解　説　論　文

ソフトウェアプログラムから ハードウェア記述を合成する高位合成技術
― プロセッサ以外の汎用プログラム実行機構 ―

Use of High-Level Synthesis to Generate Hardware from Software
– Another Alternative General-Purpose Program-Executing Mechanism to the CPU –

若林一敏 Kazutoshi WAKABAYASHI

アブストラクト　CやC++言語等のプログラム記述から，ASICやFPGAを設計するためのRTL記述を合成する「高位合成」技術の概要を述べる．まず，LSIの設計工程とその自動化の歴史と，LSIの大規模化による設計自動化の必然性について説明する．次に，高位合成技術の原理を工程ごとに解説する．高位合成は設計効率化以外に高性能化，低電力化，高信頼性化，再利用性の向上など様々な効果がある．なぜ，そのような効果が得られるかを，技術面から解説する．また，高位合成のターゲットアーキテクチャであるFSM（Finite State Machine）とデータパスからなるFSMDアーキテクチャ処理効率をCPUと比較して議論する．高位合成は基本原理が確立されプロトタイプシステムが出てから実用化まで長い時間がかかった．非常に広範な最適化技術の実装が必要だからである．これらの中から代表的な幾つかの技術を議論する．また，高位合成技術はアルゴリズム系のデータ処理回路だけでなく，制御系回路にも有効であることを示す．次に，ハードウェア向けのC記述の基本の考え方を紹介し，最後に，近年適用が広まっているFPGA向けの高位合成技術やそれを利用した新しい応用分野を紹介する．

キーワード　高位合成，動作合成，高い抽象度，コンパイラ，SoC，FPGA，動的再構成，設計自動化，C言語設計，汎用計算機構

Abstract　This paper discusses High-Level Synthesis (HLS), which reads C or C++ behavioral descriptions and generates a register-transfer level description for ASIC or FPGA. First, the LSI design process and the history of LSI design automation are introduced, and the reason why design processes were automated with the increasing size of LSI is discussed. Next, the fundamental processes in HLS are explained. The advantages of HLS are higher design efficiency, performance, reliability, and reusability. This paper illustrates how and why HLS provides such advantages. This paper also discusses the performance of FSMD, which is the target architecture of HLS and consists of a finite-state machine and a datapath in contrast to a CPU. It has taken many years to create a practical commercial HLS tool since its prototype. This is because a practical HLS tool requires a variety of optimizing techniques. This paper introduces some of the important techniques. Then, the application of HLS to not only data-dominant circuits but also control-intensive circuits is explained. Then, the difference between the C description for hardware and software is illustrated. Finally, recent HLS techniques for FPGAs and the new application area “FPGAs and C-based HLS” are explained.

Keywords　high-level synthesis, behavioral synthesis, higher level of abstraction, compiler, SoC, FPGA, dynamic reconfigurable, design automation, C-based design, general computing mechanism

1．はじめに

SoC(注1)やFPGA(注2)の設計は，動作アルゴリズムを設計し（動作設計），それを実現するブロック図レベルのハードウェア構成（RTL(注3)記述）を設計し（機能設計），それぞれのハードウェアブロックをAND，NAND等の論理回路部品で設計し（詳細論理設計），その回路の論理回路部品をLSIチップ上の配置位置と配線経路を決定する（レイアウト設計）工程からなっている．かつて数千ゲートの時代は全て人手で行われてきたが，数万ゲート，数十万ゲート，数百万ゲートと回路が大規模化するに伴い，下流のレイアウト設計から順に自動化ツールの実用化が始まった．LSIは下流工程に行くほど扱う部品点数が増大し，人手設計が時間的また能力的に困難とな

若林一敏　　正員　日本電気株式会社　共通ソリューション開発本部
E-mail　wakaba@bl.jp.nec.com
Kazutoshi WAKABAYASHI, Member (Manufacturing and Process Industries Solutions Development Division NEC Corporation, Kawasaki-shi, 211-8666 Japan)
電子情報通信学会 基礎・境界ソサイエティ
Fundamentals Review Vol.6 No.1 pp.37–50 2012年7月

(注1) System-on-a-Chip: 一つの半導体チップ上に一連の機能（システム）を集積する設計手法．マイクロプロセッサに専用機能回路が混在しているものが多い．
(注2) Field Programmable Gate Array: 製造後に構成を設定できる集積回路．現場（Field）でプログラム可能という意味．ゲートアレーは，NAND等のゲート回路がアレー上に並んだ構成を指す．
(注3) Register Transfer Level: 設計レベルを示す言葉で，レジスタが陽に見えるレベル．本稿で後に述べるレジスタ転送表を書くレベル．

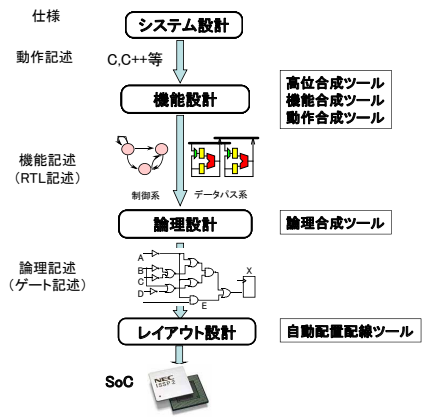


図1　SoCやFPGAの設計工程と対応する設計自動化ツール

り，自動化の必要性が高いためである．1980年代に自動レイアウトツールが，1990年代に論理設計の自動化である論理合成ツールの実用化が広まった．機能設計工程の自動化を実現する技術は，現時点で最上流の自動合成という意味で「高位合成（High Level Synthesis）」と呼ぶ．また，動作合成（Behavioral Synthesis）や機能合成（Functional Synthesis）とも呼ぶ．図1にLSIの設計工程と各工程の自動化ツールを示す．また，動作記述をC言語またはC++，SystemC言語で記述するツールが多いため，「C言語ベース設計」と呼ぶことも多い．現在主流のVerilog HDL言語やVHDL言語を使った設計は，「RTL設計」と呼ぶ．高位合成技術は従来の自動化適用ペースより遅くなっているが，2000年代から実用製品への適用が始まり，2010年代は普及の段階に入っている．筆者は市販高位合成ツールの一つであるCyberWorkBench [1] の研究開発段階から携わっており，その経験を踏まえて高位合成の技術解説を行う．

高位合成は逐次記述である動作記述から，並列記述であるRTL記述を合成する技術である．例えば，記述に100個の演算がある場合，ALU [注4] 一つで実行するのには，最低100サイクル必要である．しかし，ALUを複数用いて実行すれば，複数の演算を並列実行することで，100より少ないサイクルで実行可能である．高位合成の基本原理は，ALUや分岐制御等を並列に実行可能なVLIW（Very Long Instruction Word（超長命令語））プロセッサ向けのコンパイラと似た技術である．したがって，ハードウェア向けのC/C++コンパイラと呼ぶこともできる．しかしながら，VLIWプロセッサ等ではデータパスが固定であるのに対し，高位合成が対象とするSoCやFPGAはデータパスを自由に設計可能であるために，プロセッサに比べ大きな並列性を出す自由度があり，より高性能な回路を合成することが可能である．また，SoCやFPGAでは回路面積を小さくすることがコストに直結するため，面積の最小化が非常に重要になる．更に昨今では携帯機器向けには消費電力の削減も重要である．高位合成では，単に高性能化するだけではなく，性能，面積，電力等の目標・制約に合わせた様々なハードウェア構成（代替アーキテクチャ）を合成することが目的となる．

本稿は，**2.** で高位合成の原理と各工程での動きを小規模な例を用いて説明する．**3.** では高位合成ツールの合成する回路の基本的なアーキテクチャについて説明する．**4.** では高位合成の効果について述べ，**5.** では大規模な回路に対して高位合成を実用化する上で必要なアプローチについて述べる．**6.** でハードウェア向けC記述の特徴について述べ，最後に**7.** でまとめを行う．

(注4) Arithmetic Logic Unit: 算術演算，論理演算を実行する部品．

2. 高位合成の原理

高位合成技術は，Cプログラム等の動作記述から，SoCやFPGAなどで実現されるハードウェア記述（RTL）を合成する．ハードウェアは，同じCプログラムを組込みプロセッサ等で実行するのに比べ，高速で，低消費電力である必要がある．そのため，簡易的なソフトウェア向けコンパイラで利用されるような局所的な並列化（例：Basicブロック [注5] の中の並列化）ではなく，より大局的な並列化 [注6] が求められる．

図2に，動作記述からブロック図レベルの回路を合成する例を示す．左側の動作記述を，演算器制約1（加算器1個，乗算器1個）と演算器制約2（加算器2個，乗算器2個）の下で，高位合成した結果が，右下と右上のブロック図である．右上の図は元の動作記述をそのまま1サイクルで並列に実行する回路構成となっている．元の動作記述に加算，乗算がそれぞれ2個ずつあり，加算器，乗算器も2個ずつ利用している．つまり，動作記述がそのまま回路の構造記述となっている．これは，RTL記述から論理合成が合成する場合と同じである．一方，右下の回路は演算器が1個ずつしかないため，四つの演算を3サイクルかけて実行する回路となっている [注7]．つまり，動作記述を時間方向に折り畳んだ回路構成となっている．

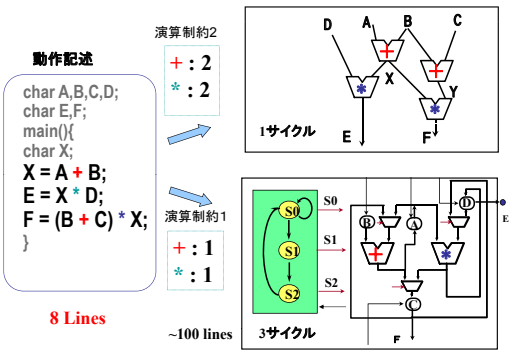


図2　高位合成の入力C記述と二つの演算資源数制約に対応して合成されたRTL

(注5) IF文等の制御構造を含まないコードの集合．

(注6) IF文等の制御構造も含めて並列化を行う．IF文の条件節と実行節や，IF文同士の並列化を行う．

(注7) 右上の回路と右下の回路では，クロック周期も異なっている．右上は加算器と乗算器を逐次に実行する遅延時間が必要で，右下は，セレクタ二つと乗算器を実行する遅延時間が必要．

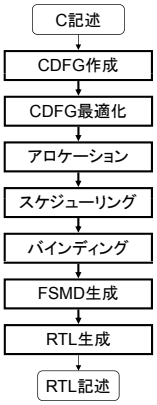


図3　高位合成の工程

このように，高位合成は，逐次動作のアルゴリズム記述から，演算器やメモリポート数等の資源制約や，実行サイクル数や遅延時間等の時間制約を守りながら，高速だが大きい回路や，低速だが小さい回路等様々な回路を合成する．以下，高位合成の原理を図2の右下の回路を合成する例を用いて説明する．

高位合成の一般的なフローを，図3に示す．最初に，通常のコンパイラ同様にC言語を解釈しパーズ木（Abstract Syntax Tree，以下ASTと呼ぶ）を作成する．ASTから演算間のデータ依存関係や制御依存関係を表す「CDFGを作成」する．高位合成では，このCDFG上でほぼ全ての操作を行う．次に「CDFGを最適化（変形）」し，演算や配列アクセスの実行ステップを決定する「スケジューリング」，演算子や変数，配列を，実際の演算器や，レジスタ，メモリに割り付ける「バインディング」が行われる．最後に，合成された回路のマイクロアーキテクチャをRTL言語（Verilog言語やVHDL言語）に変換する．RTL言語は，論理合成ツールに読み込まれて，下流のツールを経て，SoCとなる．

2.1　入力言語

高位合成の入力言語として，かつてはVHDLの動作記述等も用いられたが，現時点では実用レベルの高位合成ツールではC言語の系統が使われることが多い．アルゴリズム系開発を行うユーザはCやC++といったソフトウェア分野で標準の言語を使うことが多いこと，組込み分野では現在でもC言語が主力で使われていることの影響が大きい．ハードウェア設計者やEDA[注8]関係者は，SystemC[2]と呼ばれるLSI設計業界での標準化言語も利用している．CやC++言語をハードウェア記述に利用する際に足りない概念は幾つもあり，例えばその一つが，変数のビット幅の記述と入出力の宣言である．ハードウェアでは，全て32 bitや64 bit処理では，ハードウェアの無駄が多く，5 bitや12 bit等を定義できるとよい．逆に256 bitや1,024 bit等の大きなビット数を定義する必要もある．また，ハードウェアは並列に動くプロセスが複数存在するのが通常であるので，threadのような機構か，プロセスが並列に結合している構造を表現できる必要もある．更に，それらの間の通信で同期をとる必要があるので，ある信号が1になるまでストップするwait()文のような拡張も加えられる．SystemCは上記のような拡張，すなわちビット幅，入出力，wait()文，マクロの並列性を示すthread等がライブラリの形で提供されている．他のC準拠言語も多少の拡張や，合成用プラグマ等で上記拡張概念を高位合成に知らせる仕組みを持っている．近年の高位合成系は自動化が進んでいるため，入出力端子の宣言とそのビット幅を定義すれば，他は通常のCやC++記述のままでよいツールも増えてきている．研究としてはJava，C#，CUDA，OpenCL等様々な言語からの合成系も発表されている．

(注8) Electronic Design Automation:電子機器，半導体の設計を自動化するツールの総称．電気系のCADとも呼ばれる．

2.2　コントロールデータフローグラフ（CDFG）の作成

動作記述からデータの流れを示すデータフローグラフ（DFG）を作成する．DFGは演算の実行順序の依存関係を示している．図2の動作に対応するDFGを図4に示す．このDFGは，図2の右上の回路と等価になっていることに注意されたい．RTL記述は，回路構造を言語形式で表現したものであるから，基本的に高位合成でいうDFGと同等の構成となる．このほかに，IF文やループ文，GOTO文等プログラムの制御構造に起因する依存関係があり，これを表すグラフを制御フローグラフ（CFG）と呼ぶ．これら制御依存とデータ依存の両方を一つのグラフで表したものを制御データ依存グラフ（CDFG）と呼び，高位合成はCDFG上で合成を行うものが多い．

2.3　スケジューリングとアロケーション

CDFG上の演算をある制約条件化で並列化し，実行サイクルを決定する工程をスケジューリングと呼ぶ．図4のCDFG（動作記述には制御構造がないためCDFGとDFGが同じ形になる）を，加算器1個，乗算器1個（「演算器制約1-1」と呼ぶ）でスケジューリングする．ここで，説明を簡潔にするために，加算器と乗算器の遅延時間がクロック周期よりやや少なく，1クロックに1演算とセレクタだけが逐次に実行可能で，加算と乗算を逐次に1サイクルでは実行できないものとする[注9]．

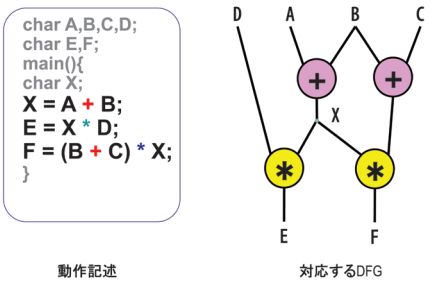


図4　動作記述と対応するCDFG（DFG）

また，全ての入力は第一サイクルで読み込まれることとする．

上記仮定の下で，スケジューリングを行う．図5に図4のCDFGを演算器制約1-1の下でスケジューリングした結果を示す．スケジューリング処理としては，まず最初のサイクルで実行可能な演算のリストを作成する．この例では，入力A，B，Cに直接つながっている二つの加算が第一サイクルで実行可能である．乗算は加算が実行してからでないと実行できない．加算器は1個しかないから，二つの加算のどちらかを実行することになる．このどちらを選べば最短のサイクル数でスケジューリングできるかを考える問題がスケジューリングアルゴリズムの基本である．実用システムでよく利用されているのは，「リストスケジューリング」と呼ぶアルゴリズムである．例えば，演算の結果を後で利用する演算の多いものを優先する等の評価関数を用いて選択する．他の演算もスケジューリングした結果を図5に示す．ここでは，A+Bを第一サイクルで実行する加算結果をレジスタに格納する（図5の□印）．他の入力もレジスタに格納する(注10)．第二サイクルでは，乗算X*Dと，加算B+Cが実行可能である．それぞれの演算結果とXの値をレジスタに格納する．第三サイクルでX*Yを実行する．以上のように，このCDFGは演算器制約1-1の下で3サイクルで実行が可能である．上記のように演算器の数やメモリのポート数，入出力の本数等の演算資源を制約として，サイクル数最小を目的とするスケジューリング手法を，「資源制約スケジューリング」と呼ぶ．反対に，サイクル数を制約にして，利用する資源数最小を目的とする手法を「時間制約スケジューリング」と呼ぶ．スケジューリングの際に演算器の種類や数を決定する必要がある．この工程をアロケーションと呼ぶ．例えば，2 bit加算器1個と，4 bit加算器1個を使うのか，4 bit加算器1個を使うか，1ポートメモリを3個使うか，2ポートメモリを2個使うか等を決める工程がアロケーションである．資源制約スケジューリングの場合は，アロケーションで演算器の種類だけでなく個数も決定するが，時間制約スケジューリングでは演算器種類だけを決定し，演算器数はスケジューラで決定する．スケジューリングアルゴリズムは，簡単なモデル化を行った場合でもNP困難であり，現実的な規模の合成課題に対して厳密解を得ることは難しく，ヒューリスティック解法で解いている．

2.4 バインディング

動作記述上の演算子を演算器に，変数や配列をレジスタやメモリにマッピングすることをバインディングと呼ぶ．図5の例では，加算器と乗算器は1個ずつしかないため，演算器のバインディングは一意に決まる．図5で演算の横にADD，

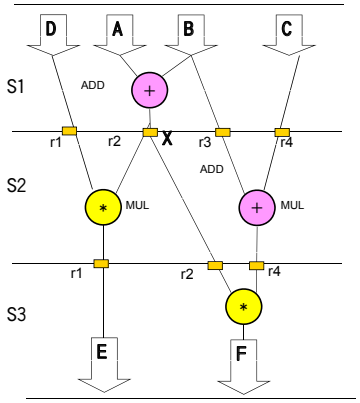


図5　図4のCDFGのスケジューリングとバインディング結果

MULと表示があるが，それぞれの演算子がADDとMULという演算器にバインドされたことを示している．次にレジスタをバインディングする．動作記述の変数名に合わせてそれぞれ個別のレジスタを利用すると多数のレジスタが必要となってしまう．CDFG上の□がレジスタを表しているから，これらの□に物理的なレジスタをバインドすればよい．図で□の側のr1，r2，r3，r4が，4個のレジスタr1，r2，r3，r4にバインドされたことを示す．ここで，変数DとEは，一つのレジスタr1にバインドされている．これを，変数DとEがレジスタr1を共用（シェア）したという．変数Dが定義されてから最後に参照される期間を変数のライフタイムと呼ぶが，変数Dと変数Eはライフタイムが重ならないためレジスタシェアが可能となる．同様に同時に利用されない演算子間でも演算器シェアが可能である．ライフタイムが重ならないのは，上記のように実行サイクルの違いによるものだけでなく，動作記述の条件排他文（IF文やSWITCH文）がある場合なら同一サイクルでもシェアが可能である．入力端子A，B，C，Dと出力端子E，Fはシェアすることなく，それぞれ個別の入力端子，出力端子にバインドした(注11)．以上のように，バインディングとはCDFG上の演算子や変数，配列，入出力を示すノードに，それを実現する演算器やレジスタ等の物理名を割り付ける工程である．バインディング主目標は，演算器やレジスタの数や，セレクタの数を最適化することである．ほかに，レジスターレジスタ間の遅延を下げること（つまり，周波数を上げること），フォールスループの除去やフォールスパスの削減等も目標となる．演算や変数でシェア可能なものを接続したグラフを作ると，バインドは最大クリーク分割問題に帰着できるが，それでもNP困難な問題とされており，厳密解を得ることは難しい．

制御回路とデータパスの生成

上記スケジューリングとバインディングで回路構成は決定している．レジスタや演算器の間の転送関係を示す「転送表」を作成し，ブロック図を描くことができる．転送表は，レジ

(注9) 現実の設計例では，1サイクルに演算を逐次に，複数段実行する（演算チェーン）が普通である．2 bit加算器と32 bit乗算器では遅延時間が大きく違い，32 bit乗算器が実行可能なクロック周期では，2 bit加算器は逐次に何段も実行できるはずである．

(注10) 入力が第一サイクルでしか入らないから．第二サイクルに入力可能ならば，第一サイクルでレジスタに格納する必要はなく，2サイクル目に読み込めばよい．

(注11) 入力変数や出力変数も同時に利用しない場合は入出力端子をシェア可能である．

転送先	転送元	条件	状態
r1	D	1	S1
	MUL.out	1	S2
r2	ADD.out	1	S1
	r2	1	S2
r3	B	1	S1
r4	C	1	S1
	ADD.out	1	S2

(a)

転送先	転送元	条件	状態
ADD.L	A	1	S1
	r3	1	S2
ADD.R	B	1	S2
	r4	1	S3
MUL.L	r1	1	S2
	r2	1	S3
MUL.R	r2	1	S2
	r4	1	S3

(b)

転送先	転送元	条件	状態
E	r1	1	S3
F	MUL.out	1	S3

(c)

図6　転送表　　(a) レジスタ転送表，(b) 演算器入力端子転送表，(c) 出力端子転送表

スタ，演算器の入力端子，回路の出力端子等のデータ転送先ごとに，データ転送される元の信号名とその転送が行われる状態（サイクル）と，実行条件を記述した表である．図5のスケジューリングとバインディングの結果に対する転送表を図6に示す．ADD.Lは加算器ADDの左入力端子，ADD.Rは右入ADD.outは出力端子を表す．MULも同様．条件の1は常に真を示す．図4の動作記述に条件文がないため条件が全て1になっている．条件分岐文がある場合は，条件節の演算等が「条件」になる．レジスタ転送表(a)の1行目はレジスタr1は状態S1で入力変数Dから，状態S2で乗算器MULの出力端子からデータを常に入力することを示している．セレクタの左右の入力の選択信号が省略されている．転送表のとおりに動作するように制御FSMの状態信号で選択する．例えば，加算器の左入力のセレクタは，状態S1で左入力である入力端子Aを選択し，状態S2で右入力であるレジスタr3を選択する．他も同様である．この転送表を元に，描かれたブロック図を図7に示す．左の状態遷移図は制御用の有限状態機械（Finite State Machine; FSM）の状態遷移を示している．スケジューリング結果の3サイクルの逐次動作を3状態のFSMで示している．右が転送表から導かれたデータパスである．ブロック図はRTL記述と同レベルである．このRTLを論理合成ツールにかけることにより，ゲート回路に変換できる．このように制御をFSMで行い，演算実行をデータパスで行うというアーキテクチャをFSMDと呼ぶ．多くの高位合成はこのFSMDを合成のターゲットアーキテクチャに採用している．

2.5　CDFG最適化（トランスフォーメーション）

図4のCDFGは非常に簡単な例だったため，CDFGの最適化は必要なかった．実用的な記述の場合は，CDFGに対し様々な最適化を行う．CDFGの形状を変更する場合はトランス

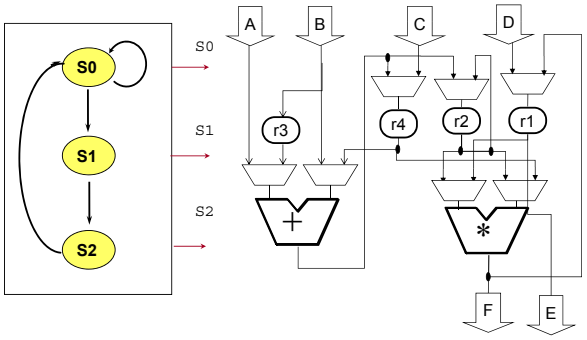
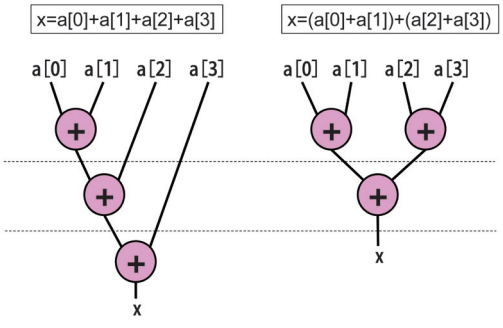


図7　合成されたFSMD

フォーメーション[3]と呼ぶ．CDFG最適化は，コンパイラの言語レベルの最適化と同等である．高位合成ツールでも言語レベル（やAST）で最適化してもよい．CDFG最適化は，非常に様々なものがある．例えば，C言語の変数のintは，プロセッサのワード長に合わせて32 bitや64 bitと考えられるが，ハードウェアではぎりぎりの大きさにしたい．入力や出力のビット幅や値域を設計者が与えることにより，それにつながる演算や変数のビット幅を削減することができる．ビット幅の削減により，面積や遅延を大幅に改良することができる．また，出力に関係ないコード群（デッドコード）を削除したり，a = 3;b = a+4; という文を，b = 7とする等の定数の伝搬，関数のインライン展開，ループのアンローリング等コンパイラで知られた最適化も行われる．また，C記述で，配列a[i]の値を何度も読み出す記述をした場合に，もし，配列aをメモリにバインドした場合，配列アクセスはメモリのポート数分しか1サイクルにアクセスできない．このような場合は，a[i]の結果を一旦レジスタに保持してから繰り返し利用すると，メモリのポート数に関係なく，並列に値を読み出すことが可能となる．また，ループボディにx+= a[i]というような記述を持つループを展開した場合，“x = a[0]+a[1]+a[2]+a[3]”というような記述となる．C言語の意味では，左優先で値を計算するが，その場合，多数の加算を逐次に実行する必要があり，サイクル数が多くなってしまう．高位合成では加算の順序を変更して並列度を上げることができる．図8に先の例の変換する例を示す．図8 (a) は，加算を左優先で逐次に実行する場合のDFGである．図8 (b) は加算の実行順序を変更し，x =



(a)　　(b)

図8　Tree Height Reduction　　(a) 逐次実行のDFG，(b) THR後のDFG

(a[0]+a[1]) + (a[2]+a[3]) に対応するDFGになっている．図8 (a) は逐次に三つの加算，図8 (b) は逐次に二つの加算となるので，加算器が2個以上ある場合は，サイクル数を削減できる．このような変換はDFGの木の高さが減少することになるためTree Height Reductionと呼ばれる．ソフトウェアコンパイラでは演算の実行順序を変更すると，演算結果の桁あふれ等により結果が変わる可能性があるため，通常禁止されている．高位合成では桁あふれ等がないように演算器のビット幅が自動調整されるため，演算の実行順序の変更が可能である．

また，初期化付き配列を，メモリでなく組合せ回路で実現する場合は，CDFGの配列ノードを組合せ回路ノードに変更する等，ハードウェア構成を意識したトランスフォーメーションも適用可能である．配列の添え字の解析による最適化等，信号処理で研究されてきた手法も例題によって大きな効果がある．また，a*b+a*cをa* (b+c) とする等，数式処理をある程度行うシステムもいろいろ研究されているが，どのような数式変換が良い結果をもたらすかの判断はスケジューリング，バインディングと同時でないと難しい．CDFG最適化は，高位合成結果に大きな影響を持つが，全ての最適化技法が入ったツールは存在しないので，ツールで最適化されない場合は，C記述を設計者が人手で最適化する必要がある．このような言語レベルの最適化は非常に多くのバリエーションを持っており，互いに副作用を持つことも多く，ツール設計が難しい部分であるが，効果も非常に高い．

2.6　高位合成の各工程の相互依存性

図3の各工程は，それぞれ独立な工程として描かれているが，高品質な回路を合成するためには，それぞれの工程を融合する必要がある．CDFGの最適化，スケジューリング，バインディングはそれぞれ依存し合っており，全て同時に解くのが理想である．例えば，スケジューリングをしながらCDFG変換をしたり，スケジューリングとバインディングを並行して行うなどである．また，ディープサブミクロンのLSIやFPGAでは配線遅延が大きいため，レイアウトも並行して行うのが理想である．ただ，それぞれNP困難な問題であるため，全てを一度に解く効率的な手法の開発が難しく，他の工程の影響を考慮しつつ，それぞれの問題を解くのが現時点での現実解である．

3．高位合成ツールの合成回路

多くの高位合成はFSMDを合成する．FSMDは制御用の有限状態機械 (Finite State Machine) と，演算を実行するデータパス (Datapath) からなるシーケンシャル動作実行可能なマイクロアーキテクチャである．FSMとデータパスの間は通常制御信号 (状態デコード信号や比較器等の出力等) で結ばれる．このような高位合成のターゲットアーキテクチャの構造や制御信号のネーミングルール等を理解していると，合成されたRTL構造の把握が容易となり，デバッグやECO(注12)に有

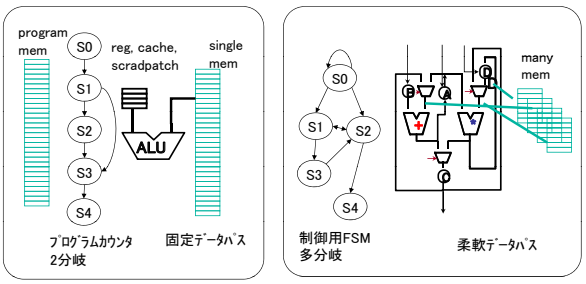


図9　CPUと高位合成回路をFSMDでモデル化

効である．FSMDは，逐次の動作記述をデータパスの持つ演算資源を並列実行する．CPUは，その特殊な形式とも考えられる．すなわち，ALUが一つ (または数個) しかなく，逐次に命令を実行しており，FSMが2分岐しかないカウンタで構成されている．完全並列回路は逐次記述を完全に並列に実行する回路で，パイプライン回路等がよく使われるが，回路規模が大きくなる．図9にCPU (左) と高位合成される回路 (右) をFSMDで表現したモデルを示す．CPUはALUが1個または数個で，分岐も通常2分岐しかできない．データパスは固定である．メモリ空間は一次元で配列群は全てこのメモリ空間に保持され，1個ずつしかアクセスできない (キャッシュミスやパイプラインハザード等はここでは考えない)．一方高位合成回路は演算器の種類や数が多く，制御も複雑な多分岐を実現できる．メモリも配列ごと等で独立に保持でき (CPUが配列数分のスクラッチパッドメモリを保持しているような状況)，多数の配列に同時にアクセス可能である．データパスがC記述に合わせて最適化されるため，例えば，多数のレジスタ間の転送も並列に1クロックで行える．また，ビット処理もシフトマスク等は必要なく，配線の接続だけで行える．更に分岐やループを並列実行することも可能である (FSMは複数持てる)．つまり，FSMDはCPUのような小面積回路から，完全並列回路のような高性能・大面積回路まで実現できる．高位合成ツールは演算資源に合わせて適切なFSMDを合成できる．信号処理や暗号，通信I/Fをはじめほとんどのディジタルアプリケーションはは FSMDで対応可能である．したがって，原理的には，高位合成はFSMDが対応可能なアプリケーションは全て高位合成ツールで合成可能である(注13)．

4．高位合成の効果

高位合成を適用し，それに適した設計フローを構築することによりRTL設計に対し多くの効果が期待できる．設計期間や設計工数の削減等の「開発費の削減」，SoCのハードウェアと組込みソフトウェアの協調検証による「信頼性の向上」，性能やクロック周波数の向上やチップ面積や消費電力の削減と

(注12) Engineering Change Order: LSIのレイアウト後に空きセルと空き端子を利用して，LSIの論理や遅延を補修すること．高位合成や論理合成から全ての工程をやり直す時間がない場合に利用する．

(注13) 高位合成は制御系回路に向かない等ということがよくあるが，それは特定のツールが制御系回路に向かないという意味である．高位合成手法は原理的には制御系，データ処理系の両方に適用可能である．

いった「チップコスト削減や性能向上」等がある．更に，RTL設計ではハードウェア化できなかったような複雑なアルゴリズムを純粋なハードウェア化することが可能となり，CPU実行に対して高速化できるなど装置の機能性の向上効果もある．昨今のディジカメや高機能プリンタではこのような技術が利用されている．以下，高位合成の代表的な効果，利便性について説明する．

(1) 設計効率化（設計期間削減，設計工数削減，信頼性向上）による開発費削減

設計の記述レベルをRTLから動作レベルに抽象度を上げることにより，記述に必要な行数が数分の一から数十分の一になり，設計工数の大幅削減が可能となる[(4)]．また，動作レベルとRTLではシミュレーション速度が数十倍から数百倍異なり，チップの正しさの検証に必要な時間も大幅に縮小される．高度な暗号や顔照合等の複雑な処理ではRTLシミュレーションでは数週間から数か月かかるような場合もあり，SoC設計においては動作レベルで設計・検証することが必須になっている．C設計により，よりたくさんの検証が可能となりチップ品質も向上する．また，RTLシミュレーションは大規模SoCでは数Hz程度しかでないことも多く，ハードウェアと組込みソフトウェアの協調シミュレーションすることが時間的にできず，SoCができてから組込みソフトウェアの検証が必要であった．ハードウェア記述をC記述にすることにより，シミュレーションが高速化し，協調シミュレーションが現実的になり，組込みソフトをチップ製造の前に検証することが可能となる．

(2) アーキテクチャ探索による高性能・小面積な回路の合成によるチップコスト削減

「高位合成ツールを適用すると設計生産性は向上するが，性能・面積等の回路品質は人手設計より低下するはずだ」という認識があるが，必ずしも正しくない．原理的にだけいえば，高位合成回路が人手RTL設計回路を凌駕する可能性が高い．理由は，設計者が通常考えないような回路構成をも高位合成は探索するからである．図4の左の動作記述に対して，設計者は右上のような並列回路を設計する傾向がある．これに対し動作合成は面積制約や時間制約に合わせて，時間方向

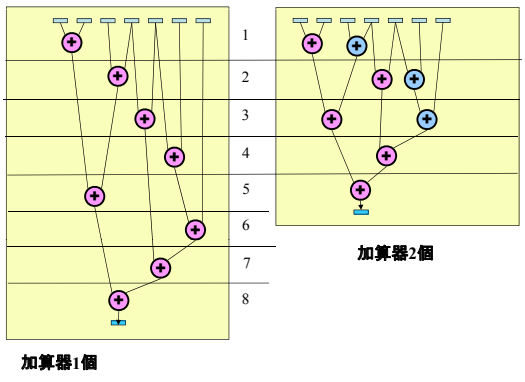


図10　異なる演算器制約に対するスケジューリング結果

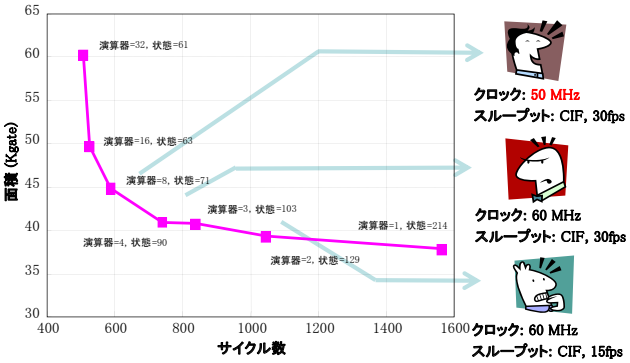


図11　面積と性能のトレードオフの例

に折り畳むことにより小面積な右下のような回路を合成することができる．演算資源制約を変更してスケジューリングすることで様々な回路を出す例を図10に示す．同一のCDFGを加算器1個と2個という二つの演算器制約でスケジューリングしている．このように，利用可能な演算資源の数を変更することで，実行サイクル数（性能）とのトレードオフを考えた設計が可能となる．高位合成は，資源が1個から数百個までしらみつぶしに探索することができるのに対し，人手RTL設計では，経験や勘等で資源数を決めて一種類を設計するしかない．そして，そのRTL記述を他の回路でも使い回す．この事実が，高位合成が人手RTL設計をしのぐ回路を合成できる理由の一つである．

より現実的な例としてMPEG4向けの逆コサイン変換の「面積とレイテンシ[(注14)]のトレードオフカーブ」を図11に示す．通常，設計者は左端の性能の高い（並列性の高い）RTL記述を一つ作成し，様々なプロジェクトで再利用する．RTLの設計・検証には多大な工数がかかるためである．高位合成の場合は，RTL記述ではなく，動作記述を使い回し，設計制約性能に合わせたRTLを合成できるため，より小さな回路を合成することが可能となる．例えば，携帯電話向けの誤り訂正符号のViterbiデコーダの人手RTL設計と高位合成で比較した場合，高位合成は10分の1の面積の回路を合成できた．携帯電話は高い性能を要求しないため，レジスタや演算器のシェアにより面積を小さくすることができている．人手設計では，デバッグのやりやすさや再利用性を考慮して，このような最小化回路を設計することは少ない．このように性能に即して，時間方向に折り畳んだ回路を合成するときは，高位合成の回路最小化は大きな効果がある．一方，データを毎クロック読み込むようなパイプライン回路の場合は，人手設計も高位合成も原理的にほぼ同じアーキテクチャの回路となり，面積性能に大きな差が出ることはない．

また，合成回路の性能に関しても人手設計を凌駕する場合はしばしば存在する．特に，条件分岐をたくさん含むような動作記述のスケジューリングは，スペキュレーション[(注15)]や

(注14) 実行に必要なサイクル数．

(注15) 投機的実行　if文やswitch文等の条件分岐がある場合に，条件節の真偽が計算される前に，THEN節やELSE節の演算を実行すること．CPUでは条件分岐の結果を予測して，どちらかの分岐を実行することが多いため，投機的実行と呼ばれる．

複数の条件分岐文を並列化するため，また，複雑な処理のパイプライン化は人手設計では極めて困難であるため，高位合成適用の効果が高い．

(3) 設計資産の再利用性の高さ

先に述べたように，設計資産を動作レベル（CやSystemC言語）で保持する場合と，RTLで保持する場合を比較すると，動作レベル資産の方の再利用性がはるかに高い．RTLの設計資産では実行サイクル数や通信I/Oの変更は難しく，大きな周波数の変更も難しい．一方，動作レベルの設計資産の場合は，前節で示したとおり，一つの動作記述から合成制約を変更するだけで異なるサイクル数や面積のRTLを合成でき，変更は容易である．性能，周波数は，制約を変更して高位合成するだけで変更可能であるし，通信I/Oプロトコル，機能の変更や増減も，動作記述の軽微な修正や，動作記述レベルのパラメータ記述（テンプレートや#DEFINE）で変更可能である．高位合成は，新規設計時の設計効率の向上に寄与するが，それ以上に，既存の動作記述を再利用する場合の効果がRTL設計に比べはるかに高い．信号処理や暗号処理等，対象データによって性能が大きく変わるような分野や，通信プロトコルが変更されるような分野等で効果が高い．

(4) 大規模複雑なシーケンス制御回路の設計容易化，変更容易化

高位合成ツールの動作記述は通常クロックの概念のないUntimed記述が一般的であり，自動スケジューリングで回路を合成するというイメージが強く，SDRAMやSDカードの通信モジュール，シーケンサ，制御回路等の通信シーケンスを持つ回路の設計には向かないという認識が広がっている．しかし，動作記述にクロック境界を記述可能とすると，C言語やSystemC言語で制御回路のシーケンスを表現し，合成することができる．簡単な例を示しながら，クロックを意識した動作記述の考え方を説明する．例えば，出力端子Xに，値1を1サイクル間，2を2サイクル間，3を3サイクル間出力する回路のRTL記述と動作記述を図12に示す．図の記述例は，NECのCyberWorkBench向けのC言語で利用されている

```
switch (State) {
  case 0: { if ( c )
              { X=1;
                State=1;}
            else
              { State=0; }
  case 1: { X=2;
            State=2;}
  case 2: { X=2;
            State=3;}
  case 3: { X=3;
            State=4;}
  case 4: {X=3;
            State=5 ;}
  case 5: {X=3;
            State=0 ;}
}
```

(a)

```
while(1) {
wait( c );
 X=1;  // 1を1clock間出力
$
 X=2;  // 2を2clock間出力
$
$
 X=3;  // 3を3clock間出力
$
$
$
}
```

(b)

```
while (1) {
wait( c );
 // iをi clock間出力
for (i=1;i<=3;i+) {
  X=i;
  for (j=1;j<=i, i++) $
}
}
```

(c)

図12　状態遷移のRTL記述（明示的FSM）と動作記述（暗黙的FSM）　(a) 明示的FSM（RTL），(b) 暗黙的FSM（動作），(c) 暗黙的FSM

“$”記号でクロック境界を示している[5]（注16）．SystemCのsc_cthreadのwait()文でも同様に考えることができる．図12 (a) のRTL記述では，状態レジスタ（State）を陽に定義して，状態遷移によって動作を記述する．明示的なFSMと呼ぶ．一方，図12 (b)，(c) は，等価な動作を動作記述した例であり，プログラム同様上から下へ実行される．$と$の間の演算群が1クロックで実行され，次のクロックで$の次の演算群を実行する（注17）．こちらもFSMの動きを記述しているが，状態遷移や状態レジスタを明示せず，暗黙的FSMと呼ばれる．$記号一つで1クロックであるから，$が二つある場合は2クロック消費することになる．動作記述ではXは変数であるから，代入は一度行えばよい．回路的には，端子とレジスタを組み合わせて，複数クロックの間値を保持して出力する．このような回路実装は高位合成が作成する．明示的FSMで何十状態，何百状態の状態遷移を正しく記述し，仕様変更があった場合に正しく修正するのは困難である．動作記述では，クロック動作が上から下へ進んでいく（逐次記述）．関数やIF文，ループ等の構造化プログラミングの制御構造が利用できるため，タイミングを考慮した上での，記述・修正が容易である．図12の例で，変数2を3サイクル出すことに変更する場合，(a) の明示的FSMでは状態2と3の間に新たな状態を追加して，状態遷移の修正が必要だが，(b) の動作記述なら，X = 2の後ろの$を一つ追加するだけでよい．なお，動作記述ではGOTO文も利用できるが，GOTO文だけで書くと，明示的FSMに近くなってしまうので，構造化言語風に記述するのが修正しやすく，理解しやすい記述の肝となる．

5. 実用的な高位合成のアルゴリズム

高位合成は，アロケーション，スケジューリング，バインディング等の最適化問題の集合である．それぞれが，NP困難等複雑な問題であり，それぞれの最適解を求めるのも困難であり，それぞれの最適解は全体の最適解である保証はないため，ある程度以上のサイズの記述の最適解を求めるのは現実的なアプローチではない．また，高位合成では言語からCDFGに変換する部分でも多様なバリエーションがある．現実的な計算時間で，相応の合成結果を求めるように高位合成アルゴリズムを開発しなければならない．以下，高位合成アルゴリズムで注意すべき点を幾つか述べる．

5.1 制御構造の扱い

逐次記述である動作記述から，高性能なRTLを合成するためには，スケジューラの並列性の抽出が重要である．演算レベルの並列性だけでなく，IF文，ループ文，GOTO文等制御

（注16）$記号はANSI-Cにはない追加文法である．$をヌル文字にdefineすることでCコンパイル可能となる．$を意識したサイクルシミュレーションもCyberWorkBenchの環境では可能である．

（注17）SystemCのwait()文は，waitとwaitの間は1クロックという意味はないが，高位合成がそのように合成すると同様のFSMが合成できる．

の依存関係がある部分の並列化が，CPU向けのソフトウェアコンパイラに比べても格段に重要となる．IF文に関しては，スペキュレーションとIF文自体の並列化が鍵である．スペキュレーションは，条件節よりもTHEN節，ELSE節の演算を先に実行することを示すが，THEN節，ELSE節の片方，両方をスペキュレーションする等あらゆる形態のスペキュレーションが実行可能である．また，IF文やループ文自体を複数並列化する技法も重要である．IF文間にデータ依存関係がある場合でも，並列化は可能である．CPUでは分岐を制御するブランチユニットは2分岐を一つ持つ程度のため，コンパイラが並列化することはできないが，高位合成の場合は多数のIF文の並列化，スペキュレーションを同時に行うことができる[6]．

5.2 演算チェーン，複合演算の扱い

CPUは1ワード幅のALUしか持っていないことが多く，1クロックに1ALU実行が通常である．しかし，ハードウェアの場合，2，3 bitの演算器から数十bitの演算器までいろいろな演算器が利用でき，それぞれの遅延が大きくことなるため，1クロックで複数の演算を逐次に実行できるのが普通である．例えば，小さいビット幅の演算なら，a+b+c+d+e等多くの演算を1クロックで実行可能である．このように演算を逐次に実行することを演算チェーンと呼ぶ．加算器を連続した場合，全体の演算時間がそれぞれの演算時間の総和より短くなることを「演算チェーン効果」と呼ぶ．例えば，遅延5 nsの加算器を二つ逐次に実行する場合，10 nsより小さい7 ns程度で実行できる．加算器の遅延時間が入力LSB(注18)から出力MSB(注19)までの最長経路で決まることが多いが，加算器を二つチェーンさせると，前段の加算器のLSBがすぐに後段の加算器のLSBに入力し計算が始まることで，チェーン効果が生まれる．更に，5入力加算器や，積和演算器など複数の演算を実行可能な演算器を専用組合せ回路として合成する場合もあり，これらを考慮することは重要である．演算チェーンを扱えない高位合成のスケジューリングアルゴリズムは，実用的な価値はない（ソフトウェアコンパイラはチェーンを考慮する必要は通常ない）．整数線形計画法等を利用した手法は，演算チェーンを扱うと計算量が著しく増大して実用に値しないことが多く，ツール設計では注意が必要である．

5.3 ビット幅最適化

ソフトウェア向けのC記述は，全ての変数がint，char等で記述されている．しかし，ハードウェアでは12 bit，48 bit等のint，charでは表現できない様々なビット幅を扱う必要がある．そのため，高位合成向けのC言語，SystemC言語は，任意のビット幅を表現できる記法を持っているのが普通である．また，入力信号と出力信号以外の信号のビット幅を指定するのは面倒であるから，それらの変数を入出力信号の値域から自動的に推定し，ビット幅を自動的に調整する機能が必要である．桁あふれ等が起こらないように調整は行われるため，ソフトウェア用のC言語設計の場合と異なり，桁あふれ等を設計者がケアする必要はない（少ない）．また，高位合成用のC言語は固定小数点を型として持っており，そのまま合成できるツールが多く，この場合，設計者は小数点の位置の把握やオーバーフロー時の処理等をツールに任せることができる[7]．

5.4 配列アクセスの最適化，配列添え字の排他性解析

信号処理では配列添え字の解析が非常に重要である．典型的な例を挙げる．

```
for（i = 0; i<7; i++）{x[i] = a[i] + a[i+1]+a[i+2];}
```

という記述で，配列aを記述どおり単純にメモリにバインドすると，メモリアクセスが1回のループで3回必要になる．この記述を，

```
t = a[0]; s = a[1];
for（i = 0; i<7;i++）{ u = a[i+2]; x[i] = t+s+u , t = s; s = u; }
```

と書き換えれば，配列aをバインドしたメモリにはループ一度に1回アクセスすればよくなるため，高速化が期待できる．このような記述は画像処理等でしばしば見られるため，様々な方法が提案されている[8]，[9]．上記はCPU(注20)やDSP(注21)，GPU等でのコンパイラでも重要な技術であるが，ハードウェアコンパイラで重要なのは添え字の排他性解析である．CPUの配列は通常1ポートメモリ上にあると考えられるが，ハードウェアの配列はマルチポートメモリやレジスタファイルにあるのが一般的であり，配列の異なるアドレスには同時にアクセス可能であることが多い．そのため，添え字が同一でないことを認識できれば，並列度を向上することが可能となる．

5.5 自動パイプライン化（ループフォールディング）

画像などのストリームデータの処理は，通常パイプライン処理で実現される．高位合成の場合は，動作記述で書かれたプロセス全体をパイプライン回路にするか，動作記述中のループ文のイタレーションを自動的にパイプライン化することで行われる[10]．パイプライン化では，メモリのアクセス競合や演算器の競合（構造ハザード），あるイタレーションでの計算結果を次のイタレーションで利用する場合のスケジューリング等人手設計では設計ミスが入りやすい部分であるた

（注18）Least SignificantBit: 最下位ビット
（注19）Most Significant Bit: 最上位ビット
（注20）Digital Signal Processor: ディジタル信号処理に特化したプロセッサ（現在では音声処理が主）．
（注21）Graphics Processing Unit:3Dグラフィックス等の処理に特化したプロセッサ．

め，自動化の効果が顕著に現れる．高位合成を，パイプライン処理にしか適用していないユーザも多い．人手設計のパイプラインはデータが毎サイクル入る回路がほとんどであるが，高位合成の場合は，2サイクルごと，3サイクルごとに1回データを読むなども合成することで，ハードウェア面積を節約することが可能である．パイプラインをストールした場合のデータの保持機構等は動作記述しなくても自動で付加されるため設計効率化とともに設計ミスも防げる．また，ネストしたループ文を，多重パイプラインにする等は人手設計者では一般的に難しく余り採用されていないが，高位合成を利用すると簡単に実現できる[11),12)]．ループは様々な最適化が可能であり，複数のループを一つのループに融合するループマージ[13)]や，固定回ループを回数分展開してしまうループアンロール等様々な最適化が適用でき，どの最適化を選択するかで，面積，性能に大きな差が出る．

5.6 高級データ構造

高位合成では，多次元配列，構造体，ポインタ，クラス等高級データ構造を用いてハードウェアの動作記述を行うことができるようになっている．CPUではこれらは一次元のメモリ空間に展開すると考えるのが普通であるが，高位合成ではこれらを，別々のメモリやレジスタ等に割り当て，並列にアクセスできるようにするのが普通である．例えば，a=b[i]+c[i]; のような記述で，配列b，cを別々のメモリにバインドし，同時にアクセス可能とする．ポインタや構造体も1クロックでアクセスするようにハード化する．ただし，面積とのトレードオフを考えた上で選択する必要がある．

5.7 再帰呼出し，動的データ構造

現在の市販の高位合成が合成対象外としている典型的な構文は，再帰呼出しと動的データ構造である．それぞれ，合成自体は原理的には困難ではない[14)~16)]．再帰呼出しでは，関数の内部変数分の数のスタックを用意して，全て並列に読み書きすることで高速な再起呼出しの回路は回路は合成可能である(注22)．しかし，再帰の呼出し回数がコンパイル時に静的に決定されず，実行時に動的に決まる場合にハードウェアに不向きなこと，内部変数分のスタックを全て用意するのは面積的に不利なこと，スタックを一つのメモリで実現すると，実行性能がCPU並みになってしまうことなどの問題がある．動的データ構造も，実行時に必要なデータ量が決定するのでハードウェアには向かない．シミュレーション等で必要なデータ量を十分に予測した上で，高位合成することは可能である．リンクドリストや動的なポインタ等も同様の理由で，ハードウェア向きでないため，現在の市販ツールは対象外としている．ただし，高位合成を組込み用のASICの設計に利用する場合ではなく，C記述を動的再構成プロセッサやFPGAで加速化するような用途では対応する必要がある．

(注22) CPUと同様に単一スタックを用いて実現することも可能であるが，その場合は高速化ができないので，ハードウェア化する意義が低くなる．

5.8 レイアウトツールの考慮

高位合成で注意しなければならないのは，配線の集中である．配線が集中すると，ASIC，FPGAともレイアウトが難しくなり，近くに配置できずに遅延が大きくなったり，配線が不能になってしまうこともある．配線を意識しないで，論理合成結果だけを見ると，問題が認知できないので注意が必要である．例えば，以下の例を考える．

```
a[i+5] = a[i] + a[i+1] + a[i+3] + a[i+4]:
```

ここで，配列aをレジスタ群で実現し，この1行を1サイクルで実行する回路を合成するとする．その場合，五つのアドレスをデコードするためのデコーダが五つ必要になる．これらのデコーダが，レジスタの入力，出力に必要になるため，配線は非常に込み入った回路となってしまう．レジスタ群で実現した多数の配列を並列アクセスするのは，配線の問題で非常に難しい．配線本数などツール警告に注意することが重要である．このように配列をレジスタ群で実現する場合，添え字のデコーダが回路規模，配線とも大きいため，ループの展開等により，添え字を定数化することが，高性能化，小面積のためには重要である．

また，近年のディープサブミクロンのASICでは配線遅延がゲート遅延に比べ相対的に大きくなっており，高位合成時に概略配置を意識して配線遅延を考慮してスケジューリング等を行う必要が出てきている．高位合成中に簡易なフロアプランを実行したり，フロアプラン結果を高位合成にフィードバックするなど様々な方法が考えられる[17),18)]．

5.9 Engineering Change Order (ECO)に関して

ASIC設計の場合に高位合成した回路をECOするケースを考える．複雑なアルゴリズムを自動スケジューリングで最適化し，たくさんの変数や演算がレジスタや演算器をシェアしている場合に，機能をスペアセル等で修正することは非常に難しく，ECOは極めて困難となる．これは人手RTL設計でも同様である．タイミング関係のECOならば，修正は可能である．制御回路はチップ設計最終段階で仕様変更や修正が入ることがあり，機能修正が必要なECOの必要性が高い部分である．制御回路でもレジスタのシェアリングの程度が大きいとECOが困難になる．ECOが必要になる可能性のある部分はレジスタシェアリングを抑制しておけば，ECOは容易になる．

ECOを行うためには，高位合成回路のアーキテクチャ（本稿ではFMSD）をよく理解することと，元の動作記述（CやSystemC）とRTL，ゲート回路との関係を把握することが重要である．後者は，高位合成ツールの動作記述とRTL記述の変数間の対応関係表示機能を使うことでかなり対応できる．FPGAの場合は，ECOは必要ない．

5.10 低電力化

動作合成の適した分野として，クロックゲーティング（Clock Gating）技術がある．プログラム全体に対してCDFGを作成するため，動作言語では，一見分からないような隠れたクロックゲーティング条件を見つけることができる．簡単な例を示す．

```
x = a;
if (c1) y = x+1;
```

というような記述を合成する場合を考える．xは全ての条件でaの値を書き込まれている．しかし，条件c1をx = aより前に移動し（code motion），

```
if (c1) {
  x = a;
  y = x+1; }
```

とすれば，機能は変化ないが，変数xを格納するレジスタへの書込みを条件c1が真の時だけにできる．すなわち，c1が偽の場合は，レジスタ書込みを停止できる．このようなclock gating条件をcode motionすることなく，スケジューリング段階でのスペキュレーション探索で実行することで，より低電力な回路を合成することができる．

5.11 アーキテクチャ自動探索

図10，図11に示したように，高位合成では，資源制約や時間制約によって様々なアーキテクチャが合成できる．その他にも，ループや関数，配列等の実現方法で様々なアーキテクチャを合成できる．例えば，配列の実現方法として，1ポートメモリや2ポートメモリ更にはレジスタ群がある．レジスタ群で実現する場合は，デコーダを何個持つかによって同時にアクセス可能なレジスタ数を制御できる．初期化付き配列ならば組合せ回路で合成することもできる．ループは，順序回路，パイプライン，ループ展開等様々な方法で実現できる．関数もインライン展開，サブルーチンコール，関数自体をFSMDとして複数合成する方法等様々な方法がある．このように高位合成が出力可能な多数のアーキテクチャの中から，最適なアーキテクチャを発見することが，人手RTL設計を凌駕する回路を設計する鍵となる．しかしながら，どのような変換が目的に沿った回路を出すかは，ツールに習熟する必要があり，ツールの使い始めには容易ではない．そのため，様々な合成を多数自動実行し，意味のある合成結果だけをトレードオフカーブの形で提示するツールが提供されている[19]．非常に広い探索空間を効率的に探索する必要があり，今後更に研究を進める必要のある部分である．

5.12 FPGA向け動作合成

FPGAをASICのラピッドプロトタイプとして利用する場合は，ASIC用に高位合成したRTLをそのままFPGA用の論理合成ツールにかければよい．しかし，最終製品にFPGAを利用する場合は，FPGAデバイスに即したRTLを合成する必要がある[20]．FPGAはASICと異なり，計算資源（LUT(注23)等）と記憶資源（レジスタ，メモリ）の総量がデバイスごとに固定されている．そのため，ASIC合成と同様の回路を合成すると，信号処理系のパイプライン処理回路では計算資源を多く使用し，レジスタが余る傾向があり，逆に通信系の回路では，レジスタを多く使用し，計算資源が余る傾向がある．FPGA用のRTL記述は，このような計算資源と記憶資源の使用量を，対応デバイスに合わせて合成することが望ましい．例えば，信号処理系でレジスタが豊富に使える場合は，変数によるレジスタシェアリングを抑制することで，資源をバランス良く使うとともに，レジスタファンアウトを減少することによる遅延抑制効果も得られる(注24)．また，近年の高性能タイプのFPGAは，専用の積和演算（MAC）ブロックやメモリブロックを持っている．MACブロックを利用するとLUTに比べ高速に演算ができるし，メモリブロックを利用すると面積効率が良くなる．ただし，それらの専用ブロックは，配置位置が固定されているため，配線遅延が大きくなる傾向があり，クロック周波数を上げるためには，中継用のレジスタを2，3個挿入する必要がある．中継用レジスタを近傍に配置することで，配線周波数を向上することができる．これは，高位合成のスケジューリングで，専用ブロックへの読み書きに必要なサイクル数を増大することで実現できる．レジスタ挿入は，回路の実行サイクル数（レイテンシ）が増加するため，パイプライン回路のスループットには影響がないが，順序回路では注意が必要である．その他，FPGAはそのベンダやシリーズごとに，また，論理合成ツールごとに特性が大きく異なるため，それぞれに適したRTLを出力する必要がある．例えば，LUTの場合，4入力の場合と，8入力の場合は，セレクタと演算の融合のさせ方が大きく異なり(注25)，マッピングの遅延や面積が大きく異なるため，その差異を高位合成で扱う必要がある．しかし，FPGAメーカーの供給する論理合成ツールのバージョンごとの面積・遅延性能の変化も大きいため容易ではない．ASIC向けの場合は，ASICベンダが異なる場合でも，セルの遅延や面積の違いは高位合成用ライブラリで吸収でき，合成回路の面積，遅延の予測もFPGAに比べれば容易である．

5.13 動的再構成可能チップ向け動作合成

FPGAは，LUT等で構成されビット単位の論理演算を実行

（注23）Look-up Table:SRAMで論理関数を実現するセル．

（注24）FPGAではファンアウトが多い場合，ファンアウト先の全てのLUTやレジスタが近くに配置するのは難しい．

（注25）4入力の場合はセレクタと演算器が別々のLUTになるのに対し，8入力の場合はセレクタと演算が同じLUTにマッピングできることがある．

するのに向いており，細粒度アーキテクチャと呼ばれる．乗算等の一部算術回路は専用ブロックで対処している．一方，加算等の算術処理に重点を置き，LUTの代わりにALUを並べた書換え可能チップもあり，粗粒度アーキテクチャと呼ばれる（ALUのビット幅は，8，16，32 bit等いろいろなものがある）．このような再構成可能チップの中には実行時に動的に（クロック速度で）再構成可能（コンフィギュレーションを変更可能）になっているものもある．動的再構成チップは，人手RTL設計が非常に困難であり，高位合成をコンパイラとして使うことが必然である．コンパイラの戦略としては，例えば，スケジューリング結果の各状態を動的再構成チップのコンテキスト(注26)に当てはめ，クロックごとに状態遷移するのに合わせて，コンテキストを切り換えればよい[21]．データパスが可変なプロセッサと考えることができる．GPGPU(注27)等のSIMD(注28)系プロセッサは複数データ処理を並列化するだけであるが，動的再構成では高位合成ツールが高性能であれば，データ処理の並列化に加え，条件分岐やループ等の制御構造も並列化できるため，一般的なアルゴリズムの高速化に向いている．FPGAに比べた場合は，算術回路の実行はALUで実行できるため有利であるが，論理演算やビット処理はALUで行うと著しく不利となる．算術演算にはALU，論理演算にはLUTが向いているため，双方をあるバランスで保持したチップ形態に集約すると考えられる．これらに，更にCPUも搭載すると，組込み用汎用チップとなる．

5.14　高位合成回路の機能とタイミングの検証

高位合成されたRTL記述は最適化されており，合成されたRTLを検証するのは容易ではない．高位合成を利用する場合，機能の検証は動作記述のC言語やSystemCの環境で行えばよいが(注29)，動作記述がUntimedな場合，サイクルタイミングの検証は動作記述上では行えない．信号が一方向に流れる信号処理等の回路の場合は，通信プロトコルが単純なため，合成されたRTLのシミュレーションで入出力信号を観測するだけで，タイミング検証を行うことも難しくはない．しかし，相互に通信するReactiveな回路群を設計する場合は，入出力信号のチェックだけでは，サイクルタイミングの合わない原因を発見，修正することは困難である．入出力データによって，サイクル動作が変化するような回路では，設計者が記述した動作記述レベルでタイミング検証を行えることが重要である．近年，動作記述上で，合成RTLのタイミング検証を行うツールも提供されている[1]．これは，ソフトウェアの世界で，コンパイラが生成したアセンブラを検証した時代から，シンボリックソースコードデバッグが可能になったのと同様の進歩である．これらが進歩すれば，RTLはアセンブラ同様，単に中間フォーマットとしての存在となり，通常は設計者が見る必要はなくなると考えられる．

(注26) FPGAはコンフィギュレーションメモリを一つ持つが，動的再構成チップは，これを複数持ち，動的に切り換えることで回路構成を切り換えることができる．各コンフィギュレーションをコンテキストと呼ぶ．

(注27) General-purpose computing on graphics processing unit:GPUで汎用の数値計算を行うこと．

(注28) Single Instruction Multiple Data:一つの命令で複数データの演算処理を同時に行う設計方法．

(注29) 高位合成ツールのバグを検出するために，動作シミュレーション結果と合成RTLのシミュレーション結果の一致性を調べるなど，動作記述とRTL記述の一致性検証が別途必要になる．

```
int data[100],sum[50];
readsum(){
  int i;
// 100個のデータ格納
  for(i = 0; i < 100; i++)
    data[i] = input(k);
// 奇数番地と偶数番地を足して出力
  for(i = 0; i > 15; i++)
    sum[i]=data[i*2]+data[i*2+1];
}
```

(a)

```
int data1,data2,sum[50];
readsum(){
  int i;
  for(i = 0; i < 50; i++){
    data1 = input(k);  //奇数
    data2 = input(k);  //偶数
    sum[i]=data1+data2;
}
```

(b)

図13　ハードウェア向けのC記述　(a) ソフトウェア向け記述，(b) ハードウェア向け記述

6．ハードウェア向けC記述の特徴

ソフトウェアのプログラムには，同じ機能を実現するのに，高速な記述，実行形式のサイズの小さな記述がある．高位合成によるハードウェア合成の場合も同様に，動作C記述によって面積や速度が本質的に異なる．これは，合成ツールの最適化機能では埋め切れない本質的なものであり，理解して記述することが重要である．典型的な例を図13に示す．100個のデータを外部から読み込み，奇数番目と偶数番目の入力を足して，結果を配列sumに代入する機能を，二つの方法で記述した．図13 (a) は，ソフトウェアプログラムでよく使う書き方で，データを全て読み込んでから，データを処理し，出力している．PC等のアプリケーションプログラムでは，この記述で問題はないが，ハードウェアには向かない記述である．図13 (b) にハードウェア向け記述を示す．両者を比較すると，(a) は (b) に比べ配列を1個余分に使うことでメモリを余計に使い面積が大きく，ループの繰返し回数と配列へのアクセス回数が多くなるため，実行性能も劣る．PC向けアプリケーションではデータを読み込んだ上で処理するのは理解しやすく，普通のスタイルであるが，ハードウェア向けの動作記述では，入力したデータで計算できるものはすぐに計算し，データの保持期間をできるだけ短い期間とすることで，面積と実行サイクル数ともに小さくすることが可能となる．これが，ハードウェア向け動作記述の基本の考え方である．ソフトウェア向けC記述を高位合成を利用してハードウェア合成する場合は，この変換を行わないと，メモリ使用量が不必要に大きくなり，十分な性能を得られないことがある．

7．まとめ

近年実用化が進んでいる高位合成について，動作記述から様々なRTLを合成する原理と，実用化に必要な技術を概説した．動作合成のターゲットアーキテクチャであるFSMDを用

いて，CPUと高位合成回路をモデル化し，専用ハードゥエアがCPUより高速動作可能な原理を示した．

更に，高位合成が人手RTL設計に比べて適用可能な範囲が限られているということが必ずしも正しくないことを示した．特に，制御系回路は高位合成向けでないといわれることが多いが，ステートマシンを陽に設計するRTL設計に比べ，シーケンス制御できる高位合成（C設計）の優位性を示した．高位合成の効果に関して，RTL設計に比べ短い記述量で設計でき，シミュレーション速度が数十倍から数百倍高速であることから，設計期間や設計工数の短縮や信頼性向上に大きく寄与することを示した．ただし，それだけでなく，高位合成が多数の制約に基づいて様々なアーキテクチャを探索できるため，面積や電力の最適化も可能になることを実例も示しながら議論した．また，C記述の設計資産が，RTL記述の設計資産に比べ，性能や周波数，更に通信I/Fや機能の変更の容易さの観点から数段高い再利用性を持つことも説明した．

最後に，高位合成を実用化するための技術群について議論した．動作記述の最適化，アーキテクチャの自動探索，ECO，FPGA向け高位合成や動作記述（C記述）のチューニング方法等について説明した．C設計では高位合成だけでなく，いかにC記述を検証するかが重要であるが，本稿では合成面に焦点を当て検証の詳細は触れていない．

高位合成はLSIの自動設計ツールという意味では十分に成熟してきており，身の回りの多くの商用チップの設計に利用されている．携帯電話，ディジタルカメラ，プリンタ，セットトップボックスやテレビ等から衛星搭載装置や携帯基地局，メインフレーム計算機等の幅広い分野での実績がある．また，今後は，LSIの設計にとどまらず，FPGA等のリコンフィギュラブルなチップを利用した，汎用アクセラレータのコンパイラという側面が強くなってくると考えられる．従来，DSPやGPUが担っていた機能を，「リコンフィギュラブルチップ+C設計ツール」が担うという意味である．ARM等のCPUとFPGAが混載されたチップも利用可能になってきており，今後このような傾向が進むと思われる．証券取引等のビッグデータの初期データ処理や、センサとモータ制御といったM2M(注30)の世界でも上記チップの活用が進むと考えられるが，CPU向けに記述されたC / C++プログラムを完全自動で高速化するためには，まだまだたくさんの技術開発をしなければならない．また，現在の高位合成のより上位であるシステムレベルの合成・最適化機能の拡充も重要である．

(注30) Machine-to-Machine: 機械同士が人間を介さずに相互に情報交換し，自動的に最適な制御が行われるシステム．例えば，多数のセンサ情報を基に，モーター等の制御を自動最適化する．

文　献

(1) K.Wakabayashi and B.C.Schafer, "'All-in-C' behavioral synthesis and verification with cyberworkbench, from C to tape-out with no pain and a lot of gain," in High-Level Synthesis from Algorithm to Digital Circuit, P.Coussy and A.Morawiec eds., pp.113–127, Springer, 2008.
(2) SystemC, http://www.accellera.org/home/
(3) S.Gupta, N.Dutt, R.Gupta, and A.Nicolau, "SPARK: A high-level synthesis framework for applying parallelizing compiler transformations," Proc. 16th Int'l Conf. on VLSI Design, pp. 461–466, Jan. 2003.
(4) H.Kurokawa, H.Ikegami, M.Otsubo, K.Asao, K.Kirigaya, K.Misu, S.Takahashi, T.Kawatsu, K.Nitta, H.Ryu, K.Wakabayashi, M.Tomobe, W.Takahashi, A.Mukouyama, and T.Takenaka, "Study and analysis of system LSI design methodologies using C-based behavioral synthesis," IEICE Trans. Fundamentals, vol.E86-A, no.4, pp.787–798, April 2003.
(5) N.Kobayashi, K.Wakabayashi, N.Shinohara, H.Tanaka, and T.Kanoh, "Design experiences with high-level synthesis system cyber I and behavioral description language BDL," Proc. of APCHDL, pp.139–142, Oct. 1994.
(6) K.Wakabayashi, "Unified representation for speculative scheduling: generalized condition vector," IEICE Trans. Fundamentals, vol. E89-A, no.12, pp.3408–3415, Dec. 2006.
(7) B.C.Schäfer, Y.Iguchi, W.Takahashi, S.Nagatani, and K.Wakabayashi, "Fixed point data type modeling for high level synthesis," IEICE Trans. Electron., vol. 93-C, no. 3, pp. 361–368, March 2010.
(8) U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A.Rountev, and P.Sadayappan, "Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model," Int'l Conf. Compiler Construction, pp. 132–146, April 2008.
(9) J. Cong, W. Jiang, B. Liu, and Y. Zou, "Automatic memory partitioning and scheduling for throughput and power optimization," Proc. of ICCAD, pp.697–704, Nov. 2009.
(10) A.Kondratyev, L.Lavagno, M.Meyer, and Y.Watanabe, "Realistic performance-constrained pipelining in high-level synthesis," Proc. of DATE, pp.1–6, March 2011.
(11) 竹中　崇, 若林一敏, 中越優佳, "不完全ネストループに対するループパイプライン," 信学技報, VLD2011–126, pp. 37–42, March 2012.
(12) A. Morvan, S. Derrien, and P. Quinton, "Efficient nested loop pipelining in high level synthesis using polyhedral bubble insertion," Proc. of Int'l Conf. on Field-Programmable Technology, pp.1–10, Dec. 2011.
(13) 加藤勇太, 瀬戸謙修, 丸泉琢也, "外側ループシフトを使用した高位合成向け自動ループ併合," 信学技報, VLD2011-69, DC2011-45, pp.103–108, Nov. 2011.
(14) G.Ferizis and H.A.ElGindy, "Mapping recursive functions to reconfigurable hardware," Int'l Conf. on Field Programmable Logic and Applications, pp.1–6, Aug. 2006.
(15) D.R.Ghica, A.Smith, and S.Singh, "Geometry of synthesis IV: compiling affine recursion into static hardware," Proc. ACM SIGPLAN Int'l Conf. on Functional Programming, pp.221–233, Sept. 2011.
(16) I.Skliarova and V.Sklyarov, "Recursion in reconfigurable computing: A survey of implementation approaches," Int'l Conf. on Field Programmable Logic and Applications, pp. 224–229, Sept. 2009.
(17) Z.Gu, J.Wang, R.Dick, and H.Zhou, "Unified incremental physical-level and high-level synthesis," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.26, no.9, pp. 1576–1588, Sept. 2007.
(18) L.Zhong and N.K.Jha, "Interconnect-aware low-power high-level synthesis," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.24, no.3, pp.336–351, March 2005.
(19) B.C.Schäfer and K. Wakabayashi "Design space exploration acceleration through operation clustering," IEEE Trans. Comput. - Aided Des. Integr. Circuits Syst., vol.29, no.1, pp.153–157, Jan. 2010.

(20) A.Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J.H. Anderson, S. Brown, and T. Czajkowski, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," Int'l Symp. on Field Programmable Gate Arrays, pp. 33–36, Feb. 2011.
(21) T.Toi, T.Okamoto, T.Awashima, K.Wakabayashi, and H.Amano, "Iterative synthesis methods estimating programmable-wire congestion in a dynamically reconfigurable processor," IEICE Trans. Fundamentals, vol.E94-A, no.12, pp.2619–2627, Dec. 2011.
(VLD研究会提案，平成24年3月28日　受付)

若林一敏 (正員)

1984東大・工，1986同大学院修士課程了，2006同大学院博士課程了，博士(工学)．1993スタンフォード客員研究員．1986日本電気株式会社入社．以来，LSIの設計方法自動化の研究に従事し，現在開発したEDAシステムの事業化を行っている．現在，組込みシステムソリューション(事)シニアエキスパート，システムIPコア研究所主幹研究員．北陸先端大客員教授．平23喜安記念業績賞，平20年度本会基礎・境界ソサイエティ功労賞，平16山崎貞一賞，平12坂井記念特別賞，88学術奨励賞，2003，2007LSIオブ・ザ・イヤーでグランプリ等受賞．本会VLD委員長，情処SLDM主査，ASPDAC General Chair，CODES-ISSS TPC chair，回路とシステムワークショップ委員長等を歴任．著書「High Level VLSI Synthesis」(共著)，STARC寄付講座教科書等．C設計のエバンジェリストを自任．