ハードウェア設計言語 DDL による計算機設計支援システム†

川戸信明†† 斎藤隆夫†† 上原貴夫††

近年，電子計算機の LSI 化が進み，近い将来 VLSI も実現されようとしている．このような状況では，新しく計算機を開発する際，論理設計の誤まりを早期に見出だし修正することが非常に重要になってくる．これは，フィールドでの技術的変更がほとんど不可能になるためである．したがって，論理設計の高信頼化を可能にする設計法の必要性が最近特に認識されるようになってきた．

本稿で報告するシステムは，設計の初期段階であるレジスタ・トランスファ・レベルで機能設計を容易に検証できるようにすることを目的としている．この段階で充分な検証を行った後，ゲート・レベルの設計に進めば，機能設計と物理設計の分離が可能となり，高信頼化が達成できると考えられる．

本システムは Dietmeyer らによる DDL をハードウェア設計言語として採用している．ただし，大型計算機の設計にも使用可能なようにその構造的記述の拡張を行った．このシステムは現在2つの支援ソフトウェアからなっている．一つは設計の検証を行うためのシミュレータで，他はレジスタ・トランスファ・レベルの設計からゲートレベルの設計に自動変換するトランスレータである．

本文では，DDL の拡張機能を説明した後，シミュレータおよびトランスレータの構成，処理の詳細について述べる．

1. はじめに

最近の電子計算機は高集積度の LSI を用いて設計されており，近い将来には VLSI も実現されようとしている．このような状況では，フィールドにおける技術的変更は困難になり，設計ミスの与える影響は重大になる．したがって，論理設計の高信頼化が今後一層重要になる．このためには，論理設計ミスを未然に防ぎかつその発見が容易にできる設計手法の確立が必要である．

従来，計算機の設計者は，仕様が決定すると，ブロック図，ステート・ダイヤグラム，タイム・チャート等を用いて機能設計を行っていた．これらの方法は完全にシステムの構成と機能を表現するものではなく，その標準化も困難である．したがって，設計者間の情報伝達や管理上に問題がある．特に，大型機の設計では，この期間は長期にわたり，重大となる．また，シミュレーションを行うためには，ゲート・レベルの設計の完成を待たねばならず，そのシミュレータの効率上，充分なシミュレーションが行われないこともある．

そこで，我々はレジスタ・トランスファ・レベルを論理設計に積極的に取り入れ，この段階で充分に機能設計の検証を行った後，ゲート・レベルの設計に進めば，設計ミスを早期に効率的に発見でき，論理設計の高信頼化が達成されると考え，ハードウェア設計言語 DDL[1)~5)] を導入拡張し，その支援ソフトウェアの開発を行ってきた．レジスタ・トランスファ・レベルの設計言語としては，並列動作の記述が容易で，state machine 記述，豊富なオペレータおよびトランスレーション・アルゴリズムが確立しているという特徴を持つことから，D. Dietmeyer と J. Duley により提案された DDL を採用した．ただし，大型計算機のように複数の設計者グループより設計されるシステムにも使えるように，構造的記述の拡張を行った．支援ソフトウェアとしては，設計の検証を行うためにシミュレータと機能設計をゲート・レベルに自動変換するトランスレータを開発した．シミュレータは DDL とゲート・レベルの両者を入力としてシミュレーションを行えるようにも構成されている．トランスレータは，従来設計者が行っていた作業を自動化するもので，これにより得られた設計にもとづきテクノロジに応じた設計が行われる．ここに発生する設計ミスはシミュレータによって検出される．また，ゲート・レベルの設計が一部のみでも，DDL で記述された他の部分と組み合わせてシミュレーションを行うことができる．

本論文では，DDL の拡張機能と，シミュレータおよびトランスレータの構成，アルゴリズム，性能について述べる．

2. DDL を用いた論理設計

図1に DDL を用いた場合の設計の各段階とそれを

† Computer Aided Design System Based upon Hardware Description Language DDL by NOBUAKI KAWATO, TAKAO SAITO, and TAKAO UEHARA (Computer Science Laboratory, Fujitsu Laboratories, Ltd.).

†† (株)富士通研究所電子研究部第一研究室

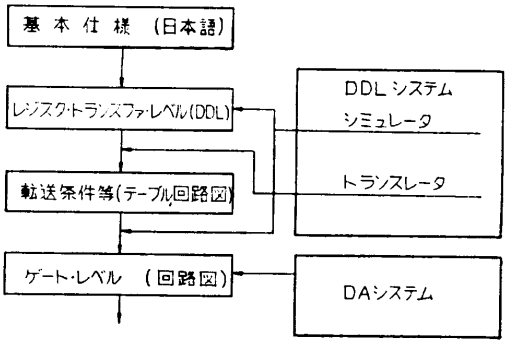

図 1 設計の流れと DDL システム
Fig. 1 Flow of design and DDL system.

支援するシステムを示す．設計は以下のように行われる．まず，システムの基本仕様が決定され，次に DDL を用いてその構造と機能を記述する．その後，基本動作に関するレビュとシミュレーションによる詳細な動作の確認を充分に行う．これが完了すれば，トランスレータを用いて，DDL 記述からゲート・レベルの設計に自動変換する．次に設計者は半導体テクノロジを考慮して，ゲート・レベルの設計を修正する．これは，トランスレータの発生した AND/OR ゲートを NAND ゲートに変換したり，LSI チップへの分割などが必要なためである．この時点で発生し得る設計ミスは，シミュレーションによるゲート・レベルと DDL による機能仕様との比較によって検出する．

3. DDL 言語仕様の拡張

本システムは，大型計算機の設計にも使用することを目的としている．通常，大型計算機は複数のユニットに分割されて設計される．このためには，従来の〈SYSTEM〉および〈AUTOMATON〉という構造宣言だけでは不充分で，〈UNIT〉，〈BLOCK〉および〈EXTERNAL〉宣言を新たに定義した．〈UNIT〉はシミュレーション可能な最小単位であり，〈SYSTEM〉は複数の〈UNIT〉からなる．ユニット間の信号線は〈EXTERNAL〉として宣言される．〈BLOCK〉はユニットを更に分割する場合に用いる．これらの宣言により，ユニットへの分割とその間のインタフェースが決まると，以後設計は各ユニット独立に行える．これらの宣言の関係を**図 2** に示す．なお，小規模システムでは〈SYSTEM〉，〈AUTOMATON〉だけでもよい．ゲート・レベルと結合する場合には，DDL 記述とゲート・レベル記述とのインタフェースは〈EXTERNAL〉信号線のみ許される（**図 3**）．

以上の構造的記述の拡張の他に，乗除算やシフト等の組み込み関数の使用を可能とした．

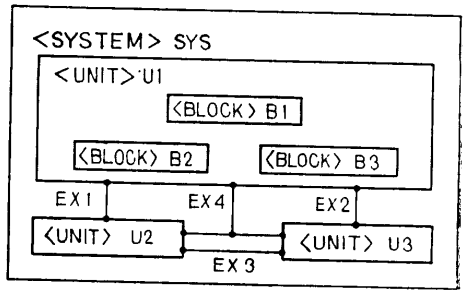

図 2 構造宣言文の拡張
Fig. 2 Structured declarations.

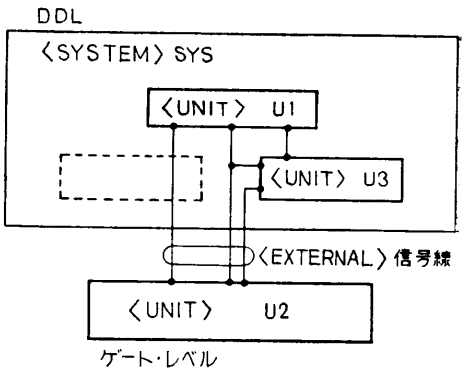

図 3 結合シミュレーション
Fig. 3 Combined simulation.

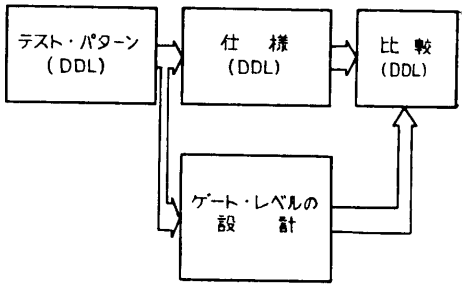

図 4 結合シミュレーションによるゲート・レベル設計の検証
Fig. 4 Exhaustive simulation.

4. DDL シミュレータ

DDL シミュレータはレジスタ・トランスファ・レベルにおける機能設計の検証と，DDL で与えられる仕様と設計者によるゲート・レベルの設計との一致を確認するために使用される（**図 4**）．さらに，ゲート・レベルのシミュレーションを効率よく行うためにも使われる．

4.1 構 成

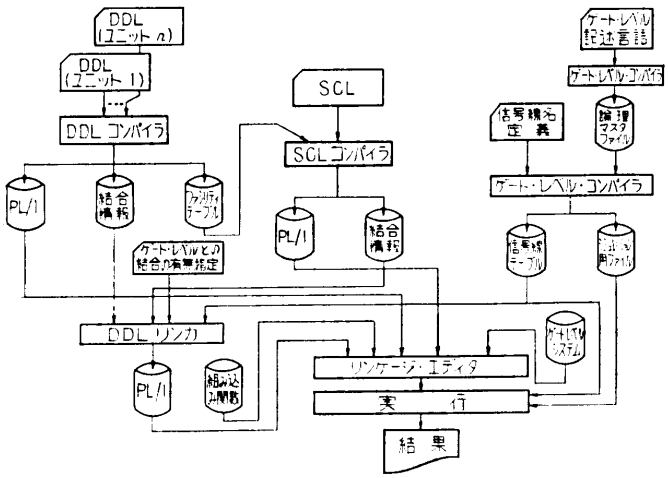

図 5 DDL シミュレータの構造
Fig. 5 Structure of DDL simulator.

図5に本シミュレータの構成を示す．シミュレータの入力は，DDL とシミュレーションの制御のための SCL (Simulator Control Language) 記述である．入力となる DDL の最小単位は〈UNIT〉であり，この単位で分割コンパイルができる．DDL コンパイラは DDL を PL/1 に変換し，ファシリティ・テーブルおよび変換された PL/1 のサブルーチン名とそれが呼ばれる条件等からなる結合情報を出力する．SCL コンパイラはファシリティ・テーブルを参照して，SCL を PL/1 に変換し，同時にイベント時刻等の結合情報を出力する．DDL リンカは組み合わせるべきユニットに関する DDL と SCL の結合情報をもとに，メイン・プログラムを PL/1 で作成する．次に，メイン・プログラムと DDL および SCL コンパイラが作成したモジュールおよび組み込み関数に対するモジュールとが結合され，シミュレーションが行われる．ゲート・レベルと結合する場合は，ゲート・レベル設計言語と，信号線に与える名前がさらに入力される．DDL リンカは DDL の〈EXTERNAL〉信号線名と同一名が宣言されている場合はこれをインタフェースとして，結合システムのためのメイン・プログラムを作成する．

シミュレータは以上のように構成されているので，インタフェースの変更がない限り，無修正のユニットを再コンパイルする必要はなく，メイン・プログラムだけ再作成すればよい．したがって，効率的にシミュレーションを行える．

4.2 アルゴリズム

本シミュレータはコンパイル方式のシミュレータで，シミュレーションはイベント時刻でのみ行なわれる[3]．イベント時刻とは，以下の時刻である．

（1） クロックが 0 から 1 に変化する時刻．
（2） クロックが 1 から 0 に変化する時刻．
（3） ディレイが新しい論理値を取る時刻．
（4） SCL で指定される入出力時刻．

上記の各イベント時刻において，シミュレーション動作は以下の step 1 から step 12 の順序で行われる．ただし，これらの step はすべて実行されるのではなく，必要な step のみが選択的に行われる．

step 1：イベント時刻の決定．
step 2：クロックを 0 から 1 に変化させる．
step 3：クロックを 1 から 0 に変化させる．
step 4：SCL で指定された入力を実行する．
step 5：ディレイの値を新しい論理値に変更する．
step 6：1→0 の変化をしたクロックで制御されるレジスタの値を新しい論理値に変更する．
step 7：組み合わせ回路の論理値を計算する．
step 8：組み合わせ回路として定義されているクロックが 1→0 の変化をしたら step 6 に戻る．変化がなければ step 9 へ．
step 9：0→1 の変化をしたクロックで制御されるレジスタの次に取るべき論理値を計算し，記憶する．
step 10：ディレイの新しい論理値を計算し，変化があれば，その値を記憶し，ディレイ時間後の時刻をイベント時刻として登録する．
step 11：SCL で指定された出力を実行する．
step 12：SCL で指定された終了条件を判定し条件が満たされればシミュレーションを終了させる．そうでなければ step 1 に戻る．

なお，シミュレーションは 2 値で行われる．

ゲート・レベルとの結合シミュレーションは既存のテーブル駆動型の 3 値 (0, 1, X) ゲート・レベル・シミュレータを組み込み，その制御命令を用いることにより実現している．したがって，この場合は step 6 でゲート・レベル側で定義されたレジスタの値を更新する命令を出し，step 7 を以下のように修正して，DDL とゲート・レベルとのインタフェースの値が収束するまで両者の組み合わせ回路のシミュレーションを繰り返す．なお，収束の判定において，X は論理値 0 とみなし，警告メッセージのみ与えている．

```
⟨SYSTEM⟩ SIMPLE:
  ⟨TIME⟩ P⟨(100)⟩, PC⟨(9)⟩.
  ⟨REGISTER⟩ READ, RUN, CLEAR, …….
  ⟨TERMINAL⟩ RR, CW, DCRR, …….
  ⟨AUTOMATON⟩ CPU: P:
    ⋮
  ⟨END⟩ CPU.
  ⟨AUTOMATON⟩ MEMORY: P:
    ⟨REGISTER⟩ MAR(16), MDR(16), RCYCLE,…….
    ⟨STORAGE⟩ M(1024, 16).
    ⟨STATES⟩
      IDLE: MEMAV=1,
        |*RR@CW@DCRR@DCCW*|→RC 1,
          RCYCLE<−RR|DCRR, |*RR@CW*|
          MAR<−CPUTOMEM; MAR<−DCTOMEM.;
          →IDLE..
        RC 1: |*CW*|MDR<−CPUTOMEM;
          |*DCCW*|MDR<−DCTOMEM; MDR<−B'O(16)'.,
          →RC 11.
          ⋮
  ⟨END⟩ MEMORY.
⟨END⟩ SIMPLE.
```

図 6 DDL 記述の例

Fig. 6 DDL description.

```
#C@: PROC; /* COMBINATIONAL CIRCUIT */
   DCL 1 @O EXT ALIGNED, /* FACILITIES IN
            SYSTEM */
         2(P, PC) BIT(1), /* TIME */
         2(RR, RR#) BIT(1), /* TERMINAL */
                  .
   DCL 1 @D EXT ALIGNED,
               /* FACILITIES IN AUTOMATON
                  MEMORY */
         2(MEMORY$MAR, MEMORY$MAR#) BIT(16),
                  .
   DCL #CHANGE BIN FIXED(15) INIT(1);
   P=#CLKSTA ('P'); PC=#CLKSTA ('PC');
   DO WHILE (#CHANGE¬=0);
      RR#='0' B;      /* CLEAR ALL TERMINALS */
      CW#='0' B;
         .
      #CHANGE=0;
      CALL #C1;
      CALL #C2;
         .
 /* UPDATE ALL TERMINALS AND IF ANY CHANGE
          THEN INCREASE #CHANGE BY 1     */
     IF RR#¬=RR THEN DO; RR=RR#; #CHANGE
          =#CHANGE+1; END;
     IF CW#¬=CW THEN DO; CW=CW#; #CHANGE
          =#CHANGE+1; END;
         .
   END;
END #C@;
                  .
#C2: PROC; /* AUTOMATON MEMORY */
   DCL 1 @O EXT ALIGNED,
               /* FACILITIES IN SYSTEM */
         .
   DCL 1 @D EXT ALIGNED,
               /* FACILITIES IN AUTOMATON
                  MEMORY */
         .
   IF MEMORY$IDLE THEN DO; /* STATE IDLE */
      MEMAV#='1' B;
   END;
   IF MEMORY$WR1 THEN DO;
      IF MEMORY$RCYCLE THEN DO;
         MEMBUS#=MEMORY$MDR;
      END;
   END;
END #C2;
                  .
#U2: PROC; /* 1 TO 0 TRANSITION OF CLOCK */
          /* AUTOMATON MEMORY */
   DCL 1 @O EXT ALIGNED,
               /* FACILITIES IN SYSTEM */
         .
   DCL 1 @D EXT ALIGNED,
               /* FACILITIES IN AUTOMATON
                  MEMORY */
         .
   DCL S#2(7) BIT(1) EXT;
   DCL I#2(8) BIT(1) EXT;
         .
   IF S#2(1) THEN DO; /* STATE IDLE */
      IF I#2(1) THEN DO;
         MEMORY$IDLE=MEMORY$IDLE#;
         MEMORY$RC1=MEMORY$RC1#;
         MEMORY$RCYCLE=MEMORY$RCYCLE#;
         IF I#2(2) THEN DO;
            MEMORY$MAR=MEMORY$MAR#;
         END;
         ELSE DO;
            MEMORY$MAR=MEMORY$MAR#;
         END;
      END;
      ELSE DO;
         MEMORY$IDLE=MEMORY$IDLE#;
      END;
   END;
END #U2;
#T2: PROC; /* 0 TO 1 TRANSTION OF CLOCK */
          /* AUTOMATON MEMORY */
   DCL 1 @O EXT ALIGNED,
               /* FACILITIES IN SYSTEM */
         .
   DCL 1 @D EXT ALIGNED,
               /* FACILITIES IN AUTOMATON
                  MEMORY */
         .
   DCL S#2(7) BIT(1) EXT;
   DCL I#2(8) BIT(1) EXT;
         .
   S#2(1)=MEMORY$IDLE; /* STATE IDLE */
   IF S#2(1) THEN DO;
      I#2(1)=#EOR (#EOR (#EOR (RR, CW), DCRR),
            DCCW);
      IF I#2(1) THEN DO;
         MEMORY$IDLE#='0' B;
         MEMORY$RC1#='1' B;
         MEMORY$RCYCLE#=RR|DCRR;
         I#2(2)=#EOR (RR, CW);
         IF I#2(2) THEN DO;
            MEMORY$MAR#=CPUTOMEM;
         END;
         ELSE DO;
            MEMORY$MAR#=DCTOMEM;
         END;
      END;
      ELSE DO;
         MEMORY$IDLE#='1' B;
      END;
   END;
END #T2;
```

図 7 DDL コンパイラが発生する PL/1 プログラム

Fig. 7 PL/1 compiled from DDL.

step 7-1: DDL 側の組み合わせ回路の論理値を計算しゲート・レベルとのインタフェースの論理値を求める.

step 7-2: 上のように決定されたインタフェースの論理値を用いて，ゲート・レベルの組み合わせ回路の論理値を計算し，インタフェースの論理値を求める.

step 7-3: step 7-1 と 7-2 で求めたインタフェースの論理値が一致すれば step 8 へ，そうでなければ step 7-1 に戻る．ただし，インタフェースの論理値としては step 7-2 で求まった値を用いる.

以上のアルゴリズムを実現するため，DDL コンパイラは DDL 記述を以下の PL/1 に翻訳する.

type 1: 信号線の結合オペレーションに対応するモジュール.

type 2: クロックの 0→1 変化に対応したレジスタの転送オペレーションを行うモジュール.

type 3: クロックの 1→0 変化に対応したレジスタの転送オペレーションを行うモジュール.

type 4: ディレイ動作に対応するモジュール.

type 5: オペレータに対応するモジュール.

なお，オートマトン内の各状態は，1ビットのレジスタとみなす．この時，ファシリティ・テーブルと結合情報が同時に出力される．たとえば，**図6**の DDL 記述[14]は**図7**の PL/1 に翻訳され，**図8**のファシリティ・テーブルと**図9**の結合情報が出力される.

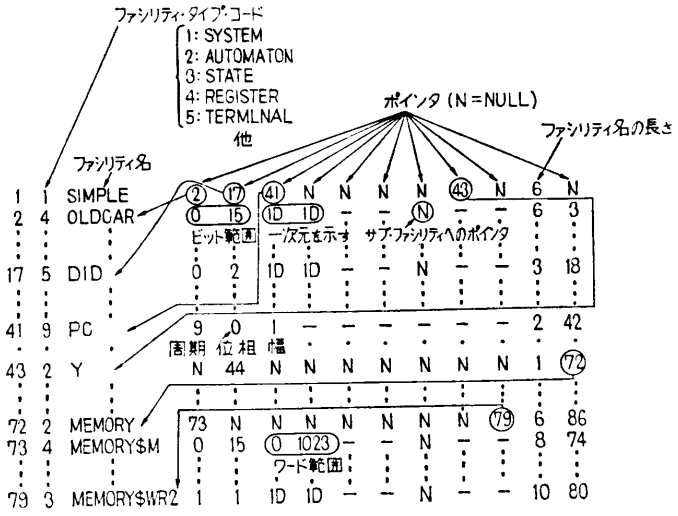

図 8 ファシリティ・テーブル

Fig. 8 Facility table.

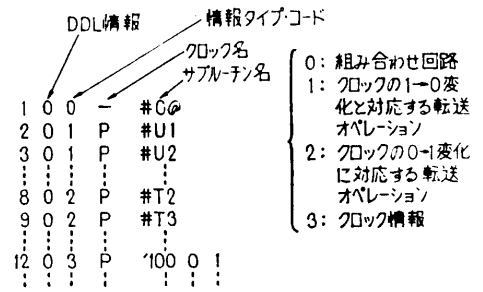

図 9 DDL コンパイラが発生する結合情報

Fig. 9 Linkage information (DDL).

```
SCL SIMPLE;
        /* TEST PROGRAM */
   INIT AT 0 MEMORY$M(0)=B '0100000000001111';
   INIT AT 0 MEMORY$M(1)=B '0111000000010000';
                ⋮
   INIT AT 0 MEMORY$M(24)=B '0000000000000110';
   INIT AT 0 RUN=B '1';
        /* TRACE OF FACILITIES */
   TRACE (P) READ, CPU$GO, CPU$CR, CPU$CAR,
             CPU$ACC,
                ⋮
             (C(8), S, B, S, B, S, X, S, X, S, X
                ⋮
             B, S(2), B, S, X, S, X);
             /* END CONDITION */
   END ON CPU$CAR='0000000000010000' B;
LCS;
```

図 10 SCL 記述の例

Fig. 10 SCL description.

一方，SCL コンパイラは SCL 記述を以下の PL/1 モジュールに翻訳するとともに，結合情報を出力する.

type 1: 入力を行うモジュール.

type 2: 出力を行うモジュール.

type 3: 終了条件を判定するモジュール.

たとえば，**図 10** の SCL 記述からは**図 11** の PL/1 と**図 12** の結合情報が出力される.

DDL リンカは結合情報をもとに，クロック，イベント時刻，ディレイ等に関するテーブルを作成し，既述のアルゴリズムを実現するメイン・プログラムを作り出す．たとえば，図9と図 12 の結合情報から**図 13** のメイン・プログラムが作成される.

以上の DDL および SCL コンパイラは，信頼性の向上と開発工程の短縮を目的として，コンパイラ・コンパイラ[6]を用いて開発された(**図 14**)．DDL および SCL の文法はそれぞれ約 300 個，約 100 個の生成規則からなる

```
#SINTO: PROC; /* INITIALIZATION */
   DCL 1 @O EXT ALIGNED,
            /* FACILITIES IN SYSTEM */
         .
   DCL 1 @D EXT ALIGNED,
            /* FACILITIES IN AUTOMATON
               MEMORY */
         .
   MEMORY$M(1)='0100000000001111' B;
   MEMORY$M(2)='0111000000010000' B;
         .
   RUN='1' B;
   CPU$IF1='1' B; /* INITIAL STATE */
   MEMORY$IDLE='1' B;
         .
END #SINTO;
#SOUT: PROC; /* OUTPUT */
   DCL 1 @O EXT ALIGNED,
            /* FACILITIES IN SYSTEM */
         .
   DCL 1 @D EXT ALIGNED,
            /* FACILITIES IN AUTOMATON
               MEMORY */
         .
   DCL #TRCFL FILE PRINT;
   DCL #TRCMRX(132) CHAR(14) INIT((132)(14)' ');
   DCL #TRLN CHAR(132);
         .
   ON ENDPAGE (#TRCFL) BEGIN;
      SUBSTR (#TRCMRX(8), 5, 9)='CLOCK /P/';
      SUBSTR (#TRCMRX(12), 11, 3)='RUN';
      SUBSTR (#TRCMRX(14), 7, 7)='CPU$IF1';
         .
      DO #I=1 TO 14;
         DO #J=1 TO 132;
            SUBSTR (#TRLN, #J, 1)
               =SUBSTR (#TRCMRX(#J), #I, 1);
         END;
      END;
   END;
   IF #CLKUPD ('P')|(#TIME=0) THEN DO;
         .
      PUT FILE (#TRCFL) EDIT (#CLKNO, READ,
         CPU$GO,
                  .
            (COLUMN(1), F(8), X(1), B, X(1),
                  .
   END;
         .
END #SOUT;
#SEND: PROC (#ENDBT); /* END CONDITION */
   DCL 1 @O EXT ALIGNED,
            /* FACILITIES IN SYSTEM */
         .
   DCL 1 @E EXT ALIGNED,
            /* FACILITIES IN AUTOMATON
               CPU */
         .
   DCL #TIME BIN FIXED(31) EXT; /* SIM. TIME */
         .
   IF CPU$CAR='0000000000010000' B THEN DO;
         .
      #ENDBT='1' B;
   END;
END #SEND;
```

図 11 SCL コンパイラが発生する PL/1 プログラム

Fig. 11 PL/1 compiled from SCL.

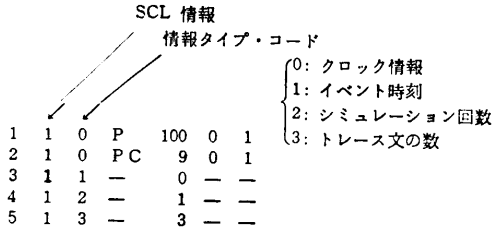

図 12 SCL コンパイラが発生する結合情報

Fig. 12 Linkage information (SCL).

```
#SIMCTL: PROC OPTIONS (MAIN);
            /* MAIN PROGRAM FOR DDL
               SIMULATOR */
   DCL 1 #CLOCKTBL(2) EXT ALIGNED,
            /* CLOCK TABLE */
         2 #IDENTIFIER CHAR(32) INIT ('P', 'PC'),
         2 #PERIOD BIN FIXED(31) INIT (100, 9),
         2 #PHASE BIN FIXED(31) INIT (0, 0),
         2 #WIDTH BIN FIXED(31) INIT(1, 1),
         2 #STATE BIT(1) INIT((2)(1) '0' B),
         2 #TRANSFER BIT(1) INIT((2)(1) '0' B),
         2 #UPDATE BIT(1) INIT((2)(1) '0' B);
   DCL #EVENTTBL(1) BIN FIXED(31) INIT(0);
                        /* EVENT TABLE */
   DCL #TIME BIN FIXED(31)EXT; /* SIM. TIME */
   CALL #SINTO; /* INITIALIZATION */
   DO WHILE ('1' B);
      #TIME=2147483647; /* EVENT TIME */
      DO #I=1 TO 2;
         IF #CLOCKTBL (#I). #NEXTEVENT<TIME
            THEN #TIME=#CLOCKTBL (#I).
               #NEXTEVENT;
      END;
      DO #I=1 TO 1;
         IF #EVENTTBL (#I)<#TIME THEN
            #TIME=#EVENTTBL (#I);
      EDN;
      DO #I=1 TO 2;
         IF #CLOCKTBL (#I).#NEXTEVENT=#TIME &
            #CLOCKTBL (#I). #STATE='0' B THEN
               DO;
               #CLOCKTBL (#I). #NEXTEVENT=
                  #CLOCKTBL (#I). #NEXTEVENT
                     +#CLOCKTBL (#I). #WIDTH;
                        .
         END;
      END;
      IF #CLOCKTBL(1). #UPDATE THEN DO;
                        /* CLOCK P, 1 TO 0 */
         CALL #U1;
         CALL #U2;
               .
      END;
         .
      CALL #C@; /* COMBINATIONAL CIRCUIT */
      IF #CLOCKTBL(1). #TRANSFER THEN DO;
                        /* CLOCK P, 0 TO 1 */
         CALL #T1;
         CALL #T2;
               .
      END;
         .
      CALL #SOUT; /* OUTPUT */
         .
      CALL #SEND (#ENDBT);
                        /* END CONDITION */
      IF #ENDBT THEN GO TO #ENDSIM;
         .
   END;
#ENDSIM:
         .
END #SIMCTL;
```

図 13 DDL リンカの発生するメイン・プログラム

Fig. 13 Main program generated by DDL linker.

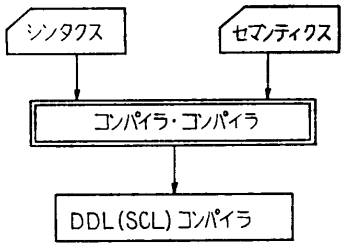

図 14 コンパイラ・コンパイラの使用
Fig. 14 Use of compiler-compiler.

LALR(2)文法（2個まで区切り記号を先読みすることにより解析可能な文法）で定義されている．セマンティクスは PL/1 で書かれ，DDL は約2万ステップ，SCL は約5千ステップからなる．開発に要した工数は約2人年である．

4.3 使用例と性能

図6の DDL と図 10 の SCL 記述を用いて，シミュレーションを行った結果を**図 15** に示す．本シミュレータは M 190 上にインプリメントされており，性能例を**図 16** に示す．

5. DDL トランスレータ

DDL トランスレータは機能設計から回路設計への自動変換を行うサポート・ソフトウェアであり，設計ミスを減少させ，回路設計を効率よく行うために利用

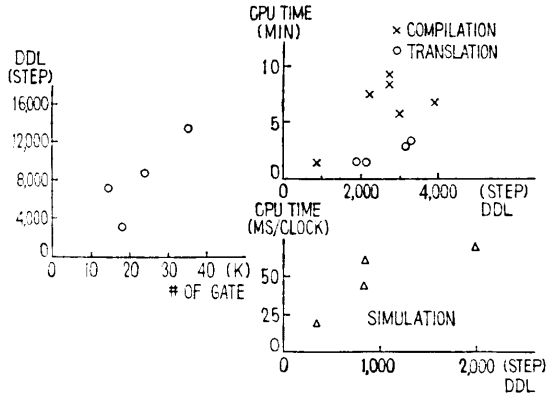

図 16 DDL シミュレータの性能
Fig. 16 Performance of DDL simulator.

される．

5.1 構　成

本トランスレータは4つのプログラムから構成されている（**図 17**）．パス1はソースリストを出力し，ファシリティ・テーブルを作成するモジュールである．パス2は入力 DDL 記述を IF 文のみから成る DDL 記述へと変換する．パス3はネスト構造の IF 文を単純な IF 文に変換するモジュールである．パス4は各ファシリティについての入力組合せ回路等をまとめ，ハードウェアの階層構造を反映する形式で出力する．

本トランスレータは DDL コンパイラ同様コンパイ

CLOCK / PASS	READ DATA	CPU $ AGO	CPU $ IR	CPU $ CAR	CPU $ CAC	CPU $ BUF	MEM AV	RR	CW	CPU TOM	MEM BUS	MEMMMM EMEEE OMMMM ROOOO YRRRR $YYYY I$$$$ DRRWW LCCRR E1212	MEMORY $ CAR	MEM DR	MEM DR
792	1	0	6013	000D	0002	0001	0	0	0	0000	0000	00001	1	000C	6013
793	0	1	6013	000D	0002	0001	1	1	0	0013	0000	10000	1	000C	6013
794	0	0	6013	000D	0002	0001	0	0	0	0000	0000	01000	1	0013	6013
795	0	0	6013	000D	0002	0001	0	0	0	0000	0000	00000	1	0013	0000
796	0	0	6013	000D	0002	0001	0	0	0	0000	0000	00000	1	0013	0000
797	0	0	6013	000D	0002	0001	0	0	0	0000	0000	00000	1	0013	0000
798	0	0	6013	000D	0002	0001	0	0	0	0000	0000	00000	1	0013	0000
799	0	0	6013	000D	0002	0001	0	0	0	0000	0000	00000	1	0013	0000
800	0	0	6013	000D	0002	0001	0	0	0	0000	0000	00000	1	0013	0000
801	0	0	6013	000D	0002	0001	0	0	0	0000	0000	00100	1	0013	0000
802	1	0	6013	000D	0002	0001	0	0	0	0000	0005	00010	1	0013	0005
803	1	0	6013	000D	0002	0005	0	0	0	0000	0000	00000	1	0013	0005
804	1	0	6013	000D	FFFD	0005	0	0	0	0000	0000	00000	1	0013	0005

図 15 シミュレーション結果
Fig. 15 Simulation results.

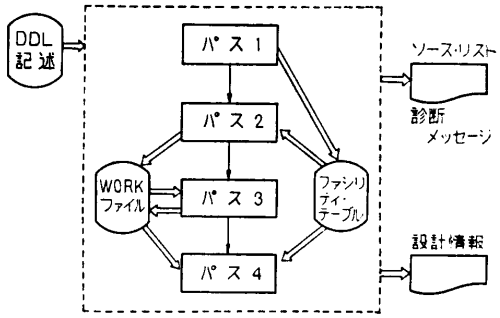

図 17 DDL トランスレータの構造
Fig. 17 Structure of DDL translator.

ラ・コンパイラを用いて作成され（図 14），シンタクス部分は DDL コンパイラと共通である．

5.2 変換処理

トランスレータは入力 DDL 記述中のファシリティ宣言文を除去し，パス 1 で作成したファシリティ・テーブル（図 8）を用いて，オペレーションに関係するファシリティのビット範囲を明確にし，サブ・ファシリティはメイン・ファシリティに変換する．CASE 文は IF 文の集合に変換される．〈SUBSTITUTE〉構文（マクロ）で定義されたオペレーションは，それを参照している部分に挿入される．〈STATES〉構文は，シーケンス制御回路がオペレーションを制御するような IF 文へと変換される．また暗黙の了解に従い省略されている部分を明確にする．これらの変換によりネスト構造の IF 文からなる DDL 記述が生成され，これはパス 3 で条件の分配処理により，次の条件付基本オペレーションからなる DDL 記述へと変換される[4]．

1) レジスタ転送オペレーション
   |* condition *| sink<−source.
2) ターミナル結合オペレーション
   |* condition *| sink=source.

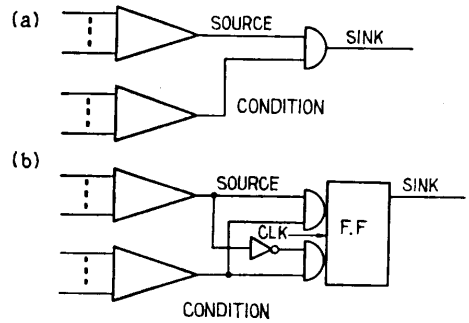

図 18 条件付き基本オペレーションの実現法の例
Fig. 18 Schematic representation of conditional basic operation.

3) ステート遷移オペレーション
   |* condition *| −>state-id.

ここに，|* *|. は IF 文である．**図 18** に条件付基本オペレーションの実現例を示す．上述のように変換された条件付基本オペレーションは，パス 4 において，SINK となるファシリティについてまとめられ，

```
〈AUTOMATON〉 MEMORY: P:
  |* MEMORY$IDLE *|MEMAV=1.,
  |*(RR(0)@CW(0)@DCRR(0)@DCCW(0)) &
     MEMORY$IDLE *|
  →MEMORY$RC1.,
  |* (RR(0)@CW(0)@DCRR(0)@DCCW(0)) &
     MEMORY$IDLE *|
  MEMORY$RCYCLE(0)<−RR(0)|DCRR(0).,
        ⋮
  |* CW(0) & MEMORY$RC1 *|
  MEMORY$MDR (0:15)<−CPUTOMEM (0:15).,
  |* DCCW(0) &¬CW(0) & MEMORY$RC1 *|
  MEMORY$MDR (0:15)<−DCTOMEM (0:15).,
  |* ¬DCCW(0) &¬CW(0) & MEMORY$RC1 *|
  MEMORY$MDR (0:15)<−B '0000000000000000'.,
        ⋮
```

図 19 構文の簡単化の例
Fig. 19 Example of syntax reduction.

61 MEMORY$MDR (0:15) 16 BIT (S) REGISTER				
NO	SINK RANGE	SOURCE	CONDITION	CLK
61-1	(0:15)	B '00000000 00000000'	¬DCCW(0) &¬CW(0) & MEMORY$RC1	P
61-2		CPUTOMEM (0:15)	CW(0) & MEMORY$RC1	P
61-3		DCTOMEM (0:15)	DCCW(0) &¬CW(0) & MEMORY$RC1	P
61-4		MEMORY$M (MEMORY$MAR (0:15))	MEMORY$RCYCLE (0)&MEMORY$RC2	P

図 20 テーブルによる変換結果の出力例
Fig. 20 Example of translation result (table).

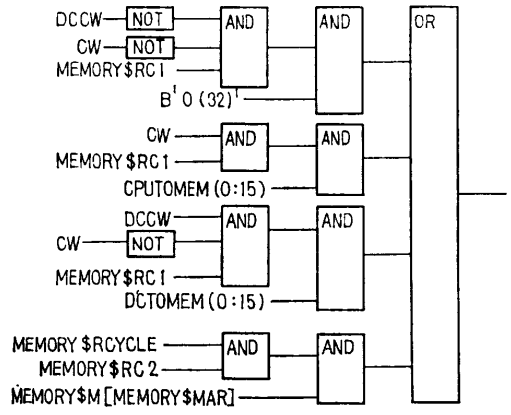

図 21 回路図による変換結果の出力例
Fig. 21 Example of translation result (logic diagram).

さらにハードウェアの階層構造を反映した順序にソートされ，表として出力される．また，これを回路図形式で出力することも可能である．

5.3 使用例と性能

図6の DDL 記述を入力した場合のトランスレーションの中間結果を**図 19** に，出力例を**図 20, 21** に示す．また，図 16 にトランスレータの性能を示す．

6. むすび

電子計算機の論理設計において，レジスタ・トランスファ・レベルで機能の検証を充分に行い，ゲート・レベルの設計に進めば，その信頼度を高めることができる．このために，設計言語 DDL を導入し，大型計算機の設計にも使用できるように構造記述の拡張を行った．さらに，その支援ソフトウェアとして，シミュレータおよびトランスレータを開発した．これらはコンパイラ・コンパイラの使用により短期間に高信頼度で開発された．このシステムにより，さらに管理の合理化，教育，設計の財産化も可能になる．

今後の問題としては，現在検証はレビュとシミュレーションによっているが，記号処理による DDL 記述の検証等のより効率的な設計の検証方法を確立することが残されている．また，DDL からゲート・レベルへのトランスレーションをより高信頼度で効率よく行う手法の開発が必要である．

謝辞 Wisconsin 大学 Dietmeyer の教授，有益な助言，協力を頂いた富士通槌本課長，徳倉課長ならびに荒川調査役に感謝致します．また，日頃御指導頂く富士通研究所宮川電子研究部長に謝意を表します．

参考文献

1) Duley, J. R. and Dietmeyer, D. L.: A Digital System Design Language (DDL), IEEE TC, Vol. C-17 (1968).
2) ibid: Translation of a DDL Digital System Specification to Boolean Equations, ibid., Vol. C-18 (1969).
3) Arndt, R. L. and Dietmeyer, D. L.: DDLSIM-A Digital Design Language Simulator, Proc. NEC, Vol. 26 (1970).
4) Dietmeyer, D. L.: Logic Design of Digital System, Allyn and Bacon (1971).
5) Breuer, M. A., editor: Digital System Design Automation: Language, Simulation & Data Base, Computer Science Press (1975).
6) 牧之内ほか：MLTG-マイクロプログラミング・ランゲージ・トランスレータ・ジェネレータ，情報処理学会論文誌，Vol. 20, No. 1 (1979).

(昭和54年5月11日受付)
(昭和54年9月20日採録)