解　説

論理とその VLSI 設計への応用†

藤田昌宏†† 川戸信明††

1. はじめに

VLSI 設計における誤り，特に，論理設計段階の誤りは，設計・製造期間の増大を招き，莫大な損害をもたらす可能性が大きい．また，人工衛星や機密業務に用いられるようなチップは，価格よりも信頼性が重要視される．このように高信頼性が要求されるようになると，自動合成や検証を行う設計手法が必要となるが，そのためには，まず設計すべきチップの仕様や動作を厳密に記述しておかなければならない．この，だれが見ても誤解のない記述法として，論理を用いることが考えられる．本稿では，さまざまな論理を VLSI を含めたハードウェア記述に応用することに関して，現在各地で行われている研究を解説する．

VLSI の仕様が論理で厳密に定義してあれば，行った設計の検証や，自動合成を計算機支援で行う道が開ける．自動合成は，現状では，大きなものを扱うことは難しく，また，人手設計との品質比較の上でも問題がある．しかし，合成に関する研究は，現在非常に活発化しており，今後が十分期待できる[1),28)]．

一方，検証についても，完全自動化は無理である．したがって，人手が介入することになるが，その際，ある程度使用する論理を知り，かつ，検証自体にも慣れないと検証できないので，専門家以外は難しく，したがって検証コストも高くなる．しかし，逆に専門家が使うと仮定すれば，検証技術自体は実用化レベルに達しており，すでに検証を専門に引き受ける会社も登場している．政府機密に関するシステムに用いるチップのように誤りの許されないものは，少々コストが高くついても，検証を行わなければならなくなっている[2)]．

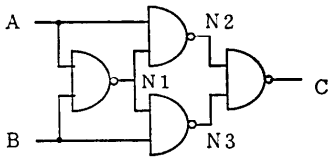
論理といっても，命題論理，1階の述語論理，高階の述語論理，時間に関する拡張を行った論理など，さまざまなものの応用が研究されている．拡張された論理を使用する理由は，並列動作するハードウェアを正確に分かりやすく書きたいことや，加算器のように似たパターンの繰り返しの構造をしているものをコンパクトに表現したいことなどによる．一般に，論理を拡張すればするほど，その計算機での取り扱いは難しくなる．反面，拡張された論理のほうが，VLSI を正確にしかも簡潔に記述しやすい．これは，検証などの支援を行うときに，完全自動をめざすか，人手の介入を認めるかとも絡んでいる．検証についていうと，現状では，VLSI の中の，資源の排他制御のような誤設計を生じやすい部分を自動検証し，後はシミュレーションのみとするか，反対に人手介入を行いながら VLSI の全体を検証していくかのどちらかが現実的である．

本稿では，さまざまな論理の応用の典型例を順に示す．ここでは，VLSI を記述するレベルとして，レジスタ・トランスファ・レベルからゲート回路，あるいは，CMOS のスイッチ・レベルを考える．したがって，合成・検証もこれらのレベル間のものとする．本稿を研究の現状理解に役立てていただければ幸いである．

2. 古典論理による記述とその支援

本章では，まず，最も単純な論理の応用として，通常の論理（ここでは，古典論理と呼ぶ）について考える．記述すべき VLSI を組合せ回路とフリップフロップに分けて考えると，組合せ回路の部分はそのまま命題論理式で表現できる．たとえば，図-1(a)の回路は，(b)の式で記述できる．これは，ネット名を中間変数として表現したものである．最近，活発に研究されている論理回路簡単化は，このように組合せ回路を命題論理式で表現している[3),4)]．ハードウェアは並列動作するが，論理式で表現されたものはつねに成立するので，並列動作はそのまま表現されることになる．しかし，この表現は AND ゲートや OR ゲートなどモジュールを意識しにくい表現となっており，た

† Application of Logic to VLSI Design by Masahiro FUJITA and Nobuaki KAWATO (Fujitsu Laboratories Ltd.).
†† 富士通研究所



(a) Exclusive-OR 回路

N1=～(A∧B)∧
N2=～(A∧N1)∧
N3=～(B∧N1)∧
C =～(N2∧N3)
(b) 論理での表現1

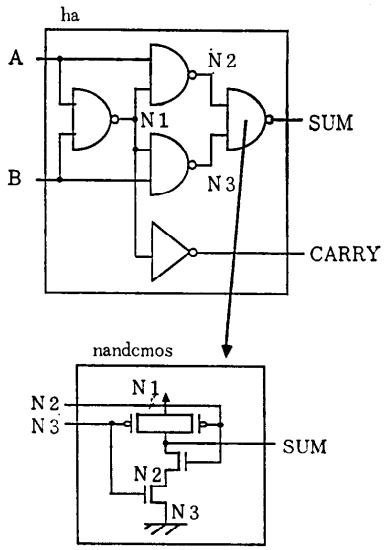
∃N1, N2, N3.
nand (A, B, N1)∧
nand (A, N1, N2)∧
nand (B, N1, N3)∧
nand (N2, N3, C)
ただし, nand (X, Y, Z)≡Z=～(X∧Y)
(c) 論理での表現2

図-1 組合せ回路とその論理による表現

とえば，階層設計を表現するのは，必ずしも容易ではない．これに対し，各ハードウェア・モジュールを一つの述語に対応させ，モジュール名を述語名，各引数を入出力変数とし，各モジュール間の接続を存在記号(∃)の付いた変数で表現すると，モジュール構造がそのまま表現できる (**図-1**(c))．このようにすれば，**図-2**(a)に示す半加算器の階層構造も(b)のように記述できる．

制御部は以上のように命題論理の範囲で十分記述できるが，ALU やデコーダなどのデータパスを命題論理で記述しようとすると，記述量が膨大になるだけでなく，記述した結果も分かりにくい．たとえば，**図-3**(a)は命題論理で記述しようとすると(b)のようになるが，1階の述語論理で記述すれば，(c)のように簡潔になる．さらに，より高位なレベルで VLSI を記述する場合には，0，1の2値だけでなく，整数値などを利用したくなり，命題論理では無理になる．ただし，命題論理の範囲では，原理的に計算機による自動検証が可能であるが，述語論理になるといつも自動検証できるとは限らない．述語論理による記述については，**4**章も参照されたい．

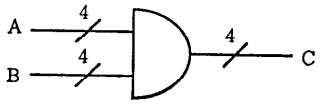
一方，フリップフロップは，同じ端子でも時刻とともに値が変化する可能性があるため，そのままでは，記述できず，なんらかの工夫が必要である．最も単純な工夫は時刻が変化するたびに変数を追加していく方法である．つまり，同じ端子であっても，時刻が異なれば異なる変数を割り当てる．たとえば，**図-4**(a)に示すDフリップフロップは，(b)のように記述する．



(a) 半加算器の階層構造

ha (A, B, SUM, CARRY)≡
∃N1, N2, N3.
nandcmos (A, B, N1)∧
nandcmos (A, N1, N2)∧
nandcmos (B, N1, N3)∧
nandcmos (N2, N3, SUM)∧
invcmos (N1, CARRY).
nandcmos (A, B, C)≡
∃N1, N2, N3.
pwr (N1)∧ptr (A, N1, C)∧
ptr (B, N1, C)∧ntr (B, C, N2)∧
ntr (A, N2, N3)∧gnd (N3).
pwr (A)≡A=1
gnd (A)≡A=0
ptr (G, S, D)≡～G→(S=D)
ntr (G, S, D)≡G→(S=D)
(b) 論理での表現

図-2 階層的な回路

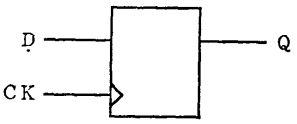


(a) データパス系の回路

(c[0]=a[0]∧b[0])∧
(c[1]=a[1]∧b[1])∧
(c[2]=a[2]∧b[2])∧
(c[3]=a[3]∧b[3])
(b) 命題論理での表現

$0 \leq \forall i \leq 3.$ c[i]=a[i]∧b[i]
(c) 述語論理での表現

図-3 述語論理での表現



(a) Dフリップフロップ

$(CK \rightarrow Qn=D) \wedge (\sim CK \rightarrow Qn=Q)$

(b) 論理での表現1

$(CK(t) \rightarrow Q(t+1)=D(t)) \wedge (\sim CK(t) \rightarrow Q(t+1)=Q(t))$

(c) 論理での表現2

図-4 順序回路の論理での表現

```
ha (A, C, SUM, CARRY):-
    nand 2 (A, B, N 1),
    nand 2 (A, N 1, N 2),
    nand 2 (B, N 1, N 3),
    nand 2 (N 2, N 3, SUM),
    inv (N 1, CARRY).
```

(a) Prolog による半加算器の記述

```
nand 2 (I 1, 0, 1).
nand 2 (0, I 2, 1).
nand 2 (1, 1, 0).
inv (0, 1).
inv (1, 0).
```

(b) 各素子の記述

```
nand 2 (I 1, 1, [inv, I 1]).
nand 2 (1, I 2, [inv, I 2]).
nand 2 (I 1, [inv, I 1], 1).
nand 2 ([inv, I 2], I 2, 1).
nand 2 (I, I, [inv, I]).
nand 2 (I 1, I 2, [inv, [and, I 1, I 2]]).
inv (I 1, [inv, I 1]).
inv ([inv, I 2], I 2).
```

(c) 記号シミュレーションを行うための定義の拡張例

図-5 Prolog による組合せ回路の記述

もう一つの方法は，時刻を表す特別な変数 t を導入し，各変数の値を t の関数として表現する方法であり，図-4(c)のように記述する．このように工夫をすることで，一応順序回路も表現できる．しかし，信号値としては2値であったとしても，時刻を陽に指定して順序回路を記述する場合には，命題論理の枠内では記述できない．3章で示す時相論理は，この工夫を論理の中に取り入れたものであり，信号値が2値であれば，命題論理の範囲で記述できるようにしたものであるといえる．

さて，図-2 に示したような各モジュールを一つの述語に対応させる記述法は，そのまま論理型言語 Prolog による記述に応用できる．図-2(b)に対応する Prolog の記述を図-5(a)に示す．ここでは，文献5) のシンタックスを使用する．また，nand ゲートとインバータの定義を(b)に示す．これは，取り扱う値

```
| ?- ha(1, 0, SUM, CARRY).
SUM=1
CARRY=0
yes
| ?- ha(1, 0, 0, 1).
no
| ?- ha(A, B, 0, 1).
A=1
B=1
yes
| ?- ha(A, B, 1, 0), write((A, B)), write(' : '), fail.
1, 0 : 0, 1
no
```

(a) 実行例

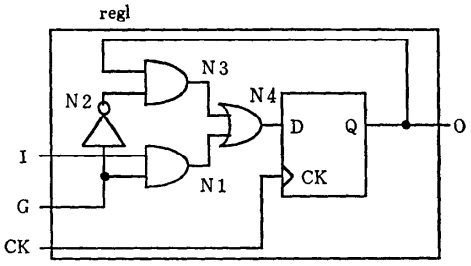
```
| ?- ha(a, 0, SUM, CARRY).
SUM=[inv, [inv, a]]
CARRY=0
yes
| ?- ha(a, b, SUM, CARRY).
SUM=[inv, [and, [inv, [and, a, [inv, [and, a, b] ] ] ],
                [inv, [and, b, [inv, [and, a, b] ] ] ] ] ]
CARRY=[inv, [inv, [and, a, b] ] ]
yes
```

(b) 記号シミュレーションの実行例

図-6 図-5 のプログラムの実行例

を0，1の二つの定数に限定した場合であるが，他の値，たとえばハイ・インピーダンス値を含めることも容易である．このように定義が与えられていると，Prolog の処理系を用いて直接シミュレーションを実行することができる．これは，通常のシミュレーションとは異なり，ユニフィケーションにより指定された入出力値を回路が満たすか否かを調べるため，図-6(a)に示すようにさまざまな性質を質問できる．さらに，各基本ゲートの定義を拡張することにより，記号値を取り扱うようにすることもできる．記号値を取り扱うための nand ゲートの定義を図-5(c)に示す．ここでは，リスト構造で論理式を表現する．この定義を用いることにより，図-6(b)に示すように，記号シミュレーションを実行することができる．ただし，実用的な記号シミュレータとするためには，計算途中での論理式簡単化が不可欠であり，そのためのプログラムを挿入する必要がある．

以上のように組合せ回路は容易に Prolog で記述・推論することができた．しかし，順序回路の記述には，上で述べたように工夫が必要である．一つは時間が扱えるように拡張された Prolog を用いることで，3章で説明する．もう一つの方法は，フリップフロップの記述に，引数として，現在と次の内部状態を付け



(a) 1ビットレジスタ

```
reg1([I,G,CK],[Q],STATE,NEXTSTATE):-
  and2([I,G],[N1]),
  inv([G],[N2]),
  and2([Q,N2],[N3]),
  or2([N3,N1],[N4]),
  dff([N4,CK],[Q],STATE,NEXTSTATE).

dff([D,0],[Q],Q,Q).
dff([D,1],[Q],Q,D).

and2([1,1],[1]).
and2([0,I2],[0]).
and2([I1,0],[0]).
or2([0,0],[0]).
or2([1,I2],[1]).
or2([I1,1],[1]).
inv([1],[0]).
inv([0],[1]).
```

(b) Prolog による記述

```
sim_forward([],[],_).
sim_forward([PresentIn | RestIn],[PresentOut | RestOut],State):-
   reg1(PresentIn,PresentOut,State,NextState),
   write((PresentIn,PresentOut,State,NextState)),
   nl,
   sim_forward(RestIn,RestOut,NextState).
```

(c) 時間軸に対して順方向にシミュレーションを行うプログラム

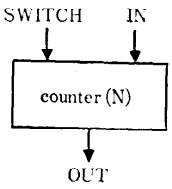
```
sim_backward([],[],_).
sim_backward([PresentIn | RestIn],[PresentOut | RestOut],State):-
   reg1(PresentIn,PresentOut,PreviousState,State),
   write((PresentIn,PresentOut,PreviousState,State)),
   nl,
   sim_backward(RestIn,RestOut,PreviousState).
```

(d) 時間軸に対して逆方向にシミュレーションを行うプログラム

図-7 Prolog による順序回路の記述

加える方法である．例として，**図-7**(a)に示す1ビット・レジスタを考える．これは，クロック CK が1で，かつ，ゲートGも1のときには，入力 IN を内部に取り込み，それ以外のときは，現在の値を保つものである．これは，(b)のように Prolog で記述できる．これで，現在時刻のみのシミュレーションは実行できるが，ある時間シーケンスのシミュレーションは，回路の定義を(c)のように再帰的に呼ぶことで処理する．(c)中では，各変数値の時間シーケンスをリスト構造で表現している．さらに，(b)の定義は，時間の進む方向にも，遡る向きにも利用できるため，(c)を(d)に書き直すと，時間を遡る向きにシミュレーションを実行させることもでき，検証などを行うときに有用である[6]．

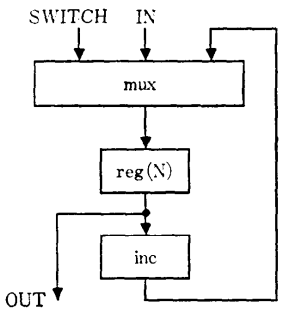
述語論理を用いたハードウェア記述・支援システムは，定理証明システムとともに，各種開発されてきた[7],[8]．ここでは，その中の一つとして，Gordon らの LCF-LSM[9] を取りあげる．これは，定理証明支援



(a) カウンタ

```
count(N)=dev{SWITCH, IN, OUT}, {OUT=N};
        count(SWITCH->IN|N+1).
```

(b) LCF-LSM によるカウンタの記述



(c) カウンタの実装

```
mux=dev{SWITCH, IN, N1, N2},
        {N1=(SWITCH->IN|N2}; mux.
reg(N)=dev{N1, OUT}, {OUT=N}; reg(N1).
inc=dev{OUT, N2}, {N2=OUT+1}; inc.
count_imp(N)=mux∧reg(N)∧inc.
```

(d) (c)に対する LCF-LSM の記述

図-8 LCF-LSM によるカウンタの記述と検証[9]

システム LCF[10] の上に構築されており，CCS[11] に基づく状態遷移表現によって順序回路を記述し，ユーザが対話的に検証していくシステムである．ここでは，簡単な例で概要を述べる．図-8(a)に示すカウンタを LCF-LSM では，(b)のように記述する．count がモジュール名であり，N は count の内部状態を表す変数（整数値をとる）である．dev で囲まれた変数は入出力変数を表し，論理的には，存在記号（∃）に対応する．次の $OUT=N$ が出力値を決める式であり，最後の count の後の括弧の中は次の内部状態を計算する式である．もし入力変数 SWITCH が1なら，次の内部状態は入力変数 IN と同じ値となり，もし0なら（現在の内部状態+1）となる（このカウンタは永久にカウントアップし続ける）．さて，このカウンタを図-8(c)のようにマルチプレクサ，レジスタ，インクリメンタを用いて実装するとすると，これは，(d)のように記述される．LCF-LSM では，(d)が(b)と同じ動作をすることの検証を定理証明として行うことを支援する．具体的には，(d)の式を(b)の式に変形する際の途中の計算式を記憶し，ユーザの指定により，式の簡単化や変形規則の適用を行う．LCF-LSM システムを用いて，16ビット・プロセッサや，リングバス用のインタフェース・チップが検証されており，計算時間も十分対話的に使える程度に収まっている．現在，実用化に向けて，企業内でも使用されようとしている[2]．

LCF-LSM と同じような思想であるが，処理形に Prolog を用いて，できるだけ自動化したシステムとして，Barrow の VERIFY[12] がある．これは，階層・構造化設計されたものを対象とし，各設計階層間の同一性を検証していく．LCF-LSM で検証された程度の規模の例題を，1 MIPS 程度の計算機で数分～数十分の CPU 時間で自動検証できている．

また，定理証明法を応用した，自動合成についても研究されている．これは，定理証明で使った式変形をそのまま合成に使用しようとするものであり，現在のところ，組合せ回路を対象としている[13]．

3. 時相論理 (Temporal Logic) による記述とその支援

2章で述べたように，古典論理そのままでは，順序回路を表現できないため，なんらかの工夫を行う必要があった．この工夫を論理体系の中に取り込んだ論理が時相論理 (Temporal Logic) である．古典論理で順序回路を表現できないのは，各変数は1度値が決まると後で変更することができないからである．これに対し，時相論理では，各変数は時刻が異なれば，異なる値をとってもよい．したがって，時刻とともに状態を変化させるものを表現できる．各変数が時刻とともに値を変化させるとき，その変化の仕方を指定するのが時相演算子 (Temporal Operator) である．つまり，「時相論理＝古典論理＋時相演算子」と考えてよい．実は，時相論理にもさまざまな種類があり，それぞれハードウェア記述への応用研究がされている[14),15),17),24)]．ここでは，代表的なものとして，まず，Linear Time Temporal Logic (LTTL と略す)[14] と，次に Interval Temporal Logic (ITL と略す)[15] について述べる．

LTTL には，次の4つの時相演算子がある．

□P (always)：現在から先ずっとPが成り立つ

◇P (sometime)：現在から考えていつかPが成り立つ

○P (next)：次の時刻でPが成り立つ

Pならば次の時刻にQである	P→○Q
Pならばいつかは Qであるということがつねにいえる	□(P→◇Q)
Qより先にPが真になる	～(～P U Q)
Pが立ち上がると次の時刻にQも立ち上がる	(～P∧○P)→○(～Q∧○Q)
Pならば，Qが立ち上がるまでRである	P→(R U (～Q∧○Q))

図-9 時相論理による時間順序関係の記述

(1) □P≡P∧○□P
(2) ◇P≡P∨(～P∧○◇P)
(3) P U Q≡Q∨(P∧～Q∧○(P U Q))

図-10 時相演算子の性質

P U Q (until): 現在から考えて，Qが成り立つまでPが成り立つ

これらの演算子を用いると，図-9 に示すようにタイムチャートなどで表現される時間順序関係を表現できる．

さて，命題論理の範囲の LTTL には，以下に示すように，決定手続きが存在し，これを利用すると，自動検証や，LTTL の式から状態遷移図の自動合成などを行うことができる[16)~18)]．

時相演算子には，図-10 に示すような性質がある[16)]．これは，時相論理の公理から得られるものであり，たとえば，(1)は，「いつもPであるということは，現在Pであり，次の時刻にもいつもPであるということがいえる」ということを表している．図-10 の性質を利用すると，任意の時相論理式は現在に対する条件と，次の時刻に対する条件に分けられることが分かる．これは，とりもなおさず，任意の時相論理式は状態遷移表現に展開できることを意味する．このように展開できれば，任意の時相論理式が充足可能であるか否かは，できた状態遷移表現上に無限の遷移シーケンスがあるか否かに対応する[16)]．たとえば，図-11(a)の式は(b)のような状態遷移に展開できるため，無限の遷移シーケンスをもち，したがって，充足可能である．よって，時相論理で記述されたハードウェアの検証ができるとともに，状態遷移図の自動合成も可能であることが分かる．検証時間も制御回路に絞れば，1 MIPS 程度の計算機で数秒～数分の CPU 時間に収まっている[23),27)]．これら，検証・合成については，文献 6), 18) を参照されたい．

以上は，命題論理の範囲であるが，もちろん述語論理の範囲で記述することもできる．ここでは，述語論理の範囲の ITL に基づく言語として，Moszkowski の Tempura[19)] と Tokio[20),21)] を紹介する．まず，ITL[15)] について述べる．

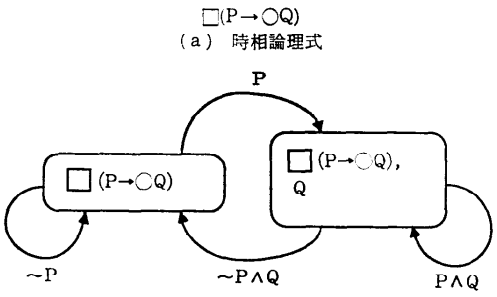


図-11 時相論理式の状態遷移表現への展開

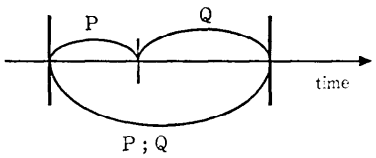


図-12 chop 演算子: P; Q

ITL は，古典論理に二つの時相演算子，';' (chop), と ○ (next) を加えたものである．LTTL が時刻ごとに記述していたのに対し，ITL はインターバルと呼ばれる時区間ごとに記述する．○は LTTL のそれとほぼ同じ意味である．また，P; Q は図-12 に示すように，現在考えているインターバルを前半と後半の二つに分け，前半でPが成立し，後半でQが成立することを意味する．ITL では，積 (∧) で並列実行を表現し，；で逐次動作を表現する．○と ；演算子により，LTTL の各演算子を含め，beg, fin, ←, len, empty, keep などさまざまな記述に便利な演算子を定義できる (定義の仕方については，文献 13) を参照されたい)．ITL では，システムの動作は，インターバルごとに記述していく．インターバルは状態を有限個集めたものなので，通常の状態遷移表現よりも，より柔軟な記述ができる．beg, fin はそれぞれ，インターバルの最初と最後の時刻を示す．また，← は temporal assignment と呼ばれるものであり，次のように定義される．

$$A \leftarrow B \equiv \forall c.\ \mathrm{beg}(B=c) \rightarrow \mathrm{fin}(A=c)$$

ただし，c は定数 (時刻が変わっても値は変化しない)

つまり，Bのインターバルの最初の時刻での値がAのインターバルの最後の時刻での値になるという，レジスタ・トランスファに対応している．したがって，← と ; には次の性質がある．

$((B \leftarrow A);(C \leftarrow B)) \rightarrow (C \leftarrow A)$

len は，インターバルの長さ（そのインターバルに含まれる状態の数−1）を示すものであり，これを用いると次のようにディレイ素子を表現できる．

delay $(n, \mathrm{OUT}, \mathrm{IN}) \equiv \Box(\mathrm{len}=n \rightarrow (\mathrm{OUT} \leftarrow \mathrm{IN}))$.

これは，n ユニットタイムのディレイであり，任意のインターバルについて，もしその長さが n ならば，入力値が出力に送られる．また，keep (A) は，A が現在のインターバルの最後の時刻以外のすべての時刻で成り立つことを示す（図-14(a)で使用する）．

このように，ITL には；演算子があるため，通常のレジスタ・トランスファ・レベルのハードウェア記述言語と同じような記述も行うことができるため使いやすい．現在，ITL のサブセットに基づく言語として，Tempura と Tokio があり，それぞれ，インタプリタやコンパイラが作成されている[19),22)]．Tokio は Prolog の拡張として定義されているため，ユニフィケーションや自動バックトラックなど Tempura にない機能も備わっている．

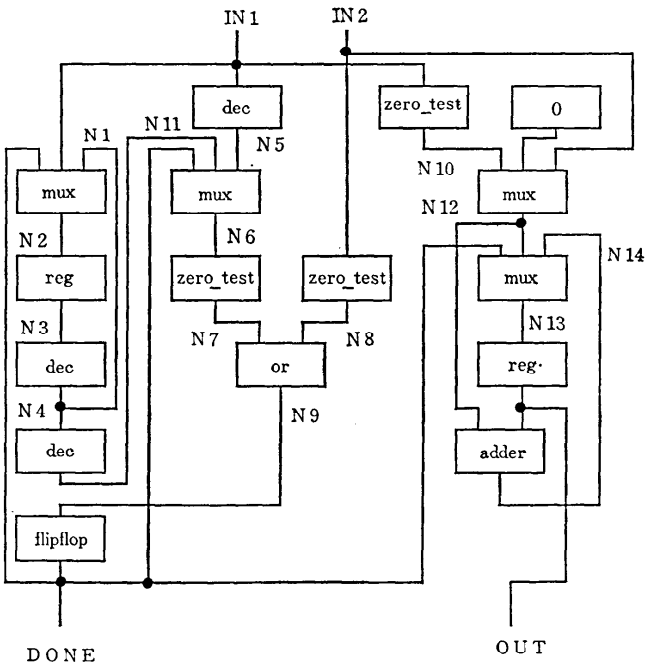
これらの言語では，各変数は時刻とともに値を変化してよいため，図-7 の 1 ビット・レジスタも図-13 のようにそのまま記述できる．さらに，手続的な記述も宣言的な記述も行え，たとえば，図-14(a)に示す掛算器の手続的な記述を(b)の回路のように実装するとした，(c)のような宣言的な記述もできる．このように，ハードウェアの動作も構造も同じ論理の上で表現できる．もちろん，Tempura や Tokio は，ITL の任意の式が実行できるわけではないが，ハードウェア

```
define regl (I, G, CK, O)
    =if (G=1 and CK=1) then ((next O)=I)
                       else ((next O)=O).
```

図-13 Tempura による 1 ビットレジスタの記述

```
mult (IN 1, IN 2, DONE, OUT)←
   if (IN 1=0) then (OUT ⟨- 0, DONE ⟨- 1)
            else (Ans ⟨- IN 2, IN 11 ⟨- IN 1-1, IN 2 ⟨- IN 2, keep (DONE=0)
              && mult 1 (Ans, IN 11, IN 2, DONE, OUT)).
mult 1 (Ans, IN 11, IN 2, DONE, OUT)←
    if (IN 11=0) then (OUT ⟨- Ans, DONE ⟨- 1)
              else (Ans ⟨- Ans+IN 2, IN 11 ⟨- IN 11-1, IN 2 ⟨- IN 2, keep (DONE=0)
                && mult 1 (Ans, IN 11, IN 2, DONE, OUT)).
```

(a) Tokio による掛算器の手続型の記述



(b) 実 装 例

```
mult_imp(IN1,IN2,DONE,OUT):-
      mux(DONE,IN1,N1,N2),
      reg(N2,N3),
      dec(N3,N4),
      dec(N4,N15),
      dec(IN1,N5),
      mux(DONE,N11,N5),
      zero_test(N6,N7),
      zero_test(IN2,N8),
      or2(N7,N8,N9),
      zero_test(IN1,N10),
      mux(N10,0,IN2,N12),
      mux(DONE,N12,N14,N13),
      reg(N13,OUT),
      adder(N12,OUT,N14).

mux(S,IN1,IN2,OUT):-
      if (S=0) then (OUT=IN1) else (OUT=IN2).
reg(IN,OUT):-
      @ OUT = IN.  % @: next operator
dec(IN,OUT):-
      if (IN=0) then (OUT=0) else (OUT is IN - 1).
zero_test(IN,OUT):-
      if (IN=0) then (OUT=1) else (OUT=0).
or2(IN1,IN2,OUT):-
      OUT is IN1 \/ IN2.
adder(IN1,IN2,OUT):-
      OUT is IN1 + IN2.
```

(c) (b)に対する Tokio の記述

図-14 Tokio による掛算器の記述

記述言語としては十分な能力を備えている.

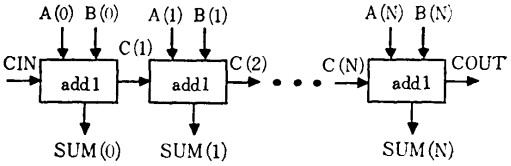
以上のほかにも時相論理は各種提案されている. 詳細は文献 23), 24) を参照されたい.

4. 高階論理 (Higher Order Logic) による記述とその支援

2章で1階の述語論理によるハードウェア記述について触れたが, 本章では, 高階の述語論理によるハードウェア記述とその検証について述べる. さて, ハードウェアの構造と動作を記述するには, 各モジュールを述語に対応させるのがよかった. しかし, 1階の述語論理では, 図-15(a)に示すように, 「1ビットの加算器を n 個直列に接続して, n ビットの加算器を作る」ということは, 述語に束縛記号を付ける必要があるため表現できない. さらに, このようにしてできた加算器が確かに n ビット加算器であるということを, 帰納的に証明したいが, それもできない. 以上のことを論理体系の中で行うには, より高階の論理を使えばよい. 高階論理 (Higher Order Logic) は, 1階の述語論理と比べて, 以下のような利点がある.

(1) 関数や述語に束縛記号を付けられるので, 帰納法を直接表現できる.

(2) 関数や述語が他の関数や述語の引数になれるので, たとえば, 下のように繰り返し表現ができる.



(a) n ビット加算器

```
adder(N)(A, B, CIN, SUM, COUT)≡
∃C. (CIN=C(0))∧
    iterate (0, n) (λI.
    add 1 (A(I), B(I), C(I), SUM(I), C(I+1)))∧
      C(N+1)=COUT).
```

(b) 高階論理による記述

図-15 n ビット加算器の高階論理による記述

interate $(m, n, f)=f(m)\wedge f(m+1)\wedge\cdots\wedge f(n)$

ただし, m, n は整数, f は任意の関数

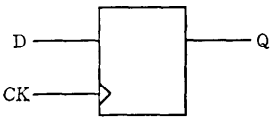
(3) λ 抽象ができる.

先ほどの加算器は図-15(b)のように高階論理で記述できる. この式に対し, 帰納法を適用することにより, 確かに n ビットの加算器になっていることを証明できる.

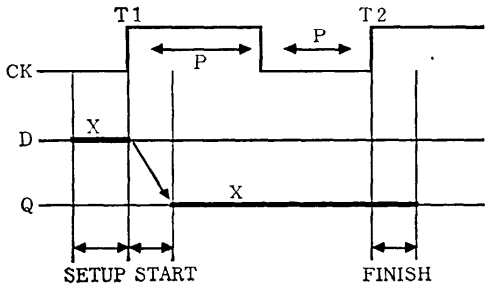
高階論理が有用であると考えられるもう一つの理由は, 前章の時相演算子に相当するものを高階論理の中で定義できるということである. 今, t を時刻 (整数とする), f を述語, x を定数とすると, 図-16 のようにさまざまな時間に関する性質を高階論理で表現で

① fは時刻T1から時刻T2まで値Xで固定:
stb(f, X, T1, T2)≡∀T. (T1≦T∧T≦T2)→f(T)=X.
② 時刻T1より後，最初にfが成立するのは時刻T2である:
next (T1, T2)f≡T1≦T2∧f(T2)∧
∀T. (T1<T∧T<T2)→∼f(T).
③ 時刻Tでfが立ち上がる:
rise Tf≡∼f(T−1)∧f(T).
④ 時刻Tでfは立ち上がるか立ち下がる:
edge Tf≡(∼f(T−1)∧f(T))∨(f(T−1)∧∼f(T)).
⑤ fの二つのエッジ (edge) の間はN時間以上ある:
period f N≡∀T1, T2. (edge f T1)∧
next (T1, T2)(edge f)→(T2−T1>N).

図-16 高階論理による時間に関する述語の定義

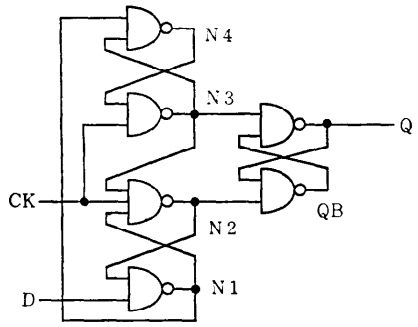


(a) エッジ・トリガ・Dフリップフロップ



DTYPE (SETUP, START, FINISH, P) (D, CK, Q)≡
∀T1, T2. ((rise CK T)∧
(next (T1, T2) (rise ck))∧
(period CK P))→
(∀X. stb (D, X, T1−SETUP, T1)→
stb (Q, X, T1+START, T2+FINISH)).

(b) タイムチャートと仕様



(c) nand ゲートによる実装

nand (I1, I2, O)≡∀T. O(T+1)=∼(I1(T)∧I2(T)).

(d) nand ゲートの定義

図-17 エッジ・トリガ・Dフリップフロップの記述

きる．言い替えると，3章の時相論理は時間に関する高階の述語を時相演算子として取り込んだ論理であると考えられる．

例として，図-17(a)のような仕様のエッジ・トリガのDフリップフロップを考える．この仕様は高階論理で，(b)のように記述できる．今，(c)のようにnand ゲートを用いて実装し，各 nand ゲートは(d)のような仕様を満たしてるとすると，次のことを証明することができ，(c)の回路は確かにエッジ・トリガのDフリップフロップになっていることが分かる．

∀d, ck, q.

d-imp (d, ck, q)→d type (2, 4, 1, 3) (d, ck, q)

以上のことを最初に提案したのは，Hanna[25]であり，現在支援ツールを作成中である．また，Gordon[26]も2章の LCF-LSM を発展させ，LCF 上に高階論理の検証支援ツールを作成している．いずれも実用化は近い．

5. おわりに

以上，論理をハードウェア記述に応用する研究について紹介した．さまざまな論理が使われているが，研究の方向としては，2章のもの⇨3章のもの⇨4章のものと進んでいる．特に，4章のアプローチは，システムをソフトウェアも含めた構造・動作などのすべての面を統一的に記述する方向を示しており，今後の発展が期待される．

支援ツールの研究も活発であり，特に，検証に関しては，ユーザをある程度教育することにより，十分実用チップを取り扱えるレベルに達している．システムが大規模になれば，仕様を厳密に記述しておく必要があることは，言うまでもない．しかし，その実行は容易ではない．問題は，論理に対して拒絶反応を示す設計者が多いことであろう．この点は，とにかく慣れることが必要である．本稿がその手助けになれば幸いである．

参考文献

1) 笹尾：VLSI 化回路設計方式，小特集：VLSI 設計の新しい流れ，情報処理，Vol. 28, No. 5 (1987).
2) Gordon, M. J. C. and Herbert, J.: Formal Hardware Verification Methodology and Its Application to a Network Interface Chip, IEE Proceeding-E, Vol. 133, No. 5, pp. 255-270 (Sep. 1986).
3) Brayton, R. K. and McMullen, C. T.: Synthe-

sis and Optimization of Multi-Stage Logic, ICCD '84 (Oct. 1984).
4) 笹尾, 東田: 論理合成システム: MACDAS, 情報処理学会設計自動化研究会資料, 34-2 (1986).
5) Clocksin, W. F. and Mellish, C. S.: Programming in Prolog, Springer-Verlag (1981).
6) Fujita, M. Tanaka, H. and Moto-oka, T.: Logic Design Assistance with Temporal Logic, 7th Computer Hardware Description Language and their Applications, Tokyo (Aug. 1985).
7) Wojcik, A. S., Kljoich, J. and Srinivas, N.: A Formal Design Verification System Based on an Automated Reasoning System, Proc. 21st Design Automation Conference (June 1984).
8) Wagner, T. J.: Hardware Verification, Dept. of Computer Science, Stanford Univ., Report STAN-CS-77-632 (Sep. 1977).
9) Gordon, M. J. C.: LCF-LSM, University of Cambridge Computer Laboratory Technical Report, No. 41 (1983).
10) Gordon, M. J. C., Milner, R. and Wadsworth C.: Edinburgh LCF: A Mechanised Logic of Computation, Lecture Notes in Computer Science, No. 78, Springer-Verlag (1979).
11) Milner, R.: A Calculus of Communicating Systems, Lecture Notes in Computer Science, No. 92, Springer-Verlag (1980).
12) Barrow, H. G.: VERIFY: A Program for Proving Correctness of Digital Hardware Design, Artif. Intell., Vol. 24, No. 1-3, pp. 437-492 (1984).
13) Kabat, W. C. and Wojcik, A. S.: Automated Synthesis of Combinational Logic Using Theorem-Proving Techniques, IEEE Trans. Comput., Vol. C-34, No. 7, pp. 610-632 (July 1985).
14) Manna, Z. and Pnueli, A.: Verification of Concurrent Programs, Part 1: The Temporal Framework, Dept. of Computer Science, Stanford Univ., Report STAN-CS-81-836 (June 1981).
15) Moszkowski, B. C.: Reasoning about Digital Circuit, Dept. of Computer Science, Stanford Univ., Report STAN-CS-83-970 (July 1983).
16) Wolper, P.: Temporal Logic Can Be More Expressive, 22nd Annual Symposium on Foundation of Computer Science (Oct. 1981).
17) Clarke, E. M. and Emerson, E. A.: Design and Synthesis of Synchronization Skeltons Using Branching-Time Temporal Logic, Proc. Logics of Programs, New York (May 1981).
18) Fujita, M., Tanaka, H. and Moto-oka, T.: Specifying Hardware in Temporal Logic and Efficient Synthesis of State-Diagrams Using Prolog, Proc. FGCS '84, Tokyo (Nov. 1984).
19) Moszkowski, B. C.: Executing Temporal Logic Programs, Cambridge Univ. Press (1986).
20) Aoyagi, T., Fujita, M. and Moto-oka, T.: Temporal Logic Programming Language Tokio: Programming in Tokio, Proc. Logic Programming Conference '85, Lecture Notes in Computer Science, No. 221, Springer-Verlag (1986).
21) Kono, S., Aoyagi, T., Fujita, M. and Tanaka, H.: Implementation of Temporal Logic Programming Language Tokio, Proc. Logic Programming Conference '85, Lecture Notes in Computer Science, No. 211, Springer-Verlag (1986).
22) Fujita, M., Kono, S., Tanaka, H. and Moto-oka, T.: Tokio: Logic Programming Language Based on Temporal Logic and Its Compilation to Prolog, Proc. 3rd International Conference on Logic Programming, Lecture Notes in Computer Science, Springer-Verlag (1986).
23) Dill, D. L., Clarke, E. M.: Automatic Verification of Asynchronous Circuits Using Temporal Logic, IEE Proceedings-E, Vol. 133, No. 5, pp. 276-282 (Sep. 1986).
24) 平石, 矢嶋: 正則集合と表現等価な正則時相論理 RTL, 情報処理学会論文誌, Vol. 28, No. 2 (1987).
25) Hanna, F. K. and Daeche, N.: Specification and Verification of Digital Systems Using Higher-Order Predicate Logic, IEE Proceedings-E, Vol. 133, No. 5, pp. 242-254 (Sep. 1986).
26) Fujita, M., Kono, S., Tanaka, H. and Moto-oka, T.: Aid to Hierarchical and Structure Logic Design Using Temporal Logic and Prolog, IEE Proceedings-E, Vol. 133, No. 5, pp. 283-294 (Sep. 1986).
27) 中村, 河野, 藤田, 田中: Tokio に基づく論理回路の検証, 情報処理学会設計自動化研究会資料, 34-1 (1986).
28) 笹尾: シリコンコンパイレーション講習会論文集, 情報処理学会 (1987).

(昭和62年2月5日受付)