計算機アーキテクチャ　98－16
設　計　自　動　化　65－16
(1993. 1. 22)

計算機アーキテクチャ記述言語 CARD-L

高橋 雅史, 小林 広明, 中村 維男

東北大学工学部 機械知能工学科

あらまし

新たな計算機アーキテクチャに基づく計算機を考案した場合、そのアーキテクチャの評価と検証のためにシミュレーションを行う必要がある。しかし、現在までに提案されているモデルの表現方法は、実行効率、記述効率、表現能力において、計算機のモデル化には適さない点がある。本稿では、計算機のモデル化を行い、そのモデルを簡潔に表現可能な記述言語 CARD-L を提案する。そして、この言語の文法と、その意味を定義する。特に、文脈から制御を自動的に抽出するための手法ついて詳しく述べる。最後に、記述例を通し、様々なアルゴリズムによる機能の記述が可能であることを示す。

和文キーワード　ハードウェア記述、計算機アーキテクチャ、モデル化、シミュレーション、実行制御

The Computer Architecture Description Language: CARD-L

Masafumi Takahashi, Hiroaki Kobayashi, Tadao Nakamura

Department of Machine Intelligence and Systems Engineering,
Faculty of Engineering, Tohoku University

Abstract

Simulation experiments must be carried out to evaluate the performance when a new computer architecture is designed. However, the previously proposed description languages for this purpose have some problems in the aspects of execution efficiency, description efficiency and description ability. In this paper, we present a Computer ARchitecture Description Language (CARD-L) that can effectively construct simulation models for target architectures. The semantics of CARD-L is described in detail. Moreover, a strategy for automatic abstraction of controlling factors from descriptions is presented. Finally, the description capability for various functions is discussed by using some examples.

英文 key words　Hardware Description, Computer Architecture, Modeling, Simulation, Execution Controlling

1 はじめに

近年、複雑な情報処理に対する高速化の要求に答えるため、多様な計算機アーキテクチャが提案されている。これにともない、実機を必要としないシミュレーションによる性能評価、および、検証の必要性が高まっている。

シミュレーションを行うには、評価対象となる計算機をモデル化する必要があり、そのモデルの表現形態は 3 つに分けることができる。1 つめは、待ち行列モデルなどに基づく、GPSS などの解析言語による表現である。このモデルの作成は簡単であり、モデルを利用した性能評価も容易であるが、基本要素についての統計データは別の方法を用いて用意しなければならない。また、入力と出力の間の論理的な関係が表現されないために、モデルを検証に利用することができないという問題点を持つ。2 つめが prolog などの論理型プログラミング言語によるモデル記述 [1] である。論理型プログラミング言語は入力と出力の論理的関係を簡潔、かつ、正確に表現する。そして、言語処理系を用いることにより、モデルの検証を行うことができる。しかし、論理型記述言語の処理系は実行する計算機への負荷が大きく、大規模なモデルの処理を高速に行うことは難しい。最後はハードウェア記述言語 [2] を用いた表現である。前述の 2 つの表現方法では、高度に抽象化された簡潔な表現が可能であるのに対して、この方法では多くの情報を記述に含めなくてはならなず、モデル作成者への負担が大きい。しかし、モデル内部にタイミングの情報と論理関係の情報を共に含むため、性能評価と検証の両方に用いることができる。また、実際のハードウェアとのギャップが小さいために、回路への変換も比較的容易である。

現在、ハードウェア記述言語は回路設計に広く用いられており、使用する製造技術や設計手法にあわせて様々な種類のものが利用されている。ところが、一般にハードウェア記述言語では信号の衝突を避けるための排他制御、および、順序制御を記述者の責任で処理しなければならない。しかしながら、多くの要素が同時に動作するハードウェアの制御を細かく記述することは、時間並列的に進行するモデルの取り扱いを苦手とする人間にとって大きな負担になる。そこで、本報告ではこのような負担を軽減した記述が可能な言語 Computer ARchitecture Description Language (CARD-L) を提案し、その言語仕様と意味について述べる。

CARD-L は、計算機アーキテクチャの検証と性能評価のためのシミュレーションモデル記述を目的に設計され、特に

- 文脈による順序制御の表現
- 容易な排他制御の表現
- 簡潔な同期回路の記述
- 既存のプログラミング環境との高い親和性

などの特徴を有する。

2 設計方針

本節では CARD-L の言語設計方針と、それによってもたらされる効果について述べる。

2.1 制御記述の簡略化

CARD-L は手続型言語の構文形態をもち、その文脈から明らかになる制御 [3] に関しては処理系が抽出する。これにより、単純であるが多くの労力を必要とする制御に関する記述の負担から記述者を解放し、処理アルゴリズムや共有資源の制御といった、より高度な問題の記述に専念できるようにしている。

2.2 抽象化

一般に計算機の設計は

1. 機械命令レベル
2. レジスタ間転送レベル
3. ロジックレベル
4. デバイスレベル

といった順序で詳細化が進み、完成に至る。そして、シミュレーションモデルもこの流れにしたがって詳細化を進められることが望まれる。これを実現するには記述言語が様々な抽象度による記述を可能にしていなければならない。しかし、対象とする抽象度の範囲が広くなるほど、言語仕様が大きくなり、言語の基本とするモデルも複雑化する。この相反する条件について考慮した結果、機械命令レベルからレジスタ間転送レベルでのモデル記述を CARD-L の対象とした。これらのレベルでは製造技術への依存が小さく、レベルごとのモデルのギャップも小さいために、簡単な言語仕様にまとめることが可能であり、記述量も少なくて済むという傾向がある。

2.3 同期回路

同期回路による設計は現在広く用いられている設計手法であり、時間を量子化して扱うことにより

- 素子のばらつきの影響を受けにくい
- テストが容易

という特徴を持つ。これらの特徴は CAD との親和性が高く、今後も主要な設計手法として用いられていくと予想される。そこで、CARD-L は同期回路の記述性を向上させるような言語設計を行った。例えば、特別な記述がない場合であっても、レジスタを意味する変数への代入はクロックに同期して行われる。

2.4 記述環境

言語を利用するには、言語仕様に基づく記述を作成するための編集環境と、記述を処理する処理系が必要である。さらに、編集や処理を支援する支援環境が準備されていることが望ましい。

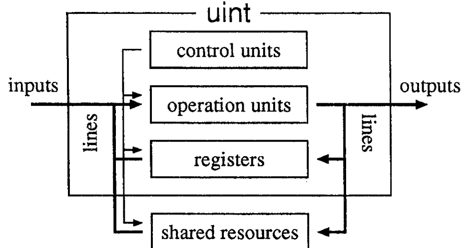

図 1: ユニット

新たな言語の利用環境を構築する場合、処理系の作成は避けられないが、記述環境や支援環境までもすべて新規に作成することは現実的ではない。そこで、CARD-L 処理系の実行は、uinx を OS とするワークステーションクラスの計算機で行うこととし、これらの計算機に広く普及しているプログラミング環境を利用することにした。その結果、CARD-L は C 言語に近い文法をもつ、テキストベースの記述言語となり、unix の備える様々なテキスト処理環境も利用可能となった。

3 基本モデル

3.1 計算機の構成要素

一般に、計算機のハードウェアは図 1に示すようなブラックボックスで表現することが可能であり、CARD-L ではこのようなブラックボックスをユニットと呼ぶ。そして、ユニットには処理ユニットと制御ユニットがある。CARD-L で記述可能な最も小規模なユニットを基本ユニットと呼び、単文で表現する。また、複数のユニットから構成されるユニットを複合ユニットと呼び、複文、あるいは、関数で表現する。

処理ユニットは、入力データや内部データに、目的にあった演算処理を加えるユニットであり、CARD-L では代入文で表現される。代入文は変数、演算子、関数呼び出しから構成され、演算子は組み合わせ回路に、変数はレジスタと回路間の接続にそれぞれ対応する。また、関数呼び出しは複合ユニットを一つの処理ユニットとして記述するために用いる。

制御ユニットは、処理ユニットを正しいタイミングで駆動するという役割を持ち、制御文と文脈による変数値の拘束条件で表現する。CARD-L の大きな特徴である制御に関する記述量の軽減は、制御ユニットの動作を文脈から自動的に抽出することによってもたらされる。

ユニットはそれぞれ自律的に動作している。例えば、関数として記述されたユニットは関数呼び出しによって受動的に起動されるのではなく、入力引数の値の確定といった起動条件を監視しており、条件が満たされると能動的に処理を開始する。ユニットの状態遷移を図 2に示す。ここで、

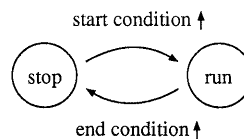

図 2: ユニットの状態遷移

起動条件は

$$\begin{aligned}\text{起動条件} \equiv\ & \text{制御条件} \wedge \text{排他制御条件} \\ & \wedge \text{入力条件} \wedge \text{出力条件} \wedge \text{同期条件}\end{aligned}$$

で定義される。また、図中の '↑' は、それぞれの条件が真になるときに遷移が起こることを示す。

ユニットの動作に関しては 6節で更に詳しく述べる。

3.2 データ

処理系内部では '0' と '1' の 2 値、または、'0','1','u','z' の 4 値の列でデータを表現する。ここで、'u' は任意値、'z' はハイインピーダンス状態値を表す。比較演算子は 2 項のビット列を 1 ビット単位に比較し、全てのビットの値が一致すれば真を返す演算子であるが、'u' と任意の値の比較結果は常に一致と判断する。この性質を利用して Don't care 条件を表すことができる。代入する値が 'z' である代入文は、排他制御条件と出力条件が常に真とみなされる。また、値が 'z' である変数への代入文は出力条件がつねに真とみなされる。これを利用して双方向バスを記述することができる。それぞれのデータにはデータ番号という補助情報が付加されており、排他制御の判断基準として用いられる。

3.3 時間

時間は量子時間と周期時間の組み合わせで記述する。量子時間は処理系が扱う最小の時間単位であり、物理的な時間との対応関係は記述者の定義に依存する。一方、周期時間は同期回路におけるクロック周期に相当する時間であり、量子時間か周期時間を用いて定義する。また、複数のクロックが使用されているシステムを記述するために、周期時間は複数定義可能である。複数のクロックの位相関係もまた、量子時間か周期時間を用いて定義する。

4 CARD-L の処理系

CARD-L の処理系の構成を図 3に示す。プリプロセッサはマクロの展開やライブラリファイルの読み込みといった前処理を行うプログラムであり、C 言語のプリプロセッサを利用する。プリプロセッサの出力はコンパイラに送られ、アセンブラを経てバイナリ形式のオブジェクトコードとして出力される。インタプリタはこのオブジェクトコードを解釈して評価や検証に必要なデータを出力する。

コンパイラは内部的に、中間コードを生成する。この中間コードはテキスト形式であるため、必要ならば人手による編

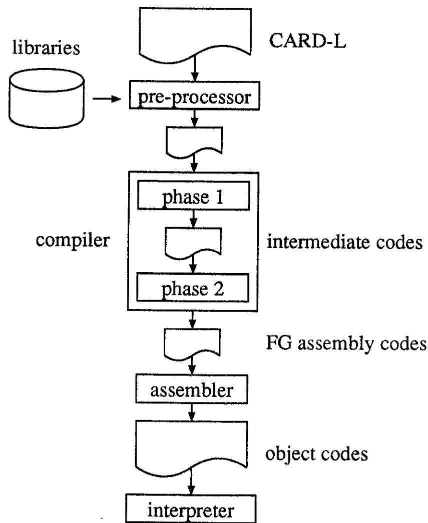

図 3: CARD-L 処理系の構成

集も可能である。CARD-L から中間言語への変換は、制御文展開と関数展開の処理で実現する。これらの処理については5節で述べる。コンパイラの出力する、Flow Graph(FG) アセンブリコードはインタプリタへの入力となるオブジェクトコードと 1 対 1 関係にある言語で、コンパイル段階で抽出可能な、起動条件と終了条件についての情報を格納している。これらの条件については6節で述べる。

5 CARD-L の文法

5.1 変数

変数は処理ユニット中の配線とレジスタに対応し、配線 (line) 型とレジスタ (register) 型がある。line 型変数は値を記憶することができない。そのため、line 型変数の値を一定に保つ必要がある期間は、その値を代入した代入文が動作を続けなければならない。また、register 型変数以外の素子によって、記憶動作が行われることを防ぎ、テストを容易にするために、register 型変数を含まないデータパスのループを禁止している。5.2で述べる名前付きクロックは変数と同様に、加減乗除や代入が行えるが、これらはコンパイル段階で値が決定している必要があるため変数には含まれない。

変数のより細かな性質を記述するために様々な修飾子を用意した。同期修飾子はレジスタ型の変数を修飾する。これには次のような種類がある。

rise クロックの立ち上がりでデータを取り込み、出力する
fall クロックの立ち下がりでデータを取り込み、出力する
ms クロックの立ち下がりでデータを取り込み、立ち上がりで出力する
nms クロックの立ち上がりでデータを取り込み、立ち下がりで出力する
high クロック H の期間がスルーである D ラッチ
low クロック L の期間がスルーである D ラッチ

これら修飾子による修飾は、同期条件と値の確定時刻の決定に影響を与える。クロック修飾子が省略された場合は rise 型とみなされる。

変数が伝達、あるいは、保持するデータのビット幅を指定するために、ビット幅修飾子 fix と variable を用意した。ただし、fix 型変数の宣言では変数名にビット幅指定子 “.[定数]” を付加しなければならない。

定義されたビット幅を越えるデータを fix 型の変数に代入する場合には上位ビットが切り捨てられる。一方、variable 型の変数はどのようなビット幅のデータであっても扱うことが可能であり、処理系は十分に大きなビット幅で定義された fix 型変数として扱う。variable 型は、扱うデータのビット幅が予想できないという条件で記述が行われるライブラリでの利用を想定して用意した。また、詳細な仕様が未定な設計段階での記述にも有効である。ビット幅修飾子とビット幅指定子が共に省略されると variable 型とみなされる。

変数が持つデータを、より大きなビット幅をもつデータへ変換する時に、符号拡張を行うか否かは符号修飾子によって指定する。符号修飾子は signed と unsigned の 2 種類であり、省略時には unsigned とみなされる。

5.2 クロック文

同期回路ではクロックが非常に重要な意味を持つ。それぞれのレジスタに入力されるクロックの波形は、register 型変数を定義する文の直前に記述された clock 指定文で示される。clock 指定文には

< *clock*指定文 >::=
clock(< *low* >：< *high* >：< *phase* >);
< *clock*指定文 >::= clock < 識別子 >;

の 2 形式がある。最初の形式で指定されるクロックは図 4 のようになる。2 つめの形式はクロックを名前で参照するための書式であり、参照されるクロックを名前付きクロックと呼ぶ。名前付きクロックは

< 名前付きクロック定義文 >::= clock < 識別子 >
= (< *low* >：< *high* >：< *phase* >);

で定義する。名前付きクロックは複数箇所で同じクロックを利用するようなユニットの記述に有効である。

5.3 代入文

代入文は変数の値を更新するための文であり、その形式は

< 単純代入文 >::=< 識別子 >=< 式 >;

である。また、値の更新を受ける変数を被代入変数と呼ぶ。

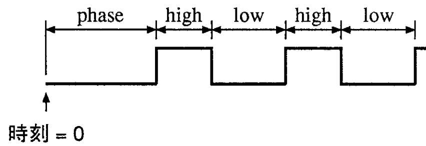

図 4: クロック波形

5.4 制御文

制御文は文の実行状況を変化させるための文であり、それぞれの構文はBNF によって次のように定義される。

< *while* 文 > ::= while(< 式 >) < 文 >
< *do-while*文 > ::= do < 文 > while(< 式 >);
< *repeat* 文 > ::= repeat(< 式 >) < 文 >
< *forever*文 > ::= forever < 文 >
< *select* 文 > ::= select < 文 >
< *continue* 文 > ::= continue;
< *break*文 > ::= break;
< *label*文 > ::= label < 識別子 >:< 文 >
< *switch* 文 > ::= switch(< 式 >) < 文 >
< *case* 文 > ::= case < 式 >:< 文 >
< *breaksw*文 > ::= breaksw;
< *if*文 > ::= if(< 式 >) < 文 >
< *if-else*文 > ::= if(< 式 >) < 文$_1$ > else < 文$_2$ >
< *goto* 文 > ::= goto < 識別子 >;
< *return* 文 > ::= return;

ここで、制御文が< 式 >を持つ場合には、< 文 >は< 式 >によって保護されているという。if 文、if-else 文、label 文、goto 文以外の制御文は制御文展開処理で次のように展開される。

while 文
```
    label start:
        if (< 式 >) < 文 >
        else goto end;
        goto start;
    label end:
    label exit:
```

*do-while*文
```
    label start:
        < 文 >
        if (< 式 >) goto start;
    label end:
    label exit:
```

repeat 文
```
    label start:
        if (< 式 >) < 文 >
        else goto end;
        goto start;
    label end:
        < 文 >
    label exit:
```

*forever*文
```
    label start:
        < 文 >
        goto start;
    label end:
    label exit:
```

continue 文
```
        goto start;
```

*break*文
```
        goto exit;
```

switch 文
```
        比較値 = < 式 >;
        < 文 >
    label endsw:
```

case 文
```
        if (< 式 > == 比較値) < 文 >
```

*breaksw*文
```
        goto endsw;
```
return 文
```
        goto endfunc;
```

5.5 関数

関数定義には

<関数定義 >::= unit[< 制御型 >]
([< 仮引数列$_1$ >][:< 仮引数列$_2$ >][:< 仮引数列$_3$ >])
{< 宣言文 >< 文 >} < 関数名列 > ([< 定数値列 >]);
<関数定義 >::= unit[< 制御型 >] < 関数型名 >
([< 仮引数列$_1$ >][:< 仮引数列$_2$ >][:< 仮引数列$_3$ >])
{< 宣言文 >< 文 >}[< 関数名列 > ([< 定数値列 >])];
<関数定義 >::= unit
< 関数型名 >< 関数名列 > (< 定数値列 >);

の 3 つの形式がある。ここで、[,] で囲まれた部分は省略可能である。< 制御型 >は 6.1で述べる排他制御条件に影響を与える関数の修飾子である。関数定義の 1 つめの形式では、< 宣言文 >と< 文 >で示される構造のユニットの実体を、< 関数名列 >に含まれる関数名の数だけ用意する。2 つめの形式では、< 宣言文 >と< 文 >で示される構造に< 関数型名 >という名前を付け、3 つめの形式で参照できるようにする。関数名列が与えられている場合には、実体の用意も合わせて行う。3 つめの形式では< 関数型名 >によって構造を参照し、実体を用意する。

< 仮引数列$_1$ >と< 仮引数列$_2$ >にはそれぞれ、関数への入力と関数からの出力に対応する仮引数名を指定する。また、< 仮引数列$_3$ >は関数の実体を用意するときに、定数値列の値に置き換えられる仮引数名を指定する。この置き換えの機能を利用して、1 つの関数構造定義で多くの少しずつ異なる関数実体を用意することが可能になる。

関数展開処理によって関数定義の構造記述部分は
```
    {
        < 宣言文 >
        label startfunc:
```

```
        ref (< 仮引数名_11 >);
        ref (< 仮引数名_12 >);
            :
        ref (< 仮引数名_1m >);
        < 文 >
        label endfunc:
        inc (識別子_1);
        inc (識別子_2);
            :
        inc (識別子_n);
        goto startfunc;
    }
```

に展開される。ここで、< 仮引数名$_{ij}$ >は< 仮引数列$_i$ >に含まれる、j 番目の仮引数名である。また、< 仮引数列$_2$ >に含まれる仮引数が、被代入変数となる文は "set < 文 >" に置換される。ただし、< 仮引数名$_{1j}$ >への代入、および、< 仮引数名$_{2j}$ >の参照は文法違反となる。

< 識別子$_1$ >,...,< 識別子$_n$ >は< 宣言文 >で定義される局所変数の変数名である。inc 文は< 識別子 >で指定される変数が持つデータのデータ番号を 1 増加させる。set 文と ref 文については 6.3で述べる。

6 ユニットの動作制御

3.1で述べたようにユニットの動作は起動条件と終了条件に支配されている。本節では、これらの条件について述べる。ただし、定義文などは動作を行わないため、条件は定義されない。そこで、本節では制御文と代入文をまとめて単に文と表現する。また、これらの規則は記述を中間コード展開した後に適用される。

6.1 起動条件

制御条件は制御文が他の文に与える条件である。制御文は動作により以下の処理を行う。

if 文 < 式 >が真なら< 文 >の制御条件を真にする

if-else 文 < 式 >が真なら< 文$_1$ >、偽なら< 文$_2$ >の制御条件を真にする

select 文 < 文 >の制御条件を真にした後、< 文 >に含まれるいずれかの文の動作開始とともに< 文 >の制御条件を偽にする

label 文 ':' と、保護されていない goto 文に挟まれる全ての文の制御条件を真にする

goto 文 対応する label 文の制御条件を真にする

ただし、< 文 >が複文の場合、記述順に制御条件を変化させるものとする。

排他制御条件は代入文に対して定義される条件であり、これ以外の文に関しては常に真である。同一時刻に同じ被代入変数に値を代入するような文があれば、排他制御によって 1 つだけが選択され、その文の排他制御条件が真になる。一方、選択されなかった文の排他制御条件は偽になる。同一関数内に属する文の間の排他制御は、その関数が FIFO 制御型であれば、代入データのデータ番号が最も小さい文を選択する。また、LIFO 制御型であれば、代入データのデータ番号が最も大きい文を選択する。ただし、データ番号 0 は最も優先順位が低い。fast 制御型の場合と、文が同一関数に属さない場合は、最も早く入力条件が真になった文からランダムに 1 つを選択する。

入力条件は、代入文であれば右辺式、条件式を持つ制御文であれば条件式の値が決まることで真になる。条件式を持たない制御文の場合は常に真である。

単項演算子とその項からなる式の値は、被演算項の値が決まることにより確定する。また、式のデータ番号は被演算項のデータ番号に等しい。2 項演算子とその項からなる式の値は 2 つの被演算項の値が共に決まることにより確定する。式のデータ番号は被演算項のデータ番号のより大きな番号に等しい。ただし、論理和演算子は 1 つ以上の被演算項が真に、論理積演算子は 1 つ以上の被演算項が偽に決定することでも値が確定する。この場合の式のデータ番号は、確定した被演算項のデータ番号のより大きな番号に等しい。変数からなる項は、時刻が値の確定時刻よりも後であれば値が確定している。式のデータ番号は最後に代入されたデータのデータ番号に等しい。ただし、共有変数に格納されたデータのデータ番号は 0 である。定数値からなる式の値は常に確定している。定数式のデータ番号は 0 である。式は、それ自身が項になることがある。

出力条件は代入文に対して定義される条件であり、これ以外の文に関しては常に真である。出力条件は文脈上、先行する文の参照する変数の値を、代入文の動作により破壊しないための条件である。被代入変数が参照される可能性があるか否かは、注目している代入文より以前に制御条件が真になり、かつ、動作が完了していない文の参照変数集合を調べることで知ることができる。もし、被代入変数が参照変数集合に含まれていなければ、出力条件は真となる。単純代入文の参照変数集合は右辺に含まれる変数の集合であり、関数呼び出しの参照変数集合は< 引数列$_1$ >の変数の集合である。また、複文の参照変数集合は複文に含まれる文の参照変数集合の和集合である。if 文、および、if-else 文、select 文の参照変数集合はそれぞれ、< 文 >の参照変数集合、あるいは、< 文$_1$ >と< 文$_2$ >の参照変数集合の和集合である。これ以外の制御文の参照変数集合は空集合である。

同期条件はレジスタ型変数を被代入変数とする代入文に対して定義される条件であり、これ以外の文に関しては常に真である。この条件は、取り込み時刻と現在時刻が一致すると真になる。

6.2 終了条件

制御文の動作終了条件は次のようになる。

if 文, if-else 文 制御条件を真にした< 文 >に含まれる全ての文の動作終了

select 動作を開始した文の動作終了

label 常に真

goto 常に真

また、代入文は被代入変数の値が決まることで動作を終了する。被代入変数の値の確定時刻は、line 型であれば右辺の値の確定と同時刻、register 型であれば、右辺の値の確定後、一度の取り込み時刻を経たあとの出力時刻である。制御文と代入文の制御条件は実行終了によって偽となる。

6.3 関数の動作

関数の動作は、

ref文 ::= ref(< 仮引数名 >);

set 文 ::= set < 文 >;

という特殊な文の動作で表現される。

ref 文は呼び出し側で< 仮引数名 >に相当する式の値が確定したときに入力条件が真になる特殊な代入文である。また、ref 文は引数の値を関数に渡すとともに、そのデータにデータ番号を付加する。付加されるデータ番号は 1 から始まり、ref 文が動作するごとに 1 づつ増加する。また、ref 文の出力条件は関数の制御型が single であれば、関数内のすべての起動条件が偽になることで真になる。制御型が multi であれば、出力条件の規則は仮引数を被代入変数とする代入文と同じになる。

set 文は仮変数に相当する、呼び出し側の被代入変数への出力条件を< 文 >に反映する特殊な文である。動作によって、文の右辺の値を被代入変数に代入する。入力条件の規則は通常の代入文と同じである。

7 記述例

同じ処理内容であっても、様々な実現方法が考えることができる。本節では、積算器を例に複数の実現方法があることを示し、それぞれの特徴について述べる。図 5は以下で説明する 3 種類の積算器を呼び出す関数の定義である。

7.1 積算器のモデル化

図 6から図 8は、積算器の定義記述であり、それぞれのブロック図を図 9から図 11に示す。

関数名 mul_1 を持つ関数は入力が与えられるとすぐに出力を返す。一方、関数名 mul_2 と mul_3 を持つ関数はそれぞれ、4, 3 クロックで出力を返す。また、mul_1 と mul_2 は、修飾子 single によって修飾されているために、呼び出し側から a, b 値を受け取ると、関数内の動作条件が全て偽になるまで次のデータを受け取らない。それに対して、mul_3 は修飾子 multi によって修飾されているために、a, b の参照が全て終わり、値が変化してもよい状態になると次の値を受け付ける。CPU のパイプラインの記述も、この動作を応用することで可能である。また、関数型名 multiplier_1 と multiplier_2 で参照される関数構造では、仮引数 w に適当な定数を与えることで、必要に応じた大きさのデータが扱える積算器の実体が用意できる。

```
unit single example( : : )
{
    unit multiplier_1 mul_1(4);
    unit multiplier_2 mul_2(4);
    unit multiplier_3 mul_3(4);
    line a.[4], b.[4];
    register r1.[8], r2.[8], r3.[8];
    a = 3;
    b = 12;
    mul_1(a, b: r1);
    mul_2(a, b: r2);
    mul_3(a, b: r3);
};
```

図 5: 呼び出し側記述

```
unit single multiplier_1
  (a.[w], b.[w]: r.[w * 2]: w)
{
    r = a * b;
};
```

図 6: 組み合わせ回路による記述

図 12は図 7で示した記述の、中間コードへの変換結果である。処理系はこの中間コードに対して条件規則を適用し、文脈から制御ユニットの動作を抽出する。

8 結び

本稿では、計算機アーキテクチャの評価と検証を目的に、計算機のモデル化を行い、そのモデルを表現するための記述言語 CARD-L を提案した。そして、この言語の意味についての説明を行い、文脈から制御の抽出が可能であることを示した。CARD-L を用いることにより、制御に関する記述を大幅に減少させることが可能である。また、言語の柔軟性を示すために、3 種類のアルゴリズムに基づく乗算器を記述した。

しかし、現在の言語仕様では、非同期回路の記述が全く不可能である。回路の遅延時間も表現可能な、より精密な

```
unit single multiplier_2
  (a.[w], b.[w]: r.[w * 2]: w)
{
    register s[w * 2], i[w];
    s = 0;
    i = 0;
    repeat(i < w - 1) {
        if (b.[i])
            s = s + (a << i);
        i = i + 1;
    }
    r = s;
};
```

図 7: シフト演算による記述 1

```
unit multi fifo multiplier_3
  (a.[w], b.[w]: r.[w * 2]: w)
{
    register c[w - 1].[w];
    register d[w - 1].[w];
    register s[w - 1].[w * 2];
    s[0] = a & b.[0] ** (w * 2);
    c[0] = a;
    d[0] = b;
    s[1] = s[0] + c[0]
      & d[0].[1] ** (w * 2);
    c[1] = c[0];
    d[1] = d[0];
    s[2] = s[1] + c[1]
      & d[1].[2] ** (w * 2);
    c[2] = c[1];
    d[2] = d[1];
    r = s[2] + c[2]
      & d[2].[3] ** (w * 2);
};
```

図 8: シフト演算による記述 2

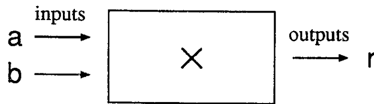

図 9: 組み合わせ回路による構成

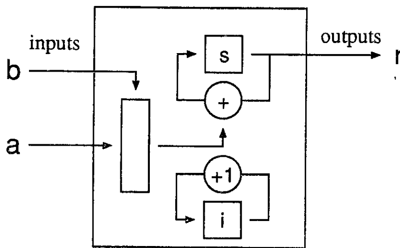

図 10: シフト演算による構成 1

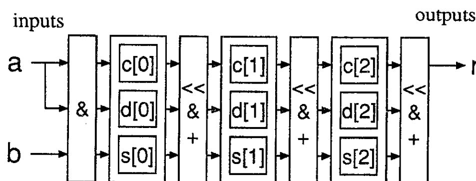

図 11: シフト演算による構成 2

```
unit single multiplier_2
  (a.[w], b.[w]: r.[w * 2]: w)
{
    register s.[w * 2], i.[w];
label l1:
    ref (a);
    ref (b);
    s = 0;
    i = 0;
label l2:
    if (i < 3) {
        if (b.[i]) {
            s = s + (a << i);
        }
        i = i + 1;
    }
    goto l2;
    if (b.[i]) {
        s = s + (a << i);
    }
    i = i + 1;
    set r = s;
label l3:
    inc i;
    inc s;
    goto l1;
};
```

図 12: multiplier_2 の展開結果

計算機モデルを構築し、非同期回路についてもある程度の解析を可能にすることが、今後の課題であると考える。

参考文献

[1] Y. Dotan and B. Arazi, "Concurrent Logic Programming as a Hardware Description Tool," *IEEE Trans. Computers*, Vol. 39, No. 1, Jan. 1990, pp. 72–88.

[2] M. Shahdad et al., "VHSIC Hardware Description Language," *Computer*, Vol. 18, No. 2, Feb. 1985, pp. 94–103.

[3] M. Girkar and C. D. Polychronopoulos, "Automatic Extraction of Functional Parallelism from Ordinary Programs," *IEEE Trans. Parallel Distributed Syst.*, Vol. 3, No. 2, Mar. 1992, pp. 166–178.