1
1 S2 S3 0
0 S1 S2 0
1 S1 S0 0
- S3 S4 –
0 S4 S1 0
1 S4 S1 0
.end_kiss
```

```
.latch_order LatchOut_v1 LatchOut_v2
LatchOut_v3
.code S0 011
.code S2 111
.code S1 101
.code S3 001
.code S4 110
...remainder of file not shown.
```

Sequential Synthesis

- The network equations may now be displayed

```
sis> p
[3] = LatchOut_v1' + LatchOut_v3' + x'
[4] = LatchOut_v1' x' + LatchOut_v2'
[5] = LatchOut_v1 + LatchOut_v2
{z} = LatchOut_v2 LatchOut_v3 x'
sis> ps
seq735.blf pi= 1 po= 1 nodes= 4 latches= 3
lits(sop)= 11 lits(fac)= 11 #states(STG)= 5
```

Sequential Synthesis

The extract_seq_dc Comand

- In cases where unused states exist, the "extract_seq_dc" command will identify them so that they can be used as dont-care states which may help to further simplify the circuit equations

```
sis> extract_seq_dc
number of latches = 3 depth = 4 states
visited = 5
sis> full_simplify
sis> ps
seq735.blf pi= 1 po= 1 nodes= 4 latches= 3
lits(sop)= 11 lits(fac)= 11 #states(STG)= 5
```

- In this case, no improvement has been achieved

Sequential Synthesis

- Technology mapping may now be carried out

```
sis> rlib lib2.gen
sis> rlib -a lib2_lat.gen
sis> map -W
sis> pgc
dff : 3 (area=4640.00)
inv2x : 3 (area=928.00)
nand2 : 1 (area=1392.00)
nand3 : 1 (area=1856.00)
nor3 : 1 (area=1856.00)
oai21 : 1 (area=1856.00)
Total: 10 gates, 23664.00 area
```

- We can see that the resulting circuit contains three d-type flip-flops as expected

Explicit State Assignment

- The following example, **`count5.blf`** , shows a specification for a simple 5-state binary up-counter.
- Although this design requires no primary inputs, the KISS format does require that at least one input be present in the state table
- In this example, an input called **a** is included, but its value is ‘dont-care’ throughout the table and consequently it does not appear in the circuit equations
- The binary values of the states **`zero`** to **`four`** are defined explicitly using the **`.code`** statements

Explicit State Assignment

The code Statement

```
.model
.inputs a
.outputs qc qb qa
.start_kiss
.i 1
.o 3
.s 5
- zero one 000
- one two 001
- two three 010
- three four 011
- four zero 100
.end_kiss
.code zero 000 #state assignment
.code one 001
.code two 010
.code three 011
.code four 100
.end
```

Explicit State Assignment

- The assigned state table is now read in and the "stg_to_network" command used to generate the circuit equations

```
sis> rl count5.blf
sis> stg_to_network
sis> print_latch -s
input: {[3]} output: LatchOut_v1 init val: 0 cur val: 0
input: {[4]} output: LatchOut_v2 init val: 0 cur val: 0
input: {[5]} output: LatchOut_v3 init val: 0 cur val: 0
```

- The "print_latch" command shows the signal names and current states of the d-type latches that have been generated

```
sis> ps
count5.blf pi= 1 po= 3 nodes= 6 latches= 3
lits(sop)= 17 lits(fac)= 15 #states(STG)= 5
sis> p
[3] = LatchOut_v2 LatchOut_v3
[4] = LatchOut_v2 LatchOut_v3' + LatchOut_v2' LatchOut_v3
[5] = LatchOut_v1' LatchOut_v3'
{qc} = LatchOut_v1
{qb} = LatchOut_v2 LatchOut_v3 + LatchOut_v2 LatchOut_v3'
{qa} = LatchOut_v2 LatchOut_v3 + LatchOut_v2' LatchOut_v3
```

Design Verification

- The "simulate" command is used to display the values of the primary output signals as a function of the primary inputs (and, for sequential designs, the current state).

```
sis> rl count5.blf
sis> stg_to_network
sis> print_state
Network state: 000
STG state: zero (000)
sis> sim 0
Network simulation: Outputs: 0 0 0 Next state: 001
STG simulation: Outputs: 0 0 0 Next state: one (001)
sis> sim 0
Network simulation: Outputs: 0 0 1 Next state: 010
STG simulation: Outputs: 0 0 1 Next state: two (010)
sis> sim 0
Network simulation: Outputs: 0 1 0 Next state: 011
STG simulation: Outputs: 0 1 0 Next state: three (011)
```