PLC命令列を論理回路に変換するツールの実装と評価

Converting PLC instruction sequence into logic circuit: implementation and evaluation

豊橋技術科学大学 大学院 工学研究科
知識情報工学専攻 市川研究室

033701　秋中 昌訓

目次

1　はじめに

機械装置を制御するコントローラとしてProgrammable Logic Controller（PLC, 別名シーケンサまたはプログラマブルコントローラ）が利用されている．特に，順序制御を行う生産設備機械のメインコントローラとして広く導入されている[1)]．PLCは一種の計算機であり，記述された制御論理は制御システムに合わせて変更可能なため，生産現場での制御内容の変更に柔軟に対応することができる．しかし，大規模な制御では処理速度の不足が問題になる場合がある．また，PLCプログラムはソフトウェアであるため複製や解析が容易で，技術情報の流出やコピー製品の出現が懸念される．

そこで近年，FPGA（Field Programmable Gate Array）を利用したPLCプログラムのハードワイヤード化が検討されている[7)]．FPGAは再構成可能なLSIであるため，PLCの柔軟性を保ったまま処理速度の改善が可能である．また，1チップで数十〜数百万ゲートを集積できるため大規模な制御論理も小さな面積で実装することができる．部品点数の削減や消費電力の低減による低コスト化も期待できるため，高性能を必要とする大規模システムだけでなく，安価な小規模システムにも応用可能である．また，ハードウェアであるFPGAの解析はPLCプログラムの解析より困難であるため，近年のFPGAの秘匿性向上技術とあわせて，知的所有権の保護にも有効であると考えられる．

その反面，FPGAを用いるには，既存のPLCプログラムの蓄積を放棄して，新たに制御論理を論理回路として記述しなおさなければならない．そこで本研究では，PLCからFPGAへの移行を容易にするため，PLC命令列をハードウェア記述言語VHDLに変換するツールを作成した．さらに，本ツールを用いて実応用のPLCプログラムをFPGA上で実装評価した結果も報告する．

本研究の目的は，全てのPLCをFPGAに置き換えることではない．既存のPLCで処理速度や実装上の問題が生じた場合に，PLCソフトウェアの資産を生かしつつ最新技術を利用するために，PLCからFPGAへの移行を容易にする技術を提示する．本手法は，PLCでは実現が難しい程の高速性や，秘匿性が強く要求される分野に特に適している．

2　関連研究

PLC の制御プログラムを FPGA 上の論理回路で実現するというアイデアは，既に幾つか提案されている．Adamski ら[2)3)] は，制御論理をペトリネットに基づくルールベース言語で記述し，ハードウェア記述言語 PALASM に変換して PLD（Programmable Logic Device）に実装する手法を検討した．Wegrzyn ら[4)5)] は，上述のルールベース言語から VHDL に変換するシステムを報告した．池下ら[6)] は，SFC（Sequential Function Chart）表現の PLC プログラムを Verilog-HDL に変換するツールを報告した．これらの研究は，いずれも機能レベルの制御プログラムから論理回路への変換を行っている．これに対し本研究では，PLC の命令列から論理回路への変換を行う．解析は複雑になるが，より広範囲での適用が可能になる．

宮澤ら[7)] は，ラダー図表現の PLC プログラムから VHDL への変換手法について報告した．ラダー図は PLC 命令列とほぼ同等の記述方法であるが，宮澤らが検討した例は非常に小さく単純で，実応用に関する定量的な評価も行なわれていない．これに対し本研究では，実応用の PLC プログラムから論理回路への変換を行い，定量的な評価を示す．

池田[8)] と山本[9)] は，命令列表現の PLC プログラムをツールを用いて VHDL へ変換することで，PLC 上と FPGA 上での比較評価を行った．これらの研究では，PLC の逐次処理をそのまま FPGA 上で実現するため，ハードウェアの持つ並列性が活かされず性能面での向上が最小限に留まってしまうという問題があった．そこで本研究では，処理の依存関係を解析し並列に動作させるなどの手法を提案し，論理規模や性能面での改善を行った．また，変換ツールの未対応命令への処理を追加することでより広範囲な応用に適用できるよう改良した．

3 PLC

本章では，まずPLCのプログラム記述について述べ，次にPLCを構成する要素について述べる．そしてPLCの動作方式及び外部機器との入出力について述べる．

3.1 PLCプログラム

PLCプログラムとは，PLCが処理する任意の制御論理を記述したプログラムである．その記述にはラダー図が広く利用されている．ラダー図は，リレーによりシーケンス制御を行っていた時代に，リレーや接点の接点方法を図式化する方法として規定された記述方式のため，図1に示すようにはしご状のグラフィカルな図で表現される．このラダー図における処理の最小単位を段と呼ぶ．また，その段は条件部と処理部から構成される．条件部には処理部を制御するための条件が，処理部には（a）リレーへの出力命令や（b）データ処理命令などが設定可能である．処理はラダー図の上から下に向かって一段ずつ逐次的に行われ，一番下の段が処理された後は再び一番上の段に戻って処理が行われる．実行速度が十分に速ければこの同時性，即時性があるといえる．

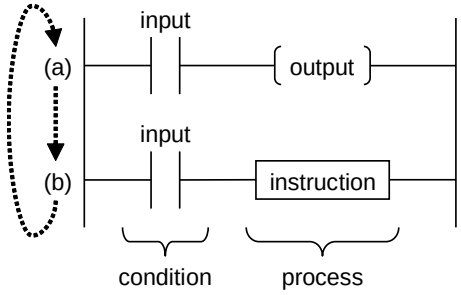


図1 ラダー図の概要

PLCは複数のメーカー（三菱電機，オムロン，etc.）から製品として提供されており，それぞれのPLCで用いられている命令語や用語は多少異なっている．しかし，機能や意味としては等価なものが多いため，1つの機種について理解できればその他の機種についての理解も容易である．本研究では，三菱電機FX2N PLC[12]用のPLCプログラムを扱った．これは，評価に用いたサンプルがFX2N用に記述されているためである．そのため以降の説明では，FX2N用の記述を用いている．

3.2 PLC を構成する要素

表 1 PLC を構成する要素

要素記号	機能
K, H	定数（10 進, 16 進）
X	入力リレー
Y	出力リレー
M	補助リレー
T	タイマ
C	カウンタ
D	データレジスタ
S	ステート
P	ポインタ
I	割り込み信号

PLC には，入出力用のリレーなどの各種要素（デバイス）が内蔵されている（表 1）．このうち，本研究では評価用のサンプルで使用されている 7 種類の要素を扱うこととした．その役割と機能について以下に示す．

定数 PLC プログラム上で数値を扱う場合には必ず記号 K または H で表したものを用いる．この記号に続く数値がそれぞれ 10 進，16 進の定数であることを示す．主にタイマやカウンタの設定値あるいは，応用命令のオペランド中の数値指定に用いる．

入出力リレー PLC の入力端子に接続されている入力用のリレーを記号 X で表す．この入力リレーはプログラムによって駆動することはできない．これに対し出力端子に接続されている出力用のリレーを記号 Y で表す．この出力リレーの状態はプログラム内で読み出して条件として記述することもできる．これらは常開接点（a 接点）と常閉接点（b 接点）のふたつの状態を持つ．

補助リレー 出力リレーと同様に PLC 内の各種要素の接点によって動作する補助用のリレーを記号 M で表す．そのため補助リレーはプログラム内では出力リレーと同様に動作するが，外部負荷を直接駆動することはできないためこの場合には出力リレーを用いる必要がある．なお，PLC の運転中に電源を遮断すると出力リレーや補助リレーはすべて OFF 状態となる．また，補助リレーは KnM（n は自然数）という形の 4 ビット単位でまとめて使用することが可能である．通常はデータレジスタやバッファメモリとのビッ

ト幅を合わせるため 16 ビット（n=4）でまとめられる．

タイマ　PLC 内のクロックパルスを加算計数し，これが所定の設定値に達したときに出力接点が動作するタイマを記号 T で表す．設定値としてはプログラム内の定数を用いたり，データレジスタの内容で間接指定することができる．

カウンタ　PLC 内のコイルの駆動回数を加算計数するカウンタを記号 C で表す．また，設定値を指定しこれが所定の設定値に達したときに出力接点を動作させることもできる．タイマと同様設定値は定数やデータレジスタの内容で指定することができる．カウント値のリセットには RST 命令を用いる．

データレジスタ　数値データを格納するためのデータレジスタを記号 D で表す．データ幅は 16 ビットで，二つのデータレジスタを組合わせて 32 ビットの数値データを格納することもできる．この 32 ビット指定をする場合は記述されているデータレジスタの番号とこれに続く番号のデータレジスタを組み合わせて使用する．このとき記述されているデータレジスタが下位 16 ビットに，続くデータレジスタが上位 16 ビットに割り当てられる．そのためデータレジスタを扱う場合には，16 ビット命令であってもミスを防ぐため基本的に偶数番号のデータレジスタを用いることが推奨されている．データレジスタに一度書込まれたデータは，他のデータを書込まない限り変化しない．ただし，PLC の運転中に電源を遮断するとすべてのデータが 0 にクリアされる．

3.3　特殊要素

補助リレーやデータレジスタには，特殊な機能が定義されているものがある．このうち本研究では以下に示す特殊補助リレーを扱う．

- 電源投入状態を表す RUN モニタの a 接点（M8000）と b 接点（M8001）
- 電源投入直後のスキャンタイムの間だけ立ち上がるイニシャルパルスの a 接点（M8002）と b 接点（M8003）
- 1 秒周期で発振する 1s クロック（M8013）

3.4　PLC の動作方式

多くの PLC は以下に示す処理手順をくり返し実行することでシーケンス制御を行なっている（図 2）．このような方式を一括入出力方式（リフレッシュ方式）と呼び，入力処理から出力処理までの実行時間をスキャンタイム（演算周

期）と呼ぶ．このスキャンタイムが短いほど高速なシステムといえる．

入力処理　プログラムの実行前にシーケンサの全入力端子の ON/OFF 状態を入力イメージメモリに読込む．プログラムの実行中に入力が変化しても入力イメージメモリの内容は変化せず，次のサイクルの入力処理時にこの変化を読込む．なお，PLC では入力接点が ON → OFF，OFF → ON に変化してもその ON/OFF の判定までには入力フィルタによる応答おくれ（約 10ms）がある．

プログラム処理　プログラムメモリの命令内容に応じて，入力イメージメモリやその他要素のイメージメモリから各要素の ON/OFF 状態を読出し，PLC プログラムの最初から順次演算を行なって，そのつど結果をイメージメモリに書込む．したがって，各要素のイメージメモリはプログラムの実行に伴って逐次内容が変化していく．

出力処理　全ての命令の実行が終わると出力のイメージメモリの ON/OFF 状態を出力ラッチメモリへ転送し，これがシーケンサの実際の出力となる．なおシーケンサ内の外部出力用接点は，出力用素子の応答おくれ時間をおいて動作する．

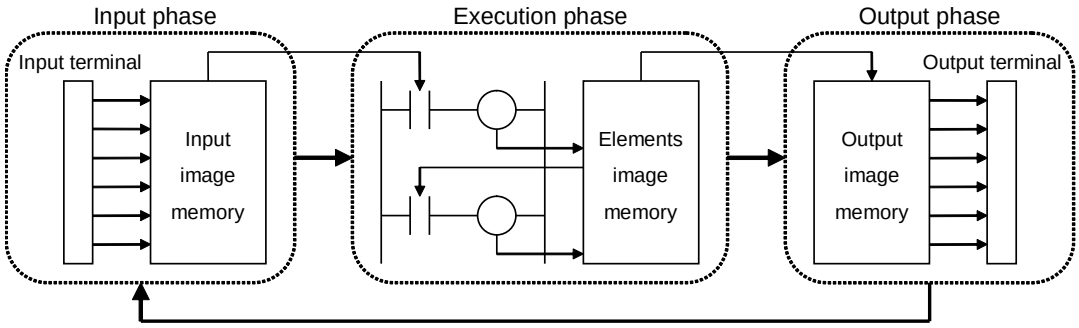


図 2　一括入出力方式の概要

なお，一括入出力方式と対比した方式として遂次入出力方式（ダイレクト方式）がある[13]．この方式は入力と出力の ON/OFF 動作をすぐ取り込んで処理する方式のため動作がわかりやすい反面，入力状態がプログラム処理中に変化してしまう可能性があるので注意が必要となる．また，逐次入出力方式に比べて一括入出力方式の方が一度にすべての入出力を行うため動作が遅くなるように感じられるが，一括入出力方式はイメージメモリを用いたプログラム処理を行

うためこの実行時間が短く，総合的な遅れは逐次入出力方式の方が大きい．本研究で用いたFX2N PLCでは一括入出力方式で動作するため，以降の記述ではこの動作を前提としている．

3.5 外部機器入出力

パルスジェネレータやA/Dコンバータなど，外部に接続されたユニットとの入出力には外部機器入出力命令が用いられる．PLCではこの命令がプログラム中にあった場合，外部機器に内蔵されたBFM（バッファメモリ）を介して，前述の入出力処理を待たず即座に入出力が行われる．BFMは外部機器に内蔵された1アドレス16ビットのRAMメモリのことで，その内容は各機器の制御目的に応じて決められている．図3に32ビット指定による外部機器入出力命令の例を示す．PLC側のデータレジスタ，外部機器のBFM共に16ビットのデータであるため32ビット指定による命令の場合それぞれ指定された番号に続くデータレジスタ及びBFMが用いられる．

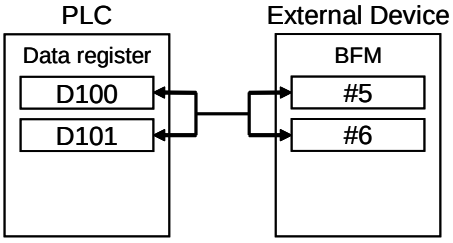


図3 外部機器との入出力の例

4　変換ツールの機能

本章では，まず本研究の位置付けについて述べる．そして変換ツールにおける各種要素の扱いとサポートする命令について述べる．また，変換された制御回路への入出力についても述べる．

4.1　研究の全体像

近年の FPGA ではソフトマクロの CPU コアを利用することで，ソフトウェアによる制御を行うことも可能である．そのため，例えばイーサネットのプロトコルスタックといった通信インターフェースは，リアルタイム OS 上でこうした通信を行うアプリケーションを動作させることで実現できる．プロトコルスタックやユーザインターフェースのように，ソフトウェアによる制御が適している機能もある．また，実現しようとする制御論理のうち速度が要求されないような制御部分にはソフトウェアを利用し，速度が要求される制御部分には専用回路を利用するといったこともできる．これらは組み込みプロセッサに専用回路を周辺機器としてバス接続することによって実現できる．

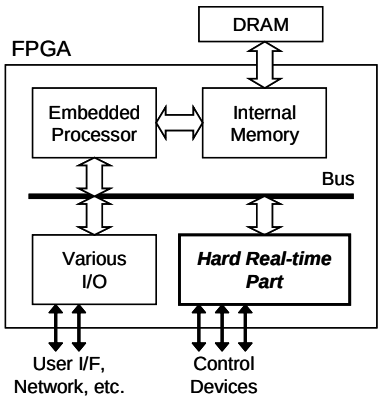


図 4　本研究の位置付け

本研究では，特に速度が要求されるハードなリアルタイム制御用に専用回路を作成することを研究目的とする（図 4）．ただし，この専用回路は PLC プログラムのみを用いて生成するため，山本[9] の作成したパルスジェネレータといった周辺回路とは直接接続することができない．これは，後述するように PLC プログラムからでは周辺機器が何であるか判別できないためである．なお，池田[8] は PLC プログラムから生成された制御回路の入出力信号のうち，周辺回路と接

続する信号を指定することで，制御回路と周辺回路を接続するインターフェース回路生成ツールを作成した．しかし，山本の作成したパルスジェネレータにしか対応しておらず，制御回路の記述にも手を加える必要があるといった問題がある．そのため今後，生成した制御回路と周辺回路を接続するためのインターフェース生成手法について検討していく必要がある．

4.2 変換ツールのサポートする要素

前述のとおり，本研究では 7 種類の要素を扱う．VHDL 記述での各要素の扱いについて以下に示す．

定数 VHDL では n を定数とするとそれぞれ “10#n#”，“16#n#” と記述することで各進数に合わせた数値として扱う．

入出力リレー これらは a 接点と b 接点のふたつの状態を持つため，その状態を 1 ビットのデータタイプ std_logic で保持するよう記述する．

補助リレー 入出力リレーと同様 1 ビットのデータタイプで保持するよう記述する．ただし，補助リレーは KnM（n は自然数）という形の 4 ビット単位でまとめることができるので，その場合は連結してベクタタイプ std_logic_vector として扱う．まとめて使用する場合には記述された補助リレーの番号から連続する番号の補助リレーが用いられる．

タイマ PLC と同様 FPGA のクロックパルスを加算計数し，これが所定の設定値に達したときに出力接点が動作するように記述する．この設定時間に到達したかどうかの状態は 1 ビットのデータタイプで保持する．ただし，FPGA で実装する場合は外部回路から与えるクロック周波数を定数として記述する必要がある．

カウンタ カウント値の保持に 16 ビットのベクタタイプを用い，指定コイルへの出力があった場合にアップカウンタとして動作するよう記述する．また，設定値に到達したかどうかの状態は 1 ビットのデータタイプで保持する．

データレジスタ 16 ビットのベクタタイプで対応し，32 ビット指定がある場合には連続する 2 つのデータレジスタを連結して使用する．

特殊要素 3.3 節で挙げた特殊補助リレーに対応している．各特殊補助リレーの動作を図 5 に示す．なお，1s クロック（M8013）ではタイマと同様クロックパルスをカウントしているため使用時の動作周波数を与える必要がある．

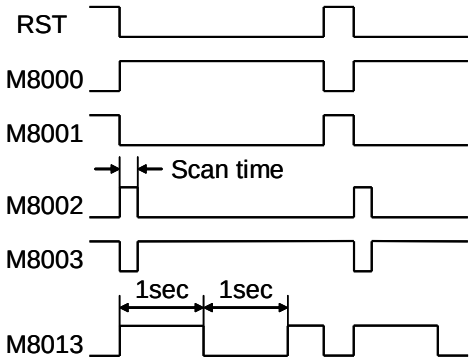


図 5　特殊要素の動作

4.3　変換ツールのサポートする命令

本変換ツールが対応しているPLCの命令語を表6に示す．FX2N PLCで使用できる命令語のうち，7章のサンプルで使用される命令語に対応している．FX2N PLCで使用できる命令には基本命令と応用命令の2種類がある．基本命令は三菱電機のFXシリーズをはじめとして多くのPLCで共通に使用できる命令である．これに対し応用命令は機種に依存した命令で，特定の機種でしか使用できない命令が多く含まれる．例えば，算術演算命令はFXシリーズのすべてで使用可能であるがデータレジスタのビット回転命令などはFX2N，FX2NCシリーズのみでしか使用することはできない．

表 2　変換ツールのサポート命令

分類	ニーモニック
基本命令	LD（LD=）, LDI, LDP, LDF, AND（AND=）, ANI, OR, ORI, ANDP, ANDF, ORP, ORF, ANB, ORB, MC, MCR, OUT, SET, RST, PLS, PLF, NOP, END
応用命令	MOV（DMOV）, ADD（DADD）, SUB（DSUB）, MUL（DMUL）, DIV（DDIV）, TO（DTO）, FROM（DFROM）, BMOV, WAND, ROL, ZRST, HEX, INC

本研究では，前述のとおりFX2N用のPLCプログラムを扱うために，三菱電機のPLCプログラミング環境GX Developer[14]を用いた．また，ラダープログラムから命令列への変換にはGX Converter[15]を用いた．

4.4 入出力の記述

本ツールで作成した制御回路への入力ポートにはマスタークロック（CLK），リセット信号（RST），及びPLCプログラムで用いている入力リレー（X）のすべてが記述される．リセット信号は制御回路のリセットを行うもので，これがON状態になることで補助リレー（M）やデータレジスタ（M）などの内容がクリアされるといった初期化が行われる．そしてOFF状態となることで制御を開始する．また出力ポートには出力リレー（Y）のすべてが記述される．

データレジスタ（D）に対する数値の読出し/書込みは，通常外部機器入出力命令などの応用命令を用いて行うが，データアクセスユニット（タッチパネルなどの表示器）からは直接読出し/書込みを行うことができる．しかし，PLCプログラムからではどのデータレジスタがタッチパネルと接続されているかなどは判別することができない．そのためこうしたユニットとの入出力を行うためには，次のどちらかの方法を用いる．

- 入出力したい信号を指定して外部機器IO命令をPLCプログラムに追記する．
- まず制御回路のVHDL記述中のエンティティ宣言（外部とのインターフェースを記述する部分）に入出力用の信号を追記する．そして，入出力処理に該当する記述部分でその信号と入出力したい内部信号を接続するよう記述する．

これらを自動的に行うための枠組みは今後の課題である．

5　各段の変換方法

本章では，ラダープログラムの段単位での変換方法について述べる．なお，以降の変換例においてレジスタへのクロック入力は特に必要な場合を除いて省略している．また，各段で実行された結果を保持するレジスタはその段の処理が実行状態となった場合のみ動作するが，この動作条件を示すイネーブル信号も省略している．

5.1　条件部

条件部は基本的に AND，OR，NOT からなる論理回路に変換する．図 6 にその例を示す．

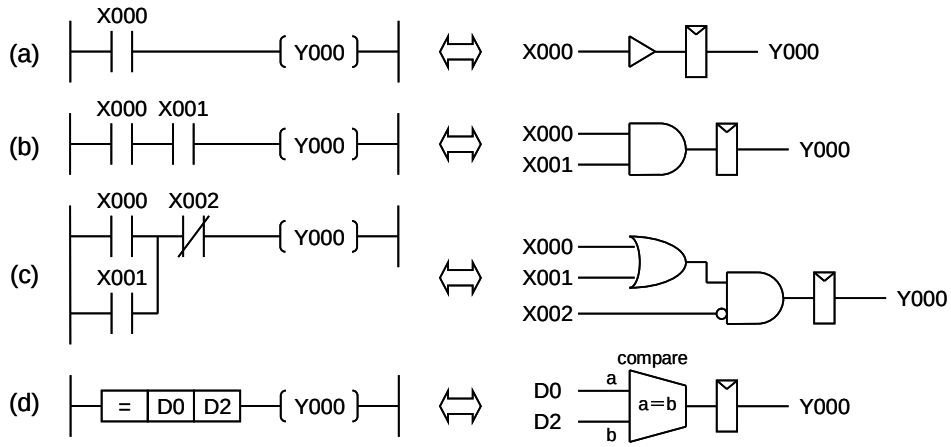


図 6　条件部の変換

図 6（a）は X000 が ON の間 Y000 が ON となる論理である．図 6（b）は X000 と X001 が共に ON の間 Y000 は ON となる論理である．図 6（c）は X000 または X001 が ON の上で，X003 が OFF の間 Y000 が ON となる論理である．また，図 6（d）のように条件部に比較を用いることも可能である．この例では，D0 と D1 の値が同じである間 Y000 は ON となる．

5.2　母線分岐

母線分岐命令（MC，MCR）を用いることでラダー図の母線切り替えを行うことができる．図 7 にこの命令を用いたラダー図の例を示す．

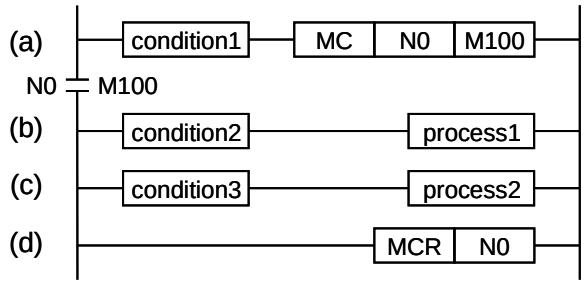


図 7　母線分岐の例

図 7（a）の段では条件部が成立しているとき，母線の切り替えを行う．母線切り替えが行われた場合，図 7（d）の段の母線復帰命令が実行されるまで，その間に記述された図 7（b）（c）の段が順次実行される．これに対し，図 7（a）の段の条件部が成立せず母線の切り替えが行われなかった場合，ラダー図の処理は図 7（d）の母線復帰命令の下の段へ移る．

これは，母線分岐命令が記述してある段の条件部と，母線切り替え後の段の条件部がそれぞれ AND の関係にあることと等価である．そのため図 7 に示すラダー図を，変換ツールでは図 8 に示すラダー図とみなして変換する．

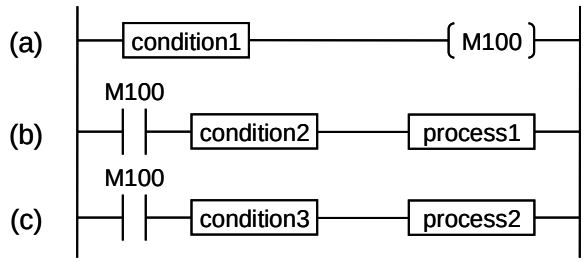


図 8　変換ツールでの母線分岐命令の扱い

5.3　カウンタ

カウンタ（C）指定されたリレーの駆動条件を成立させることで，その回数を加算計数し，これが所定の設定値に達したときにその指定されたリレーを駆動させることができる．このカウンタの現在値はデータ処理命令によって参照することもできる．図 9 にカウンタを用いたラダー図とその変換例を示す．

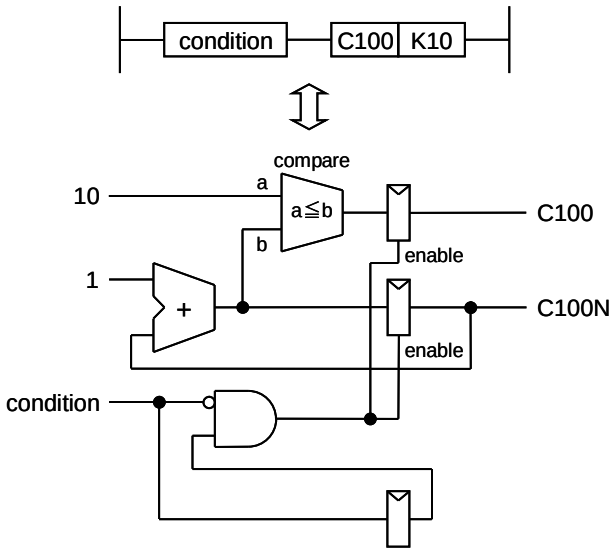


図 9　カウンタの変換

このラダー図では，条件が ON となるたびにカウンタリレー C100 の現在値カウンタを増加させ，その値が設定値 K10 に等しくなることで C100 は ON となる．その後，条件が成立してもカウンタの値は変化しない．そしてリセット命令（RST）が C100 に対して実行されると C100 のカウンタの現在値は 0 にリセットされ，その状態も OFF になる．この様子を図 10 に示す．

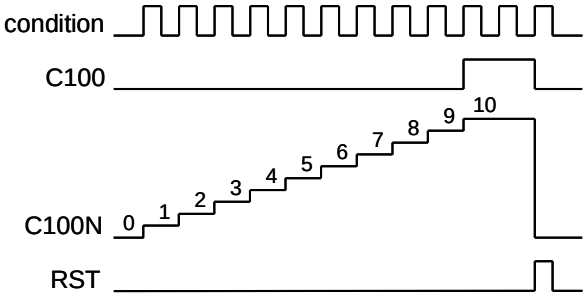


図 10　カウンタの動作

FPGA でも同様で，前スキャン時の条件の状態を保持しておくことで，条件が ON となるたびにカウント値 C100N を増加させる．そして C100N の値が設定値である 10 と等しくなれば C100 の状態を ON にする．以降 C100 に対して RST 命令が実行されるまで C100N の値は変化しない．

5.4 タイマ

タイマ（T）指定されているリレーの駆動条件が成立している間，クロックパルスを加算計数し，これが所定の設定値に達したときにその指定されたリレーを駆動させることができる．図 11 にタイマを用いたラダー図とその変換例を示す．

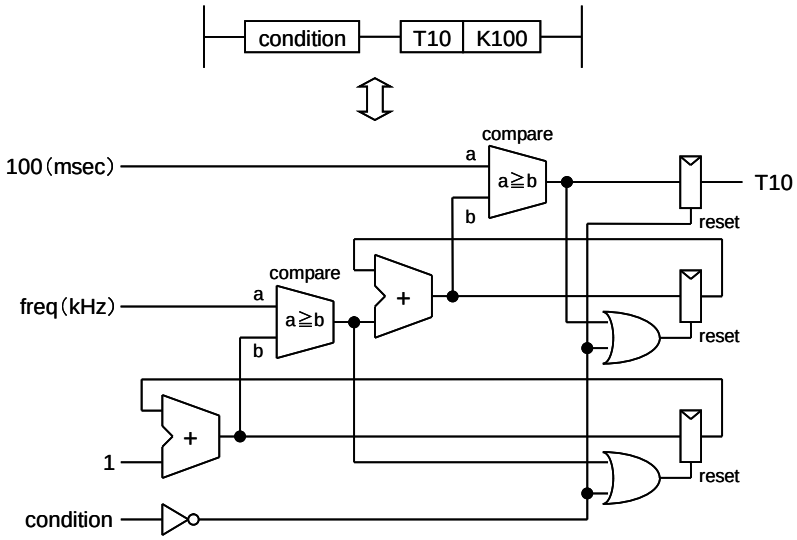


図 11　タイマの変換

このラダー図では，タイマリレー T100 の駆動条件が ON になると T100 用の現在値カウンタは 1ms クロックパルスを加算計数し，その値が設定値 K100 に等しくなることで T100 は ON となる．すなわち，T100 は駆動条件が ON となったあと 100ms で動作する．そして駆動条件が OFF になるとタイマはリセットされ T100 は OFF となる．

FPGA でも同様で，まず T100 の駆動条件が ON となっている間クロックパルスを加算計数し，freq に記述された動作周波数（kHz）の値まで達するとミリ秒単位での現在値を増加させる．そしてその現在値と設定された値（ms）が等しくなったときに T100 の状態を ON にする．駆動条件が OFF になるとカウント値はすべて 0 にリセットされ，T100 の状態も OFF となる．なお，前述のとおり動作周波数は整数型 freq に対して kHz 単位で記述する必要がある．

5.5　動作保持命令

指定したリレーを ON 状態で保持する場合には動作保持命令（SET）を用いる．図 12 にこの命令を用いたラダー図とその変換例を示す.

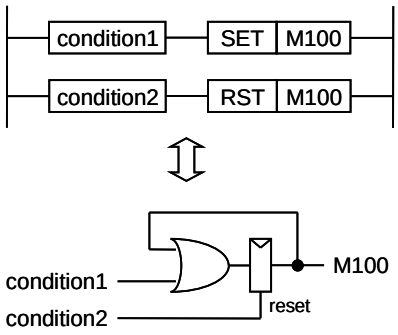


図 12　動作保持命令の変換

このラダー図では，条件 1 が一度 ON になることで，以降これが OFF となっても M100 は ON の状態を保持し続ける．そして，条件 2 が ON となることでリセット命令（RST）が実行され M100 は OFF の状態にリセットされる.

これに対し，FPGA では M100 の状態を自己保持回路で構成する．これによって条件 1 が一度 ON になると以降 M100 は ON の状態を保持し続ける．そして条件 2 が ON となることで M100 の状態は OFF にリセットされる.

5.6　立ち上がり（下がり）微分出力命令

指定したリレー（Y，M のみ）を 1 スキャンの間駆動させる場合，立ち上がり（下がり）微分出力命令（PLS，PLF）を用いる．図 13 にこれらの命令を用いたラダー図とその変換例を示す.

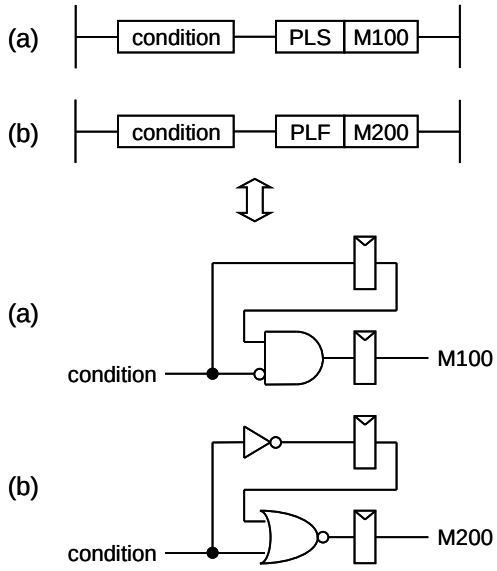


図 13　立ち上がり（下がり）微分出力命令の変換

図 13（a）では，条件部が ON となったあと 1 スキャンタイムの間 M100 が ON となる．図 13（b）では，逆に条件部が OFF となったあと 1 スキャンタイムの間 M200 が ON となる．この様子を図 14 に示す.

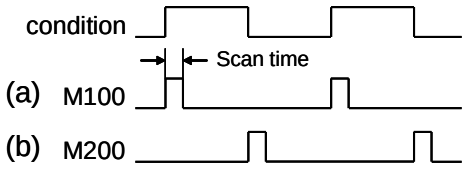


図 14　立ち上がり（下がり）微分出力の動作

FPGA では，カウンタと同様それぞれの段の前スキャン時の条件の状態を保持しておくことで，駆動条件の立ち上がり時（立下り時）を判別し，1 スキャンタイムの間指定したリレー（M100，M200）を ON にする.

5.7　データ転送命令

データレジスタへ値を代入する場合にはデータ転送命令（MOV，DMOV）を用いる．図 15 にこの命令を用いたラダー図とその変換例を示す.

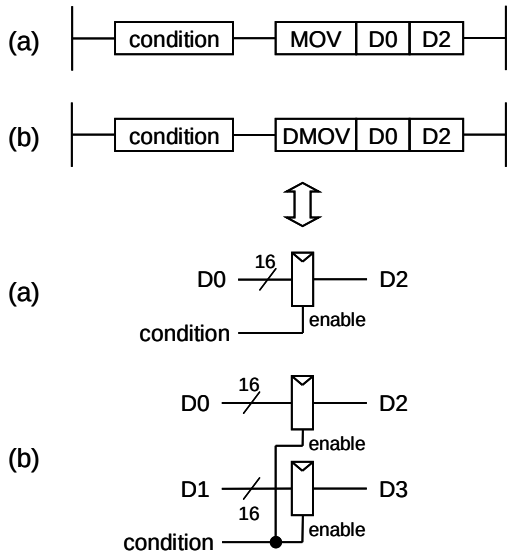


図 15　データ転送命令の変換

図 15（a）は 16 ビット指定でのデータ転送命令である．条件部が ON のとき，D0 の値を D2 へ代入する．また，図 15（b）は 32 ビット指定でのデータ転送命令である．前述のとおり，32 ビット指定されている場合には記述されているデータレジスタとそれに続くデータレジスタが用いられる．そのため，この例では条件部が ON のとき，D0，D1 の値はそれぞれ D2，D3 へ代入される．

定数を代入する場合には，数値をデータレジスタの型であるベクタタイプに変換してから格納する．この変換にはタイプ変換関数 CONV_std_logic_vector() を用いた．条件部が ON のとき，16 進数の “FF” を D0 に代入する記述を図 16（16 ビット指定），図 17（32 ビット指定）に示す．ただし，クロックの入力やフリップフロップの生成についての記述は，後述するためここでは省略する．

```
if (condition = '1') then
  D0 <= CONV_std_logic_vector(16#FF#, 16);
end if;
```

図 16　定数を転送する場合（16 ビット指定）の VHDL 記述

```
if (condition = '1') then
  MOVT := CONV_std_logic_vector(16#FF#, 32);
  D0 <= MOVT(15 downto 0);
  D1 <= MOVT(31 downto 16);
end if;
```

図 17　定数を転送する場合（32 ビット指定）の VHDL 記述

16 ビット指定のデータ転送命令では定数をタイプ変換関数で 16 ビットのベクタタイプへ変換し直接 D0 へ代入する．これに対し，32 ビット指定のデータ転送命令では定数を 32 ビットのベクタタイプへ変換し，一度ベクタタイプの変数 MOVT に代入する．そしてその値を下位 16 ビットと上位 16 ビットに分割し，D0 と D1 にそれぞれ格納する．

5.8　算術演算命令

算術演算命令はそれぞれの命令に対応する演算器に変換する．図 18 に除算命令を用いたラダー図とその変換例を示す．

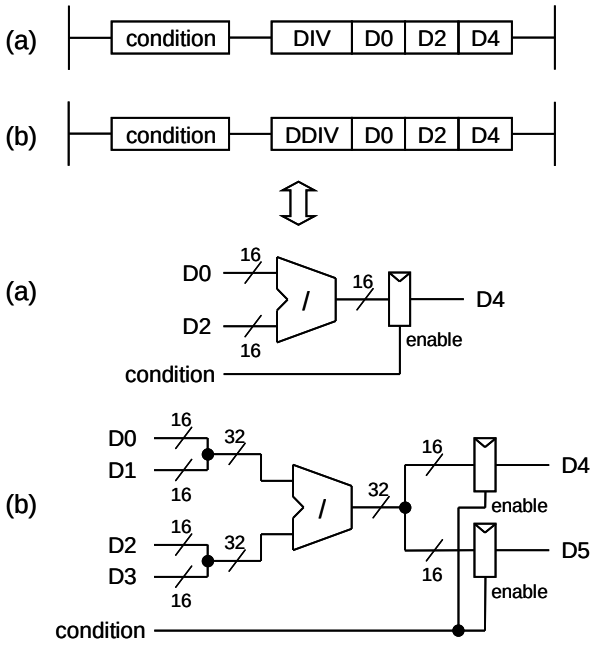


図 18　算術演算命令（除算）の変換

図 18（a）は 16 ビット指定での除算命令である．条件部が ON のとき，D0 の

値を D2 の値で除算して D4 にその結果を格納する．また，図 18（b）は 32 ビット指定での除算命令である．条件部が ON のとき，連結された D0 と D1 の値を，連結された D2 と D3 の値で除算し，その結果を下位 16 ビットと上位 16 ビットに分割してそれぞれ D4，D5 に格納する．

本変換ツールにおいて算術演算の記述には，VHDL 用の算術演算パッケージ “std_logic_arith” 及び “std_logic_unsigned” を利用した演算子記述（‘+’，‘-’，‘*’，‘/’）を用いた[16]．ただし，除算の記述にはデータレジスタの型であるベクタタイプを直接用いることができない．そのため，算術演算時には一時的にベクタタイプから整数型（integer）に変換することで対応した．図 19 に図 18（a）の VHDL 記述を，図 20 に図 18（b）の VHDL 記述を示す．

```
if (condition = '1') then
  CALCA := CONV_INTEGER(D0);
  CALCB := CONV_INTEGER(D2);
  CALCR := CALCA / CALCB;
  D4 <= CONV_std_logic_vector(CALCR, 16);
end if;
```

図 19　図 18（a）の VHDL 記述

```
if (condition = '1') then
  CALCA := CONV_INTEGER(D1 & D0);
  CALCB := CONV_INTEGER(D3 & D2);
  CALCR := CALCA / CALCB;
  CALCT := CONV_std_logic_vector(CALCR, 32);
  D4 <= CALCT(15 downto 0);
  D5 <= CALCT(31 downto 16);
end if;
```

図 20　図 18（b）の VHDL 記述

変数のうち CALCA，CALCB，CALCR は 32 ビットの整数型，CALCT は 32 ビットのベクタタイプである．まず計算する 2 つの入力値を CALCA と CALCB

に代入する．このとき，データレジスタの値はタイプ変換関数CONV_INTEGER()を用いてベクタタイプから整数型へ変換してから代入する．32 ビット指定がされている場合には，2 つのデータレジスタを連結してから変換し代入する．次に四則演算記号を用いて算術演算を行い，その結果を CALCT に代入する．そして，16 ビット指定であれば CONV_std_logic_vector() を用いて今度は整数型から 16 ビットのベクタタイプに変換しデータレジスタに値を格納する．32 ビット指定がされている場合には，同様に一度 32 ビットのベクタタイプに変換し CALCT に代入する．その後下位と上位 16 ビットずつに分割し 2 つのデータレジスタに値を格納する．

この記述では，16 ビット指定，32 ビット指定のどちらでも一度 32 ビットの整数型に変換している．しかし，実際には演算器に入力される値のビット幅やその値（定数の場合）を論理合成ツールが解析し，各演算命令に応じて最適な演算器が生成される．

5.9 外部機器入出力

外部機器との入出力には外部機器入出力命令（TO，FROM）が用いられ，PLC ではこの命令がプログラム中にあった場合，外部機器の BFM を介して即座に入出力が行われることを説明した．そこで FPGA では，この命令がプログラム中にあった場合，外部機器の BFM に見立てたメモリに対して，シングルポート RAM へアクセスすることと同様に実装する（図 21）．複数の外部機器にはそれぞれの機器の番号が割り振られているため，この外部機器番号及びそれぞれの機器の BFM アドレスを書込み/読込み先のメモリアドレスとして用いる．ただし，外部機器番号は PLC に近い順番に割り振られているだけで，PLC プログラムからではその外部機器がどういったものかが判別できないため，外部機器の制御回路の仕様によっては入出力先のメモリアドレスを手動で書き換える必要がある．入出力するデータのビット幅は 32 ビットとし，命令に 32 ビット指定がされていない場合には下位 16 ビットのみ使用してデータの転送を行う．この 16/32 ビット指定の判別は，書込み制御信号（ライトイネーブル）に 1 ビットの判定用ビットを付け足してメモリ側で行っている．そのため 32 ビット指定がされている状態でメモリへの書込みが行われると，指定されているメモリアドレスとそれに連続したアドレスに対してデータを下位 16 ビットと上位 16 ビットに分割して書込みを行う．本メモリはクロックを用いた同期式のメモリである

ため書込み/読込みを行う際にはそれぞれ 1 クロック必要となる．そのため外部機器入出力命令があった場合，メモリへの入出力を行うためのステートが必要である．このような実装方法では，外部機器への入出力の順序が保たれるという利点がある．

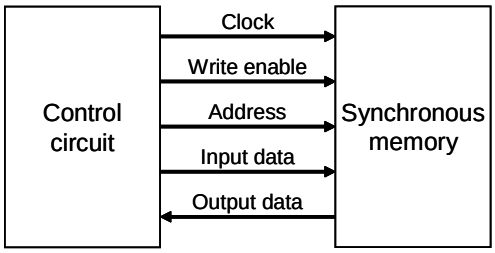


図 21　同期式メモリへのアクセス

5.10　その他応用命令

データ転送命令，算術演算命令，外部機器入出力命令以外には次に示す応用命令が変換できる．これらの命令はこれまで説明した命令の応用形であるため，変換例は省略する．

- インクリメント命令（INC）
  データレジスタの値をインクリメントする．図 22 では，条件部が ON のとき D0 に 1 を加算する．

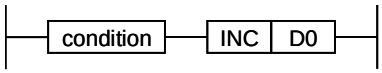


図 22　インクリメント命令

- 一括転送命令（BMOV）
  連続した複数のデータレジスタの値を一括で転送する．図 23 では，D0 から連続する 3 つのデータレジスタの値（D0～D2）を D5 から連続する 3 つのデータレジスタ（D5～D7）へ転送する．

図 23　一括転送命令

- 一括リセット命令（ZRST）
  2 つのデータレジスタ間の値をすべて 0 にリセットする．図 22 では，D0 から D10 まで連続するデータレジスタの値を 0 にリセットする．

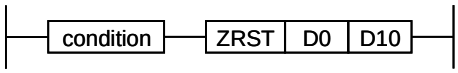


図 24　一括リセット命令

- 論理積演算命令（WAND）
  データレジスタの各ビットの論理積をとる．図 22 では，D0 と D2 の各ビットの論理積をとって，その結果を D4 に格納する．

図 25　論理積演算命令

- 左回転命令（ROL）
  データレジスタのビットを定数値分左回転する．図 22 では，D0 を 4 ビット左シフトし，オーバーフローした 4 ビットを下位 4 ビットに格納する．

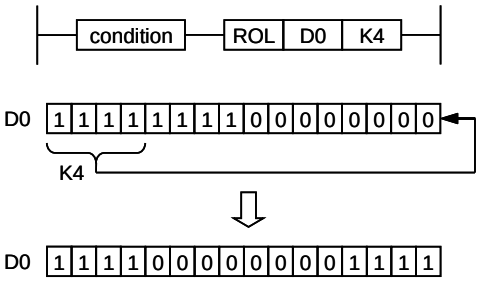


図 26　左回転命令とその動作

- ASCII → HEX 変換命令（HEX）
  データレジスタの ASCII コードを 16 進数へ変換する．図 22 では，D0 の上下各 8 ビットに格納されている ASCII コードを 16 進データに変換して，2 文字分 D2 へ転送する．この場合 D0 に格納された「A」と「3」の ASCII コード 41_H と 33_H を，16 進データの A_H と 3_H に変換して，1 文字分（16 進なので 4 ビット）ずつ D2 の下位ビットから埋めていく．

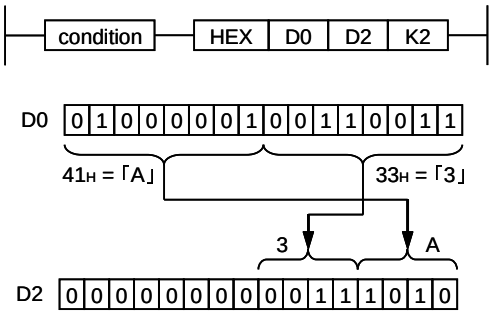


図 27　ASCII → HEX 変換命令とその動作

6　PLC プログラム全体の変換手法

本章では PLC プログラム全体の変換手法について述べる．

6.1　Sequential Design

ラダー図の実行制御方式をそのままハードウェア化するには，ラダー図の各段に順序回路の 1 状態を割り当てて，各段を逐次処理すればよい．この変換方法を Sequential Design（SD）と呼ぶ．そのため，SD では 1 クロックごとにラダーの 1 段分の処理を実行する（図 28）．この変換手法をブロック図で表すと図 29 になる．SD で 1 回のスキャンに必要な状態遷移数は，ラダー図に含まれる段の数 ρ に，一括入出力方式における入力処理及び出力処理用の状態を加えた $\rho+2$ となる．1 状態を 1 クロックで実現すれば，1 スキャンあたりのクロック数は $\rho+2$ となる．

この設計方法のベースとなる VHDL 記述を図 30 に示す．クロック同期で状態遷移を行い，1 段ずつ順番に処理する．代入には基本的に信号代入を用いる．

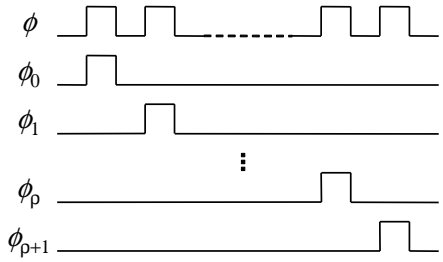


図 28　Sequential Design のクロックオーバーレイ

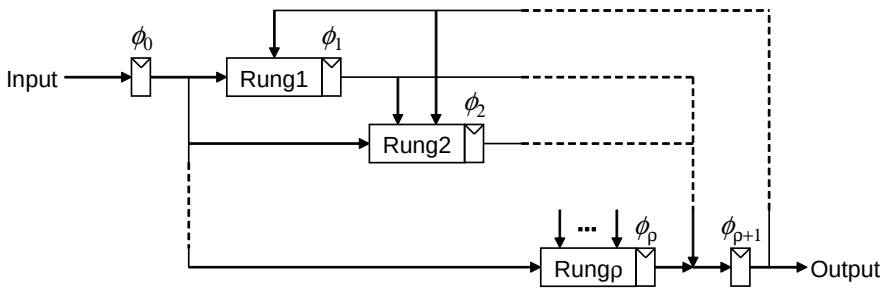


図 29　Sequential Design のブロック図

```
<ライブラリ宣言，パッケージ呼び出し>
<エンティティ宣言>
architecture RTL of SEQUENCE is
<信号宣言>
begin
  process (CLK)
  <変数宣言>
  begin
    if (CLK'event and CLK = '1') then
      if (RST = '1') then
        <信号・変数の初期化>
      else
        case STATE is
          when 0 =>
            <入力処理>
            STATE <= STATE + 1;
          when 1 =>
            <1 段目の処理>
            STATE <= STATE + 1;
          when 2 =>
            <2 段目の処理>
            STATE <= STATE + 1;
          ...
          when ρ =>
            <ρ 段目の処理>
            STATE <= STATE + 1;
          when ρ+1 =>
            <出力処理>
            STATE <= 0;
          when others =>
            STATE <= 0;
        end case;
      end if;
    end if;
  end process;
end RTL;
```

図 30　SD の VHDL 記述

図 31 に簡単な計算をする PLC プログラムを示す.

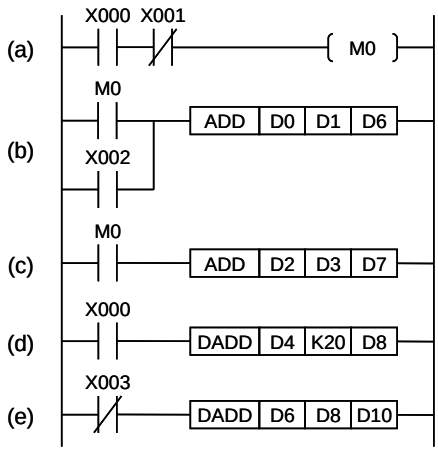


図 31　サンプルプログラムのラダー図

このサンプルプログラムは（a）〜（e）の 5 つの段から構成され，次に示す一連の処理を順次実行する.

- （a）：X000 が ON，X001 が OFF のとき，M0 は ON になる．それ以外では M0 は OFF になる.
- （b）：M0 もしくは X002 が ON のとき，D0 + D1 → D6 の 16 ビット計算を行う.
- （c）：M0 が ON のとき，D2 + D3 → D7 の 16 ビット計算を行う.
- （d）：X000 が ON のとき，D4 + 20 → D8 の 32 ビット計算を行う.
- （e）：X003 が OFF のとき，D6 + D8 → D10 の 32 ビット計算を行う.

なお，ここでは説明のためデータレジスタの番号に奇数を用いているが，16/32 ビット演算が混在する場合は特に，保守性の面から偶数番号のみを使うようにする．ただし奇数番号を用いても PLC 及び本変換ツールでの実装上の問題は無い.

このサンプルプログラムを SD で変換したときの回路を図 32 に示す．ここでは，データレジスタの入力に D0〜D5 を，出力に D10，D11 を用いた．本変換ツールではこの回路を生成するための VHDL 記述を生成しているが，実際には論理合成ツールの最適化により，制御論理が等価な異なる回路が生成される．このサンプルプログラムの命令列及びそれから生成した VHDL 記述については付録 A.3 を参照のこと.

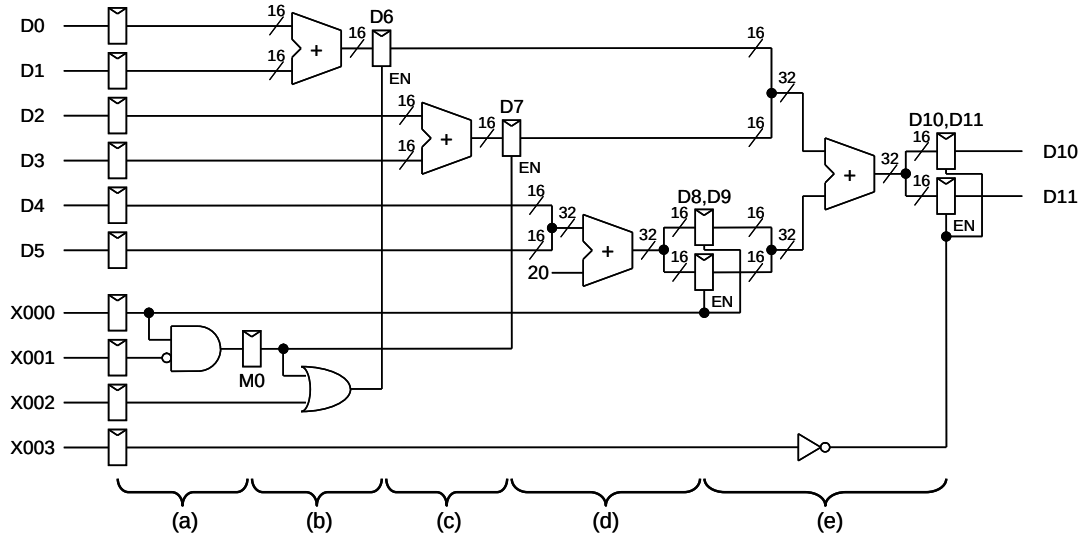


図 32　サンプルプログラムを SD で変換したブロック図

この回路の動作について述べる．SD では以下に示す処理に 1 クロックずつ割り当てて逐次実行する．

(1)　入力処理を行う．すべての入力ピンの状態（入力リレー及び D0〜D5）をレジスタに取り込む．

(2)　（a）の段の処理を行う．X000 が ON，X001 が OFF のとき，M0 のレジスタは ON 状態を保持する．それ以外では OFF 状態を保持する．

(3)　（b）の段の処理を行う．M0 か X002 が ON のとき，D0 と D1 の値が加算され D6 のレジスタに保持される．

(4)　（c）の段の処理を行う．M0 が ON のとき，D2 と D3 の値が加算され D7 のレジスタに保持される．

(5)　（d）の段の処理を行う．X000 が ON のとき，連結された D4 と D5 の値が定数 20 と加算され D8 及び D9 のレジスタにそれぞれ保持される．

(6)　（e）の段の処理を行う．X003 が OFF のとき，連結された D6，7 の値と D8，9 の値が加算され D10，11 のレジスタにそれぞれ保持される．

(7)　出力処理を行う．出力用レジスタの状態（D10 と D11）を出力ピンに送る．

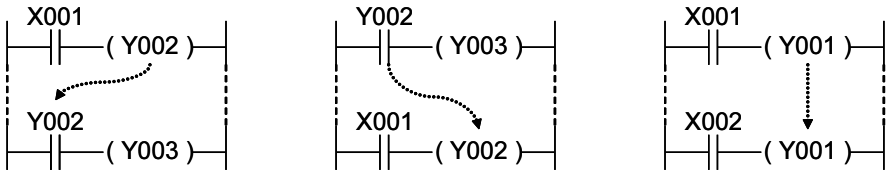
以上により，この回路では 7 クロックかけて 1 スキャンの処理を行う．

6.2　Levelized Design

SD を高速化するためには，複数の段を同じ 1 状態に割り当てて，状態遷移数を減らす必要がある．そこで本節では，並列実行可能な制御論理を並列に処理

する設計 Levelized Design（LD）について検討する．

命令列の実行結果を SD と同じにするためには，各段の間の依存関係を適切に維持する必要がある．依存関係の例を図 33 に示す．ある段の出力がそれより下位の段において参照されることをデータ依存（図 33(a)）と呼ぶ．上の段の出力が定まってから下の段を実行する必要があるので，下の段は上の段より後のクロックで実行しなければならない．同様に，前の段の入力がそれより後の段において上書きされることを逆依存（図 33(b)）と呼び，ある段の出力がそれより下位の段において上書きされることを出力依存あるいはダブルコイルと呼ぶ（図 33(c)）．いずれの場合も，元のラダー図で上位の段を下位の段より先に実行しなければならない．



(a) データ依存　　(b) 逆依存　　(c) 出力依存

図 33　依存関係の例

これらの依存関係を全ての段において維持しつつ，各段を実行可能な最も早い状態に割り当てて処理を行なう．具体的には，依存関係に従って各段にレベル付けを行い，レベルの低い段から順番に処理する．この考え方は論理シミュレーションにおける Levelized Compiled Code simulation[17] と全く同じである．

まず，一括入出力方式における入力処理のレベルを $Level_0$，依存関係による制約のない段をレベル $Level_1$ に割り当てる．$Level_i$，$Level_j$，... からの依存関係を持つ段は，$Level_{max(i,j,...)+1}$ に割り当てる（図 34）．全ての段を割り当てた後，最も深いレベルが $Level_\lambda$ であれば，出力処理のレベルを $Level_{\lambda+1}$ に割り当てる．こうして得られる LD のブロック図を図 35 に示す．LD の 1 スキャンに必要な状態遷移数（クロック数）は，$\lambda + 2$ である．

この設計方法のベースとなる VHDL 記述を図 36 に示す．各段の依存関係を維持するようにして，1 つの状態に複数の処理を記述する．SD と同様，代入には基本的に信号代入を用いる．

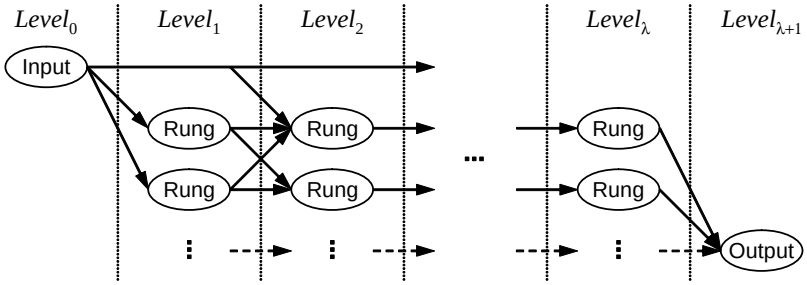


図 34　Levelized Design の概要

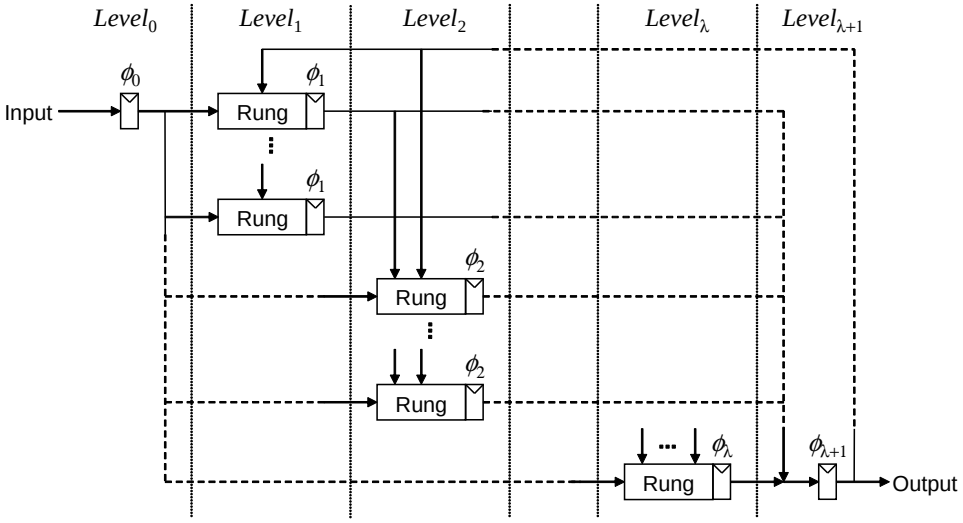


図 35　Levelized Design のブロック図

```
...
  case STATE is
    when 0 =>
      <入力処理>
      STATE <= STATE + 1;
    when 1 =>
      <レベル 1 の段の処理>
      <レベル 1 の段の処理>
      ...
      STATE <= STATE + 1;
    when 2 =>
      <レベル 2 の段の処理>
      <レベル 2 の段の処理>
      ...
      STATE <= STATE + 1;
    ...
    when λ =>
      <レベルλの段の処理>
      <レベルλの段の処理>
      ...
      STATE <= STATE + 1;
    when λ+1 =>
      <出力処理>
      STATE <= 0;
    when others =>
      STATE <= 0;
    end case;
  ...
```

図 36　LD の VHDL 記述

図 31 のサンプルプログラムを LD で変換したときの回路を図 37 に示す.

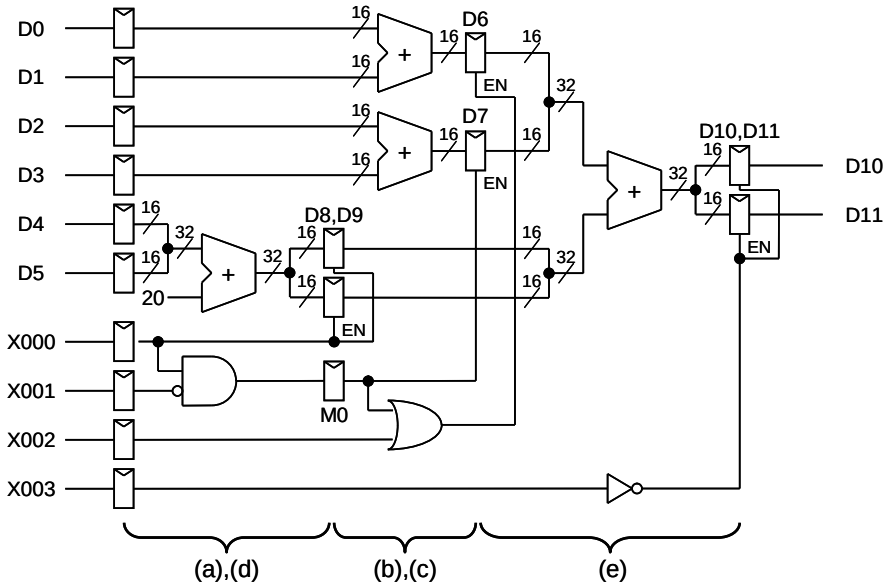


図 37　サンプルプログラムを LD で変換したブロック図

この回路の動作について述べる．LD ではまずはじめに各段にレベル付けを行う．このサンプルプログラムでは,（a）の段に対して（b）,（c）の段がデータ依存の関係に,（b）,（c）及び（d）の段に対して（e）の段が出力依存の関係にある．この依存関係に従ってレベル付けし，同じレベルの段を 1 クロックで並列に実行する．以下の番号はそのレベルに相当し，各レベルに 1 クロックずつ割り当てられる．

（1）　入力処理を行う．

（2）　（a）及び（d）の段の処理を行う．条件が ON のとき M0 及び D8，9 のレジスタが更新される．

（3）　（b）及び（c）の段の処理を行う．条件が ON のとき D6 及び D7 のレジスタが更新される．

（4）　（e）の段の処理を行う．条件が ON のとき D10，11 のレジスタが更新される．

（5）　出力処理を行う．

以上により，この回路では 5 クロックかけて 1 スキャンの処理を行う．よってこのサンプルプログラムでは，LD により設計することで SD と比較して 1 スキャンあたりの状態遷移数が 2 減少する．

6.3 Flat Design

LD では依存性に従って各段を処理しているが，各レベルは 1 クロックを使って処理されている．しかし実際には，依存性さえ守れば各レベルに 1 クロックを割り当てる必要はない．LD のレベル間の記憶素子を除去してラダー全体を組合せ回路として生成し，毎スキャンの終了時に入出力の更新を行うように変換できる．この設計を FlatDesign（FD）と呼ぶ（図 38）．

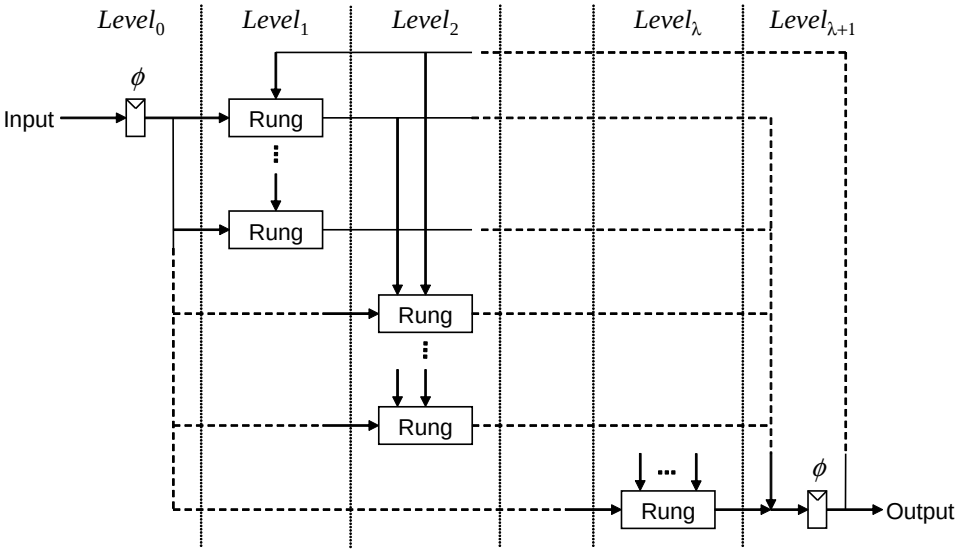


図 38　Flat Design のブロック図

FD では，$Level_j$ への入力は $Level_i(0 \leq i < j)$ から直接配線で与えられる．そして $Level_j$ の段で処理した結果は記憶素子に格納されることなく下位のレベルへ直接渡される．また，SD や LD では入力処理及び出力処理に各 1（合計 2）状態を与えていたが，FD ではこれらも統合する．結果として，FD は 1 スキャンを 1 クロックで実行することができる．レベル間にまたがった論理最適化が行なわれるため，論理規模と遅延時間の両面で効果が期待できる．ただし FD であっても，外部入出力命令が存在する場合は，そのための状態を設けなければならないので，そのぶん 1 スキャンに必要なサイクル数は増加する．

ここで，LD から FD へ変更することによる全体の実行時間の短縮について述べる．図 39 にその例を示す．矢印は段の依存関係を示している．各段には，1 ビットの ON/OFF のみを行う実行時間の短い処理（薄い灰色の部分）や，演算処理といった比較的実行時間の長い処理（濃い灰色の部分）が含まれる．LD で

は，このうち最も実行時間の長い処理によってクロックサイクル時間が決められる．そのため例え実行時間の短い処理しか 1 つの状態に割り当てられていない場合でも，その状態は 1 クロック与えられるため 1 クロックサイクル時間が必要である．これに対し，FD では依存関係さえ守っていれば，実行可能な段を次々に処理することができるため，その LD における無駄な待ち時間を削除することができる．また LD では，各レベルで段が実行されるとその結果を保持するためにレジスタの更新を毎クロックする必要がある．これに対し FD では，すべての処理が終了したときのみレジスタの更新をすればよい．さらにここでは省略したが，LD では入出力処理用の状態が割り当てられるためその分全体の実行時間は長くなる．

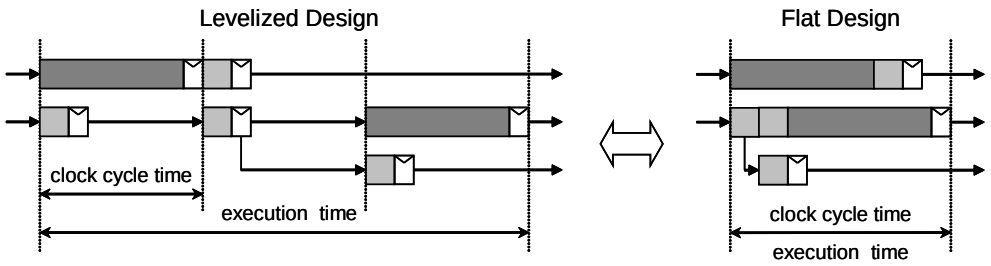


図 39　FD による実行時間短縮の例

この設計方法のベースとなる VHDL 記述を図 40 に示す．これまでの設計と異なり，代入にはすべて変数を用いた即時実行をすることで 1 クロックでの実行が可能である．外部機器 IO 命令がある場合にはこれまでの設計方法と同様に信号 STATE を用いることで状態遷移を行って入出力を行う．

```
...
architecture RTL of SEQUENCE is
begin
  process (CLK)
  <変数宣言>
  begin
    if (CLK'event and CLK = '1') then
      if (RST = '1') then
      <変数の初期化>
      else
        <入力処理>
        <1 段目の処理>
        <2 段目の処理>
        ...
        <ρ段目の処理>
        <出力処理>
      end if;
    end if;
  end process;
end RTL;
```

図 40　FD の VHDL 記述

図 31 のサンプルプログラムを FD で変換したときの回路を図 41 に示す．なお，視認性の面からここでは 32 ビット演算における 2 つのデータレジスタの連結記述を一部省略した．実際にはこれまでのブロック図と同様に連結処理を行っている．

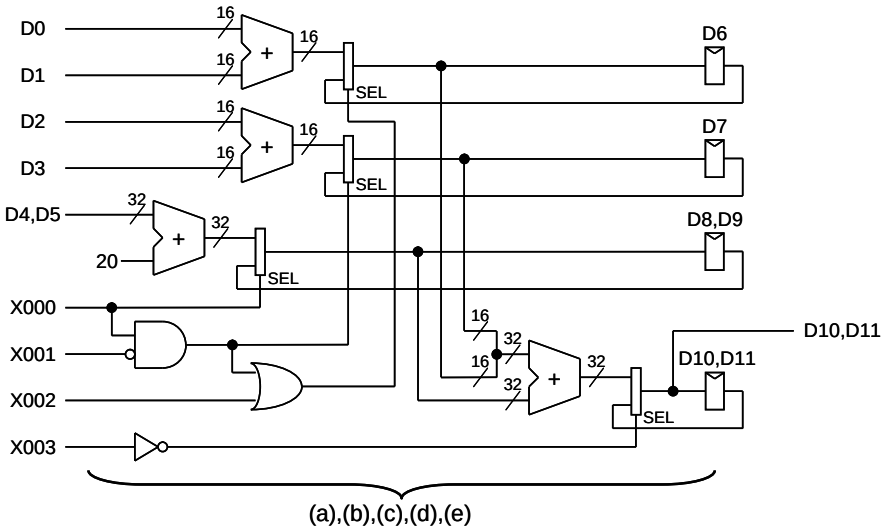


図 41　サンプルプログラムを FD で変換したブロック図

この回路の動作について述べる．LD では各レベルで段が実行されるとその結果はすべてレジスタに保持され以降のレベルに渡されていた．しかし FD では段が実行された後の結果を直接以降の段に渡す．そのため，毎スキャン終了時の最終的な結果のみがレジスタに保持される．ここで，次以降のスキャンにも結果が用いられる D6〜D11 はレジスタが生成されるが，毎スキャン状態が更新される（正確には毎スキャン時の条件部の状態を表す）M0 はレジスタが生成されない．条件部が成立したときのみ結果が更新される（b）〜（e）の段の処理ではマルチプレクサが用いられ，その条件部の状態をセレクト信号として，更新された結果か，更新される前の結果かが選択される．また，入出力処理のための状態は割り当てられないため，クロック毎にすべての入力ピンの状態を直接内部の処理で使用し，その結果を出力ピンへ送る．

以上により，この回路はラダーを組み合わせ回路として構成しているが，全体として見ると状態が 1 つしかない順序回路といえる．このため 1 スキャンを 1 クロックで実行可能である．

6.4　資源制約

SD では各サイクルに 1 段しか動作しないにもかかわらず，演算命令数と同数の演算器を生成するのは無駄である．LD でも同様に，各サイクルには 1 レベルしか動作しないので，全ての演算器が同時に動作するわけではない．同時に動作しない演算器は，マルチプレクサやバスで共有することにより，資源の膨張

を抑止できる可能性がある．

本変換ツールでは大規模な制御論理向けに，資源制約を行うオプションを用意した．演算器の生成数に上限を設け，共有して使用することにより，回路規模の増加を抑制する（図 42）．同時に実行可能な段の数に対して演算器数が不足する場合は，リストスケジューリングにより状態に段を割り当てる[18]．そのため本変換ツールでは優先度の高い段から演算器への割り当てを行う．

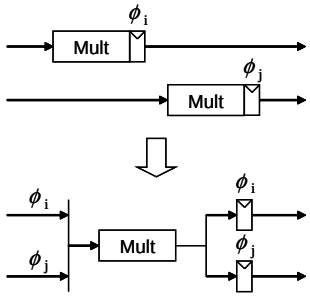


図 42　演算器の共有

この設計方法のベースとなる VHDL 記述を図 43 に示す．ここでは，乗算器を 1 つ用いた場合を示している．対応するステートで計算される値を先に記述しておき，そのステートにきたとき演算を行い結果を返す．16bit 演算でも 32bit 演算でも同じ演算器を用いるため，記述されている中で最も大きい bit 幅の演算器が生成される．

```
...
architecture RTL of SEQUENCE is
  <信号宣言>
begin
  MULA1 <=
    CONV_INTEGER(D0) when STATE = 1 else
    ...
    CONV_INTEGER(D3 & D2) when STATE = 20 else
    0;
  MULB1 <=
    10#10# when STATE = 1 else
    ...
    CONV_INTEGER(D5 & D4) when STATE = 20 else
    0;
  MUL1 <= MULA1 * MULB1;

  process (CLK)
  ...
    when 1 =>
      if (condition = '1') then
        D1 <= CONV_std_logic_vector(MUL1, 16);
      end if;
      ...
    when 20 =>
      if (condition = '1') then
        CALCT := CONV_std_logic_vector(MUL1, 32);
        D6 <= CALCT(15 downto 0);
        D7 <= CALCT(31 downto 16);
      end if;
    ...
```

図 43　演算器（乗算）を共有したときの VHDL 記述

図 31 のサンプルプログラムを SD で演算器の最大生成数を '1' として変換したときの回路を図 44 に示す.

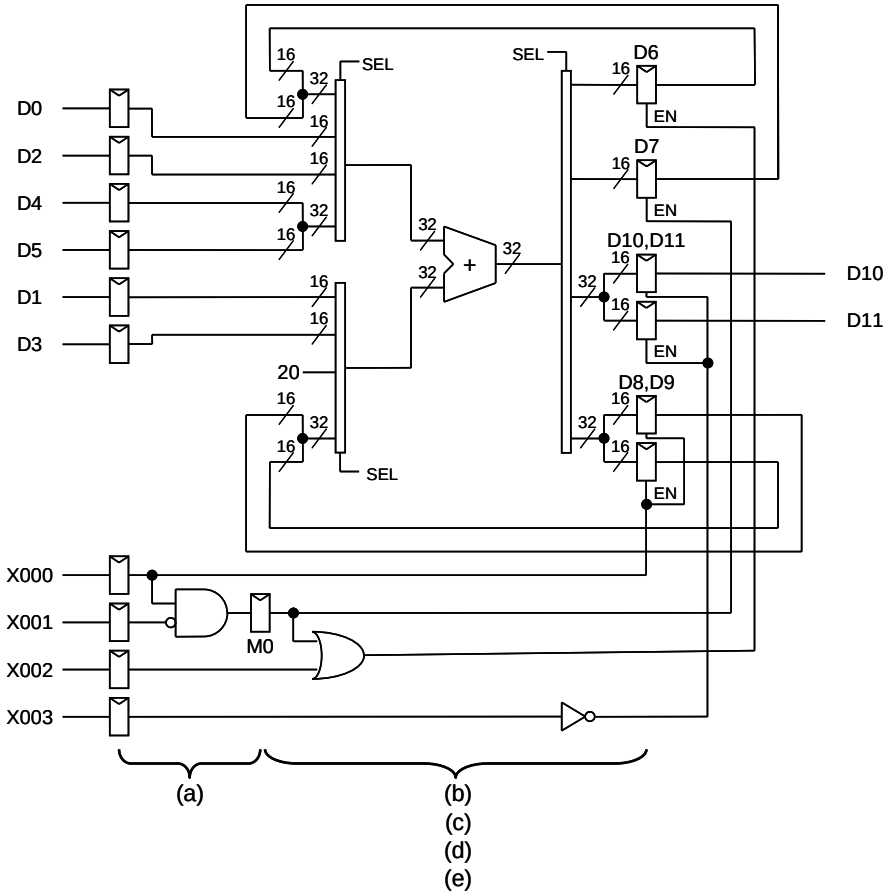


図 44　サンプルプログラムに資源制約を行ったブロック図

この回路の動作について述べる．各クロックでの処理内容は図 32 で説明した回路と同じである．そのため 1 スキャンを 7 クロックで実行する．ただし，この回路では加算器を 1 つだけ生成しこれを共有して使用するため，演算器の前後にマルチプレクサが生成される．このマルチプレクサのセレクト信号には，加算を行う段の実行状態（ステート）が割り当てられ，実行状態にある段の計算値（加算される 2 つの値）と計算結果の出力先が選択される．演算命令と同数の演算器を生成していたこれまでの回路では，16 ビット演算が行われる場合にはビット幅が 16 の演算器を生成していたが，このように 16/32 ビットの演算命令が混在し共有される場合には前述のとおりビット幅の大きい 32 ビットの演算器が生成される．

7　評価

変換ツールの評価を行った．本章ではその評価結果とそれに対する考察を示す．

7.1　評価環境

評価には次の 3 つのサンプルを用いた．各サンプルの総命令数と演算命令数及びその演算幅を表 3 に示す．

- PID 制御プログラム
  自動制御において広く利用されているフィードバック制御の一種のプログラムである．
- A 社サンプルプログラム
  ある会社（A 社）で用いられている演算処理を主とした制御プログラムである．このサンプルはもともと三菱電機の PLC A シリーズの命令で記述されいたため池田[8] が FX2N 用の命令に書き直したものを用いた．
- 八洲熱学サンプルプログラム
  八洲熱学株式会社と共同開発中の整列巻取機の制御プログラムである．本プログラムのみ外部機器 IO 命令による入出力がある．タッチパネルとの入出力のために外部機器 IO 命令を追加して評価を行った．

表 3　評価用サンプルの総命令数と演算命令の内訳

評価サンプル	総命令数	演算命令			
		加減算命令	乗算命令	除算命令	演算幅
PID 制御	21	4	3	0	32bit
A 社	165	6	12	9	32bit
八洲熱学	1351	5	11	2	16/32bit

ターゲットデバイスには Altera 社の Stratix II 及び APEX20KE を用いた．それぞれのデバイスにおける評価環境を表 4 に示す．論理合成ツール（Quartus II）のバージョンがデバイスごとに異なるのは，ターゲットデバイスを APEX20KE として論理合成を行った場合に，バージョン 6.0 SP1 では内部エラーが発生したためである．そのため APEX20KE をターゲットデバイスとした場合には，論理合成可能なバージョン 5.0 を用いた．

表 4　論理合成環境

Family	Stratix II	APEX20KE
Device	EP2S60F672C5ES	EP20K600EBC652-3
Total ALUTs/LEs	48,352（ALUTs）	24,320（LEs）
Total pins	493	488
Total memory bits	2,544,192	311,296
DSP block elements	288	–
Quartus II Version	6.0 SP1	5.0
Optimization	Balanced（default）	
OS	WindowsXP SP2	Windows2000 SP4
CPU	AthlonXP 2600+	Athlon64 3500+
MEM	1GB	2GB

演算器を共有した場合の評価では，各演算器（加減算器，乗算器，除算器）の最大生成数を設定して評価を行った．次節以降の評価結果では，この最大生成数を n として演算器を共有する回路を “共有 $\times n$” と示す．これに対し，演算命令 1 つに演算器を 1 個生成する回路を “固有” と示す．また，FPGA 上の実行時間（スキャンタイム）は，各回路のクロックサイクル時間と 1 スキャンあたりの状態遷移数の積で求めた．そして PLC 上での実行時間はすべての段の条件部で条件が成立したときの時間（worst case）とした．この PLC 上での実行時間の見積もりには池田が作成した PLC 実行時間計算ツール[8] を用いた．ただし，そのままでは今回評価に用いたサンプルの実行時間を見積もることはできなかったため，対応する命令や要素を追加して使用した（付録 A.1）．

7.2　Stratix II を用いた評価

Stratix II をターゲットデバイスとして評価を行った．表 5 は PID 制御プログラムの評価結果である．SD（固有）では，FX2N PLC での実行時間に対して 7189 倍高速化し，論理規模は ALUT で 1%，DSP block で 8%程度と非常に小さい．LD（固有）では SD（固有）に対して 1.4 倍高速化し，FD では SD（固有）に対して 6.9 倍高速化した．LD（固有），FD 共に論理規模は SD（固有）とほとんど同じである．LD（固有）の高速化が SD（固有）に対して小さいが，これは PID 処理の命令依存性が大きく，レベル付けによる並列化を行っても状態遷移数は最大で 25%しか減少しないためである．演算器を共有した場合，DSP Block は減少するが，論理規模（ALUT）は SD（固有）に対して SD（共有 $\times 1$）で 1%，LD（固有）に対して LD（共有 $\times 1$）でも 1%程度しか減少しない．こ

れは本プログラムでは論理規模に対する乗算器の占める割合が大きく，これがDSP Blockで構成されるためである．

表6はA社サンプルプログラムの評価結果である．まずLD（固有）ではSD（固有）に対して5.8倍高速化し，FDではSD（固有）に対して43.7倍の高速化を達成した．本プログラムは複数の出力に対して制御を行っているため，処理の並行度が高く，レベル付けによる並列化で状態遷移数が大きく減少する（最大で84%減少）．そのためLDの効果がPID制御プログラムよりも大きい．また演算器を共有した場合，共有している演算器数に比例してDSP Blockが減少するとともに，ALUTはSD（固有）に対してSD（共有 ×1）で51%，LD（固有）に対してLD（共有 ×1）で52%減少する．これは論理規模の大きい除算器がALUTで構成され共有されるためである．

表7は，八洲熱学サンプルプログラムの評価結果である．まずLD（固有）ではSD（固有）に対して5.3倍高速化した．これはA社のプログラムと同様に並列度の高い制御を行っているためである．FDではSD（固有）に対して4.6倍高速化したが，LD（固有）に対して14%低速となった．これは外部機器IO命令が多いため，入出力用の状態遷移が必要で，FDでも状態遷移数がほとんど変わらないためである．さらに，LDでは1状態の処理を並列に実行するのに対して，FDでは依存関係にある処理を組み合わせ回路として実行するため，動作周波数がLDよりも低くなるためである．また演算器を共有した場合，DSP Blockは減少するが，ALUTはSD（固有）に対してSD（共有 ×1）で9%，LD（固有）に対してLD（共有 ×1）で13%増加してしまった．これは，共有のためのマルチプレクサや，汎用的演算器が生成されることが原因であると考えられる．本プログラムのように演算の占める割合が少なく，計算する値に定数が多く含まれる場合，専用演算器を生成する方が論理規模は小さくなることがある．

なお評価において，演算器の共有数は固有に生成した場合と状態遷移数が変わらなくなるまでとした．これは，共有している演算器数をそれ以上増加させても状態遷移数は減少しないからである．ただし状態遷移数が同じ場合でも，固有に生成した回路と共有して生成した回路では演算器数が同じとは限らない．例えば，表6で乗算器が割り当てられるDSP Blockの使用量は，LD（固有）で56であるのに対してLD（共有 ×4）では32と減少している．しかし，表5から表7に示したすべてのサンプルにおいて，LDの状態遷移数が同じ“固有”と“共有”の回路では，“共有”の回路の回路の方が動作周波数は低下し，論理規模

もほとんど変わらないか増加してしまっている．これは表 7 の考察にもあるように，共有のためのマルチプレクサや，汎用的演算器が生成されることが原因であると考えられる．

表 5　PID 制御プログラムの評価結果（StratixII）

デバイス	設計	演算器	状態遷移数	動作周波数 (MHz)	論理規模 (ALUTs)	メモリ (bits)	DSP block	実行時間 (sec.)
PLC	—	—	—	—	—	—	—	7.98×10^{-4}
FPGA	SD	固有	12	108.57	486	0	24	1.11×10^{-7}
		共有× 1	12	87.63	479	0	8	1.37×10^{-7}
	LD	固有	9	113.51	480	0	24	7.93×10^{-8}
		共有× 1	9	88.32	475	0	8	1.02×10^{-7}
	FD	固有	1	61.90	448	0	24	1.62×10^{-8}

表 6　A 社サンプルプログラムの評価結果（StratixII）

デバイス	設計	演算器	状態遷移数	動作周波数 (MHz)	論理規模 (ALUTs)	メモリ (bits)	DSP block	実行時間 (sec.)
PLC	—	—	—	—	—	—	—	1.61×10^{-3}
FPGA	SD	固有	74	8.47	5,850	1,280	56	8.74×10^{-6}
		共有× 1	74	6.50	2,859	1,280	8	1.14×10^{-5}
	LD	固有	12	8.00	5,681	0	56	1.50×10^{-6}
		共有× 1	17	6.81	2,704	0	8	2.50×10^{-6}
		共有× 2	14	6.48	3,961	0	16	2.16×10^{-6}
		共有× 3	13	6.35	5,010	0	24	2.05×10^{-6}
		共有× 4	12	6.57	6,309	0	32	1.83×10^{-6}
	FD	固有	1	5.00	4,624	0	56	2.00×10^{-7}

表 7　八洲熱学サンプルプログラムの評価結果（StratixII）

デバイス	設計	演算器	状態遷移数	動作周波数 (MHz)	論理規模 (ALUTs)	メモリ (bits)	DSP block	実行時間 (sec.)
PLC	—	—	—	—	—	—	—	3.12×10^{-2}
FPGA	SD	固有	467	16.30	4,752	9,216	16	2.87×10^{-5}
		共有× 1	467	7.94	5,201	9,216	8	5.88×10^{-5}
	LD	固有	90	16.58	4,180	3,584	16	5.43×10^{-6}
		共有× 1	90	7.89	4,705	1,536	8	1.14×10^{-5}
	FD	固有	89	14.17	4,006	1,280	16	6.28×10^{-6}

7.3 APEX20KE を用いた評価

昨年本研究についての概要を報告したとき，評価時のターゲットデバイスには APEX20KE を用いていた[10]．今回さらに変換ツールの改良を行ったため比較用に APEX20KE を用いて評価を行った．その改良の主な内容は，LD における演算時の無駄なステートを削除することで 1 スキャンあたりの状態遷移数を削減したことと，FD による評価を行えるようにしたことである．また，その報告時にはなかった八洲熱学のサンプルプログラムについても評価を行い，Stratix II と APEX20KE の比較を行えるようにした．これらは Altera 社の製品の中でハイエンドモデルとローエンドモデルに位置づけられる．なお，評価環境にもあるように APEX20KE に DSP Block は搭載されていないためここでは省略している．

表 5 は PID 制御プログラムの評価結果である．SD（固有）では，FX2N PLC での実行時間に対して 1810 倍高速化し，論理規模は LE で 10%程度と十分小さい．LD（固有）では SD（固有）に対して 1.3 倍高速化し，FD では SD（固有）に対して 7.8 倍高速化した．LD（固有），FD 共に論理規模は SD（固有）とほとんど同じである．演算器を共有した場合，論理規模（LE）は SD（固有）に対して SD（共有 ×1）で 32%，LD（固有）に対して LD（共有 ×1）で 31%減少する．これは Stratix II とは異なり，乗算器が DSP Block ではなく LE で構成され共有されるためである．

表 9 は A 社サンプルプログラムの評価結果である．まず LD（固有）では SD（固有）に対して 6.2 倍高速化し，FD では SD（固有）に対して 42.9 倍の高速化を達成した．LE は SD（固有）に対して SD（共有 ×1）で 41%，LD（固有）に対して LD（共有 ×1）で 42%減少する．これは Stratix II と同様論理規模の大きい除算器が LE で構成され共有されるためである．

表 10 は，八洲熱学サンプルプログラムの評価結果である．まず LD（固有）では SD（固有）に対して 5.0 倍高速化した．FD でも SD（固有）に対して 5.0 倍高速化と，LD（固有）とほとんど変わらない結果であった．また演算器を共有した場合，LE は SD（固有）に対して SD（共有 ×1）で 2%，LD（固有）に対して LD（共有 ×1）で 6%増加してしまった．

これらの結果を Stratix II で評価した結果と比較しても，実行時間と DSP Block の有無による論理規模への影響に違いはあるが，設計手法の違いによる傾向の変化は見られなかった．

表 8　PID 制御プログラムの評価結果（APEX20KE）

デバイス	設計	演算器	状態遷移数	動作周波数 (MHz)	論理規模 (LEs)	メモリ (bits)	実行時間 (sec.)
PLC	—	—	—	—	—	—	7.98×10^{-4}
FPGA	SD	固有	12	27.24	2,429	0	4.41×10^{-7}
		共有× 1	12	27.60	1,657	0	4.35×10^{-7}
	LD	固有	9	27.17	2,420	0	3.31×10^{-7}
		共有× 1	9	26.38	1,681	0	3.41×10^{-7}
	FD	固有	1	17.64	2,528	0	5.67×10^{-8}

表 9　A 社サンプルプログラムの評価結果（APEX20KE）

デバイス	設計	演算器	状態遷移数	動作周波数 (MHz)	論理規模 (LEs)	メモリ (bits)	実行時間 (sec.)
PLC	—	—	—	—	—	—	1.61×10^{-3}
FPGA	SD	固有	74	3.75	8,135	1,280	1.97×10^{-5}
		共有× 1	74	2.91	4,773	1,024	2.54×10^{-5}
	LD	固有	12	3.76	8,047	0	3.19×10^{-6}
		共有× 1	17	2.98	4,673	0	5.70×10^{-6}
		共有× 2	14	3.04	6,820	0	4.61×10^{-6}
		共有× 3	13	3.21	7,878	0	4.05×10^{-6}
		共有× 4	12	3.06	9,273	0	3.92×10^{-6}
	FD	固有	1	2.18	7,567	0	4.59×10^{-7}

表 10　八洲熱学サンプルプログラムの評価結果（APEX20KE）

デバイス	設計	演算器	状態遷移数	動作周波数 (MHz)	論理規模 (LEs)	メモリ (bits)	実行時間 (sec.)
PLC	—	—	—	—	—	—	3.12×10^{-2}
FPGA	SD	固有	467	6.03	7,171	4,608	7.74×10^{-5}
		共有× 1	467	3.40	7,315	6,656	1.37×10^{-4}
	LD	固有	90	5.83	6,132	4,992	1.54×10^{-5}
		共有× 1	90	3.47	6,521	4,992	2.59×10^{-5}
	FD	固有	89	5.69	6,050	2,176	1.56×10^{-5}

7.4　回路生成時間

本変換ツールでは PLC 命令列で記述された制御論理と同等の VHDL 記述を生成する．そしてこれを論理合成ツールでコンパイル（論理合成）することで，FPGA 上にダウンロード可能な回路記述を生成する．そのため，FPGA で実現

している制御論理に変更があった場合には，そのVHDL記述を変更して再度論理合成を行う必要がある．ここで，論理合成に必要な時間が非常に長ければ，PLCの利点である柔軟性が大きく損なわれてしまうことになる．そこで，先の3つのサンプルの論理合成時間を測定した．

表11はその結果である．まず，ターゲットデバイスの違いによる傾向の変化はほとんど見られなかった．次にサンプルの違いであるが，生成した回路の論理規模ではなく，プログラムの総命令数が多いほど合成時間は延びる傾向がある．そして設計手法による違いであるが，FDはソフトウェア言語と同様の記述（Behavior記述）を用いているため，ある程度データパスや制御パスを意識して記述（RTL記述）されたSDやLDと比べて論理合成ツールによる解析時間が長いと予想されたが，LDやSDと比べて特に遅くはなかった．全体としてみると，短いもので2分以内，長いものでも30分前後で合成は完了している．今後さらに規模の大きなプログラムを扱うことで合成時間の増加が予想されるが，そのような場合でも数時間で完了するため十分実用的であるといえる．

表11　論理合成時間

設計	演算器	Stratix II			APEX20KE		
		PID制御 (sec.)	A社 (sec.)	八洲熱学 (sec.)	PID制御 (sec.)	A社 (sec.)	八洲熱学 (sec.)
SD	固有	113	330	1685	220	806	1889
	共有×1	96	286	1713	139	695	2193
LD	固有	98	306	403	226	782	578
	共有×1	96	265	473	155	708	707
	共有×2	–	302	–	–	864	–
	共有×3	–	751	–	–	1625	–
	共有×4	–	783	–	–	1808	–
FD	固有	103	342	362	238	913	546

7.5　他メーカーFPGAでの評価

ここまではAltera社のFPGAを対象として評価を行ってきた．本節では他メーカーとしてXlinx社のFPGAで評価を行うことを検討する．

本変換ツールでは除算器の生成のためにVHDL記述で演算子‘/’を用いていることを説明した．Quartus IIでは，この記述をマクロコアのLPMに変換しているため本変換ツールで生成したVHDL記述でも合成が可能である．しかし，

Xilinx 社の論理合成ツール（ISE）は，演算子 '/' による記述では除数が 2 のべき乗のとき以外合成することができない．そのため，Xilinx 社の FPGA を用いるためには，除算の記述にマクロコアを直接記述するなどの変更が必要である．今後，こうした異なるメーカーの FPGA にも対応できるよう変換ツールを拡張することが課題として挙げられる．

8　おわりに

本研究ではPLCプログラムの命令列をハードウェア記述言語VHDLに変換するツールの改善を行った．本ツールを用いて三菱電機FX2N PLC向けの3つのサンプルプログラムを変換し，Altera StratixII，APEX20KE FPGA上に実装して評価を行った．開発中の整列巻取機のPLCプログラムについて各設計をStratixII上で比較評価した結果，SDではPLCと比較して約1100倍，FDではSDの4.6倍（PLCの約5000倍）の高速化が得られた．論理規模はSDが4752 ALUT，FDが4006 ALUTで，現行FPGAの容量と比較して充分小さい．

特にFDは，動作周波数は低くなるものの状態遷移数も減少するため，今回提案した3つの設計方法の中では最も大きな高速化が得られた．ただし外部機器IO命令が多く含まれる場合，状態遷移数の削減が制限されるため，FDによる効果も制約される．演算器の共有で論理規模の削減が期待できるが，プログラムの性質次第で効果の大小が大きく影響された．

今後の課題は，まず変換ツールの対応命令を増やすなどの機能拡張を行うことである．例えば，評価用サンプルでは使用されていなかった条件ジャンプ命令やサブルーチンコール命令などは現段階で対応していないため，こうした命令が含まれている場合でも変換できるようにする．次に，山本[9)]の作成したパルスジェネレータだけでなく，高速カウンタやシリアル通信ユニットなどシーケンス制御でよく用いられる周辺機器の制御回路を作成し，変換ツールで作成したメインの制御回路と組み合わせて使用できるようにすることである．例えば，シリアル通信ユニットであれば，FPGA上にソフトマクロのCPUを構成し，その上でシリアル通信を実現するアプリケーションを動作させ，制御回路とバスで接続することでその動作を実現するといった方法が考えられる．また，今回提案した手法を実際の機械に組み込んでその効果を確認し，より大規模な応用に関してもFPGA技術が適用可能であるか検討を進めていきたい．

謝辞

最後まで根気強く御指導してくださった市川先生にこの場を借りて御礼申し上げます．また，本研究でプログラムを作成する際，池田先輩の作成したプログラムを大変参考にさせて頂きました．この場を借りて御礼申し上げます．最後に，苦楽を共にした河合君をはじめ研究室の皆さんに感謝致します．

付録

A.1　実行時間計測ツールについての補足

PLC 上での実行時間の見積もりには，実行時間計測ツール（exetime）を利用した．これは池田[8] が作成したもので，今回新たに追加した命令や要素に対応できるように改良して使用している．使い方は，次に示すように PLC 命令列を記述したテキストファイルを引数として与えるだけでよい．

```
% ./exetime pid.txt
exetime = 797.720
exetime = 632.720
exetime = 715.220
```

ここでは PID 制御プログラムのファイル（pid.txt）を引数として与えた．表示されている実行時間は上から順に，条件部がすべて成立しているときの実行時間（単位はすべて μsec.），条件部がすべて成立していないときの実行時間，この平均の実行時間である．評価にはこのうち条件部がすべて成立しているときの実行時間（worst case）を用いた．これは，FPGA では worst case でも実行時間が変わらないからである．

本プログラムは C 言語で記述されているが字句解析に flex，構文解析に bison を利用しているため，ビルド（make）時には C コンパイラ（gcc）と別にこれらのツールを用意する必要がある．本研究では flex 2.5.4，bison 1.35，gcc 2.95.3 を利用することでビルドしている．

A.2　変換ツールについての補足

PLC 命令列から VHDL 記述への変換には，本研究で作成した VHDL 変換ツール（seq2vhd）を利用する．使い方は，実行時間計測ツールと同様 PLC 命令列を記述したテキストファイルを引数として与えるだけでよい．ただし，以下のオプションを与えることで設計手法等を変更することができる．

-s　Sequential Design で設計する．

-l　Levelized Design で設計する．

-f　Flat Design で設計する．

-an　加減算器の最大生成数を n として共有する．

-mn　乗算器の最大生成数を n として共有する．

-dn　除算器の最大生成数を n として共有する．

-cn　すべての演算器の最大生成数を n として共有する．

-o　出力ファイル名を指定する．デフォルトのファイル名はSEQUENCE.vhd.

-s，-l，-f のオプションのうち，どれも指定がされていない場合には-s による設計を行う．複数指定することはできない．PID 制御プログラム（pid.txt）を Levelized Design かつ，すべての演算器の最大生成数を 1 として共有するよう変換し，output.vhd というファイル名で出力する例を次に示す．

```
% ./seq2vhd -lc1 pid.txt -o output.vhd
```

本プログラムは C 言語で記述されており，flex や bison は使用していない．ただし，オプションの解釈には getopt() 関数を用いているためコンパイラには gcc を利用する．本研究では gcc 2.95.3 を利用することでビルドしている．

A.3　サンプルプログラムの変換例

6 章で説明に用いたサンプルプログラムの変換例を示す．図 45 にそのラダー図（再掲）と対応する命令列（GX Converter[15] により生成したもの）を示す．

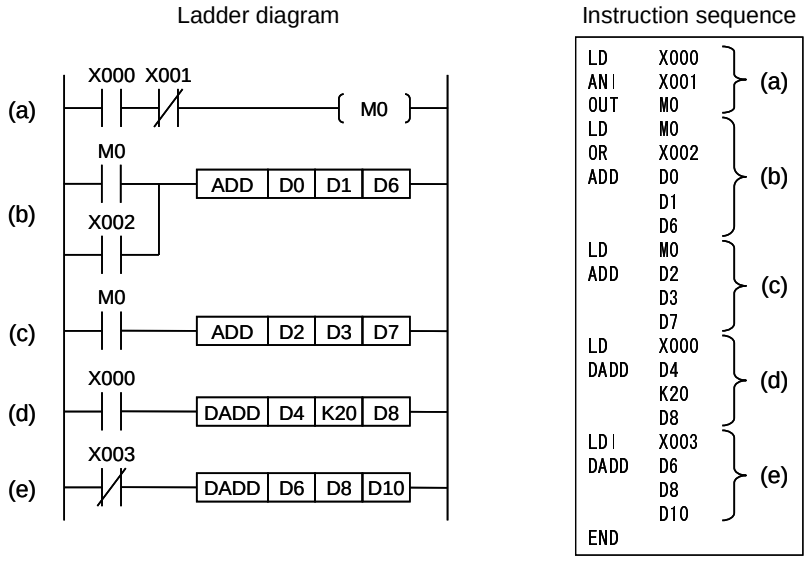


図 45　サンプルプログラムのラダー図と命令列

この命令列を記述したテキストファイルを変換ツールに入力として与え，出力された VHDL 記述を以下に示す．ただし，データレジスタの入出力記述については前述のとおり必要なものを手動で記述している．また，実際には本サン

プルで使用していない特殊レジスタなどの記述がされているがここでは省略した．これら使用していない信号は論理合成時に削除されるためである．

- Sequential Design

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity SEQUENCE is
  port (
    CLK  : in  std_logic;
    RST  : in  std_logic;
    X000IN : in  std_logic;
    X001IN : in  std_logic;
    X002IN : in  std_logic;
    X003IN : in  std_logic;
    D0IN   : in  std_logic_vector(15 downto 0);
    D1IN   : in  std_logic_vector(15 downto 0);
    D2IN   : in  std_logic_vector(15 downto 0);
    D3IN   : in  std_logic_vector(15 downto 0);
    D4IN   : in  std_logic_vector(15 downto 0);
    D5IN   : in  std_logic_vector(15 downto 0);
    D10OUT : out std_logic_vector(15 downto 0);
    D11OUT : out std_logic_vector(15 downto 0)
);
end SEQUENCE;

architecture RTL of SEQUENCE is

  signal STATE : integer;
  signal X000 : std_logic;
  signal X001 : std_logic;
  signal X002 : std_logic;
  signal X003 : std_logic;

  signal M0 : std_logic;

  signal D0 : std_logic_vector(15 downto 0);
  signal D1 : std_logic_vector(15 downto 0);
  signal D6 : std_logic_vector(15 downto 0);
  signal D2 : std_logic_vector(15 downto 0);
  signal D3 : std_logic_vector(15 downto 0);
  signal D7 : std_logic_vector(15 downto 0);
  signal D4 : std_logic_vector(15 downto 0);
  signal D5 : std_logic_vector(15 downto 0);
  signal D8 : std_logic_vector(15 downto 0);
  signal D9 : std_logic_vector(15 downto 0);
  signal D10 : std_logic_vector(15 downto 0);
  signal D11 : std_logic_vector(15 downto 0);

begin
```

```
process (CLK)
  variable CALCA : integer;
  variable CALCB : integer;
  variable CALCR : integer;
  variable CALCT : std_logic_vector(31 downto 0);
begin
  if (CLK'event and CLK = '1') then
    if (RST = '1') then
      STATE <= 0;

      M0 <= '0';

      D0 <= (others => '0');
      D1 <= (others => '0');
      D6 <= (others => '0');
      D2 <= (others => '0');
      D3 <= (others => '0');
      D7 <= (others => '0');
      D4 <= (others => '0');
      D5 <= (others => '0');
      D8 <= (others => '0');
      D9 <= (others => '0');
      D10 <= (others => '0');
      D11 <= (others => '0');

    else
      case STATE is
        when 0 =>
                     X000 <= X000IN;
                     X001 <= X001IN;
                     X002 <= X002IN;
                     X003 <= X003IN;

                     D0   <= D0IN;
                     D1   <= D1IN;
                     D2   <= D2IN;
                     D3   <= D3IN;
                     D4   <= D4IN;
                     D5   <= D5IN;

                     STATE <= STATE + 1;

        when 1 =>
                     if ((X000 and not X001) = '1') then
                       M0 <= '1';
                     else
                       M0 <= '0';
                     end if;

                     STATE <= STATE + 1;

        when 2 =>
```

```
                if ((M0 or X002) = '1') then
                  CALCA := CONV_INTEGER(D0);
                  CALCB := CONV_INTEGER(D1);
                  CALCR := CALCA + CALCB;
                  D6 <= CONV_std_logic_vector(CALCR, 16);
                end if;

                STATE <= STATE + 1;

        when 3 =>
                if (M0 = '1') then
                  CALCA := CONV_INTEGER(D2);
                  CALCB := CONV_INTEGER(D3);
                  CALCR := CALCA + CALCB;
                  D7 <= CONV_std_logic_vector(CALCR, 16);
                end if;

                STATE <= STATE + 1;

        when 4 =>
                if (X000 = '1') then
                  CALCA := CONV_INTEGER(D5 & D4);
                  CALCB := 10#20#;
                  CALCR := CALCA + CALCB;
                  CALCT := CONV_std_logic_vector(CALCR, 32);
                  D8 <= CALCT(15 downto 0);
                  D9 <= CALCT(31 downto 16);
                end if;

                STATE <= STATE + 1;

        when 5 =>
                if (not X003 = '1') then
                  CALCA := CONV_INTEGER(D7 & D6);
                  CALCB := CONV_INTEGER(D9 & D8);
                  CALCR := CALCA + CALCB;
                  CALCT := CONV_std_logic_vector(CALCR, 32);
                  D10 <= CALCT(15 downto 0);
                  D11 <= CALCT(31 downto 16);
                end if;

                STATE <= STATE + 1;

        when 6 =>
                D10OUT <= D10;
                D11OUT <= D11;

                STATE <= 0;

        when others =>
                STATE <= 0;

    end case;
```

```
      end if;
    end if;
  end process;

end RTL;
```

- Levelized Design

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity SEQUENCE is
  port (
    CLK  : in  std_logic;
    RST  : in  std_logic;
    X000IN : in  std_logic;
    X001IN : in  std_logic;
    X002IN : in  std_logic;
    X003IN : in  std_logic;
    D0IN   : in  std_logic_vector(15 downto 0);
    D1IN   : in  std_logic_vector(15 downto 0);
    D2IN   : in  std_logic_vector(15 downto 0);
    D3IN   : in  std_logic_vector(15 downto 0);
    D4IN   : in  std_logic_vector(15 downto 0);
    D5IN   : in  std_logic_vector(15 downto 0);
    D10OUT : out std_logic_vector(15 downto 0);
    D11OUT : out std_logic_vector(15 downto 0)
);
end SEQUENCE;

architecture RTL of SEQUENCE is

  signal STATE : integer;
  signal X000 : std_logic;
  signal X001 : std_logic;
  signal X002 : std_logic;
  signal X003 : std_logic;

  signal M0 : std_logic;

  signal D0 : std_logic_vector(15 downto 0);
  signal D1 : std_logic_vector(15 downto 0);
  signal D6 : std_logic_vector(15 downto 0);
  signal D2 : std_logic_vector(15 downto 0);
  signal D3 : std_logic_vector(15 downto 0);
  signal D7 : std_logic_vector(15 downto 0);
  signal D4 : std_logic_vector(15 downto 0);
  signal D5 : std_logic_vector(15 downto 0);
  signal D8 : std_logic_vector(15 downto 0);
  signal D9 : std_logic_vector(15 downto 0);
  signal D10 : std_logic_vector(15 downto 0);
```

```
  signal D11 : std_logic_vector(15 downto 0);

begin

  process (CLK)
    variable CALCA : integer;
    variable CALCB : integer;
    variable CALCR : integer;
    variable CALCT : std_logic_vector(31 downto 0);
  begin
    if (CLK'event and CLK = '1') then
      if (RST = '1') then
        STATE <= 0;

        M0 <= '0';

        D0 <= (others => '0');
        D1 <= (others => '0');
        D6 <= (others => '0');
        D2 <= (others => '0');
        D3 <= (others => '0');
        D7 <= (others => '0');
        D4 <= (others => '0');
        D5 <= (others => '0');
        D8 <= (others => '0');
        D9 <= (others => '0');
        D10 <= (others => '0');
        D11 <= (others => '0');

      else
        case STATE is
          when 0 =>
                        X000 <= X000IN;
                        X001 <= X001IN;
                        X002 <= X002IN;
                        X003 <= X003IN;

                        D0   <= D0IN;
                        D1   <= D1IN;
                        D2   <= D2IN;
                        D3   <= D3IN;
                        D4   <= D4IN;
                        D5   <= D5IN;

                        STATE <= STATE + 1;

          when 1 =>
                        if ((X000 and not X001) = '1') then
                          M0 <= '1';
                        else
                          M0 <= '0';
                        end if;
```

```
                    if (X000 = '1') then
                      CALCA := CONV_INTEGER(D5 & D4);
                      CALCB := 10#20#;
                      CALCR := CALCA + CALCB;
                      CALCT := CONV_std_logic_vector(CALCR, 32);
                      D8 <= CALCT(15 downto 0);
                      D9 <= CALCT(31 downto 16);
                    end if;

                    STATE <= STATE + 1;

        when 2 =>
                    if ((M0 or X002) = '1') then
                      CALCA := CONV_INTEGER(D0);
                      CALCB := CONV_INTEGER(D1);
                      CALCR := CALCA + CALCB;
                      D6 <= CONV_std_logic_vector(CALCR, 16);
                    end if;

                    if (M0 = '1') then
                      CALCA := CONV_INTEGER(D2);
                      CALCB := CONV_INTEGER(D3);
                      CALCR := CALCA + CALCB;
                      D7 <= CONV_std_logic_vector(CALCR, 16);
                    end if;

                    STATE <= STATE + 1;

        when 3 =>
                    if (not X003 = '1') then
                      CALCA := CONV_INTEGER(D7 & D6);
                      CALCB := CONV_INTEGER(D9 & D8);
                      CALCR := CALCA + CALCB;
                      CALCT := CONV_std_logic_vector(CALCR, 32);
                      D10 <= CALCT(15 downto 0);
                      D11 <= CALCT(31 downto 16);
                    end if;

                    STATE <= STATE + 1;

        when 4 =>
                    D10OUT <= D10;
                    D11OUT <= D11;

                    STATE <= 0;

        when others =>
                    STATE <= 0;

      end case;
    end if;
  end if;
end process;
```

```
end RTL;
```

- Flat Design

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity SEQUENCE is
  port (
    CLK  : in  std_logic;
    RST  : in  std_logic;
    X000IN : in  std_logic;
    X001IN : in  std_logic;
    X002IN : in  std_logic;
    X003IN : in  std_logic;
    D0IN   : in  std_logic_vector(15 downto 0);
    D1IN   : in  std_logic_vector(15 downto 0);
    D2IN   : in  std_logic_vector(15 downto 0);
    D3IN   : in  std_logic_vector(15 downto 0);
    D4IN   : in  std_logic_vector(15 downto 0);
    D5IN   : in  std_logic_vector(15 downto 0);
    D10OUT : out std_logic_vector(15 downto 0);
    D11OUT : out std_logic_vector(15 downto 0)
);
end SEQUENCE;

architecture RTL of SEQUENCE is

begin

  process (CLK)
    variable CALCA : integer;
    variable CALCB : integer;
    variable CALCR : integer;
    variable CALCT : std_logic_vector(31 downto 0);

    variable X000 : std_logic;
    variable X001 : std_logic;
    variable X002 : std_logic;
    variable X003 : std_logic;

    variable M0 : std_logic;

    variable D0 : std_logic_vector(15 downto 0);
    variable D1 : std_logic_vector(15 downto 0);
    variable D6 : std_logic_vector(15 downto 0);
    variable D2 : std_logic_vector(15 downto 0);
    variable D3 : std_logic_vector(15 downto 0);
    variable D7 : std_logic_vector(15 downto 0);
    variable D4 : std_logic_vector(15 downto 0);
```

```
  variable D5 : std_logic_vector(15 downto 0);
  variable D8 : std_logic_vector(15 downto 0);
  variable D9 : std_logic_vector(15 downto 0);
  variable D10 : std_logic_vector(15 downto 0);
  variable D11 : std_logic_vector(15 downto 0);

begin
  if (CLK'event and CLK = '1') then
    if (RST = '1') then
      M0 := '0';

      D0 := (others => '0');
      D1 := (others => '0');
      D6 := (others => '0');
      D2 := (others => '0');
      D3 := (others => '0');
      D7 := (others => '0');
      D4 := (others => '0');
      D5 := (others => '0');
      D8 := (others => '0');
      D9 := (others => '0');
      D10 := (others => '0');
      D11 := (others => '0');

    else
                      X000 := X000IN;
                      X001 := X001IN;
                      X002 := X002IN;
                      X003 := X003IN;

                      D0   := D0IN;
                      D1   := D1IN;
                      D2   := D2IN;
                      D3   := D3IN;
                      D4   := D4IN;
                      D5   := D5IN;

                      if ((X000 and not X001) = '1') then
                        M0 := '1';
                      else
                        M0 := '0';
                      end if;

                      if ((M0 or X002) = '1') then
                        CALCA := CONV_INTEGER(D0);
                        CALCB := CONV_INTEGER(D1);
                        CALCR := CALCA + CALCB;
                        D6 := CONV_std_logic_vector(CALCR, 16);
                      end if;

                      if (M0 = '1') then
                        CALCA := CONV_INTEGER(D2);
                        CALCB := CONV_INTEGER(D3);
```

```
                        CALCR := CALCA + CALCB;
                        D7 := CONV_std_logic_vector(CALCR, 16);
                      end if;

                      if (X000 = '1') then
                        CALCA := CONV_INTEGER(D5 & D4);
                        CALCB := 10#20#;
                        CALCR := CALCA + CALCB;
                        CALCT := CONV_std_logic_vector(CALCR, 32);
                        D8 := CALCT(15 downto 0);
                        D9 := CALCT(31 downto 16);
                      end if;

                      if (not X003 = '1') then
                        CALCA := CONV_INTEGER(D7 & D6);
                        CALCB := CONV_INTEGER(D9 & D8);
                        CALCR := CALCA + CALCB;
                        CALCT := CONV_std_logic_vector(CALCR, 32);
                        D10 := CALCT(15 downto 0);
                        D11 := CALCT(31 downto 16);
                      end if;

                      D10OUT <= D10;
                      D11OUT <= D11;
      end if;
    end if;
  end process;

end RTL;
```

- 演算器共有× 1

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity SEQUENCE is
  port (
    CLK  : in  std_logic;
    RST  : in  std_logic;
    X000IN : in  std_logic;
    X001IN : in  std_logic;
    X002IN : in  std_logic;
    X003IN : in  std_logic;
    D0IN   : in  std_logic_vector(15 downto 0);
    D1IN   : in  std_logic_vector(15 downto 0);
    D2IN   : in  std_logic_vector(15 downto 0);
    D3IN   : in  std_logic_vector(15 downto 0);
    D4IN   : in  std_logic_vector(15 downto 0);
    D5IN   : in  std_logic_vector(15 downto 0);
    D10OUT : out std_logic_vector(15 downto 0);
    D11OUT : out std_logic_vector(15 downto 0)
```

```
);
end SEQUENCE;

architecture RTL of SEQUENCE is

  signal STATE : integer;
  signal X000 : std_logic;
  signal X001 : std_logic;
  signal X002 : std_logic;
  signal X003 : std_logic;

  signal M0 : std_logic;

  signal D0 : std_logic_vector(15 downto 0);
  signal D1 : std_logic_vector(15 downto 0);
  signal D6 : std_logic_vector(15 downto 0);
  signal D2 : std_logic_vector(15 downto 0);
  signal D3 : std_logic_vector(15 downto 0);
  signal D7 : std_logic_vector(15 downto 0);
  signal D4 : std_logic_vector(15 downto 0);
  signal D5 : std_logic_vector(15 downto 0);
  signal D8 : std_logic_vector(15 downto 0);
  signal D9 : std_logic_vector(15 downto 0);
  signal D10 : std_logic_vector(15 downto 0);
  signal D11 : std_logic_vector(15 downto 0);

  signal ADDA1 : integer;
  signal ADDB1 : integer;
  signal ADD1 : integer;

begin

  ADDA1 <=
           CONV_INTEGER(D0) when STATE = 2 else
           CONV_INTEGER(D2) when STATE = 3 else
           CONV_INTEGER(D5 & D4) when STATE = 4 else
           CONV_INTEGER(D7 & D6) when STATE = 5 else
           0;
  ADDB1 <=
           CONV_INTEGER(D1) when STATE = 2 else
           CONV_INTEGER(D3) when STATE = 3 else
           CONV_INTEGER(10#20#) when STATE = 4 else
           CONV_INTEGER(D9 & D8) when STATE = 5 else
           0;
  ADD1 <= ADDA1 + ADDB1;

  process (CLK)
    variable CALCT : std_logic_vector(31 downto 0);
  begin
    if (CLK'event and CLK = '1') then
      if (RST = '1') then
        STATE <= 0;
```

```
  M0 <= '0';

  D0 <= (others => '0');
  D1 <= (others => '0');
  D6 <= (others => '0');
  D2 <= (others => '0');
  D3 <= (others => '0');
  D7 <= (others => '0');
  D4 <= (others => '0');
  D5 <= (others => '0');
  D8 <= (others => '0');
  D9 <= (others => '0');
  D10 <= (others => '0');
  D11 <= (others => '0');

else
  case STATE is
    when 0 =>
                X000 <= X000IN;
                X001 <= X001IN;
                X002 <= X002IN;
                X003 <= X003IN;

                D0   <= D0IN;
                D1   <= D1IN;
                D2   <= D2IN;
                D3   <= D3IN;
                D4   <= D4IN;
                D5   <= D5IN;

                STATE <= STATE + 1;

    when 1 =>
                if ((X000 and not X001) = '1') then
                  M0 <= '1';
                else
                  M0 <= '0';
                end if;

                STATE <= STATE + 1;

    when 2 =>
                if ((M0 or X002) = '1') then
                  D6 <= CONV_std_logic_vector(ADD1, 16);
                end if;

                STATE <= STATE + 1;

    when 3 =>
                if (M0 = '1') then
                  D7 <= CONV_std_logic_vector(ADD1, 16);
                end if;
```

```
                        STATE <= STATE + 1;

          when 4 =>
                        if (X000 = '1') then
                          CALCT := CONV_std_logic_vector(ADD1, 32);
                          D8 <= CALCT(15 downto 0);
                          D9 <= CALCT(31 downto 16);
                        end if;

                        STATE <= STATE + 1;

          when 5 =>
                        if (not X003 = '1') then
                          CALCT := CONV_std_logic_vector(ADD1, 32);
                          D10 <= CALCT(15 downto 0);
                          D11 <= CALCT(31 downto 16);
                        end if;

                        STATE <= STATE + 1;

          when 6 =>
                        D10OUT <= D10;
                        D11OUT <= D11;

                        STATE <= 0;

          when others =>
                        STATE <= 0;

        end case;
      end if;
    end if;
  end process;

end RTL;
```

参　考　文　献

1) 熊谷英樹: “シーケンス制御 プログラム定石集,” 日刊工業新聞社, 2003.

2) M. A. Adamski and J. L. Monteiro: “PLD implementation of logic controllers,” in *Proc. IEEE Int'l Symp. Industrial Electronics (ISIE '95)*, vol. 2, 1995, pp. 706–711.

3) M. A. Adamski and J. L. Monteiro: “From interpreted petri net specification to reprogrammable logic controller design,” in *Proc. IEEE Int'l Symp. Industrial Electronics (ISIE 2000)*, vol. 1, 2000, pp. 13–19.

4) M. Wegrzyn, M. A. Adamski, and J. L. Monteiro: “The application of reconfigurable logic to controller design,” *Control Engineering Practice*, vol. 6, pp. 879–887, 1998.

5) A. Wegrzyn and M. Wegrzyn: “Petri net-based specification, analysis and synthesis of logic controllers,” in *Proc. IEEE Int'l Symp. Industrial Electronics (ISIE 2000)*, vol. 1, 2000, pp. 20–26.

6) M. Ikeshita, Y. Takeda, H. Murakoshi, N. Funakubo, and I. Miyazawa: “An application of FPGA to high-speed programmable controller, development of the conversion program from SFC to Verilog,” in *Proc. IEEE Int'l Conf. Emerging Technologies and Factory Automation (ETFA '99)*, vol. 2, 1999, pp. 1386–1390.

7) I. Miyazawa, T. Nagao, M. Fukagawa, Y. Itoh, T. Mizuya, and T. Sekiguchi: “Implementation of ladder diagram for programmable controller using FPGA,” in *Proc. 7th IEEE Int'l Conf. Emerging Technologies and Factory Automation (ETFA '99)*, vol. 2, 1999, pp. 1381–1385.

8) 池田 亮: “PLC プログラムのハードウェア変換ツールに関する研究,” 豊橋技術科学大学 知識情報工学専攻 平成 16 年度修士学位論文, 2005.

9) 山本 浩司: “FPGA による実時間制御システムに関する研究,” 豊橋技術科学大学 知識情報工学専攻 平成 16 年度修士学位論文, 2005.

10) S. Ichikawa, M. Akinaka, R. Ikeda, and H. Yamamoto: “Converting PLC instruction sequence into logic circuit: A preliminary study,” in *Proc. IEEE Int'l Symp. Industrial Electronics (ISIE '06)*, 2006, pp. 2930–2935.

11) 八洲熱学株式会社, http://www.yashima-ne.co.jp.

12) 三菱電機株式会社, FX1S/FX1N/FX2N/FX1NC/FX2NC シリーズ プログ

ラミングマニュアル, 2006/11, JY992D62001 ver. L.
13) 三菱電機株式会社, MELFANS web FA 用語辞典, http://wwwf2.mitsubishielectric.co.jp/term/index.html.
14) 三菱電機株式会社, GX Developer Version 8 オペレーティングマニュアル, 2006/10, SH080356 ver. S.
15) 三菱電機株式会社, GX Converter Version 2 オペレーティングマニュアル, 2005/12, SH080122 ver. H.
16) 長谷川裕恭: “VHDL によるハードウェア設計入門,” CQ 出版株式会社, 1995.
17) M.Chiang and R.Palkovic, “LCC simulators speed development of synchronoushardware,” *Computer Design*, vol.25, no.5, pp. 87–92, 1986.
18) 笠原博徳: “並列処理技術,” コロナ社, 1991.