特別研究報告書

題目

Hybrid 加算器の合成に関する研究

Synthesis of hybrid adders

指 導 教 員

橘　昌良　助教授

報 告 者

学籍番号: 1055078

氏名: 幾井崇博

平成　15 年　1 月　27 日

高知工科大学　電子･光システム工学コース

目次

1　はじめに・・・・・・・・・・・・・・・・・・・・・・・・・1

2　加算回路の種類と特徴・・・・・・・・・・・・・・・・・・・2

2-1　加算回路の種類と特徴・・・・・・・・・・・・・・・・2

2-2　Half adder と Full adder・・・・・・・・・・・・・2

2-3　Ripple carry adder・・・・・・・・・・・・・・・4

2-4　Carry lookahead adder・・・・・・・・・・・・・6

2-5　Carry skip adder・・・・・・・・・・・・・・・16

2-6　Carry select adder・・・・・・・・・・・・・・・25

2-7　Hybrid adder について・・・・・・・・・・・・・27

3　加算回路の記述・・・・・・・・・・・・・・・・・・・・・29

3-1　文法の構成・・・・・・・・・・・・・・・・・・・・・29

3-1-1　RCA の記述・・・・・・・・・・・・・・・・・32

3-1-2　CLA の記述・・・・・・・・・・・・・・・・・33

3-1-3　CSK の記述・・・・・・・・・・・・・・・・・36

3-1-4　CSE の記述・・・・・・・・・・・・・・・・・37

3-2　信号線の表記・・・・・・・・・・・・・・・・・・・・・38
3-2-1　回路の名前・・・・・・・・・・・・・・・・・・・・38
3-2-2　接続を記述する関数・・・・・・・・・・・・・・・39
3-2-3　その他の記述方法・・・・・・・・・・・・・・・・40
3-3　記述の拡張性・・・・・・・・・・・・・・・・・・・・41
3-4　構文規則について・・・・・・・・・・・・・・・・・・41

4　今後の展開・・・・・・・・・・・・・・・・・・・・・・43
4-1　本来の目的・・・・・・・・・・・・・・・・・・・・・43
4-2　目的の達成方法・・・・・・・・・・・・・・・・・・・43
4-2-1　回路の文法作成・・・・・・・・・・・・・・・・・43
4-2-2　回路文法からの回路合成方法・・・・・・・・・45

5　結論・・・・・・・・・・・・・・・・・・・・・・・・・49

謝辞・・・・・・・・・・・・・・・・・・・・・・・・・・・50
参考文献・・・・・・・・・・・・・・・・・・・・・・・・・51

1 はじめに

現在、回路設計の規模は飛躍的に大きくなった。そして今後、さらに大きくなっていくことが予想される。レイアウトパターンからの設計、すなわちフルカスタムの設計を回路すべてにおいて行うことは、非常に時間がかかってしまい現実的ではない。このため、高速に設計を行うために、HDL を用いた RTL レベルでの設計が行われている。しかし現状ではさらなる設計期間の短縮化が求められており、この要求に対する結果を出そうと研究が行われている。

LSI の設計では、その計算速度、面積、消費電力が非常に重要視される。しかし現状の設計手順では、設計が終了するまでその回路の面積や速度を測ることはできない。従って、最適化な回路かどうか判断するためには、回路の構成を変化させせることを繰り返し、処理をすることで最適な回路を探していく方法を取らざるを得ない。しかし、期待する面積や、速度の値を入力することで回路を生成することが可能になれば、非常に効率の良い設計方法となる。

回路設計は、通常次の手順で行う。回路の記述⇒合成⇒最適化⇒性能予測である。これを繰り返していくことでデータを蓄積していく。この際、回路合成や最適化の工程は、製造プロセスの違いが大きく影響する。投入される技術により、最適な回路が変化するからである。この工程に必要な期間を短縮するためには、合成から性能予測に至る工程についての期間短縮を行う必要があることは明らかであるが、回路の記述に必要な期間を短縮することにより、多くの構成について性能予測を行うこともできるはずである。

今回、注目したのは回路の記述である。合成、最適化は製造プロセスにあったソフトウェアで行うことができるが、回路の記述は人間によって行わなければならない。そこでソフトウェアで構成と記述を生成することができれば、非常に効率のよい回路の生成が可能である。

今回の研究では、加算回路を効率よく設計するための回路の文法を考える。どのような回路構成でも実現できる文法が必要なため、その文法と構成について研究した。

第二章では今回使用する 4 種類の加算回路、Ripple carry adder、Carry lookahead adder、Carry skip adder、Carry select adder と、それらを組み合わせた Hybrid adder の基本的な構成について説明した。第三章では、加算回路を高速に記述をする方法とその文法について考察した。第四章は、やり遂げることができなかった今後の展開とする。第五章は結論である。

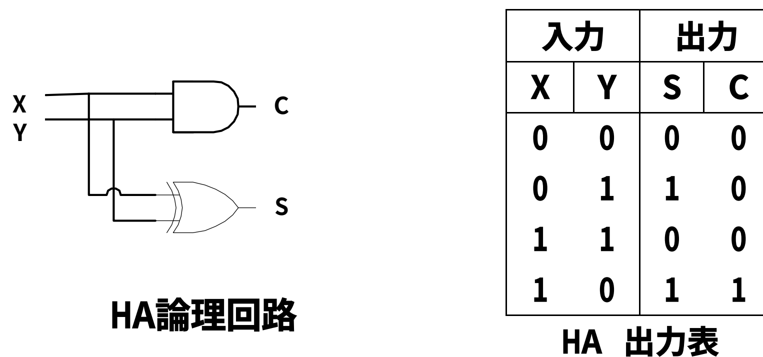
2　加算回路の種類と特徴

2-1　加算回路の種類と特徴

　加算回路は、設計思想の違いによって幾つかの種類がある。また加算器の構成を組み合わせることによって、Hybrid 加算器が構成できる。以下に今回使用する主な加算器の構成を説明していく。

2-2　Half adder と Full adder

　最初に加算回路の一番基礎になる HA（Half Adder）と FA（Full Adder）について説明する。
　HA は入力 A,B に対し出力 Sum（和）と Caryy（桁上げ）を出力する。図 2-1 に論理回路(一例)と出力表を示す。

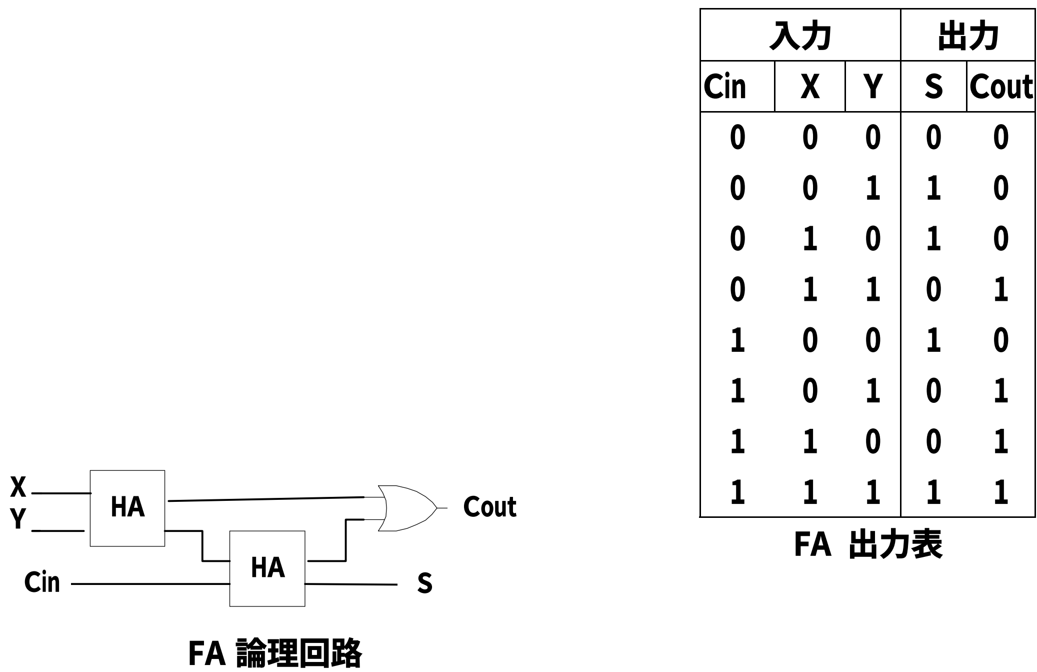


入力		出力	
X	Y	S	C
0	0	0	0
0	1	1	0
1	1	0	0
1	0	1	1

HA論理回路　　HA 出力表

図2-1 HAの例

　FA（Full Adder）は、HA に入力値として Carry In を考慮したものである。演算では Carry が発生する。従って最下位ビットではない場合、下の桁からの桁上げ信号の入力を考慮した演算回路を作らなければならない。
　図 2-2 に論理回路と出力表を示す。



入力			出力	
Cin	X	Y	S	Cout
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

FA 出力表

図2-2 FAの例

これまで説明してきたFA論理回路は現実的ではない。なぜならゲートレベルで考えた場合、ゲート数が非常に多くなるからである。

図2-3に、実際に使われている構成のFA回路図を示す。

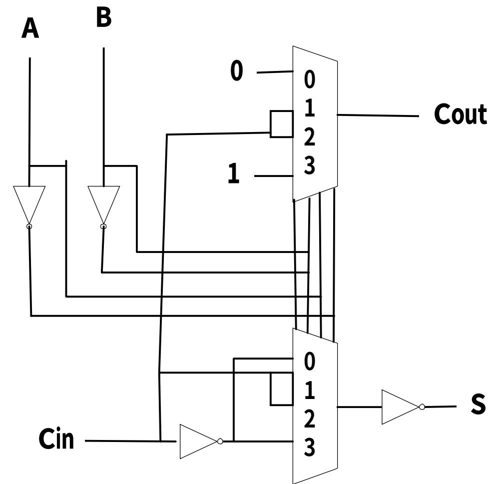


図2-3 MUXを使用したFA

この回路は Multi Plexer(MUX)を使用した FA 回路になる。実際はこのような回路構成となる。

2-3　Ripple carry adder

Ripple　Carry　Adder について説明する。Ripple carry adder は FA 回路を必要ビット数並列につないで作られた加算回路である。回路の構成が非常に単純であるため、面積を非常に小さくすることができる。ただし速度が遅い。

図 2-4 に 64bit 加算回路の構成を示す。

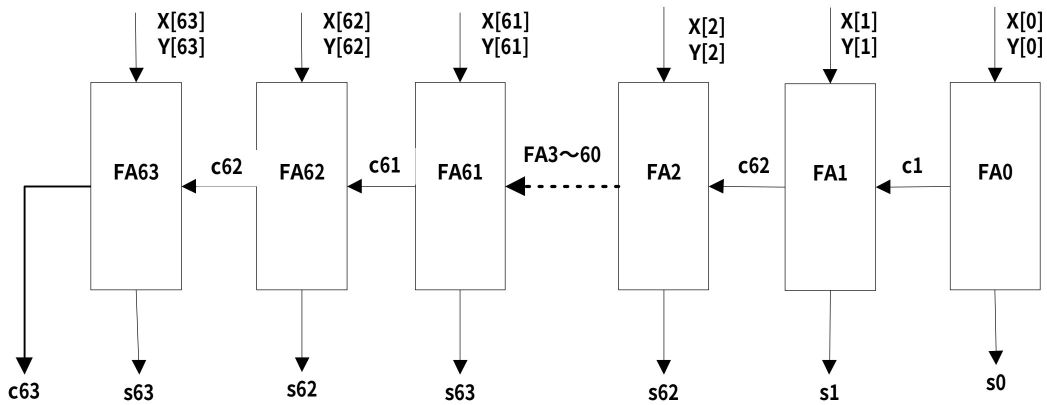


図2-4 Ripple carry adderの構成例

次に Ripple carry adder のクリティカルパス図 2-5 を示す。クリティカルパスの多くは、入力最下位ビットから入った入力値が、最上位の Carry まで通過する場合である。

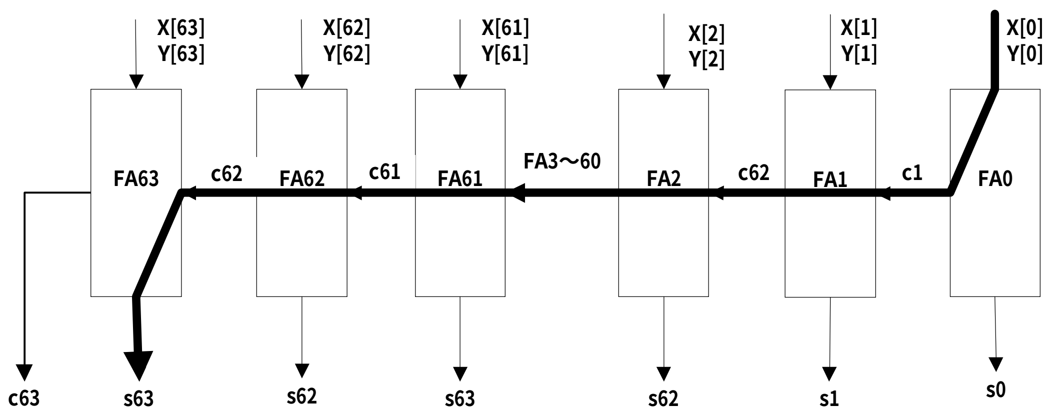


図2-5 クリティカルパスの例

Ripple carry adder の遅延計算は式（2-1）のようになる。

$$T_{ripple} = T_{FA}(x, y \to c_{out}) + (k-2) \times T_{FA}(c_{in} \to c_{out}) + T_{FA}(c_{in} \to s) \qquad (2\text{-}1)$$

図 2-5 の遅延が、同様のビット数の加算回路で見た場合の最大のクリティカルパスとなる。よっていかに遅延を減少させていくか考える。

2-4　Carry lookahead adder

Carry lookahead adder について説明する。加算回路における遅延の原因は、下からの桁上げの加算、すなわち Carry である。下位の計算が終わらない限り、その上位に存在する計算を行うことができない。よって Carry の桁上げによる遅延をどれだけ小さくできるかによって計算速度は大きく変わってくる。図 2-6～図 2-8 に例を示す。

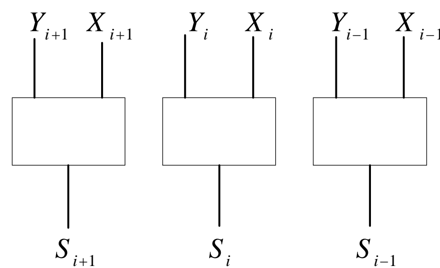


図　2-6 carry free

図 2-6 は 1 ステージで全ての処理が可能なパターンである。しかし実際にはこのような回路は存在しない。なぜなら Carry があるからである。従って図 2-7 ようなパターンとなる。

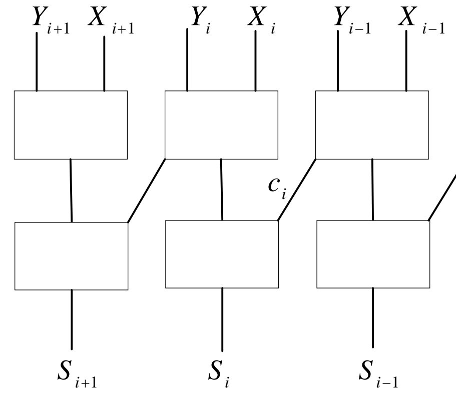


図 2-7 Two-stage carry free

図 2-7 は二段のステージを利用して計算をする。この場合は一段前の carry のみ桁上げが可能となる。この場合は、使用する数字を冗長２進のように一度変換する必要がある。しかしこの方法では、変化させた数字を計算後に元に戻す作業が必要になるので、連続して加算を行うばあいでなければ効率がよくない。

図 2-8 は計算したいビットの前の入力値を考慮することで Carry の発生を考える方法である。

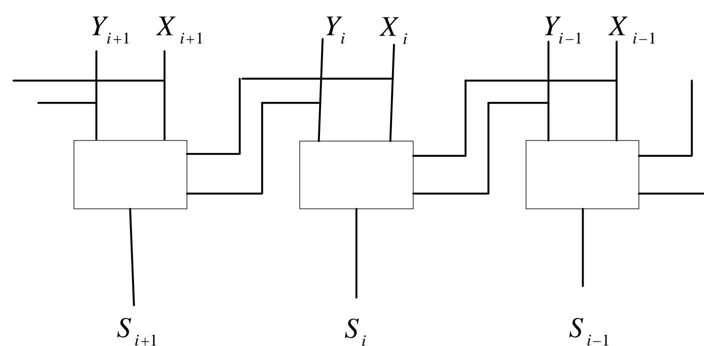


図 2-8 one stage with lookahead

この回路では、あらかじめ計算したい場所より下位の入力を考慮して、Carry を入力するようにしている。これから説明する Carry lookahed adder はこのよ

うな方法で考えていく。すなわち Carry の発生と伝播を一括に生成する方法である。

i ビット目に注目した場合、次の式（2-2）が考えられる。

$$g_i = X_i \cdot Y_i$$
$$p_i = X_i \oplus Y_i \qquad (2\text{-}2)$$

このときの「g」は carry generator、「ｐ」は carry propagator を現す。carry generator は下位の桁からの入力にかかわらず、現在のビットより上位に carry が発生することを示し、また carry propagator は下位からの carry が存在した場合、現在のビット位置より上位に carry を発生することを示す。

この時、それぞれの発生率は式（2-3）のようになる。

carry generation = 1/4
carry propagation = 1/2　　（2-3）
carry annihilation = 1/4

ここで g と p を用いて Sum, S_i と $carry, C_i$ をあらわすと式(2-4)が考えられる。

$$S_i = X_i \oplus Y_i \oplus Ci$$
$$C_{i+1} = g_i + p_i \cdot C_i \qquad (2\text{-}4)$$

この式において $i = n, n-1, n-2 \cdots, 2,1,0$ の漸化式を展開すると式（2-5）のように展開される。

$$C_n = g_n + C_{i-1} p_{n-1}$$
$$= g_{n-1} + (g_{n-2} + C_{n-2} p_{n-2}) p_{n-1}$$
$$= g_{n-1} + g_{n-2} p_{n-1} + C_{n-2} p_{n-2} p_{n-1} \qquad (2\text{-}5)$$
$$= g_{n-1} + g_{n-2} p_{n-1} + g_{n-3} p_{n-2} p_{n-1} + C_{n-3} p_{n-3} p_{n-2} p_{n-1}$$
$$= g_{n-1} + g_{n-2} p_{n-1} + g_{n-3} p_{n-2} p_{n-1} + g_{n-4} p_{n-2} p_{n-1} p_{n-1} + C_{n-4} p_{n-4} p_{n-3} p_{n-2} p_{n-1}$$

このように $C_{in} = C_0$ になるまで繰り返せば C の値を求めることができる。例えば、4 ビット Carry lookahead adder の場合での C の値は

$$
\begin{aligned}
C_1 &= g_0 + C_0 p_0 \\
C_2 &= g_1 + g_0 p_1 + C_0 P_1 P_0 \\
C_3 &= g_2 + g_1 p_2 + g_0 p_1 p_2 + C_0 p_0 p_1 p_2 \\
C_4 &= g_3 + g_2 p_3 + g_1 p_2 p_3 + g_0 p_1 p_2 p_3 + C_0 p_0 p_1 p_2 p_3
\end{aligned}
\qquad (2\text{-}6)
$$

式（2-6）となる。これを求める回路を Full carry lookahead という。この構成を図 2-9 に示す。

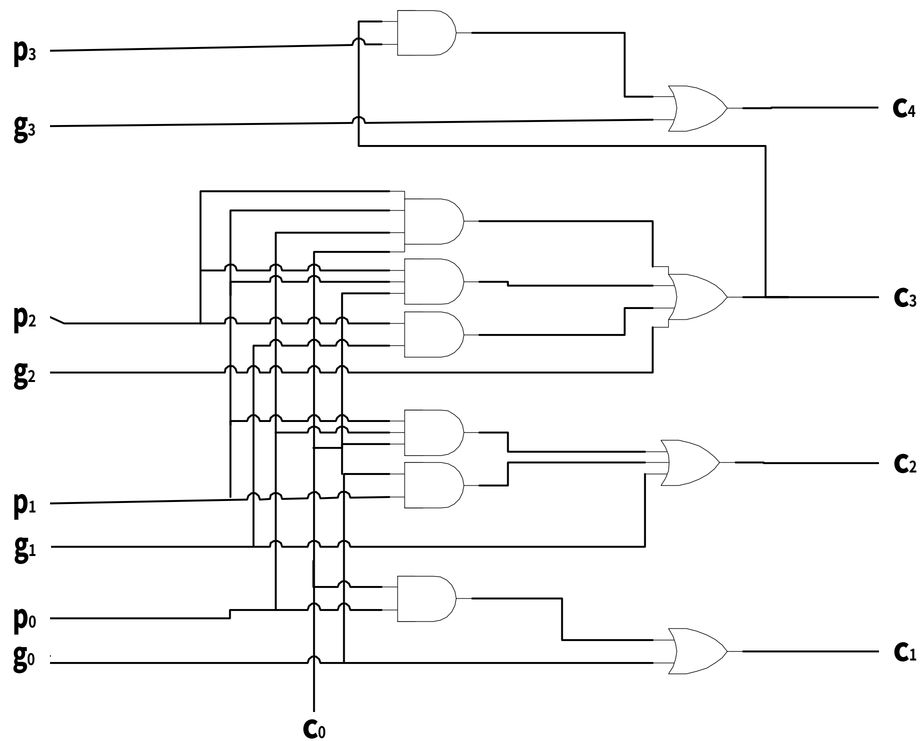


図2-9 4bit Full carry lookahed

この回路を利用して 4 bit Carry lookahead adder を作る。図 2-10 に回路を示す。

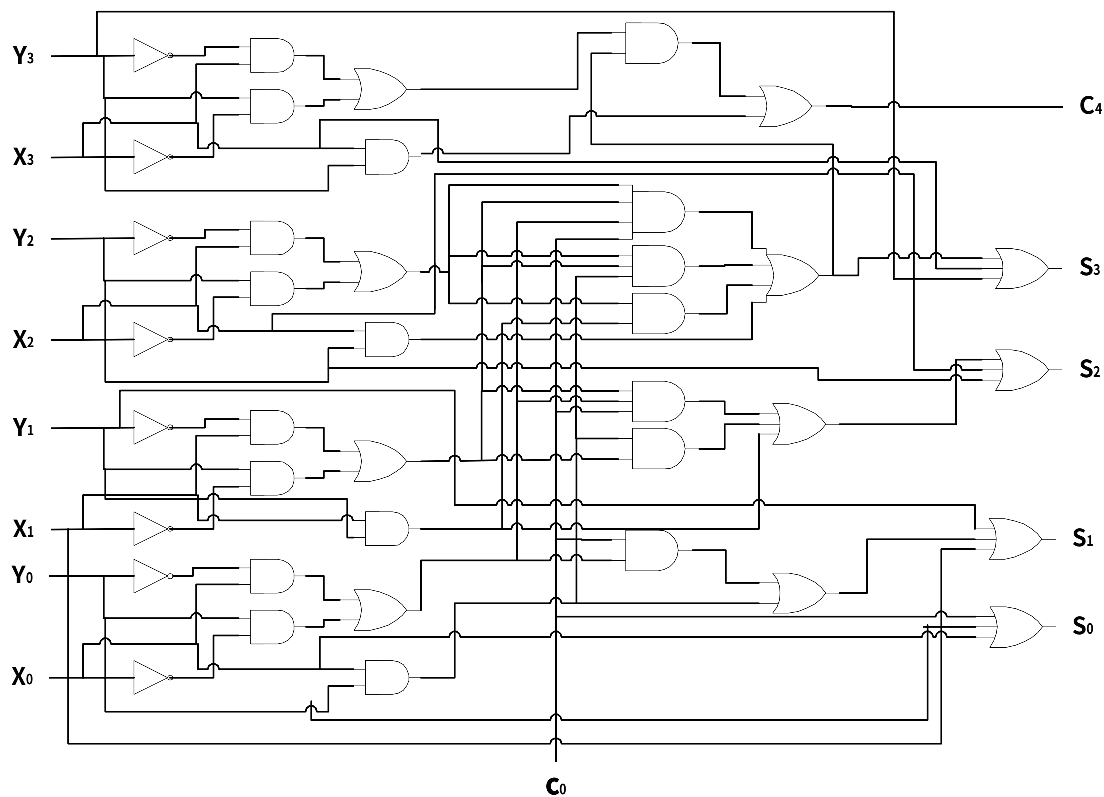


図 2-10 4bit carry lookahead adder

この回路を単純に 16 個並列に並べれば 64bit adder が完成する。しかし、これでは十分に早くなってはいない。

ここで注目を置く場所は g と p である。g と p を n ビット目から n+3 までのブロックとして考え、それぞれ block generate、block propagate とする。このとき、式（2-7）が発生する。

$$g_{[n,n+3]} = g_{n+3} + g +_{n+2}\ p_{n+3} + g_{n+1} p_{n+2} p_{n+3} + g_n p_{n+3} p_{n+2} p_{n+1}$$
$$p_{[n,n+3]} = p_{n+3} p_{n+2} p_{n+1} p_n \quad (2\text{-}7)$$

この時、16bit の塊について考えてみる。先ほど使った式（2-7）と

$$C_4 = g_3 + g_2 p_3 + g_1 p_2 p_3 + g_0 p_1 p_2 p_3 + C_0 p_0 p_1 p_2 p_3 \quad (2\text{-}8)$$

について比較すると、

$g_{[n,n+3]}$の

$$g_{n+3} + g +_{n+2}\ p_{n+3} + g_{n+1} p_{n+2} p_{n+3} \tag{2-9}$$

がC_4の

$$g_3 + g_2 p_3 + g_1 p_2 p_3 + g_0 p_1 p_2 p_3 \tag{2-10}$$

に対応し、 $p_{[n,n+3]}$の

$$p_{n+3} p_{n+2} p_{n+1} p_n \tag{2-11}$$

の部分が

$$p_0 p_1 p_2 p_3 \tag{2-12}$$

に対応している事がわかる。つまり、これらの部分を n から n+3 の 4bit の塊として考えていくと、16bit の carry generator、carry propagator として考えることができる。式であらわすと式（2-13）ようになる。

$$C_{16} = g_{[12,15]} + g_{[8,11]} p_{[12,15]} + g_{[4,7]} p_{[8,11]} p_{[12,15]} + g_{[0,3]} p_{[4,7]} p_{[8,11]} p_{[12,15]} + C_0 p_{[0,3]} p_{[4,7]} p_{[8,11]} p_{[12,15]} \tag{2-13}$$

図 2-11 は full carry lookahead に$g_{[n,n+3]}$と$p_{[n,n+3]}$の出力を組み合わせた 4 bit lookahead carry generator である。

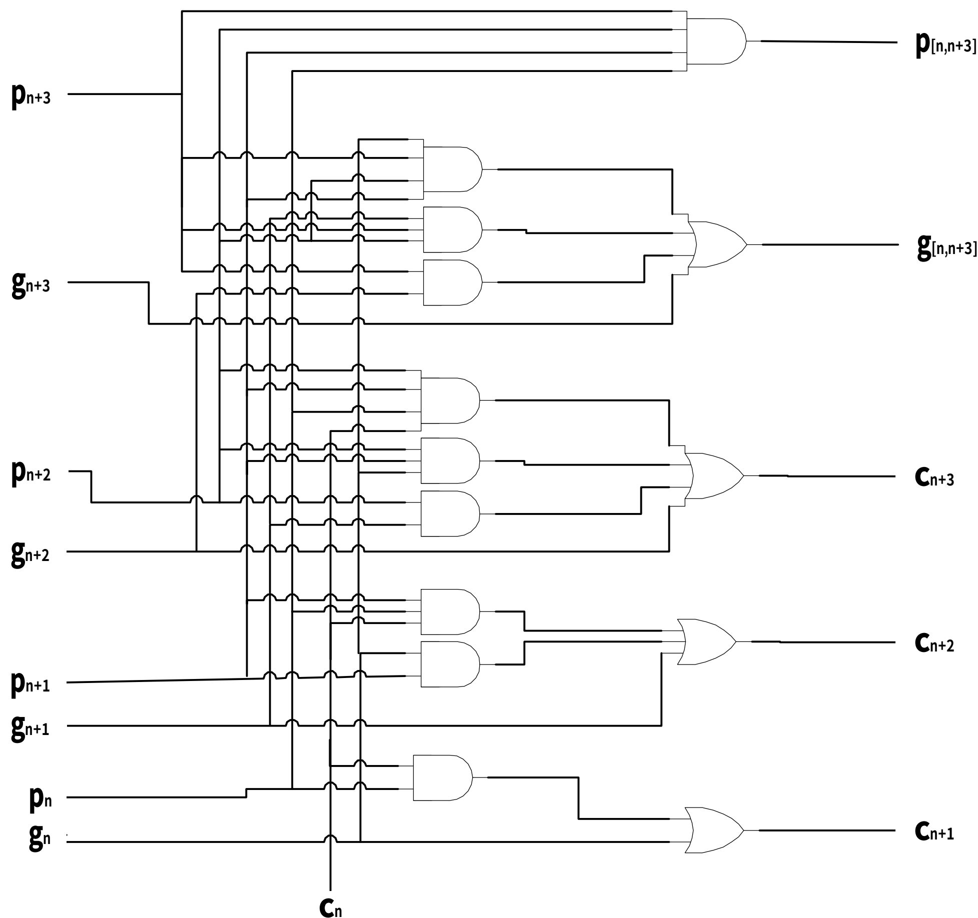


図2-11 4bit carry lookahead generator

この回路を利用して、16bit の carry generator、carry propagator を作り出すことができる。

ここで、64bit などの加算回路を構成する際に、全体を適当な bit 単位に分割して、それぞれの単位ごとに carry generator、carry propagator の考えに基づいて論理を構成することでさらに高速化が可能となる。

同様に考えていくと、64bit adder は次のように考えていくことができる。

上記で説明したとおり、

$$C_4 = g_3 + g_2 p_3 + g_1 p_2 p_3 + g_0 p_1 p_2 p_3 + C_0 p_0 p_1 p_2 p_3 \quad (2\text{-}14)$$

という式が発生するからであるから、C_{16}を表す式は、

$$g_{[n,n+3]} = g_{n+3} + g +_{n+2}\ p_{n+3} + g_{n+1} p_{n+2} p_{n+3} + g_n p_{n+3} p_{n+2} p_{n+1}$$
$$p_{[n,n+3]} = p_{n+3} p_{n+2} p_{n+1} p_n \qquad (2\text{-}15)$$

になる。この式への対応を考えて

$$C_{16} = g_{[12,15]} + g_{[8,11]} p_{[12,15]} + g_{[4,7]} p_{[8,11]} p_{[12,15]} + g_{[0,3]} p_{[4,7]} p_{[8,11]} p_{[12,15]} + C_0 p_{[0,3]} p_{[4,7]} p_{[8,11]} p_{[12,15]} \qquad (2\text{-}16)$$

上記の式が発生する。

今度はこの 16bit を一塊と考えて、

$$g_{[n,n+15]} = g_{[n+12,n+15]} + g_{[n+8,n+11]} p_{[n+12,n+15]} + g_{[n+4,n+7]} p_{[n+12,n+15]} p_{[n+8,n+11]}$$
$$+ g_{[n,n+3]} p_{[n+12,n+15]} p_{[n+8,n+11]} p_{[n+4,n+7]} \qquad (2\text{-}17)$$
$$p_{[n,n+15]} = p_{[n+12,n+15]} p_{[n+8,n+11]} p_{[n+4,n+7]} p_{[n,n+3]}$$

となるから、C の値はそれぞれ

$$C_{16} = g_{[0,15]} + C_0 p_{[0,15]}$$
$$C_{32} = g_{[16,31]} + g_{[0,15]} p_{[16,31]} + C_0 p_{[16,31]} p_{[0,15]}$$
$$C_{48} = g_{[32,47]} + g_{[16,31]} p_{[32,47]} + g_{[0,15]} p_{[32,47]} p_{[16,31]} + C_0 p_{[32,47]} p_{[16,31]} p_{[0,15]}$$
$$C_{64} = g_{[48,63]} + g_{[32,47]} p_{[48,63]} + g_{[16,31]} p_{[48,63]} p_{[32,47]} + g_{[0,15]} p_{[48,63]} p_{[32,47]} p_{[16,31]}$$
$$+ C_{n-4} p_{[48,63]} p_{[32,47]} p_{[16,31]} p_{[0,15]} \qquad (2\text{-}18)$$

となる。

図 2-12 はこれらの方法で作った Carry lookahead generator である。

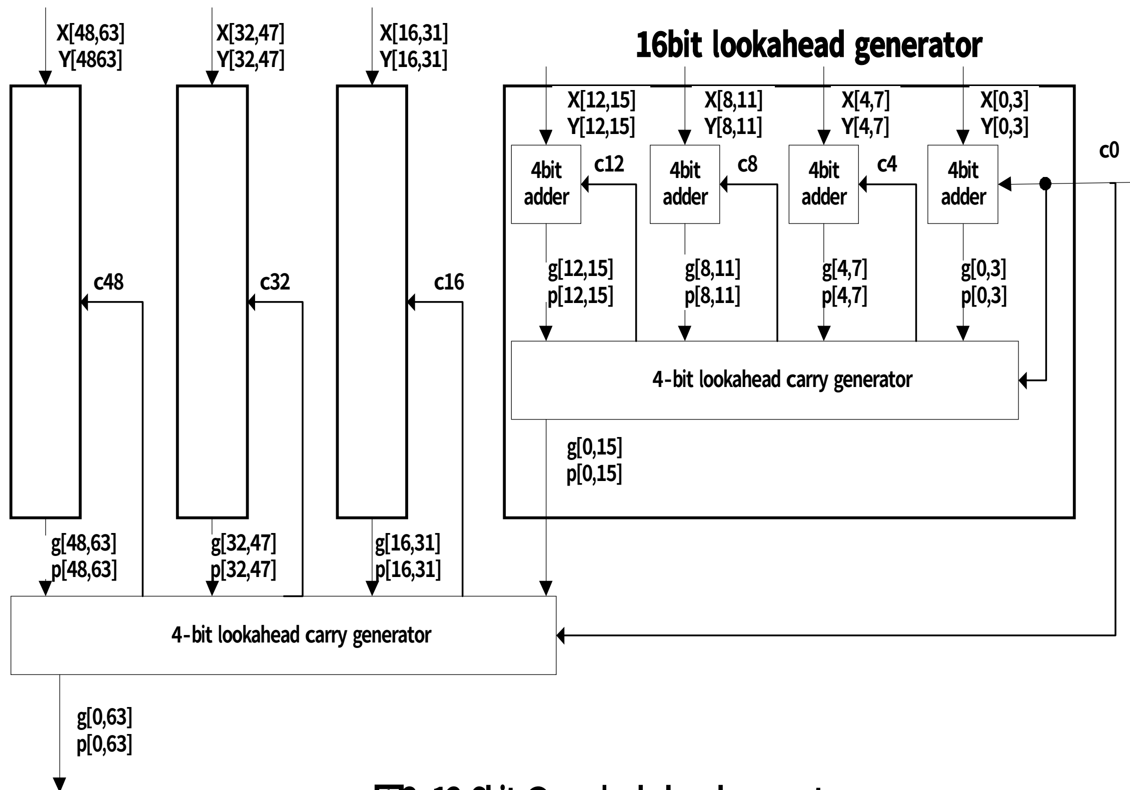


図2-12 6bit Carry lookahead generator

この方法で計算していくと、わずか 13 ゲートレベルで計算が可能である。ただし、計算ビット数が増えていくと必要な回路も増加していくため、面積が増加し、遅延も発生する。

今回、4bit を一塊として使用することにはわけがある。例えば 2bit や 3bit を一塊にしても良いのだが、段数が多くなってしまう上、面積も大きくなってしまうためあまりメリットがない。17bit 演算など、4ｎの端数が存在する場合に使える。逆に 5bit、6bit など、塊を大きくする方法もあるが、回路の電子回路的な制約があり、あまり大きくすることができない。場合によっては段数が減るメリットがあるが、回路の制約条件によって変わってくる。

ここで、次のような方法も考えられる。

$$p = p'p'' \qquad g = g' + g''p' \tag{2-19}$$

式（2-19）が実現できるので、それぞれの g, p の値について、次の図 2-13 のよ

うにまとめることができる。

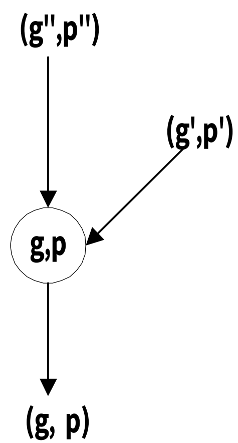


図2-13 (g、p) の生成

これを組み合わせていくことによって、g_i, p_i の値を求めることができる。図2-14 は Breng-Kung による並列先見グラフである。

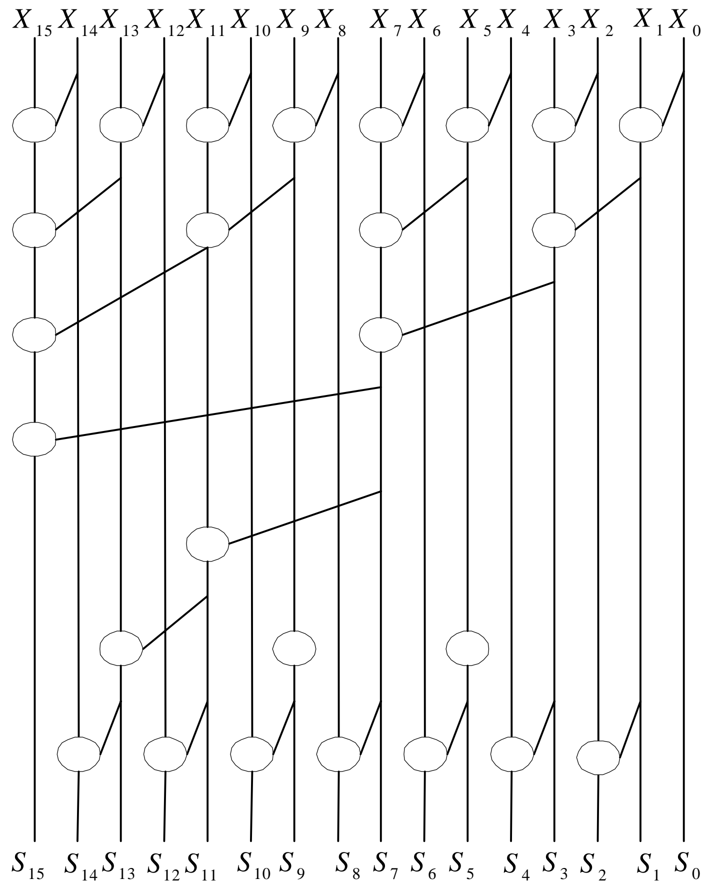


図　2-14 Breng-Kungによる並列先見グラフ

他にも、Kogge-Stone 並列グラフや、その他の組み合わせが存在する。

2-5　Carry skip adder

Carry skip adder について説明する。図 2-15 のような４ビットの塊であらわした Ripple carry adder があるとする。この時のあるステージｎからｎ+3 の範囲について考えてみる。仮に４つのステージ全てから propagate の信号が発生した場合、この回路の計算結果を待たずに次の５番目のステージに carry を送ることが可能となる。従ってその信号は

$$p_{[n,n+3]} = p_n p_{n+1} p_{n+2} p_{n+3} \quad (2\text{-}20)$$

式（2-20）のようにすることができる。すなわちそれぞれの propagate 信号の論理積をとればよい。このような方法を Carry skip adder という。

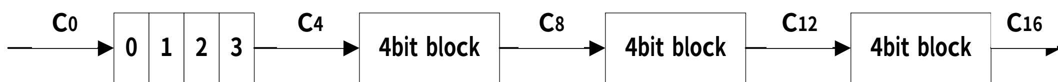


図2-15 Ripple carry adder

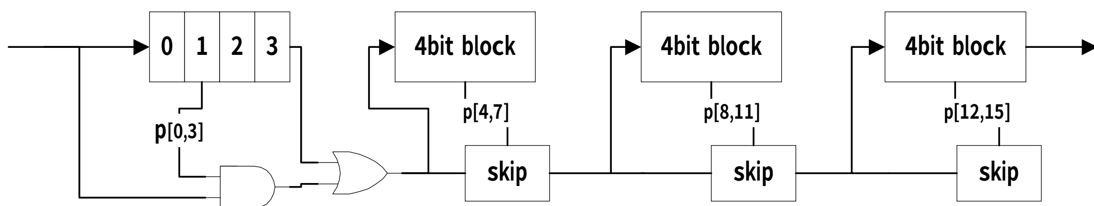


図2-16 Simple carry skip adder

ここで気になるのはクリティカルパスだが、Carry の発生条件次第では通常の Ripple carry adder と速度が変化しない場合もある。ここで図 1-16 の Carry skip adder において 0bit 目で carry が発生するとする。そして Ripple carry adder の 1-3 ビット目を通過した後、OR ゲートを抜ける。次に中間の二段目、三段目の skip adder を通り、最後にラストのブロックの 12 から 15 ビット目を抜けることとなる。このとき、OR を 0.5 ステージに値すると考えると、Ripple carry adder の動作が 16 ステージかかるのに対し、Carry skip adder は 8.5 ゲートで計算を終了することができる。

前述の例で示したような、１ブロックをｂ bit とした場合、k bits carry skip adder における worst case carry propagation による遅延は、１つの skip block を 1bit ripple carry adder として対応させる。そのとき遅延 $dT_{fixed-skip-adder}$ は

$$T_{fixed-skip-adder} = (b-1) + 0.5 + (k/b-2) + (b-1) \quad (2\text{-}21)$$
$$\approx 2b + k/b - 3.5$$

ということになる。

この時、最適なブロックのサイズは$dT_{fixed-skip-adder}/db$＝0 となる方程式よって導き出すことができる。

$$\frac{dT_{fixed-skip-adder}}{db} = 2 - \frac{k}{b^2} = 0 \quad (2\text{-}22)$$
$$b = \sqrt{k/2}$$

これにより、最適なブロックサイズである Adder の遅延の上限は、式（2-22）を利用することで

$$T_{fixed-skip-adder} = 2\sqrt{k/2} + \frac{k}{\sqrt{k/2}} - 3.5 = 2\sqrt{2k} - 3.5 \quad (2\text{-}23)$$

式（2-23）を導くことができる。

この式を利用すると、例えば 32bit の carry skip adder において b＝4 となるから、$dT_{fixed-skip-adder} = 12.5$となる。これを 32bit ripple carry adder と比較すると、約 2.6 倍の propagation delay の時間差が生まれることになる。

これまでは propagation delay しか考えてこなかった。しかし実際の計算では carry の生成や消滅もある。従って adder による delay を増加させること無く、より大きい ripple stage を作ることができる。すなわち、様々な大きさのスキップブロックを利用した carry skip adder ができる。

ブロックサイズの決め方について考えてみる。まず、図 2-17 のようにブロックサイズをそれぞれ$b_0, b_{1,} \cdots b_{t-1}$と置く。carry path １と carry path 2 について考えて見ると、どちらもブロックb_0からはじまり、終了はそれぞれb_{t-1}とb_{t-2}のブロックとなっている。

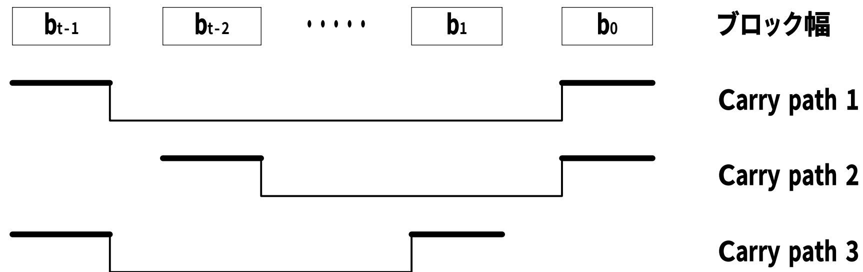


図 2-17 Carryパスの例

見たとおり carry path 2 は、carry path 1 より 1 スキップ小さい。従って 1 スキップ＝1 ripple carry adder とみなすことができる。よってブロックのビット数はb_{t-2}よりb_{t-1}を 1 ビット大きくすることが可能である。同様に carry path 1 と carry path 3 を比べてみる。同様に考えていくと、ブロックのビット数はb_0よりb_1を 1 ビット大きくすることが可能である。ここで$b_0 = b_{t-1} = b$、ブロック数ｔが偶数であるとすると、最適なブロックの幅は、式（2-24）のようになる。

$$b \quad b+1 \quad \cdots \quad \frac{b+1}{2}-1 \quad \frac{b+1}{2}-1 \quad \cdots \quad b+1 \quad b \tag{2-24}$$

最初に説明した、$b_0 = b_{t-1} = b$であるが、これは正しいことが説明できる。なぜならトータルの遅延は個々の遅延の値というより$b_0 + b_{t-1}$といえるからである。

ここで、ブロック数ｔのトータル bit 数は、

$$\begin{aligned} &2\left[b + (b+1) + \cdots + (\frac{b+t}{2} - 1)\right] \\ &= t(b + \frac{t}{4} - \frac{1}{2}) \end{aligned} \tag{2-25}$$

となる。ここでトータル数をｋとして、ｂについて解くと、

$$b = \frac{k}{t} + \frac{t}{4} + \frac{1}{2} \tag{2-26}$$

この式が成り立つ。ここで前述で使用した

$$T_{fixed-skip-adder} = (b-1) + 0.5 + (k/b-2) + (b-1) \qquad (2\text{-}27)$$

を利用すると、$T_{\mathrm{var}-skip-adder}$ の値は、

$$\begin{aligned} T_{\mathrm{var}-skip-adder} &= 2(b-1) + 0.5 + t - 2 \\ &= \frac{2k}{t} + \frac{t}{2} - 2.5 \end{aligned} \qquad (2\text{-}28)$$

となり、最適なブロック数は、

$$\begin{aligned} \frac{dT_{\mathrm{var}-skip-adder}}{dt} &= \frac{-2k}{t^2} + \frac{1}{2} = 0 \\ t &= 2\sqrt{k} \end{aligned} \qquad (2\text{-}29)$$

となる。ここで最適な遅延の上限は

$$T_{\mathrm{var}-skip-add} \approx 2\sqrt{k} - 2.5 \qquad (2\text{-}30)$$

である。これにより、最適なブロックサイズである Adder の遅延の上限が fixed-size skip adder より約$\sqrt{2}$だけ小さくなることがわかる。この方法によって考えた carry skip adder を図 2-18 に示す。

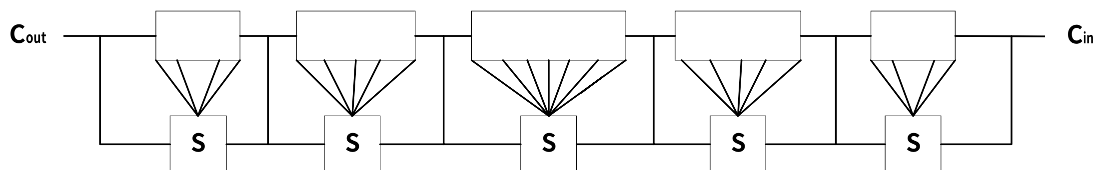


図2-18　1段階のスキップを考慮したCaryy skip adder

ここでは、ブロックサイズの最適な大きさをどのように決めるか考える。例えば、g_i, p_i,X-1 段目から X 段目までのスキップ信号、スキップ、結果の和の計算がそれぞれ１ユニット分の時間がかかるとする。この条件下において、８ユニット以内の最大 bit 数の carry skip adder を作るとする。

まず、最初の p とその伝播で 2unit 使う。それぞれ 1 スキップ１ユニットであるから、最後のブロックをのぞいて数字を割り振っていくと

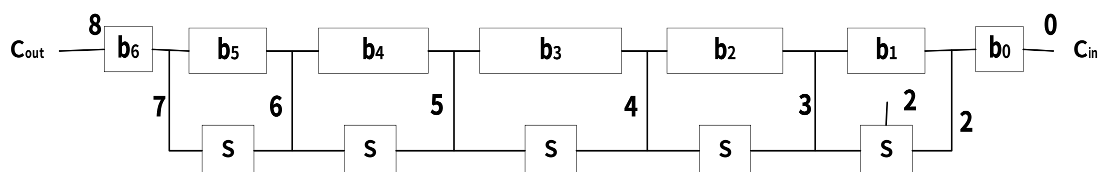


図 2-19 遅延を8とした場合の、Carry skip adder

図 2-19 のような図ができあがる。（Cin の段階では０として考える）。次に各ブロックサイズについて考えていく。まず、ブロックの右側の制限はブロックの出力タイミングによって決まってくる。例えば,b_1 について考えてみる。出力制限の上限が３ユニットであるので、３bit を超える幅のブロックにすることはできない。ただし、b_0 の値は入力 Cin への対応があるので、－１して、1bit になる。

次に左側のブロックの制限であるが、これは入力タイミングによって決まる。b_1 のブロックについて考えてみる。最大のユニット数が８だから、すでに入力にすでに 2 使われているので 6 を超えるユニット数にすることはできない。すなわち、この b_1 のブロックの上限は３ということになる。

これに基づいて、それぞれのブロックの最大 bit 数を考えていくと、右側から

$$1+3+4+4+3+2+1=18 \tag{2-31}$$

という結果が導き出される。

これまでは、一段階のみの carry skip について考えてきた。しかし、さらに

高速化するためには多段階の carry skip について考慮する必要がある。次に二段階の carry skip adder の例を図 2-20 に示す。

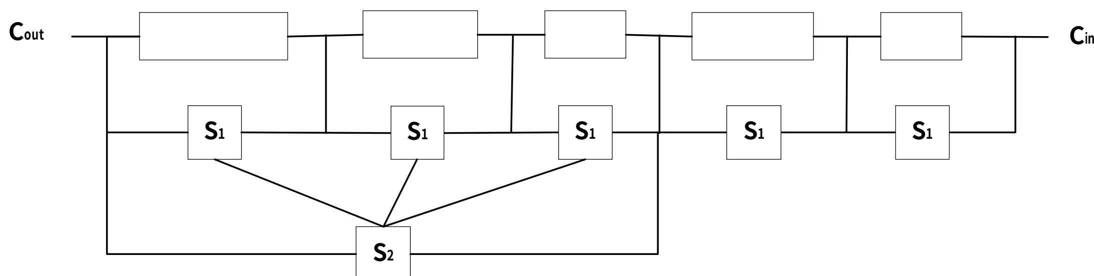


図2-20 2段階のCarry skip adder

この回路は、後ろ側３ブロックを二段目のスキップ回路によってスキップするように設計されている。二段目の skip 回路は一段目の skip 回路の AND をとることで動作する。単純に比較した場合、３ユニット分速い速度で動作することが可能である。

ここで問題となってくるのは、Carry skip adder において、その adder 回路が短かった場合、またスキップの動作出力を考えると、逆に遅くなってしまい、その skip の効力が非常に小さいものとなってしまう。そこで効果があまり得られなかったり、効果がない skip 回路は取り除いてしまう。次にその結果を示す。

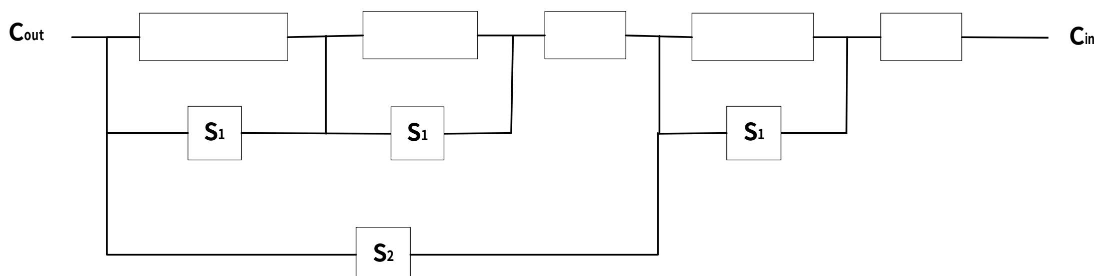


図2-21 効果が薄いskipを取り除いたCarry skip adder

ここで skip と adder による遅延が、同等のものである場合、skip コントロー

ル信号を出力する遅延時間を考慮すると、skip 回路は 1bit ないし、2bit 小さくする必要がでてくるので注意する。

それでは最適な 2 段階の Carry save adder はどうなるか考えてみる。条件は 1 段階の carry skip adder と同様として、８ユニットでの動作について考える。

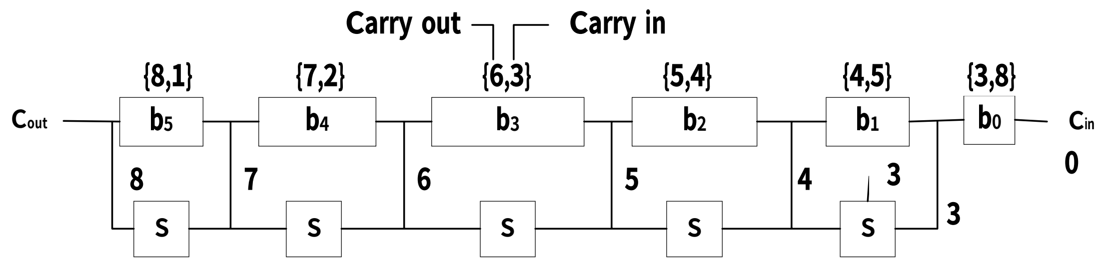


図2-22　二段目を考慮に入れたCarry skip adder

まず、二段階目の skip について考えていく。ここにある skip 回路が二段目の回路である場合、それぞれの入出力タイミングを考えると、一段目のブロック（サブブロック）の数は、それぞれの入出力の遅延、{Carry in = α、Carry out = β} を利用して、

$$\gamma = \min\{\alpha - 1、\beta\} \quad (2\text{-}32)$$

となる。この条件で、１レベルのみで行った考え方を利用すると数が出てくる。次に i 番目のサブブロックのサイズであるが、$0 \le i \le \gamma - 1$ の条件において、

$$\min(\beta - \gamma + i + 1, \alpha - 1) \quad (2\text{-}33)$$

が導き出される。これにより、それぞれのブロックのサイズが決まる。

例として、b_2 のブロックについて考えてみる。{Carry in ＝5、Carry out ＝ 4} だから、 γ= {4,4} となるので、ブロック数は 4 となる。

次にその中で 2 番目のブロックについて考えてみると min（5-4+2+1、4-1）＝（４、3）という結果になり、解答は３ブロックになる。結果を図 1-23 と表 1-1 にまとめてある。

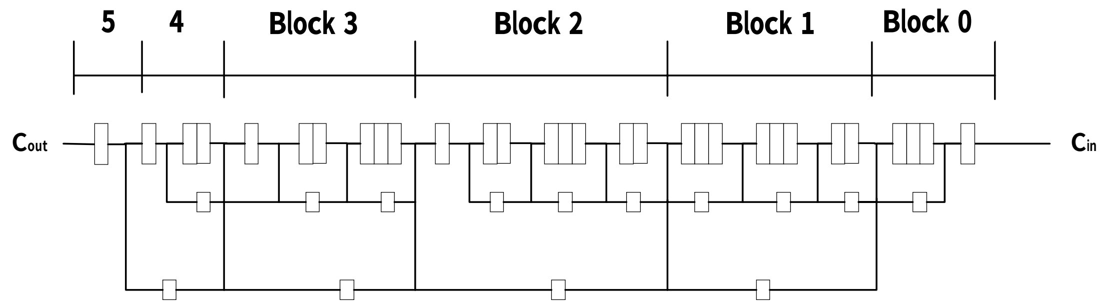


図2-23 最適化された二段Carry skip adder

表1-1 最適化されたブロックの一覧

BLOCK	Carry In	Carry out	サブブロック数	サブブロック bit 幅	ブロック幅
0	3	8	2	1,3	4
1	4	5	3	2,3,3	8
2	5	4	4	2,3,2,1	8
3	6	3	3	3,2,1	6
4	7	2	2	2,1	3
5	8	1	1	1	1

2-6　Carry select adder

Carry select adder について説明をする。Carry select adder を構成する場合、幾つかの加算器と、selector が必要となってくる。そのため Carry select adder の概念もあるが、個々の加算器や selector の速度によって全体の速度が変化してくる。従ってここでは Carry select adder の概念についてのみ考察すること

にする。

carry select adder の概念は、あらかじめ下位からの carry を 0 または１として計算しておき、実際の桁上げされてきた carry の値によって、計算しておいた値を選択する、というものである。

まず、１段階だけの carry select adder について考えてみる。ｋbit adder の場合、ｋ/2 bit の adder が 3 個と 1 つの selector が必要となる。上位ｋ/2bit として、2 つの adder にそれぞれ 0,1 の値を入力し計算する。同時に下位 k/2 bit を計算する。そして下位 bit の carry の出力によって、あらかじめ carry として 0 と１入力しておいた計算結果を select する。図 2-24 に例を示す。

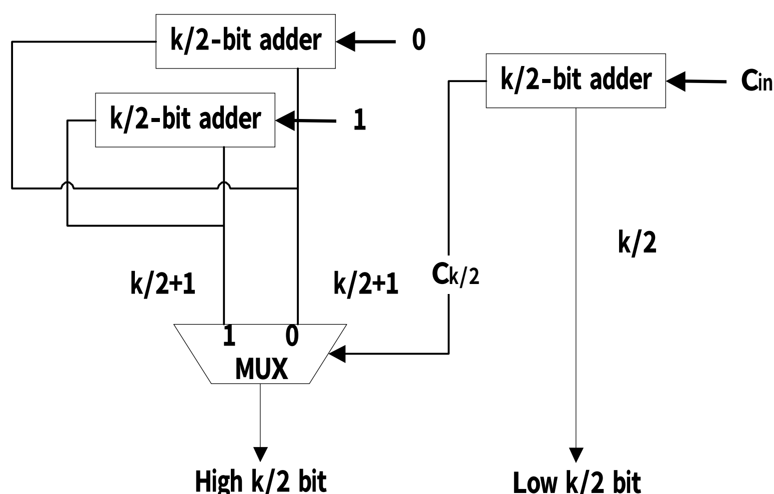


図2-24 k/2ビットを組み合わせたkビットCarry select adder

必要であるコストと遅延について考えてみる。Ripple carry adder で同様の k bit adder を作る場合のコストと遅延をそれぞれ C（k）,T(k)と置くと、Carry select adder における C_{select}, T_{select} の値は、式（2-34）のようになる。

$$
\begin{aligned}
C_{select} &= 3C(k/2) + k/2 + 1 \\
T_{select} &= T(k/2) + 1
\end{aligned}
\qquad (2\text{-}34)
$$

もちろん、ここで使用するｋ/2 adder は Ripple carry adder であるから、Carry

lookahead adder を使用すれば、違ったコストと遅延になる。

次に考えられる方法は、同じビット数の Adder を組み合わせた Adder を、異なる bit 数の Adder と組み合わせる方法である。すなわち $(a+b)=k$ となる a、ｂのビット数の Adder を用意し、b+1 の multiplexer を用意すればよい。

他のパターンは多段階の carry select adder である。図 2-24 は二段階の carry select adder を描いた。これは加算結果と carry をｋ/4 adder を用いて作ったものである。

動作手順としては、それぞれのｋ/４bit adder の計算をした後、一段目の３個の multiplexer で select する。$C_{k/2}$ ビット目は、二段目の multiplexer の select に使用する。二段目の multiplexer は $C_{k/2}$ の入力によってｋ/2 bit 幅の bit を select する必要がある。

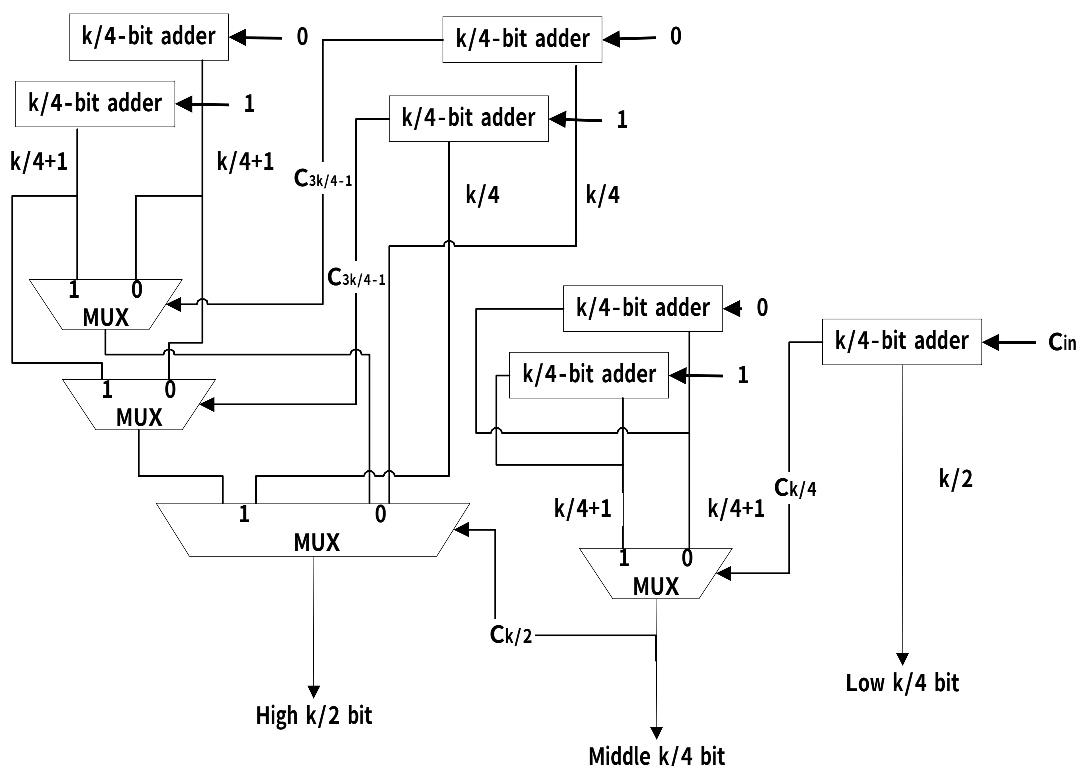


図2-25 k/4bit Carry select adder

Carry select adder は carry lookahead adder と同様、pipeline 化することが容易である。しかし、adder を多段化していく上でそのベストの選択を取ること

を考えなければならない。また最終的にはｋ/ｘとした時の xbit adder や、multiplexer にそのパフォーマンスは左右されてしまう。

2-7 Hyblid adder について

これまで述べてきた、加算回路はそれぞれ特徴があった。これを組み合わせることで、よりバランスの良い、Hybrid な加算器を作ることができる。

例えば、Carry lookahead generator のｇ、ｐの値を素早く計算するために、あらかじめ計算しておいて、それを Carry select adder によって出力するという方法がある。すなわち Carry lookahead /carry select adder である。図 2-26 に例を示す。

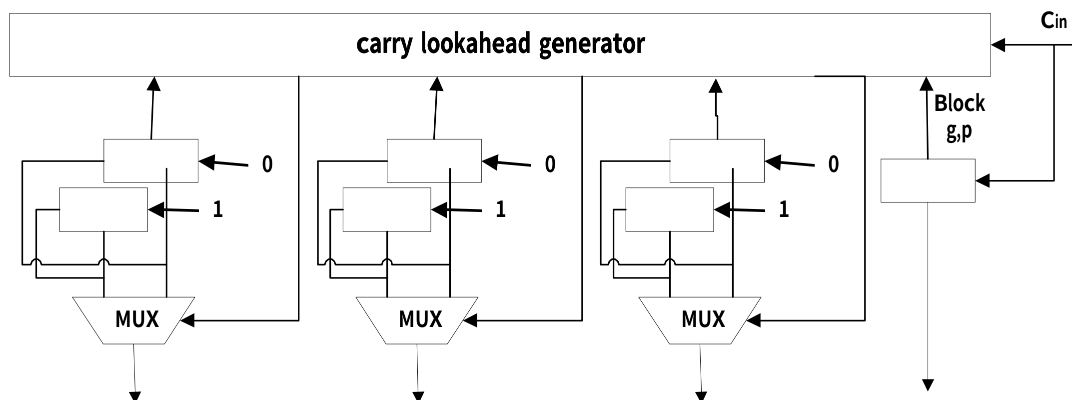


図2-26 Hybrid carry lookahead/carry select adder

次に Ripple carry adder と Carry lookahead の組み合わせについて考えてみる。carry lookahead adder だけでは面積が非常に大きくなるために、ripple carry の carry 伝達を組み合わせる。Carry lookahead generator /Ripple carry adder
である。

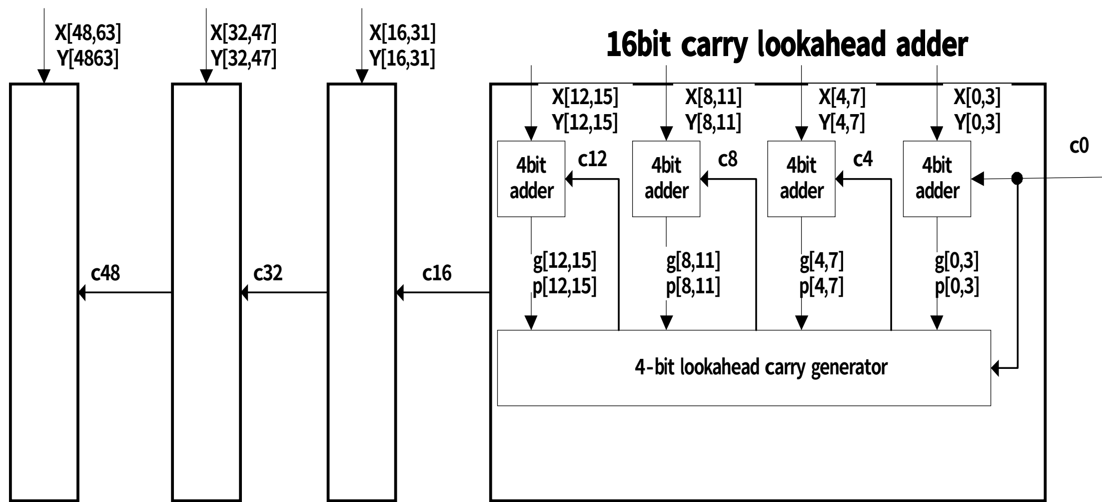


図2-27 Hybrid carry lookahad/ripple carry adder

このように組み合わせ方によって、様々な種類のadderを作ることができる。

3　加算回路の記述

3-1　文法の構成

　この章では、加算回路の記述について述べていく。加算回路の記述として重要なことは、あらゆる Adder が記述可能であるということと、この記述をコンピュータがどれだけパーサーしやすいか、ということである。今回の研究では、二章で述べた４種類の加算器の記述を可能とすることを目標とした。
　この文法の構成は大きく分けて、回路の構造を現す部分と、信号線を現す部分に分けることができる。まず、構造を現す部分について説明する。

例題を使って細かく文法の構成を説明していく。

例　RCA（８）

　RCA（Ripple carry adder）が関数本体となり、「(　)」は内部の構造を示す。この例の場合、図 3-1 のような接続構造が出来あがる。

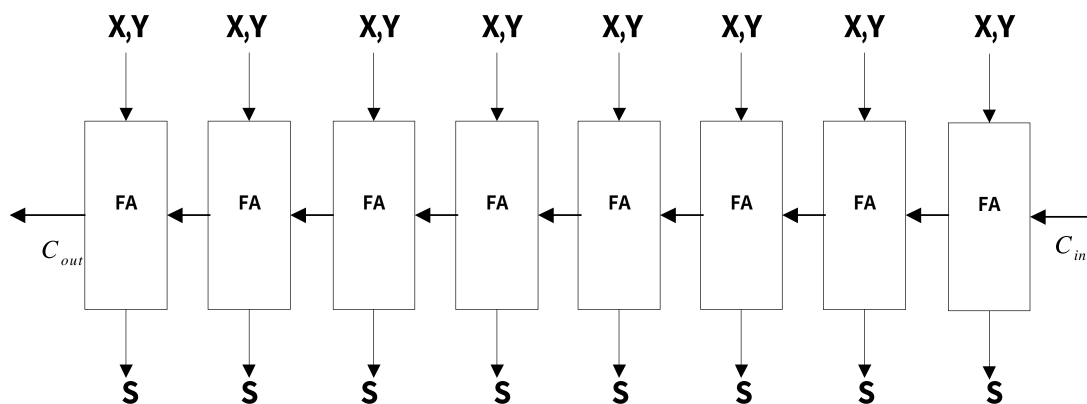


図3-1 RCA (8) の構造

　RCA という関数が使われた時点で Cin や Cout などの入力、出力の構造が確定するために、入出力に関して特に指示は行わない。(　) の内部は、内部に加

算器の指定がない限り FA が基本構造として使われる。

（　）内部の数字はその関数が何ビットの加算器であるかを示している。この場合,指示がないので８個の FA が使われることになる。

内部に関数がある場合,その関数をその外側の関数が構造どおりに接続する。

例　RCA（12 RCA(8) RCA(4)）

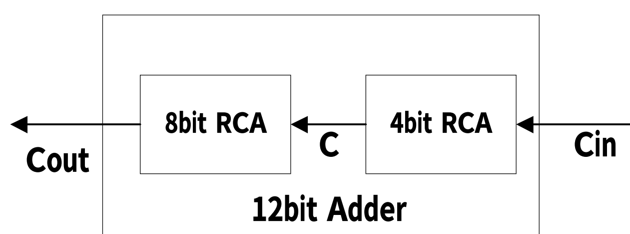


図3-2 回路の内部構造

この例題の場合は、図 3-2 のように内部の加算器の Carry が Ripple carry の構造として接続されていく。一番外側の関数ビット数の値が、内部で使われている関数のビット数の合計値と合致しない場合,エラーとなるので注意する。また、内部で使用される信号が足りない場合等もエラーとなるので注意しなければならない。それぞれの関数で使用される信号の一覧を表 3-1 に示す。

表3-1 加算器の入出力表

	RCA	CLG	FCLG	CLA	CSK
Xin	○	×	×	○	○
Yin	○	×	×	○	○
Cin	○	○	○	○	○
Cout	○	○	○	○	○
S	○	×	×	○	○

以下に構造を現す関数で使用するキーワードを示す。それぞれ、・・・の後の内容が加算回路の種類を示す。

RCA・・・Ripple carry adder
CLA・・・Carry lookahead adder
CLG・・・Carry lookahead generator
FCLG・・・Full carry lookahead generator
CSK・・・Carry skip adder
CSE・・・Carry select adder

これだけのキーワードがあれば Ripple carry adder、Carry skip adder、Carry select adder、Carry lookahead adder の Hybrid 加算器の生成は可能となる。

3-1-1　RCA の記述

記述例

RCA (8)　　　　(3-1)

これは 8 ビットの Ripple carry adder という意味がある。実際の内部構造は図 3-1 のような構成になっているが、実際に必要とする Input/Output は図 3-2 のような例になる。

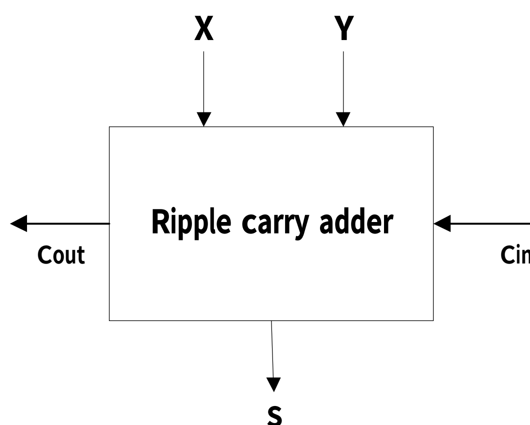


図3-3 実際のInput/Outputの例

3-1-2　CLA の記述

Carry lookahead adder を作りたい場合は、まず CLA の記述に必要である Full carry lookahead generator を生成する。

記述例

FCLG(4)(1 2 3 4))　　　　　　　　　　　　　　　　　　　　（3-2）

これで 4 ビットの Carry lookahead generator が完成である。さらに Carry lookahead adder が作りたい場合、Ripple carry adder が必要となるので次のような記述になる。

記述例

CLA(4 FCLG(4 (1 2 3 4)) RCA(4))　　　　　　　　　　　　　　（3-3）

これで 4 ビットの Carry lookahead adder が完成となる。
図 3-3 に例を示してみる。

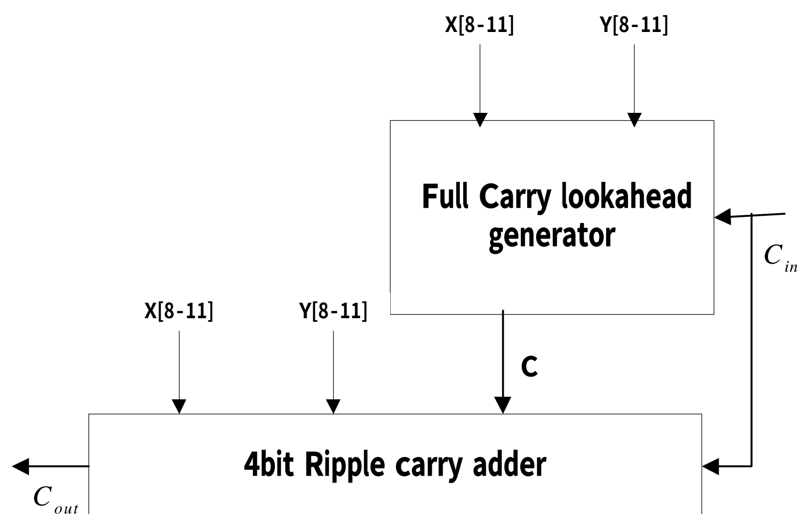


図3-4 Full carry lookahead generatorとRipple carry adderの接続例

次に 16bit の Carry lookahead adder の場合を考えてみる。例として図 3-5 を示す。

記述例

CLA(16 FCLG(16 (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16) CLG(4) CLG(4) CLG(4) CLG(4) RCA(16)))

（3-4）

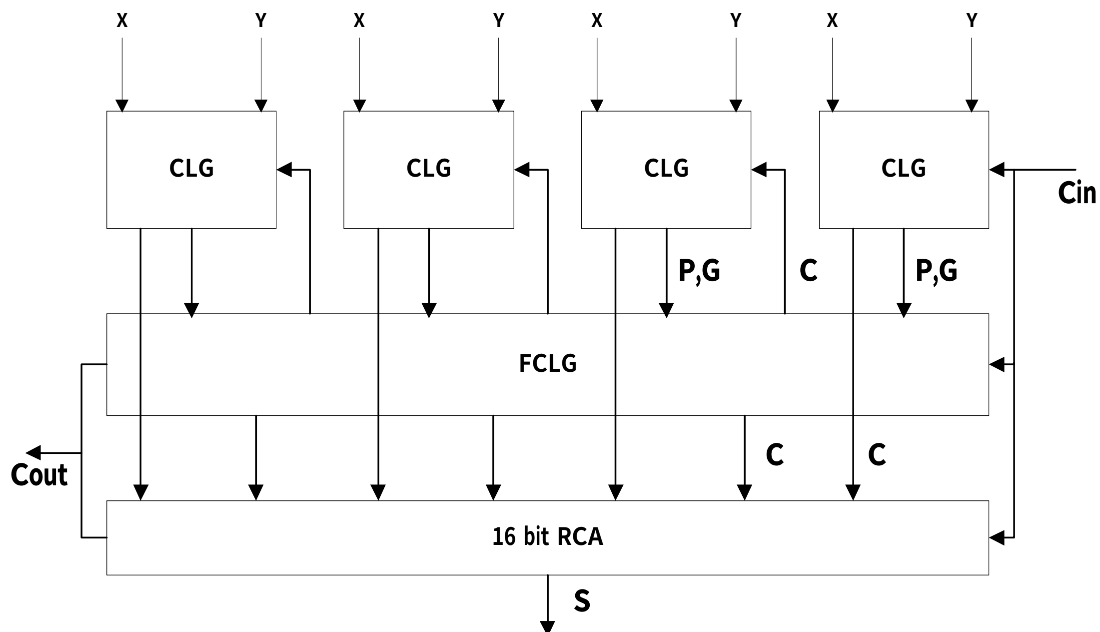


図3-5 CLAの接続例

まず、16 ビットの FCLG の構成について記述する。ここでは内部で CLG を 4 個使用している。関数の中の(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16)は FCLG 中の Cout の出力を指定している。この C の指定は FCLA の場合のみ使用することができる。以下のような回路の場合、Cout の出力は全て必要ないので、記述は次のようになる。

記述例

CLA(16 FCLG(16 (4 8 12 16) CLG(4) CLG(4) CLG(4) CLG(4))
CSE(4(RCA(４) RCA(４)) CSE(4 RCA(４) RCA(４))
CSE(4 RCA(4) RCA(4)))

（3-5）

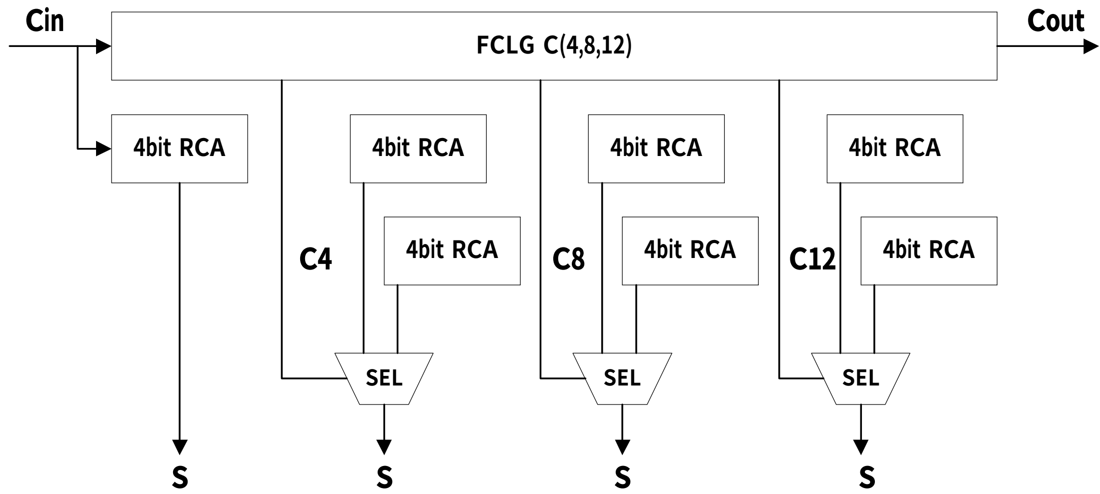


図3-6 FCLGとCSEの接続例

C の出力が限定され、CSE に入力されていることがわかると思う。

3-1-3　CSK の記述

記述例

CSK(12 (1 3) (3 5) (5 9) (9 11) (1 9))　　（3-6）

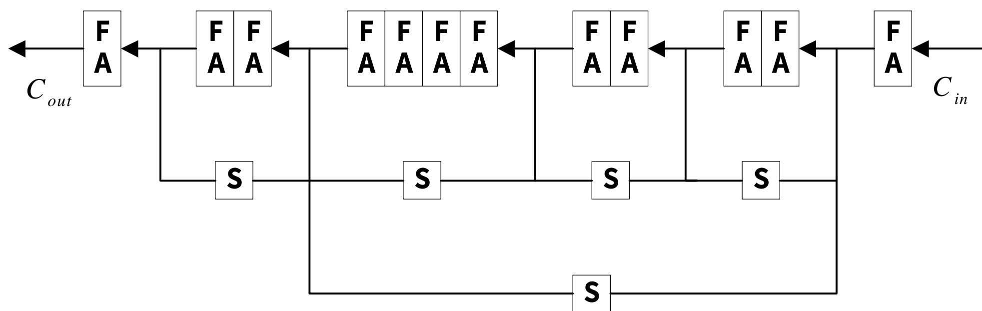


図3-7 CSKの例

これは 12 ビットの Ripple carry adder が存在し、（　）の内部の数字がスキップ回路の場所を指定している。この場合も、内部に指示が無ければ default で FA を使用する。図 3-7 に例の回路図を示す。

内部の回路ブロックが存在する場合は、以下の図のような接続と、スキップの関係が生まれる。

記述例

```
CSK(52 CLA(16 FCLG(16 (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15  16)
          CLG(4)  CLG(4)  CLG(4) CLG(4)  RCA(16)))
      CSE(12 RCA(12)  RCA(12))
      RCA(8)
      CLA(16 FCLG(16 (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15  16)
          CLG(4)  CLG(4)  CLG(4) CLG(4)  RCA(16)))
    (1 2) (1 4))
```

（3-7）

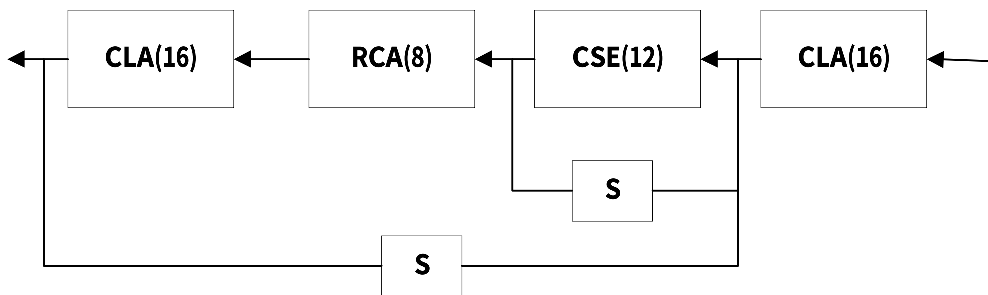


図3-8 CSKの内部接続

接続関係は、図のようにRipple carryの形で手前のビットから接続されていく。そして、（　）の内部で同様に SKIP 回路が接続されていく。

3-1-4　CSE の記述

記述例
CSE(8 RCA(8) RCA(8))　　（3-8）

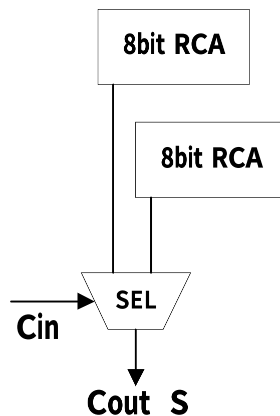


図3-9 CSEの回路例

Carry select adder は Cin の入力があって初めて動作する。通常使用する場合は、

記述例
RCA(16 RCA(8) CSE(8 RCA(8) RCA(8))　　（3-9）

という形にして、Cin を指定してやる必要がある。

3-2　信号線の表記

これまでの例で作ってきた加算器を接続し、Hybrid adder を実現する。この実現するために、個々の Adder に名前を付ける方法とそれぞれの接続関係を指定していく方法を取る。また、回路に名前をついた場合でなければ使用できない関数の説明をする。

3-2-1　回路の名前

記述例

SET A (RCA (8))　　(3-10)

SET の記述の後に続く文字列がその加算器の名前になる。(　)の中は、名前の付けられた加算器の構造を、構造を現す関数によってしめす。この記述により 8 ビット Ripple carry adder の A という名前の加算器が生成される。

3-2-2　接続を記述する関数

それぞれの加算器の接続例を示す。

記述例

CON ((A B) (D E))　　(3-11)

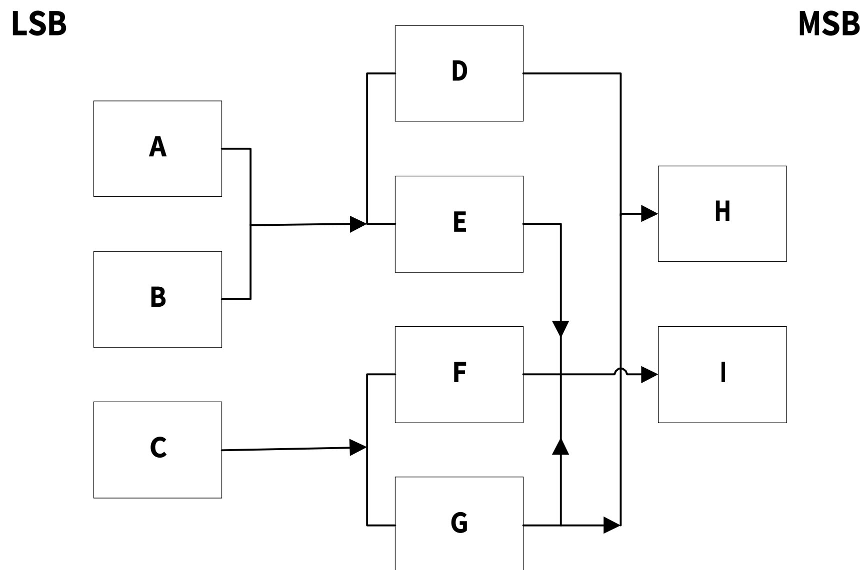


図 3-10 接続の例

図 3-10 のように、加算器に 4-3-1 で説明したような方法で A～I まで名前がつ

いているものと仮定する。上記の記述例はＡ Ｂと Ｄ Ｅの接続を意味している。最初の（　）の中の名前か信号が出力され、後の（　）の中の名前の回路に入力される。

記述例

CON ((A B) (DE))
CON ((C) (F G))
CON ((DG) (H))
CON ((E F G) (I))　　（3-12）

3-2-2　その他の記述方法

上記だけの記述方法では実現不可能な,例えば図 4-9 のような回路を作成したい場合がある。その場合,次のような記述の方法を取る。

記述例

SK ((A B) (H))　　（3-13）

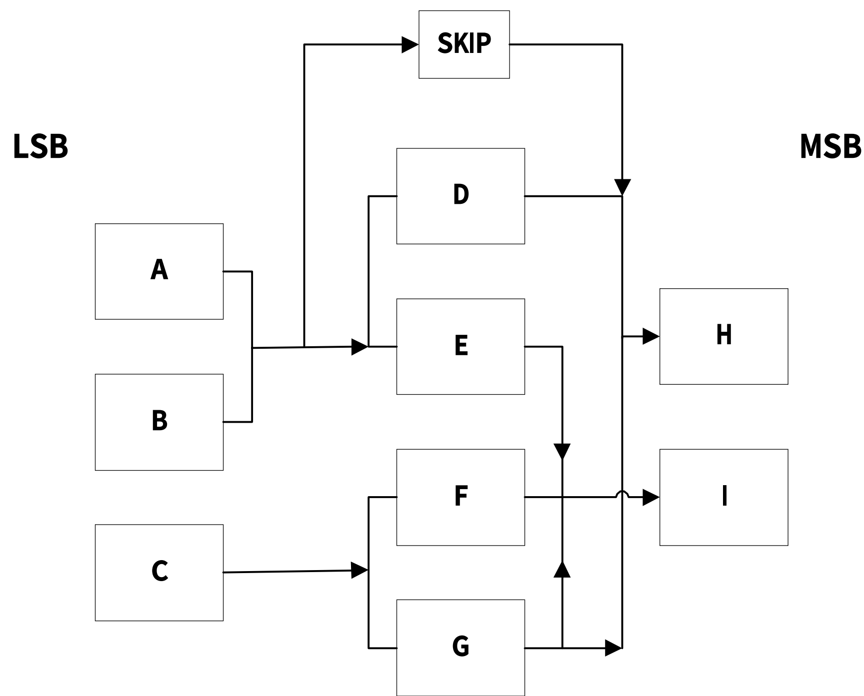


図　3-11 Skip回路がついたパターン

この記述は AB の出力を受けて D,E,F,G をスキップし、H への入力を示す。CSK では parallel に指定した複数のブロックをスキップすることはできないために、SK という関数を使用する。

また図 4-8 同士を select したい場合は、いったん一つの Adder として考えた後に select することとなる。

記述例　SET X((CON (A B) (DE))
(CON (C) (F G))
(CON (DG) (H))
(CON (E F G) (I)))

SEL(X X)

（3-14）

とすることで実現可能となる。

3-3 記述の拡張性

今回,記述の構文として指定したのは Ripple carry adder、Carry lookahaed adder、Carry skip adder、Carry select adder の四種類である。しかし Carry save adder や binary lookahead carry adder など、他にも加算器が存在する。これらの加算器は、それぞれ必要な入出力が決まっているために、関数として指定してやれば、容易に実現が可能である。ある規定の形が存在している、と仮定できるものならば導入は比較的容易である。

この考え方は、ALU など、より複雑なものにも持ち込むことができる。ただし、今回の加算器ではありえない、例えば Serial adder など Carry のループが発生する場合,それぞれの信号の接続関係が複雑になるために、Carry に名前をつけて、接続を指示していく必要がある。よって複雑になってしまい、単純化したはずのメリットがなくなってしまう恐れがある。

3-4 構文規則について

構文規則について示していく。

字句規則

keyword(キーワード)

CLA　CLG　CON　CSE　CSK　RCA　SEL　SET　SK

Identifier（識別子）

digit　0 1 2 3 4 5 6 7 8 9

non-digit　a b c d e f g h i j k l m n o p q r s t u v d x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Constant（定数）

integer-constant

Puncturator(区切子)

（　）

構文規則

Statement(文)

statement :

labeled-statement

expression-statement

labeled-statement :

identifier : statement

case costant-expression : statement

default : statement

4　今後の展開

4-1 本来の目的

本来の目的は、加算器の回路の構成によって、面積、速度、また消費電力がどれだけ変化しているか確認することであった。そのために、様々な構成の加算回路を作成し、その性能を現したデータベースを作成し、グラフにまとめる必要がある。

完成したライブラリの情報により、加算器の構造によってどの回路構成を使用した場合に面積が小さくなるか、速度が速くなるか影響を確認することができる。理論上は高速であるが、現実には面積が大きく速度が遅いといった実験データが取れることが予想された。また、そのライブラリを利用して、面積や速度等の数値を入力することにより、作成したい加算回路を生成することも可能である。

4-2 目的の達成方法

4-2-1　回路の文法作成

今回の回路の文法は、それ自身単体で役に立つものではない。他のソフトウェアと組み合わせて初めて意味が出てくる。

回路の文法を利用して、数多くの回路のパターンを記述する方法を考える必要がある。それには、64 ビットなら 64 ビットの加算器という制約の中で、最小単位となる加算回路のビット数、その回路の構成、どのような接続関係にあるかを指定しなければならない。

一つの方法として、TREE を使用して回路を生成する方法がある。まず、TREE の葉となる部分を最小単位となる Adder のビット数とする。次に加算回路の種類である。４種類の加算回路の構成から選び出し、指定する必要がある。

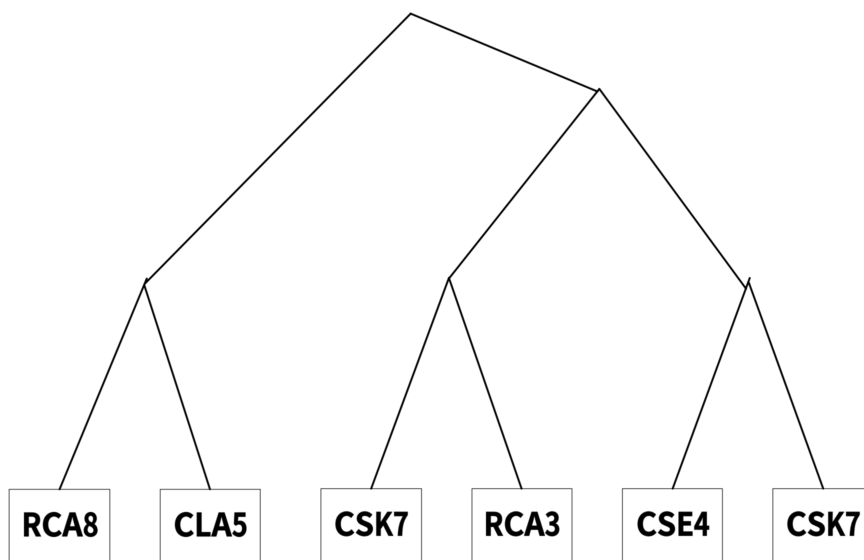


図　4-1 TREEの回路指定

その後、図 4-2 のようにそれぞれの分岐地点でどのような構成の回路になるか割り振ればよい。

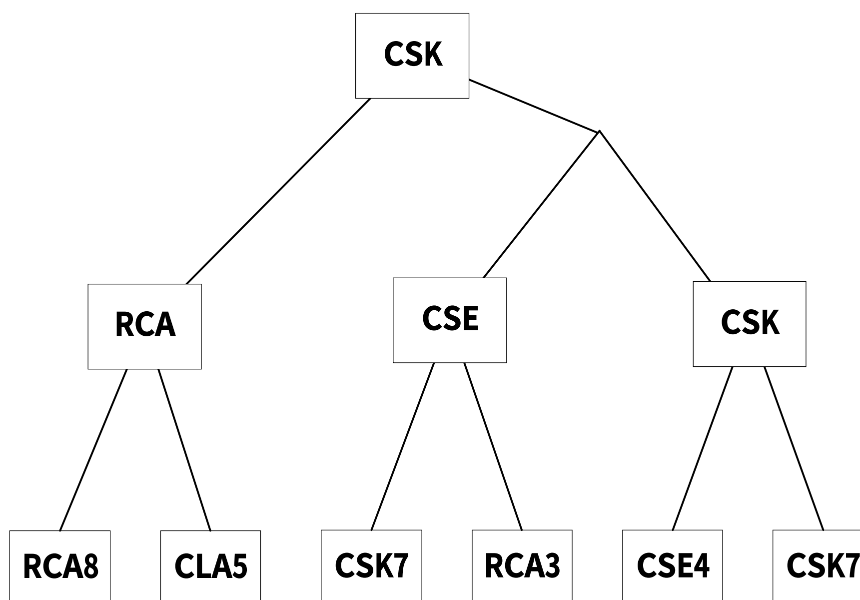


図　4-2 交点の接続

次にこの回路の組み合わせを考える。この考え方では、回路の接続方法が、直列なもののみしか生成できないので違う方法で組み合わせを考える必要がある。(図 4-3 参照)

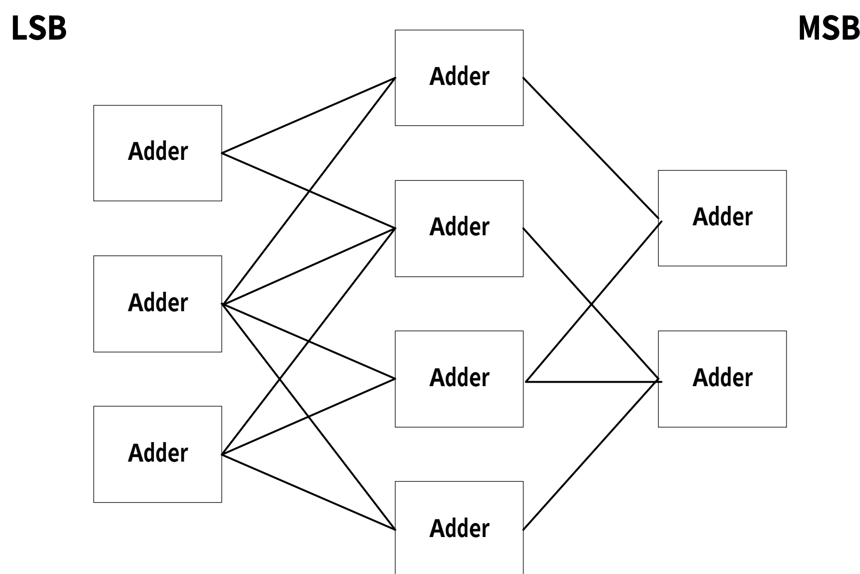


図 4-3 並列な回路接続

4-2-2 回路文法からの回路合成方法

次の作業として、回路の文法から実際の回路に直す必要がある。最近では、HDL から実際の回路を合成することができるので、HDL のソースに変換した後、合成と最適化を行う方法が良いと考えられる。合成と最適化の方法は、既存のソフトウェアに委ねることにする。面積や速度は最終的には回路のプロセスや構成によって左右されるために、そのルールにあったライブラリをその都度造る必要があるからである。そのため、既存のソフトウェアとそれにあったプロセスルールに従って作成するほうがよい。

生成方法としては、構文解析を行い、その出力を HDL の形で出力する方法が考えられる。

例えば、RCA(8)と記述すれば

```
library ieee;
use ieee.std_logic_1164.all;
entity RCA8 is
      port (
            X1, X2, X3, X4, X5, X6, X7, X8: in std_logic;
            Y1, Y2, Y3, Y4, Y4, Y6, Y7, Y8: in std_logic;
            S1, S2, S3, S4: out std_logic;
            Cin : in std_logic;
            Cout: out std_logic
      );

end RCA8;
architecture BEHAVIOR of RCA8 is
signal c1, c2, c3, c4, c5, c6, c7 :std_logic;

begin
   S1 <= X1 xor Y1 xor Cin;
   c1 <= X1 and Y1 and Cin;

   S2 <= X2 xor Y2 xor c1;
   c2 <= X2 and Y2 and c1;

   S3 <= X3 xor Y3 xor c2;
   c3 <= X3 and X3 and c2;

   S4 <= X4 xor X4 xor c3;
   c4 <= X4 and X4 and c3;

   S5 <= X5 xor Y5 xor c4;
   c5 <= X and Y5 and c4;

   S6 <= X6 xor Y6 xor c5;
   c6 <= X6 and Y6 and c5;
```

```
   S7 <= X7 xor Y7 xor c6;
   c7 <= X7 and X7 and c6;

   S8 <= X8 xor X8 xor c7;
   Cout <= X8 and X8 and Cout;


end BEHAVIOR;
```

例 4-1　VHDL による記述

例 4-1 のようなソースファイル（VHDL による記述）が生成されるようにすればよい。

このファイルを Design Analyzer などのツールを利用して合成､最適化する。そしてライブラリにデータを保存し、再度記述の生成、HDL 記述変換と作業を繰り返してライブラリを作成する。概略を図 4-4 に示す。

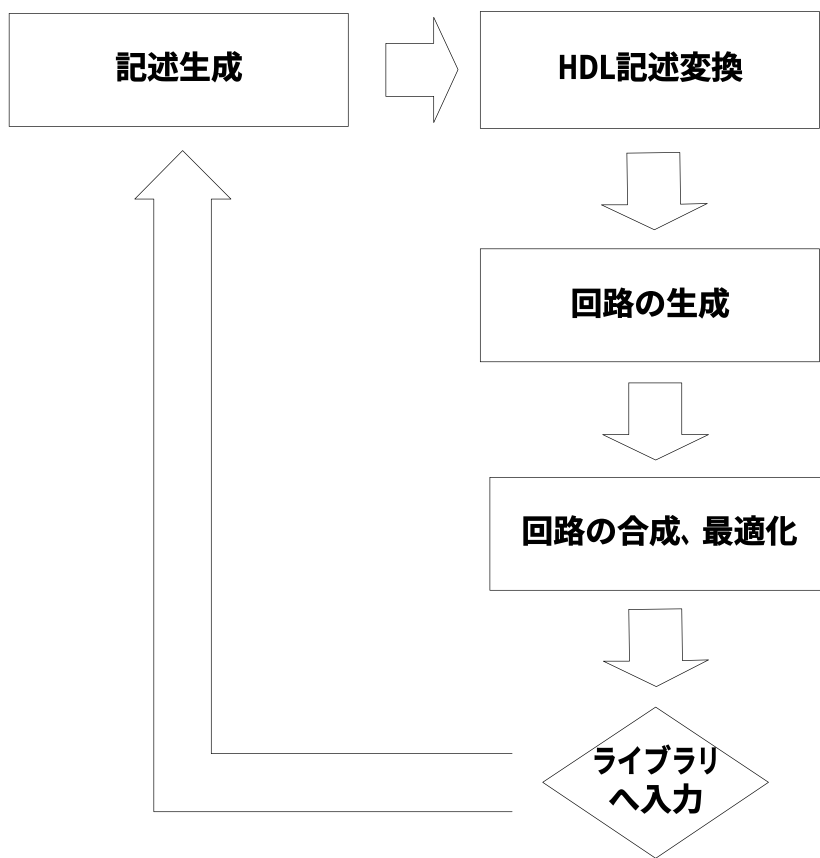


図4-4 ライブラリの作成ルーチン

4　結論

今回の研究では、多くの種類の加算回路を検討し、その加算回路の構成を実現することのできる文法を作ることに成功した。これは、無駄な部分を無くし、短縮され洗練された形となっている。今後、この研究を展開するためには、今回作成した文法により記述された加算器を HDL へ変換するソフトウェアが必要となる。このようなソフトウェアができれば記述そのものを自動生成する手法の研究への展開も期待できる。

本手法は加算回路以外にも、メモリや CPU 等の構成にも拡張できるものであるといえる。しかし、このためには Carry 以外の信号の接続関係を現す記述の開発が必要となる。

謝辞

　今回の卒業研究に際して、丁寧なご助言、御鞭撻下さりました、高知工科大学・電子光システム工学科　橘 昌良助教授に心から感謝いたします。
　研究中、親切丁寧なご助言くださった、高知工科大学・電子光システム工学科学長、原 央　教授、ならびに矢野 政顕教授に厚くお礼申し上げます。
また、同じ研究室において色々助言してくださった皆様にお礼申し上げます。

参考文献

[1]　Behrooz Parhami,
Computer Arithmetic　ALGORITHMS AND HARDWARE DESIGNS,
Oxford University Press　(2000)

[2] 中澤　喜三郎
計算機アーキテクチャと構成方式
朝倉書店　（1995）

[3]　五月女　健治,
yacc/lex　プログラムジェネレータ on UNIX,
テクノプレス　(1996)

[4]　RICHARD KESLEY, WILLIAM CLINGER, JONATHAN REES
Re*vised*[5]　Report on the Algorithmic Language Scheme
http://www.scheme.org　(1998)