VLSI CAD: LOGIC TO LAYOUT:
ESPRESSO Software Package

Rob A. Rutenbar
University of Illinois at Champaign-Urbana

ESPRESSO[1] is a 2-level logic optimization tool developed by researchers at the University of California at Berkeley. Early versions of the idea were developed at the T.J. Watson Research Labs; the final version of the tool was developed at Berkeley. Rick Rudell's Master's Thesis[2] from 1986 is a very complete explanation of the development and software implementation of the tool. ESPRESSO is a hugely successful optimizer: it is the tool that pioneered the reduce-expand-irredundant heuristic, and the ideas are built into most modern logic synthesis systems.

ESPRESSO has many options: you can instruct ESPRESSO "how hard" to work to optimize your logic. However, we will run ESPRESSO in its standard, default mode, which will suffice for our teaching purposes.

As we described in lecture, you can start a 2-level optimization problem with a *partially specified truth table* (TT) with input don't cares. Let's look at a small example of an ESPRESSO input file, shown below. (**Note**: we add the *//comments* for clarity; but you *cannot* put these comments in the actual ESPRESSO input file! *Don't add these comments! ESPRESSO will not like this!*)

```
.i 4              // There are 4 inputs
.o 1              // There is 1 output
                  // ESPRESSO can optimize several functions at the same time
.ilb w z y z      // These are the names of the inputs
.ob f             // This is the name of the output
0-11   1          // One  line of the TT.  Order does not matter.   "-" = input don't care
01-1   1          // More TT
1011   1          // More TT
1111   -          // More TT.  We explicitly tell ESPRESSO that f is don't care here.
```

You do *not* need to explicitly specify every row of the TT. Any row that does not match something in your input file is assumed to have the function(s) output = 0. If you run ESPRESSO on this input, this is what will be produced as output:

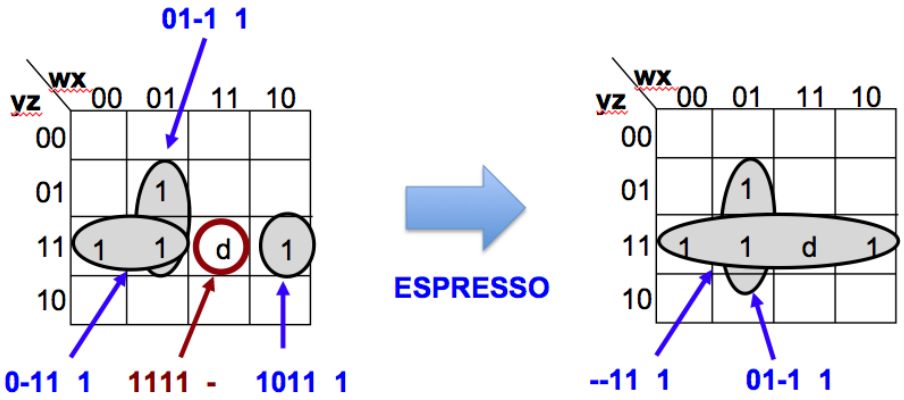
---

[1] Copyright (c) 1988, 1989, Regents of the University of California
[2] Richard L. Rudell, "Multiple Valued Logic Minimization for PLA Synthesis", EECS Department, UC Berkeley, Technical Report No. UCB/ERL M86/65, 1986. http://www.eecs.berkeley.edu/Pubs/TechRpts/1986/734.html

```
.i 4              // There are still 4 inputs
.o 1              // There is still 1 output
.ilb w z y z      // These are still the names of the inputs
.ob f             // This is the still the name of the output
.p 2              // The result optimal SOP form has 2 product terms (AND gates)
--11    1         // One product term is:  yz
01-1    1         // One product term is:  w'xz
.e                //This is the end of the ESPRESSO output file
```

ESPRESSO's output format is almost the same as the input format. You read the lines specifying the product terms like PCN cubes: a Don't Care "**–**" means the variable is omitted from the product; a **0** means you complement that variable; a **1** means you use the positive form of that variable. So, the result of this optimization is that the function is optimized to be: **f = yz + w'xz.** There are **2** products (AND gates), **5** literals (input wires to AND gates), and **1** OR gate with **2** inputs, in this final design.

Since this is a very small problem, we can check ESPRESSO's work with a simple Kmap:



On the left, we see the lines of the input truth table each associated with a (non-prime) cover in the Kmap: some lines cover several **1's** in the TT, because they have input don't cares. Some lines cover a single cell of the Kmap. The explicit output Don't Care appears on the Kmap as a "**d**" in cell **1111**. On the right, we see that ESPRESSO has covered the Kmap in the way would expect to do it by hand: 2 easy product terms, and we would cover the Don't Care "**d**" because it helps us make a better, smaller logic circuit.

This example is quite small. The real power of ESPRESSO is we can do very *large* functions, with *many* input variables, *many* simultaneous functions, and *thousands* of literals and products, and still get excellent, highly optimized (i.e., small) logic.

Content-type: text/html

ESPRESSO

Section: Misc. Reference Manual Pages (5OCTTOOLS)
Updated: 22 August 1986
Index Return to Main Contents

NAME

espresso - input file format for espresso(1OCTTOOLS)

DESCRIPTION

Espresso accepts as input a two-level description of a Boolean function. This is described as a character matrix with keywords embedded in the input to specify the size of the matrix and the logical format of the input function. Programs exist to translate a set of equations into this format (e.g., eqntott(1OCTTOOLS), bdsyn(1OCTTOOLS), eqntopla(1OCTTOOLS)). This manual page refers to Version 2.3 of *Espresso*.

Comments are allowed within the input by placing a pound sign (#) as the first character on a line. Comments and unrecognized keywords are passed directly from the input file to standard output. Any white-space (blanks, tabs, etc.), except when used as a delimiter in an embedded command, is ignored. It is generally assumed that the PLA is specified such that each row of the PLA fits on a single line in the input file.

KEYWORDS

The following keywords are recognized by *espresso*. The list shows the probable order of the keywords in a PLA description. [d] denotes a decimal number and [s] denotes a text string. The minimum required set of keywords is **.i** and **.o** for binary-valued functions, or **.mv** for multiple-valued functions.

.i [d]
: Specifies the number of input variables.

.o [d]
: Specifies the number of output functions.

.mv [num_var] [num_binary_var] [d1] . . . [dn]
: Specifies the number of variables (num_var), the number of binary variables (num_binary_var), and the size of each of the multiple-valued variables (d1 through dn).

.ilb [s1] [s2] . . . [sn]
: Gives the names of the binary valued variables. This must come after **.i** and **.o** (or after **.mv**). There must be as many tokens following the keyword as there are input variables.

.ob [s1] [s2] . . . [sn]
: Gives the names of the output functions. This must come after **.i** and **.o** (or after **.mv**). There must be as many tokens following the keyword as there are output variables.

.label var=[d] [s1] [s2] ...
: Specifies the names of the parts of a multiple-valued variable. This must come after **.mv**. There must

be as many tokens following the keyword as there are parts for this variable. Note that the variables are numbered starting from 0.

.type [s]
Sets the logical interpretation of the character matrix as described below under "Logical Description of a PLA". This keyword must come before any product terms. [s] is one of f, r, fd, fr, dr, or fdr.

.phase [s]
[s] is a string of as many 0's or 1's as there are output functions. It specifies which polarity of each output function should be used for the minimization (a 1 specifies that the ON-set of the corresponding output function should be used, and a 0 specifies that the OFF-set of the corresponding output function should be minimized).

.pair [d]
Specifies the number of pairs of variables which will be paired together using two-bit decoders. The rest of the line contains pairs of numbers which specify the binary variables of the PLA which will be paired together. The binary variables are numbered starting with 0. The PLA will be reshaped so that any unpaired binary variables occupy the leftmost part of the array, then the paired multiple-valued columns, and finally any multiple-valued variables. If the labels have been specified using **.ilb**, then the variable names may be used instead of the column number.

.symbolic [s0] [s1] . . . [sn] ; [t0] [t1] . . . [tm] ;
Specifies that the binary-valued variables named [s0] thru [sn] are to be considered as a single multiple-valued variable. Variable [s0] is considered the most significant bit, [s1] the next most significant, and [sn] is the least significant bit. This creates a variable with 2**n parts corresponding to the decodes of the binary-valued variables. The keywords [t0] thru [tm] provide the labels for each decode of [s0] thru [sn]. ([t0] corresponds to a value of 00...00, [t1] is the value 00...01, etc.). The binary-variables may be identified by column number, or by variable name when **.ilb** is used. The binary-variables are removed from the function after the multiple-valued variable is created.

.symbolic-output [s0] [s1] . . . [sn] ; [t0] [t1] . . . [tm] ;
Specifies that the output functions [s0] ... [sn] are to be considered as a single symbolic output. This creates 2**n more output variables corresponding to the possible values of the outputs. The outputs may be identified by number (starting from 0), or by variable name when **.ob** is used. The outputs are removed from the function after the new set of outputs is created.

.kiss
Sets up for a *kiss*-style minimization.

.p [d]
Specifies the number of product terms. The product terms (one per line) follow immediately after this keyword. Actually, this line is ignored, and the ".e", ".end", or the end of the file indicate the end of the input description.

.e (.end)
Optionally marks the end of the PLA description.

LOGICAL DESCRIPTION OF A PLA

When we speak of the ON-set of a Boolean function, we mean those minterms which imply the function value is a 1. Likewise, the OFF-set are those terms which imply the function is a 0, and the DC-set (don't care set) are those terms for which the function is unspecified. A function is completely described by providing its ON-set, OFF-set and DC-set. Note that all minterms lie in the union of the ON-set, OFF-set and DC-set, and that the ON-set, OFF-set and DC-set share no minterms.

The purpose of the *espresso* minimization program is to find a logically equivalent set of product-terms to

represent the ON-set and optionally minterms which lie in the DC-set, without containing any minterms of the OFF-set.

A Boolean function can be described in one of the following ways:

1)
By providing the ON-set. In this case, *espresso* computes the OFF-set as the complement of the ON-set and the DC-set is empty. This is indicated with the keyword **.type f** in the input file.

2)
By providing the ON-set and DC-set. In this case, *espresso* computes the OFF-set as the complement of the union of the ON-set and the DC-set. If any minterm belongs to both the ON-set and DC-set, then it is considered a don't care and may be removed from the ON-set during the minimization process. This is indicated with the keyword **.type fd** in the input file.

3)
By providing the ON-set and OFF-set. In this case, *espresso* computes the DC-set as the complement of the union of the ON-set and the OFF-set. It is an error for any minterm to belong to both the ON-set and OFF-set. This error may not be detected during the minimization, but it can be checked with the subprogram "-Dcheck" which will check the consistency of a function. This is indicated with the keyword **.type fr** in the input file.

4)
By providing the ON-set, OFF-set and DC-set. This is indicated with the keyword **.type fdr** in the input file.

If at all possible, *espresso* should be given the DC-set (either implicitly or explicitly) in order to improve the results of the minimization.

A term is represented by a "cube" which can be considered either a compact representation of an algebraic product term which implies the function value is a 1, or as a representation of a row in a PLA which implements the term. A cube has an input part which corresponds to the input plane of a PLA, and an output part which corresponds to the output plane of a PLA (for the multiple-valued case, see below).

SYMBOLS IN THE PLA MATRIX AND THEIR INTERPRETATION

Each position in the input plane corresponds to an input variable where a 0 implies the corresponding input literal appears complemented in the product term, a 1 implies the input literal appears uncomplemented in the product term, and - implies the input literal does not appear in the product term.

With type *f*, for each output, a **1** means this product term belongs to the ON-set, and a **0** or **-** means this product term has no meaning for the value of this function. This type corresponds to an actual PLA where only the ON-set is actually implemented.

With type *fd* (the default), for each output, a **1** means this product term belongs to the ON-set, a **0** means this product term has no meaning for the value of this function, and a **-** implies this product term belongs to the DC-set.

With type *fr*, for each output, a **1** means this product term belongs to the ON-set, a **0** means this product term belongs to the OFF-set, and a **-** means this product term has no meaning for the value of this function.

With type *fdr*, for each output, a **1** means this product term belongs to the ON-set, a **0** means this product term belongs to the OFF-set, a **-** means this product term belongs to the DC-set, and a **~** implies this product term has no meaning for the value of this function.

Note that regardless of the type of PLA, a **~** implies the product term has no meaning for the value of this function. **2** is allowed as a synonym for **-**, **4** is allowed for **1**, and **3** is allowed for **~**.

MULTIPLE-VALUED FUNCTIONS

Espresso will also minimize multiple-valued Boolean functions. There can be an arbitrary number of multiple-valued variables, and each can be of a different size. If there are also binary-valued variables, they should be given as the first variables on the line (for ease of description). Of course, it is always possible to place them anywhere on the line as a two-valued multiple-valued variable. The function size is described by the embedded option **.mv** rather than **.i** and **.o**.

A multiple-output binary function with *ni* inputs and *no* outputs would be specified as **.mv** *ni+1 ni no*. **.mv** cannot be used with either **.i** or **.o** - use one or the other to specify the function size.

The binary variables are given as described above. Each of the multiple-valued variables are given as a bit-vector of **0** and **1** which have their usual meaning for multiple-valued functions. The last multiple-valued variable (also called the output) is interpreted as described above for the output (to split the function into an ON-set, OFF-set and DC-set). A vertical bar **|** may be used to separate the multiple-valued fields in the input file.

If the size of the multiple-valued field is less than zero, than a symbolic field is interpreted from the input file. The absolute value of the size specifies the maximum number of unique symbolic labels which are expected in this column. The symbolic labels are white-space delimited strings of characters.

To perform a *kiss*-style encoding problem, the keyword **.kiss** should be included in the file. The third to last variable on the input file must be the symbolic "present state", and the second to last variable must be the "next state". As always, the last variable is the output. The symbolic "next state" will be hacked to be actually part of the output.

EXAMPLE #1

A two-bit adder which takes in two 2-bit operands and produces a 3-bit result can be described completely in minterms as:

```
# 2-bit by 2-bit binary adder (with no carry input)
.i 4
.o 3
0000  000
0001  001
0010  010
0011  011
0100  001
0101  010
0110  011
0111  100
1000  010
```

```
1001  011
1010  100
1011  101
1100  011
1101  100
1110  101
1111  110
```

It is also possible to specify some extra options, such as:

```
# 2-bit by 2-bit binary adder (with no carry input)
.i 4
.o 3
.ilb a1 a0 b1 b0
.ob s2 s1 s0
.pair 2 (a1 b1) (a0 b0)
.phase 011
0000  000
0001  001
0010  010
      .
      .
      .
1111  110
.e
```

The option *.pair* indicates that the first binary-valued variable should be paired with the third binary-valued variable, and that the second variable should be paired with the fourth variable. The function will then be mapped into an equivalent multiple-valued minimization problem.

The option *.phase* indicates that the positive-phase should be used for the second and third outputs, and that the negative phase should be used for the first output.

EXAMPLE #2

This example shows a description of a multiple-valued function with 5 binary variables and 3 multiple-valued variables (8 variables total) where the multiple-valued variables have sizes of 4 27 and 10 (note that the last multiple-valued variable is the "output" and also encodes the ON-set, DC-set and OFF-set information).

```
.mv 8 5 4 27 10
.ilb in1 in2 in3 in4 in5
.label var=5 part1 part2 part3 part4
.label var=6 a b c d e f g h i j k l m n
    o p q r s t u v w x y z a1
.label var=7 out1 out2 out3 out4 out5 out6
    out7 out8 out9 out10
0-010|1000|100000000000000000000000000|0010000000
10-10|1000|010000000000000000000000000|1000000000
0-111|1000|001000000000000000000000000|0001000000
0-10-|1000|000100000000000000000000000|0001000000
00000|1000|000010000000000000000000000|1000000000
00010|1000|000001000000000000000000000|0010000000
01001|1000|000000100000000000000000000|0000000010
0101-|1000|000000010000000000000000000|0000000000
0-0-0|1000|000000001000000000000000000|1000000000
10000|1000|000000000100000000000000000|0000000000
11100|1000|000000000010000000000000000|0010000000
```

```
10-10|1000|00000000001000000000000000|0000000000
11111|1000|00000000000100000000000000|0010000000
                                      .
                                      .
                                      .
11111|0001|00000000000000000000000001|0000000000
```

EXAMPLE #3

This example shows a description of a multiple-valued function setup for *kiss*-style minimization. There are 5 binary variables, 2 symbolic variables (the present-state and the next-state of the FSM) and the output (8 variables total).

```
.mv 8 5 -10 -10 6
.ilb io1 io0 init swr mack
.ob wait minit mrd sack mwr dli
.type fr
.kiss
--1--     -        init0     110000
--1--     init0    init0     110000
--0--     init0    init1     110000
--00-     init1    init1     110000
--01-     init1    init2     110001
--0--     init2    init4     110100
--01-     init4    init4     110100
--00-     init4    iowait    000000
0000-     iowait   iowait    000000
1000-     iowait   init1     110000
01000     iowait   read0     101000
11000     iowait   write0    100010
01001     iowait   rmack     100000
11001     iowait   wmack     100000
--01-     iowait   init2     110001
--0-0     rmack    rmack     100000
--0-1     rmack    read0     101000
--0-0     wmack    wmack     100000
--0-1     wmack    write0    100010
--0--     read0    read1     101001
--0--     read1    iowait    000000
--0--     write0   iowait    000000
```

EXAMPLE 4

This example shows the use of the **.symbolic** keyword to setup a multiple-valued minimization problem.

```
.i 15
.o 4
.ilb SeqActive<0> CacheOp<6> CacheOp<5> CacheOp<4>
        CacheOp<3> CacheOp<2> CacheOp<1> CacheOp<0>
        userKernel<0> Protection<1> Protection<0>
        cacheState<1> cacheState<0> PageDirty<0>
        WriteCycleIn<0>
```

```
.ob CacheBusy<0> dataMayBeValid<0> dataIsValid<0>
        WriteCycleOut<0>

.symbolic CacheOp<6> CacheOp<5> CacheOp<4> CacheOp<3>
        CacheOp<2> CacheOp<1> CacheOp<0> ;
        FET NA PHY_FET PR32 PRE_FET PW32 RA32 RD32
        RD64 RDCACHE RFO32 RFO64 TS32 WR32 WR64 WRCACHE ;

.symbolic Protection<1> Protection<0> ;
        PROT_KRO_UNA PROT_KRW_UNA PROT_KRW_URO PROT_KRW_URW ;

.symbolic cacheState<1> cacheState<0> ;
        CS_Invalid CS_OwnPrivate CS_OwnShared CS_UnOwned ;

.p 22
0000001--010110 0001
0000001-1-00110 0001
00001011-01011- 0100
000010111-0011- 0100
0000--001--01-- 0100
0000-10--0-1--- 0100
0000-10-1--1--- 0100
00000-0--0-1--- 0100
00000-0-1--1--- 0100
0000-10--0--1-- 0100
0000-10-1---1-- 0100
00000-0--0--1-- 0100
00000-0-1---1-- 0100
---1---------- 1000
--1----------- 1000
-1------------ 1000
1------------- 1000
-------0------- 1000
----1---------- 1000
-----0--------- 1000
------0-------- 1000
--------------1 1110
.e
```

Index
