Memoirs of the Faculty of Science
Kochi University (Information Science)
Vol. 26 (2005), No. 2

冗長２進加算器と乗算器の性能評価

宮原克典　横山真登　國信茂郎

高知大学理学部数理情報科学科

Abstract

近年の集積回路の高集積度による VLSI（大規模集積回路）の出現により、HDL（ハードウェア記述言語）を用いた機能設計からトップダウン設計が重要になってきている。

本研究では、HDL を用いて従来の２進加算器と冗長２進加算器を設計し、論理合成を行うことによって乗算器のように繰り返し加算を行うような場合の冗長２進加算器の優位性について調べた。

冗長２進加算器と乗算器の性能評価

１　序論

現在、われわれの身の回りにある電子機器には大規模集積回路（Large Scale Integration：LSI）が多数搭載されている。例えば、パソコンやゲーム機などに搭載されている。最近の傾向として、パソコンやゲーム機などの小型・携帯化そして、高速化が進んでいる。そのために、LSI も小型かつ高速に動作するものが必要である。

かつてのコンピュータは真空管が中核をなしていた。1946 年に、米ペンシルバニア大学で世界初の実用汎用電子計算機として ENIAC（Electronic Numerical Integrator and Calculator）が開発されたが、これは、約１８０００本の真空管を使用しており、長さ８０フィート、高さ８．５フィート、幅は数フィートあった。また、１秒間に５０００回の整数演算が可能であった。このように、中核テクノロジに真空管を使用するコンピュータが 1 コンピュータ世代と呼ばれる 1950 年ごろから 1959 年までのコンピュータにみられる。

1960 年代になると第 2 世代と呼ばれる中核にトランジスタ（電気的なオン／オフ動作をするスイッチ）を用いるコンピュータが開発されるようになり、より安価なコンピュータが作られた。トランジスタを用いた場合の相対コスト性能比（真空管を用いた場合を１とする）は３５である。

1970 年代には第 3 世代と呼ばれ、数十～数百のトランジスタを 1 チップにまとめて搭載した集積回路（Integrated Circuit）が用いられた。このときの相対コスト性能比は９００であり、大幅に対コスト性能が向上している。

1980 年代からは 1 チップに搭載するトランジスタ数は数百～数百万へ大幅に増加し、相対コスト性能比は 2,400,000 まで上がった。このチップは第 3 世代の集積回路よりもさらに集積度が増大しているために超大規模集積回路（Very Large Scale Integrated circuit）と呼ばれ、頭文字をとって VLSI と呼ばれる。これが第 4 世代である。

我が研究室ではこの VLSI の設計についての研究を行っている。昨年までの研究で浮動小数点プロセッサの HDL による設計が行われている。この回路に使用するための加算回路について様々な回路を HDL により設計し、規模や発生する遅延時間についての性能を比較することによって、プロセッサ全体の性能を向上させることが本論文の目的である。

本論文は第 1 章において、VLSI 設計や HDL の基本を記し、第 2 章では一般的な加算器を紹介する。そして、第 3 章において実際に通常 2 進加算器を HDL で記述しその性能を調べ、第 4 章において冗長 2 進を利用することによって桁上げの伝播を押さえることのできるという優位性について記し、HDL で設計してその性能について調べた。第 5 章では通常 2 進加算器と冗長 2 進加算器の性能について比較し、第６章では乗算器における冗長 2 進加算器の優位性について調べた。

２　基本的な加算器の紹介

計算機の構造はメモリやレジスタのような記憶回路と加算器や乗算器といったデータを加工するための機能回路に分けられる。本論文では機能回路のうち一般的な回路であり、さまざまな演算回路の基本となる加算器について、いくつかの回路を HDL で設計・論理合成して、それらの性能を比較し、より高速な加算器について調べた。

この章では加算器の中でも基本的なものについて簡単に紹介し、次章にて実際に HDL で記述した加算器について述べる。

2.1　ビット加算器

１ビットの２つの数を加算するためにビット加算器を使用する。ビット加算器には半加算器（Half Adder）と全加算器（Full Adder）がある。

2.1.1　半加算器

半加算器は１ビットの２つの数 A、B を加算し、その和 S と上位への桁上げ Cout を出力する回路である。表 2.1 にその真理値表を示す。和 S は A と B の排他的論理和であり、桁上げ Cout は A と B の論理積である。また、回路図を図 2.1 に示す。

入力		出力	
A	B	S	Cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

表 2.1　半加算器の真理値表

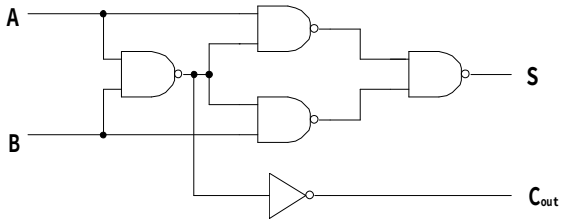


図 2.1　半加算器の回路図

2.1.2　全加算器

半加算器では 2 つの数の加算であったが、全加算器では下位からの桁上げも考えて 3 つの数 A、B、Cin を加算し、その和 S と上位への桁上げ Cout を出力する回路である。表 2.2 にその真理値表を示す。和 S は A と B と Cin の排他的論理和であり、桁上げ Cout は A と B と Cin の多数決論理である。また、回路図を図 2.2 に示す。

入力			出力	
A	B	Cin	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

表 2.2　全加算器の真理値表

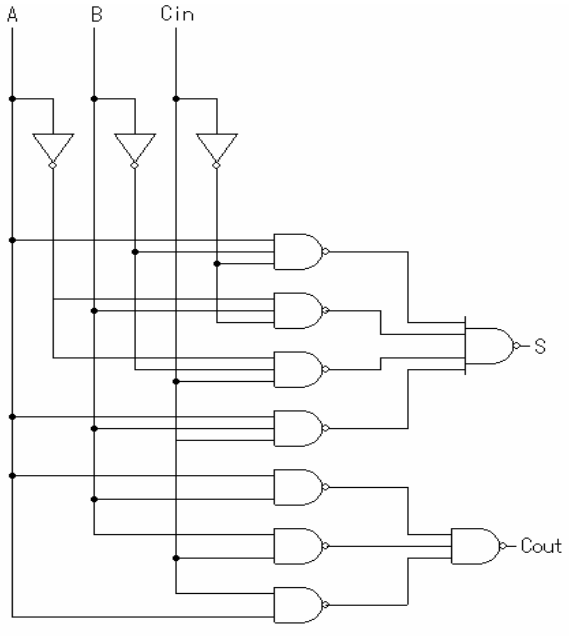


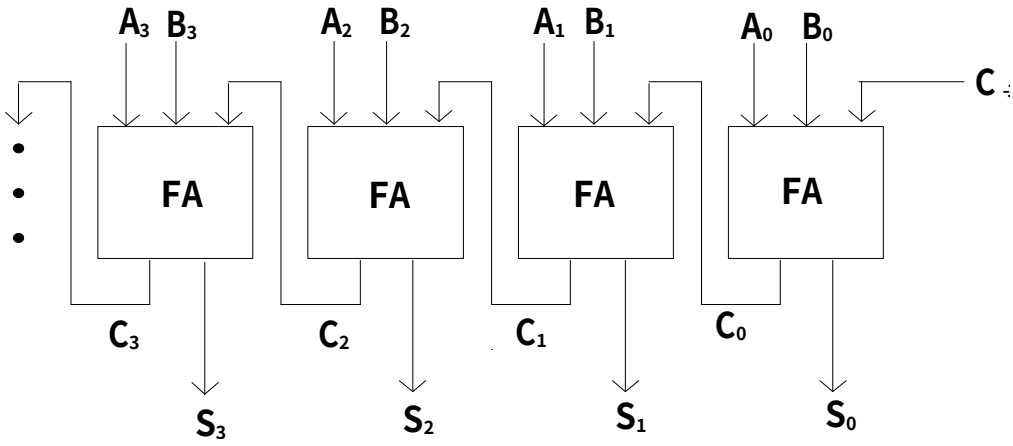
図 2.2　全加算器の回路図

2.2　複数ビット加算器

前に述べたビット加算器は 1 ビット単位の加算を行うための回路であった。ここでは複数長のビットで表された 2 進数の加算をするための加算器について基本的なものを紹介する。

2.2.1　順次桁上げ加算器（Ripple Carry Adder）

n ビットの加算をするために全加算器を各ビットに置き、１つ下からの桁上げ信号 Cout を Cin へ入力するようにしたもの。LSB（Least Significant Bit：最下位ビット）から MSB（Most Significant Bit：最上位ビット）へと桁上げ信号が各ビット位を順に下から上がって行くために順次桁上げ加算器と呼ばれている。この回路の利点は簡単な回路構成で実現できることである。欠点は桁上げ信号が最下位から順次伝搬するために計算速度が遅いことである。このため、高速な演算を要求される計算機には向いていない。図 2.3 に回路図を示す。



FA：全加算器（Full Adder）

図 2.3　順次桁上げ加算器の回路図

2.2.2 桁上げ先見加算器（Carry Lookahead Adder）

順次桁上げ加算器では、桁上げ信号が下から上へと順次上がっていく。その桁上げ信号を先に予見することができればこの回路を高速化できる。

まず、桁上げ信号の性質について考えてみると、

1）桁上げ信号は、０か１のいずれかである。

2）桁上げ信号は、考えている桁より上位の桁の入力には無関係である。

3）入力 A、B が０と０の場合は、そのビットからの桁上げ信号は、そのビットへ下から上がってくる桁上げ信号には関係なく０となる。

4）入力 A、B が０と１、または１と０の場合は、そのビットからの桁上げ信号は、そのビットへ下から上がってくる桁上げ信号に等しい。この状態を Pass 状態と呼ぶ。

5）入力 A、B が１と１の場合は、そのビットからの桁上げ信号は、そのビットへ下から入ってくる桁上げ信号には関係なく 1 となる。この状態を Generate 状態と呼ぶ。

これらの性質を利用して桁上げ先見ユニット(CLU：Carry Lookahead Unit)を設計することによって,この回路は高速な演算が可能であり、高速計算機に使用されている。図 2.4 に 4 ビット桁上げ先見ユニットの回路図を示す。各ビットの桁上げ生成伝搬ユニットによって、Pass 状態のときは信号 P を出力し、Generate 状態の時は信号 G を出力する。信号 P は入力 A、B の排他的論理和であり、信号 G は入力 A、B の論理積である。そして、図 2.5 の式に従い各桁の桁上げ信号 C を出力する。また、図 2.6 には 4 ビット桁上げ先見加算器の回路図を示す。

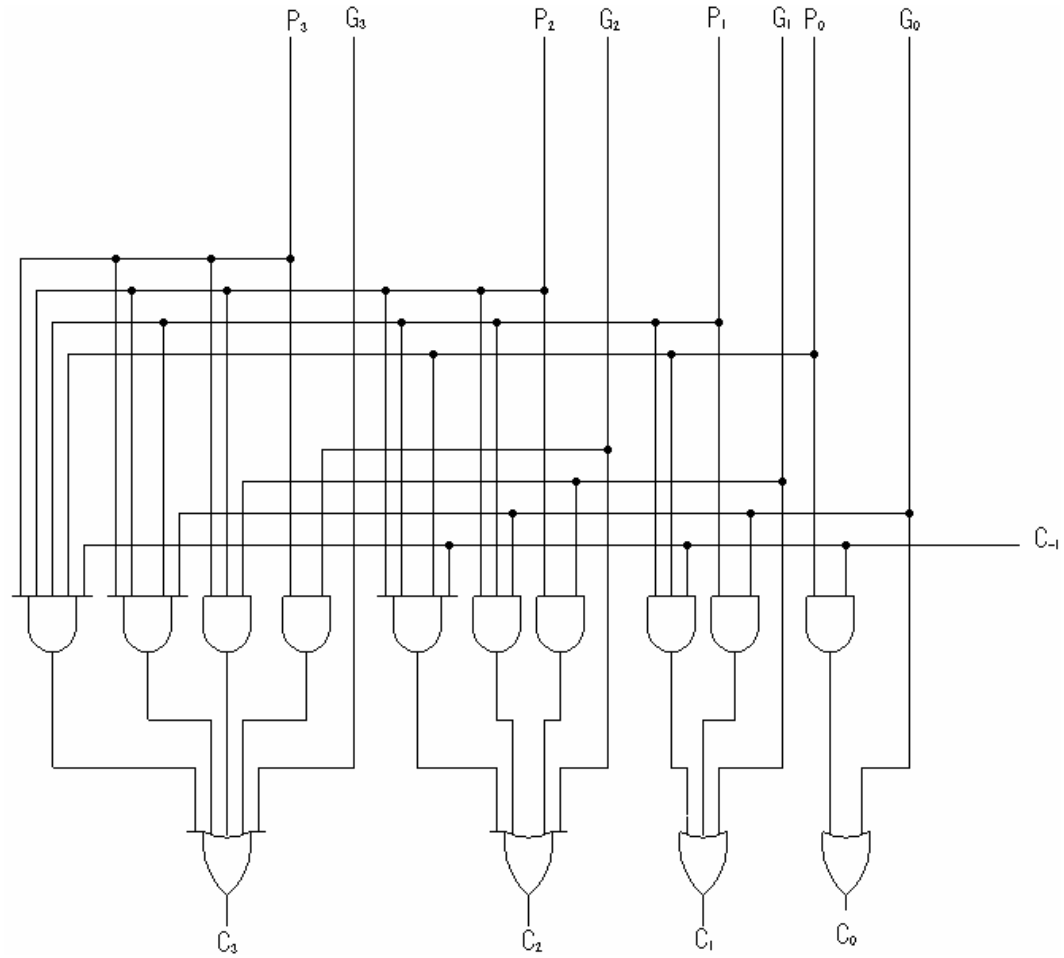


図 2.4　4 ビット桁上げ先見ユニット

$C_0 = G_0+P_0C_{-1}$

$C_1 = G_1 +P_1C_0 = G_1+P_1G_0+P_1P_0C_{-1}$

$C_2 = G_2+P_2C_1 = G_2+P_2G_1+P_2P_1G_0+P_2P_1P_0C_{-1}$

……

$C_k = G_k+P_kC_{k-1} = G_k+P_kG_{k-1}+P_kP_{k-1}G_{k-2}+ \cdots +P_kP_{k-1}\cdots P_1G_0+P_kP_{k-1}\cdots P_0C_{-1}$

図 2.5　桁上げ信号の論理式

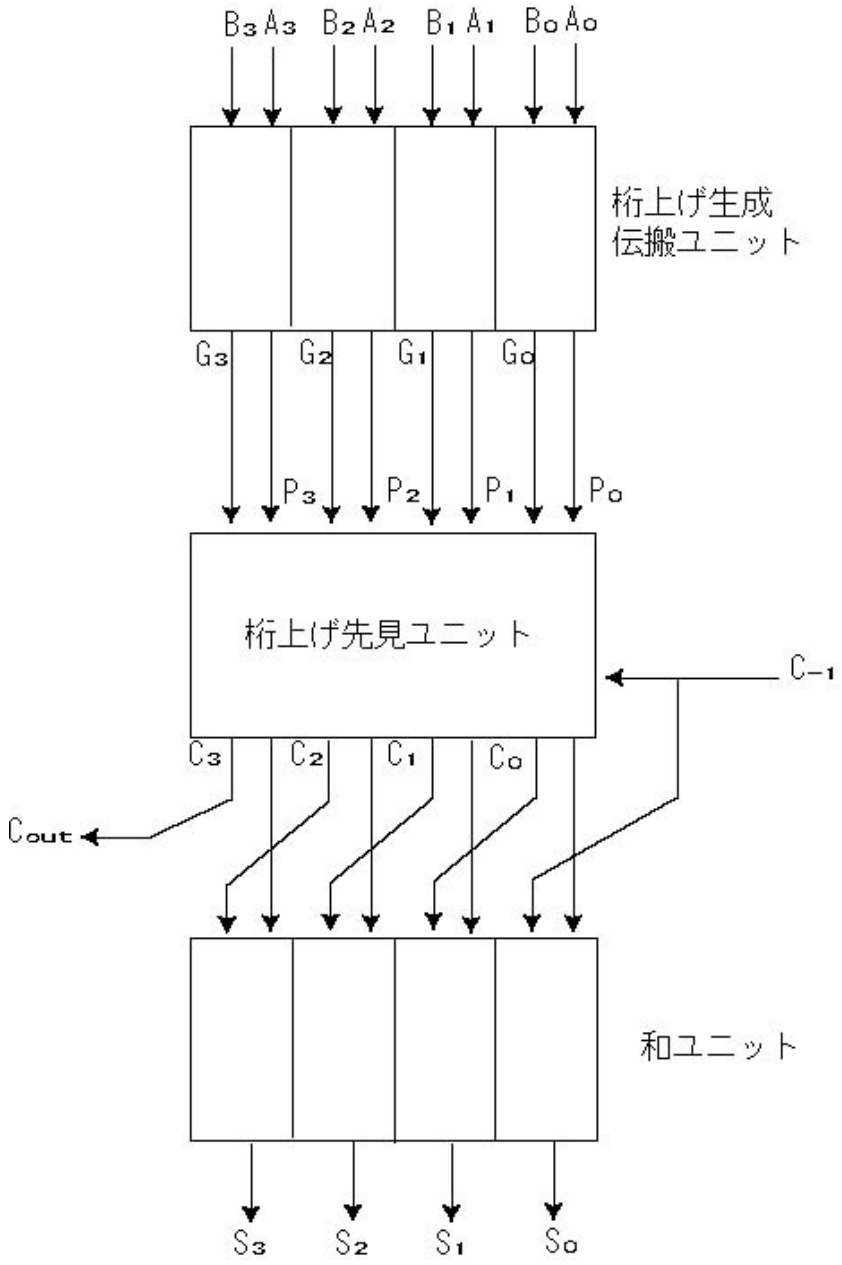


図 2.6　4 ビット桁上げ先見加算器の回路図

また、より高速化を目指すために 4 ビット分を一括してみた場合の Pass 信号と Generate 信号を出力する回路がある。その回路図を図 2.7 に示す。次章での実験に用いた通常 2 進を扱う加算器の回路はこの回路を基本として設計している。

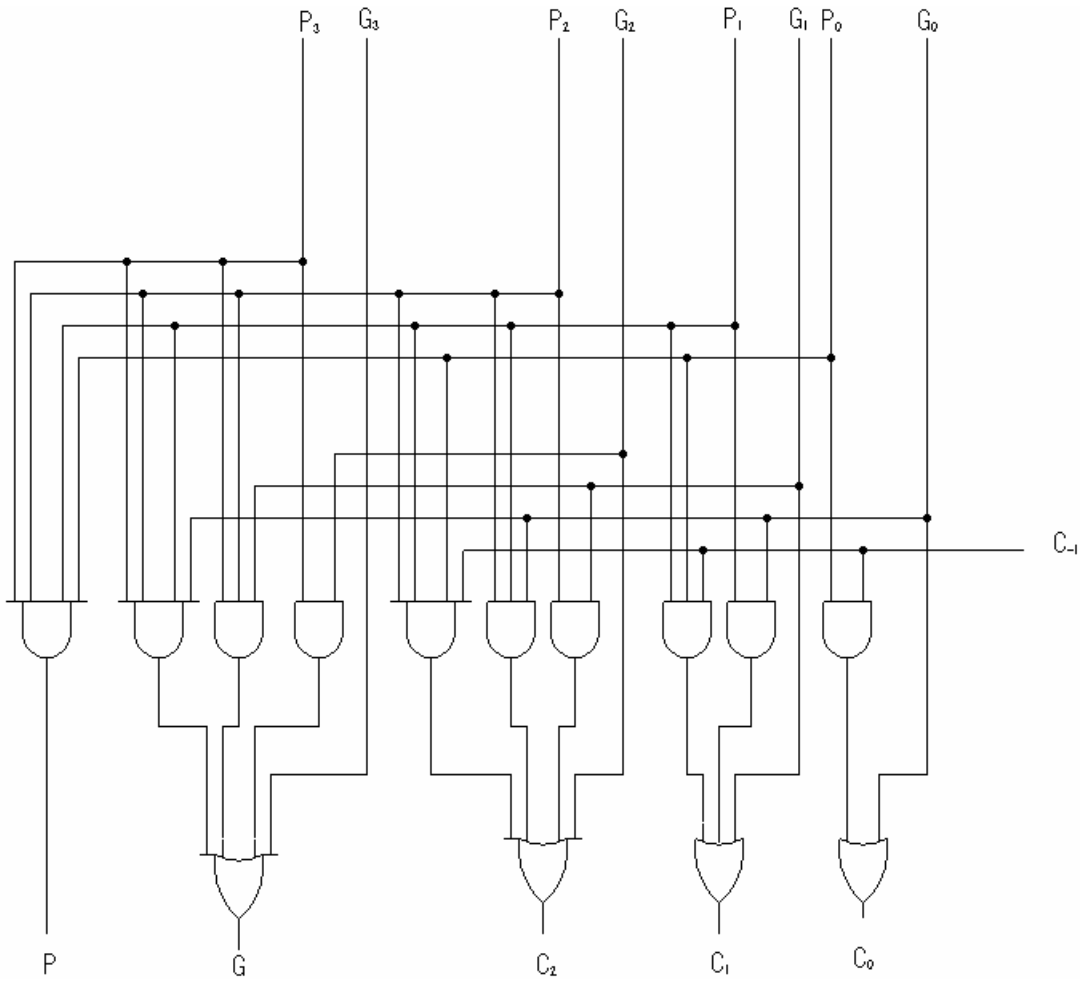


図 2.7　4 ビット分を一括して信号 P と信号 G を出力する CLU の回路図

3　通常 2 進加算器の設計

この章ではいくつかの通常 2 進加算器を Verilog-HDL で記述し、論理合成ツールによって論理合成を行い出力された回路について、ゲート数と遅延時間を比較して、より効率のよい加算器について調べてみた。今回はすべて 16 ビットの加算を行うものとした。今回、論理合成には Xilinx 社の“ CoolRunner XPLA3 CPLDs ”のテクノロジを使用した。また、遅延時間の計算には 3 入力 NAND を基準とした表 3.1 を基に計算した。

表 3.1　基本論理ゲートの 1 ゲート当たりの信号伝播遅延時間と、3 入力 NAND ゲートの遅延時間を 1⊿に規格化したときの基本論理ゲートの規格値

ゲート名	1 ゲート当たりの信号遅延時間	
	実測値 (ps)	規格値
インバータ	120	0.5⊿
2 入力 NAND	160	1⊿
2 入力 NOR	260	1⊿
3 入力 NAND	240	1⊿
3 入 NOR	430	1.5⊿
インバータ＋伝送ゲート	270	1⊿
伝送ゲートを利用した XOR（XNOR）	330－350	1.5⊿
複合ゲート	400－490	1.5⊿

冗長２進加算器と乗算器の性能評価

3.1　桁上げ先見加算器の順次桁上げによる加算器

図3.1のように前章で紹介した桁上げ先見加算器を4ビットごとに４つ使用しそれぞれ順次桁上げ方式によって接続して16ビット加算を行う加算器である。

この回路は比較的容易に設計できるが、この回路では桁上げがすべてのCLAを通らなければならないので、大きな遅延が発生する。

この加算器を以下に示すようにVerilog-HDLで記述し、論理合成を行ったときのゲート数は236個で、遅延時間は61.0⊿となる。

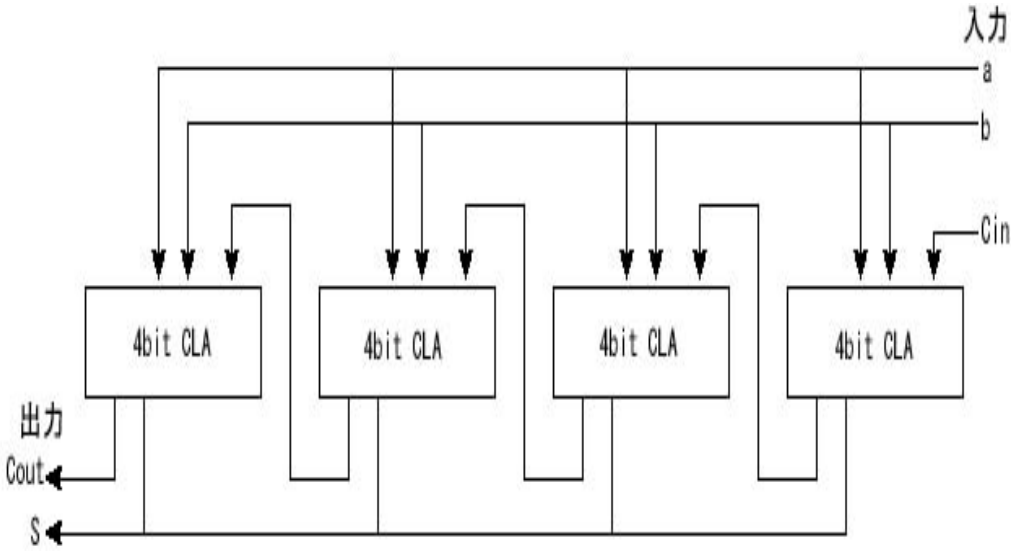


図 3.1　4 ビット長桁上げ先見回路の順次桁上げによる 16 ビット加算器

```
/*4 ビット長桁上げ先見回路の順次桁上げによる 16 ビット加算器*/
module Ripple1(a,b,cin,s,cout);
        input   [15:0] a,b;
        input          cin;
        output [15:0] s;
        output         cout;
        wire    [2:0]  c;

        CLA_4 i0(.a(a[3:0]),.b(b[3:0]),.cin(cin),.s(s[3:0]),.cout(c[0]));
        CLA_4 i1(.a(a[7:4]),.b(b[7:4]),.cin(c[0]),.s(s[7:4]),.cout(c[1]));
        CLA_4 i2(.a(a[11:8]),.b(b[11:8]),.cin(c[1]),.s(s[11:8]),.cout(c[2]));
        CLA_4 i3(.a(a[15:12]),.b(b[15:12]),.cin(c[2]),.s(s[15:12]),.cout(cout));
endmodule


/*****************************4 ビット桁上げ先見加算器*****************************/
module CLA_4(a,b,cin,s,cout);
        input   [3:0] a,b;
        input         cin;
        output [3:0] s;
        output        cout;

        wire [2:0]    c;
        wire [3:0]    x,y;

        Circuit_A i0(.a(a[0]),.b(b[0]),.cin(cin),.p(x[0]),.g(y[0]),.s(s[0]));
        Circuit_A i1(.a(a[1]),.b(b[1]),.cin(c[0]),.p(x[1]),.g(y[1]),.s(s[1]));
        Circuit_A i2(.a(a[2]),.b(b[2]),.cin(c[1]),.p(x[2]),.g(y[2]),.s(s[2]));
        Circuit_A i3(.a(a[3]),.b(b[3]),.cin(c[2]),.p(x[3]),.g(y[3]),.s(s[3]));

        CLU_4 i4(.p(x),.g(y),.cin(cin),.cout(cout),.c(c));
endmodule
```

```
/*****************************1 ビット桁上げ先見回路*****************************/
module Circuit_A (a,b,cin,p,g,s);

        input           a,b,cin;
        output          s,p,g;

        wire w0, w1, w2, w3, w4, w5;

        assign  w0 = ~(a & b),
                g  = ~w0,
                w1 = ~(a & w0),
                w2 = ~(b & w0),
                p  = ~(w1 & w2),

                w3 = ~(cin & p),
                w4 = ~(cin & w3),
                w5 = ~(p & w3),
                s  = ~(w4 & w5);
endmodule

/*****************************4 ビット桁上げ先見ユニット*****************************/
module CLU_4 (p,g,cin,cout,c);

        input   [3:0] p,g;
        input         cin;
        output        cout;
        output [2:0] c;

assign          c[0] = g[0] | (cin & p[0]),
                c[1] = g[1] | (g[0] & p[1]) | (cin & p[0] & p[1]),
                c[2] = g[2] | (g[1] & p[2]) | (g[0] & p[1] & p[2]) |
                        (cin & p[0] & p[1] & p[2]),
                cout = g[3] | (g[2] & p[3]) | (g[1] & p[2] & p[3]) |
                        (g[0] & p[1] & p[2] & p[3]) | (cin & p[0] & p[1] & p[2] & p[3]);

endmodule
```

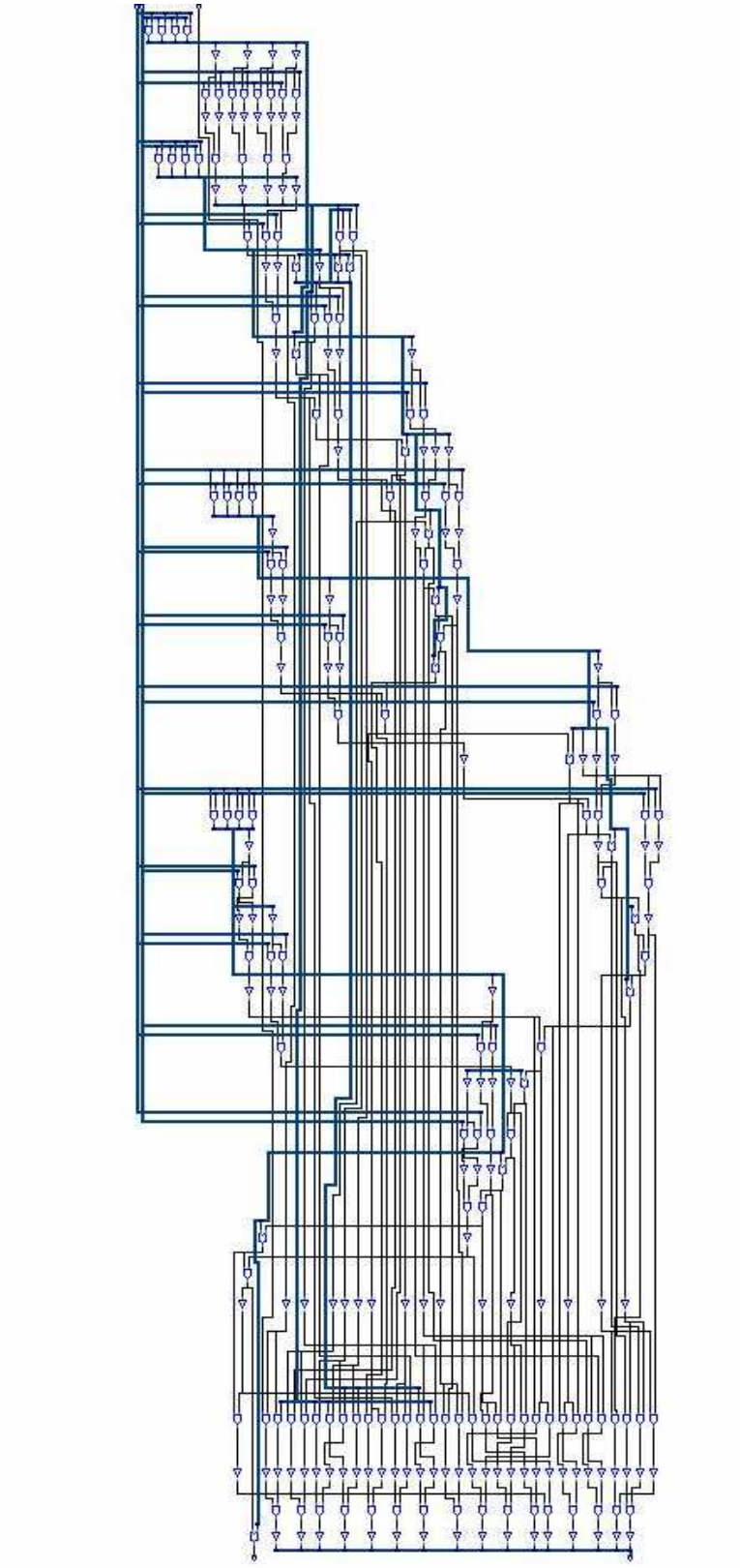


図 3.2　4 ビット長桁上げ先見回路の順次桁上げによる 16 ビット加算器の RTL 図

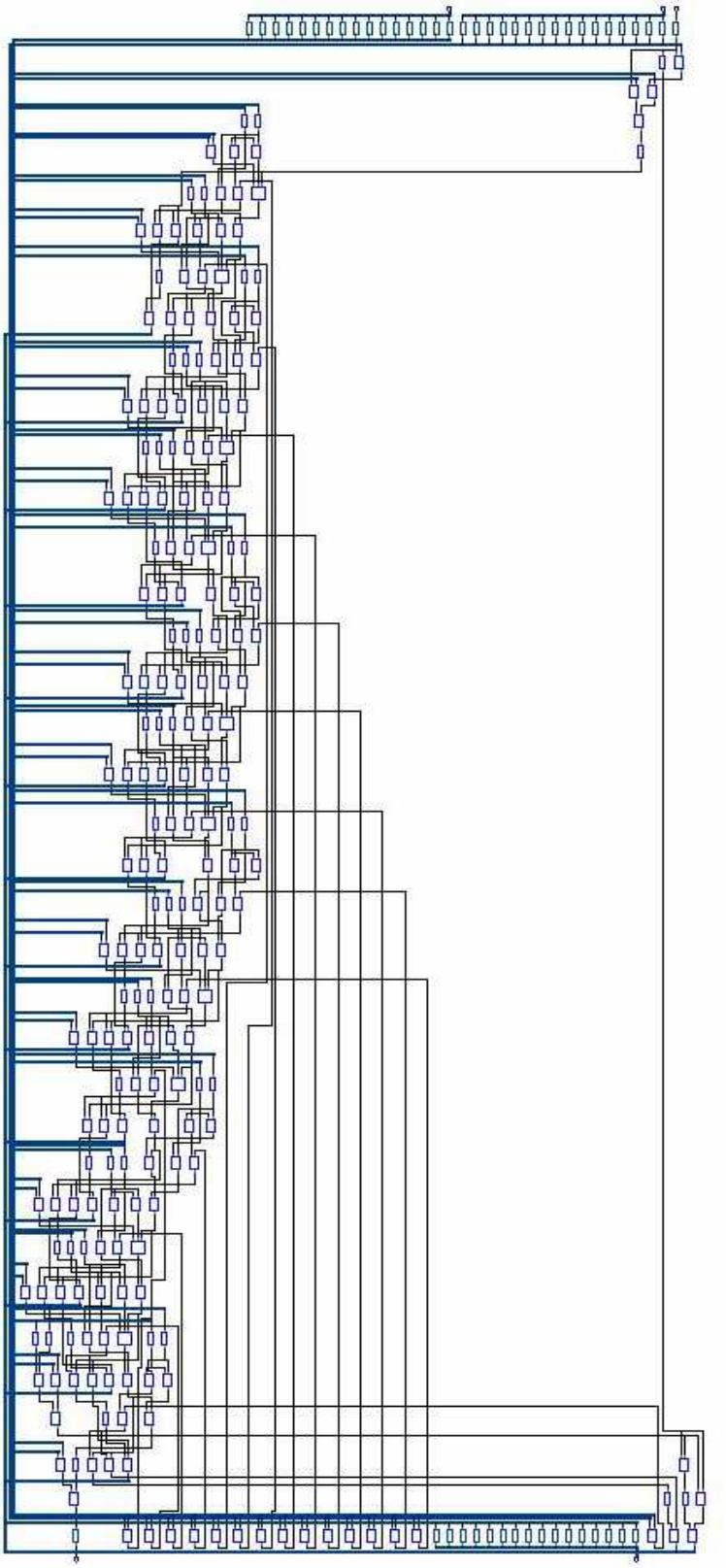


図3.3　4ビット長桁上げ先見回路の順次桁上げによる16ビット加算器の論理レイアウト
（テクノロジマッピング）

3.2　8 ビット長ブロック桁上げ先見回路の順次桁上げによる 16 ビット加算器

前の節で述べた加算器は 4 ビットの桁上げ先見加算器を 4 個並べて 16 ビット加算器を設計していたが、この節の加算器は図 3.2 のように 8 ビットの桁上げ先見加算器を 2 個つないで 16 ビットの加算を行うようにした。この加算器も比較的容易に設計することができ、このときの遅延は桁上げが 8 ビット桁上げ先見加算器 2 個を通過する時間である。

この加算器を Verilog-HDL で記述し、論理合成を行ったときのゲート数は 176 個で、遅延時間は 51.5⊿となった。

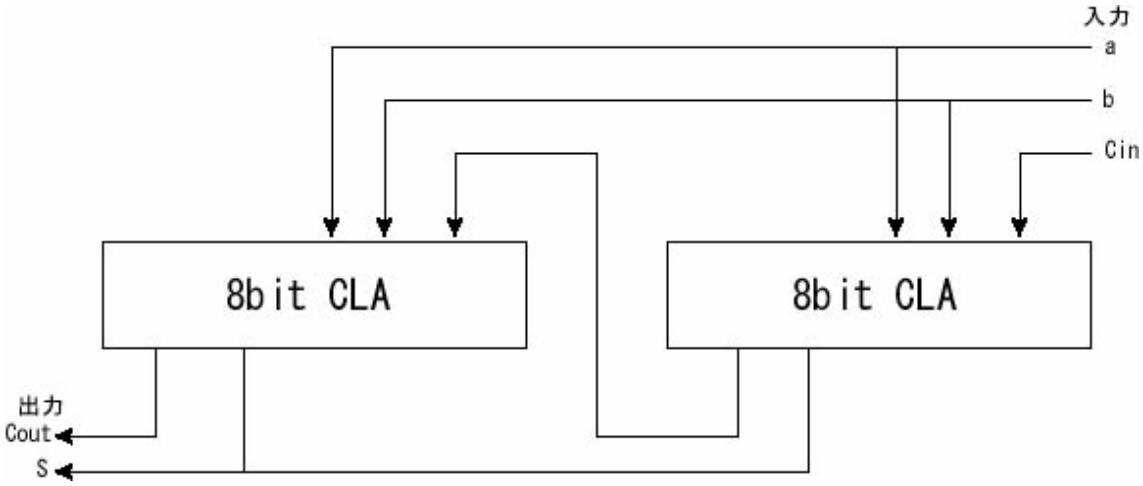


図 3.4　8 ビット長ブロック桁上げ先見回路の順次桁上げによる 16 ビット加算器

```
/*8 ビット長ブロック桁上げ先見回路の順次桁上げによる 16 ビット加算器*/
module Ripple2(a,b,cin,cout,s);
        input   [15:0]  a,b;
        input           cin;

        output  [15:0]  s;
        output          cout;

        wire            c1;

CLA_8   i0 (.a(a[7:0]),.b(b[7:0]),.cin(cin),.cout(c1),.s(s[7:0]));
CLA_8   i1 (.a(a[15:8]),.b(b[15:8]),.cin(c1),.cout(cout),.s(s[15:8]));


endmodule


/**************************8 ビット桁上げ先見加算器*****************************/
module CLA_8 (a,b,cin,cout,s);
        input   [7:0]   a,b;
        input           cin;

        output  [7:0]   s;
        output          cout;

        wire            p0,p1,p2,p3,p4,p5,p6,p7,g0,g1,g2,g3,g4,g5,g6,g7,
                        c0,c1,c2,c3,c4,c5,c6;

Circuit_A       i0 (.a(a[0]),.b(b[0]),.cin(cin),.p(p0),.g(g0),.s(s[0]));
Circuit_A       i1 (.a(a[1]),.b(b[1]),.cin(c0),.p(p1),.g(g1),.s(s[1]));
Circuit_A       i2 (.a(a[2]),.b(b[2]),.cin(c1),.p(p2),.g(g2),.s(s[2]));
Circuit_A       i3 (.a(a[3]),.b(b[3]),.cin(c2),.p(p3),.g(g3),.s(s[3]));
Circuit_A       i4 (.a(a[4]),.b(b[4]),.cin(c3),.p(p4),.g(g4),.s(s[4]));
Circuit_A       i5 (.a(a[5]),.b(b[5]),.cin(c4),.p(p5),.g(g5),.s(s[5]));
Circuit_A       i6 (.a(a[6]),.b(b[6]),.cin(c5),.p(p6),.g(g6),.s(s[6]));
```

```
Circuit_A       i7 (.a(a[7]),.b(b[7]),.cin(c6),.p(p7),.g(g7),.s(s[7]));

CLU_8           i8 (.g0(g0),.g1(g1),.g2(g2),.g3(g3),.g4(g4),.g5(g5),.g6(g6),.g7(g7),
                    .p0(p0),.p1(p1),.p2(p2),.p3(p3),.p4(p4),.p5(p5),.p6(p6),.p7(p7),
                  .cin(cin),.c0(c0),.c1(c1),.c2(c2),.c3(c3),.c4(c4),.c5(c5),.c6(c6),.c7(cout));

endmodule

/*****************************1 ビット桁上げ先見回路*****************************/
module Circuit_A (a,b,cin,p,g,s);

        input           a,b,cin;
        output          s,p,g;

        wire w0, w1, w2, w3, w4, w5;

        assign  w0 = ~(a & b),
                g  = ~w0,
                w1 = ~(a & w0),
                w2 = ~(b & w0),
                p  = ~(w1 & w2),

                w3 = ~(cin & p),
                w4 = ~(cin & w3),
                w5 = ~(p & w3),
                s  = ~(w4 & w5);
endmodule

/***************************8 ビット桁上げ先見ユニット***************************/
module CLU_8
(g0,g1,g2,g3,g4,g5,g6,g7,p0,p1,p2,p3,p4,p5,p6,p7,cin,c0,c1,c2,c3,c4,c5,c6,c7);

        input   g0,g1,g2,g3,g4,g5,g6,g7;
        input   p0,p1,p2,p3,p4,p5,p6,p7;
        input   cin;
```

```
        output   c0,c1,c2,c3,c4,c5,c6,c7;


wire    w00,w01,w02,w03,w04,w05,w06,w07,w08,w09,w10,w11,w12,w13,w14,w15,
        w16,w17,w18,w19,w20,w21,w22,w23,w24,w25,w26,w27,w28,w29,w30,w31,
        w32,w33,w34,w35;

assign
        w00 = cin & p0,                 //c0

        w01 = g0 & p1,                  //c1
        w02 = cin & p0 &p1,

        w03 = g1 & p2,                  //c2
        w04 = g0 & p1 & p2,
        w05 = cin & p0 & p1 & p2,


        w06 = g2 & p3,                  //c3
        w07 = g1 & p2 & p3,
        w08 = g0 & p1 & p2 & p3,
        w09 = cin & p0 & p1 & p2 & p3,

        w10 = g3 & p4,                  //c4
        w11 = g2 & p3 & p4,
        w12 = g1 & p2 & p3 & p4,
        w13 = g0 & p1 & p2 & p3 & p4,
        w14 = cin & p0 & p1 & p2 & p3 & p4,

        w15 = g4 & p5,                  //c5
        w16 = g3 & p4 & p5,
        w17 = g2 & p3 & p4 & p5,
        w18 = g1 & p2 & p3 & p4 & p5,
        w19 = g0 & p1 & p2 & p3 & p4 & p5,
        w20 = cin & p0 & p1 & p2 & p3 & p4 & p5,
```

```
w21 = g5 & p6,                          //c6
w22 = g4 & p5 & p6,
w23 = g3 & p4 & p5 & p6,
w24 = g2 & p3 & p4 & p5 & p6,
w25 = g1 & p2 & p3 & p4 & p5 & p6,
w26 = g0 & p1 & p2 & p3 & p4 & p5 & p6,
w27 = cin & p0 & p1& p2 & p3 & p4 & p5 & p6,

w28 = g6 & p7,                          //c7
w29 = g5 & p6 & p7,
w30 = g4 & p5 & p6 & p7,
w31 = g3 & p4 & p5 & p6 & p7,
w32 = g2 & p3 & p4 & p5 & p6 & p7,
w33 = g1 & p2 & p3 & p4 & p5 & p6 & p7,
w34 = g0 & p1 & p2 & p3 & p4 & p5 & p6 & p7,
w35 = cin & p0 & p1 & p2 & p3 & p4 & p5 & p6 & p7,



c0 = g0 | w00,
c1 = g1 | w01 | w02,
c2 = g2 | w03 | w04 | w05,
c3 = g3 | w06 | w07 | w08 | w09,
c4 = g4 | w10 | w11 | w12 | w13 | w14,
c5 = g5 | w15 | w16 | w17 | w18 | w19 | w20,
c6 = g6 | w21 | w22 | w23 | w24 | w25 | w26 | w27,
c7 = g7 | w28 | w29 | w30 | w31 | w32 | w33 | w34 | w35;


endmodule
```

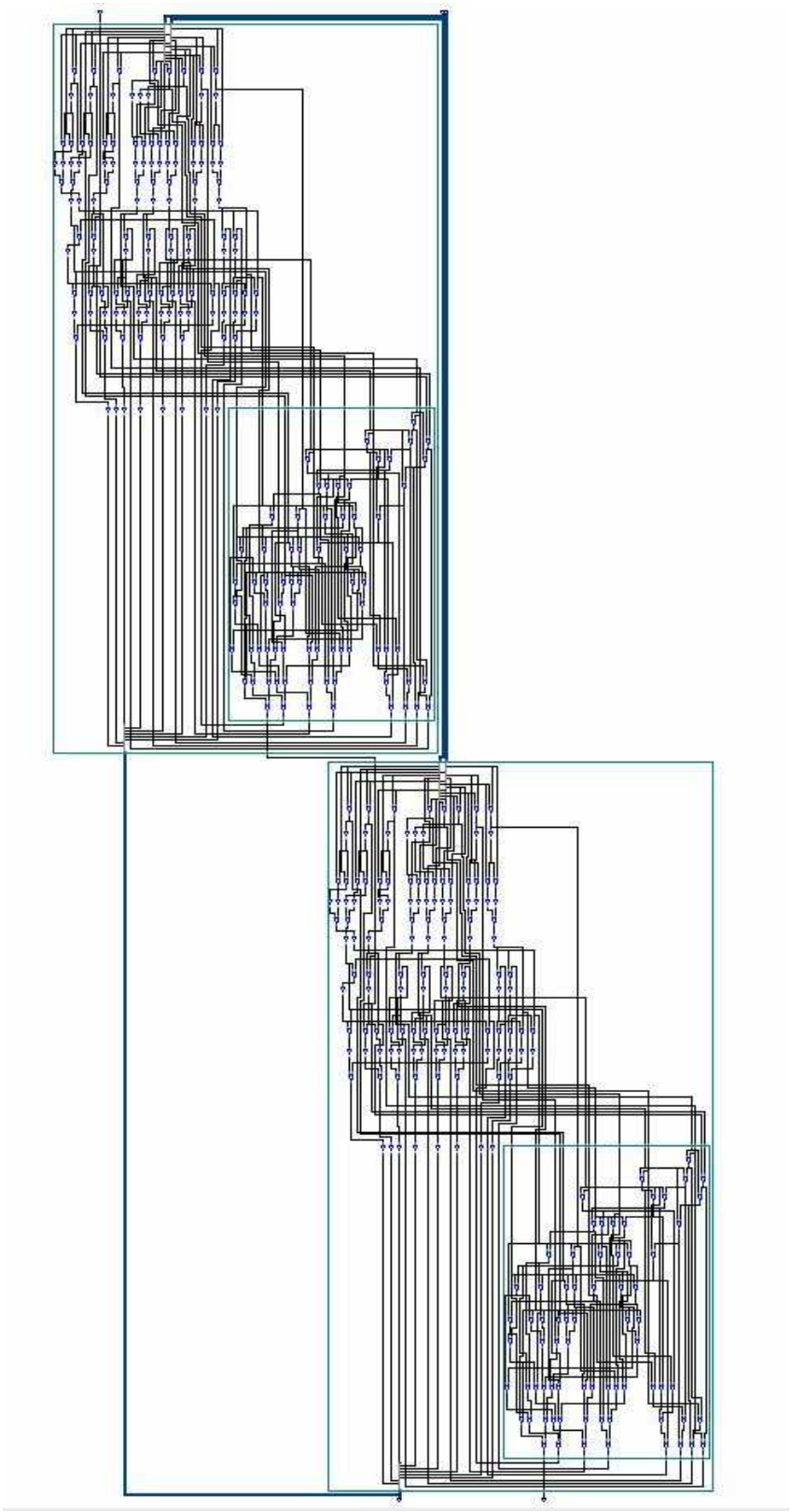


図 3.5　8 ビット長ブロック桁上げ先見回路の順次桁上げによる 16 ビット加算器の RTL 図

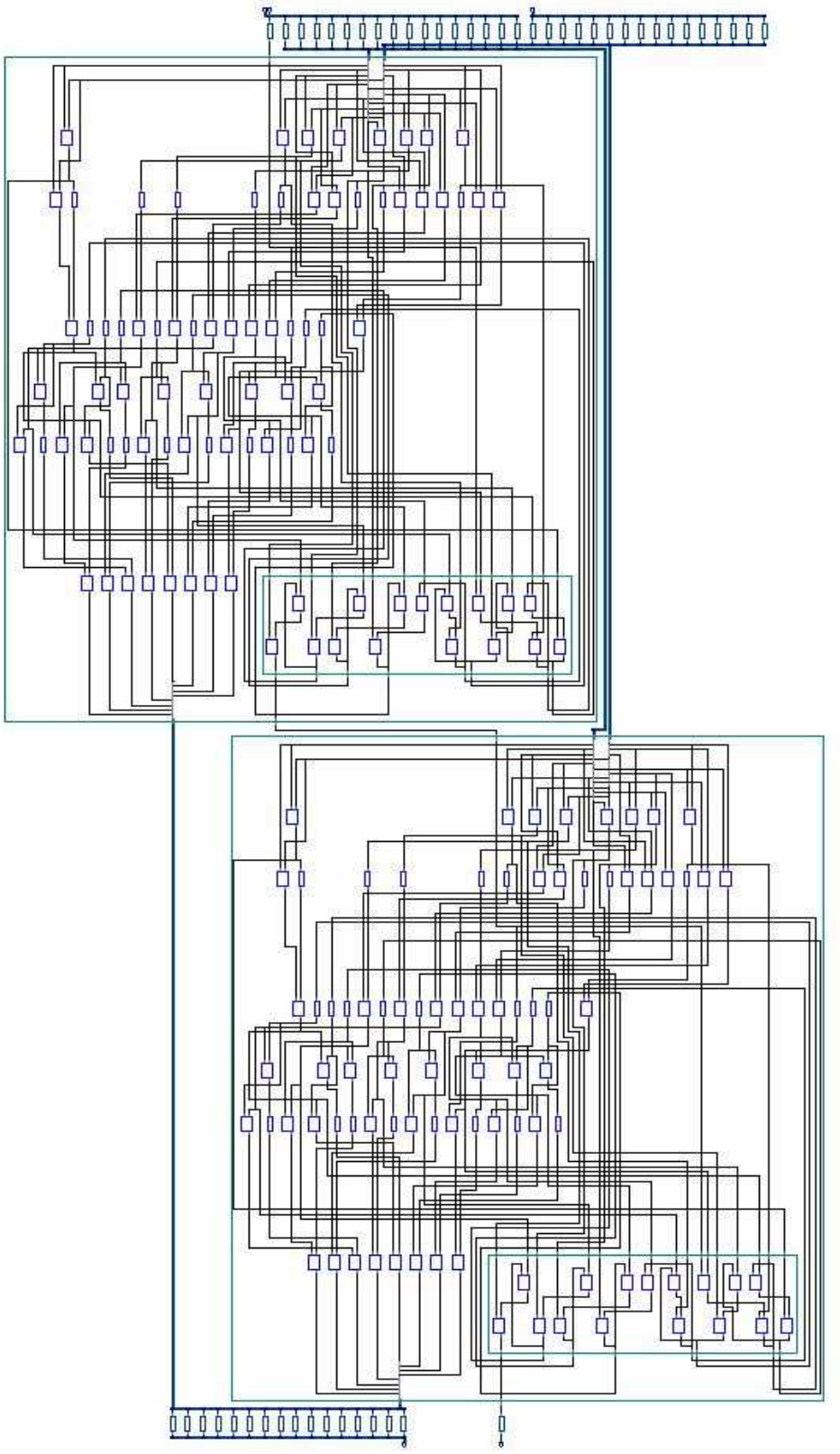


図 3.6　8 ビット長ブロック桁上げ先見回路の順次桁上げによる 16 ビット加算器の論理レイアウト（テクノロジマッピング）

3.3　桁上げ先見加算器の 2 段構成による 16 ビット加算器（１）

上位ビットの桁上げ信号は 4 ビットのようなビット数の少ない桁上げ先見回路を何段か重ねることにより効率的に見通すことが出来る。そこで、図 3.3 のように 4 ビット長ブロック桁上げ先見加算器を 4 個並列に並べ、各々から 4 ビット単位の P 信号と G 信号を出力する。その信号を 4 ビット長ブロック桁上げ先見回路に入力し、4 ビットごとの桁上げ信号を 4 ビット桁上げ先見加算器に出力するようにした。こうすることにより、4 ビットごとの加算が並列に処理できることとなり、より高速な回路を設計した。

この回路を Verilog-HDL で記述し論理合成を行ったときのゲート数は 286 個、遅延時間は 29.5⊿であった。

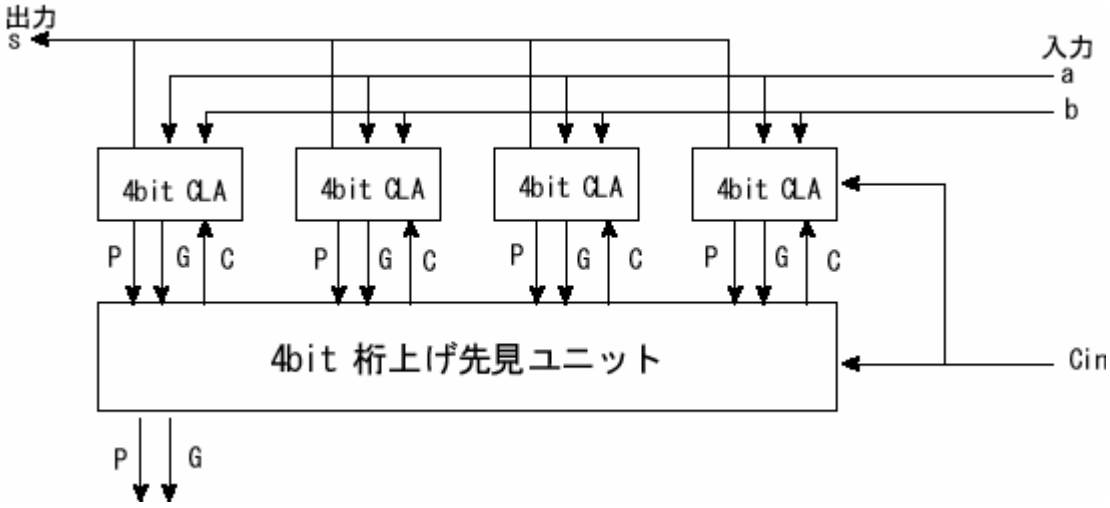


図 3.7　4 ビットブロック桁上げ先見回路と 4 ビット桁上げ先見加算器の
２段構成による 16 ビット加算器

```
/*4 ビットブロック桁上げ先見回路と 4 ビット桁上げ先見加算器の
                                        2 段構成による 16 ビット加算器*/
module Tree1(a,b,cin,s,P,G);

        input   [15:0]   a,b;
        input            cin;

        output           P,G;
        output  [15:0]   s;
        wire    [3:0]    p,g;
        wire    [2:0]    c;

CLA_4 i0(.a(a[3:0]),.b(b[3:0]),.cin(cin),.s(s[3:0]),.P(p[0]),.G(g[0]));
CLA_4 i1(.a(a[7:4]),.b(b[7:4]),.cin(c[0]),.s(s[7:4]),.P(p[1]),.G(g[1]));
CLA_4 i2(.a(a[11:8]),.b(b[11:8]),.cin(c[1]),.s(s[11:8]),.P(p[2]),.G(g[2]));
CLA_4 i3(.a(a[15:12]),.b(b[15:12]),.cin(c[2]),.s(s[15:12]),.P(p[3]),.G(g[3]));

CLU_4 i4(.p(p[3:0]),.g(g[3:0]),.cin(cin),.P(P),.G(G),.c(c[2:0]));

endmodule
/*************************4 ビット桁上げ先見加算器*****************************/
module CLA_4(a,b,cin,s,cout);
        input   [3:0] a,b;
        input         cin;
        output [3:0] s;
        output        cout;

        wire [2:0]    c;
        wire [3:0]    x,y;

        Circuit_A i0(.a(a[0]),.b(b[0]),.cin(cin),.p(x[0]),.g(y[0]),.s(s[0]));
        Circuit_A i1(.a(a[1]),.b(b[1]),.cin(c[0]),.p(x[1]),.g(y[1]),.s(s[1]));
        Circuit_A i2(.a(a[2]),.b(b[2]),.cin(c[1]),.p(x[2]),.g(y[2]),.s(s[2]));
        Circuit_A i3(.a(a[3]),.b(b[3]),.cin(c[2]),.p(x[3]),.g(y[3]),.s(s[3]));

        CLU_4 i4(.p(x),.g(y),.cin(cin),.cout(cout),.c(c));
```

```
endmodule
/*****************************4 ビット桁上げ先見ユニット*****************************/
module CLU_4 (p,g,cin,cout,c);

        input   [3:0] p,g;
        input         cin;
        output        cout;
        output [2:0] c;

assign          c[0] = g[0] | (cin & p[0]),
                c[1] = g[1] | (g[0] & p[1]) | (cin & p[0] & p[1]),
                c[2] = g[2] | (g[1] & p[2]) | (g[0] & p[1] & p[2]) |
                        (cin & p[0] & p[1] & p[2]),
                cout = g[3] | (g[2] & p[3]) | (g[1] & p[2] & p[3]) |
                        (g[0] & p[1] & p[2] & p[3]) | (cin & p[0] & p[1] & p[2] & p[3]);

endmodule
```

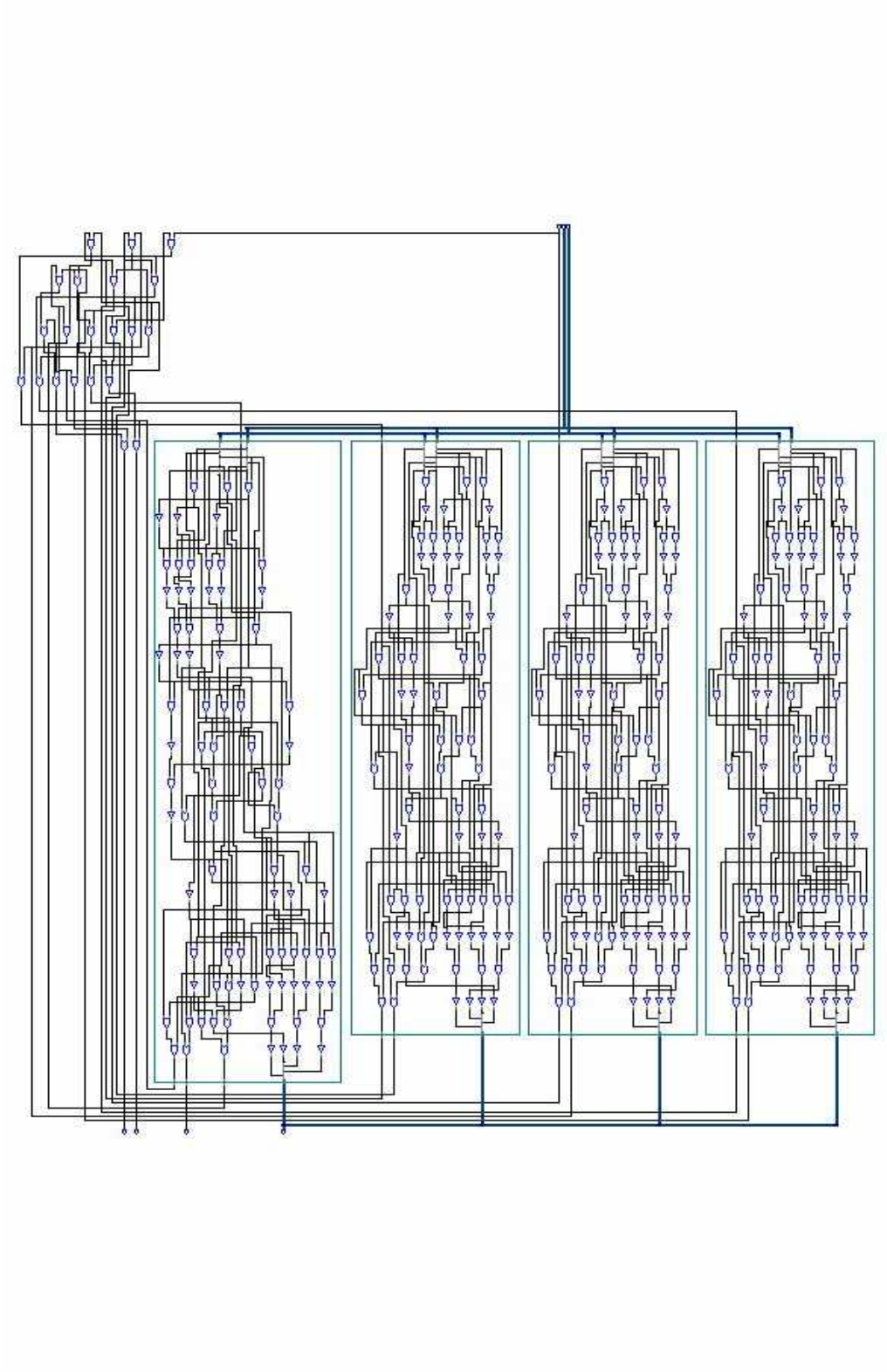


図 3.8　4 ビットブロック桁上げ先見回路と 4 ビット桁上げ先見加算器の
2 段構成による 16 ビット加算器の RTL 図

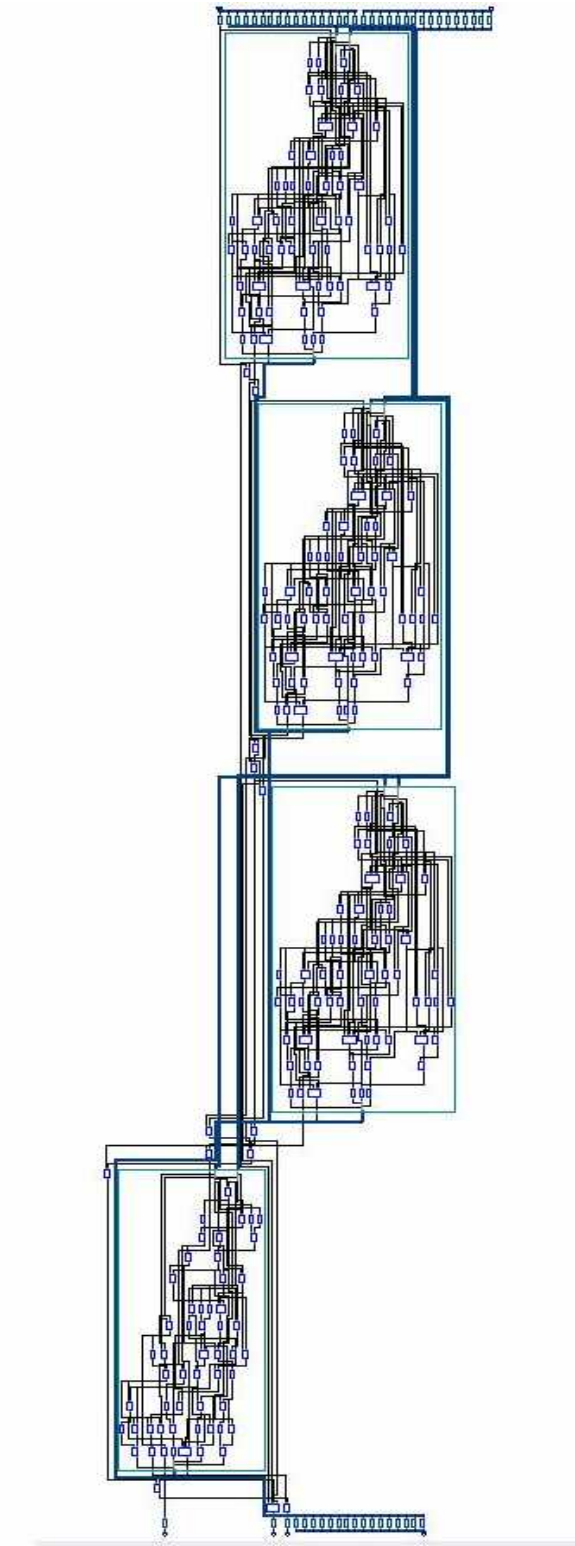


図 3.9　4 ビットブロック桁上げ先見回路と 4 ビット桁上げ先見加算器の
２段構成による 16 ビット加算器の論理レイアウト（テクノロジマッピング）

3.4　桁上げ先見加算器の 2 段構成による 16 ビット加算器（２）

次に設計したものは、２ビット長ブロック桁上げ先見回路と８ビット桁上げ先見加算器の２段構成による１６ビット加算器である（図 3.4)。この回路は８ビットごとの P、G を２ビット桁上げ先見回路で受けて、８ビットごとの桁上げ先見加算器に返す構成である。

この回路を Verilog-HDL で記述し論理合成するとゲート数は 225 個、遅延時間は 34.5⊿となった。

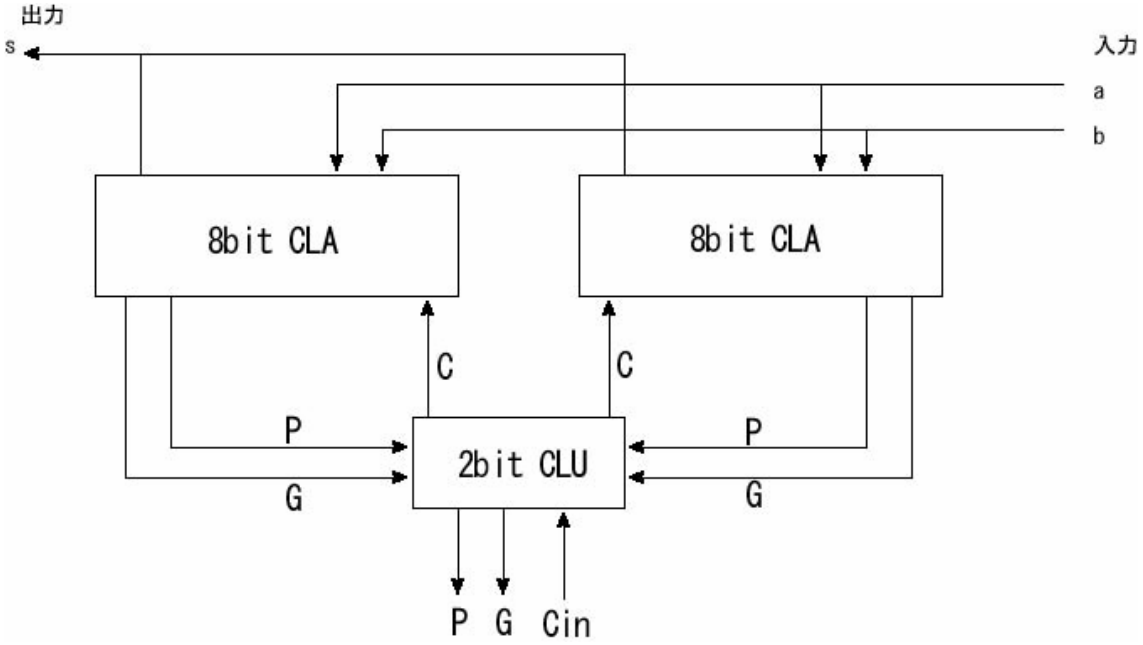


図 3.10　２ビット長ブロック桁上げ先見回路と８ビット桁上げ先見加算器の２段構成による１６ビット加算器

```
/*２ビット長ブロック桁上げ先見回路と８ビット桁上げ先見加算器の
                                        ２段構成による１６ビット加算器*/
module Tree2(a,b,cin,s,P,G);
        input   [15:0]   a,b;
        input            cin;

        output  [15:0]   s;
        output           P,G;

        wire    [1:0]    p,g;
        wire             c;
        CLA_8 i0(.a(a[7:0]),.b(b[7:0]),.cin(cin),.P(p[0]),.G(g[0]),.s(s[7:0]));
        CLA_8 i1(.a(a[15:8]),.b(b[15:8]),.cin(c),.P(p[1]),.G(g[1]),.s(s[15:8]));

        CLU_2 i2(.p0(p[0]),.g0(g[0]),.p1(p[1]),.g1(g[1]),.cin(cin),.P(P),.G(G),.cout(c));
endmodule

/**************************2 ビット桁上げ先見ユニット**************************/
module CLU_2 (p0,g0,p1,g1,cin,P,G,cout);

        input p0,p1,g0,g1,cin;
        output P,G,cout;
        wire w1,w2;

        assign  w1   = p0&cin,
                cout = w1|g0,
                P    = p0&p1,
                w2   = g0&p1,
                G    = w2|g1;
 endmodule
```

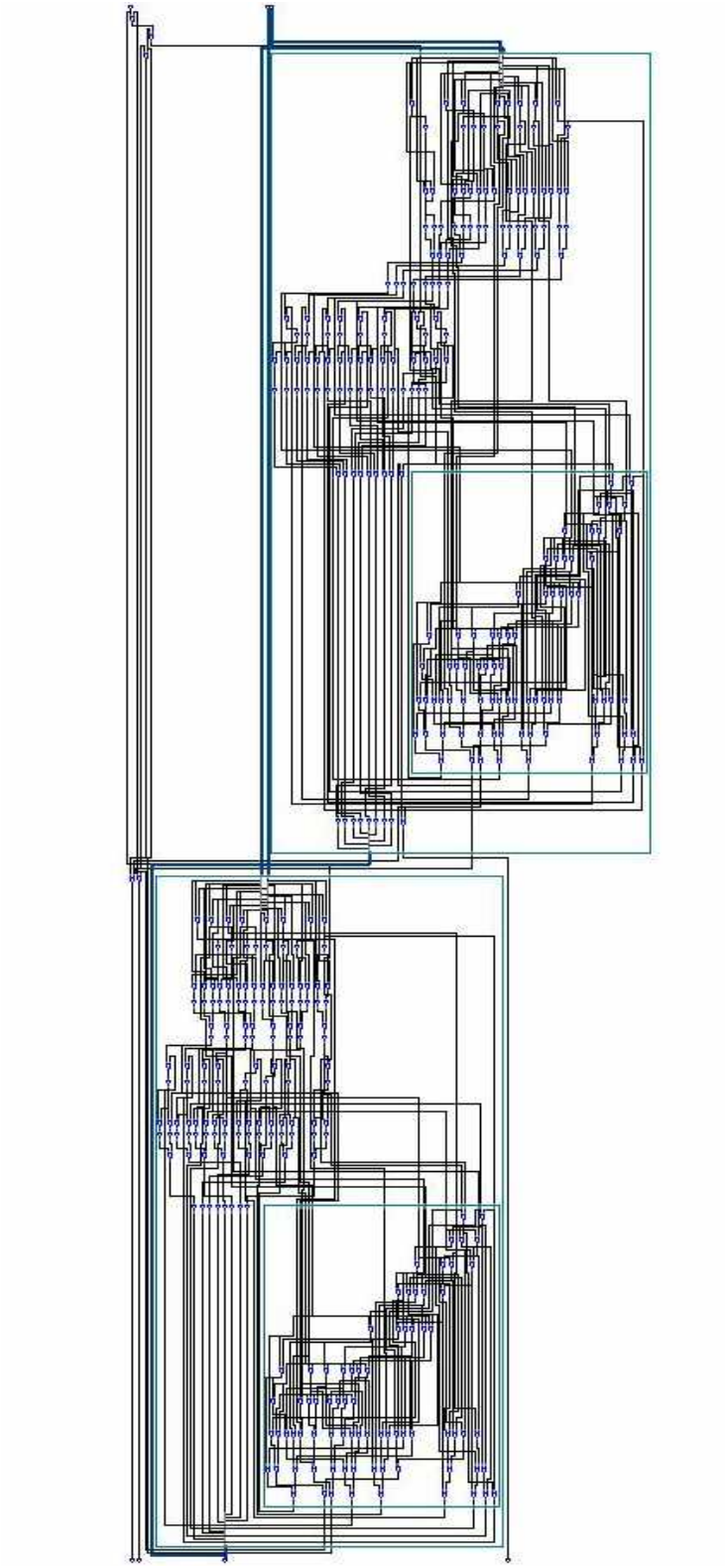


図 3.11　**２ビット長ブロック桁上げ先見回路と８ビット桁上げ先見加算器の２段構成による１６ビット加算器の** RTL **図**

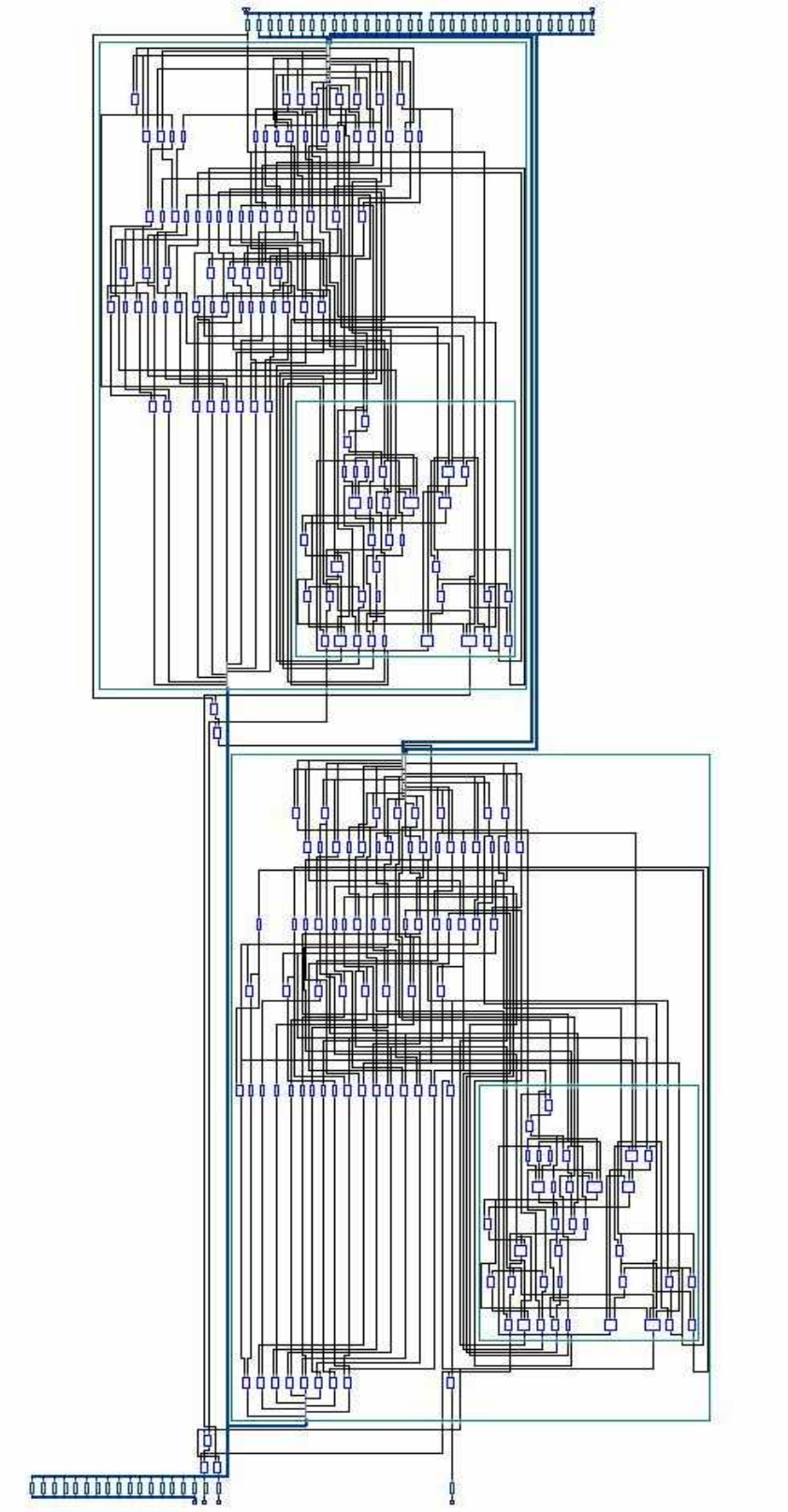


図 3.12　２ビット長ブロック桁上げ先見回路と８ビット桁上げ先見加算器の２段構成による１６ビット加算器の論理レイアウト（テクノロジマッピング）

3.5　桁上げ先見加算器の 3 段構成による 16 ビット加算器

先に述べた 2 段構成の加算器（1）では 4 ビット長ブロック桁上げ先見回路を使用していた。この回路をさらに高速化するために図 3.5 のように桁上げ先見回路 2 ビット長ブロック桁上げ先見回路を 2 段構成にして同じ働きをさせるようにした。このことにより、4 ビット長の場合よりも単純な回路を使用することができ、クリティカルパスにおいて通過するゲート数を減少することができると考えた。

この回路を Verilog-HDL で記述し、論理合成を行ったところ、ゲート数は 303 個、遅延時間は 23.0⊿となった。

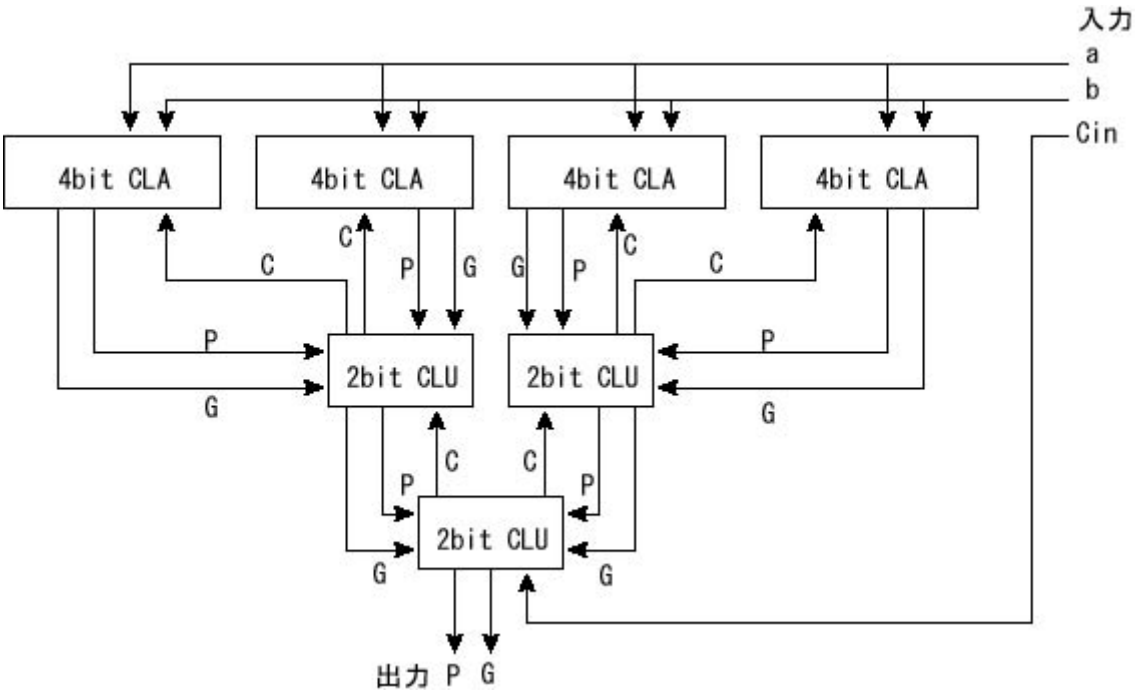


図 3.13　2 ビット長ブロック桁上げ先見回路と 4 ビット長桁上げ先見加算器の 3 段構成による 16 ビット加算器

```
/*２ビット長ブロック桁上げ先見回路と 4 ビット長桁上げ先見加算器の
                                        3 段構成による 16 ビット加算器*/
module Tree3(a,b,cin,P,G,s);

        input   [15:0]  a,b;
        input           cin;
        output          P,G;
        output  [15:0]  s;

        wire            p0,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,p13,p14,p15,
                        g0,g1,g2,g3,g4,g5,g6,g7,g8,g9,g10,g11,g12,g13,g14,g15,
                        w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12,w13,w14,w15;
        wire    [15:0]  cout;

Unit_Circuit_A  i0 (.a(a[0]),.b(b[0]),.cin(cout[0]),.p(p0),.g(g0),.s(s[0]));
Unit_Circuit_A  i1 (.a(a[1]),.b(b[1]),.cin(cout[1]),.p(p1),.g(g1),.s(s[1]));
Unit_Circuit_A  i2 (.a(a[2]),.b(b[2]),.cin(cout[2]),.p(p2),.g(g2),.s(s[2]));
Unit_Circuit_A  i3 (.a(a[3]),.b(b[3]),.cin(cout[3]),.p(p3),.g(g3),.s(s[3]));

Unit_Circuit_A  i4 (.a(a[4]),.b(b[4]),.cin(cout[4]),.p(p4),.g(g4),.s(s[4]));
Unit_Circuit_A  i5 (.a(a[5]),.b(b[5]),.cin(cout[5]),.p(p5),.g(g5),.s(s[5]));
Unit_Circuit_A  i6 (.a(a[6]),.b(b[6]),.cin(cout[6]),.p(p6),.g(g6),.s(s[6]));
Unit_Circuit_A  i7 (.a(a[7]),.b(b[7]),.cin(cout[7]),.p(p7),.g(g7),.s(s[7]));

Unit_Circuit_A  i8 (.a(a[8]),.b(b[8]),.cin(cout[8]),.p(p8),.g(g8),.s(s[8]));
Unit_Circuit_A  i9 (.a(a[9]),.b(b[9]),.cin(cout[9]),.p(p9),.g(g9),.s(s[9]));
Unit_Circuit_A  i10(.a(a[10]),.b(b[10]),.cin(cout[10]),.p(p10),.g(g10),.s(s[10]));
Unit_Circuit_A  i11(.a(a[11]),.b(b[11]),.cin(cout[11]),.p(p11),.g(g11),.s(s[11]));

Unit_Circuit_A  i12(.a(a[12]),.b(b[12]),.cin(cout[12]),.p(p12),.g(g12),.s(s[12]));
Unit_Circuit_A  i13(.a(a[13]),.b(b[13]),.cin(cout[13]),.p(p13),.g(g13),.s(s[13]));
Unit_Circuit_A  i14(.a(a[14]),.b(b[14]),.cin(cout[14]),.p(p14),.g(g14),.s(s[14]));
Unit_Circuit_A  i15(.a(a[15]),.b(b[15]),.cin(cout[15]),.p(p15),.g(g15),.s(s[15]));

Unit_Circuit_B  i16(.cin(cin),.p0(p0),.p1(p1),.p2(p2),.p3(p3),
                        .g0(g0),.g1(g1),.g2(g2),.g3(g3),.cout(cout[3:0]),
```

```
                    .P(w1),.G(w2));
Unit_Circuit_B  i17(.cin(w11),.p0(p4),.p1(p5),.p2(p6),.p3(p7),
                    .g0(g4),.g1(g5),.g2(g6),.g3(g7),.cout(cout[7:4]),
                    .P(w3),.G(w4));
Unit_Circuit_B  i18(.cin(w15),.p0(p8),.p1(p9),.p2(p10),.p3(p11),
                    .g0(g8),.g1(g9),.g2(g10),.g3(g11),.cout(cout[11:8]),
                    .P(w5),.G(w6));
Unit_Circuit_B  i19(.cin(w14),.p0(p12),.p1(p13),.p2(p14),.p3(p15),
                    .g0(g12),.g1(g13),.g2(g14),.g3(g15),.cout(cout[15:12]),
                    .P(w7),.G(w8));

Unit_Circuit_C  i20(.p0(w1),.p1(w3),.g0(w2),.g1(w4),
                    .cin(cin),.P(w9),.G(w10),.cout(w11));
Unit_Circuit_C  i21(.p0(w5),.p1(w7),.g0(w6),.g1(w8),
                    .cin(w15),.P(w12),.G(w13),.cout(w14));

Unit_Circuit_C  i22(.p0(w9),.p1(w12),.g0(w10),.g1(w13),
                    .cin(cin),.P(P),.G(G),.cout(w15));
endmodule

/*****************************1 ビット桁上げ先見回路*****************************/
module Circuit_A (a,b,cin,p,g,s);

        input           a,b,cin;
        output          s,p,g;

        wire w0, w1, w2, w3, w4, w5;

        assign  w0 = ~(a & b),
                g  = ~w0,
                w1 = ~(a & w0),
                w2 = ~(b & w0),
                p  = ~(w1 & w2),

                w3 = ~(cin & p),
                w4 = ~(cin & w3),
```

```
                w5 = ~(p & w3),
                s  = ~(w4 & w5);
endmodule


/*************************2 ビット桁上げ先見ユニット*****************************/
module Unit_Circuit_B (cin,p0,p1,p2,p3,g0,g1,g2,g3,P,G,cout);
        input           p0,p1,p2,p3,g0,g1,g2,g3,cin;
        output          P,G;
        output  [3:0]   cout;
assign
        P = p0 & p1 & p2 & p3,
        G = g3 | (p3 & g2) | (p3 & p2 & g1) | (p3 & p2 & p1 & g0),
        cout[0] = cin,
        cout[1] = g0 | (cin & p0),
        cout[2] = g1 | (g0 & p1) | (cin & p0 & p1),
        cout[3] = g2 | (g1 & p2) | (g0 & p1 & p2) | (cin & p0 & p1 & p2);
endmodule

/*************************2 ビット桁上げ先見ユニット*****************************/
module Unit_Circuit_C(p0,g0,p1,g1,cin,P,G,cout);

        input p0,p1,g0,g1,cin;
        output P,G,cout;

        wire w1,w2;

        assign  w1 = p0&cin,
                        cout = w1|g0,
                        P = p0&p1,
                        w2 = g0&p1,
                        G = w2|g1;
 endmodule
```

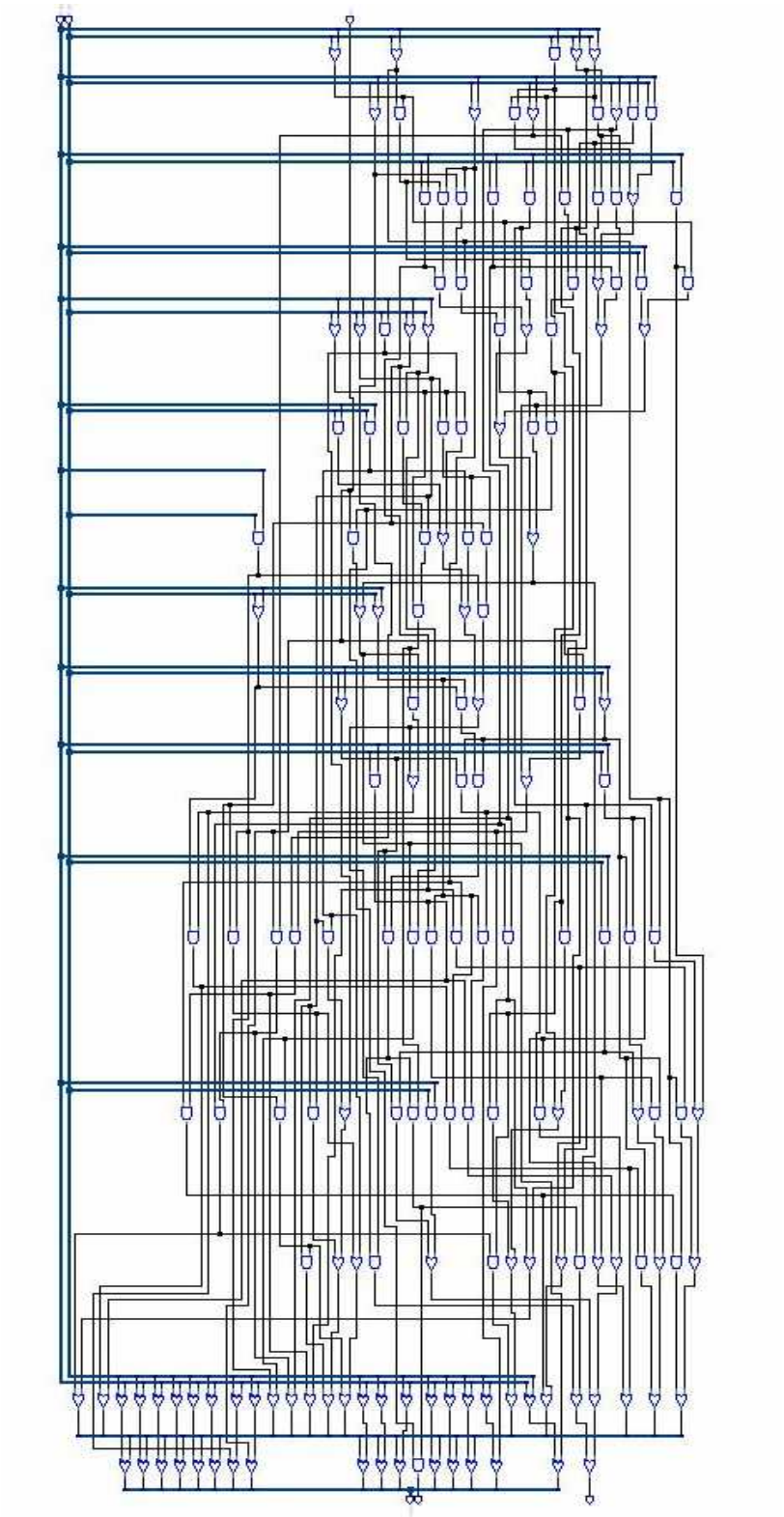


図 3.14　２ビット長ブロック桁上げ先見回路と 4 ビット長桁上げ先見加算器の
3 段構成による 16 ビット加算器の RTL 図

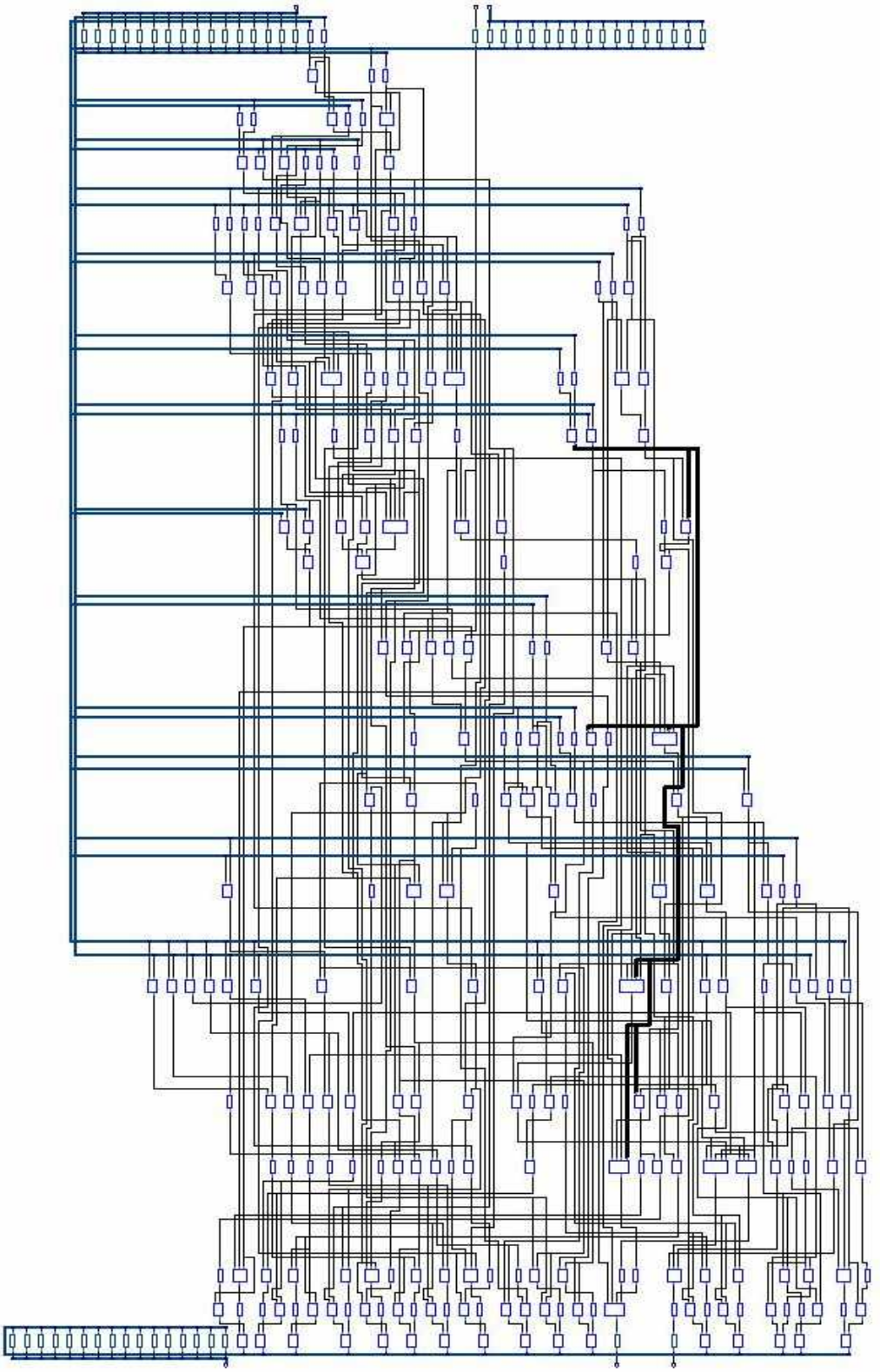


図 3.15　**２ビット長ブロック桁上げ先見回路と** 4 **ビット長桁上げ先見加算器の** 3 **段構成による** 16 **ビット加算器の論理レイアウト（テクノロジマッピング）**

3.6　桁上げ選択加算器（Carry Select Adder）

桁上げ信号が 0 の場合と 1 の場合を 2 個の加算器を使って計算結果を予測しておき、信号が決まったところでマルチプレクサを使用することによってどちらかを選択する。通常、図 3.6 のように桁上げ先見加算器とマルチプレクサを複数使用する。この方法によって、4 ビットごとの加算を並列に処理することができる。また、下位からの桁上げ信号が 0 の場合と 1 の場合両方を出力し、桁上げ判定回路によって上位への桁上げ信号を決定する。その信号を次の 4 ビット加算器の下位からの桁上げ信号 1 の場合、0 の場合のそれぞれの答えと共にマルチプレクサに入力してどちらかの答えを出力する。このとき、4 ビットごとの答えの出力の遅延は桁上げ判定回路における遅延時間の差となる。

この回路は高速に動作することが期待できるが、ビット長が大きくなるほど桁上げ判定回路は複雑になってしまう。

この回路を Verilog-HDL で記述して論理合成を行ったところ、ゲート数は 344 個、遅延時間は 24.5⊿となった。

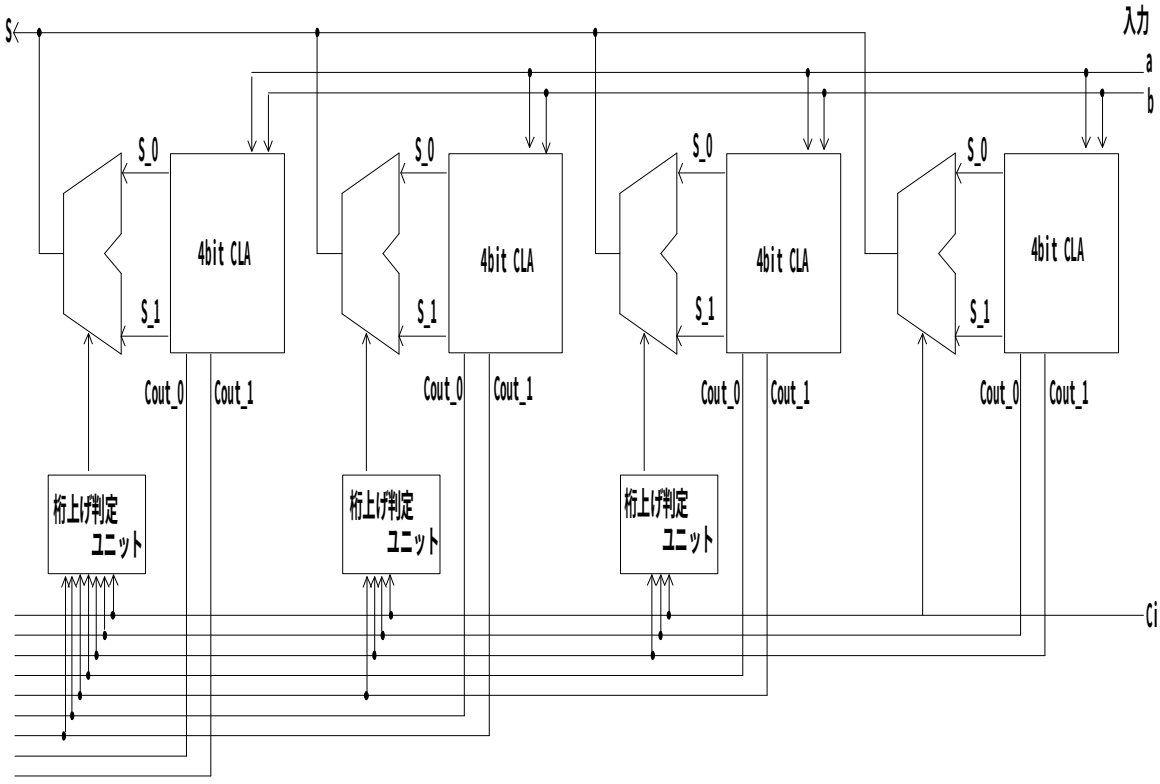


図 3.16　16 ビット桁上げ選択加算器

```
/*16 ビット桁上げ選択加算器*/
module CSA(a,b,cin,P,G,s);

        input    [15:0]    a,b;
        input              cin;
        output   [15:0]    s;
        output             P,G;

        wire     [15:0]    s0;
        wire               p1,p2,p3,p4,g1,g2,g3,g4;
        wire               carry1,carry2,carry3;

CLA   i0 (.a(a[3:0]),.b(b[3:0]),.s0(s0[3:0]),.P(p1),.G(g1));
CLA   i1 (.a(a[7:4]),.b(b[7:4]),.s0(s0[7:4]),.P(p2),.G(g2));
CLA   i2 (.a(a[11:8]),.b(b[11:8]),.s0(s0[11:8]),.P(p3),.G(g3));
CLA   i3 (.a(a[15:12]),.b(b[15:12]),.s0(s0[15:12]),.P(P),.G(G));

Carry i4 (.cin(cin),.p1(p1),.p2(p2),.p3(p3),.g1(g1),.g2(g2),.g3(g3),
                .carry1(carry1),.carry2(carry2),.carry3(carry3));

mux   i5 (.x(s[3:0]),.a(s0[3:0]),.cin(cin));
mux   i6 (.x(s[7:4]),.a(s0[7:4]),.cin(carry1));
mux   i7 (.x(s[11:8]),.a(s0[11:8]),.cin(carry2));
mux   i8 (.x(s[15:12]),.a(s0[15:12]),.cin(carry3));

endmodule

/* 4 ビット桁上げ先見加算器 */
module CLA (a,b,s0,P,G);
        input    [3:0]     a,b;

        output   [3:0]     s0;
        output             P,G;

        wire               c00,c10,c20,p0,p1,p2,p3,g0,g1,g2,g3;
```

```
Circuit_A       i0 (.a(a[0]),.b(b[0]),.p(p0),.g(g0),.s(s0[0]));
Circuit_B       i1 (.a(a[1]),.b(b[1]),.cin(c00),.p(p1),.g(g1),.s(s0[1]));
Circuit_B       i2 (.a(a[2]),.b(b[2]),.cin(c10),.p(p2),.g(g2),.s(s0[2]));
Circuit_B       i3 (.a(a[3]),.b(b[3]),.cin(c20),.p(p3),.g(g3),.s(s0[3]));

CLU             i4 (.g0(g0),.g1(g1),.g2(g2),.g3(g3),
                    .p0(p0),.p1(p1),.p2(p2),.p3(p3),
                    .c0(c00),.c1(c10),
                    .c2(c20),.P(P),.G(G));

endmodule

/*1 ビット桁上げ先見加算器 A*/
module Circuit_A (a,b,p,g,s);

    input       a,b;
    output      s,p,g;

assign
    s = a ^ b,
    p = a | b,
    g = a & b;
endmodule

/*1 ビット桁上げ先見加算器 B*/
module Circuit_B (a,b,cin,p,g,s);

    input       a,b,cin;
    output      s,p,g;

assign
    s = a ^ b ^cin,
    p = a | b,
    g = a & b;
endmodule
```

```
/*4 ビット長ブロック桁上げ先見ユニット*/
module CLU (g0,g1,g2,g3,p0,p1,p2,p3,c0,c1,c2,P,G);

input  g0,g1,g2,g3;
input  p0,p1,p2,p3;

output          c0,c1,c2,P,G;



assign
        c0 = g0,
        c1 = g1 | (g0 & p1),
        c2 = g2 | (g1 & p2) | (g0 & p1 & p2),
        G   = g3 | (g2 & p3) | (g1 & p2 & p3) | (g0 & p1 & p2 & p3),
        P   = p0 & p1 & p2 & p3;
endmodule

/*桁上げ判定ユニット*/
module Carry (cin,p1,p2,p3,g1,g2,g3,carry1,carry2,carry3);
        input    cin,p1,p2,p3,g1,g2,g3;
        output   carry1,carry2,carry3;
assign
        carry1 = g1 | (p1 & cin),
        carry2 = g2 | (p2 & carry1),
        carry3 = g3 | (p3 & carry2);
endmodule
```

```
/*マルチプレクサ*/
module mux (x,a,cin);
        input   [3:0]   a;
        input   cin;
        output [3:0]    x;

        reg [3:0]    x;
        always @(a or cin) begin
                if (cin == 1'b0) x = a;
                else                    x = a+1'b1;
        end
endmodule
```

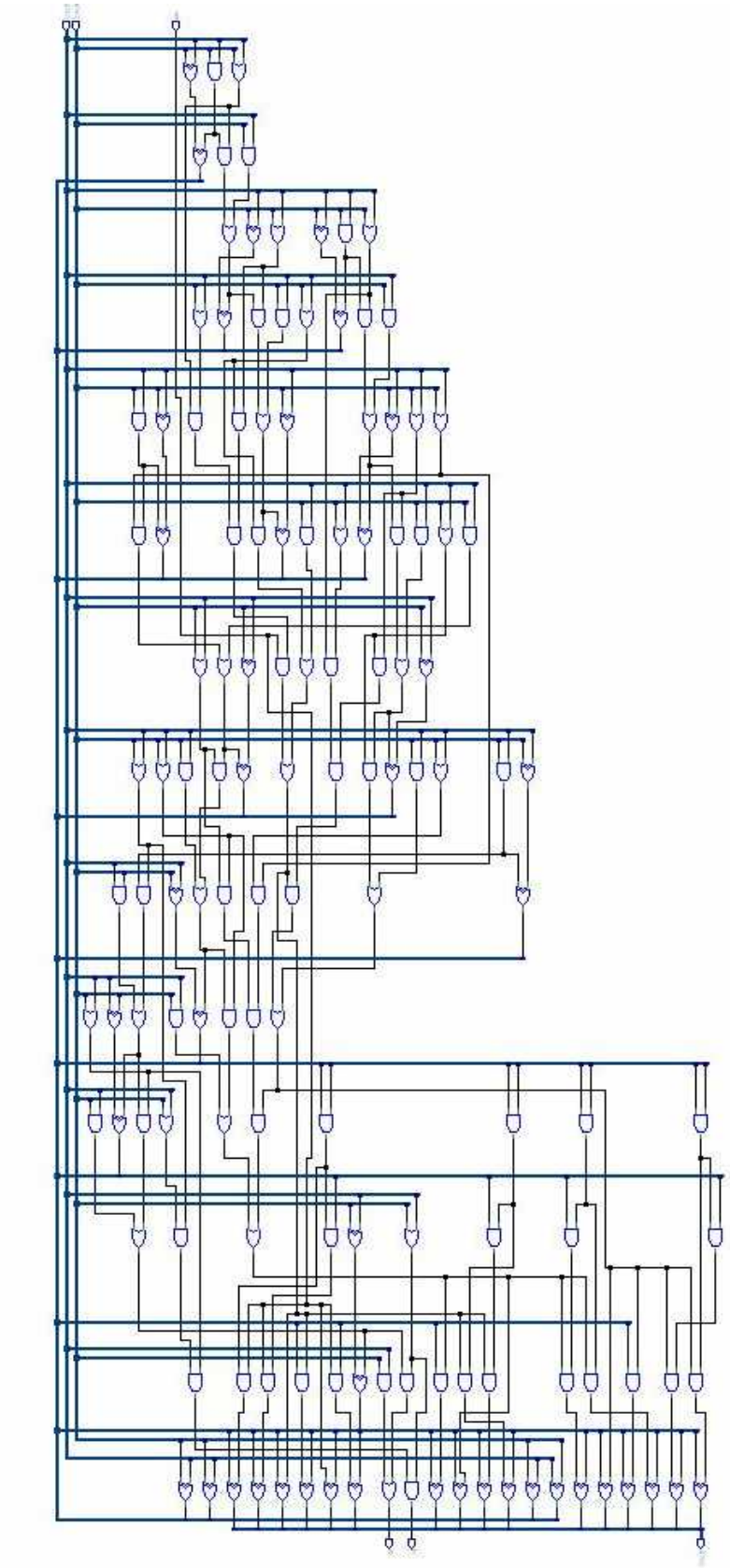


図 3.17　16 ビット桁上げ選択加算器の RTL 図

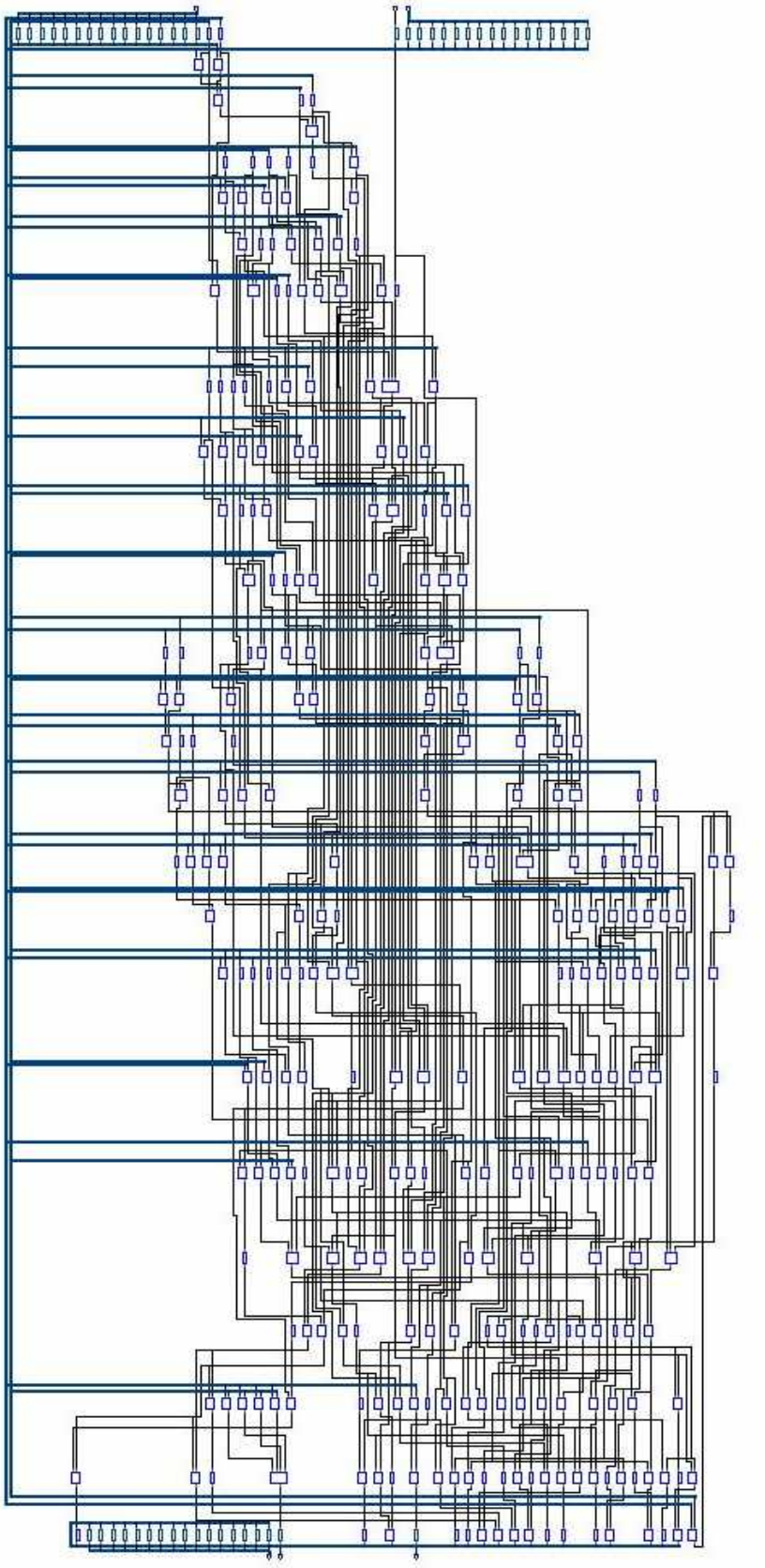


図 3.18　16 ビット桁上げ選択加算器の論理レイアウト（テクノロジマッピング）

4　冗長 2 進加算器

4.1　　冗長 2 進数表現

冗長 2 進数表現とは、{-1、0、1}の冗長な 2 進数を利用する表現である。最下位の桁は 2^0 の重みを持ち、下位から i 番目の桁は 2^{i-1} の重みを持つ。そのため各桁は通常 2 進表現では 1 と 0 だけだが、冗長 2 進表現では-1 の値を持つ。これを利用し、冗長 2 進表現では 1 つの数字をいくつかの方法で表すことが可能である。例えば 10 進数の 5 を冗長 2 進数 4 桁で表す場合、「0101」SD2、「10-1-1」SD2、「1-101」SD2 などがあり、1 つの数に対して幾通りかに表すことが出来る。

冗長 2 進数体系で、この冗長性を利用して、2 進の加算において桁上げの伝搬を高々1 桁にする方法がある。これは次の 2 つのステップで可能となる。

(1)第一ステップでは、各桁で、被加数 x_i と加数 y_i から $x_i + y_i = 2c_i + s_i$ となるように中間桁上げ $c_i(\in\{-1,0,1\})$ と中間和 $s_i(\in\{-1,0,1\})$ を定める。このとき、中間和 s_i と 1 つ下位からの中間桁上げ c_{i-1} がともに 1 あるいは-1 となることがないように、次の表 4.1 の規則に従い、1 つ下位の x_{i-1} と y_{i-1} も調べて c_i と s_i を決定する。

表 4.1　　冗長 2 進体系での計算規則

被加数 x_i	加数 y_i	一つ下位の桁 x_{i-1} y_{i-1}	中間桁上げ c_i	中間和 s_i
1	1		1	0
1	0	両方とも非負	1	-1
0	1	少なくとも一方負	0	1
0	0		0	0
1	-1		0	0
-1	1		0	0
0	-1	両方とも非負	0	-1
-1	0	少なくとも一方負	-1	1
-1	-1		-1	0

(2)第 2 ステップでは、各桁で $s_i + c_{i-1} = z_i$ となるように和 $z_i(\in\{-1,0,1\})$を求める。
このとき表 4.1 の規則より、新たな桁上げは生じない。図 4.1 に桁上げが伝搬しない加算の例を示す。和 z_i は $x_i, y_i, x_{i-1}, y_{i-1}, x_{i-2}, y_{i-2}$ の 6 つから決まるので、組み合わせ回路による並列加算が演算数の桁数 n(ビット長 n)に関係なく一定の数で行える。また、通常 2 進数は各桁が 0 か 1 であるから、そのままで冗長 2 進数とみなすことができる。

被加数		[1	0	−1	0	−1	0	0	−1]	(87)
加数	＋	[1	−1	1	0	0	1	1	−1]	(101)
中間和		0	1	0	0	−1	−1	1	0	
中間桁上げ	＋	1	−1	0	0	0	1	0	−1	
和		1	−1	1	0	0	0	−1	0	0	(188)

図 4.1　　桁上げが伝搬しない加算の例

4.1.1　冗長 2 進加算器の構成(1)

冗長 2 進加算器は、全加算器を基本セルとして、順次桁上げ加算器のように単純 1 列で構成することが出来る。しかも計算を並列に行うので、ビット長に関係なく計算時間は一定であるという優れた特徴を持つ。しかし、3 値の冗長 2 進数を 2 値の通常 2 進数で表現すると 2 倍の信号を必要とするため、加算器を構成する場合大幅なゲート数の増加を必要とする。図 4.2 は冗長 2 進加算器の例である。

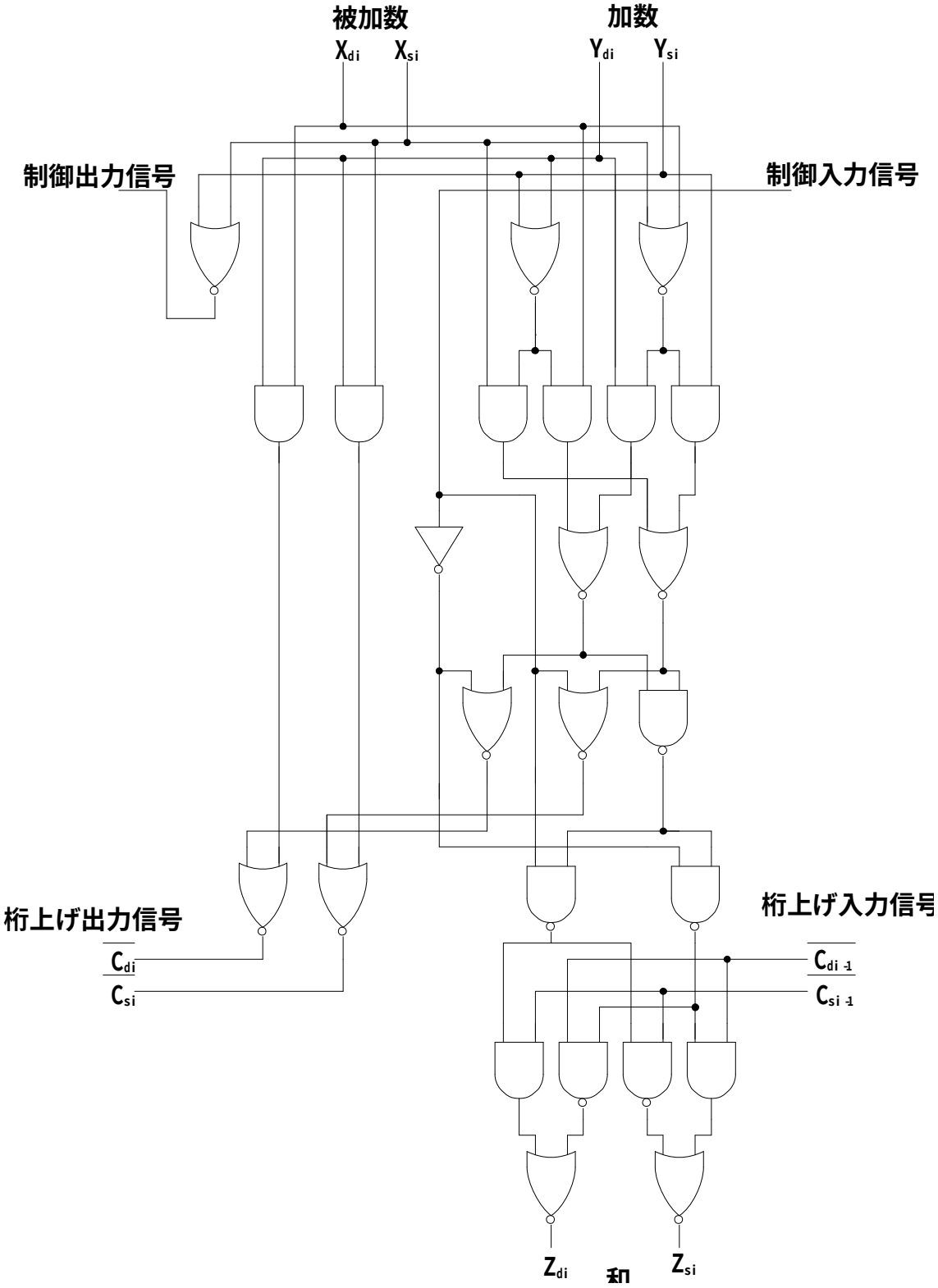


図 4.2　従来の冗長 2 進加算器の論理回路図

4.2 冗長 2 進加算器の簡略化

前述にもあるように、冗長 2 進加算器の最大の課題は、通常 2 進加算器と比較して構成に要するゲート数が非常に多くなるという点である。ゲート数の増加は、単に加算器の占有面積が大きくなるばかりでなく、信号伝搬の遅延時間も大きくなり、冗長 2 進加算器の優れた特徴である高速処理が実現できなくなる。そこで、冗長 2 進アルゴリズムに新しい方法を導入して新規な冗長 2 進加算器を提案し、その優位性を議論する。

4.2.1 冗長 2 進加算器の構成(2)

表 4.1.の冗長 2 進数体系での計算規則から、中間和 s_i ($\in\{-1,0,1\}$)と中間桁上げ c_i ($\in\{-1,0,1\}$)は次に示す関係がある。

(a) 1 つ下位の桁の被加数 x_{i-1} と加数 y_{i-1} が共に非負($\in\{0,1\}$)の時、
　(ⅰ)中間和 s_i は非正($\in\{-1,0\}$)で、且つ
　(ⅱ)1 つ下位の桁の中間桁上げ c_{i-1} は非負($\in\{0,1\}$)である。

(b) 1 つ下位の桁の被加数 x_{i-1} と加数 y_{i-1} のうち少なくとも一方が負の時、
　(ⅰ)中間和 s_i は非負で、且つ
　(ⅱ)1 つ下位の桁の中間桁上げ c_{i-1} は非正である。

よって、上記(a)あるいは(b)が成立するとき、中間和 s_i および中間桁上げ c_{i-1} は冗長 2 進数{-1,0,1}のうち 2 値{0,1}あるいは{-1,0}で表現できる。これは加算器を構成するとき、より少ないゲート数で実現できることを意味する。次に、この方法を議論する。まず 2 値を持つ変数 P_{i-1}($\in\{0,1\}$)を導入する。変数 P_{i-1} は上記(a)が成立するとき $P_{i-1}=0$、上記(b)が成立するとき $P_{i-1}=1$ となる変数である。さらに、式(4－1)、および式(4－2)を導入して、非負($\in\{0,1\}$)の変数 u_i、および v_{i-1} を定義する。変数 u_i、および v_{i-1} は各々中間和 s_i、および中間桁上げ c_{i-1} に対応している。なお、式(4－2)の＋は算術和である。

$$u_i = P_{i-1} - s_i \qquad (4-1)$$

$$v_{i-1} = P_{i-1} + c_{i-1} \qquad (4-2)$$

変数 P_{i-1}、被加数 x_{i-1}、加数 y_{i-1}、中間和 s_i、中間桁上げ c_{i-1}、および変数 u_i、v_i の関係をまとめると表 4.2 のようになる。

表 4.2　　変数 P_{i-1}、被加数 x_{i-1}、加数 y_{i-1}、中間和 s_i、中間桁上げ c_{i-1}、および変数 u_i、v_i の関係

変数 (P_{i-1})	被加数、加数 (x_{i-1}, y_{i-1})	中間和 (s_i)	一つ下位の桁の中間桁上げ (c_{i-1})	変数 (u_i, v_{i-1})
0	{0,1}	{-1,0}	{0,1}	{0,1}
1	少なくとも一方は-1	{0,1}	{-1,0}	{0,1}

本来 3 値である変数 s_i および c_{i-1} を 2 値の変数 u_i および v_{i-1} で表現出来ることは、加算器がより簡単に構成できることを示している。次にその具体例を示す。

3 値{-1,0,1}を持つ冗長 2 進表現は、表現ビット数が通常の符号なし 2 進表現{0,1}の 2 倍必要である。以下では、一般に冗長 2 進数の 1 桁 t_i を 2 ビットの符号なし 2 進数(2 値変数) t_i^s、t_i^a とし、表 4.3 に示すように冗長 2 進数 t_i の“ -1 ”、“ 0 ”、“ 1 ”を各々t_i^s、t_i^a の“ 11 ”“ 00 ”“ 01 ”に割り当てる。ここで、添字 s は符号ビットを、符号 a は絶対値部のビットの表現に相当している。また、ここでは、t_i は x_i、y_i、s_i、z_i を示す。

表 4.3　　冗長 2 進数 t_i と 2 進数 t_i^s、t_i^a との対応

t_i	t_i^s	t_i^a
-1	1	1
0	0	0
1	0	1

変数 P_i は表 4.2 より、変数 u_i、および中間和 s_i の絶対値 s_i^a は各々式(４－１)、および表 4.1 より、変数 vi は表 4.4 より、各々式(４－３)～(４－５)で表される。なお、表 4.4 は変数 P_i、P_{i-1}、被加数 x_{i-1}、加数 y_{i-1}、中間和 s_i、中間桁上げ c_{i-1}、および変数 u_i、v_i の関係を示し、被加数 x_i、加数 y_i、中間和 s_i、中間桁上げ c_i は表 4.1 から、変数 P_i は式(４－３)から、変数 v_i は式(４－２)の中間桁上げ c_i および変数 P_i との関係から、変数 P_{i-1} は式(４－４)の中間和 s_i との関係、および表 4.2 の中間和 s_i との関係から導かれる。

$$P_i = x_i^s + y_i^s \qquad (4-3)$$

$$u_i = s_i^a \oplus P_{i-1} \qquad (4-4)$$

$$s_i^a = x_i^a \oplus y_i^a$$

$$v_i = s_i \cdot \overline{P_{i-1}} + \overline{x_i^s \cdot y_i^s} \cdot x_i^a \cdot y_i^a \qquad (4-5)$$

表 4.4　変数 P_{i-1}、被加数 x_{i-1}、加数 y_{i-1}、中間和 s_i、中間桁上げ c_{i-1}、および変数 u_i、v_i の関係

被加数 (x_i^s,x_i^a)	加数 (y_i^s,y_i^a)	中間桁上げ (c_i)	中間和 (s_i)	変数 (P_i)	変数 (v_i)	変数 (P_{i-1})	変数 (u_i)
0 1	0 1	0 1	0 0	0	1	0	0
						1	1
0 1	0 0	0 1	1 1	0	1	0	1
0 0	0 1	0 0	0 1	0	0	1	0
0 0	0 0	0 0	0 0	0	0	0	0
						1	1
0 1	1 1	0 0	0 0	1	1	0	0
						1	1
1 1	0 1	0 0	0 0	1	1	0	0
						1	1
0 0	1 1	0 0	1 1	1	1	0	1
1 1	0 0	1 1	0 1	1	0	1	0
1 1	1 1	1 1	0 0	1	0	0	0
						1	1

さらに、和 z_i は、 $z_i = s_i + c_{i-1}$ および式(４－１)、式(４－２)から得られる。

$$z_i^s = u_i \cdot \overline{v_{i-1}}$$

$$z_i^a = u_i \oplus v_{i-1} \qquad (4-6)$$

式(４－３)～(４－６)から冗長 2 進加算器は図 4.3 のように表せる。図からも分かるように、図 4.2 と比較してみるとゲート数が大幅に削減出来ている。これは本来 3 値である冗長 2 進加算器の内部変数 P_i、u_i、および v_i が 2 値で実現出来たことに由来している。前述したよう冗長 2 進加算器と同様に、和 z_i は x_i , y_i , x_{i-1}, y_{i-1} , x_{i-2}, y_{i-2} の 6 つのみから決まる。これは信号伝搬が 3 個の冗長 2 進加算器のみで決定されることを意味し、やはり計算時間はビット長に依存せず一定となる。

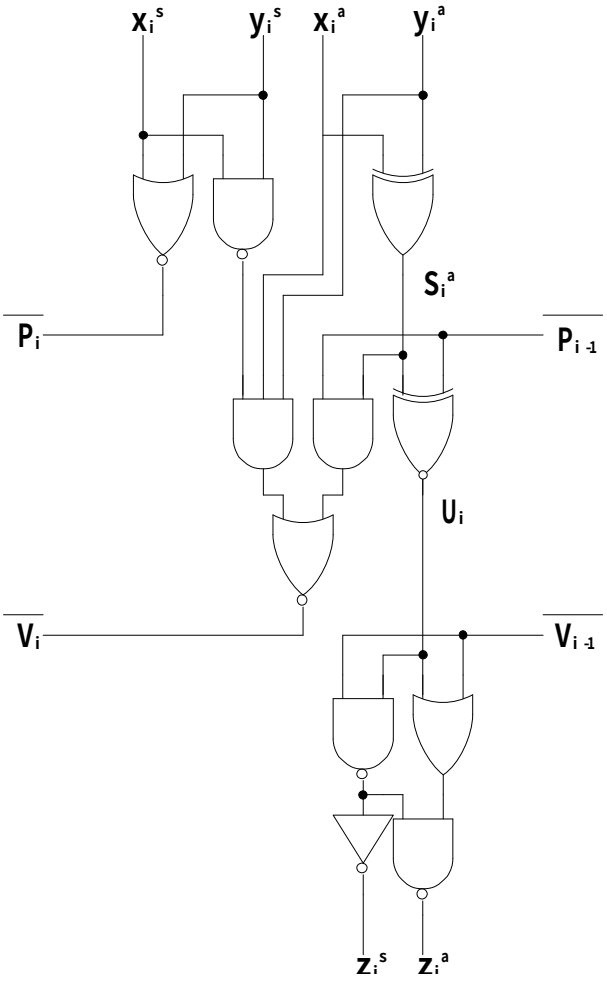


図 4.3　　新規な冗長 2 進加算器の論理回路図

```
/*新規な 1 ビット長冗長 2 進加算器*/
module RBA(x,y,z,pin,vin,pout,vout);
        input   [1:0] x,y;
        input         pin,vin;
        output [1:0] z;
        output        pout,vout;

        wire w0,w1,w2,w3,w4,w5,w6;

        assign  pout = ~( x[1] | y[1]),
                w0   = ~( x[1] & y[1]),
                w1   = x[0] ^ y[0],
                w2   = w0 & x[0] & y[0],
                w3   = w1 & pin,
                vout = ~(w2 | w3),
                w4   = ~(w1 ^ pin),
                w5   = ~(w4 & vin),
                w6   = w4 | vin,
                z[1] = ~w5,
                z[0] = ~(w5 & w6);
endmodule
```

```
/*1 ビット長冗長 2 進加算器による 4 ビット加算器*/
module SD2_4_adder(x,y,z,pin,vin,pout,vout);
        input   [7:0] x,y;
        input         pin,vin;
        output [7:0] z;
        output        pout,vout;

        wire w0,w1,w2,w3,w4,w5,w6,w7;

        SD2_adder i0(.x(x[1:0]),.y(y[1:0]),.z(z[1:0]),.pin(pin)
                                        ,.vin(vin),.pout(w0),.vout(w1));
```

```
        SD2_adder i1(.x(x[3:2]),.y(y[3:2]),.z(z[3:2]),.pin(w0)
                                                ,.vin(w1),.pout(w2),.vout(w3));
        SD2_adder i2(.x(x[5:4]),.y(y[5:4]),.z(z[5:4]),.pin(w2)
                                                ,.vin(w3),.pout(w4),.vout(w5));
        SD2_adder i3(.x(x[7:6]),.y(y[7:6]),.z(z[7:6]),.pin(w4)
                                                ,.vin(w5),.pout(pout),.vout(vout));

endmodule
```

/*1 ビット長冗長 2 進加算器による 16 ビット加算器*/

```
module SD2_16_adder(x,y,z,pin,vin,pout,vout);
        input   [31:0] x,y;
        input          pin,vin;
        output [31:0] z;
        output         pout,vout;

        wire w0,w1,w2,w3,w4,w5,w6,w7;

        SD2_4_adder i0(.x(x[7:0]),.y(y[7:0]),.z(z[7:0]),.pin(pin)
                                                ,.vin(vin),.pout(w0),.vout(w1));
        SD2_4_adder i1(.x(x[15:8]),.y(y[15:8]),.z(z[15:8]),.pin(w0)
                                                ,.vin(w1),.pout(w2),.vout(w3));
        SD2_4_adder i2(.x(x[23:16]),.y(y[23:16]),.z(z[23:16]),.pin(w2)
                                                ,.vin(w3),.pout(w4),.vout(w5));
        SD2_4_adder i3(.x(x[31:24]),.y(y[31:24]),.z(z[31:24]),.pin(w4)
                                                ,.vin(w5),.pout(pout),.vout(vout));

endmodule
```

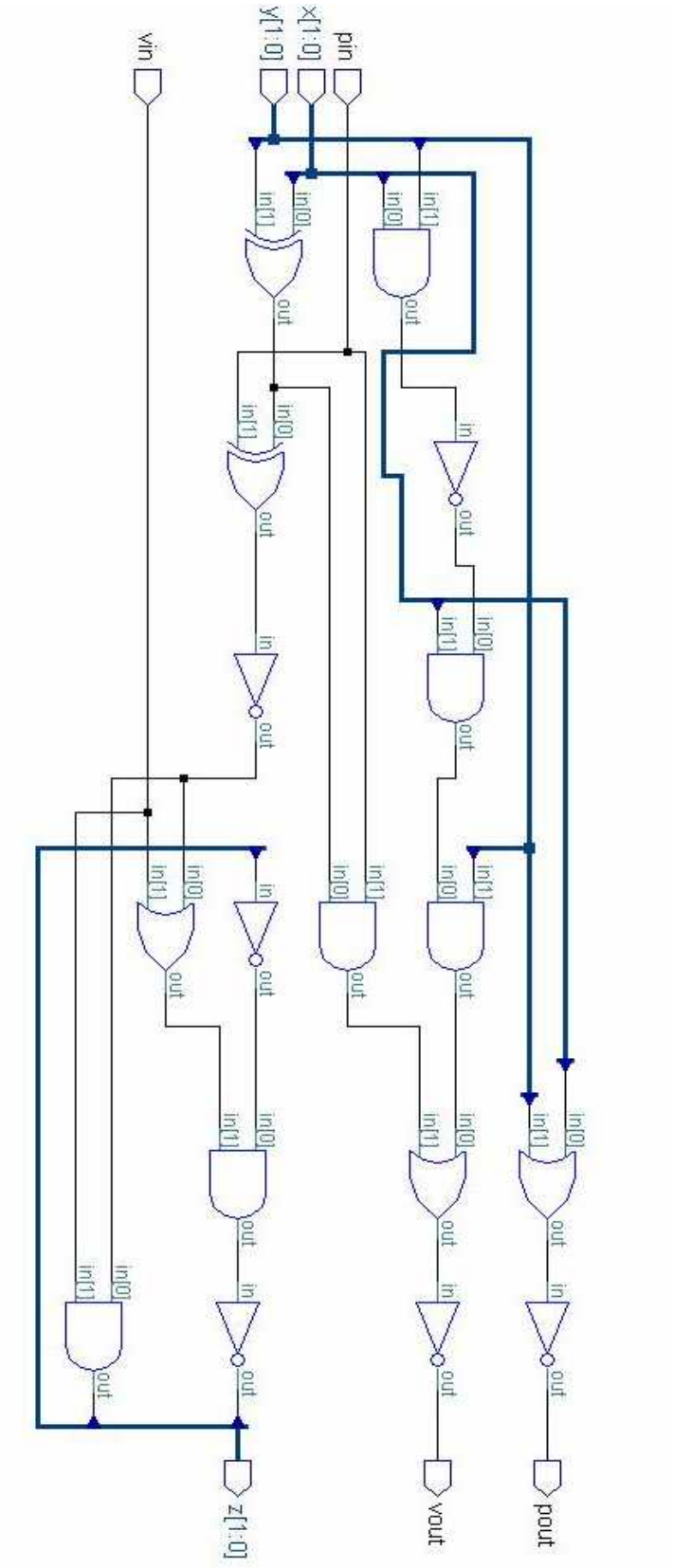


図 4.4　新規な 1 ビット長冗長 2 進加算器の RTL 図

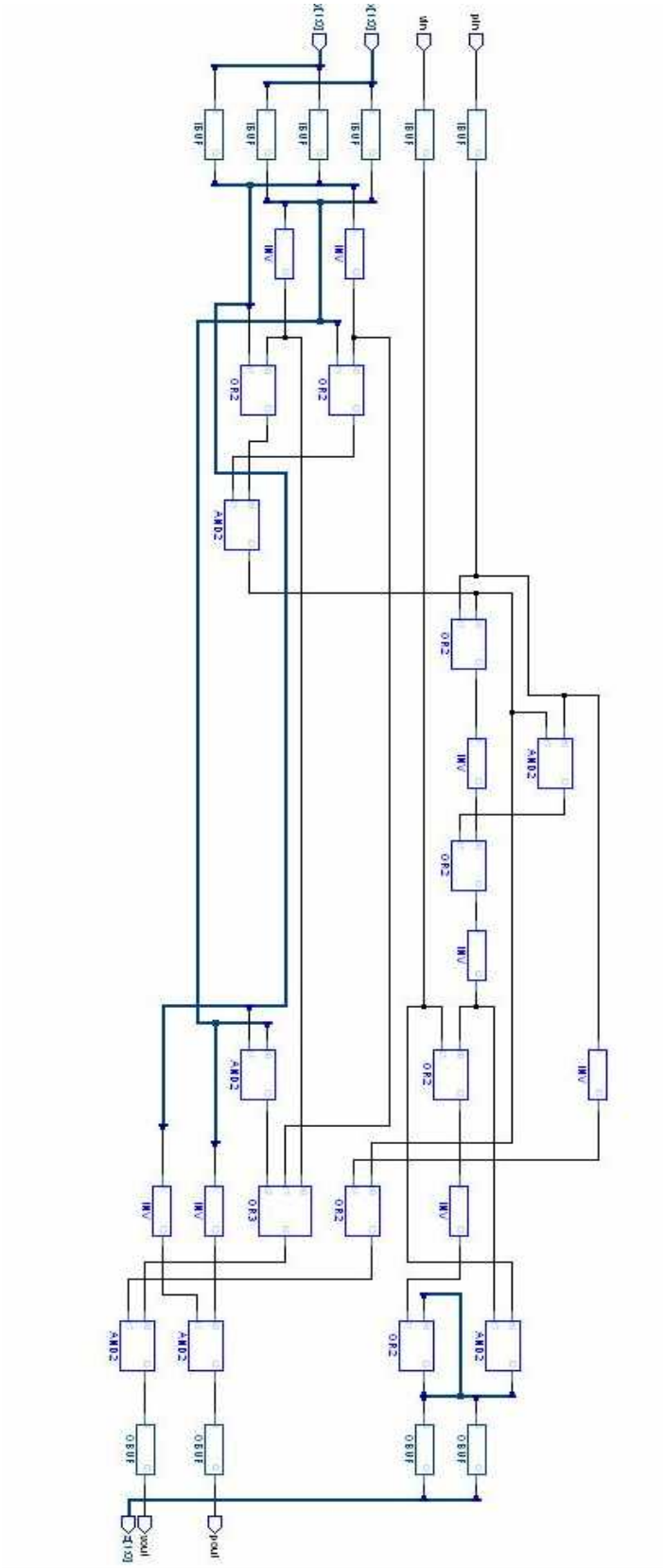


図 4.5　新規な 1 ビット長冗長 2 進加算器の論理レイアウト（テクノロジマッピング）

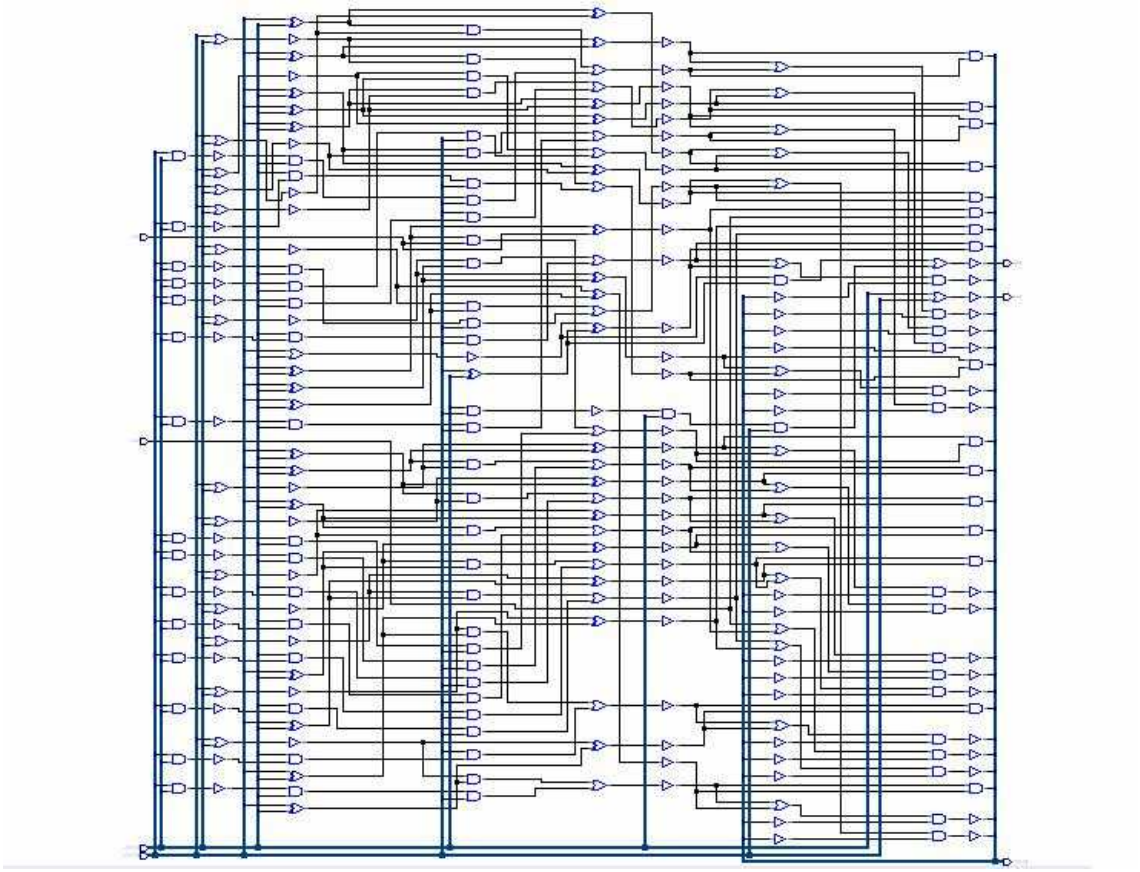


図 4.6　1 ビット長冗長 2 進加算器による 16 ビット加算器の RTL 図

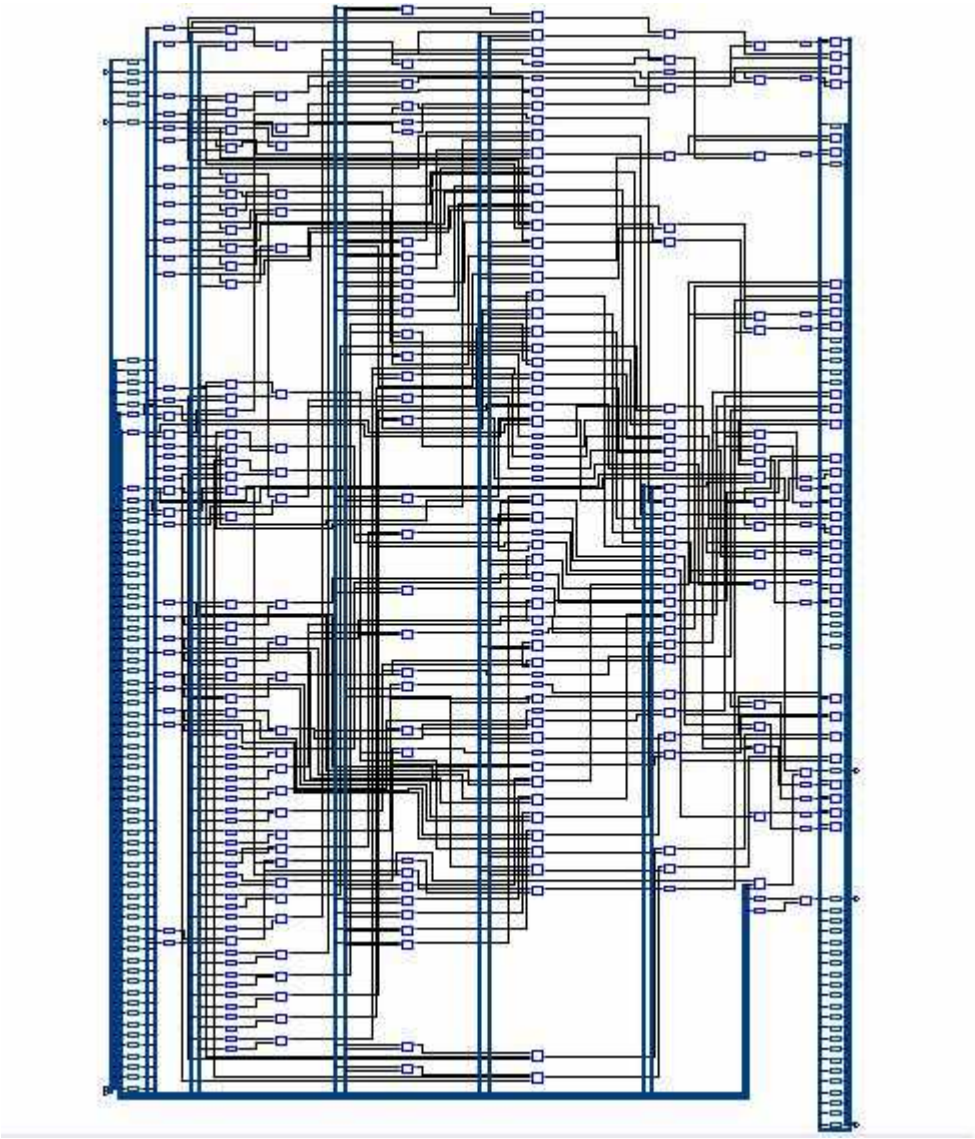


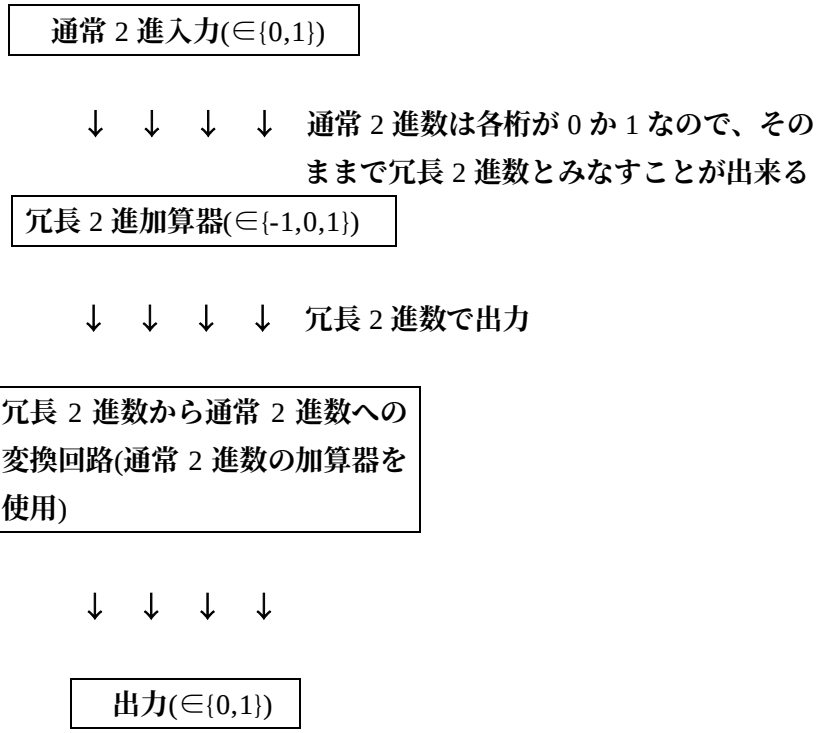
図 4.7　1 ビット長冗長 2 進加算器による 16 ビット加算器の論理レイアウト
（テクノロジマッピング）

4.3　冗長 2 進数から通常 2 進数への変換

冗長 2 進加算器では結果が冗長 2 進数で出力されるので、通常 2 進数への変換回路が必要とされる。冗長 2 進を通常 2 進へ変換するには、冗長 2 進表現で 1 である桁だけを 1 とする 2 進数から-1 である桁だけを 1 とする 2 進数を引く減算が必要であり、冗長 2 進加算器以外の加算器を用いる必要がある。

この過程で通常の 2 進数体系の加算器を使わなければならず、結果的には加算の桁上げ伝搬の遅延問題は解決されていない。演算の結果を計算機から出力するためには通常 2 進数表現が必要であるが、計算機内部での演算処理方法については何ら制限がないため、冗長 2 進数による演算の高速化は、除算器のように連続して部分加算が多数必要な演算にその有用性が発揮される。下に冗長 2 進加算器を使用した場合の入力から出力までの簡単な図を示す。

図 4.8　冗長 2 進加算器を使用した場合の入力から出力までの流れ



/*冗長 2 進数から通常 2 進数への変換回路を加えた冗長 2 進加算器による 16 ビット加算器

```
*/
module RBA2(x,y,z,cin,vout,cout0,cout1);
        input   [15:0] x,y;
        input          cin;
        output [15:0] z;
        output         vout,cout0,cout1;

        wire [31:0]    s;

        SD2_16_adder i0(.x(x),.y(y),.z(s),.vout(vout));

        SD2_Decorder i1(.a(s),.b(z),.cin(cin),.cout0(cout0),.cout1(cout1));

endmodule
```

/*冗長 2 進数から通常 2 進数への変換器(ここでは変換器に 4 ビット長ブロック桁上げ先見回路と 2 ビット桁上げ先見加算木の 3 段構成による 16 ビット加算器を使用した)*/

```
module SD2_Decorder(a,b,cin,cout0,cout1);
        input   [31:0]  a;
        input           cin;
        output  [15:0]  b;
        output          cout0,cout1;
        wire    [15:0]  x,y,y1;

  Decoder_SD2_1 i0 (.a(a[1:0]),.b(x[0]));
  Decoder_SD2_1 i1 (.a(a[3:2]),.b(x[1]));
  Decoder_SD2_1 i2 (.a(a[5:4]),.b(x[2]));
  Decoder_SD2_1 i3 (.a(a[7:6]),.b(x[3]));
  Decoder_SD2_1 i4 (.a(a[9:8]),.b(x[4]));
  Decoder_SD2_1 i5 (.a(a[11:10]),.b(x[5]));
  Decoder_SD2_1 i6 (.a(a[13:12]),.b(x[6]));
  Decoder_SD2_1 i7 (.a(a[15:14]),.b(x[7]));
  Decoder_SD2_1 i8 (.a(a[17:16]),.b(x[8]));
```

```
   Decoder_SD2_1 i9 (.a(a[19:18]),.b(x[9]));
   Decoder_SD2_1 i10 (.a(a[21:20]),.b(x[10]));
   Decoder_SD2_1 i11 (.a(a[23:22]),.b(x[11]));
   Decoder_SD2_1 i12 (.a(a[25:24]),.b(x[12]));
   Decoder_SD2_1 i13 (.a(a[27:26]),.b(x[13]));
   Decoder_SD2_1 i14 (.a(a[29:28]),.b(x[14]));
   Decoder_SD2_1 i15 (.a(a[31:30]),.b(x[15]));

   Decoder_SD2_2 i16 (.a(a[1:0]),.b(y[0]));
   Decoder_SD2_2 i17 (.a(a[3:2]),.b(y[1]));
   Decoder_SD2_2 i18 (.a(a[5:4]),.b(y[2]));
   Decoder_SD2_2 i19 (.a(a[7:6]),.b(y[3]));
   Decoder_SD2_2 i20 (.a(a[9:8]),.b(y[4]));
   Decoder_SD2_2 i21 (.a(a[11:10]),.b(y[5]));
   Decoder_SD2_2 i22 (.a(a[13:12]),.b(y[6]));
   Decoder_SD2_2 i23 (.a(a[15:14]),.b(y[7]));
   Decoder_SD2_2 i24 (.a(a[17:16]),.b(y[8]));
   Decoder_SD2_2 i25 (.a(a[19:18]),.b(y[9]));
   Decoder_SD2_2 i26 (.a(a[21:20]),.b(y[10]));
   Decoder_SD2_2 i27 (.a(a[23:22]),.b(y[11]));
   Decoder_SD2_2 i28 (.a(a[25:24]),.b(y[12]));
   Decoder_SD2_2 i29 (.a(a[27:26]),.b(y[13]));
   Decoder_SD2_2 i30 (.a(a[29:28]),.b(y[14]));
   Decoder_SD2_2 i31 (.a(a[31:30]),.b(y[15]));

assign y1=~y+1'b1;

   Tree3 i32(.a(x),.b(y1),.cin(cin),.P(P),.G(G),.s(b));

endmodule
```

/* Decoder_SD2_1 回路*/

```
module Decoder_SD2_1(a,b);
        input    [1:0]     a;
        output             b;
        reg                b1;

always @(a)
begin
case (a[1:0])
        2'b00:   b1 <= 1'b0;
        2'b01:    b1 <= 1'b1;
        2'b11:    b1 <= 1'b0;
        default:b1 <= 1'bx;
endcase
end
assign b = b1;
endmodule



/* Decoder_SD2_2 回路*/
module Decoder_SD2_2(a,b);
        input    [1:0]     a;
        output             b;
        reg                b1;

always @(a)
begin
case (a[1:0])
        2'b00:   b1 <= 1'b0;
        2'b01:    b1 <= 1'b0;
        2'b11:    b1 <= 1'b1;
        default:b1 <= 1'bx;
endcase
end
assign b = b1;
```

```
endmodule
```

冗長 2 進加算器、4 ビット長ブロック桁上げ先見回路と 2 ビット桁上げ先見加算木の 3 段構成による 16 ビット加算器は上記で記述したものを利用した。

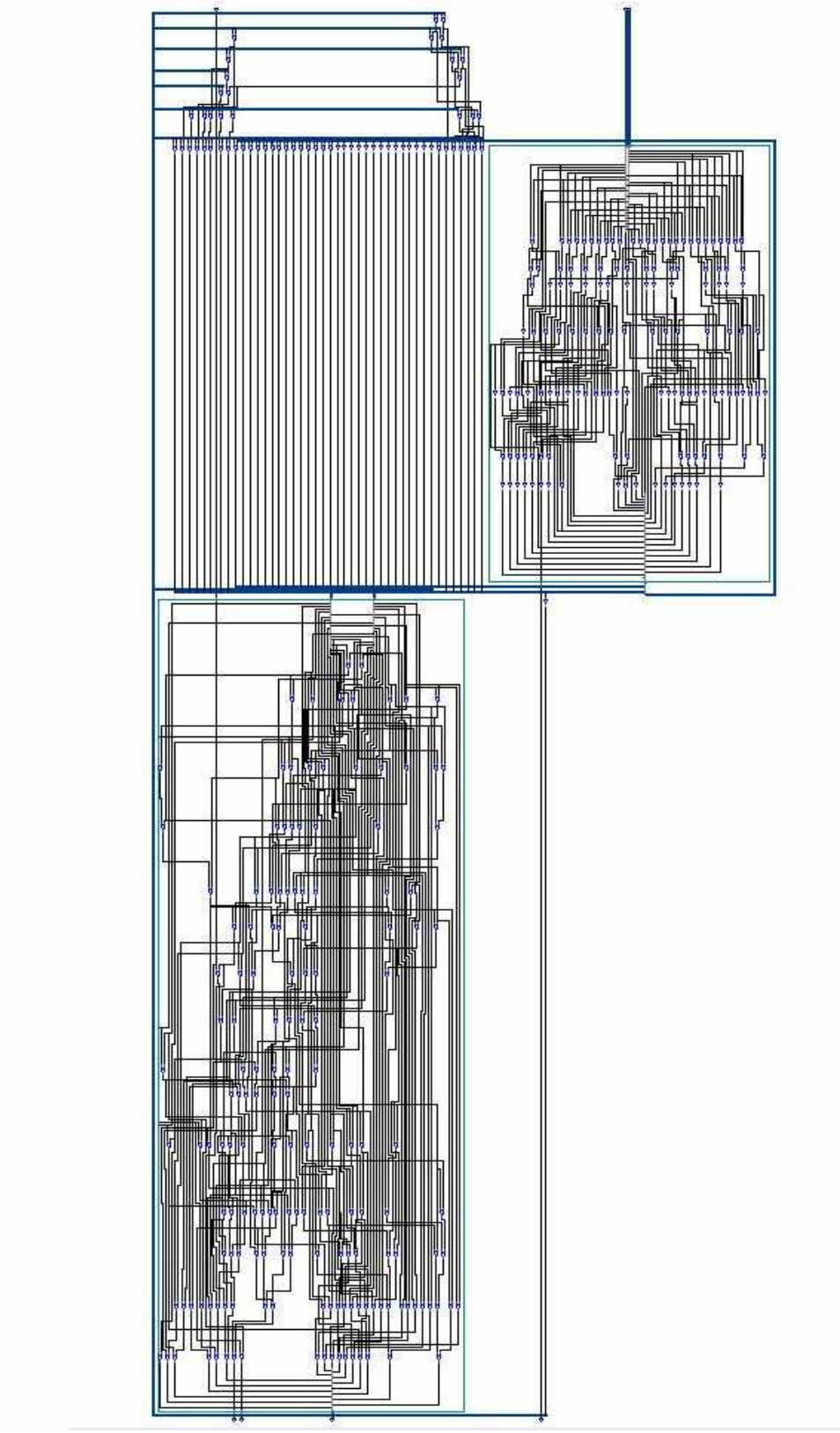


図 4.9　冗長 2 進数から通常 2 進数への変換回路を加えた冗長 2 進加算器による 16 ビット加算器の RTL 図

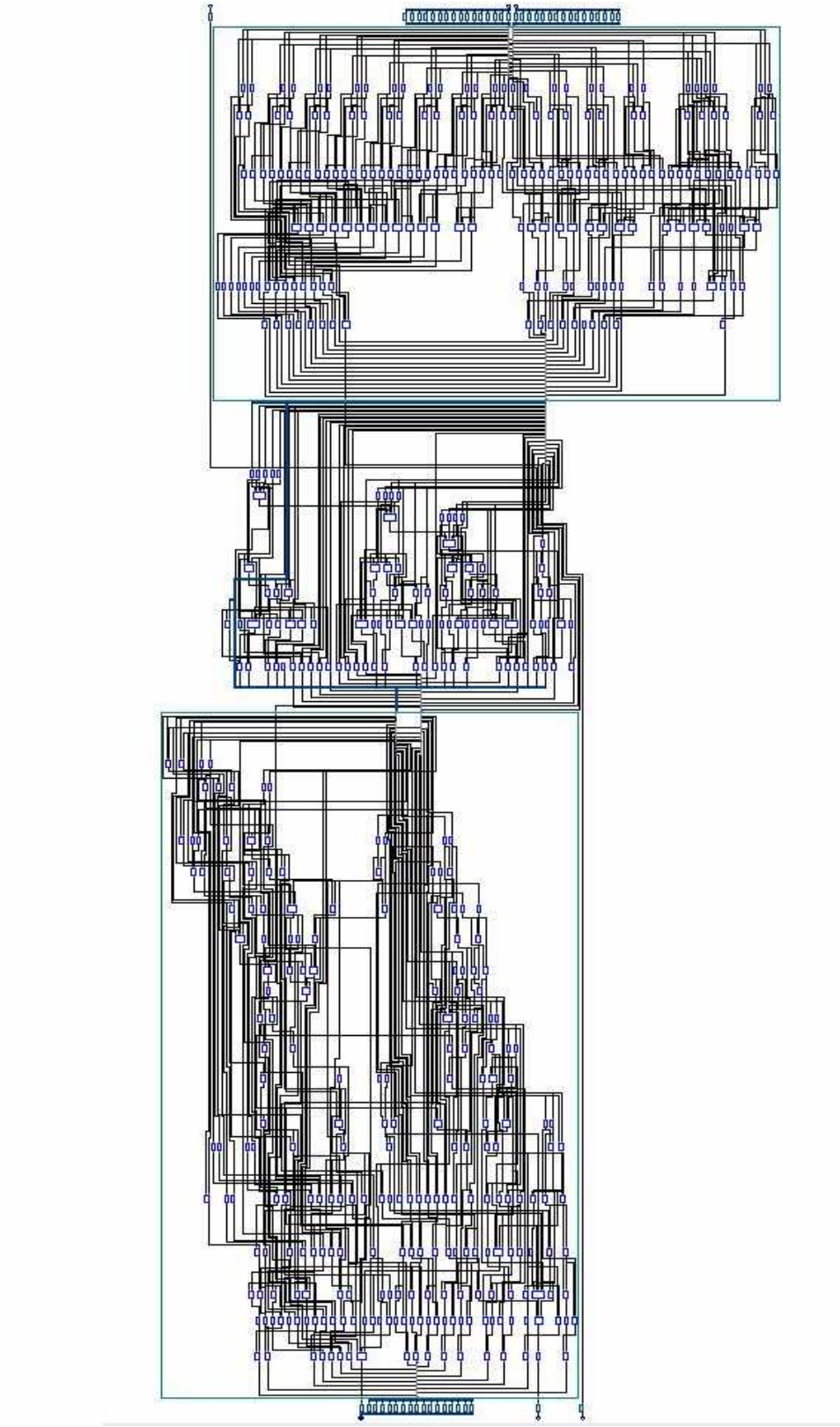


図 4.10　冗長 2 進数から通常 2 進数への変換回路を加えた冗長 2 進加算器による 16 ビット加算器の論理レイアウト（テクノロジマッピング）

５　加算器の性能比較

第３章と第４章にて様々な１６ビット加算器について HDL で記述し、論理合成を行いその性能を調べた。下の表にその結果をまとめた。

加算器	ゲート数	遅延時間
Ripple１	２３６個	６１.０⊿
Ripple２	１７６個	５１.５⊿
Tree1	２８６個	２９.５⊿
Tree2	２２５個	３４.５⊿
Tree3	３０３個	２３.０⊿
CSA	３４４個	２４.５⊿
RBA	３２３個	９.５⊿
RBA2	５７７個	３９.０⊿

表 5.1　加算器の性能比較

Ripple1：4 ビット長桁上げ先見回路の順次桁上げによる 16 ビット加算器

Ripple2:8 ビット長ブロック桁上げ先見回路の順次桁上げによる 16 ビット加算器

Tree1：4 ビットブロック桁上げ先見回路と 4 ビット桁上げ先見加算器の
２段構成による 16 ビット加算器

Tree2：2 ビット長ブロック桁上げ先見回路と８ビット桁上げ先見加算器の
２段構成による 16 ビット加算器

Tree3：2 ビット長ブロック桁上げ先見回路と 4 ビット長桁上げ先見加算器の
３段構成による 16 ビット加算器

CSA：16 ビット桁上げ選択加算器

RBA：16 ビット冗長 2 進加算器

RBA2：16 ビット冗長 2 進加算器を使用し通常 2 進表現で出力する加算器

5.1 通常２進表現の加算における性能

表 5.1 から冗長２進加算器が最も速く計算結果を出力できることがわかる。しかし、通常２進数同士を１回だけ加算して通常２進の和を出力したい場合では、冗長２進加算器の性能を発揮することはできない。これは冗長２進加算器が通常の２進数を入力して加算を行った場合でも冗長２進表現による答えしか出力することはできないためである。このとき冗長２進表現を従来の２進表現に戻してやる必要があり、そのために通常２進のための加算器を使用する必要がある。このとき、桁上げの伝播が発生してしまうので結局最初から通常の２進加算器を使用したほうが良くなってしまう。ゆえに、通常の２進加算を１回だけ行う場合は冗長２進加算器の使用をしないほう良い。

冗長２進加算器を除いてその性能を比較した場合、２ビット長ブロック桁上げ先見回路と 4 ビット長桁上げ先見加算器の３段構成による 16 ビット加算器が最も速く計算結果を出力することができる。

以上より今回調べた加算器では、通常２進数の１回だけの加算には２ビット長ブロック桁上げ先見加算回路と４ビット長桁上げ先見加算器の３段構成による１６ビット加算器が最も適している。

5.2 繰り返し加算における性能

乗算器のような繰り返し加算を行う場合は、中間和は冗長２進表現でも問題が無い。そこで、冗長２進加算器を繰り返し使用して最後に冗長２進から従来の２進数表現に戻してやるだけでよい。そのため、繰り返し加算を行う場合は冗長２進加算器の性能を十分に発揮することができる。乗算器については次章で詳しく述べる。

5.3 ビット長と速さの関係

今回は 16 ビット長加算器について設計し比較をしてきたが、もっと大きなビット長になった場合の加算器の遅延時間について図 5.１のグラフに示す。この図を見てわかるように順次桁上げで接続された加算器は線形に遅延時間が増加していき、段構成に接続された加算器は遅延時間が緩やかに増加していくことがわかる。それに対して、冗長 2 進加算器は桁上げ信号の伝播が高々1 桁しか起こらないために、ビット長によらず常に一定の遅延時間で計算することが可能である。

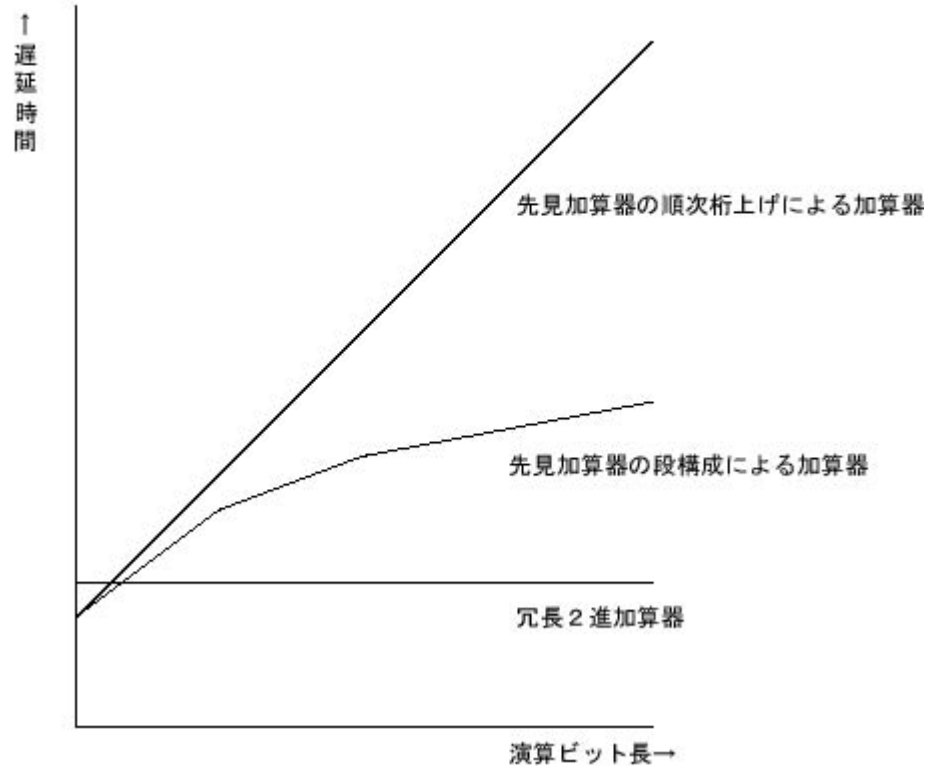


図 5.1　演算ビット長と遅延時間の関係

6　乗算器

6.1　乗算器の基本動作

演算回路の中の重要な回路として乗算器がある。乗算器は加算を繰り返し行い結果を出す回路である。まず図 6.1 に 2 つの数$\mathbf{x}, \mathbf{y}$(ともに 4 ビット)の乗算結果が出るまでの簡単な図を示す。

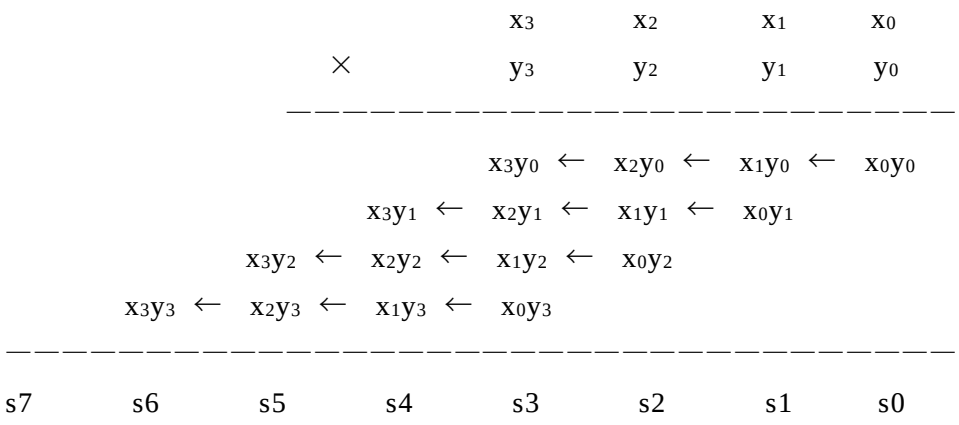


図 6.1　乗算の基本動作(←：桁上げ)

乗算器にはまず被乗数と乗数が入力され、部分積が求められる。部分積は乗数と被乗数との論理積で求めることができる。図 6.1 の 1 つの例を上げると、乗数が y_0、被乗数が x_3〜x_0、この乗数と被乗数の部分積が x_3y_0、x_2y_0 、x_1y_0、x_0y_0 である。これを乗数 y_3〜y_1 まで繰り返し行い、それぞれの部分積を求める。ここで注意しなければならないのは、y_1 は y_0 より 1 ビット重い数であるため、部分積も乗数が y_0 の場合よりも 1 ビット左へシフトする。同様に y_2 の部分積も y_1 のときより左へ 1 ビット、y_3 の部分積も y_2 のときより左へ 1 ビットシフトした部分積となる。これを加算することによって x と y の乗算結果を求めることができる。

6.2 乗算器の高速化

6.2.1 部分積生成過程における高速化

部分積の生成過程における高速化は、部分積の数を減らす方法に帰する。最も基本的な部分積の生成過程は、乗数の 1 ビットに対して AND 操作で部分積を生成する方法であり、乗数のビット数に等しい数の部分積が生ずる。そこでこの部分積を減らす手法である Booth アルゴリズムを説明する。この手法は高速の乗算器 LSI 上に実現する方法として、広く用いられている。

Booth アルゴリズムでは、乗数を 2 桁ごとに区切り、冗長 2 進表現を用いてどちらか一方の桁が必ず 0 になるように変換する。これにより、変換後の乗数の表現を用いた乗算では、乗数 2 桁毎に 1 つの部分積が生成される。そのため、Booth アルゴリズムを適用しない場合に比べ、部分積の数を約半分にすることができる。この変換をリコードと呼ぶ。

2 桁ごとに区切った乗数を y_{2j+1}、y_{2j} とし、これに 1 つ下位の桁 y_{2j-1} を加えた 3 桁を用いて、リコード後の表現、符号を表す $r^{(s)}_j$、1 の重みを持つ $r^{(1)}_j$、および 2 の重みを持つ $r^{(2)}_j$ を得る。また Booth アルゴリズムでは、リコードされた表現をもとに、被乗数シフト操作と正負の反転操作によって部分積を求める。これらの操作を行う回路は、Booth アルゴリズムを適用することによって削減される加算器よりも小さな回路となるため、2 次の Booth アルゴリズムを用いることによって、乗算器の遅延時間と回路の面積を削減することができる。表 6.1 に Booth アルゴリズムのリコード規則を示す。

	y_{2j+1}	y_{2j}	y_{2j-1}	$r^{(s)}_j$	$R^{(1)}_j$	$r^{(2)}_j$
2	0	1	1	0	0	1
1	0	1	0	0	1	0
1	0	0	1	0	1	0
0	0	0	0	0	0	0
-0	1	1	1	1	0	0
-1	1	1	0	1	1	0
-1	1	0	1	1	1	0
-2	1	0	0	1	0	1

表 6.1　y_{2j+1}、y_{2j}、y_{2j-1} と $r^{(s)}_j$、$r^{(1)}_j$、$r^{(2)}_j$ 間のリコード

表 6.1 のリコード規則をもとに、A×B の計算の例を示す。

B		部分積
• 000	→	0
• 010	→	＋A
• 100	→	－2A
• 110	→	－A
• 001	→	＋A
• 011	→	＋2A
• 101	→	－A
• 111	→	0

・B を 2bit ＋ 重複 1bit = 3bit 毎に分割して、部分積を形成する。

・区切った 3bit と、対応する部分積は左の通りである。

図 6.2　A×B の計算例

以上のように、上記 2 ビットの Booth アルゴリズムは部分積の数を約半分にでき高速化が図れる。また、2 の補数表示に伴う補正の処理を必要としない優れた特長を有するために、高速乗算の手法として広く用いられる。

6.2.2　冗長 2 進加算器の乗算器における優位性

第 2、3、4 章でいろいろな加算器について述べ、第 5 章でその加算器の性能について比較した。そこでは、冗長 2 進加算器は高速演算という特長を持つが、その後通常 2 進数に変換するため高速演算という冗長 2 進加算器の優位性が発揮できなかった。しかし、冗長 2 進加算器を用いた 2 個の部分積の加算は高々1 桁の伝播遅延のみで演算されるため、図 6.3 に示すように冗長 2 進数体系で部分積を単純に 2 つずつ 2 分木状に加え合わせ、最後に冗長 2 進表現で表された積を容易に得ることが可能であり、その後に冗長 2 進数から通常 2 進数への変換器をつけたとしても、部分積の数が多ければ多いほど冗長 2 進加算器の高速演算が発揮できるようになる。

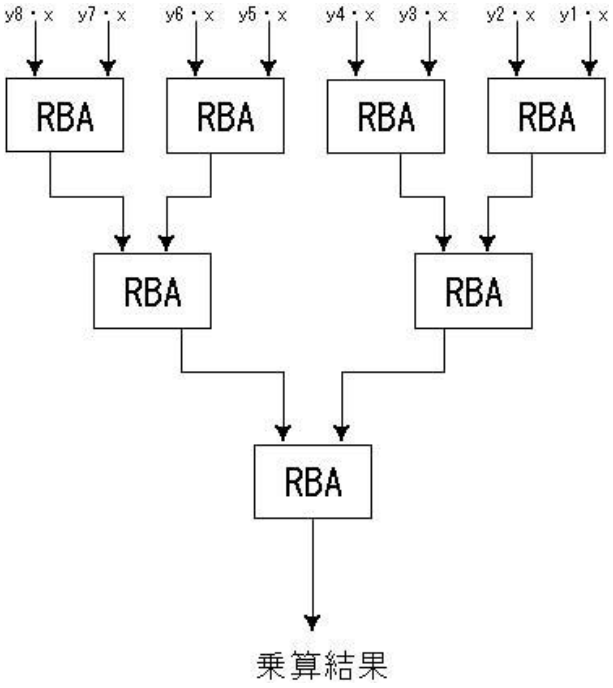


図 6.3　冗長 2 進加算器を用いた加算木
RBA：冗長 2 進加算器

次に使用する加算器の bit 数と遅延時間の関係について述べる。まず表 6.2 に n 個の部分積における加算器の段数および使用する加算器の bit 数を示す。

部分積の数	加算器の段数	使用する加算器の bit 数
4	2	8
$4 < n \leqq 8$	3	16
$8 < n \leqq 16$	4	32
$16 < n \leqq 32$	5	64
$32 < n \leqq 64$	6	128

表 6.2　n 個の部分積における加算器の段数および使用する加算器の bit 数

表 6.2 を見れば分かるように、部分積の数が 16、32 と増えるに従って使用する加算器の bit 数も増加している。使用する加算器の bit 数が 8、16 など少ない場合には冗長 2 進加算器の優位性はあまり発揮できないかもしれない。しかし、使用する加算器の bit 数が 32、64 となると、冗長 2 進加算器のような bit 数が増えても演算時間は変わらないという特長を大いに発揮することができる。このように、通常 2 進加算器を用いた乗算器よりも高速になるのは明らかである。よって冗長 2 進加算器は乗算器のような連続して加算を行い、さらに加算の bit 数が多いような回路でその特長を発揮できる。

6.3 結論

2 つの数の加算において、冗長 2 進加算器は桁上げ信号の伝播が高々１つだけという特長によって他の加算器よりもはるかに高速に動作する。しかし、冗長 2 進加算器は冗長 2 進数表現での出力しかできない。よって通常 2 進表現で出力を行うためには別の加算器を用いるしかないため、その性能を十分に発揮することがなかなかできなかった。そのため、通常 2 進数の加算を 1 回だけ行う際に最も効率的であったものは「２ビット長ブロック桁上げ先見回路と 4 ビット長桁上げ先見加算器の３段構成による 16 ビット加算器」である。

しかし、乗算器における部分積の加算においては、中間和は冗長 2 進数であっても問題は無い。また、加算を繰り返し行い最後に冗長 2 進数から通常の 2 進数に変換するための回路を付加したとしても、十分冗長 2 進加算器の優位性を発揮することが可能である。よって、乗算器のように加算を多数行う場合に最も効率的であったものは「冗長 2 進加算器」であると考えられる。

参考文献

[1] ISSCC88 Session XI:High-Speed Logic THAM 11.8:A 33 MFLOPS Floating Point Processor using Redundant Binary Representation : Hisakazu Edamatsu,Takashi Taniguchi,Tamotsu Nishiyama,Shigeo Kuninobu

[2] 冗長 2 進演算アルゴリズムと高速プロセッサの実現に関する研究：國信茂郎

[3] HDL による VLSI 設計　第２版　－VerilogHDL と VHDL による CPU 設計－
：深山正幸、北川章夫、秋田純一、鈴木正國　（共立出版株式会社）

[4] パターソン＆ヘネシー　コンピュータの構成と設計　第２版　上・下
－ハードウェアとソフトウェアのインタフェース－
：David A.Patterson、John L.Hennessy、成田光彰訳　（日系 BP 社）

[5] 論理回路と計算機ハードウェア　：原田豊　（丸善株式会社）

[6] HDL Endeabor Velirog-HDL　：株式会社エッチ・ディー・ラボ