4 System Level Design

動作合成技術の動向

(株)リコー
塚本 泰隆
yasutaka.tsukamoto@nts.ricoh.co.jp

メンター・グラフィックス・ジャパン(株)
温　兆祺
siu-ki_wan@mentor.com

動作合成ツールへの期待

近年，ディジタル回路の規模が増大し回路設計期間の短縮が望まれている．このような状況の中，C言語ベースの動作合成技術が注目を浴びている[1)]．動作合成とはC言語等によりアルゴリズムを記述し，そのアルゴリズム動作を実現するディジタル回路を自動的に生成する技術である．本稿ではこの動作合成技術の動向について説明する．なお動作合成技術は，本特集2章「システムレベル設計フローと設計言語」の図-3「システムレベル設計フロー」における「実装設計」内の「HW設計」部分に適用される．また同図「アーキテクチャ決定」での「設計空間探索」部分にも適用できる．設計空間探索における動作合成技術の適用については，本特集7章「低消費電力化設計と消費電力見積り」でも紹介されている．

回路設計手法の変遷

動作合成技術について説明する前にディジタル回路設計手法の変遷について振り返ってみることにする．

```
unsigned int func( a, b, c, d )
unsigned int a,b,c,d;
{
   if( (a+b) < (c+d) ){
      e = 1;
   }
   else{
      e = 0;
   }
   return( e );
}
```

図-1　Cプログラムの例

■回路図入力（1980年代）

1980年代では，ディジタル回路設計は人手によって回路図を作成していた．これは本特集2章「システムレベル設計フローと設計言語」の図-1「HW設計の変遷」における「ゲートレベル設計」の部分に相当する．ただし回路図は紙に直接書くのではなく，回路図エディタと呼ばれるCADツールにより作成していた．たとえば所望の動作が図-1のようなCプログラムで存在している場合，人手により図-2のような回路図を作成していた．なお誌面の都合上，図-2では各変数の2ビット分だけの回路図を示した．

図-2に示した回路は，(a+b)<(c+d) ならば1を出力し，それ以外なら0を出力する回路である．図-2中のFFはフリップフロップを示している．フリップフロッ

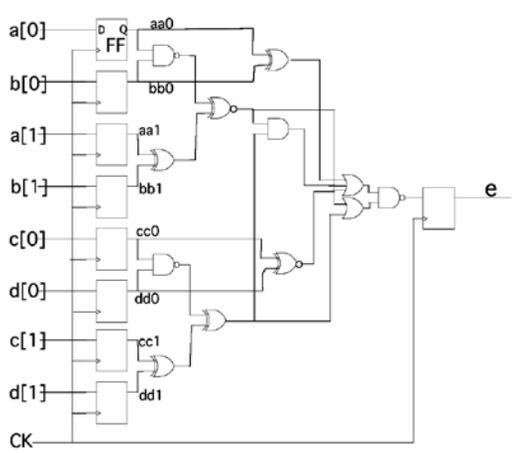

図-2　図-1に対応した回路図の例

プとはクロック信号（CK）の立ち上がりエッジにより，入力ピンDの論理値を出力ピンQに伝播する回路である．クロックエッジにより，a,b,c,dの各ビットの値がフリップフロップに取り込まれるとともに，加算や比較等の演算が実施され，次のクロックエッジによって，演算結果がeに出力される．

しかし1990年代になると回路の規模が数10万ゲート規模に達し，回路図を作成するのが困難な状況となってきた．また作成した回路図の論理動作が正しいかどうかを確認するために実施する論理シミュレーションに要する時間も許容範囲を超えてきた．ここで回路の規模をゲートという用語で表現したが，2入力のNAND素子を1ゲートと数える場合が多い．10万ゲートの回路というと，2入力のNAND素子10万個に相当する規模（面積）の回路であることを意味する．

■HDL（Hardware Description Language）記述（1990年代）

回路図の作成が困難になると，1990年代ではC言語のようなプログラミング言語によく似たハードウェア記述専用の言語を使って回路を記述するようになってきた．図-2の回路図をハードウェア記述言語により記述した例を図-3に示す．各行の先頭に行番号が記述されているが，これは説明のために記述しているだけで，実際に記述する必要はない．また，図-2では2ビット分の記述であったが，ハードウェア記述言語では図-3の[31:0]のように任意のビット長を指定でき，多ビットの回路を簡単に分かりやすく記述できる．

これはVerilog-HDLと呼ばれるハードウェア記述言語による記述例である．なおハードウェア記述言語のことを単にHDL（Hardware Description Language）と呼ぶことが多い．

図-3の8行目のalways@(posedge clk)は，信号clkが立ち上がれば以降13行目のendまでを実行するという意味である．これはまさに先述したフリップフロップ動作を意味する．また16行目から19行目までは，図-2の回路図ではフリップフロップ間の組合せ回路に相当している．

図-3のようなHDL記述をすればそれで回路設計が完了したわけではない．何らかの方法で最終的には図-2のように素子にマッピングする必要がある．そこで1980年代後半に登場したのが論理合成ツールとよばれるCADツールである．やや乱暴な言い方ではあるが，論理合成ツールは図-3のようなHDL記述を読み込み，図-2のような回路を自動的に生成するソフトウェアである．

HDLとともに頻繁に登場する用語としてRTL（Register Transfer Level：レジスタ転送レベル）がある．

```
1 module func( clk, a, b, c, d, e );
2 input clk;
3 input [31:0] a,b,c,d;
4 output [31:0] e;
5
6 reg [31:0] aa,bb,cc,dd,e;
7
8   always @(posedge clk )begin
9     aa <= a;
10    bb <= b;
11    cc <= c;
12    dd <=d;
13  end
14
15  always @(posedge clk )begin
16    if( (a+b) < (c+d))
17      e <= 1;
18    else
19      e <= 0;
20  end
21
22 endmodule
```

図-3　ハードウェア記述言語による記述例

これはレジスタ（フリップフロップ）および，レジスタ間の組合せ論理により回路動作を記述するという記述抽象度のことである．図-3はRTLでの記述である．また図-2の記述抽象度もRTLと呼べなくもないが，組合せ論理の部分が具体的な論理ゲート（論理素子）により表現されているため，RTLとは呼ばずにゲートレベルと呼ぶ．RTLでの設計は本特集2章「システムレベル設計フローと設計言語」の図-1「HW設計の変遷」では「レジスタトランスファレベル設計」部分に相当する．

このように1990年代では，HDLを使ったRTL記述および論理合成ツールにより数10万ゲート規模の回路を短期間に設計することが可能になった．しかし，さらに回路の規模が大きくなる中，設計期間をもっと短縮できないかという要求が出てきた．

動作合成ツールの普及開始

■動作合成ツールとは

図-3の15行目のalways@(posedge clk)は，信号clkが立ち上がれば以降20行目のendまでをclkの次の立ち上がりまでに実行するという意味である．この結果，15行目から20行目までに含まれている比較，加算といったすべての演算は1クロックサイクル内で実施されることになる．しかし論理合成後のゲートレベル回路において1クロックサイクル内で処理が完了できない場合は，2サイクルでこれらの演算を実施するようRTL記述を変更しなければならない．図-3のRTL記述に対して書き換えた例を図-4に示す．

このように処理サイクルを変更するためにその都度

```
module func( clk, a, b, c, d, e );
input clk;
input [31:0] a,b,c,d;
output [31:0] e;

reg [31:0] aa,bb,cc,dd,e;

   always @(posedge clk )begin
      aa <= a;
      bb <= b;
      cc <= c;
      dd <=d;
   end

   always @(posedge clk )begin
      tmp1 <= a + b;
      tmp2 <= c + d;
   end

   always @(posedge clk )begin
      if( tmp1 < tmp2 )
         e <= 1;
      else
         e <= 0;
   end

endmodule
```

図-4　処理サイクル数を変更したRTL記述例

RTL記述を書き換えていては，設計期間を短縮することはできない．また記述したRTL記述の論理動作が正しいかどうかを検証するために実施する論理シミュレーションに要する時間も許容範囲を超えてきた．

そこで`always@(posedge clk)`のようなクロック記述（レジスタ記述あるいはフリップフロップ記述）をせずに回路動作を記述しようという考えが出てくる．**図-5**に，Verilog-HDLを使ったクロック記述なしの動作記述例を示す．これは図-1のCソースコードをVerilog-HDLにより書き換えた例でもある．

図-5にはクロックに関する記述が存在しないので，もはやRTLではない．（`posedge clk`）という行が1行存在するが，これは，論理シミュレーションが無限ループに陥らないようにするための記述であり，各処理ごとのクロックを明示的に記述したものではない．このような記述抽象度は動作レベルまたはビヘイビア・レベルと呼ばれている．そして図-5のような記述から図-3や図-4のようなRTL記述を自動生成するツールが動作合成ツールである．動作合成ツールは高位合成ツールと呼ばれることもある．なお動作合成技術に関する研究は先述した論理合成技術よりも歴史は古いようである．動作レベル記述での設計は本特集2章「システムレベル設計フローと設計言語」の図-1「HW設計の変遷」では「システムレベル設計」の部分に相当する．

動作レベル記述ではクロックを意識せず記述することができ，RTL記述に比べ記述量が1/10～1/100に減る

```
module func(clk, a, b, c, d, e);
input clk;
input [31:0] a,b,c,d;
output [31:0] e;
   always begin
      if( (a+b) < (c+d) )
         e = 1;
      else
         e = 0;
      @(posedge clk);
   end
endmodule
```

図-5　Verilog-HDLによる動作レベル記述の例

```
#include <systemc.h>

SC_MODULE(func){
   sc_in< sc_uint<32> >  a,b,c,d;
   sc_out< sc_uint<32> > e;

   SC_CTOR(func){
      SC_CTHREAD(func_main,clk.pos());
   }
};

void  func::func_main()
{
   sc_in< sc_uint<32> > a,b,c,d;

   while(1){
      if( (a+b) < (c+d) )    e = 1;
      else                    e =0;
      wait(1);
   }
}
```

図-6　図-5をSystemCで書き換えた例

といわれている．この結果，動作レベルで記述すれば回路設計者1人あたりの記述できる回路の規模が大きくなる．今までと同規模の回路を設計する場合は，記述量が減り，短時間で記述できる．

動作レベルの記述は何もVerilog-HDLに限った話ではない．**図-6**は，SystemCと呼ばれているシステムレベル記述言語を使って図-5の記述を書き換えた例である．これは図-1のCプログラム記述をSystemCで書き換えた例でもある．

SystemCはC++のクラスライブラリ群であり，このためC/C++のプログラムと親和性が高いとされている．

■動作合成ツールは何をしているのか

ここでは典型的な動作合成アルゴリズムについて簡単に説明する．詳細については文献2)を参照されたい．

動作合成ツールの内部では大きく分けて2つの作業を行っている．1つはスケジューリングと呼ばれるものであり，もう1つはアロケーションである．スケジューリングとは，クロックの記述がない（つまり1クロック間にどのような演算をするかという記述がされていない）

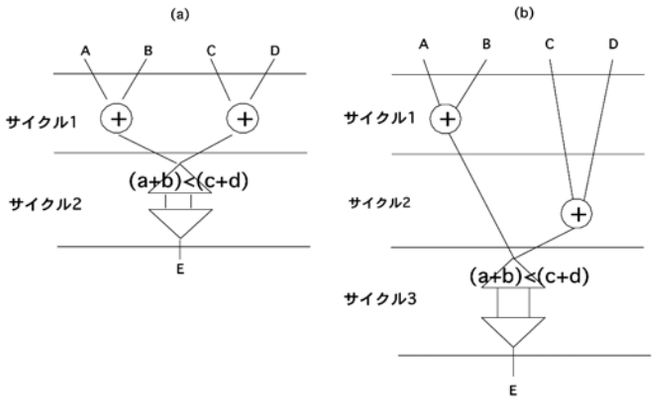

図-7　コントロール・データフロー・グラフ

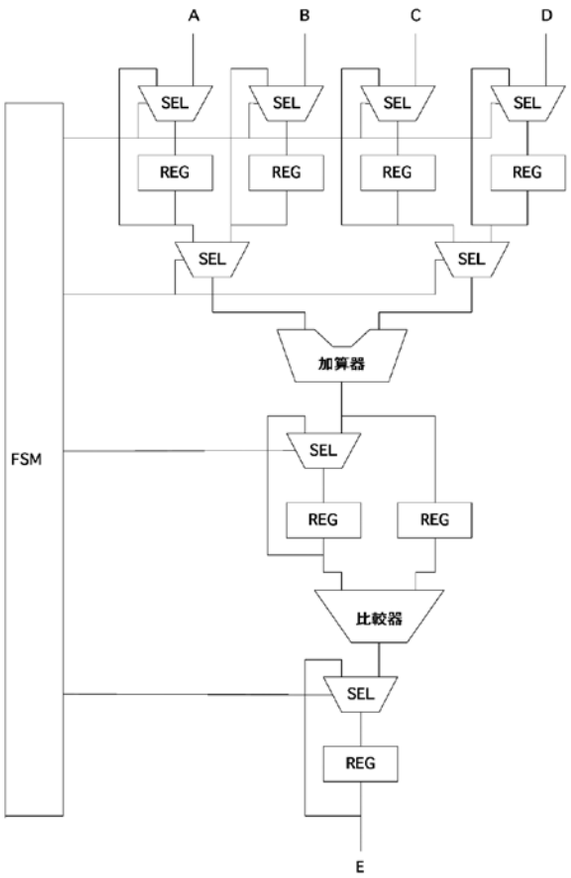

図-8　アロケーションの例

記述から，1クロック間に行う演算内容を決定する作業である．たとえば，図-5の記述を読み込み，図-7のような内部データベースを作成する．

図-7の（a），（b）はコントロール・データフロー・グラフと呼ばれる有向グラフである．なお総処理サイクル数は，人間があらかじめ指定する場合もあれば，動作合成ツールに完全お任せという場合もある．総処理サイクル数は図-7（a）では2サイクル，図-7（b）では3サイクルとなっている．スケジューリング作業により各クロックサイクルで何をするかが決まれば，使用する演算器（比較器，加算器，乗算器など）を次に決定する．この作業はアロケーションと呼ばれる．アロケーションでは，使用する演算器，レジスタ（フリップフロップ），セレクタを配置し，それぞれを接続する．この接続作業のことを特にバインディングと呼ぶ場合がある．図-7（b）のデータフローグラフに対してアロケーションを実施した例を図-8に示す．

図-7（b）中のサイクル1とサイクル2の加算は，図-8の1つの加算器を共用することにより実現されている．この共用のためにセレクタが使用される．そしてこのセレクタのセレクト信号を制御するための制御回路も生成する．この制御回路はステートマシン（Finite State Machine）により実現される．

動作合成の要素技術と動作合成ツールの現状

ここでは，先述したスケジューリングやアロケーション以外の動作合成に関する要素技術および最近の動作合成ツールの対応状況について説明する．

■同時性，並列性の自動抽出

一般にC言語のようなプログラミング言語を使用してプログラムを作成した場合，各演算は上から順に逐次的に実行される（マルチスレッド技術等による並列プログラミングもあり，その技術を使った動作合成技術も存在する．実際JAVAにより記述されたプログラムを入力とする動作合成ツールも存在するが，まだ実用段階ではない）．たとえば図-1のCソースコードでは（a+b）の後に（c+d）が実行される．こ

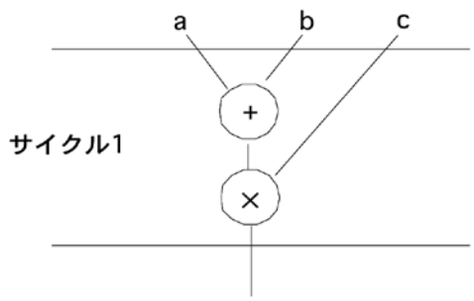

図-9　チェイニング

図-11　forループの並列動作

```
for(i=0;i<32;i++){
   a[i]=b[i]+c[i];
}

for(i=0;i<16;i++){
   d[i]=e[i]+f[i];
}
```

図-10　forループを含むCのソースコード

```
a[0]=b[0]+c[0];
a[1]=b[1]+c[1];
.
.
a[31]=b[31]+c[31];
d[0]=e[0]+f[0];
d[1]=e[1]+f[1];
.
.
d[15]=e[15]+f[15];
```

図-12　forループの展開

の「後」というのは，CPUのクロックが1サイクル以上進んだ後という意味である．一方，図-1の動作をハードウェア（ディジタル回路）により実現する場合は，図-1の（a+b）と（c+d）を同時に実行（演算）するように回路を作成することが可能である．ここでいう「同時」とは，同一クロックサイクル内でという意味である．図-7（a）のサイクル1では2つの加算が同時に行われている．また図-9のような場合では加算と乗算が逐次的に行われるがこの2つの演算は同一クロックサイクル内で行われているので同時処理としてとらえることができる．動作合成ではこれをチェイニングと呼ぶ場合がある．

一方，図-10に示す2つのforループについて考えてみる．これをハードウェアにより実現する場合，変数aとdには依存性がないため2つのforループは図-11に示すように「並列」動作させることが可能である．「並列」と「同時」の違いは，「並列」が複数サイクルにわたる処理を意味するのに対して，「同時」は同一サイクル（1サイクル）での処理を意味している．

最近の動作合成ツールは，ソースコード中の狭い範囲でなら同時性を抽出することができるが，対象範囲が広くなるとその能力にも限界がある．また並列性の抽出に関してはほとんどのツールがまだ十分にはできず，研究段階[3]である．逆に動作合成ツールの限界を知っておけば，それなりに使い道は多々あると考えられる．なお図-10のような記述に対して並列性を抽出できるというツールもある．しかしこれはforループをツールが図-12のように展開し，その後でスケジューリングを実施する場合が多い．これではループ回数が多い場合，スケジューリング対象が大きくなり動作合成では扱えなくなるので注意が必要である．

■パイプライン化

図-13に示したforループをハードウェアで実現する場合は，パイプライン化が可能である．図-14にパイプライン処理の例を示す．

最近の動作合成ツールは，図-13のような比較的単純なループに対しては自動的にパイプライン化が可能である．しかしループ内にif文等の条件分岐が存在する場合や，ループが入れ子になっている（ネストしている）場合，パイプライン化は苦手あるいはできないのが現状である．

■配列の扱い

C言語のようなプログラミング言語では配列変数が頻繁に使用される．この配列変数をハードウェアにより実現する方法には大きく分けて2つの方法がある．1つはRAM（メモリ）として実装する方法であり，もう1つはレジスタ（フリップ・フロップ）として実現す

```
for(i=0;i<32;i++){
  a[i]=b[i]*c[i];
  e[i]=a[i]*e[i];
}
```

図-13　パイプライン化が可能なforループ

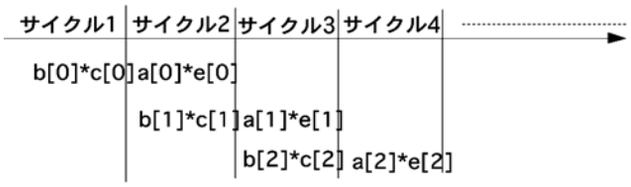

図-14　パイプライン処理

る方法である．それぞれの長所および短所について説明する．

配列をRAMにより実現すると，回路の並列動作性が低減する．これは，RAMに対して同時に複数のアドレスを指定できないからである．つまりa[1]とa[2]に同時にアクセスすることは基本的にはできない．しかしレジスタに比べ小面積で実現できる．

一方，配列をレジスタにより実現する場合は，a[1]とa[2]に同時にアクセスすることが可能であり，回路動作の並列性を上げることができる．しかしRAMに比べ面積が大きくなる．また，配列のサイズが大きい場合は，アドレスデコーダ部分の論理段数が深くなり遅延が大きくなる．

最近の動作合成ツールでは，配列をRAMで実現するのかレジスタで実現するのかを指定することができる．ただし，RAMにしたほうがいいのかレジスタにしたほうがいいのかを自動的に判断してくれるツールは少ない．

■浮動小数点の扱い

C言語等のようなプログラミング言語によりアルゴリズムを記述する場合，浮動小数点型の変数を使用する場合がある．一般に浮動小数点演算をハードウェアで実現すると回路の面積が大きくなり処理サイクル数も増える．このため回路設計者は浮動小数点演算を固定小数点演算に変換して回路を設計する場合が多い．浮動小数点演算を固定小数点演算に自動的に変換してくれる動作合成ツールは現状なく，人手による変換作業が必要である．

■再帰，ポインタ，構造体

C言語のようなプログラミング言語で使用する再帰関数はほとんどの動作合成ツールでは扱えない．またポインタ，構造体に対してはある程度対応している動作合成ツールも存在するが，そのような使い方は最初から避けたほうが無難である．ポインタ記述はあらかじめ配列に書き換えておく方がいい．ポインタを扱う動作合成技術に関しては文献4)が参考になる．

動作合成ツールは使えるか？

以上，動作合成技術の基本事項およびツールの動向について説明してきた．ブロック（関数あるいはプロセス）間での並列化やパイプライン化は人手によりあらかじめ実施しておき，各ブロック内に対して動作合成ツールを適用すれば現状のツールでも十分使い道はある．ただしここでいう「ブロック」をどのような規模で分割すればよいのかを知るためは，動作合成ツールの能力を十分に理解しておく必要がある．またC言語記述に一定のガイドラインを設ければさらに動作合成を適用しやすい環境を構築することが可能となる．ガイドラインの作成およびガイドラインに従うのは面倒だが，現状の動作合成ツールを使用するにはこのあたりが最も重要な点である．

謝辞　本稿の執筆にあたって，JEITA／EDA技術専門委員会／SLD研究会の吉永和弘（スキル），野々垣直治（東芝），杉山陽一（日本シノプシス），小林和彦（ルネサステクノロジ）の各氏にご協力いただいたことを感謝いたします．

参考文献

1）ハード設計の危機をソフト技術者が救う,日経エレクトロニクスNo.827 p.103-133（July 2002）.
2）Gajski, D. D., Dutt, N. D.,Wu, A. C. and Lin, S. Y.:HIGH-LEVEL SYNTHESIS, Kluwer Academic Publishers（1992）.
3）Gupta, S., Dutt, N., Gupta, R. and Nicolau, A. : Coordinated Parallelizing Compiler Optimizations and High-Level Synthesis, Technical Report 02-35, Department of Information and Computer Science, University of California, Irvine, CA（2002）.
4）Séméria, L. and De Micheli, G. : Resolution, Optimization, and Encoding of Pointer Variables for the Behavioral Synthesis from C, Published in IEEE Trans. on Computer Aided Design（Feb. 2001）.

（平成16年3月22日受付）