Folding of Logic Functions and Its Application to Look Up Table Compaction

Shinji Kimura — System LSI, IPS, Waseda University, 2-2 Hibikino 808-0135, Japan
Takashi Horiyama — Informatics, Kyoto University, Kyoto 606-8501, Japan
Masaki Nakanishi, Hirotsugu Kajihara — Information Science, Nara Institute of Science and Technology, 8916-5 Takayama 630-0101, Japan

Abstract

The paper describes the folding method of logic functions to reduce the size of memories for keeping the functions. The folding is based on the relation of fractions of logic functions. We show that the fractions of the full adder function have the bit-wise NOT relation and the bit-wise OR relation, and that the memory size becomes half (8-bit). We propose a new 3-1 LUT with the folding mechanisms whcih can implement a full adder with one LUT. A fast carry propagation line is introduced for a multi-bit addition. The folding and fast carry propagation mechanisms are shown to be useful to implement other multi-bit operations and general 4 input functions without extra hardware resources. The paper shows the reduction of the area consumption when using our LUTs compared to the case using 4-1 LUTs on several benchmark circuits.

1 Introduction

Field Programmable Gate Arrays (FPGAs) are very useful device to implement application specific circuits ([1], [2]). Recent FPGAs are usually constructed from Look-Up Tables (LUTs), which include memories to keep truth tables of logic functions and multiplexers to select the value of memories ([3], [4], [5]). Such LUT is very useful, but needs huge circuit resources and much delay time compared to simple logic gates. So LUT-based FPGAs usually suffer from the low area utilization and the low speed.

In the paper, we propose the folding method of logic functions to reduce the memory size for storing logic functions. The basic idea of the folding is to use the relation of fractions of logic functions. If a representation (such as a truth table) of a logic function includes 2 same parts, we can omit one of them and the size can be reduced. We show that a full adder (two functions with 3 inputs) can be represented with only 8 bit memories using the bit-wise NOT relation and the bit-wise OR relation. The number of bits is half compared to that of the usual 16 bit representation. Note that the size is equal to the memory size of LUTs.

We devise a new 3-1 LUT architecture including the NOT and OR folding mechanisms, which can implement a full adder with only one 3-1 LUT. With the fast carry propagation line, we can implement additions, AND/OR operations, the equality, and the arithmetic comparisons ($<$, $\leq$, $>$, $\geq$) on multiple-bit inputs.

The folding and the carry propagation mechanisms are also useful to implement general 4 input functions. We show that more than half of general 4 input functions can be implemented with two 3-1 LUTs. We compare the area consumption on the case using our 3-1 LUTs and that using 4-1 LUTs. By using our 3-1 LUTs, the area can be reduced up to 56 % on several benchmark circuits.

2 Basic Architecture of Look Up Table

A 3-1 Look Up Table (LUT) has 3 input ports a, b, c, and one output port y. The outputs of memories are connected to y via a multiplexer, the control lines of which are connected to the inputs a, b and c. The 8-1 multiplexer is usually implemented with a pass-transistor circuit.

A 3-1 LUT can implement any 3 input logic function $f(a, b, c)$ by storing the logic function values of $f(0, 0, 0)$, $f(0, 0, 1)$, $f(0, 1, 0)$, $f(0, 1, 1)$, $f(1, 0, 0)$, ..., $f(1, 1, 1)$ to the memories with this order. The sequence is usually called as the vector representation of $f(a, b, c)$. Note that if $(a, b, c) = (0, 0, 0)$ then $f(0, 0, 0)$ is selected by the multiplexer and y becomes the correct value.

An FPGA is a 2-dimensional array of such LUTs. FPGAs are used to implement logic functions, many of which include adders and comparators as key elements. So commercial FPGAs introduce mechanisms to implement such functions: one is a clustering of two 3-1 LUTs to implement a full adder and another is the introduction of a fast carry chain and a fast AND/OR cascade chain to pass special signals between LUTs [6].

3 Folding of Logic Functions

In this section, we introduce the folding of logic functions. To show the idea of the folding method, we consider two logic functions $f(a, b, c)$ and $g(a, b, c)$ whose vector representations are 00011110 and 10001001.

The vector representation 00011110 of f can be divided into two parts 0001 and 1110 depending on the value of a, and we can see that there exists the bit-wise NOT relation between these parts. That is

$$f(a, b, c) = \overline{a} \cdot f(0, b, c) + a \cdot \overline{f(0, b, c)}$$

where '$\overline{\ }$', '$\cdot$' and '$+$' denote logical NOT, AND and OR, respectively.

Because of the relation, we have only to keep $f(0, b, c)$. $f(1, b, c)$ is generated from $f(0, b, c)$ and an extra NOT gate. The method is called as the folding with NOT.

Second, we consider g (= 1000 1001). 1000 and 1001

0-7803-7607-2/02/$17.00 ©2002 IEEE

Table 1: Truth Table of A Full Adder

c_{in}	a	b	s	c_{out}
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

of g have no simple relation such as the bit-wise NOT, but we can find the bit-wise OR relation by considering 0001 of f. That is the OR of 1000 and 0001 becomes 1001:

$$g(a, b, c) = \overline{a} \cdot g(0, b, c) + a \cdot \{g(0, b, c) + f(0, b, c)\}$$

The method is called as the folding with OR. Note that the OR-folding uses the relation of two logic functions.

We can consider the folding methods with AND, EXOR, etc. The estimation of the usefulness and the area consumption of such foldingd is one of our future works.

We can apply these folding methods to the representation of a full adder in Table 1. A full-adder has 3 inputs c_{in}, a, b, and 2 outputs s and c_{out}, where c_{in} (c_{out}) denotes the carry input (output), and s denotes the sum. Usually we need 16 bit memories to represent both s and c_{out}.

The vector representations of s and c_{out} are 0110 1001 and 0001 0111, respectively. On s, there exists the bit-wise NOT relation between 0110 and 1001. On c_{out}, we can find the bit-wise OR relation between 0001 and 0111 by considering 0110 of s. With the extra NOT and OR gates, we can represent s and c_{out} with only 8 bit memories.

4 LUT Compaction Based on Folding

In this section, we show a new LUT architecture with NOT and OR folding mechanisms described in the former section. The folding mechanism can reduce the memory size for representing a full adder to 8 bits.

The proposed new LUT architecture is shown in Fig. 1. The LUT is constructed from 8 bit memories, an 8-1 multiplexer, two 2-1 multiplexers for the carry propagation (c_{in}), one mode bit and control gates. Dotted circles in the figure show the extra hardware compared to a simple 3-1 LUT.

If the mode bit is 0, then the LUT works as a usual 3-1 LUT depending on inputs a, b and c. Note that c_{in} should be 0 for the correct operation and c_{out} becomes 0 if c_{in} is 0.

If the mode bit is 1, then the LUT works in an arithmetic mode, and we can use c_{out} and y depending on b, c and c_{in}. y implements the NOT-folding mechanism, where the lower 4 bit memories themselves or the bit-wise NOT of them are selected by c_{in}. On the other hand, c_{out} implements the OR-folding, where the upper 4 bit memories or the OR of the upper and lower memories are selected by c_{in}.

For implementing multi-bit adders, the proposed LUTs are connected in series from lower to upper, where the c_{in}

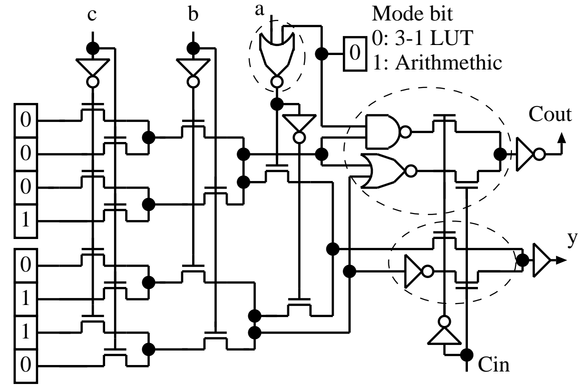

Figure 1: A 3-1 Look Up Table with NOT and OR Folding

Table 2: Truth Tables of Equality and Comparisons

c_{in}	a	b	$=$	$<, \leq$	$>, \geq$
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	0	0	0
1	0	0	1	1	1
1	0	1	0	1	0
1	1	0	0	0	1
1	1	1	1	1	1

of the upper LUT is connected to c_{out} of the lower LUT. Note that we place the multiplexers with c_{in} at the output side, that is because the fast carry propagation is important.

Each LUT can implement a full-adder with only one 3-1 LUT (including 8 bit memories) and that is just the half compared to the usual 4-1 LUT (when used as two 3-1 LUTs). k bit adder can be implemented using k 3-1 LUTs.

5 Mapping Capability of 3-1 LUTs with Foldings and Carry Propagation

5.1 Mapping of Equality, Comparison, AND/OR

The carry propagation mechanism with the OR folding can be applied to mapping several circuits, such as equality, arithmetic comparisons, and multi-bit AND and OR.

The relation of two binary numbers can be reduced to the relation of each bit with a signal passing through these bits. When mapping to our LUTs, the result of the operation on each bit with c_{in} is passed to c_{out}.

Truth tables of the equality and the comparisons are shown in Table 2. c_{in} is passed from the least significant bit to the greatest significant bit, and if c_{in} is 0, then the relation is violated at some preceding position. For example, at $=$ function, if c_{in} is 0 then two inputs are not equal at some preceding position and we should pass 0 to the upper side. So truth table becomes 0000 when c_{in} is 0.

The reason why we can map these function to our 3-1 LUT is that the OR of upper 4 bits and lower 4 bits becomes lower 4 bits. That is if $f(1, a, b) = f(0, a, b) + f(1, a, b)$ then we can map f to the carry mechanism with OR folding. It is easy to see that 0000 and 1001 of $=$ satisfy the relation.

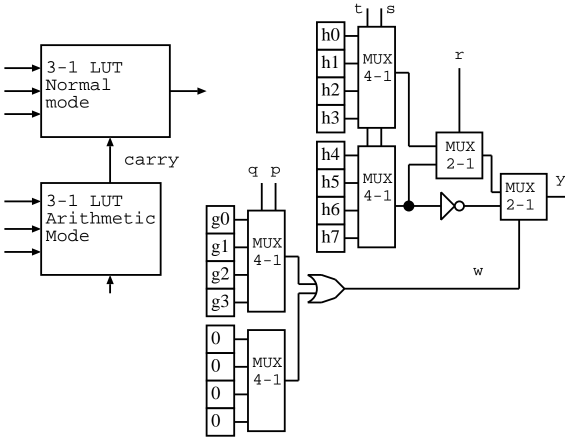

Figure 2: A Basic Structure to Implement 4 Input Functions

$<$ $(\leq)$ and $>$ $(\geq)$ also satisfy the relation. Equality and comparisons of k bit numbers can be decided with k LUTs.

As an example, we show a mapping method of $=$.

- At the least significant bit, the LUT is used in the arithmetic mode. In general, we cannot fix the value of c_{in} to the LUT, and we set 1001 0000 to the LUT for ignoring the effect of c_{in}.
- At the intermediate bit, we set 0000 1001 to the LUT.
- At the greatest significant bit, LUT is used in the normal mode to pass the result to y. We set 0 to the input a, and we set 0000 0110 to the LUT. Note that y becomes 1001 by the NOT folding when $c_{in} = 1$.

We can map the multi-bit AND (OR) function with almost the same manner. The vector representation of AND of each 2 bits is 0000 0001 depending on c_{in}. Since 0001 = 0000 + 0001 is satisfied, we can use the carry propagation mechanism of c_{out}. The multi-bit OR function can be represented as 0111 1111, and has the same property. Thus k bit AND (OR) function can be implemented with $k/2$ LUTs.

5.2 Mapping Capability of 4 Input Functions

In the section, we show a mapping method of 4 input logic functions with the carry propagation mechanism. If we can map a 4 input function to two 3-1 LUTs, then the resource usage is almost the same as a 4-1 LUT.

It is easy to see that any 4 input function can be mapped to three 3-1 LUTs with outer wireing resources. It can be shown that any 4 input function can be mapped to three 3-1 LUTs with only carry line by extending the method below.

A basic structure for implementing 4 input logic functions is the one in Fig. 2, where the upper LUT is used in the normal mode, and the lower LUT is used in the arithmetic mode. In general, we cannot fix c_{in} of the lower LUT, so we use the lower LUT for implementing only two input functions by setting 0000 to the lower 4 bit memories.

Since the output of the upper LUT depends on c_{in}, the function is represented as follows, where w is the carry generated by the lower LUT, and y is the final output.

$$w = g(p, q)$$
$$y = \overline{w} \cdot h(r, s, t) + w \cdot \overline{h(1, s, t)}$$

Table 3: Constraints on Mapping 4 Input Logic Functions

$g(0,0)=0$	$g(0,1)=0$	$g(1,0)=0$	$g(1,1)=0$
$f(0,0,c,d)$ $= h(0,c,d)$	$f(0,1,c,d)$ $= h(1,c,d)$	$f(1,0,c,d)$ $= h(0,c,d)$	$f(1,1,c,d)$ $= h(1,c,d)$
$g(0,0)=1$	$g(0,1)=1$	$g(1,0)=1$	$g(1,1)=1$
$f(0,0,c,d)$ $= \overline{h(1,c,d)}$	$f(0,1,c,d)$ $= \overline{h(1,c,d)}$	$f(1,0,c,d)$ $= \overline{h(1,c,d)}$	$f(1,1,c,d)$ $= \overline{h(1,c,d)}$

$f(a, b, c, d)$, $g(a, b)$ and $h(b, c, d)$ are assumed.

If a 4 input function f can be represented using g and h, then f can be mapped to two 3-1 LUTs.

The number of inputs of g and h is 5 $\{p, q, r, s, t\}$, and we can duplicate one of $\{a, b, c, d\}$. At first, we consider a case where b is duplicated under an order a, b, c, d: checked paris are <$g(a, b), h(b, c, d)$> and <$g(a, b), h(c, b, d)$>.

For each variable order, we should check the relation of f and h depending on the value of g. If $g(0, 0)$ is 0, $f(0, 0, c, d) = h(0, c, d)$. If $g(0, 0)$ is 1, $f(0, 0, c, d) = \overline{h(1, c, d)}$. The relations are shown in Table 3. We can check the mapping capability by testing the 16 $(= 2^4)$ cases of $g(a, b)$ for a variable order a, b, c, d. By checking all 24 $(= 4!)$ variable orders, we can cover all cases.

We have implemented the checking algorithm in C and checked all 4 input logic functions. The algorithm does the exhaustive search, and needs 166.9 seconds to check all 65536 functions on a PC with 733 MHz Pentium-III-M with 384 MB of memory. 31848 functions can be implemented with two 3-1 LUTs. On NPN (input Negation, input Permutation, output Negation) equivalent classes of 4 input logic functions, we can map 109 classes in 222 classes.

5.3 Evaluation on Benchmark Circuits

We have applied the mapping algorithm of 4 input functions to benchmark circuits. We use circuits designed in our laboratories such as a 16 bit combinational multiplier, a speech recognition circuit for monosyllables, a circuit to detect face area using the color information, a circuit to track an eye movement, a Java processor, and a 16 bit pipeline processor (pcpu). We also use fir, iir, cordic and DFT circuits in a textbook of FPGA[2].

Circuits are compiled for Altera APEX FPGA EP20K series using Quartus II version 1.1, with area optimization option and with speed optimization option. The compiled circuits are represented as a set of 4 input functions (4-1 LUTs). We apply the mapping algorithm to these functions, and sum-up the number of 3-1 LUTs.

To evaluate the area of mapped circuits, the areas of our 3-1 LUT and a usual 4-1 LUT are needed. So we have designed and implemented our 3-1 and usual 4-1 LUTs using VDEC EXD libraries. From the layout results, we can see that the area of our 3-1 LUT is about 0.56 times smaller than that of 4-1 LUT. The ratio is used to normalize the number of our 3-1 LUTs when comparing with that of 4-1 LUTs.

We have found that the memory area for storing logic

Table 4: Mapping Capability Results on Benchmark Circuits

Name	Num. of 4-1 LUTs all	(4-in)	Num. of 3-1 LUTs all	(2x3-1)	Normalized Num. of 3-1 LUTs	Area Ratio
	A		B		$C = 0.56 \times B$	C/A
Mult16x16	381	(127)	508	(127)	284	0.747
SpchRecog_area	13527	(7151)	21595	(7073)	12093	0.894
SpchRecog_speed	17151	(7891)	25238	(7769)	14133	0.824
face_recognition_area	11368	(5463)	17043	(5413)	9544	0.840
face_recognition_speed	15374	(6022)	21627	(5871)	12111	0.788
imouse_top_area	11184	(3004)	14644	(2835)	8200	0.733
imouse_top_speed	12276	(3748)	16352	(3478)	9157	0.746
javachip_area	7708	(5836)	13877	(5809)	7771	1.008
javachip_speed	11145	(7342)	18579	(7311)	10404	0.934
pcpu_area	1213	(773)	2096	(758)	1173	0.968
pcpu_speed	1593	(956)	2552	(955)	1429	0.897
cordic_area †	751	(100)	851	(100)	476	0.635
cordic_speed †	893	(182)	1075	(182)	602	0.674
fir_gen †	732	(248)	1292	(32)	723	0.988
iir_par †	670	(0)	670	(0)	375	0.560
DFT_area †	312	(40)	354	(40)	198	0.635
DFT_speed †	299	(41)	340	(41)	190	0.637

†: The circuit is in [2]. (4-in): LUTs implementing 4 input functions.
(2x3-1): The number of 4-1 LUTs in (4-in) which can be mapped to two 3-1 LUTs.

functions is the major part of LUTs. The number of bits of our 3-1 LUT is 9 and that of the 4-1 LUT is 17. Both LUTs include one extra bit for the mode selection.

The experimental results are shown in Table 4. The table shows the number of 4-1 LUTs, the number of real 4 input functions, the number of our 3-1 LUTs computed using the mapping algorithm, the number of functions mapped to two 3-1 LUTs, the number of 3-1 LUTs normalized with 0.56, and the ratio of 4-1 LUTs and normalized 3-1 LUTs.

We can see that the area ratio is varying from 0.56 to 1.008. Only one design (javachip_area) needs much area by using our 3-1 LUTs, and other designs gain better result on the area consumption. IIR filter gains the best result 0.56 since they include only 3 input functions such as adders. Processors include complex random controls with 4 input functions, but the mapping results are not so bad.

6 Conclusions

We have proposed the folding method of logic functions to reduce the memory size for storing logic functions. In the folding, the relation of fractions of logic functions is used and some part is generated from another part using extra logic gates. By the NOT and OR foldings, a full adder can be represented with only 8 bit memories.

We have shown a new LUT architecture with the NOT and OR folding mechanisms and the carry propagation mechanism, and a mapping method of k bit adders, equality, comparisons and k bit AND/OR to the new 3-1 LUTs.

We have also shown a mapping algorithm of a 4 input logic function to two 3-1 LUTs, and found that 35192 in 65536 functions can be mapped to two 3-1 LUTs.

We have implemented our 3-1 LUT with usual cell based design, and found that the area of our 3-1 LUT is 0.56 times smaller than that of the usual 4-1 LUT. The area consumption of our 3-1 LUTs are measured from the normalized number of LUTs with 0.56 on several benchmarks, and we have obtained the reduced area for almost all benchmarks.

Acknowledgement We would like to thank Prof. Katsumasa Watanabe and members of Watanabe-lab at NAIST for their discussions and comments. We should thank to VDEC in Univ. of Tokyo for their suppot on EDA tools. The work is supported in part by funds from the Ministry of ECSST, FAIS at Kitakyushu, NEC and Hitachi.

References

[1] D. Buell et.al. *Splash2: FPGAs in a Custom Computing Machine*. IEEE Coputer Science Press, 1996.

[2] Uwe Meyer-Baese. *Digital Signal Processing with Field Programmable Gate Arrays*. Kluer Academic Publishers, 2001.

[3] Varghese George and Jan M. Rabaey. *Low Energy FPGA*. Kluer Academic Publishers, 2001.

[4] *APEXII Programmable Logic Device Family Data Sheeet*. Altera Corporation, 2000.

[5] Scott Hauck, Matthew M. Hosler, and Thomas W. Fry. High-Performance Carry Chains for FPGA's. *IEEE Trans. on Very Large Scale Integration Systems*, Vol. 8, No. 2, pp. 138–147, April 2000.

[6] *http://www.altera.co.jp*.