CMU Fall'01 18-760 VLSI CAD

[120 pts] **(V2)** **Homework 2.** **Out Thu Sep 13, Due Thu Sep 27 '01.**

---

1. BDD ordering [10 pts]

We saw that variable order is highly significant for something as simple as a multiplexor. How about something like a *comparator*? A simple comparator takes two 2-bit unsigned binary numbers a1a0 and b1b0 and compares their magnitude, and sets the output z=1 just if a1a0 is **less than or equal** to b1b0. Do this:

- Is there a particularly *good* variable ordering for this function? **Show** it--show the BDD. Is there a particularly *bad* variable ordering for this function? **Show** it--show the BDD.

- **Draw** a gate-level netlist using any AND, OR, NOT, EXOR gates you want, to implement the simple comparator from the previous problem. Apply Minato's ordering heuristic (and where you need to break ties or make any arbitrary ordering decicion--just **tell** us what you did and **show** the work). **Show** what variable ordering it produces.

2. ITE for Gates [10 pts]

What is the **fewest** number of calls you need to make to ITE to implement $a \oplus b$? In other words, you don't want to use ITE several times to build AND, OR, and NOT, to do $a'b + ab'$ —that's too easy. You can do it **much** more simply if you think about it Draw the multiplexor-hardware picture of the ITE and label clearly what's going in and out of each ITE.

3. ITE Decomposition [10 pts]

Using what you know of Boolean algebra, the definition of ITE, and the properties of cofactors, show that the ITE decomposition below (from class) actually works:

$$\text{ITE}(I,T,E) = x \bullet \text{ITE}(I_x,T_x,E_x) + \bar{x} \bullet \text{ITE}(I_{\bar{x}},T_{\bar{x}},E_{\bar{x}}) \qquad \text{(EQ 1)}$$

4. ITE Recursion [10 pts]

Let $f(x, y) = x \bullet y$ and $g(x, y) = x \overline{\oplus} y$ (this is *exclusive-nor*). Assume we have a multi-rooted DAG for the BDDs representing these two functions (you need to draw them.) We want to EXOR these functions, and compute a new function $Q(x, y) = (f \oplus g)(x,y)$. Using the ITE operator as discussed in the notes, show how you would implement this EXOR operation. As in the slides, show how the recursive computation proceeds as ITE calls itself. At each node of the recursive call tree, tell what ITE is computing (label the nodes in your BDD in some sensible way) and show clearly when each recursive call terminates. Draw the final recursive call tree.

Draw the final result, i.e., show the final form of the multi-rooted DAG that now represents functions *f*, *g*, and *Q*.

5. Derived Operators [20 pts]

Suppose we have a software package that has data structures representing variables and Boolean functions as BDDs, and that the following operations are available as subroutines in this software. (We will write these in a simplified sort of C language notation):

bdd **var2func**(*var* x)   Generate the BDD corresponding to a single variable x.
Input is a single variable (of type *var*), and
the returned output is a BDD.

bdd **ITE**(*bdd* I, *bdd* T, *bdd* E)

Compute the if-then-else operation. Inputs are 3 BDDs,
called I (*if* part), T (*then* part) E (*else* part),
and the returned output is another BDD.

int **iszero**(*bdd* func)   Returns integer 1 just if func is the always-zero function.
Input is a BDD, output is integer 1 or 0 (it's *not* a BDD)

bdd **cofactor**(*bdd* func, *var* x , *int* val)

Computes cofactor of func with respect to *var* x, setting
x = val. func is a BDD, var is a variable,
val is integer 0 or 1

bdd **AND**(*bdd* f, *bdd* g)
bdd **OR**(*bdd* f, *bdd* g)
bdd **EXOR**(*bdd* f, *bdd* g)
bdd **NOT**(*bdd* f)

Compute the basic logical (gate type) operations
on BDDs. AND, OR and EXOR create new BDDs
representing the logical AND, OR and exclusive-or
of their inputs. NOT creates the BDD for the complement
of its input.

bdd CONST1, CONST0

You can assume these two BDDs are already defined.
These are just the constant 1 function and the
constant 0 function. Note that **iszero**(CONST0) == (int) 1.

No other operations are implemented, and there is no way for you to examine the BDD data structure directly.

Describe (in C-like pseudo-code notation–we *don't* need real code here!) how you would implement the following operations:

- *int* **depends**(*bdd* f, *var* x) Determine whether function f depends on the specified variable x. This means if you change x, at least sometime the output of f will also change. f is a BDD, x is a variable, **depends** returns integer 0 or 1.

- *bdd* **univquant**(*bdd* f, *var* x)
  Compute universal quantification of function f with respect to variable x. f is a BDD, x is a variable, **univquant** returns a BDD.

- *int* **opposite**(*bdd* f, *bdd* g) Determine whether two functions are complementary, i.e., if one of them is the complement of the other. f and g are BDDs, **opposite** returns integer 0 or 1.

- *bdd* **exchange**(*bdd* f, *var* a, *var* b)
  Exchange roles of specified variables in function f. For example, **exchange**(a•b + d, a, d) —> d•b + a. f is a BDD, a and b are variables, **exchange** returns a BDD.

- *bdd* **compose**(*bdd* f, *bdd* g, *var* x)
  Creates a new function (*composition* function) with variable x in f set to the output of g. The picture below clarifies what we are computing. f and g are BDDs, x is a variable, and **compose** returns a BDD.
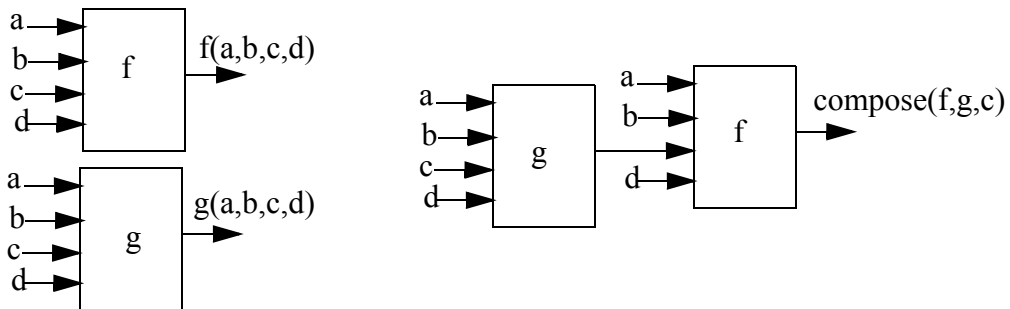
HINTS: lots of things that look difficult are easier when you **cofactor** them and look at the cofactors. Play around with ITE of various combinations of the cofactors. The first 3 operators are pretty straightforward, the last 2 are rather tricky. *None* of these things requires some sophisticated recursive algorithm, just a few lines of calls to the right operators with the right inputs. To emphasize this, here is the answer for the first part, for **depends**(*bdd* f, *var* x):

```
int depends(bdd f, var x) {
    return(1 - iszero(  EXOR( cofactor(f, x, 0), cofactor(f, x, 1) )  );

}
```

Notice how this works. If function f( ) depends on variable x, then if I change x from x=0 to x=1, there ought to be at least some pattern for the remaining inputs that makes the output of the function *change*. But this is exactly what the Boolean Difference tries to compute. So, we compute the BDD for a new function f(... x=0 ...) $\oplus$ f(... x=1 ...) using calls to **cofactor** and to **EXOR**. What does this new function tell us? If the new function is zero always, for all inputs, then you cannot affect the output of function f by changing variable x. In other words, f does not depend on variable x. So if the **iszero**( ) function returns integer 1, it means the original f function does *not* depend on x. To get the true/false return for **depends**() correct, we have to invert this, which is what the (1 - ...) does.

6. Combinational Verification in KBDD [30 pts]

kbdd is a BDD calculator done by Prof. Randy Bryant's research group that has all the operators you'd want to use to manipulate Boolean functions, and a simple command line interface to type in functions, etc. You will use this to try to verify the correctness of a logic gate network whose BDDs are much too big to do by hand.

kbdd lives in /afs/ece/class/ee760/bin/kbdd, and it works on IBM AIX boxes and on SUN Solaris boxes.

As a starting point, the following page shows a complete trace of a session with kbdd, using it to do the network repair problem on the last homework assignment. Inputs are in normal font, outputs *italics*, kbdd's prompts for input shown in bold as **KBDD**:

% /afs/ece/class/ee760/bin/kbdd

KBDD: # input variables

KBDD: boolean a b cin d0 d1 d2 d3

KBDD: #

KBDD: # define the correct equation for the adder's carry out

KBDD: eval cout a&b + (a+b)&cin

cout: a&b + (a+b)&cin

KBDD: #

KBDD: # define the incorrect version of this equation (just for fun)

KBDD: eval wrong a&b + (!(a&b))&cin

wrong: a&b + (!(a&b))&cin

KBDD: #

KBDD: # define the to-be-repaired version with the MUX

KBDD: eval repair a&b + (d0&!a&!b + d1&!a&b + d2&a&!b + d3&a&b)&cin

repair: a&b + (d0&!a&!b + d1&!a&b + d2&a&!b + d3&a&b)&cin

KBDD: #

KBDD: # make the Z function that compares the right version of

KBDD: # the network and the version with the MUX replacing the

KBDD: # suspect gate (this is EXNOR of cout and repair functions)

KBDD: eval Z repair&cout + !repair&!cout

Z: repair&cout + !repair&!cout

KBDD: # universally quantify away the non-mux vars: a b cin

KBDD: quantify u ForallZ Z a b cin

KBDD: #

KBDD: # let's ask kbdd to show an equation for this quantified function

KBDD: sop ForallZ

  !d0 & d1 & d2

KBDD: #

KBDD: # what values of the d's make this function == 1?

KBDD: satisfy ForallZ

Variables: d0 d1 d2

011

KBDD: #

KBDD: # that's it!

KBDD: quit

%

KBDD Quick Reference

boolean *var* …	Declare variables and variable ordering
Extended naming	
var[*m .. n*]	Numeric range (ascending or descending)
{*s1,s2,…*}	Enumeration
evaluate *dest expr*	*dest* := bdd for boolean expression *expr*
Operations	(decreasing precedence)
!	Complement
^	Exclusive-Or
&	And
+	Or
bdd *funct*	Print BDD DAG as lisp-like representation
sop *funct*	Print sum-of-products representation of *funct*
satisfy *funct*	Print all satisfying variable assignments of *funct*
verify *f1 f2*	Verify that two functions *f1 f2* are equivalent
size *funct* …	Compute total BDD nodes for set of functions
replace *dest funct var replace*	Functional composition: *dest* := *funct* with variable *var* replaced by *replace* function output
quantify [**u**|**e**] *dest funct var* …	*dest* := Quantification of function *funct* over variables *var* ...
e	Existential quantification is done
u	Universal quantification is done
adder *n Cout Sums As Bs* Cin	Compute functions for n -bit adder
n	Word size
Cout	Carry output
Sums	Destinations for sum outputs: *Sum.n … Sum.0*
As	A inputs: *A.n-1 ... A.2 A.1 A.0*
Bs	B inputs: *B.n-1 … B.2 B.1 B.0*
Cin	Carry input
alu181 *Cout Fs M Ss Cin As Bs*	Compute functions for '181 TTL ALU
Cout	Destination for carry output
Fs	Destinations for function outputs: *F.3 F.2 F.1 F.0*
M	Mode input
Ss	Operation inputs: *S.3 S.2 S.1 S.0*
Cin	Carry input function
As	A inputs: *A.3 A.2 A.1 A.0*
Bs	B inputs: *B.3 B.2 B.1 B.0*
mux *n Out Sels Ins*	Compute functions for 2n-bit multiplexor
n	Word size
Out	Destination for output function
Sels	Control inputs: *Sel.n-1 ... Sel.1 Sel.0*
Ins	Data inputs: *In.2n – 1 … In.1 In.0*
quit	Exit KBDD

So, what shall we use *kbdd* to verify? It turns out that *adders* are a very good choice here, because they come in so many different styles, and we know that the BDD for a basic adder is very simple and small. Look at the appendix to this assignment for a description of a very interesting industrial-strength adder, from Grant McFarland's webpage at Stanford (http://umunhum.stanford.edu/~farland/notes.html). This adder has these features:

• 32-bit carry lookahead structure (CLA)

• Faster carry propagation through the use a so-called Ling adder structure

• Conditional carry propagation using a carry select multiplexor

This is a pretty **scary** looking adder-- but these are the kinds of tricks people really play to implement very fast adders for things like microprocessors. And, this is exactly what a "real world" industrial design description looks like.

Question: Does this thing really work? The description in the appendix is pretty detailed--but, hey, mistakes happen. To check this, but to keep the complexity down, we want you to verify **2** outputs from this complicated adder: the final carry out **cout** and the most significant bit of the sum **s31.** You don't need to verify any of the other bits.

This means, we want you to build the BDDs for all these two functions of the input bits (**a0 - a31**, **b0 - b31**) and compare them to an "ordinary" adder. Are they identically the same Boolean functions? Use *kbdd* to formally verify these 2 parts of this aggressive, custom 32-bit design. Include a listing of your session with *kbdd* showing how you did this. Note especially that *kbdd* has basic n-bit adders built in, so to create the "baseline" sum-bit equations is a simple operation. For example, to do a basic 4-bit adder, this will suffice:

kbdd: *boolean a[3..0] b[3..0] s[3..0] cout cin*

kbdd: *adder 4 cout s[3..0] a[3..0] b[3..0] cin*

All the work is in *creating* the description adder in *kbdd*, and then comparing it appropriately.

Note: It's probably easiest to edit a script file that you then run through kbdd using the *source kbdd* command. In UNIX, if you type the following italics stuff:

% *script*

% */afs/ece/class/ee760/bin/kbdd*

kbdd: *source myfilename*

kbdd: *quit*

<you hit control-D>

then it will (1) start saving everything in a file called *typescript*, (2) run *kbdd*, (3) tell *kbdd* to run your commands in your file *myfilename.*   You type control-D to stop the script saving. You can then print this *typescript* file and hand it in, and include some comments so when we read it, we understand what you did.

7. Multi-Terminal BDDs [10 pts]

BDDs come in a number of specialized variants, one of which is the Multi-Terminal BDD, or MTBDD.

MTBDDs are a generalization of BDDs that allow an arbitrary number of real-valued terminals instead of just the basic binary terminals, 0 and 1. While a BDD represents function that returns a Boolean value for any assignment to its variables, an MTBDD returns a real number. More formally:

$$\text{BDDs} \qquad F : B^n \to B$$

$$\text{MTBDDs} \quad F : B^n \to R$$

Suppose you wanted to represent the following function:

$F(a,b) = 1.2$ for a=1, b=0
$F(a,b) = 2.5$ for a=0
$F(a,b) = 4.7$ for a=1, b=1

These numbers could have any number of meanings. One example would be the power in mW consumed by a particular circuit when it's input changes from "a" to "b". Or it could be the delay in ns, or the rise time of the output. The MTBDD for this function would be the following:
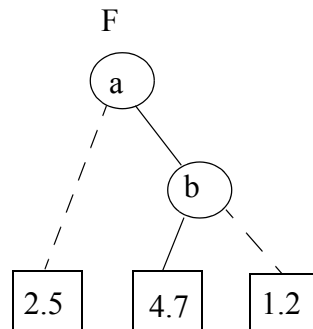

Furthermore, MTBDDs can be used to represent real-valued matrices fairly efficiently. To do so, we introduce log(# rows) + log(# columns) variables to encode the row position, and column position. For a simple 2 by 2 matrix, we get the following:
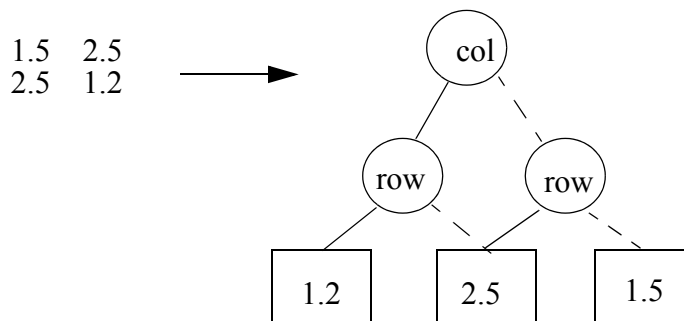
For very large matrices with lots of repeated numbers, the MTBDD representation can be considerably smaller than a simple listing of the values, and operations like matrix-addition or matrix-multiplication are correspondingly faster.

It turns out that working with MTBDDs is just as easy as with BDDs. To apply an arbitrary binary operator (function with two operands, like "a+b"), we can use the same expansion and cofactoring rules as with BDDs. The cofactor of a function F with respect to variable x, or F_x, is obtained by redrawing the MTBDD without all of the "x" nodes, and redirecting their incoming edges to the "hi" son. For example:
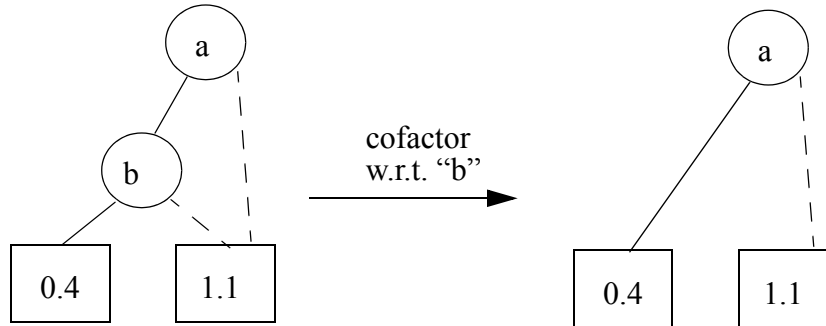
Using cofactors, we can decompose operations on MTBDDs in exactly the same manner as for BDDs:

$$F = x\, F_x + x'F_{x'}$$

$$F+G = x\, (\, F_x + G_x) + x'\, (F_{x'} + G_{x'})$$

$$F*G = x\, (\, F_x * G_x) + x'\, (F_{x'} * G_{x'})$$

This results in a nice recursive algorithm for performing arbitrary operations on two input MTBDDs that looks remarkably similar to algorithms for BDDs.

Do this:

1.) Draw the MTBDDs for the two following functions:

F(a,b) =   3 for a=0
           0 for a=1,b=0
           4 for a=1,b=1

G(a,b) =   0 for b=1
           4 for b=0

2.) Compute and draw the resulting MTBDDs for H=F+G and M=F*G

(**Hint**: This will be easier if you think about what the results should be for each of the assignments to a and b first, and then try drawing the MTBDD.)

8. Metaproducts [20 pts]

A wonderful property of BDDs is that they only represent the "abstract" Boolean function, *not* the way you chose to implement it with logic gates. This is why BDDs are so useful for verification: you can implement **x + x'** , or **x + y + x'y'**, or just plain "**1**", and in all these cases, you get the identical BDD.

Of course, sometimes we actually *want* to represent the *way* we have implemented something as logic gates. Suppose we really want to represent--for whatever odd reason--the SOP expression = **x + x'**. Is there any way we can represent this directly, without immediately resolving it to the "1" function? In other words, can we preserve the SOP product structure of this expression? The answer is "yes" -- sort of.

We need a new representation of the function that "records" the SOP structure, but which also behaves as much like a BDD as possible. It turns out there is a very elegant trick for recasting the original function in a new set of variables, and then just representing this new function as a BDD, that does much of what we want it to do. This new "SOP preserving" structure is called **metaproduct notation**. (The idea is due to Olivier Coudert, originally of Bull Research Center in France.) Here's the trick:

- For each variable **x** in your SOP form, the metaproduct formula has 2 different variables: r_x and s_x. r_x is the *occurence* variable for **x**; s_x is the *sign* variable for **x**.

- Suppose your function is **f(x,y,z,w)**. For each product term in your SOP form, for example **xy'z'**, you get a corresponding metaproduct term.
  If your literal is in positive form, like **x**, you get $(r_x s_x)$ in the metaproduct.
  If your literal is in negative form, like **x'**, you get $(r_x s_x')$ in the metaproduct.
  If a variable is missing from the product, like **w**, you get (r_w') in the metaproduct.
  (It turns out the s_w variable doesn't matter in this case, since **w** is not present, sign doesn't matter)

- So, **xy'z'** would get transformed into **($r_x s_x r_y s_y' r_z s_z' r_w'$)** in metaproduct form.

Read $r_x = \mathbf{1}$ as meaning "the variable **x** occurs in the product." Read $r_x = \mathbf{0}$ as meaning "the variable **x** does *not* occur in the product." Similarly, read $s_x = 0$ as "the polarity of x is positive" and $s_x = 1$ as "the polarity of x is negative". For example:
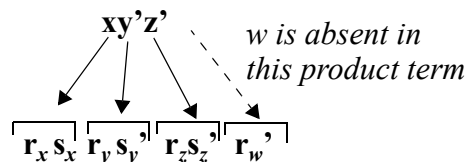
So, for example, if we actually tried to represent **f(x)=(x + x')** we would get ($r_x s_x + r_x s_x'$) for the metaproduct form. To manipulate this, we represent it as a BDD. we get one more rule here:

- Interpret the **paths** from the BDD root to the "1" leaf as specifying the individual product terms in the metaproduct form. If a variable is omitted on a path, you need to include it in **both** polarities in the final metaproduct.

This sounds more complicated than it really is. Let's try it. Do this:

- Draw the BDD for the metaproduct for the single-variable function **f(x)=x+x'**. Show how walking the paths from root to "1" leaf generates the correct metaproduct for this SOP form.

- Using what you know about BDDs, *complement* this BDD for this metaproduct. Again, walk the paths from root to "1" and generate the new metaproduct for f'(x). Interpret the result -- does this makes sense? Why?

- Draw the BDD for the metaproduct of the 4 variable function:
$$f(x,y,z,w) = yw' + xzw' + xy'zw'$$
Again, show how the paths from root to leaf in this BDD generate the correct metaproduct. (It's OK to use kbdd for this one, as long as you can figure out the BDD structure yourself.)

- Again, complement this BDD, and show that the result makes sense as a metaproduct. (For this one, you might want to use kbdd -- if it's too messy to do by hand.)

CLA and Ling Adders

1   Introduction

One of the most popular designs for fast integer adders are Carry-Look-Ahead adders. Rather than waiting for carry signals to ripple from the least significant bit to the most significant bit, CLA adders divided the inputs into groups of r bits and implement the logic equations to determine if each group will generate or propagate a carry. By combining the generate and propagate signals of r groups at with each successive stage of logic, a CLA adder can derive the carrys into each bit in order $\log_r n$ gates instead of order n for a ripple carry adder. This paper discusses the design of a very simple 32 bit CLA adder, some improvements that can be made to that adder, and a variation of CLA adders known as Ling adders.

2   A Simple CLA Adder

An overview of the adder's 4 stages is shown in figure 1 with stage 1 and the top and stage 4 at the bottom. In stage 1 the local generate and propagate signals for each bit are created. In stage 2 these signals are combined to create generate and propagate signals out of each group of 3 bits. In stage 3 the group signals are combined into 9 bit block signals. In stage 4 the carry into each block is calculated and these signals begin traveling back up the adder tree. In stage 3 the carry into each group is created, and in stage 2 the carry into each bit is created. Finally, stage 1 uses the local carry signals to calculate the final sum bits.
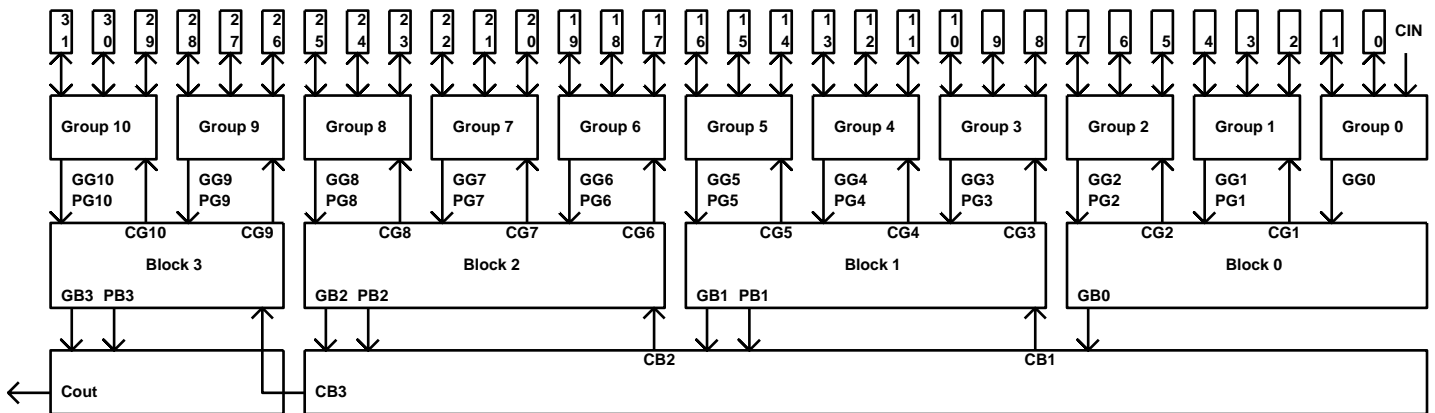

Figure 1: CLA Adder

2.1 Generate and Propagate Signals

In the first stage of logic the adder must calculate the local generate and propagate signals (g_i and p_i) which tell if each bit will generate a carry into the next bit or propagate a carry from the previous bit.

$$g_i = a_i b_i \tag{1}$$
$$p_i = a_i + b_i \tag{2}$$

In stage 2 these signals are then combined into group generates and propagates (GG_i and PG_i) for each of the ten groups as follows:

$$GG_0 = g_1 + p_1(g_0 + p_0 c_{IN}) \tag{3}$$
$$GG_1 = g_4 + p_1(g_3 + p_3 g_2) \tag{4}$$
$$\vdots$$
$$GG_{10} = g_{31} + p_{31}(g_{30} + p_{30} g_{29}) \tag{5}$$

$$PG_1 = p_4 p_3 p_2 \tag{6}$$
$$PG_2 = p_7 p_6 p_5 \tag{7}$$
$$\vdots$$
$$PG_{10} = p_{31} p_{30} p_{29} \tag{8}$$

where c_{IN} is the carry in signal to the least significant bit. Since c_{IN} is included in GG_0, no group propagate signal from group 0 is needed. The group propagate signals are formed with a simple 3 input AND gate. The group generate signals are formed with the fanin-3 generate gate shown in figure 2. In stage 3 these signals are used to create the block generate and propagate signals (GB_i and PB_i).

$$GB_0 = GG_2 + PG_2(GG_1 + PG_1 GG_0) \tag{9}$$
$$GB_1 = GG_5 + PG_5(GG_4 + PG_4 GG_3) \tag{10}$$
$$GB_2 = GG_8 + PG_8(GG_7 + PG_7 GG_6) \tag{11}$$
$$GB_3 = GG_{10} + PG_{10} GG_9 \tag{12}$$

$$PB_1 = PG_5 PG_4 PG_3 \tag{13}$$
$$PB_2 = PG_8 PG_7 PG_6 \tag{14}$$
$$PB_3 = PG_{10} PG_9 \tag{15}$$

All the blocks can use the same fanin-3 generate gate and 3 input AND gate used in the previous stage except for block 3 which contains only two groups. Its propagate signal requires only a 2 input OR, and its generate is create using a fanin-2 generate gate shown in figure 3. Having created the block generate and propagate signals, the adder begins to finally create the true carry signals.
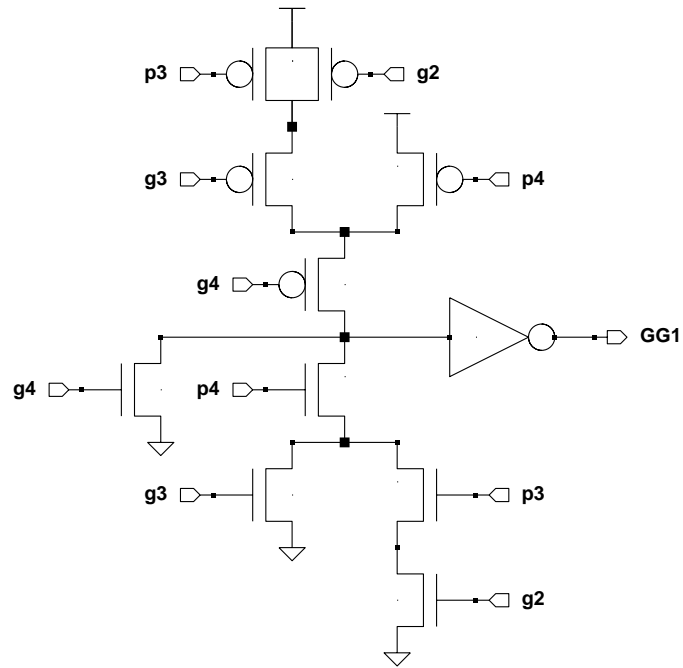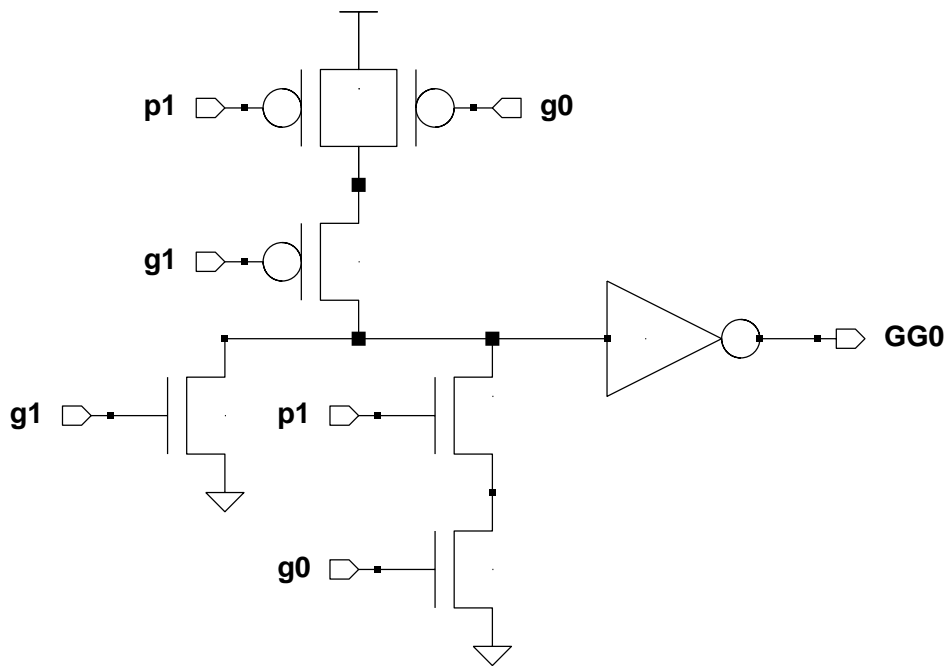
Figure 2: FanIn-3 Generate Gate

Figure 3: FanIn-2 Generate Gate

2.2   Carry Signals

In stage 4 the block generate and propagate signals are used to create the carry signals into each block (CB_i).

$$
\begin{aligned}
CB_1 &= GB_0 & (16) \\
CB_2 &= GB_1 + PB_1 GB_0 & (17) \\
CB_3 &= GB_2 + PB_2(GB_1 + PB_1 GB_0) & (18) \\
C_{OUT} &= GB_3 + PB_3 CB_3 & (19)
\end{aligned}
$$

where C_{OUT} is the overflow carry out of the entire adder. These signals then begin to travel back up the adder stages, first forming the carry into each group (CG_i) in stage 3. For block 2 these equations are as follows:

$$
\begin{aligned}
CG_6 &= CB_2 & (20) \\
CG_7 &= GG_6 + PG_6 CB_2 & (21) \\
CG_8 &= GG_7 + PG_7(GG_6 + PG_6 CB_2) & (22)
\end{aligned}
$$

In stage 2 these group carry signals are used to form the local carry into each bit (c_i). For group 8 these equations are as follows:

$$
\begin{aligned}
c_{23} &= CG_8 & (23) \\
c_{24} &= g_{23} + p_{23} CG_8 & (24) \\
c_{25} &= g_{24} + p_{24}(g_{23} + p_{23} CG_8) & (25)
\end{aligned}
$$

All of these signals can be created using the fanin-3 and fanin-2 generate gates shown in figures 2 and 3. This means each center group and block will use one fanin-3 gate and one OR gate to create generate and propagate signals for the stage below, and one fanin-3 gate and one fanin-2 gate to create carry signals for the stage above. The wiring of these groups and blocks is shown in figure 4 for group 1. hen the local carry signals reach stage 1, they are used to create the final sum bits (s_i) according to the equations:

$$
\begin{aligned}
t_i &= a_i \oplus b_i & (26) \\
s_i &= t_i \oplus c_i & (27)
\end{aligned}
$$

2.3   Critical Path

The worst case inputs for this adder are when $a_i \oplus b_i = 1$ for all the input bits and then c_{IN} is toggled. The local generate signals require 3 series transistors to form. For an N bit CLA adder combining r groups at each level, the generate signals must travel up $\lceil \log_r N \rceil - 1$ levels of $r + 1$ series transistors each. Then the signal travels down $\lceil \log_r N \rceil - 2$ levels of no more than $r + 1$
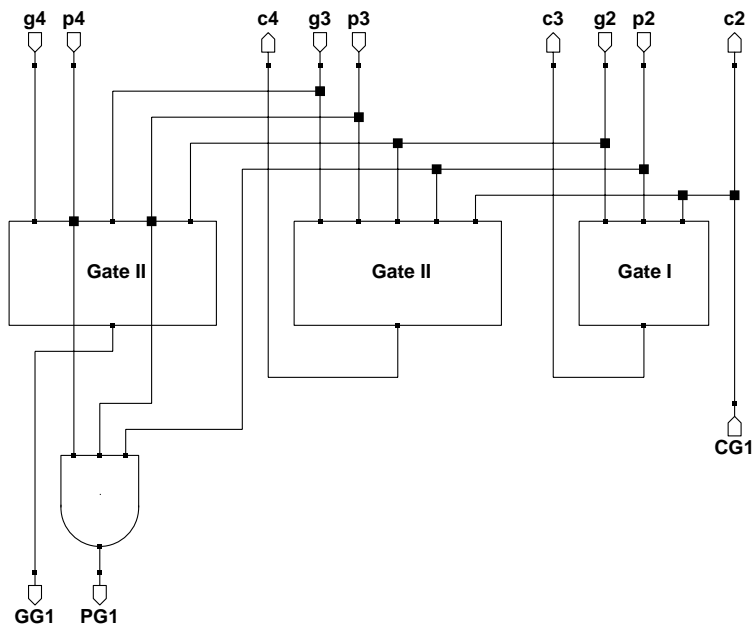
Figure 4: Group 1

series transistors. Final, the XOR to form the local sums takes 2 series transistors. Therefore, the maximum number of series transistors in the critical path can be written as:

$$T_d \quad = \quad 3 + (\lceil \log_r N \rceil - 1)(r+1) + (\lceil \log_r N \rceil - 2)(r+1) + 2 \qquad (28)$$

$$T_d \quad = \quad (2 \lceil \log_r N \rceil - 3)(r+1) + 5 \qquad (29)$$

For a 32 bit adder with $r = 3$ as described in this paper this equation gives a maximum of 25 transistors. The true critical path is 24 transistors since block 3 contains only 2 groups instead of 3. The critical path is shown in table 1. Although faster designs are possible, this adder has the

Operation	Signal	Delay	Total
Local Generate	g_i	3	3
Group Generate	GG_i	4	7
Block Generate	GB_i	4	11
Block Carry	CB_3	4	15
Group Carry	GG_{10}	3	18
Local Carry	c_{31}	4	22
Local Sum	s_{31}	2	24

Table 1: Simple CLA Critical Path

advantage of a relatively simple layout and wiring. The next section discusses changes which can

be made in this design to improve performance.

3　An Improved CLA Adder

The critical path delay of the simple CLA adder design presented in the previous section can be reduced significantly at the price of making the layout and wiring more complex.

3.1　Single Stage Group Generate

The first improvement to be made is using a single complex gate to create the group generate and propagate signals in a single stage directly from the adder inputs. In the simple design the expression used for the group 1 generate signal was as follows:

$$GG_1 = g_4 + p_4(g_3 + p_3 g_2) \tag{30}$$

Expanding this in terms of the adder inputs gives:

$$GG_1 = a_4 b_4 + (a_4 + b_4)[a_3 b_3 + (a_3 + b_3)a_2 b_2] \tag{31}$$

This equation can be implemented by an NMOS network containing 4 series transistors followed by an inverter. The PMOS network must implement the complement of this function, which normally would also require 4 series transistors. However, the relation $\overline{g_i}\,\overline{p_i} = \overline{p_i}$ can be used to simply the expression for $\overline{GG_1}$ as follows:

$$\overline{GG_1} = \overline{g_4}[\overline{p_4} + \overline{g_3}(\overline{p_3} + \overline{g_2})] \tag{32}$$
$$\overline{GG_1} = \overline{p_4} + \overline{g_4}(\overline{p_3} + \overline{g_3}\,\overline{g_2})] \tag{33}$$
$$\overline{GG_1} = \overline{a_4}\overline{b_4} + (\overline{a_4} + \overline{b_4})[\overline{a_3}\overline{b_3} + (\overline{a_3} + \overline{b_3})(\overline{a_2} + \overline{b_2})] \tag{34}$$

This simplified expression can be implemented by a PMOS network with 3 series transistors followed by an inverter. The gate implementing the group generate for group 1 is shown in figure 5. The gate implementing the group propagate is shown in figure 6. This change reduces the total number of series transistors used in forming the group generate signals from 7 to 5.

3.2　Carry Select Mux

The second improvement eliminates the need to travel back up the adder tree after the block carrys have been formed. This is done by generating two sets of sum bits. One set assumes the carry into each block will be 0, and the other set assumes it will be 1. This can occur in parallel with the generation of the block carrys which are then used to control a mux which selects the proper set of sum bits. This is the same method used in carry select adders.
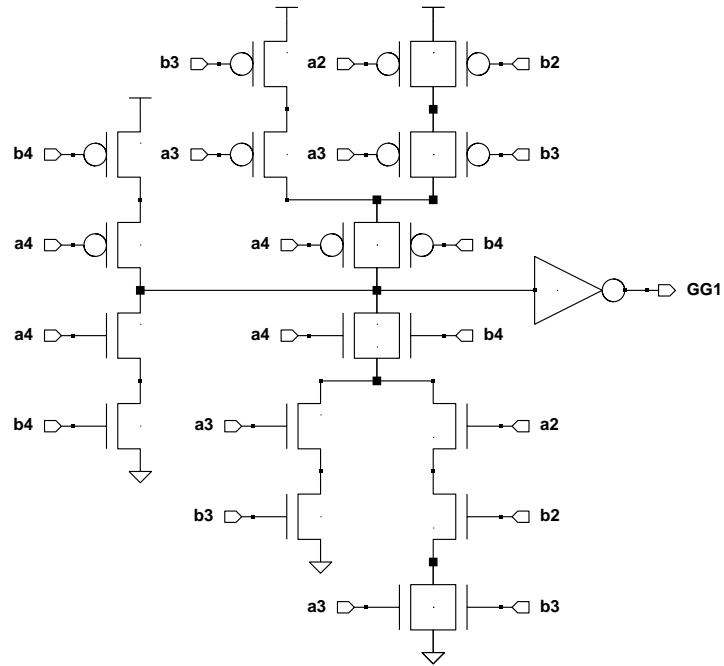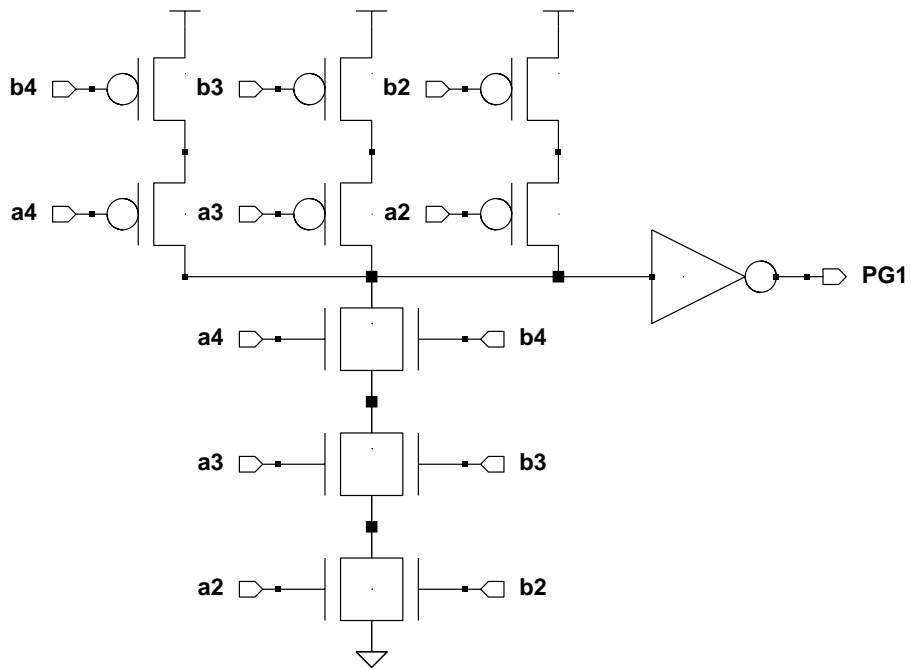
6

Figure 5: CLA Group Generate

Figure 6: CLA Group Propagate

In the simple CLA adder the equations implemented by group carry, local carry, and final sum stages for bit 23 are as follows:

$$s_{23} = t_{23} \oplus c_{23} \tag{35}$$

$$s_{23} = t_{23} \oplus CG_8 \tag{36}$$

$$s_{23} = t_{23} \oplus [GG_7 + PG_7(GG_6 + PG_6CB_2)] \tag{37}$$

This expression is converted to a mux controlled by CB_2 by defining the signals CGF_8 and CGT_8:

$$CGF_8 = GG_7 + PG_7GG_6 \tag{38}$$

$$CGT_8 = GG_7 + PG_7(GG_6 + PG_6) \tag{39}$$

The signal CGF_8 is the carry into group 8 assuming the block carry is zero, and CGT_8 assumes the block carry is one. The final sum bit is then written as:

$$s_{23} = \overline{CB_2}[CGF_8 \oplus t_{23}] + CB_2[CGT_8 \oplus t_{23}] \tag{40}$$

Using these signals, the other sum bits of the group are written in similar fashion.

$$s_{24} = \overline{CB_2}[(g_{23} + p_{23}CGF_8) \oplus t_{24}] + CB_2[(g_{23} + p_{23}CGT_8) \oplus t_{24}] \tag{41}$$

$$s_{25} = \overline{CB_2}[(g_{24} + p_{24}(g_{23} + p_{23}CGF_8)) \oplus t_{24}] + CB_2[(g_{24} + p_{24}(g_{23} + p_{23}CGT_8)) \oplus t_{24}] \tag{42}$$

Because the signals CGF_8 and CGT_8 will appear after the local generate and propagate signals, the critical path delay can be further reduced by applying the same principal to make the inputs to the mux controlled by the block carry muxes controlled by CGF_8 and CGT_8. This also allows the simplification of $g_i + p_i = p_i$ to be applied.

$$s_{23} = \overline{CB_2}[\overline{CGF_8}t_{23} + CGF_8\overline{t_{23}}] + \\ CB_2[\overline{CGT_8}t_{23} + CGT_8\overline{t_{23}}] \tag{43}$$

$$s_{24} = \overline{CB_2}[\overline{CGF_8}(g_{23} \oplus t_{24}) + CGF_8(p_{23} \oplus t_{24})] + \\ CB_2[\overline{CGT_8}(g_{23} \oplus t_{24}) + CGT_8(p_{23} \oplus t_{24})] \tag{44}$$

$$s_{25} = \overline{CB_2}\{\overline{CGF_8}[(g_{24} + p_{24}g_{23}) \oplus t_{25}] + CGF_8[(g_{24} + p_{24}p_{23}) \oplus t_{25}]\} + \\ CB_2\{\overline{CGT_8}[(g_{24} + p_{24}g_{23}) \oplus t_{25}] + CGT_8[(g_{24} + p_{24}p_{23} \oplus t_{25}]\} \tag{45}$$

The 3 bit slice which implements these functions is shown for group 8 in figure 7. sing the bit slice eliminates the need to go back up the adder tree after forming the block carrys, and reduces the critical path after the block carrys to a single mux delay.

Because of the reduced delay from the formation of the block carrys to the final sum output, C_{OUT} can no longer be implemented as a function of CB_2 as shown in equation 19 without becoming the critical path. To avoid this a fanin-4 generate gate is used to form C_{OUT} directly from the block generates and propagates.

$$C_{OUT} = GB_3 + PB_3[GB_2 + PB_2(GB_1 + PB_1GB_0)] \tag{46}$$

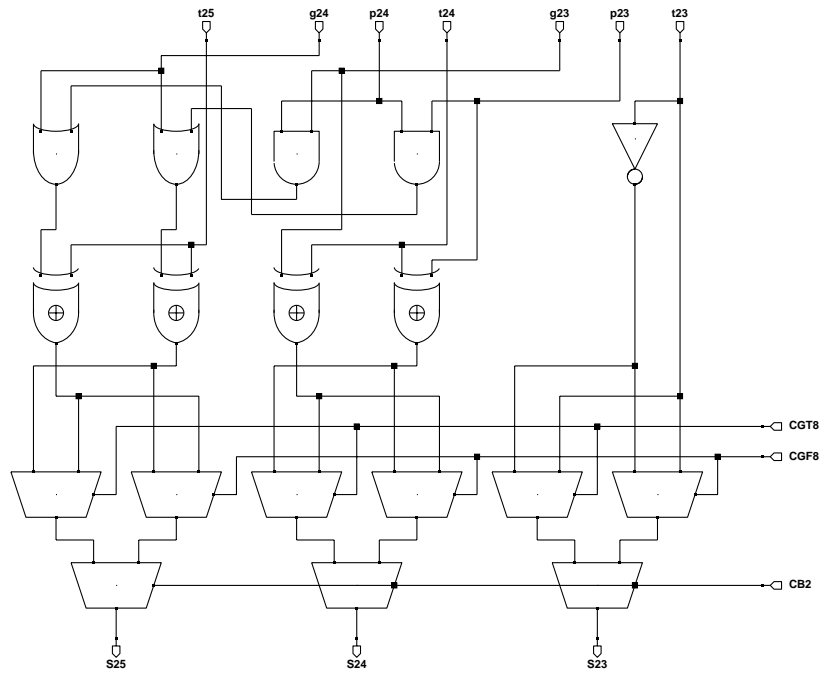This gate is shown in figure 8 and removes C_{OUT} from the critical path.
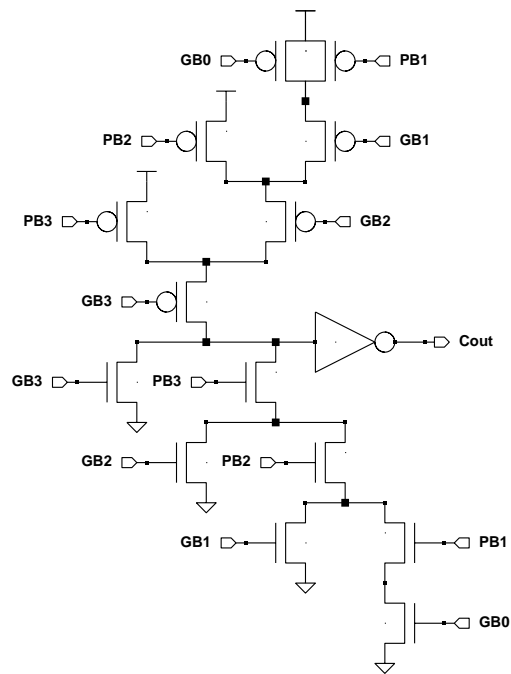
Figure 7: Sum Selection Slice

Figure 8: FanIn-4 Generate Gate

9

3.3 Critical Path

With a single stage group generate the critical path must still pass up $\lceil \log_r N \rceil - 1$ levels. Of these the first level will contain $r + 2$ series transistors and the others $r + 1$. The carry select mux eliminates the need to travel back up the levels of the adder to form the local carries. The mux delay from the arrival of the control signal is counted as one series transistor to form the complement of the control signal and one transistor to pass the input to the output. The number of series transistors in the critical path is therefore:

$$T_d = (\lceil \log_r N \rceil - 1)(r + 1) + 3 \tag{47}$$

For the 32 bit adder shown here with $r = 3$ this gives 15 series transistors. Using the single stage group generate eliminates 2 series transistors, and the carry select mux reduces the delay from the formation of the block carries from 9 series transistors to 2. The total critical path is reduced by 9 series transistors from a total of 24 to 15. The new critical path is shown in table 2.

Operation	Signal	Delay	Total
Group Generate	GG_i	5	5
Block Generate	GB_2	4	9
Block Carry	CB_3	4	13
Result Mux	s_{31}	2	15

Table 2: Improved CLA Critical Path

4 A Ling Adder

One final improvement that can be made to CLA design is the use of a pseudo-carry as proposed by Ling[1, 2]. This method allows a single local propagate signal to be removed from the critical path. To show how this is done the group generate signal for group 1 is shown below:

$$GG_1 = g_4 + p_4 g_3 + p_4 p_3 g_2 \tag{48}$$

Ling observed that each term in GG_1 contains p_4 except for the very first term which is simply g_4. However, p_4 can still be factored out of this expression by noting that $g_i = p_i g_i$.

$$GG_1 = p_4 GG_1^* \tag{49}$$
$$GG_1^* = g_4 + g_3 + p_3 p_2 \tag{50}$$

The Ling group generate signal (GG_1^*) is simpler and can be calculated more quickly than the CLA group generate signal. When expanded out the CLA and Ling group generates are as follows:

$$GG_1 = a_4 b_4 + (a_4 + b_4)[a_3 b_3 + (a_3 + b_3)a_2 b_2] \tag{51}$$
$$GG_1^* = a_4 b_4 + a_3 b_3 + (a_3 + b_3)a_2 b_2 \tag{52}$$

The gate used to implement the group generate signal is shown in figure 9 and has one less series transistor than the equivalent CLA gate shown in figure 5. he Ling group propagate signals (PG_i^*)
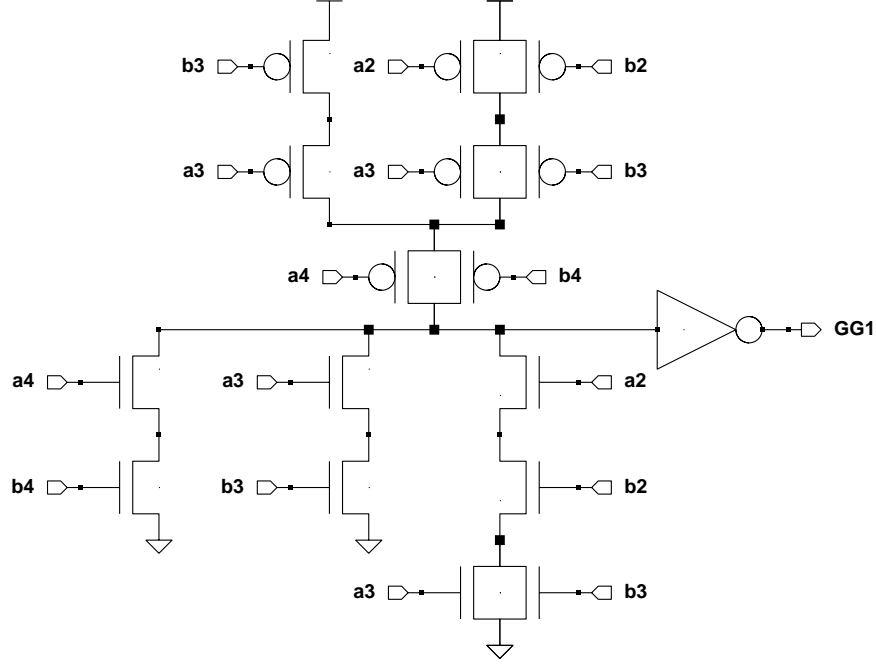

Figure 9: Ling Group Generate

are formed using the same gates as in the CLA design, but they are shifted one bit to the right. The CLA and Ling group propagate signals for group one are shown below.

$$PG_1 = p_4 p_3 p_2 \tag{53}$$
$$PG_1^* = p_3 p_2 p_1 \tag{54}$$

These Ling group generate and propagate signals are then combined in the same manner as before to create block carry signals.

$$CB_1^* = GB_0^* \tag{55}$$
$$CB_2^* = GB_1^* + PB_1^* GB_0^* \tag{56}$$
$$CB_3^* = GB_2^* + PB_2^*(GB_1^* + PB_1^* GB_0^*) \tag{57}$$
$$C_{OUT}^* = GB_3^* + PB_3^*[GB_2^* + PB_2^*(GB_1^* + PB_1^* GB_0^*)] \tag{58}$$

The true C_{OUT} is simply $p_{31} C_{OUT}^*$ which could be formed with a simple AND gate, but this would make it the critical path. Instead, the final group generate signal (GG_{10}) is formed using the CLA expression rather than the Ling group generate. Also the final group propagate (PG_{10}^*) is formed with a 4 input AND instead of a 3 input AND to include p_{31}. These changes allow the true C_{OUT} to be formed from the block generate and propagate signals as shown above without making it the critical path.

11

The final change that must be implemented to complete the Ling adder is to insert into the sum logic the local propagate signal which was factored out of each group generate. This is done simply by ANDing the CGF_i^* and CGT_i^* signals formed from the Ling group generate and propagates with the local propagate signal of the most significant bit of the previous group. This change is shown in figure 10 which depicts the sum selection logic for group 8 of the Ling adder.

4.1 Critical Path

The only difference in the critical path of the improved CLA and the Ling adder is the use of the Ling group generate is the first stage as shown in table 3. This allows the group generate signals to be formed in $r + 1$ series transistors instead of $r + 2$. The changes in the sum selection logic are off the critical path and have no effect on the total delay. Therefore, the series transistors in the critical path can be written as:

$$T_d = (\lceil \log_r N \rceil - 1)(r + 1) + 2 \tag{59}$$

For a 32 bit adder with $r = 3$ the net improvement of a Ling adder over the improved CLA adder is a total delay of 14 series transistors instead of 15.

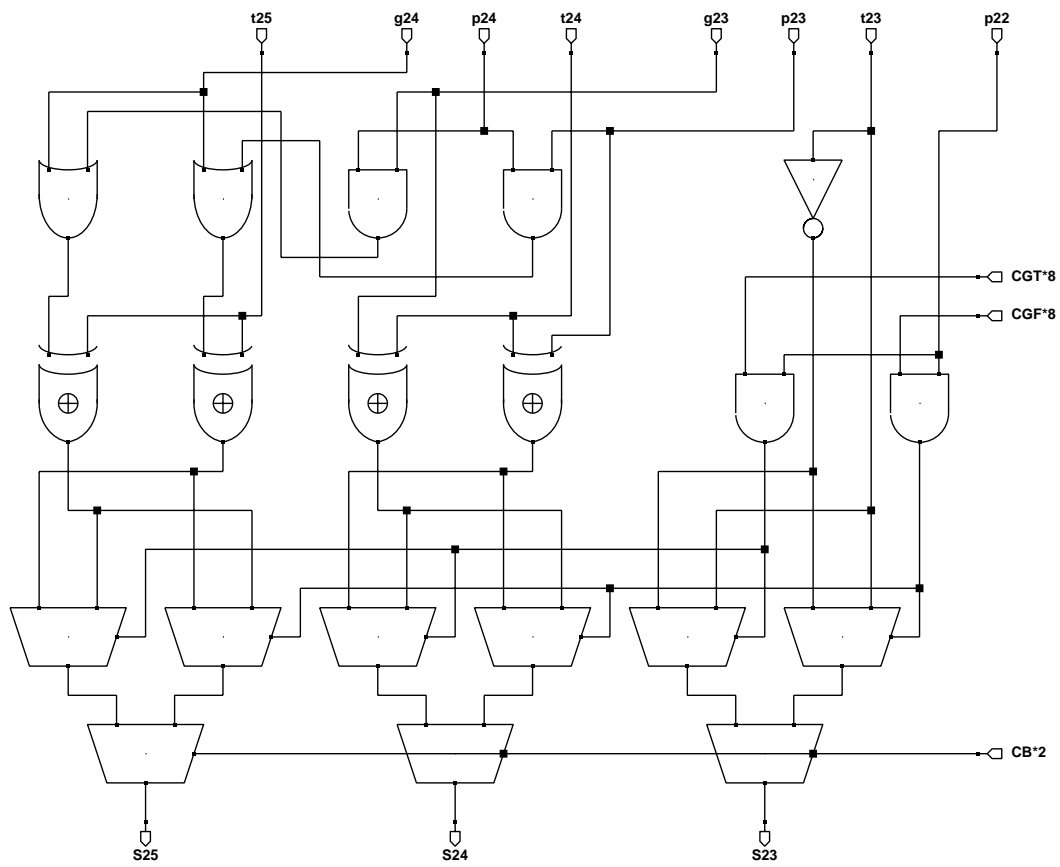| Operation	Signal	Delay	Total
Group Generate	GG_i	4	4
Block Generate	GB_2	4	8
Block Carry	CB_3	4	12
Result Mux	s_{31}	2	14

Table 3: Ling Critical Path

Figure 10: Ling Sum Selection Slice

13

References

[1] H. Ling. High speed binary parallel adder. *IEEE Transactions on Computers*, EC-15(5):799–802, October 1966.

[2] H. Ling. High speed binary adder. *IBM Journal of Research and Developement*, 25(3):156–166, May 1981.

[3] R. Brent and H. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, C-31(3):260–264, March 1982.

[4] G. Bewick, P. Song, G. DeMicheli, and M. Flynn. Approaching a nanosecond: A 32-bit adder. In *Proceedings of the International Conference on Computer Design*, pages 221–224, 1988.

[5] I. Hwang and A. Fisher. A 3.1ns 32b CMOS adder in multiple output domino logic. In *International Solid State Circuits Conference*, pages 140–141, 1988.

[6] A. Omondi. *Computer Arithmetic Systems: Algorithms, Architecture and Implementations*. Prentice Hall, 1994.

[7] N. Quach and M. Flynn. High-speed addition in CMOS. Technical Report CSL-TR-90-415, Stanford University, February 1990.

[8] S. Waser and M. Flynn. *Introduction to Arithmetic for Digital Systems Designers*. Holts, Rinehart and Winston, 1982.