

浮動小数点演算器の高速化 v4

塩谷 亮太 (shioya@ci.i.u-tokyo.ac.jp)

東京大学大学院情報理工学系研究科 創造情報学専攻

浮動小数点演算器の高速化

■ 内容：

- ◇ 背景：冗長表現と乗算器
- ◇ LZA: Leading Zero Anticipation
- ◇ LZA の補正の高速化: Lutz's mask
- ◇ 絶対値を取る工夫
 - End-around-carry adder/丸めによる加算との統合
- ◇ ウォレス木の厚みを意識した加算

■ [CMOSVLSI2014]

- ◇ 整数の加算器や乗算器の作り方が良くまとまっている
 - PPA の各種トポロジ, Booth エンコーダなど
- ◇ 色々みた中ではこの本の説明が最もわかりやすかった

■ [HFPA2018]

- ◇ 浮動小数点演算（ハードに限らず）全般が良くまとまっている
- ◇ FP 加算器や FMA の基本的な実装から各種最適化まで書かれている

背景：冗長表現と乗算器

バイナリと Carry save 表現

■ バイナリ表現

◇ 通常の 2 進数の値の表現の事

■ Carry save 表現（冗長表現）

◇ 1 つの数値を s と c の 2 組の和で表す

□ $s+c$ がバイナリ表現になる

□ =Carry save 表現では, ある 1 つの数を表す方法が複数存在する

◇ 同じ数値を表す例 :

□ バイナリ表現 : $v=4b'1110$

□ carry save 表現 : $s=4b'1001, c=4b'0101$
 $s=4b'1100, c=4b'0010...$

◇ 以下のようにも呼ばれる

□ 「redundant representations」 「partial-product」

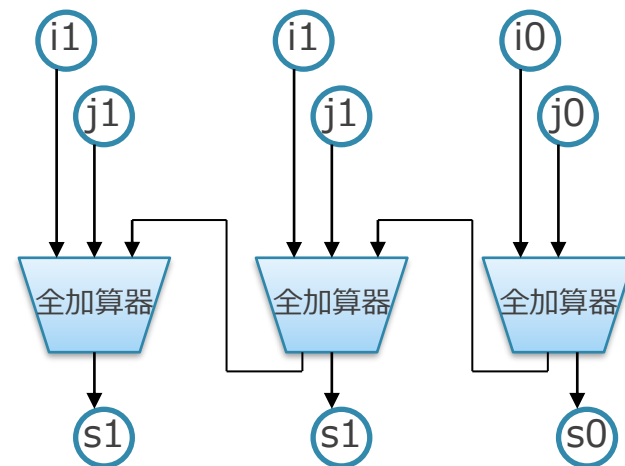
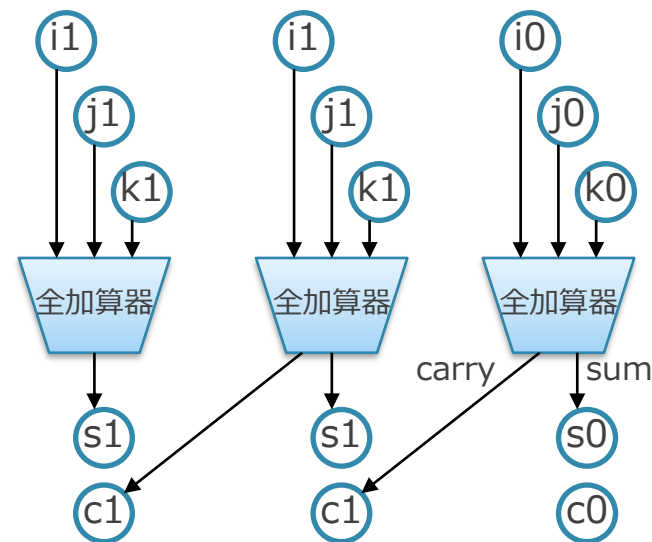
CSA と桁上げ加算器

■ CSA: Carry Saved Adder

- ◇ Carry save 表現での加算を行う加算器
 - 全加算器を並列に並べたもの
- ◇ 回路規模が $\Theta(w)$, 遅延が $\Theta(1)$
 - 回路規模はトランジスタ数を想定

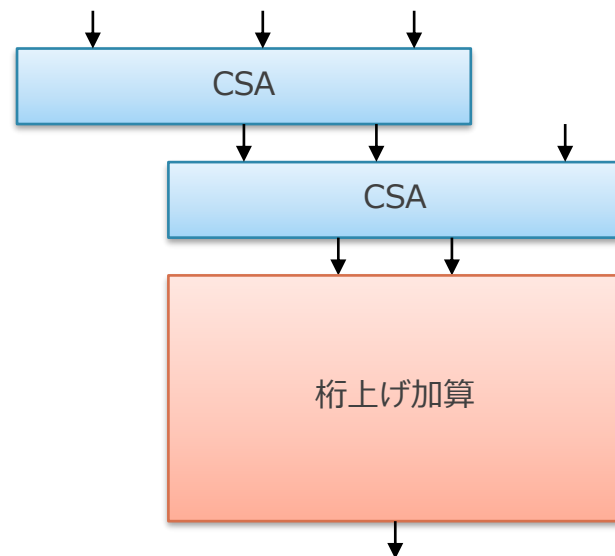
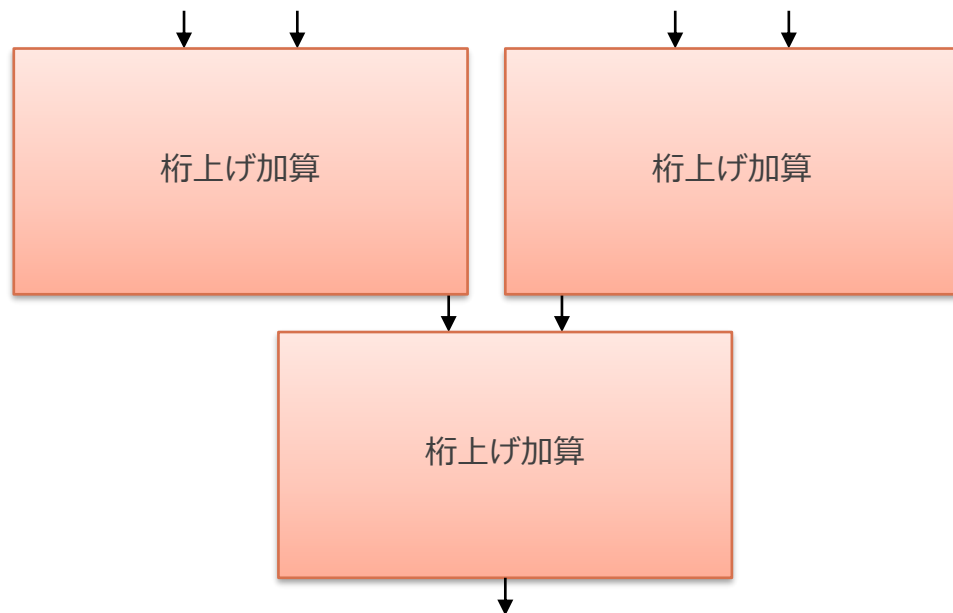
■ CPA: Carry Propagation Adder, 桁上げ加算器

- ◇ いわゆる通常の加算器
- ◇ 右図はリプルキャリーだが, 普通は **Parallel Prefix Adder (PPA)** が使われる
- ◇ PPA には色々組み方がある
- ◇ おおよそ 回路規模が $\Omega(w) \sim O(w \log w)$, 遅延が $\Omega(\log w)$



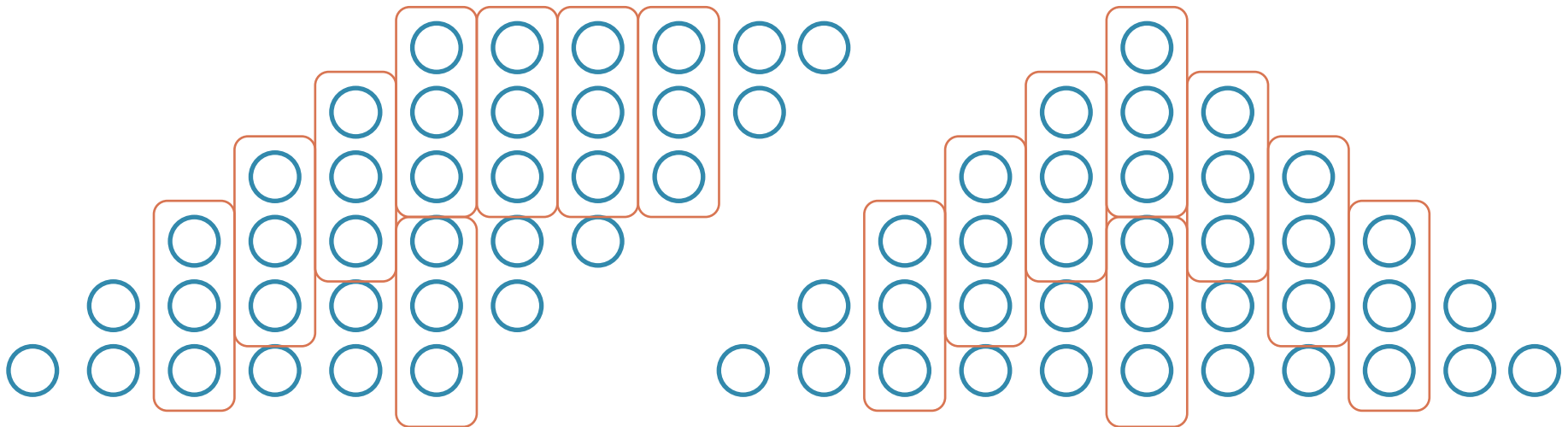
CSA による加算のチェーンは非常に軽い

- $A+B+C+D\dots$ みたいなものは,
 - ◇ CSA で 2 つの値になるまで足し切ってから,
 - ◇ 最後に 1 回だけ桁上げ加算器で加算を行ってバイナリ表現にすれば良い



CSA を使った乗算器

- 2進数の乗算は、基本的には筆算と同様に行う
 - ◇ 筆算で各桁を足し込んでいくのを CSA で行うので軽い
 - ◇ 筆算の各桁を 3 入力 2 出力 CSA でドンドン足していき、2 つになるまで潰す
- Wallace 木
 - ◇ この CSA による加算をツリー状に組んだもの



Booth エンコーディング

■ Booth エンコーディング

- ◇ 入力を冗長表現的なものに変換してから乗算する
- ◇ 部分積の数が半分になる
 - 2進に比べると10進でやると筆算の桁数がへるようなもの
- ◇ 基数4（冗長な4進表現）がよく使われる
 - -2,-1,0,1,2 倍の組み合わせに変換する
 - これらはシフトとビット反転だけで作れる

LZA: Leading Zero Anticipation

Leading Zero Count と Leading Zero Anticipation

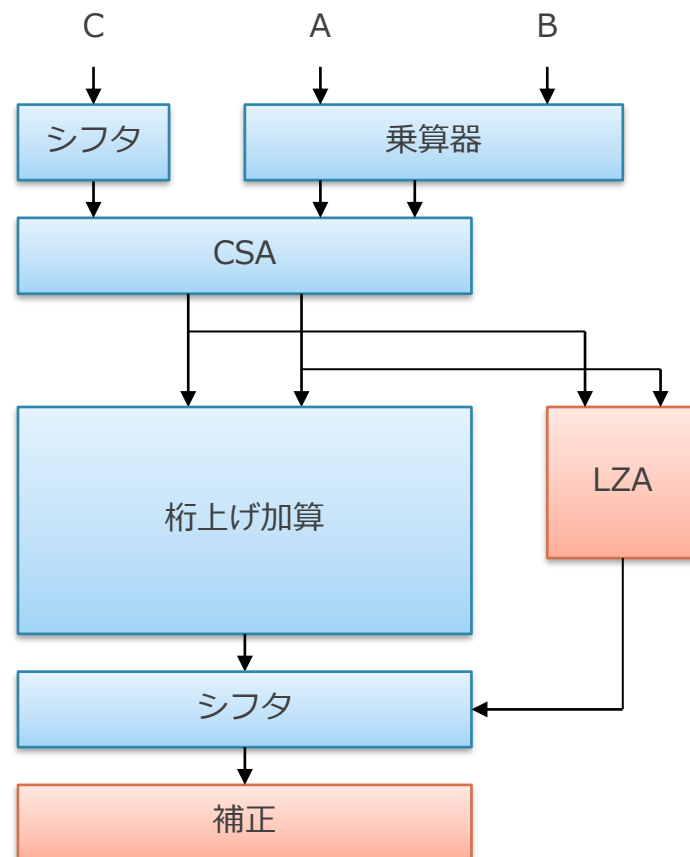
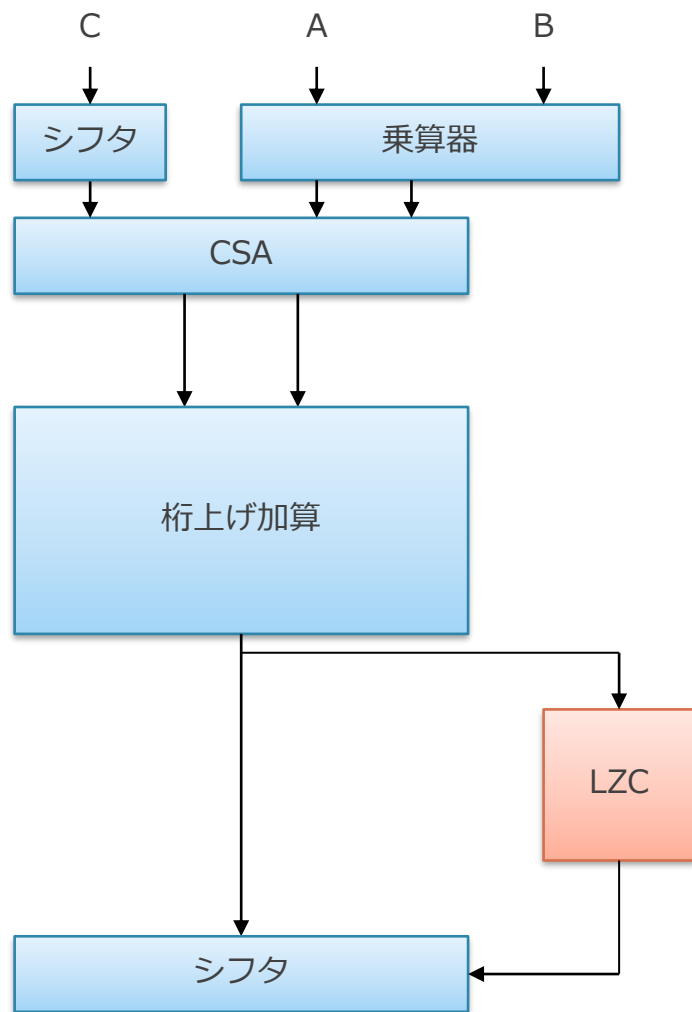
■ LZC: Leading Zero Count

- ◇ 最上位の連続したゼロの数を数える
- ◇ 浮動小数点数の正規化で使う
 - 減算でできる上位のゼロの部分をシフトで埋める

■ LZA: Leading Zero Anticipation

- ◇ 乗算の最後の桁上げ加算と並行して、ゼロの数を予測する
 - $c+s$ を計算してる間に $LZA(c, s)$ により、 $c+s$ の結果のゼロの数を予測する
- ◇ 真のゼロの数から 1 ずれる可能性がある
 - シフトした結果の MSB をみて補正する
- ◇ 結果が符号付きでもできる（後述）

LZC と LZA



LZA のやりかた

■ 方針：

1. 推定ビット列 L を作る

- A+B の真の結果に対し，上位の連続ゼロの個数が同じになるビット列を推定により作る
- 実際には真の結果から 1 ずれる場合がある

2. L に対して通常の LZC を行いゼロの数を数える

■ 例：

◇ A+B の真の結果： 0b0000_1000

◇ L： 0b0001_1011

- 上位の連続ゼロ個数が 1 つ少ない
- そこより下位はどうなっても良い

符号付き加算の LZA

- $A+B$ が負になる場合に対応する必要がある
- 結果が負の場合に対応するため以下の予測を行う
(両者は式変形すると意味的に等価)
 - ◇ $\text{abs}(A+B+1)$ の leading zero count
 - ◇ $A+B$ の leading *sign* count

符号付き加算の LZA

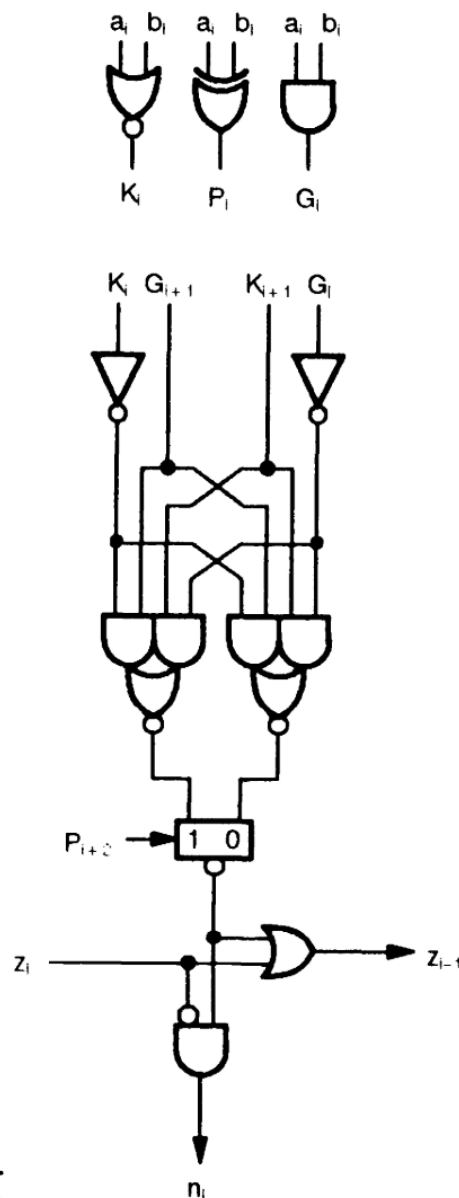
- [Knowles1991] の Fig.4 に真理値表がある
- kill, propagate, generate から最初に sign が途切れるパターンを検出

A = +141 00010001101
B = - 41 11111010111

 +100 00010100100
 PPP[^]GPKPPGPG

A = +141 00010001101
B = -127 11110000001

 + 14 00000001110
 PPPGKKK[^]KPPKG



[i+2...i]	z _i
KKK	0
KKP	1
KKG	1
KPK	x
KPP	x
KPG	x
KGK	1
KGP	1
KGG	0
PKK	1
PKP	1
PKG	0
PPK	0
PPP	0
PPG	0
PGK	0
PGP	1
PGG	1
GKK	0
GKP	1
GKG	1
GPK	x
GPP	x
GPG	x
GGK	1
GGP	1
GGG	0

Fig. 4 : Normalization Distance Predictor Cell

符号付き加算の LZA の実装例

- 以降の実装で $L()$ と $L2()$ に対して LZC を行くと、真の符号の数に対して $+0$ or -1 の予測が得られる
 - ◇ シミュレーションにより全数で確認済み
- これらの実装の違いは、
[Knowles1991] の真理値表の x の部分への対応が違っただけ

実装例 1

```
// [Schmookler12001] の Eq.1 より
localparam W = 8;
function automatic logic [W-1:0] L(input logic [W-1:0] a, input logic [W-1:0] b);
    logic [W:0] sa, sb;
    logic [W:0] H, HC, Z, ZC, G, GC;
    sa = {a[W-1], a}; // 1bit 上までみるので符号拡張
    sb = {b[W-1], b};
    H = sa ^ sb;
    HC = ~H;
    Z = ~sa & ~sb;
    ZC = ~Z;
    G = sa & sb;
    GC = ~G;
    // 一番下のビットは常に 1 でも LZA の性質的に大丈夫 (+0 or -1 を予測するので)
    return
        (HC>>1 & Z & ZC<<1) | (HC>>1 & G & GC<<1) |
        (H>>1 & G & ZC<<1) | (H>>1 & Z & GC<<1) | 1;
endfunction
```

実装例 2

```
// [Hoskote2002] より. インテルの特許だが期限切れ (そもそも多分成立していない)
localparam W = 8;
function automatic logic [W-1:0] L2(input logic [W-1:0] a, input logic [W-1:0] b);
    logic [W:0] sa, sb;
    logic [W:0] P, X, G, N, O, Z;
    logic [W:0] n, o, L;
    sa = {a[W-1], a};
    sb = {b[W-1], b};
    P = sa ^ sb;
    X = ~P;
    G = sa & sb;
    N = ~G;
    O = sa | sb;
    Z = ~O;
    n = N | (N<<1);
    o = O | (O<<1);
    L = (X>>1) & n & o
        | (P>>1) & G & (O<<1)
        | (P>>1) & Z & (N<<1);
    return L | 1;
endfunction
```

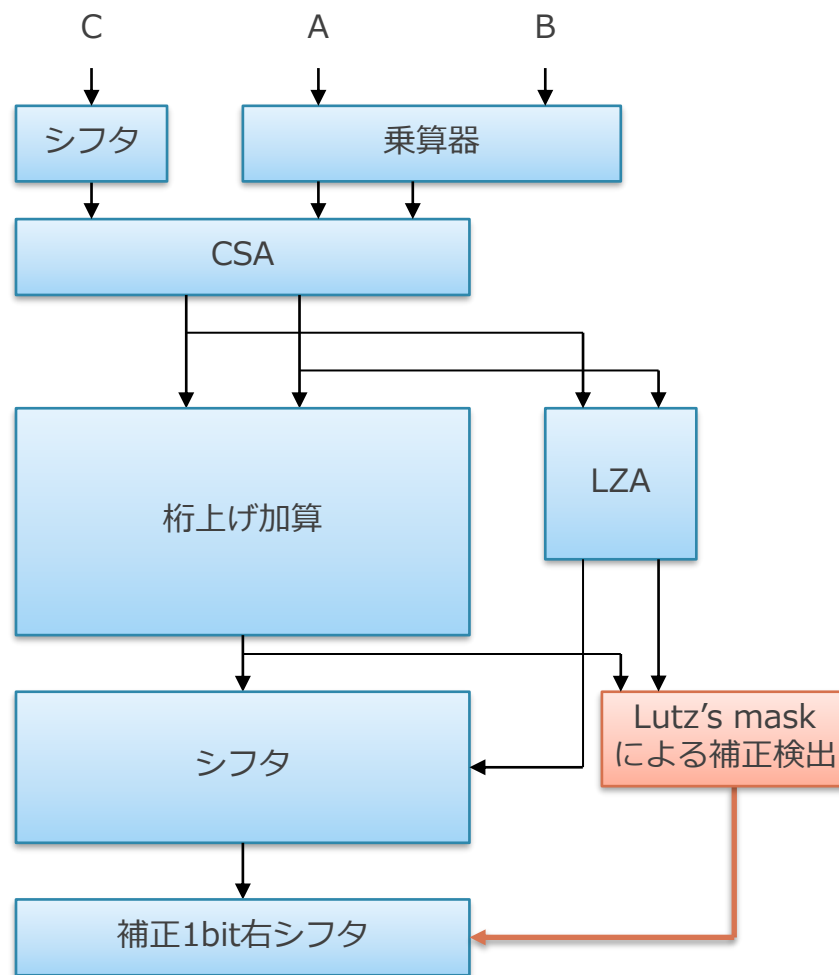
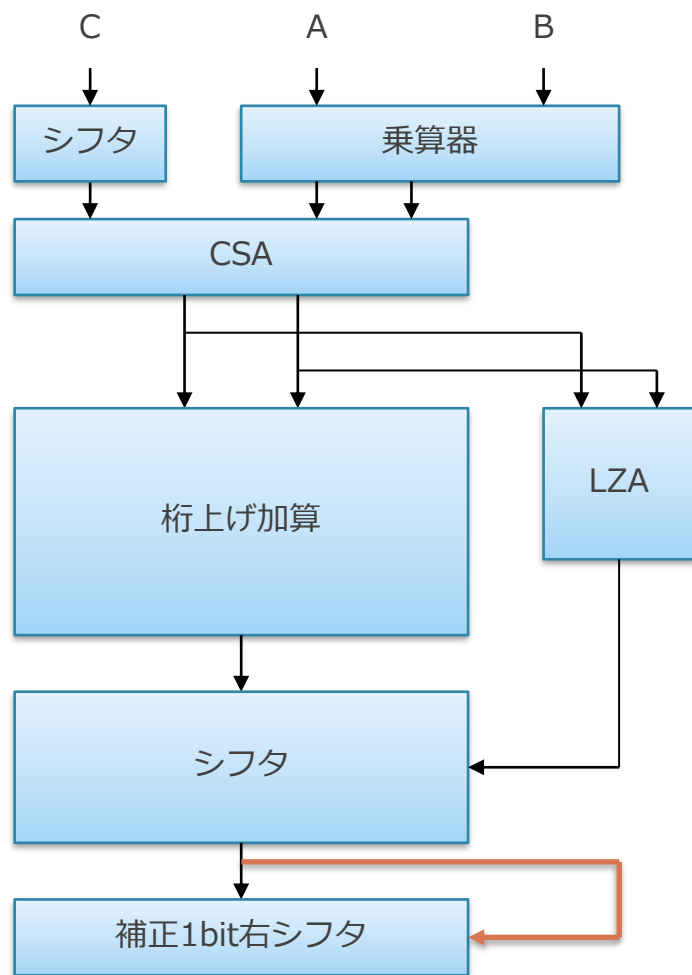
- 多くの文献では LZA は +0 or +1 を予測すると記述されている
 - ◇ 既存の LZA やシフタの回路との RTL 上の記述の互換性をたもちやすくするために, +0 or -1 として説明した
 - ◇ 推定値 L の左端のビットを増減することで「+0 or +1」か「+0 or -1」は調整可能
- 符号付き LZA は K,P,G の 3 ビットをみて推定値 L を作っている
 - ◇ 最上位の連続した 0 や 1 が, K,P,G がどのように現れると途切れるのかを全通り考えると説明できる
 - ◇ [HPEEMD2006] P.201 に場合分けによる具体的な説明がある

LZA の補正の高速化

LZA の補正

- LZA はシフト量を +0 or -1 で予測する
 - ◇ 正規化後の MSB を見て, 0 ならもう 1 ビット左シフト
 - ◇ fanout が大きく, おそらくここはそこそこ遅延が大きい
- [Lutz2017] これを正規化のシフト並列になるべく早く検出する
 - ◇ シフト後の MSB にあたるビットをマスクにより抽出,
 - ◇ OR でかき集めて高速に得る
 - シフタよりも OR ツリーの方が速い

Lutz's mask による補正検出



補正の必要性判定

■ LZA の補正結果の取得

1. $A+B$ による桁上げ加算の結果と Lutz's mask を AND 演算
2. その結果の全ビットを OR
3. この結果が 0 なら, 追加で 1 ビットシフトが必要

■ 例 :

◇ LZA L: 0b0110_1001

1. Lutz's mask: 0b0100_0000

2. $A+B$: 0b0011_1010

3. $|((A+B)\&\text{mask})$: 0b0

□ → LZA に従ったシフト後の最上位が 0 → 補正が必要

Lutz's mask [Lutz2017]

- LZA の推定値 L から, 最も上位の 1 だけが立ったマスクを作る

◇ $L=0b01001110$ の場合,
 $mask=0b01000000$

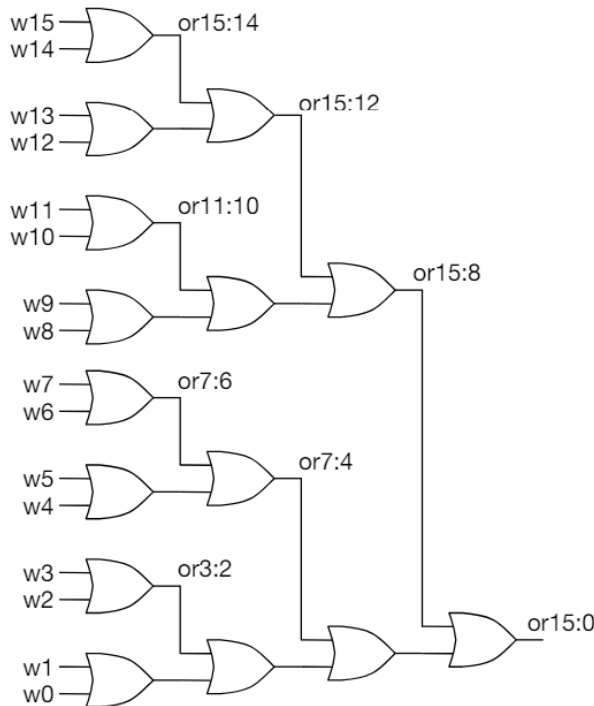


Fig. 5. OR tree for masks

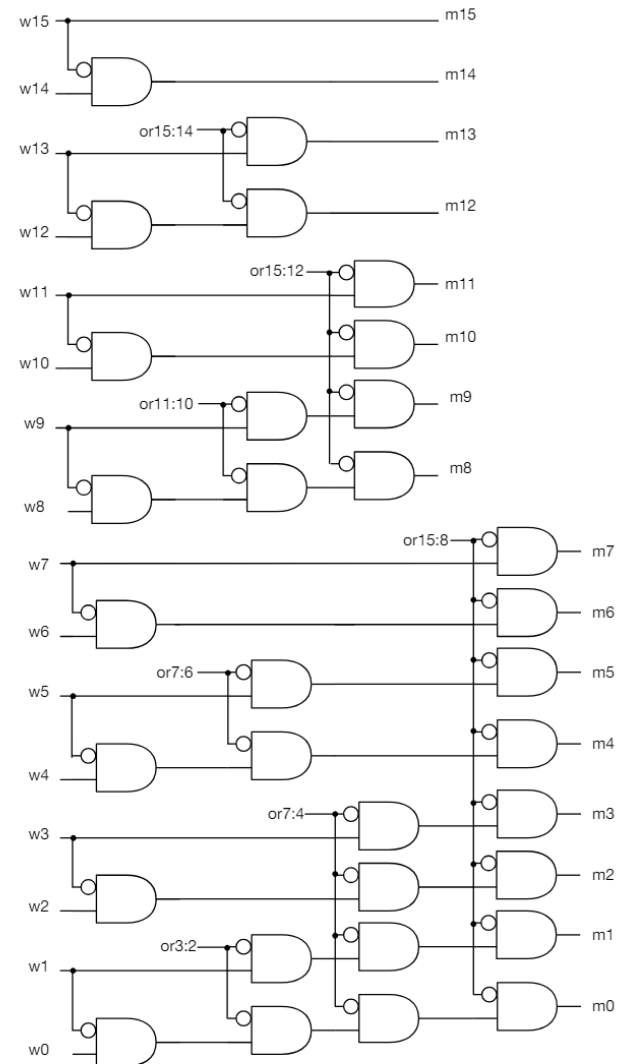


Fig. 6. First one mask

Lutz's mask の別の作り方？

■ 作り方：

◇ L を温度計エンコーディング T に変換

□ L の各ビットに対し，自身より上位に 1 がいたら 1 が立つよう OR のネットワークを組む

□ PPA と同様のプリフィクス演算になる

＊ [CMOSVLSI2014] P.668 参照

◇ T をビット反転し，右に 1 ビットシフトしてから T と AND

■ 例：L=0b01001110 の場合

◇ T=0b01111111

◇ mask=

$(\sim T \gg 1) \& T =$

$(0b10000000 \gg 1) \& 0b01111111 = 0b01000000$

応用: guard や sticky ビットの早期取得

- Lutz's mask の応用で正規化前にわかる
 - ◇ 仮数部の幅 ($+\alpha$) だけずらせば, guard bit を取り出せる
 - ◇ sticky bit では温度計エンコーディングの時点のマスクを使えば良い

絶対値を取る工夫

絶対値を取る工夫

- FP では計算結果の仮数から絶対値を取る必要がある
 - ◇ 減算の場合は必要に応じて符号反転を行う
 - ◇ 符号をみて +1 or ビット反転で実現
- $|A-B|$: $A+\sim B$ を計算して,
 - ◇ 結果が正なら結果に +1:
 - $(A+\sim B)+1 = A-B-1+1 = A-B$
 - ◇ 結果が負なら結果をビット反転:
 - $\sim(A+\sim B) = -(A+\sim B)-1 = -A-\sim B-1 = -A+B$
 - ◇ 備考: $-B=\sim B+1 \rightarrow \sim B = -B - 1$

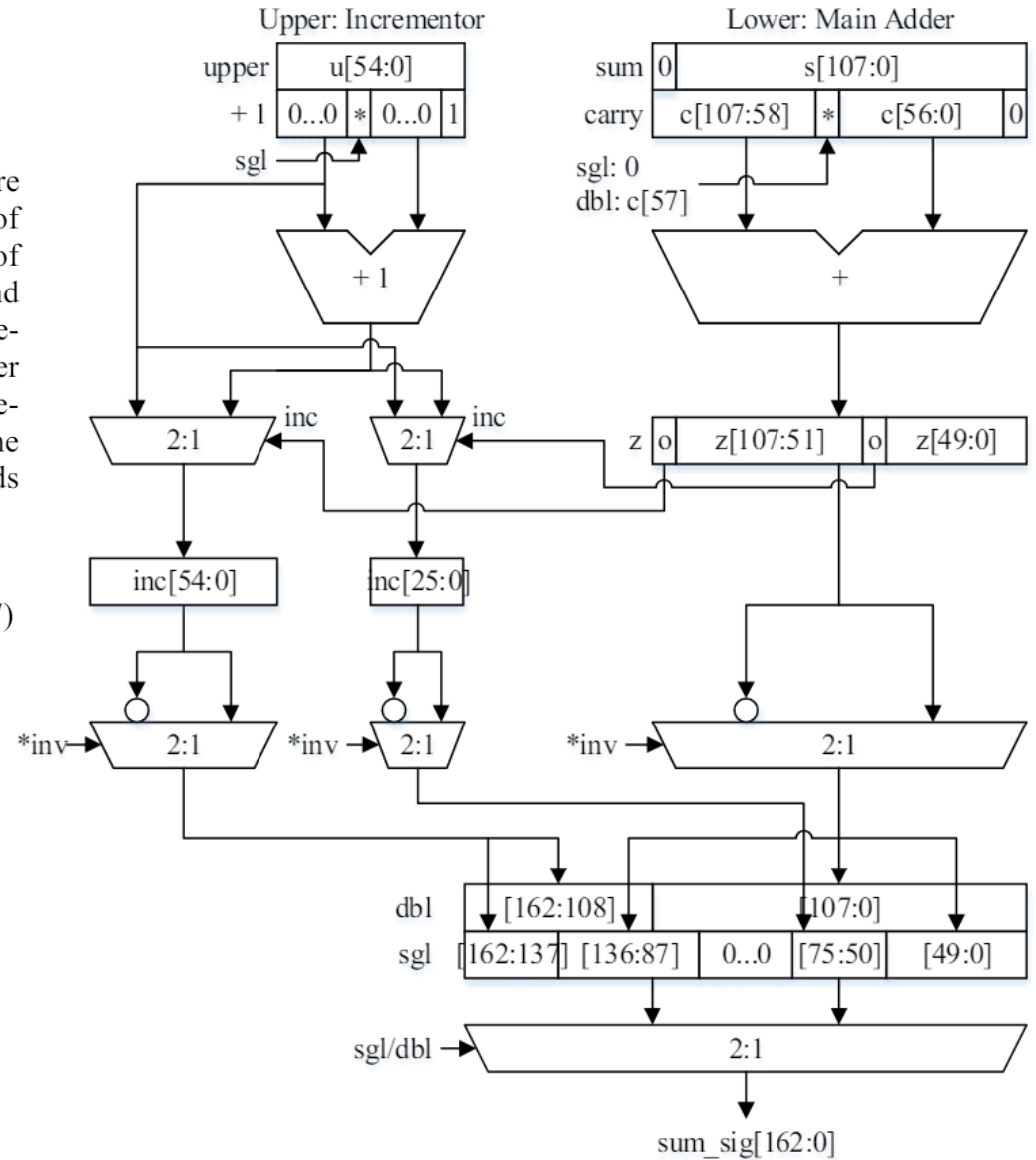
[Sohn2023] より

C. Main adder and Incrementor

The significand sum and carry from the multiplier are passed to the main adder. The main adder computes the sum of the two significands for a set of double precision, or two sets of single precision as shown in Fig. 5. Also, the upper significand from the alignment is passed to the incrementor. The incrementor adds one to the upper significand only if the main adder produces carry-out. The result of the main adder and incrementor needs to be two's complemented if it is positive. On the other hand, the result of the main adder and incrementor needs to be inverted if it is negative.

$$\begin{aligned} X - Y &= X + \bar{Y} + 1 & (X > Y) \\ Y - X &= \overline{X + \bar{Y}} & (X < Y) \end{aligned}$$

(7)



*inv = upper allones & inc & truesub

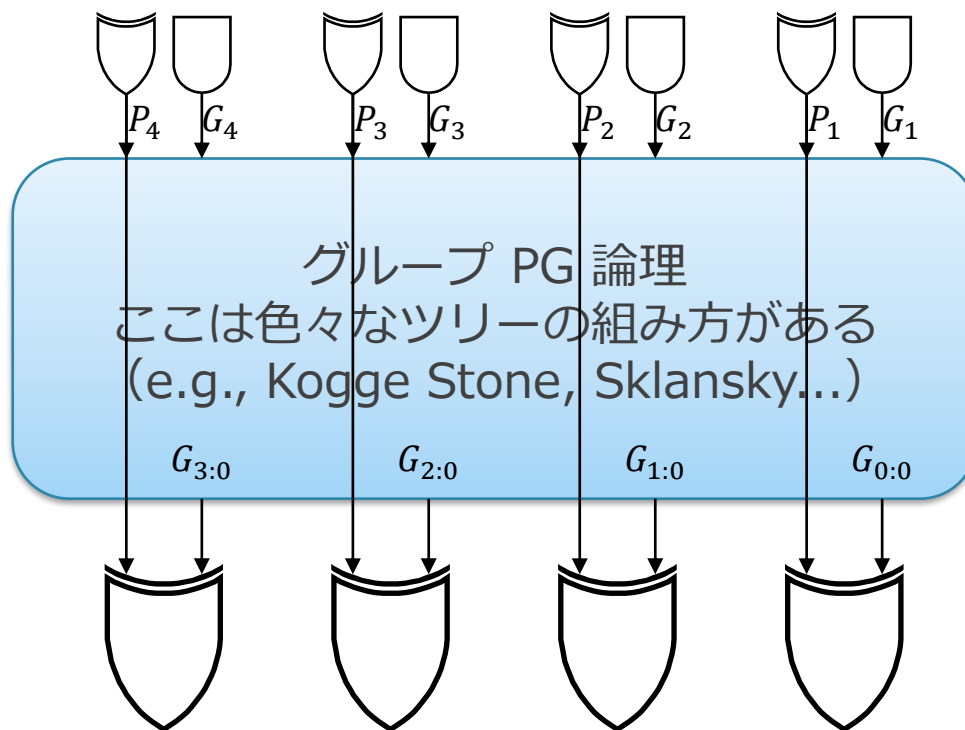
Fig. 5. Main Adder and Incrementor

加算と絶対値

- 「結果が正なら結果に +1」を実現するためには、符号を知るために一度桁上げ加算をしないといけない
 - ◇ そのままではインクリメントのための桁上げ加算が直列に行われる
- 解決方法：
 1. End-around-carry adder
 2. 丸めによる加算との統合

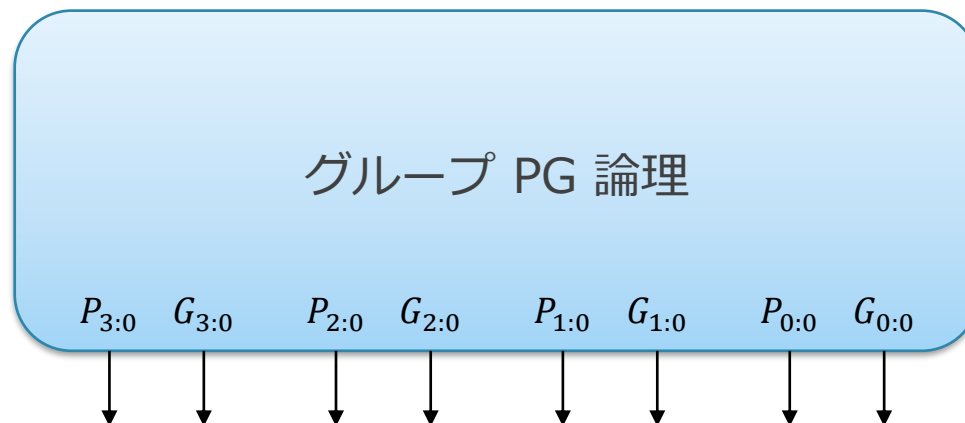
Parallel Prefix Adder (PPA)

- 自分より先行する（下位全体の）P と G を並列に計算して加算を行う
 - ◇ グループキャリー生成： $G_{i:j} = G_{i:k} + P_{i:k} \cdot G_{k-1:j}$
 - ◇ グループキャリー伝搬： $P_{i:j} = P_{i:k} \cdot P_{k-1:j}$



End-around-carry adder / Flagged adder

- グループ PG 論理は、各桁毎にそこまでのキャリー伝搬の有無を算出する
 - ◇ キャリー伝搬の情報から、最後に1回だけ追加 +1 ができる
 - $G'_{i-1:0} = G_{i-1:0} + P_{i-1:0} \cdot inc$
 - ◇ A+B の結果の符号をみて、結果に応じてさらに +1 をする事ができる
 - うまく使くと $abs(A-B)$ が1つの桁上げ加算でできる
- [CMOSVLSI2014] P.626, [HPEEMD2006] P.199 に説明あり



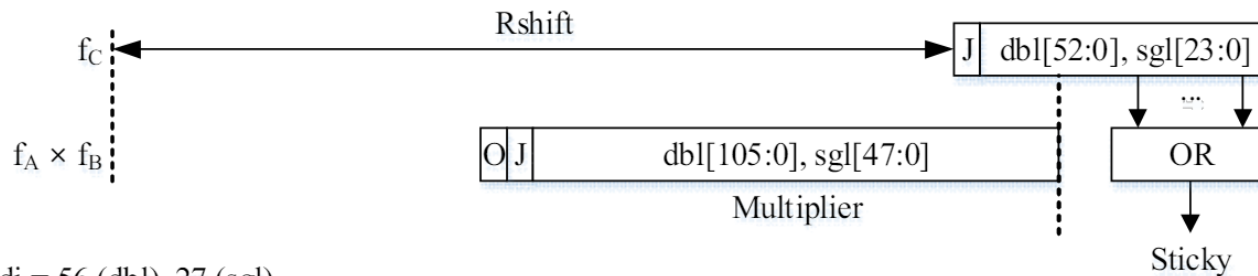
補数を取るインクリメントと round away の統合

[Sohn2023] より

- 正規化前の LSB への +1 が、
正規化後に影響を与えるのは（正規化後 LSB に繰り上がるのは）、
正規化前の下位ビット（sticky bit 相当分）が全部 1 のときのみ
 - ◇ sticky bit 相当分が全部 1 だったかを正規化と並列して検出できる
- 繰り上がりが発生した場合、結果の下位ビットは全てゼロなので、
そこからさらに切り上がることはない
 - ◇ 正規化前の LSB への +1 と、丸めの切り上げ時の +1 は排他で起きる
 - ◇ +1 する回路は丸めのところの 1 つでよい

4. Big right shift

```
exp_diff> *max
```



*adj = 56 (dbl), 27 (sgl)

$$*_{\max} = 109 \text{ (dbl)}, 51 \text{ (sgl)}$$

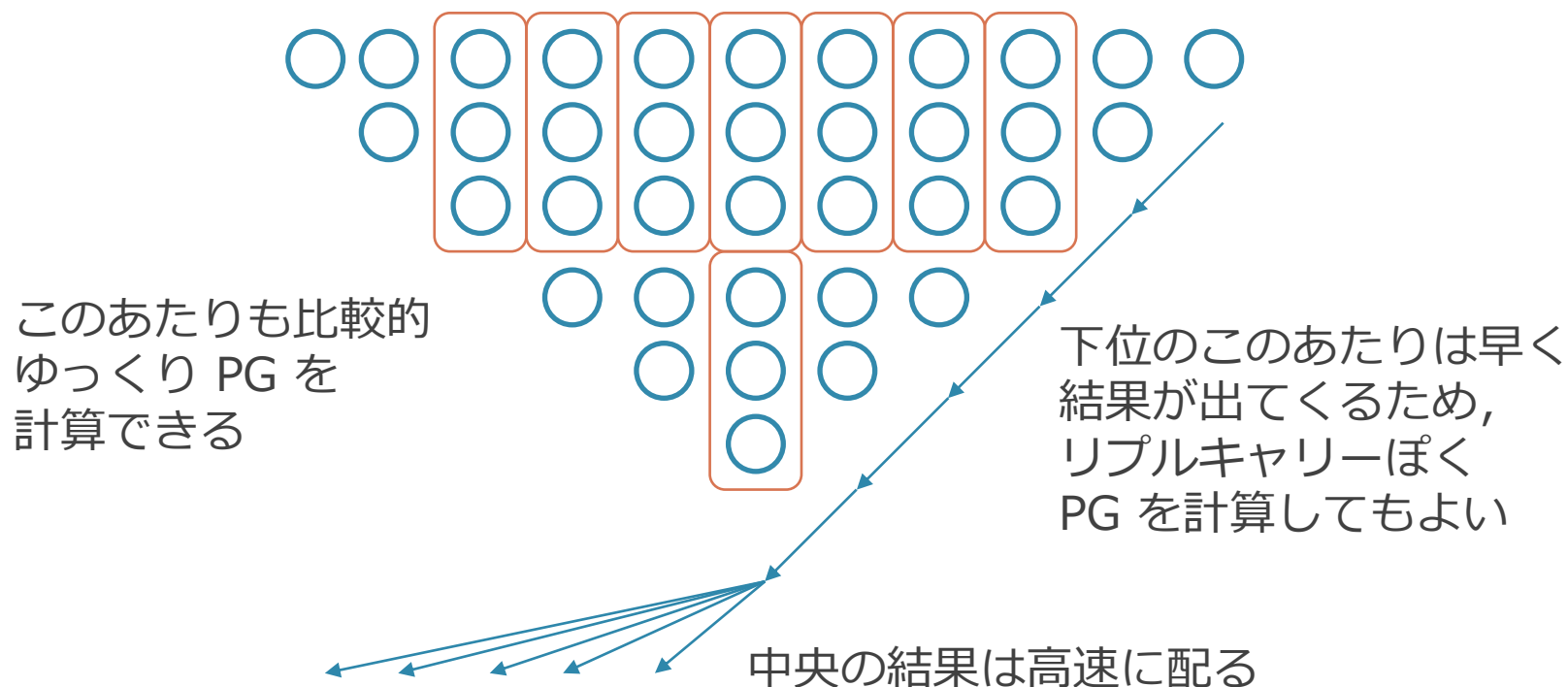
End-around-carry との比較

- End-around-carry adder を使うと, $a+b$ と $a+b+1$ を同時に求められる
 - ◇ 結果の符号ビットをみてから $+1$ 側を選択できる
 - ◇ 欠点：ただし, この加算器を手書きで作らないといけない
 - 自動合成させられない
- 丸めへの統合だと上記は必要ない
 - ◇ インクリメントはただの加算として記述できる
 - ◇ 利点：自分で桁上げ加算器を作る必要がない

ウォレス木の厚みを意識した加算

ウォレス木と加算器

- ウォレス木では中央部分の CSA の段数が厚く、左右が薄い
 - ◇ これを利用した桁上げ加算器が使える
 - ◇ 特に下位側はツリーではなく直列になっており、軽い
 - ◇ [CMOSVLSI2014] P.667 にもう少し詳しい記述あり



桁上げ加算器の自動合成を使う場合

- 前提：合成系による自動合成に頼れるなら頼るのが一番良い
 - ◇ 「 $A*B+C$ 」のように書けば合成系が自動で合成をしてくれる
 - ◇ メンテナンスコストが低いし，典型的にはかなり高速
- 前ページのような工夫を，自動合成で実現することを考える
 - ◇ 上位，中位，下位で分けて加算を記述するのが良いかもしれない
 - ◇ 下位は時間に余裕があるので，勝手にリプルキャリーぽくなることを期待
 - ◇ 上位は，下から +1 が来る場合も計算しといて選択するなど

參考資料

参考文献

- ◇ [CMOSVLSI2014] ウェスト&ハリス CMOS VLSI 回路設計 応用編:
https://www.maruzen-publishing.co.jp/item/?book_no=294477
- ◇ [Hoskote2002] Yatin Hoskote Intel Corp:
Leading Zero Anticipatory (LZA) algorithm and logic for high speed arithmetic units (期限切れ特許)
<https://patents.google.com/patent/US7024439B2/en>
- ◇ [Knowles1991] Simon Knowles:
Arithmetic Processor Design for the T9000 Transputer
<http://www.transputer.net/fbooks/t9000/t9kfpdsn.pdf>
- ◇ [Lutz2017] David R. Lutz:
Optimized Leading Zero Anticipators for Faster Fused Multiply-Adds
<https://ieeexplore.ieee.org/abstract/document/8335443>
- ◇ [HFP2018] Jean-Michel Muller , Nicolas Brunie , Florent de Dinechin , Claude-Pierre Jeannerod , Mioara Joldes , Vincent Lefèvre , Guillaume Melquiond , Nathalie Revol , Serge Torres:
Handbook of Floating-Point Arithmetic
<https://link.springer.com/book/10.1007/978-3-319-76526-6>
- ◇ [HPEEMD2006] Vojin G. Oklobdzija, Ram K. Krishnamurthy:
High-Performance Energy-Efficient Microprocessor Design
<https://link.springer.com/book/10.1007/978-0-387-34047-0>
- ◇ [Schmooklerl2001] Martin S. Schmooklerl and Kevin J. Nowka2:
Leading Zero Anticipation and Detection -- A Comparison of Methods
<https://redirect.cs.umbc.edu/~phatak/645/supl/lza/lza-survey-arith01.pdf>
- ◇ [Sohn2023] Jongwook Sohn, David K. Dean, Eric Quintana and Wing Shek Wong:
Enhanced Floating-Point Multiply-Add with Full Denormal Support (二つ目はスライド)
<https://arith2023.arithsymposium.org/papers/Enhanced%20Floating-Point%20Multiply-Add%20with%20Full%20Denormal%20Support.pdf>
https://arith2023.arithsymposium.org/slides/S8_JongwookSohn_EnhancedFloatingPointMultiplyAddWithFullDenormalSupport.pdf