The Implementation and Verification of a Conditional Sum Adder

Jungang Han

Glen Stone

Department of Computer Science
University of Calgary
Calgary, Alberta
Canada T2N 1N4

ABSTRACT

In this paper we first formulate the Conditional Sum Addition (CSA) algorithm, then design an area-time efficient Conditional Sum Adder in CMOS. We also design a Binary Look-ahead Carry adder and a fast ripple carry adder in the same technology for the comparison of their performances. finally we formally prove that the CMOS implementation of the CSA adder is correct (i.e. the implementation meets the specification of the intended behavior) by using Mike Gordon's Higher Order Logic (HOL) system.

1. Introduction

Among the possible choices of fast binary adders for VLSI implementation , carry look-ahead adders are widely used. In the case of relatively small word-size, ripple adders still have their advantages. Many practical designs combine different adders to achieve proper compromise between time and area requirements.

Brent and Kung [1] have presented a regular layout of carry look-ahead adder by reformulating the computation of carrys , we call this kind of adder Binary Look-ahead Carry (BLC) adder [2]. BLC adder performs addition of two n-bit numbers in O(log n) time using O(n log n) area. It has been implemented by Brent and Ewin in NMOS [3].

An algorithm for fast addition -- Conditional Sum Addition (CSA) was presented by J.Sklansky [4] early in 1960. It is possible to design a 64 bit adder with up to five or six times the ripple adder performance by using CSA algorithm, but it needs larger size of area [5].

In order to make the concepts of conditional sum addition clear and the algorithm appropriate to VLSI implementation, in section 2, we formulate the CSA algorithm, which was described in [4] only by examples. In section 3 , we design a CSA adder and a BLC adder in CMOS. In section 4, we compare the performance of CSA adder with BLC adder and fast ripple adder. In section 5, we briefly describe the formal verification of the CMOS implementation of the CSA adder by using Gordon's HOL system [6].

2. The CSA Algorithm

In order to specify the CSA algorithm formally, we introduce the following definitions.

Definition 1

let

$$A = A_n \ldots A_2 A_1$$

and

$$B = B_m \ldots B_2 B_1$$

be n-bit and m-bit binary numbers respectively, where A_i ,B_i are either 0 or 1, We refer to the (m + n)-bit binary number

$$C = A_n \ldots A_2 A_1 B_m \ldots B_2 B_1$$

as the **concatenation** of A and B , and use the symbol "&" to denote the operation of concatenation, i.e.

$$C = A \& B.$$

We say that A is a n-bit piece of C and B is a m-bit piece of C.

For example, if A = 101 and B = 1001, then

$$C = A \& B = 1011001.$$

Definition 2

Let

$$A = A_{i+j} \ldots A_i$$

and

$$B = B_{i+j} \ldots B_i$$

be two binary numbers or two (j+1)-bit pieces of other binary numbers. We define that the **conditional 0-sum** S_0 (**conditional 1-sum** S_1) is the bit i to bit i+j of the sum of A and B under the assumption that the carry from bit i - 1 to bit i is 0 (1), and that the **conditional 0-carry** C_0 (**conditional 1-carry** C_1) is the carry from bit i+j to bit i+j+1 while adding A and B under the assumption that the carry from bit i - 1 to bit i is 0 (1).

For example, if A = 1010 and B = 0101 then their conditional 0-sum S_0, 1-sum S_1 , 0-carry C_0 and 1-carry C_1 are

$$S_0 = 1111$$
$$S_1 = 0000$$
$$C_0 = 0$$
$$C_1 = 1 .$$

Particularly, if j = 0 in definition 2 then A and B are 1-bit binary numbers A = A_1 and B = B_1 , we can use the following logic expressions to compute the conditional sums and carrys.

$$S_0 = A_1 \oplus B_1$$
$$S_1 = \tilde{} \ (A_1 \oplus B_1)$$
$$C_0 = A_1 \ /\backslash \ B_1$$
$$C_1 = A_1 \ \backslash/ \ B_1$$

where

$\oplus$ denotes exclusive or
/\ denotes logical and
\/ denotes logical or
~ denotes logical not

and the logic 1 (true) and 0 (false) are considered to be equal to number 1 and 0 respectively .

Obviously the following lemma is correct .

Lemma 1

For any two n-bit binary numbers (n > 0) , if their conditional 0-sum , 1-sum , conditional 0-carry and 1-carry are S_0,S_1,C_0,and C_1 respectively then

$$S_1 = S_0 + 1 \ (MOD\ 2^n)$$

and

$$C_1 \geq C_0.$$

If the summand A = A_h & A_l and the addend B = B_h & B_l (provided that A_h and B_h have equal number of bits), how to compute the conditional sums and carrys of A and B using the conditional sums and carrys of A_h and B_h as well as the conditional sum and carrys of A_l and B_l ? The following theorem answers this question.

Theorem 1

Let A_l = $A_{i+j} \ldots A_{i+1}A_i$,
A_h = $A_{i+j+k} \ldots A_{i+j+1}$,
B_l = $B_{i+j} \ldots B_{i+1}B_i$,
B_h = $B_{i+j+k} \ldots B_{i+j+1}$
where i $\geq$ 0 , j > 0 and k > 0

be binary numbers or pieces of binary numbers, and
the conditional 0-sum of A_h and B_h is S_0_h,
the conditional 1-sum of A_h and B_h is S_1_h,
the conditional 0-carry of A_h and B_h is C_0_h,
the conditional 1-carry of A_h and B_h is C_1_h,
the conditional 0-sum of A_l and B_l is S_0_l,
the conditional 1-sum of A_l and B_l is S_1_l,
the conditional 0-carry of A_l and B_l is C_0_l ,
the conditional 1-carry of A_l and B_l is C_1_l,
the conditional 0-sum of (A_h & A_l) and (B_h & B_l) is S_0,
the conditional 1-sum of (A_h & A_l) and (B_h & B_l) is S_1,
the conditional 0-carry of (A_h & A_l) and (B_h & B_l) is C_0,
the conditional 1-carry of (A_h & A_l) and (B_h & B_l) is C_1.

then we have the following equations.
$S_0 = S_1$_h & S_0_l,if C_0_l = 1 otherwise $S_0 = S_0$_h & S_0_l ,
$S_1 = S_0$_h & S_1_l,if C_1_l = 0 otherwise $S_1 = S_1$_h & S_1_l ,
$C_0 = C_1$_h if C_0_l = 1 otherwise $C_0 = C_0$_h and
$C_1 = C_0$_h if C_1_l = 0 otherwise $C_1 = C_1$_h.

Proof

First we find out what the bits i to i+j of S_0 should be. According the definition S_0 is the i to i+j+k bits of (A_h & A_l) + (B_h & B_l) under the assumption that the carry from bit 1 - 1 to bit i is 0. Note that the bits i to i+j of A_h & A_l and B_h & B_l are the same as that of A_l and B_l respectively , so the bits i to i+j of the conditional 0-sum of A_h & A_l are the same as that of A_l + B_l under the same assumption, i.e. S_0_l equals the the bits i to i+j of S_0.

Next we figure out the bits i+j+1 to i+j+k of S_0. If C_0_l = 1 , then from lemma1 we have C_1_l = 1, i.e. both C_0_l and C_1-l are 1. In this case according the definition of C_0_l and C_1_l , we know that whatever the carry from bit 1 - 1 to bit i is , the carry from bit i+j to bit i+j+1 is 1 , so the bit i+j+1 to bit i+j+k of S_0 should be the same as that of S_1_h, because S_1_h is the bit i+j+1 to j+j+k of the sum of A_h and B_h under the assumption that the carry from bit i+j to bit i+j+1 is 1. IF C_0_l = 0, the carry from bit i+j to bit i+j+1 is 0 while adding (A_h *A_l) and (B_h & B_l) , so the bit i+j+1 to bit i+j+k of ((A_h & A_l) + (B_h & B_l)) are the same as that of (A_h + B_h) under the assumption that the carry from bit i+j to bit i+j+1. In another word, the bit i+j+1 to bit i+j+k of S_0 equals that of S_0_h.

From the above we have
$S_0 = S_1$_h & S_0_l if C_0_l = 1 otherwise $S_0 = S_0$_h & S_0_l.

The rest the proof are similar to the above.

From this theorem we can develop a procedure to compute the conditional 0-sum S_0, conditional 1-sum S_1, conditional 0-carry C_0 and conditional 1-carry C_1 of two longer binary numbers A = A_h & A_l and B = B_h & B_l using the conditional 0-sum S_0_h, S_0_l, conditional 1-sum S_1_h, S_1_l, the conditional 0-carry C_0_h,C_0_l, conditional 1-carry C_1_h and C_1_l.

Procedure P(S_0_l,S_1_l,C_0_l,C_1_l,S_0_h,S_1_h,C_0_h,C_1_h,S_0,S_1,C_0,C_1)
begin
if C_0_l = 1 then S_0 := S_1_h & S_0_l else S_0:= S_0_h & S_0_l
if C_0_h = 0 then S_1 := S_0_h & S_1_l else S_1:= S_1_h & S_1_l
if C_0_l = 1 then C_0 := C_1_h else C_0 := C_0_h
if C_1_l = 0 then C_1 := C_0_h else C_1:= C_1_h
end

This procedure is used iterately in following algorithm.

CSA Algorithm

input: the summand A = A_n ... A_2 A_1
the addend B = B_n ... B_2 B_1
(for convenience provided that there exists m> 0,
such that $n = 2^m$)

output: the sum of A and B
(i.e. the 0-carry along with the 0-sum of A and B)

for i = 1 to n do
begin
S_0^0 (i):= $A_i \oplus B_i$
S_1^0 (i):= ~ ($A_i \oplus B_i$)
C_0^0 (i):= A_i /\ B_i
C_1^0 (i):= A_i \/ B_i
end.
for k= 0 to m -1 do
begin
for j = 1 to 2^{m-k-1} do
begin
Procedure P(S_0^k (2j-1),S_1^k (2j-1),C_0^k (2j-1),C_1^k (2j-1),
S_0^k (2j),S_1^k (2j),C_0^k (2j),C_1^k (2j),
S_0^{k+1} (j),S_1^{k+1} (j),C_0^{k+1} (j),C_1^{k+1} (j))
end
end

the sum of A and B is
A + B = (C_0^m (1)) & (S_0^m (1)).

3. The Design of CSA and BLC Adder

3.1. CSA Adder

The CSA algorithm described in section 2 performs additions of two n-bit (n = 2^m)binary numbers(addthings) by computing the conditional sums and carrys for each 1- bit piece of the two addthings first, then for each 2-bit piece, each 4-bit piece, ..., till n-bit piece . The conditional 0-sum along with the conditional 0-carry of n-bit addend and summand is the normal sum we want. To implement the algorithm fast, we should make full use of the advantage of parallel computation, so we use n identical circuits to compute the conditional sums and carrys for each bits. the circuit shown in Fig.1 is the CMOS implementation of those logic expressions in the CSA algorithm. We use G0 to denote this circuit.

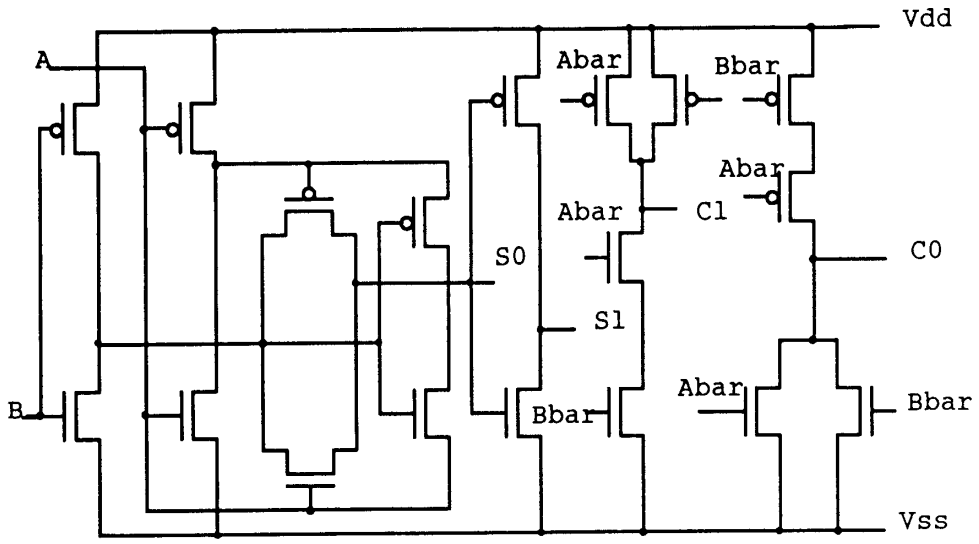

Fig.1. CMOS circuit for computing conditional sums and carrys of each bit

Once we have generated the conditional sums and carrys for each bit, all we have to do next is to pass some of them through or select some of them to output under the control of some inputs iteratively. What the procedure P does is to compute the conditional sums and carrys of 2^k-bit piece of the addthings using the conditional sums and carrys of 2^{k-1}-bit piece of the addthings.In procedure P, the input S_0_l and S_1_l are input data which are just passed to output; C_0_l and C_1_l are the input data which control the selection between input data S_0_h, S_1_h and C_0_h, C_1_h. The functional block diagram implemented P is shown in Fig.2. If we use PE to represent the hardware implementation of Procedure P, then the n-bit CSA adder can be considered as a tree which has n leaves nodes(Gs) and n-1 non-leave nodes (PEs), we call it CSA adder tree. Such a tree for 8-bit CSA adder is shown in Fig.4.

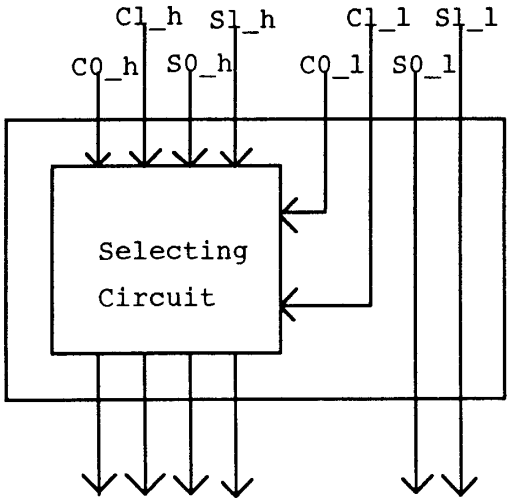

Fig.2. Block diagram of the implementation of Procedure P

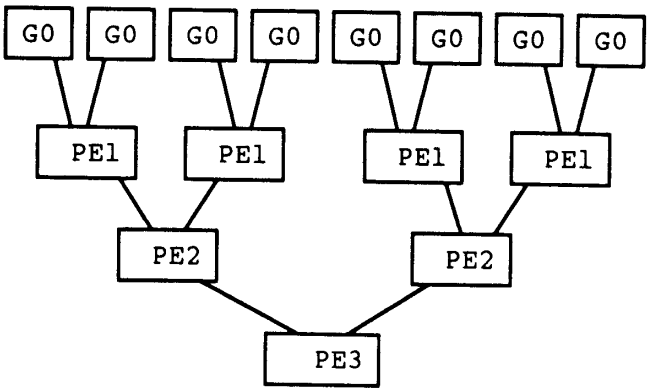

Fig.3. The adder tree for 8-bit CSA adder

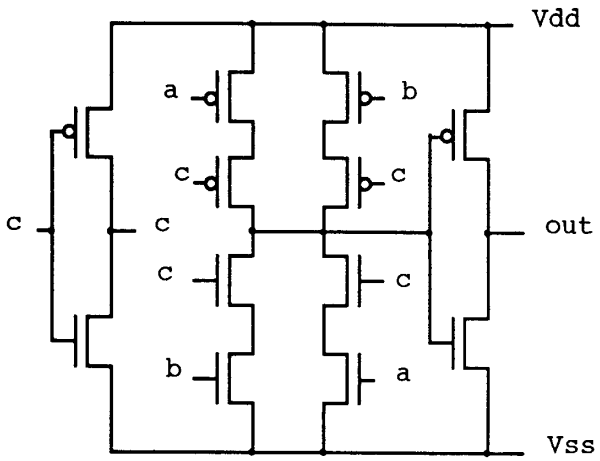

Fig.4. 2-input fast multiplexer

Unfortunately, only those PEs in the same level of the tree are exactly the same. the nodes located in different level however are similar in function and structure. The only unit circuit in PEs is 2-input multiplexer, so the speed of CSA adder mainly depends on the delay time of the multiplexers. Spice simulation tells us that the multiplexer which consists of transmission gates is slow. Furthermore, when the signals generated from cell G0 transmit toward output, The logic level will be degraded if no buffers are inserted into the path. Here we use the restoring logic multiplexers shown in Fig. 4 which acts as a strong buffer as well as a multiplexer. Some special considerations of its layout made it very fast (delay time 3.5ns).

PE1 consists of four 2-multiplexers, PE2 consists of six 2-multiplexers . In general PEn consists of $2^n + 2$ multiplexers. If the number of bit of the CSA is doubled, we have to design one new PE cell. because the delay time of cell PE is one 2-input multiplexer delay time, the CSA adder should use O(log n) time to perform n-bit addition. It has the same computational complexity as BLC adder, The floorplan for 4-bit CSA adder is shown in Fig.5.

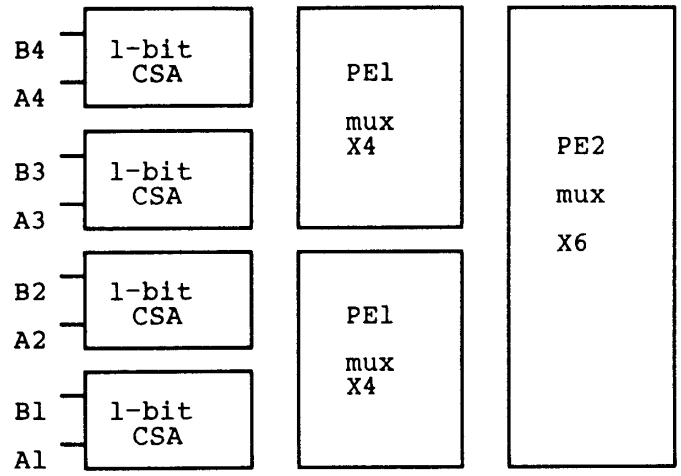

Fig.5 the floorplan of 4-bit CSA adder

3.2. BLC Adder

In order to compare the CSA adder with other fast adder, we design a BLC adder using the same CMOS technology. the design is similar to its NMOS counterpart described in [3].

We use two complementary black processors cell BA and BB(Fig.6(a) and (b)) which perform the operation "o" defined by Brent and Kung in [1] and two white processors cell WA and WB (Fig.6 (c) and (d)) which act as buffers and wire the black processor cells. The cell G0 which produces the generate term G_i and propagate term P_i from the input A_i and B_i is shown in Fig.7(a). The cell S0 which produces the sums ($S_i = P_i$ O C_{i-1} is shown in Fig.7(b).

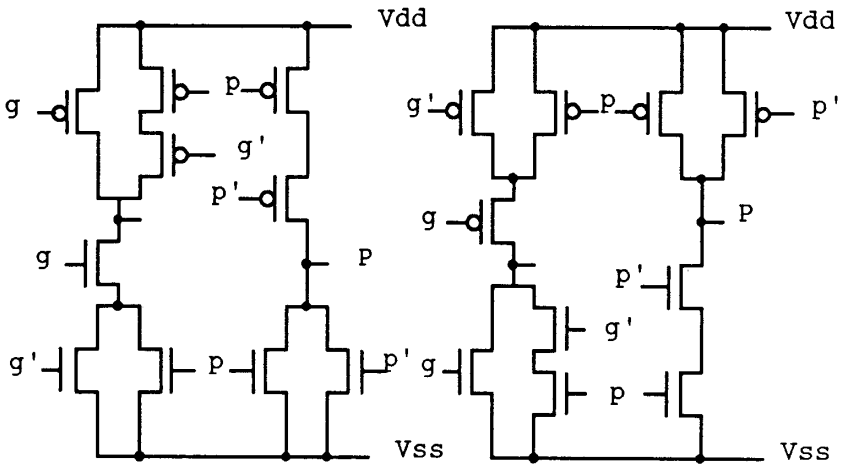

Fig.6(a) Cell BA

Fig.6(b) Cell BB

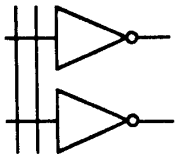

Fig.6(c) Cell WA

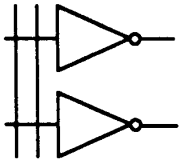

Fig.6(d) Cell WB

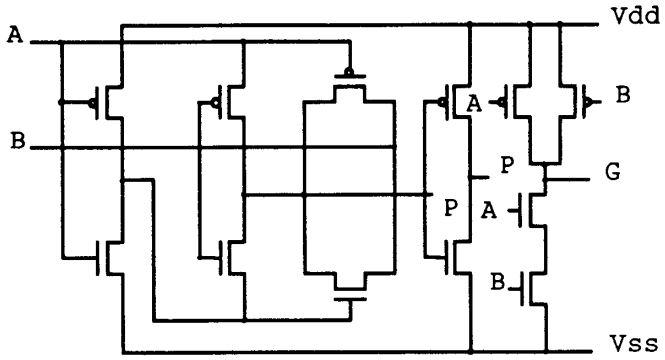

Fig.7(a) Cell G0

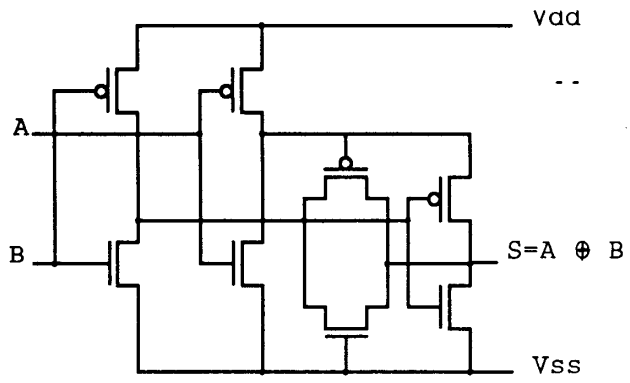

Fig.7(b) Cell S0

The floorplan for a 8-bit BLC adder is shown in Fig.8. It is fairly straightforward to expand the layout to any number of bits.

G0	BB	AB	BB	WB	WB	S0
G0	WB	WA	WA	WA	BB	S0
G0	BB	WB	WA	AB	WB	S0
G0	WB	WA	WA	WA	BB	S0
G0	BB	AB	WB	WB	WB	S0
G0	WB	WA	BB	WA	WA	S0
G0	BB	WB	WB	WA	WA	S0
G0	WB	WA	WA	WA	WA	S0

Fig.8 . Floor-plan for 8-bit BLC adder.

We have laid-out the CSA and BLC adder in static COMS. The area taken up by these adders along with the Spice simulation results are shown in table 1. We also compare them with the fast ripple adder which was designed for a signal processing chip by one of authors of this paper. We assume that the load capacitance is 0.1pF and that the highest logic "0" voltage level is 0.25 V, the lowest logic "1" voltage level is 4.5 V while counting the delay time of the Spice simulation results.

Area and Time Requirement						
No. of Bits	Different Adders					
	Ripple		BLC		CSA	
	size (λ)	delay (ns)	size (λ)	delay (ns)	size (λ)	delay (ns)
4	258X278	6	410X278	8	540X278	7
8	258X542	12	570X542	12	760X542	10
16	258X1070	24	730X1068	16	1028X1070	14
32	258X2122	48	890X2122	20	1350X2122	18
64	258X4258	96	1052X4258	24	1810X4258	22

In order to make the comparison of area easier, we use the same pitch height in the layout for all the three adders. It can be seen from table 1 that the CSA adder is faster than BLC adder, nevertheless It demands larger size of area. Notice that the ripple adder here is faster than general straightforward serial adder.

4. Formal verification of the CSA adder

We have done some Spice simulation of the circuit of CSA adder, but only for some typical inputs. By running circuit simulation (or other kind of simulation such as switch level and logic level simulation) we can predict the performance of the design and verify the correctness of the design but only for theses typical inputs. It is impossible to exhaust all acceptable inputs even for a 8-bit adder.

Hardware verification is a technique by which one can formally prove that the design meets a specification of its intended behavior.Instead of doing simulation for all possible inputs we give a formal verification to show that the design is correct for all the acceptable inputs.

Mike Gordon's HOL system [6] (a mechanization of Higher Order Logic) which can handle circuits, sub-systems and complete architectures is a powerful proof generating system. We use Higher Order Logic to specify the implementation and intended behavior of the CSA adder and prove that the implementation meets the specification of the behavior by the proof generating tools in HOL system.

We take the 8-bit CSA adder as an example to describe the idea of proof. From the 8-bit-csa adder tree in Fig.3,it can be seen that the 8-bit CSA adder consists of two 4-bit CSA adders and a processing unit PE3. As we said before, the role of PE3 is to select and transfer signals, so we take the type of word built in HOL system, which was used by Mike Gordon for proving a small computer in the level of register transfer, to represent the summand, addend and sums of the CSA adders here, and we use the type of bool to represent carry signal.

The intended behavior of a 8-bit CSA adder can be defined in Higher Order Logic by following predicate eight_bit_csa_spec, but at first, we define conditional 0-sum(s0), 1-sum(s1),0-carry(c0) and 1-carry(c1) of two 8-bit word w1 and w2 by following definitions, Note that the type of w1 and w2 in HOL are word8, represented by "(w1:word8) (w2:word8)", VAL8 is a constant in HOL and (VAL8 w1) is the value of w1.

```
let sum0_8 = new_definition
 ('sum0_8',"!(s0:word8)(w1:word8) (w2:word8).
       sum0_8 s0 w1 w2 =
     (VAL8 s0 = ((VAL8 w1 + VAL8 w2 ) < (2 EXP 8) ) =>
 (VAL8 w1 + VAL8 b) | ((VAL8 w1 +VAL8 w2 ) -(2 EXP 8)))");;

let sum1_8 = new_definition
 ('sum1_8',"!(s1:word8)(w1:word8)(w2:word8).sum1_8 s1 w1 w2 =
   (VAL8 s1 = (((VAL8 w1 + VAL8 w2 ) + 1 ) < (2 EXP 8) ) =>
     ((VAL8 w1 + VAL8 w2) +1) |
         (((VAL8 w1 +VAL8 w2) +1 ) -(2 EXP 8)))");;

let carry1_8 = new_definition
 ('carry1_8',"!(c1:bool)(w1:word8)(w2:word8).carry1_8 c1 w1 w2 =
   (c1 = (((VAL8 w1 + VAL8 w2 ) + 1 ) < (2 EXP 8) ) => F  | T)");;

let carry0_8 = new_definition
 ('carry0_8',"!(c0:bool)(w1:word8)(w2:word8).carry0_8 c0 w1 w2 =
     (c0 = ((VAL8 w1 + VAL8 b) < (2 EXP 8) ) => F  | T)");;
```

Now we define the specification of the behavior of 8-bit CSA adder:

```
let eight_bit_csa_spec = new_definition
  ('eight_bit_csa_spec',
          "!(w1:word8) (w2:word8)(s0:word8) (s1:word8)
            (c0:bool)(c1:bool).
          eight_bit_csa_spec  w1 w2 s0 s1 c0 c1  =
               sum0_8  s0 w1 w2 /\
               sum1_8 s1 w1 w2       /\
```

```
        carry0_8 c0 w1 w2 /\
        carry1_8 c1 w1 w2" );;
```

On the other hand We specify the implementation of the 8-bit CSA adder.

```
let eight_bit_csa_imp = new_definition
  ('eight_bit_csa_imp',
    "!(w1:word8)(w2:word8)(s0:word8)(s1:word8)(c0:bool)(c1:bool).
       eight_bit_csa_imp w1 w2 s0 s1 c0 c1 =
     ?s0_l s1_l c0_l c1_l s0_h s1_h c0_h c1_h.
     (four_bit_csa_imp (low_field4 w1) (low_field4 w2)
      s0_l s1_l c0_l c1_l) /\
     (four_bit_csa_imp (high_field4 w1) (high_field4 w2)
      s0_h s1_h c0_h c1_h) /\
     (PE3_imp s0_l s1_l c0_l c1_l s0_h s1_h c0_h c1_h s0 s1 c0 c1)
  ");;
```

Here "low_field4 w1" was defined to be lower 4-bit of w1, and "high_field w1" was defined to be the higher 4-bit of w1.

We can define the implementation and specification for 2-bit and 4-bit CSA adder in the same way.

We have proved following theorem:

```
|- !w1 w2.eight_bit_csa_imp w1 w2 s0 s1 c0 c1 ==>
                    eight_bit_csa_spec w1 w2 s0 s1 c0 c1
```

which means the implementation of 8-bit CSA adder meets the specification of its intended behavior.

The idea of the proof of above theorem is roughly similar to the proof of the algorithm in section 2 but involved much more details of arithmetics we have to prove a bunch of theorems about arithmetics by forward or tactics proofs before starting to prove above theorem.

The proof for 2-bit, 4-bit and 8-bit CSA adder are basically the same, but we have to go down the transistor level and use the bidirectional model of transistor for the proof of 1-bit CSA adder and processing unit PEs, though the proof is straightforward. Here we take the proof of correctness of the fast multiplexer shown in Fig.4. as a example. The specification of the intended behavior of the multiplexer:

```
let mux1_spec  = new_definition
  ('mux1_spec ',"!(a:bool)(b:bool)(c:bool) (o:bool).
            mux1_spec  a b  c o = (o = (c => b | a))");;
```

The implementation of the multiplexer :

```
let mux1_imp = new_definition
  ('mux1_imp',"!a b c o.
    mux1_imp ( a:bool)( b:bool) (c:bool)(o:bool) =
      ?p1 p2 p3 p4 p5 p6 p7 p8.
              inv_imp c p1 /\
              inv_imp p2 o /\
              pwr_spec p7 /\
              gnd_spec p8 /\
              ptran_spec a p7 p3 /\
              ptran_spec b p7 p4 /\
              ptran_spec c p3 p2 /\
              ptran_spec p1 p4 p2 /\
              ntran_spec a p8 p5 /\
              ntran_spec b p8 p6 /\
              ntran_spec p1 p5 p2 /\
              ntran_spec c p6 p2 );;
```

Here we take the transistor, power and ground as primitives, the correctness of inverter in the circuit was proved beforehand.

The correctness of the implementation of the multiplexer is given by the theorem

```
|- !a b c o.mux1_imp a b c o ==> mux1_spec a b c o
```

The following HOL script proves and stores the above theorem.

```
let STRIP_EXISTS_TAC =
        DISCH_THEN(REPEAT_TCL CHOOSE_THEN MP_TAC);;

let mux1_imp_correct = prove_thm
 ('mux1_imp_correct',
  "!a b c o.mux1_imp a b c o ==> mux1_spec a b c o",
  REPEAT GEN_TAC THEN
  REWRITE_TAC [mux1_imp;mux1_spec;inv_correct;pwr_spec;
   gnd_spec;ptran_spec;ntran_spec;inv_spec] THEN
  BOOL_CASES_TAC "a" THEN BOOL_CASES_TAC "b" THEN
  BOOL_CASES_TAC "c" THEN REWRITE_TAC[]THEN STRIP_TAC THEN
  FIRST [ ASM_REWRITE_TAC []THEN
  EVERY_ASSUM (\ th.SUBST1_TAC (SYM th) ? ALL_TAC) THEN
  FIRST [ACCEPT_TAC (ASSUME "~ p8")
      ; ACCEPT_TAC (ASSUME "p7")]
  ;
  ASM_REWRITE_TAC []THEN RES_TAC THEN
  EVERY_ASSUM (\ th.SUBST1_TAC (SYM th) ? ALL_TAC) THEN
  FIRST [ ACCEPT_TAC (ASSUME "~ p8")
      ; ACCEPT_TAC (ASSUME "p7")]]);;
```

5. Conclusion

We have formulated the CSA algorithm, and suggested a design of the CSA adder in CMOS. The layout of CSA and BLC adders as well as the Spice simulation results have shown that the CSA adder described in this paper is area-time efficient, It can perform n-bit addition in O(log n) time, but demands larger size of area. Its layout is regular but needs more wiring efforts than BLC layout. Furthermore we have formally proved that the implementation of the CSA adder is correct by using HOL proof generating system. We feel that the work we have done is a meaningful experiment for implementing and proving a device or subsystem by available CAD tools .

6. References

[1]. Brent,R.P.,and Kung,H.T. "A regular layout for parallel adders". IEEE Trans.comput. C-31(Mar.1982),pp.260-264.

[2]. Weste,N. and Eshraghan,K. Chapter 8 in "PRINCIPLES OF CMOS VLSI DESIGN" ADDISON-WESLEY PUBLISHING COMPANY 1985, pp.326-331.

[3]. Brent, R.P. and Ewin, R.R."Design of an NMOS parallel adder" TR-CS-82-06 , Department of Computer Science, The Australian National University,1982.

[4]. Sklansky, J. "Conditional sum addition logic" IRE Trans. Electron.Comput. Vol.EC-9 (June 1960), pp.226-231.

[5]. Ware,F.A.,Mcallister,W.H.,Carlson,D.S., and Vlach,R.J., "64 bit monolithic floating point processors", IEEE,J.of solid-state circuits, Vol.SC-17, No.5,Oct 1982, pp.898-907.

[6]. Gordon.M. "HOL:A Proof Generating System for Higher-Order Logic" in VLSI Specification,Verification and Synthesis,Ed.G.Birtwistle and Subrahmanyam, Kluwer 1987.PP.73-128.