Simple illustration of permutation technique

JINGREN WANG, Raina Technology, China

Permutation is an important technique in turth table based algorithm, how to permute efficiently is one of the key problems in logic synthesis. This draft gives illustration based on some permutation functions in *mockturtule*, helps students understand how permutation is performed.

1 BACKGROUND

1.1 Turth table

A turth table(TT) representation will be given here for the convinence of future illustration. The size of the TT is determined by the number of variables, TT of a complete boolean function of n varibles will have the size of 2^n, and there will be 2^{2^n} different boolean functions can be implemented.

1.2 Mask

Informally, mask is a simple idea on "only consider information related parts", which will seperate any unrelated information out. Mask exists in multiple areas such as network engineering, here under the context of bit level manipulation, it is simply a way of bit extraction.

2 MAGIC MASK

When we talking about permuting variables in TT, we are referring methods using bit level masks, in Knuth's *The Art of Programming*, these kind of masks are referred to as *Magic Masks*. First go through the content in file *lib/kitty/kitty/detail/constants.hpp*, and related unit tests cases, see if any of the hard coded *constants* makes sense.

2.1 Example of 2 variables

Given two varibles a,b, any complete boolean function can be represented by the TT in table 2. If a swap happens between a and b, the function representation will be in table 3. Obviously the second line and the third line is sawpped due to the variable value change in the function result(TT), while the first and last line remmained the same. The sawp is very intuitive when we look at the table, when formalizing to a swap function, a process is given(no order is needed here for the first 3 items):

1. Masks the unchanged line, such as the blue line in the example table 2 and 3.
2. Retrieves the result needs left/right shift.
3. Records how many bits are needed for the shifting.
4. Shit the bits need to be permutated.
5. Concatenate all changed and unchanged bits together.

More formally, under these two variables context, this will be interpreted as:

m_{un} means the mask of unchanged bits, while each one of the m_0 and m_1 represents the mask of the bit needs to be permutated. The shift bits is 1, which means m_0 needs to be shifted 1 bit left and m_1 needs to be shifted 1 bit right. This will result in a final formula of

$$tt_{res} = tt_{ori} \,\&\, m_{un} \mid tt_{ori} \,\&\, m_0 \ll n_s \mid tt_{ori} \,\&\, m_1 \gg n_s$$

in which tt_{res} represents the result TT, tt_{ori} represents the original TT, >> and << represent the bit level shifting, n_s represents the number of shifting, here it is referring to 1.

Author's address: Jingren Wang, jingrenwangcyber@gmail.com, Raina Technology, Hangzhou, Zhejiang, China.

Table 1. Masks on permutation

a_3	a_2	a_1	a_0	c_v
1	0	0	1	m_{un}
0	0	1	0	m_0
0	1	0	0	m_1

Table 2. Original boolean function of 2 variables

b	a	$f(a,b)$
0	0	a_0
0	1	a_1
1	0	a_2
1	1	a_3

Table 3. After wap of 2 variables

b	a	$f(a,b)$
0	0	a_0
1	0	a_2
0	1	a_1
1	1	a_3

Table 4. Original boolean function of 3 variables

c	b	a	$f_{a,b}(a,b,c)$	$f_{a,c}(a,b,c)$
0	0	0	a_0	a_0
0	0	1	a_1	a_1
0	1	0	a_2	a_2
0	1	1	a_3	a_3
1	0	0	a_4	a_4
1	0	1	a_5	a_5
1	1	0	a_6	a_6
1	1	1	a_7	a_7

2.2 Example of 3 variables

A function example of 3 variables is explained here just for a more detailed and clear view of permutation, since 3 variables have various different ways of permuation, the method is the same as 2 varibels, the original TT is represented as table 4, when permuted on variable a and variable b, which is the variable index 0 and 1, we have result in table 5.In addition, permute variable a and variable c, we have the result in table 6. Readers should be able to manually draw the result of the sawppimg varibles b and variables c.

Table 5. Swap 2 variables(a,b) of 3-input boolean function

c	b	a	$f_{a,b}(a,b,c)$
0	0	0	a_0
0	1	0	a_2
0	0	1	a_1
0	1	1	a_3
1	0	0	a_4
1	1	0	a_6
1	0	1	a_5
1	1	1	a_7

Table 6. Swap 2 variables(a,c) of 3-input boolean function

c	b	a	$f_{a,c}(a,b,c)$
0	0	0	a_0
1	0	0	a_4
0	1	0	a_2
1	1	0	a_6
0	0	1	a_1
1	0	1	a_5
0	1	1	a_3
1	1	1	a_7

2.3 Shift calculation

The shift value is determined by the difference of the two variables, by checking example mentioned in section 2.1 and section 2.2, when swapping variables, if varible a and b have value $\{0,0\}$ or $\{1,1\}$ before swapping, then the result will not be changed. Also note that when swapping variable a and b which has variable index 0 and 1, the result TT always constructed by a shift value of 1. The hint has been given to us that every time an encounter of a value of $\{0,1\}$ or $\{1,0\}$, the result will be permuted.

Now, consider from index perspective, when permuting variable index 0 and index 1, the only focus is on the last two column, and the set with value {0,1} and {1,0}, since other variables are not changed, so the permutation can be extend to line $\{X, 0, 1\}$ and line $\{X, 1, 0\}$, where X indicates other variables $\{x_2, x_3, x_4, ...\}$. Hence, we could say that for every number n of varibles, the swap between variable index 0 and variable index 1 will have a shift value(a shift distance) n_s between line $\{X, 0, 1\}$ and line $\{X, 1, 0\}$, which can also be interpreted as

$$n_s = 2^1 - 2^0 = 1$$

since {0,1} is the value of 2^0 and {1,0} is the value of 2^1. Also, for 3 varibles, when we permute varible index 0 and varible index 2, this create a condition for middle index to exist, so it could be interpreted as swapping condition of $\{0,X,1\}$ and $\{1,X,0\}$, in which X can be and can only be varible with index 2, $\{X = x_1\}$. So the shift different between two lines with $\{0,x_1,1\}$, $\{1,x_1,0\}$, hence,

$$n_s = 2^2 - 2^0 = 3$$

which can also be verified by the swap between a_1 and a_4, a_3 and a_6 in the table 6.
The generalized form of calculating the shift difference then becomes:

$$n_s = 2^{i_1} - 2^{i_2}, i_1 \geqslant i_2$$

This gets us the conclusion that for a pair of specific varible index i_1 and i_2 the n_s remains the same, no matter how much varible is involved.

2.4 Masks

Masks here is generally only for data retrival in TT, as shown in section 2.3, although n_s remains the same, with different number of varibles, the TT has different length. In *lib/kitty/kitty/detail/constants.hpp*, the masks has been recorded in a "`ppermutation_masks`" varible. It is a 3 dimision varible which the first dimision represents the i_1 and the second dimision represents the i_2, the retrived mask contains 3 varibles, the first one is the m_{un}, the second one is the m_p for the first i_1 TT isolation, and the third one is m_q for the i_2 TT isolation, which in detial,

$$tt_{un} = tt_{ori} \,\&\, m_{un}$$
$$tt_{i_1} = tt_{ori} \,\&\, m_p$$
$$tt_{i_2} = tt_{ori} \,\&\, m_q$$
$$tt_{res} = tt_{un} \mid tt_{i_1} \ll n_s \mid tt_{i_2} \gg n_s$$

For example, take case in table 6, the permuation of varible index 0 and index 2 can be interpreted easily as:

$$tt_{un} = \{a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0\} \,\&\, UINT64\_C(0xa5a5a5a5a5a5a5a5)$$
$$tt_{i_1} = \{a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0\} \,\&\, UINT64\_C(0x0a0a0a0a0a0a0a0a)$$
$$tt_{i_2} = \{a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0\} \,\&\, UINT64\_C(0x5050505050505050)$$

after padding the all bits with 0, you will get,

$$tt_{un} = \{a_7, 0, a_5, 0, 0, a_2, 0, a_0\}$$
$$tt_{i_1} = \{0, 0, 0, 0, a_3, 0, a_1, 0\}$$
$$tt_{i_2} = \{0, a_6, 0, a_4, 0, 0, 0, 0\}$$
$$tt_{res} = tt_{un} \mid tt_{i_1} \ll 3 \mid tt_{i_2} \gg 3$$
$$tt_{res} = \{a_7, a_3, a_5, a_1, a_6, a_2, a_4, a_0\}$$

which is the same as what the TT in 6 shows.
The examples give an specific illustration, other number of varibles can also use this magic mask technique in *mockturtle* function api.