計算機アーキテクチャ　98－12
設　計　自　動　化　65－12
（1993. 1. 21）

データパス情報を用いた上位合成手法

宮崎敏明

NTT LSI 研究所
〒243-01 神奈川県厚木市森の里若宮 3-1
e-mail: tana@nttica.ntt.jp

あらまし　本稿では、動作仕様を満たすハードウエアを、設計者等が予め与えたデータパス情報をもとに合成する手法について述べる。ここでは、ハードウエア設計上最も難しい部分である制御系の合成を中心に扱うために、通常、比較的容易に定義できるデータパス構成は設計者の入力にゆだね、それを尊重しつつ入力動作仕様を満たすハードウエアを合成する。本手法によれば、設計者が必ずしも満足しない合成結果しか得られなかった従来の手法に比べ、設計者の意図を直接反映したハードウエアを合成できる。

和文キーワード　制御回路、スケジューリング、リソースアロケーション、RTL、CFG、DFG

A High-Level Synthesis Approach Using Given Datapath Information

Toshiaki Miyazaki
NTT LSI Laboratories
3-1, Morinosato Wakamiya, Atsugi-Shi Kanagawa Pref., 243-01, JAPAN

Abstract　In this paper, we propose a high-level synthesis method using data path information given by a designer. The main purpose of this method is to generate a control unit, one of the most difficult part in hardware design. In general, designers can specify data paths easily. So, we believe that a method based on specified data path information is the best way to synthesize hardware closer to the designer's intention.

英文 key words　Control Circuit, Scheduling, Resource allocation, RTL, CFG, DFG

1 まえがき

LSIの設計需要および設計規模の増大に伴って、上位設計支援技術がより注目されるようになってきた。従来、アーキテクチャレベルの設計に際しては、机上または専用のソフトウエア・シミュレータ等を作成して評価を行うのが一般的であった。今日、汎用的な評価支援システムを作成し、評価支援範囲を拡大しようとする動きも盛んになってきている [3],[4],[5]。しかし、それらシステムは性能評価を主目的としており、上位合成システムとしての位置付けは大きくない。

一方、上位合成技術は盛んに研究され多くの有益なアルゴリズムが開発されてきた [1]。しかし、残念ながらまだ現実の設計問題に適用するレベルに達しているものは少ない。その一つの原因は、実際の設計制約をうまく反映できないことによるところが大きいと考えられる。たとえば、データパス合成においては、非現実的な演算器を持つ構成を出力するとか、非常に単純であり人手によれば一瞬に設計できる構成しか合成できないものが多い。

現時点で、実用的な合成結果を得る確実な方法は、CATHEDRAL システム [6] が採用しているように予め用意した機能モジュールの使用を前提として合成するか、詳細な制約条件を入力し、システムがそれを反映できるようにすることであると考える。

我々は、上記の考えに基づき、アーキテクチャ設計上最も難しい部分である制御系の合成を中心に扱うために、通常、比較的容易に定義できるデータパス構成は設計者の入力にゆだね、それを尊重しつつ入力動作仕様を満たすハードウエアを合成する手法を検討した。本手法によれば、設計者が必ずしも満足しない合成結果しか得られなかった従来の手法に比べ、より良いハードウエアを合成できる可能性を秘めている。

また、本手法の応用として、マイクロ制御方式の既存ハードウエアに対するマイクロコード生成 [7] やデータパスマクロを使用した制御系設計等が考えられる。

本稿では、まず、ここで扱う合成問題を述べる。次に、合成処理の流れを概説した後、個々の処理アルゴリズムについて言及する。最後に、プロトタイプを用いて行った実験結果を示す。

2 上位合成問題

ここでいう上位合成とは、機能動作仕様を入力とし、レジスタ転送レベル (RTL) または論理レベルのハードウエア構成を生成するものであるとする。近年、実用レベルに達している論理合成との違いは、入力仕様において、(1) クロックスキームが明示的に与えられていない、(2) 各演算と ALU 等の機能モジュールとの対応が明確でない、といった点が挙げられる。そのため、上位合成の分野では、上記の問題を解決するために「スケジューリング」および「リソースアロケーション」に関するアルゴリズムが盛んに研究されてきた [1]。スケジューリングは、入力動作仕様内の各処理をどの時刻で行うべきかを決定する問題であり、リソースアロケーションは、処理に必要な演算や変数を有限個の演算器やレジスタ等に割り付ける問題である。それぞれ、組合せ最適化問題またはクラスタリング問題としてモデル化されることが多い。しかし、合成対象のハードウエアモデルをどのように設定するかによって、自ずとそれらの問題に対する制約は変わってくる。たとえば、データパスをバスを用いて構成するかマルチプレクサを用いて構成するかによって、制約もアルゴリズムも異なったものとなる。

我々が合成対象とするハードウエアは、図1に示す構造をしているものとする。すなわち、データパスと制御系に大きく分けることができ、制御系からは、データパスに対して制御信号が出力され、データパスからは、後のデータパス制御に必要なコンディション信号が来るものとする。ここで、ALU のセレクタ信号やレジスタへのラッチ信号等、データパス内のあるパスを発火するのに必要な制御信号の供給先を「コントロールポイント」と総称する。

データパスの構成は、設計者が入力し、制御系に関わる動作仕様はそのデータパスを意識して与えるものとする。データパス構成が設計者によって与えられるため、設計者の細かな設計意図を合成結果に反映できる。その反面、リソース制約が、従来手法が仮定していたリソースの個数だけでなく、それらを転送路で結んだトポロジーとして与えられるため、合成処理を行う上では、より強い制約を満たすことが要求される。従来、転送路を含むリソースアロケーションを考慮したスケジューリング手法も提案されているが [2]、設計者が直接与えたデータパスのトポロジーを制約としたものはない。

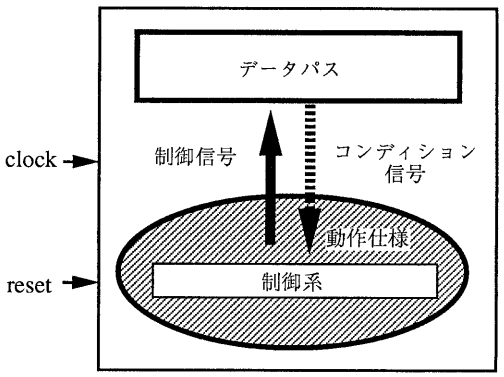

図 1: 合成対象のハードウエアモデル.

仮定

ここでは、上記ハードウエアモデルに従って合成を行う際に仮定した事項を列挙する。

1. 全ての演算は、2項演算または単項演算であり、単一の演算器で実行される。
2. 入力動作仕様内の変数とレジスタの対応は一意に決定されている。
3. スケジューリングによって得られた各ステート内の処理は、すべてそのステート内で終了する。また、各ステートは、同期クロックによって遷移するものとする。

上記の仮定は、主に現状のプロトタイプの制約からくるものであり、今後緩和できるものと考える。

3 合成手法

3.1 処理フロー

図2に本稿で述べる合成手法の概略処理フローを示す。本合成手法の入力は、データパスのトポロジーおよび動作仕様である。動作仕様が入力データパス上で動作することをリソースアロケーション／スケジューリングを行いながら検証し、全ての動作が入力データパス上で実行可能だった場合、結果を状態遷移表を含むRTL記述として出力する。

まず、入力動作仕様は、コンパイルされコントロールフローグラフ(CFG)が作成される。その後、全体スケジューリング処理によって、仮の制御ステート構成を求める。この全体スケジューリングでは、データパス情報を参照せず、CFG内の分岐条件の位置をもとに単純にステート分割する。一方、データパスは、ユーザインタフェースであるブロック図エディタを用いて定義され、機能モジュールを接続するネットリストとして準備される。

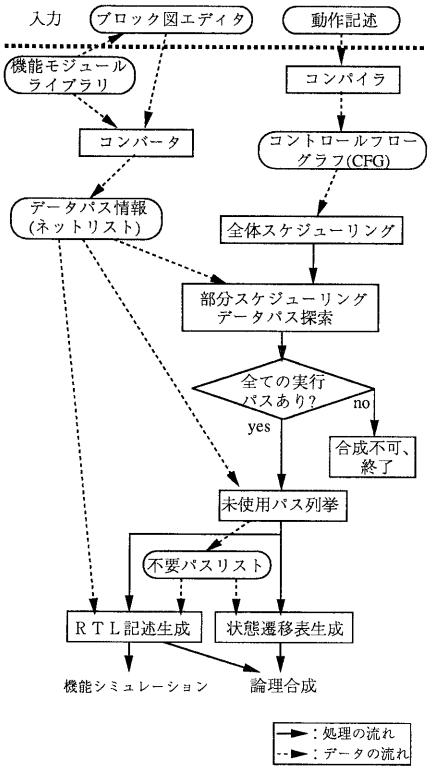

図 2: 処理フロー図.

部分スケジューリング処理では、全体スケジューリングによって生成された個々のステート内で実行されるべき演算が実際に入力データパス上で同時刻に実行できるかチェックする。このチェックは、パス探索処理を用いて行う。チェックした結果、演算器の不足、転送路の衝突等のリソース競合によって同時実行できなかった場合は、そのリソース競合を避けるようにいくつかのステートに分割する。以上の処理を、全てのステート内の演算に対して行う。もしこの部分スケジューリング処理を行っているときに、データパス上にアロケーション可能な演算器や転送路が発見できない演算が存在した場合は、合成処理を中断する。

全ての演算が実行できるステート構成が求まったならば、パス探索処理時に求まったそれぞれのデータパスへのコントロールポイント情報をもとに、状態遷移表を作成し出力する。さらに、動作仕様内の処理を実行するのに一度も使用しなかったパスを不要パスとして抽出し、データパスのネットリストを生成する際、そのパスを削除する。このとき、マルチプレクサの削除等によって変更されたコントロールポイント情報は、前述した状態遷移表の生成においても考慮される。

以下、各部の処理を詳しく述べる。

3.2 全体スケジューリング

図3にCFGの例を示す。これは、図4のVHDL記述を表したものである[1]。図中、分岐条件は、三角形のノードで表し、各々の分岐に従って実行しなければならない一連の演算を「矩形ノード」の中に記述してある。分岐条件を表す三角形のノードを「forkノード」、処理フローの併合を示す逆三角系のノードを「joinノード」と呼び、さらに矩形ノード内に記述された演算処理全体を「処理ブロック」と呼ぶことにする。

全体スケジューリングは、CFGを単純にfork/joinノードを境にステート分割する。図3の例では、破線で示したようにS0からS3の4ステートに分割され、各処理ブロック1から4は、それぞれステートS0からS3で処理される。

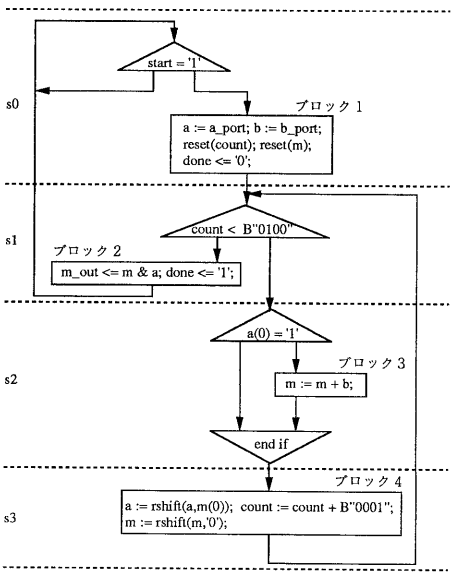

図 3: コントロールフローグラフ(CFG)の例.

```
 1 entity MULT is
 2     port (  a_port, b_port : in BIT_VECTOR(3 downto 0);
 3         m_out : out BIT_VECTOR(7 downto 0);
 4         clk : in BIT;
 5         start : in BIT;
 6         done : out BIT );
 7 end MULT;
 8
 9 architecture SHIFT_MULT of MULT is
10 begin
11     process
12         variable a, b : BIT_VECTOR(3 downto 0);
13         variable count, m : BIT_VECTOR(3 downto 0);
14     begin
15         wait until(start = '1');
16         a := a_port; b := b_port;
17         reset(count); reset(m);
18         done <= '0';
19         while(count < B"0100" ) loop
20             if(a(0) = '1') then
21                 m := m + b;
22             end if;
23             a := rshift(a, m(0));
24             m := rshift(m, '0');
25             count := count + B"0001";
26         end loop;
27         m_out <= m & a;
28         done <= '1';
29     end process;
30 end SHIFT_MULT;
```

図 4: 図 3 のコントローフローグラフに対応する入力動作仕様 (文献 [1] 参照).

3.3 データパス定義

図 5は、図 3の CFG を実行するために定義されたデータパスの一例である。ここで、加算器やレジスタ等の個々の機能モジュールの動作は、機能モジュールライブラリとして定義されており、ユーザはそれを用いてデータパスを定義するものとする。図 6は、実際にパス探索処理に用いられるデータパス情報の例である。これは、ブロック図エディタで定義した図 5のデータを、専用コンバータに通して機械的に得られる。

データパス情報は、外部端子を含む機能モジュールの定義 (type 文) とその機能 (function 文)、および、それらを結ぶネット (path 文) からなっている。

3.4 部分スケジューリング

各処理ブロック内の演算は、分岐を含まないデータフローグラフ (DFG) で表すことができる [1]。もし、各演算の入出力変数間で依存関係があったり、リソース競合があった場合、ブロック内の演算を同時に実行することはできず、別の時刻 (ステート) で実行しなければならない。これは、DFG のスケジューリング問題であるが、リソース制約が「転送路も規定されたデータパス」であるため、単に演算器等の個数をリソース制約とした従来のスケジューリング法を単純に適用することはできない。ここでは、以下に示す「データパス制約スケジューリング法」(Datapath-Constrained Scheduling: DCS) によって処理ブロック内の演算をスケジュールする。DCS 法では、As Soon As Possible (ASAP) スケジューリング法 [1]、および、次節で説明するパス探索法を用いている。

データパス制約スケジューリング法： DCS

1. 全ての演算を要素とする集合を OP_{all} とする。また、現実行ステートを表す変数を $state$ とし、$state := 1$ とする。
2. OP_{all} に対して ASAP スケジューリングを行い、最初に実行でき得る演算を要素とする集合 OP を作成する。
3. 演算 $\forall x \mid x \in OP$ に対して、パス探索を行い、求まったパスを要素とする集合 P_x を作成する。もし、$P_x = \phi$ であった場合は、スケジューリング (合成処理) 不可能として、終了する。
4. $\forall P_x \mid x \in OP$ 内の各パスを調べ、最も多くの演算がリソース競合することなく実行できるパスの組合せを 1 つ採用し、これを、現 $state$ で実行する演算の組として、集合 OP_s^{state} に登録する。このとき、複数解が求まった場合は、最初に見つかった組合せを採用することにする。
5. $OP_{all} := OP_{all} - OP_s^{state}$。もし、$OP_{all} = \phi$ であれば、終了。そうでなければ、$state := state + 1$ とし、2 へもどる。

この手法によるスケジューリング結果は、ASAP スケジューリングの性質上、早い時刻に多くの演算が集中する傾向がある。それにもかかわらず本アルゴリズムを採用した理由は、一般的なスケジューリングとは異なり、入力データパス上のコントロールポイントの総ビット数はどのようにスケジューリングされても一定であり、かつ、スケジューリングの結果得られるステート数を小さく抑える方が、後に論理合成システム等の最適化処理に頼ってステート数を小さくするよりも得策であると考えたからである。また、データパス上で複数の演算器がラッチをはさまずに立て積みされたいわゆるパイプライン演算器構成には、上記アルゴリズムでは対応できない。これは、今後の課題である。

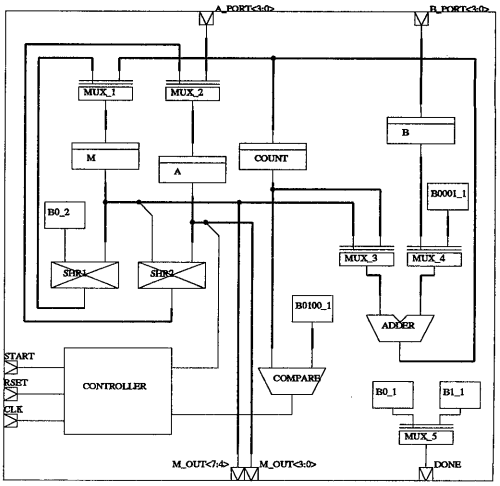

図 5: 図 3 の動作を実行するためのデータパスの例.

```
/****** Type declaration **************/
type(input_pin, 'i_a_port<3:0>').
type(output_pin, 'o_m_out<3:0>').
type(mux2, mux_5).
type(comp1, compare).
type(add1, adder).
type(rshift, shr1).
type(reg1, a).
                :
/****** Path declaration **************/
path(p1, [a,'o1<3:0>'], [o_m_out,'m_out<3:0>']).
path(p2, [a,'o1<3:0>'], [shr2,'i2<3:0>']).
path(p3, [adder,'o1<3:0>'], [count,'i1<3:0>']).
path(p4, [adder,'o1<3:0>'], [mux_1,'i2<3:0>']).
path(p5, [b,'o1<3:0>'], [mux_4,'i1<3:0>']).
                :
/****** Function declaration **********/
function(mux_5,'s(0)',[[mux_5,'o1'],=,[mux_5,'i1']]) .
function(mux_5,'s(1)',[[mux_5,'o1'],=,[mux_5,'i2']]) .
function(adder,nil,[[adder,'o1<3:0>'],=,
        [[adder,'i1<3:0>'],+,[adder,'i2<3:0>']]]) .
function(shr1,nil,[[shr1,'o1<3:0>'],=,
        [func,rshift,[shr1,'i2<3:0>'],[shr1,'i1']]]) .
function(m,'g(1)',[[m],<-,[m,'i1<3:0>']]) .
                :
```

図 6: 図 5 から専用コンバータを用いて得られたデータパス情報 (一部).

3.5 パス探索

パス探索は、処理ブロック内のどの演算同士が同時に実行できるかどうかをチェックするために、個々の演算を実行するのに必要なデータパス上の具体的な経路を求めることである。ここでは、前述したように各演算は、1つの演算器で実行されると仮定する。演算器は、図7に示す2項演算器を一般形とし、演算器のアロケーションを行った後、入出力のパス1、2、3を探索する [8]。

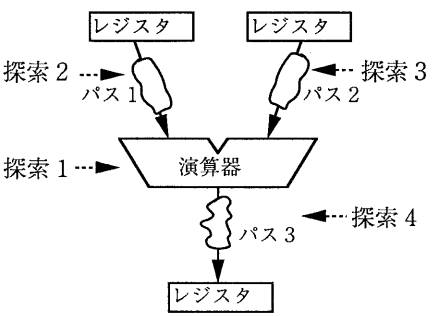

図 7: 演算器の一般形.

パス探索は、現状、全解探索を行っているが、入力データパスが機能モジュール間の接続情報として定義されており、探索空間自体大きくないため、現実的な処理時間で探索は終了する。

3.6 状態遷移表生成

最終的な制御系の合成結果は、状態遷移表として出力する。ここで、制御系への入力は、CFG 内の分岐条件 (fork ノード) が判断できるコンディション信号、およびクロック、リセット信号である。また、制御系からの出力は、デコーダを介さずに、コントロールポイントを直接制御するものとする。コントロールポイントに与える具体的な値は、部分スケジューリング処理中のパス探索を行っている際に、データパス情報内の function 文から抽出されたものを使用している。"don't care" 信号を用いたステートの最適化等は、論理合成の段階で行うものとし、ここでは行わない。ただし、不要パス削除によって変更されたコントロールポイントは考慮している。

3.7 RTL 記述生成

ここでは、合成結果をもとに RTL 言語記述の生成を行う。データパスは、機能ブロック間を結ぶネットリストとして出力される。この際、前述した不要パスリストを用いて不要な転送路は除去する。ただし、機能が明らかでないモジュールに接続されている不要パスは、そのパスを削除した場合に残るモジュールの「あきピン」に対する処理や、別モジュールへの置き換えの可否が判断できないため、除去しない。また、機能モジュールの動作記述は、別途用意しておく。

一方、制御系の RTL 記述は、状態遷移表だけでなく、実現手段すなわちマイクロ制御か布線論理かによって異なるため、機能シミュレーション用に動作記述も生成できるようにした。

4 実験

図 2の処理フローに基づくプロトタイプを、C 言語および Prolog を用いて作成した。ただし、入力動作仕様のコンパイラは、作成中であり、コントロールフローグラフはハンドコンパイルによって作成している。

図 3の入力仕様に対して、図 8に示すデータパスを入力した場合を実験した。図 8のデータパスは、図 5のそれに比べ、シフタが 1 つしかなく、図 3において、ブロック 4 内の 2 つのシフト動作を同時に行うことはできない。これより、ブロック 4 は 2 分割され、新たにステート S4 が生成された。また、図 8内のパス p1 は、図 3の動作を行う上で一回も使用されなかったため不要パスリストに登録され、RTL 記述生成時に削除された。また、これに伴って 3 入力マルチプレクサ MUX_3 は、2 入力のそれに置き換えられた。現在、モジュールの置き換えは、マルチプレクサ以外には行っていない。これは、3.7 で述べたようにそのモジュールの機能を充分把握していなければ一般に別モジュールへの置換は困難であることによる。

図 9および図 10に、最終的に得られた合成結果を示す。RTL 言語は、UDL/I を用いている。また、状態遷移表は、市販論理合成システムの入力形式で出力している。

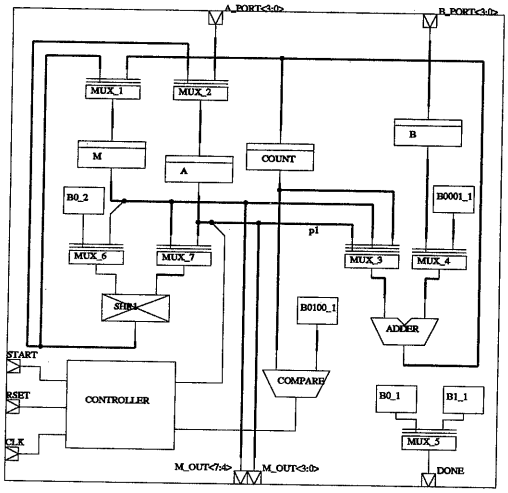

図 8: 実験に用いた入力データパス構成.

5 むすび

既知のデータパス上で、入力仕様を動作させることを前提とした合成手法を提案した。入力データパスを強い制約として合成するため、データパスを含む全てをトップダウンに合成する従来手法に比べ、設計者の意図が直接反映させ易いという特徴を有している。今後は、より多くのデータに対して実験を行い、個々のアルゴリズムの改善を図って行く。

謝辞

日頃、有益な助言を頂く LSI 研究所 設計システム研究部 星野民夫氏および安達徹氏に感謝します。また、本研究を進める上で積極的に討論していただいた同研究部 小林哲雄氏ならびに池田充郎氏に感謝します。

```
IDENT      : SMUL2 ;
DATE       : 12/02/92 ;
AUTHOR     : T.MIYAZAKI ;
VERSION    : V1.0 ;
  NAME     : SMUL2 ;
  PURPOSE  : LOGSIM ;
  LEVEL    : CHIP ;
  VERSION  : V1.0 ;
  INPUTS   : A_PORT<3:0>, B_PORT<3:0>, START, CLK, RSET ;
  OUTPUTS  : M_OUT<7:0>, DONE ;
  TYPES    : MUX2(I2#INPUTS, I1#INPUTS, O1#OUTPUTS, S#INPUTS)
           , COMP1(I1<3:0>#INPUTS, O1#OUTPUTS, I2<3:0>#INPUTS)
           , ADD1(I1<3:0>#INPUTS, O1<3:0>#OUTPUTS, I2<3:0>#INPUTS)
           , REG1(O1<3:0>#OUTPUTS, I1<3:0>#INPUTS, G#INPUTS)
           , CU(RSET#INPUTS, CLK#INPUTS, START#INPUTS, CMP#INPUTS
             , A0#INPUTS, MUX_1#OUTPUTS, MUX_2#OUTPUTS, MUX_3#OUTPUTS
             , MUX_4#OUTPUTS, MUX_5#OUTPUTS, MUX_6#OUTPUTS
             , MUX_7#OUTPUTS, A#OUTPUTS, B#OUTPUTS, COUNT#OUTPUTS
             , COUNT_R#OUTPUTS, M#OUTPUTS, M_R#OUTPUTS)
           , B0(O1#OUTPUTS)
           , B0001(O1<3:0>#OUTPUTS)
           , B0100(O1<3:0>#OUTPUTS)
           , B1(O1#OUTPUTS)
           , MUX24(I2<3:0>#INPUTS, I1<3:0>#INPUTS, O1<3:0>#OUTPUTS
               , S#INPUTS)
           , RSHIFT(I1#INPUTS, I2<3:0>#INPUTS, O1<3:0>#OUTPUTS)
           , REG1R(O1<3:0>#OUTPUTS, I1<3:0>#INPUTS, G#INPUTS, R#INPUTS
                ) ;
  MUX2 : MUX_5, MUX_6 ;
  COMP1 : COMPARE ;
  ADD1 : ADDER ;
  REG1 : A, B ;
  CU : CONTROLLER ;
  B0 : B0_2, B0_1 ;
  B0001 : B0001_1 ;
  B0100 : B0100_1 ;
  B1 : B1_1 ;
  MUX24 : MUX_1, MUX_2, MUX_3, MUX_4, MUX_7 ;
  RSHIFT : SHR1 ;
  REG1R : M, COUNT ;
  END_TYPES ;
  NET_SECTION ;
  W7<3:0> = FROM ( MUX_2.O1<3:0> ) TO ( A.I1<3:0> ) ;
  W20<3:0> = FROM ( COUNT.O1<3:0> ) TO ( COMPARE.I1<3:0>, MUX_3.I2<3:0>
           ) ;
  W16<3:0> = FROM ( B.O1<3:0> ) TO ( MUX_4.I1<3:0> ) ;
  W5<3:0> = FROM ( ADDER.O1<3:0> ) TO ( COUNT.I1<3:0>, MUX_1.I2<3:0> )
          ;
  W26<0> = FROM ( A.O1<0> ) TO ( .M_OUT<0>, MUX_7.I2<0>, CONTROLLER.A0
         ) ;
  W26<3:1> = FROM ( A.O1<3:1> ) TO ( .M_OUT<3:1>, MUX_7.I2<3:1> ) ;
  W22 = FROM ( B0_1.O1 ) TO ( MUX_5.I1 ) ;
  W23 = FROM ( B1_1.O1 ) TO ( MUX_5.I2 ) ;
  W18<3:0> = FROM ( MUX_3.O1<3:0> ) TO ( ADDER.I1<3:0> ) ;
  W19<3:0> = FROM ( MUX_4.O1<3:0> ) TO ( ADDER.I2<3:0> ) ;
  W31 = FROM ( MUX_5.O1 ) TO ( .DONE ) ;
  W17<3:0> = FROM ( B0001_1.O1<3:0> ) TO ( MUX_4.I2<3:0> ) ;
  W21<3:0> = FROM ( B0100_1.O1<3:0> ) TO ( COMPARE.I2<3:0> ) ;
  W0<3:0> = FROM ( .A_PORT<3:0> ) TO ( MUX_2.I2<3:0> ) ;
  W1<3:0> = FROM ( .B_PORT<3:0> ) TO ( B.I1<3:0> ) ;
  W8 = FROM ( B0_2.O1 ) TO ( MUX_6.I1 ) ;
  W28 = FROM ( MUX_6.O1 ) TO ( SHR1.I1 ) ;
  W29<3:0> = FROM ( MUX_7.O1<3:0> ) TO ( SHR1.I2<3:0> ) ;
  W2<3:0> = FROM ( SHR1.O1<3:0> ) TO ( MUX_2.I1<3:0>, MUX_1.I1<3:0> ) ;
  W9<0> = FROM ( M.O1<0> ) TO ( MUX_6.I2, MUX_7.I1<0>, MUX_3.I1<0>
        , .M_OUT<4> ) ;
  W9<3:1> = FROM ( M.O1<3:1> ) TO ( MUX_7.I1<3:1>, MUX_3.I1<3:1>
          , .M_OUT<7:5> ) ;
  W25 = FROM ( COMPARE.O1 ) TO ( CONTROLLER.CMP ) ;
  S16 = FROM ( .CLK ) TO ( CONTROLLER.CLK ) ;
  S14 = FROM ( .START ) TO ( CONTROLLER.START ) ;
  W6<3:0> = FROM ( MUX_1.O1<3:0> ) TO ( M.I1<3:0> ) ;
  S1 = FROM ( CONTROLLER.MUX_1 ) TO ( MUX_1.S ) ;
  S2 = FROM ( CONTROLLER.MUX_2 ) TO ( MUX_2.S ) ;
  S3 = FROM ( CONTROLLER.MUX_3 ) TO ( MUX_3.S ) ;
  S4 = FROM ( CONTROLLER.MUX_4 ) TO ( MUX_4.S ) ;
  S5 = FROM ( CONTROLLER.MUX_5 ) TO ( MUX_5.S ) ;
  S6 = FROM ( CONTROLLER.MUX_6 ) TO ( MUX_6.S ) ;
  S7 = FROM ( CONTROLLER.MUX_7 ) TO ( MUX_7.S ) ;
  S10 = FROM ( CONTROLLER.COUNT ) TO ( COUNT.G ) ;
  S8 = FROM ( CONTROLLER.A ) TO ( A.G ) ;
  S9 = FROM ( CONTROLLER.B ) TO ( B.G ) ;
  S11 = FROM ( CONTROLLER.COUNT_R ) TO ( COUNT.R ) ;
  S13 = FROM ( CONTROLLER.M_R ) TO ( M.R ) ;
  S12 = FROM ( CONTROLLER.M ) TO ( M.G ) ;
  S15 = FROM ( .RSET ) TO ( CONTROLLER.RSET ) ;
  END_SECTION ;
  END ;
CEND ;
```

図 9: 合成結果 1. データパスを中心としたネット記述.

```
.design SMUL2_st
.inputnames clk reset start cmp a0
.outputnames mul1 mul2 mul3 mul4 mul5 mul6 mul7
.outputnames a b count count_r m m_r
.clock clk rising_edge
.asynchronous_reset reset rising s0
#State Table
#                                          c
#                                          o
#       s                                 cu
#       t                          mmmmmmm  on
#       ac                         uuuuuuu  ut m
#       rma                        lllllll  n_ _
#       tp0                        1234567abtrmr
#
        0--     s0      s0         -------------
        1--     s0      s1         -1--0--11-1-1
        -0-     s1      s0         ----1--000000
        -1-     s1      s2         -------000000
        --0     s2      s3         -------000000
        --1     s2      s3         1-00---000010
        ---     s3      s4         -011-11101000
        ---     s4      s1         0----00000010
```

図 10: 合成結果 2. 制御系の状態遷移表.

参考文献

[1] D. Gajiski et al., High-Level Synthesis — Introduction to Chip and System Design, Kluwer Academic Publishers, 1992.

[2] C. H. Gebotys and M. I. Elmasry, Optimal VLSI Architectual Synthesis — Area, Performance and Testability, Kluwer Academic Publishers, 1992.

[3] D. Pease et al., " PAWS: A Performance Evaluation Tool for Parallel Compuring Systems," IEEE Computer, Vol.24, No.1, pp.18-29, January 1991.

[4] 岩下、中田、広瀬、「プロセッサのパイプライン制御に関する動作レベル設計／検証支援」、DA シンポジウム '92, pp.85-88, 平成 4 年 8 月.

[5] 赤星他、「スーパースカラプロセッサ・アーキテクチャ評価用ワークベンチ — 基本構想の検討 —」、DA シンポジウム '92, pp.77-80, 平成 4 年 8 月.

[6] H. De Man et al., " CATHEDORAL-II: A Silicon Compiler for Digital Processing," IEEE Design and Test, December 1986.

[7] 平岡他、「汎用マイクロプログラムトランスレータ MARTRAN の言語仕様」第 25 回情処全大 5N-4, pp.185-186, (昭 57).

[8] S. Takagi, " Rule Based Synthesis, Verification and Compensation of Data Paths," Proc. ICCD'84, pp.133-138, October 1984.