設計自動化 38−1
(1987・7・23)

ハードウェア記述言語 "Αιδηζ

中村 敦司　　板野 肯三

筑波大学 工学研究科　　筑波大学 電子・情報工学系

レジスタ転送レベルのハードウェア記述言語 "Αιδηζ(HaDes:Hardware Describer)を設計し、言語の妥当性を検証するために、その記述からシミュレータを生成する簡単な処理系を作成した。"Αιδηζでは、モジュールやロジックと呼ぶ局所的な記述単位を導入することにより、モジュール性の高い記述でハードウェアを表現する。複数のハードウェアを生成するテンプレートの記述によって記述単位が定義されるため、多様なハードウェアを抽象化された簡潔な記述で表現することができる。また、実行の段階では、すべての記憶要素がレジスタ転送レベルの動作をするため、記憶要素の間のデータ転送は1クロックの間に同時に起こる動作として関数性を保って記述される。

HARDWARE DESCRIPTION LANGUAGE "Αιδηζ (in Japanese)

Atsushi NAKAMURA and Kozo ITANO

Doctoral Program in Engineering, University of Tsukuba

Institute of Information Sciences and Electronics, University of Tsukuba

Tennoudai 1-1-1, Sakura-mura, Niihari-gun, Ibaraki-ken, 305 Japan

A register-transfer-level hardware description language "Αιδηζ has been designed and implemented as a simulator generator to verify the validity. In "Αιδηζ, the description units such as modules and logics are introduced for the abstract definitions of hardware structures. Actual hardware components are generated as instances by using the description units as templates. Since the generated hardware components have functional property within a single clock period, the data transfers between storage components can be manipulated on the restricted functional model.

1．はじめに

プログラム言語で用いられているアルゴリズムの記述の枠組みをハードウェアの設計に応用するために、これまでに数多くの高水準なハードウェア記述言語に関する研究がなされてきた[1-3]。これらのハードウェア記述言語では、物理的な実現の詳細に捕らわれることなくハードウェアを記述することができるという利点がある反面、ハードウェアとしての実現との間に距離がありすぎ、実用的な分野で使用されるにはまだ多くの問題をかかえている。しかし、VLSI技術の急速な進歩などによって、ハードウェアとして実現できるアルゴリズムの複雑さはソフトウェアのそれに匹敵するようになってきており、高水準のハードウェア記述言語によるハードウェアの設計の支援が重要になりつつある。

著者らはこのような状況をふまえ、関数性を保った高度な記述性を基本とし、同時にハードウェアの物理的な表現に矛盾なく調和させることのできるレジスタ転送レベルのハードウェア記述言語"Αιδηξ(HaDes:Hardware Describer)を設計し、その簡単な処理系を作成した[4,5]。"Αιδηξでは局所性の高い記述単位を導入することによってモジュール性の高い記述でハードウェアを表現する。この記述単位は、ハードウェアの部品を直接記述するのではなく、部品の構造や動作を一般化して記述し、その記述を展開することによって多種の部品を得ることができるように記述する。このとき、記述単位の中に再帰的に他の記述単位を展開することにより、複雑な構造を階層化して表現することができる。しかし、このハードウェアの構造を求める展開を動的に行うことは現実のハードウェアに合わないため、展開と展開後の動作を区別することにより、一般のオブジェクト指向型の言語のような動的な展開は行わない。また、レジスタを変数によって表現するのではなく、最も基本的な記述単位とすることにより、レジスタ以外の記述単位に1クロックの動作の中で関数性を持たせている。

本稿では、このような特徴を持つハードウェア記述言語"Αιδηξの記述単位とその特徴について述べ、次に基本的な記述の方法とその解釈について、具体例を示しながら説明する。

2．記述単位とその特徴

"Αιδηξでは、組合せ論理回路を表現するロジックと順序回路を表現するモジュールの2種類の記述単位によってハードウェアを記述する。ここでは、これらの記述単位の特徴であるテンプレートとしての記述と、記述単位の関数性について述べる。

2.1 テンプレートとその展開

記述単位は、部品の構造や動作を直接表現するのではなく、静的な定数で表現される構造をパラメータ化することにより、部品を一般的・汎用的なテンプレートとして記述する。そして、その記述を展開することによって実際の部品を得る。この一般的・汎用的な記述は多様なハードウェアの実際の部品を抽象化し、簡潔に表現したものである。

記述単位の中でプログラマの記述した記述単位の複数の実体を使用することができるため、その実体は呼び出し関係に対応した木の階層構造を構成する。また、記述単位は他の記述単位の内部へのアクセスができないため、展開された木構造の枝に対応した局所性が得られる。

2.2 構造数とコネクション

テンプレートとして記述された部品の展開の際に受渡しされるパラメータや語長などはハードウェアの構造を表す数であり、構造数と呼ぶ。構造数は実行時には値が定まっていなければならないため、展開時に評価可能な形で記述されていなければならない。それに対し、展開したハードウェアの内部を転送するデータは構造数で表されるビット幅を持つが、その値は展開したハードウェアが実際に動作しなければ決定できない。このようなデータをコネクションと呼ぶ。これらは構文のレベルで区別され、これらの混同は制限されている。

2.3 ロジックの関数性

ロジックの表す部品は入力のみによって出力が決定する部品であり、入力の変化による出力の変化は全く時間を必要としないと仮定する。このように理想化された組合せ論理回路は、完全に関数性を保って記述される。ここでいう関数性は、単なる記述の形式ではなく、記述の任意の部分を取り出したときに、その部分の出力がその部分の入力によってのみ定まるという参照の透明性(referential transparency)のことである。この性質を持った言語は副作用を持たないため、手続き型言語に代表される他の言語に比べて多くの優れた性質を持っている[6,7]。

参照の透明性を持つ言語による並列な動作の記述は計算の依存関係が矛盾を含まない。これは、ハードウェアの細かい並列性を記述するのに適した性質である。また、参照の透明性を持つ言語では、記述の意味を保ったまま記述の形式を容易に変換することができる。この性質によって、回路への変換やシミュレーションの効率化が比較的容易になると考えられる。

2.4 モジュールの関数性

モジュールの表す部品は状態を保持する記憶要素を内部に持ち、その保持する値をクロックと呼ぶ時間の最小単位に同期して変化させる部品である。内部状態を保持するため、モジュールを完全な関数性を保って記述することはできない。しかし、次に示す記述を行うことにより、制限された関数性を保って順序回路を記述する。

モジュールの中に記述した動作はすべて理想的に同時に起こると仮定し、内部状態を変更しながらこの動作を繰り返すことによって時間に依存した処理を行う。この繰り返しの個々の実行を1クロックの実行と呼んでいる。

すべて同時に起こる動作を記述することから、記述した動作の1クロックの実行の間に内部状態が変化することはない。順序回路は内部状態と入力によって次の内部状態と出力が定まる部品であることから、モジュールの記述のあらゆる部分の出力は内部状態とその部分への入力によってのみ定る。1クロックの実行の間は内部状態は一定でなければならないため、その間は記述のあらゆる部分の出力が入力のみによって定まることになり、参照の透明性が得られる。しかし、後のクロックの実行のために内部状態の変更を指定しなければならず、ロジックのように完全に参照の透明性が保たれているわけではない。このような制限の付いた参照の透明性を"Αιδηξでは1クロックの関数性と呼び、これを保ってモジュールを記述することにより、モジュールの記述においてもロジックの記述と同様の優れた性質を持たせている。

2.5 記憶要素の使用

ここでは、モジュールの記述が1クロックの関数性を保

つために、記憶要素をどのように使用するかについて述べる。

モジュールの記述の中で使用できる記憶要素は内部に展開される他のモジュールの実体である。最も基本的なモジュールとしてレジスタを使用し、他のモジュールはすべてこれを用いて記述することにより、実際に内部状態を保持する部品をレジスタに限定する。

内部の記憶要素へのアクセスは出力の参照とアクションの呼び出しのみが許され、モジュールはこれらのアルゴリズムを記述したものとなる。出力の参照は副作用を含まないアクセスであり、それによって内部状態が影響を受けることはない。アクションの呼び出しは副作用であり、内部状態の変更の指示である。呼び出されたアクションはそれに付随して渡された引数とそのクロックでの内部状態から内部のモジュールの実体のアクションを呼び出す。それは最終的にレジスタのアクションの呼び出しとなるが、レジスタのアクションによる副作用（そのレジスタの出力の変更）が次のクロックにしか現れないため、それを呼び出すアクションの副作用もアクションを呼び出したクロックの中では現れない。

3．ロジックの記述とその解釈

ここではロジックの定義の方法と、その中で用いられるコネクションを変換する回路（コネクション式と呼ぶ）の記述の方法・解釈について説明する。

3.1 ロジックの定義

コネクション式はコネクションの関数であり、プログラマはそれをロジックとして定義することができる。例えば、次の記述は既に定義されているロジックAnd,Or,Notを用いて新たなロジックxorを定義した例である。

```
logic xor〈b_w〉(in_A〔b_w〕 in_B〔b_w〕)
   output:Or〈b_w〉(And〈b_w〉(in_A Not〈b_w〉(in_B))
                   And〈b_w〉(Not〈b_w〉(in_A) in_B))
end
```

ここで、'logic … end'はロジックを定義する構文であり、その中の'output :…'はロジックの本体の記述を行う構文である。

最初の行には、定義するロジックの名前(xor)の指定、構造数の仮引数(b_w)とコネクションの仮引数(in_A,in_B)の宣言、コネクションの仮引数のビット幅(〔b_w〕)の指定が記述されている。構造数の仮引数は記述したロジックを展開する際のパラメータを受け取るものであり、コネクションの仮引数は記述した部品の入力である。'〔b_w〕'は部品への入力のビット長を指定しているものであるが、このようなハードウェアの構造は一般にコネクションを用いて動的に決定することができない。そのため、'〔…〕'の内側にはコネクションを指定してはならず、構造数の式（構造式）を記述しなければならない。

ロジックの定義の本体はプログラム言語の複文に相当する構文（パラグラフと呼んでいる）で記述される。パラグラフはコネクション式の繰り返しであり、最後のコネクション式の評価値をパラグラフの評価値とする。定義の本体の評価値がそのロジックの出力を意味する。この例では、コネクション式は１個しか指定されていない。

3.2 コネクション式

(1)ロジックの使用

先の例のようにして定義されたロジック、あるいは、ライブラリとして既に用意されているロジックは、次のように記述することによって呼び出される。

Or〈 … 〉(…)

これはOrと名付けられたロジックの使用であり、これとAnd,Notの使用例が先のロジックの本体の中に記述されている。'〈…〉'は構造数の実引数を指定し、'(…)'はコネクションの実引数を指定する。

このようなロジックの使用はソフトウェアとしては通常のプログラム言語の関数呼び出しと同様に解釈され、引数を評価した後に呼び出されるという最内側評価(inner most evaluation)によって実行される。また、ロジックの呼び出しのハードウェアとしての解釈は、引数のコネクションを入力、評価値を出力とみなし、定義されたテンプレートの実体をそこに展開することである。ロジックは完全な関数性が保証されているため、呼び出された場所に定義を展開してもよい。

(2)ビット操作

実際のハードウェアでは、単純な配線の組み合わせでデータのビット単位の分割や連結といったビット操作が実現できる。しかし、通常のプログラム言語でこれらをシミュレートするためには、シフトあるいは乗除算と論理演算を組み合わせた複雑な記述が必要である。"Αιδηζでは、コネクションからの指定した一部分の切り出しと複数のコネクションの連結をおこなう構文を用意し、単純な記述でこれらを表現することができる。コネクションの連結はロジック名と構造数パラメータのないロジックの呼び出しとして記述される。また、コネクションの切り出しはコネクション式の後ろに'〔m..n〕'の構文を続けることによって指定する。この場合は、コネクション式の出力の下位n番目からm番目まで(n≦m)のビットを切り出して出力とする。mとnは構造式でなければならない。これらの記述例を次に示す。

(upper_byte 〔7..0〕 lower_byte 〔7..0〕)

この例はupper_byteとlower_byteで示される２つのコネクションをの下位8ビットを連結した16ビットのコネクションを評価値とするコネクション式である。

(3)定コネクション

常に一定の値を持つコネクション（定コネクションと呼ぶ）は構造式の値を用いて表現するため、構造数の式はコネクション式として制限なく記述できる。これは、ハードウェアの定数としてコネクションを指定するということであり、具体的には次の構文によって記述する。

{ 構造式 }

但し、構造数パラメータや数字といった項の構文をとるものについては'{}'を省略して記述することができる。

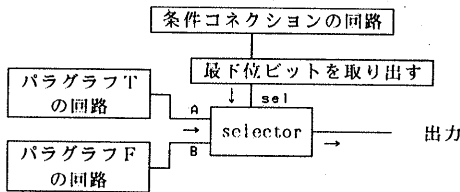
構造数は概念としてはビット幅をもたないが、コネクションは一定のビット幅をもった値である。このため、構造数を変換したコネクションに対して与えるビット幅が問題となる。現在の実現では、実現の容易さと処理の効率の良さから、計算機の語長を持つコネクションに変換している。計算機の語長を越えるコネクションに変換する必要がある場合には、コネクションのビット幅を任意に拡張するロジックを用いて任意のビット幅を持ったコネクションに変換する、このロジックはライブラリとして用意している。

(4)条件分岐 on-other

onは通常のプログラム言語の'if…then…else…'に相当し、一般に次の構文で記述される。

on(条件コネクション式) パラグラフT
other　　パラグラフF

これは条件コネクション式の出力の最下位ビットの値によって評価するパラグラフを切り替えるものである。言語の関数性が保たれているため、２つのパラグラフを評価してから値を決定しても、条件コネクションによって指定されたパラグラフのみ評価しても問題はない。従って、ソフトウェアとして実行する場合には、通常のプログラム言語と同様に、条件コネクション式の出力によってパラグラフTとFのどちらか一方しか評価しないが、ハードウェアとして実現する場合には、次の図に示す回路のように、パラグラフTとFのどちらも同時に動作する。そして、その出力のみが条件コネクションの最下位ビットによって選択される。



(5)選択 decode

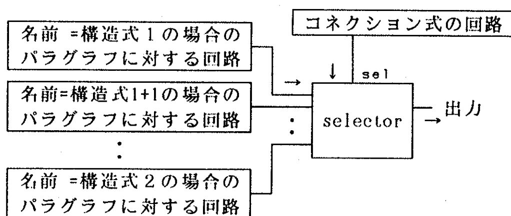
decodeはコネクションを構造数に変換する唯一のコネクション式であり、一般に次の構文で記述される。

decode(コネクション式)
　= 〔構造式１ . 名前 . 構造式２〕パラグラフ　,

これは、コネクション式の評価値が構造式２以上、構造式１以下の値を取る場合に、名前をコネクションの評価値と同じ値を取る構造数としてパラグラフを評価し、その評価値を出力するコネクション式である。また、名前の有効範囲はパラグラフの中だけであり、この名前によって関数性が乱されることはない。次に例を示す。

decode(bit_place) =〔integer.n.0〕target〔n..n〕,

この回路は、コネクションbit_placeの値が0以上構造数integer以下の場合に、構造数nをその値とし、コネクションtargetの下位からn(=bit_place)番目のビットの値を評価値として出力する。コネクション式decodeのハードウェアとしての解釈は次に示す回路となる。



これは、名前の取り得る構造数の数だけパラグラフを実現する回路を用意し、コネクション式の評価値に対応したパラグラフの出力を選択するものである。しかし、コネクション式の値が指定された範囲内にあるかどうかを検査し、名前をコネクション式の値と同じ値を持つ構造数としてパラグラフを一度だけ実行すれば、シミュレーションを行うことができる。

(6)分配 let

１つの出力を多くの部品の入力に分配するために、単一代入規則を守った局所的な変数を使用することができる。単一代入規則によって変数を記憶要素ではなく、コネクションの流れる導線につけた名前とみなすことができ、関数性を損なうことなく記述性を改善している。また、名前の局所性によって名前の重複を許し、記述性をさらに改善している。この変数は次の構文で宣言し、値を定義する。

let 名前 = コネクション式

この構文で宣言された名前は、この式を含む最も局所的なパラグラフの中で、この式の次のコネクション式からそのパラグラフの終わりまで有効である。その中でこの名前はコネクションの引数と同様に使用される。ただし、より局所的なパラグラフの中で、同じ名前が宣言された場合には、そちらの名前の有効範囲が優先する。

3.3 ロジックの再帰的な定義

ロジックの再帰的な定義は回路パターンの柔軟な表現ができるという点で優れている。また、以上の言語仕様とロジックの再帰的な定義ができれば、数学的に計算可能であると言われているあらゆる関数をロジックで記述することができる。しかし、ロジックの再帰呼び出しの回数はテンプレートとしての記述の展開の回数を意味するため、ロジックの再帰的な定義では構造数のパラメータによって再帰呼び出しが終了しなければならない。また、１クロックで動作することのできるハードウェアの量の限界から、こういった再帰呼び出しのロジックによって余りに複雑なハードウェアを記述するのは現実的ではない。

４．モジュールの記述とその解釈

クロックにまたがる記憶要素を持ち、数クロックで機能を実現する順序回路はモジュールによって記述する。"Αιδηζの記述の対象は複雑なアルゴリズムを実現するハードウェアであるため、このような順序回路の記述が最終的な目的となる。ロジックは順序回路の記述の中で、記憶要素を持つ部品の間で転送されるコネクションを変換するアルゴリズムの記述のために用いられる。

ここでは、まず順序回路を記述するための基本的な概念について説明し、その後で、具体的な記述法とその解釈、記述例について示す。

4.1 単相クロックによる制御

"Αιδηζは他のレジスタ転送レベルの記述言語と同様に理想化されたクロックを用いて全体を制御する。つまり、全ての動作はこのクロックに遅延なく同期して起こり、クロックの間にどこかの値が変化することはない。ただし、他の記述言語と異なり、"Αιδηζは単相のクロックしか持たない。モジュールを展開した部品の木構造の最も上位にくる部品に単相のクロックを与え、その木構造の節でアクションの呼び出しを制御することにより、上位の部品から与えられるクロックをそれほど意識することなくモジュールを記述することができる。

4.2 リソースとその構造

リソースはモジュールの具体的な実体であり、テンプレートとして記述したモジュールの実際の記憶領域をモジュール内に展開したものである。モジュールの関数性を保つために、モジュールはリソース以外の記憶要素を持たない。

また、リソースの内部へのアクセスと異なるリソース間のリソースの共有を禁止することによってリソースの独立性を保ち、モジュールの記述性を高めている。

(1)出力とアクション

リソースの階層は出力とアクションによって結合される。出力は下位のリソースから出力されているコネクションを変換し、上位のリソースに渡すものである。これは副作用を持たないため、引数を持たないロジックと同様に記述される。アクションはコネクションの引数を持ち、その引数と下位のリソースの出力によって、下位のリソースのアクションを呼び出すかどうか（呼び出す場合にはその引数も）を決定するものである。木の階層構造の末端にくるリソースが状態を保持し、その状態の変更がアクションによって行われるため、アクションは状態の変更の指令と見なすことができる。また、アクションの呼び出されなかったリソースの状態は変化しないため、アクションをクロックの分配と見なすこともできる。

リソースのハードウェアとしての解釈はデータを記憶する記憶素子とそれらに接続された出力・アクションに対応する組合せ論理回路である。出力・アクションの組合せ論理回路は、ロジックのように使用された場所に実体を生成するのではなく、記憶素子の実体と組を成して生成される。これはハードウェアとして実現した場合に出力やアクションの論理回路と記憶素子は必ず結合する必要があり、これらを記憶素子から離れた場所に使用される数だけ用意するのはハードウェアによる実現を考えた場合に好ましくないと考えられるためである。このような解釈から、入力線を意味するアクションの引数の値と出力線を意味する出力の値が１クロックの間一定であることの保証が必要になる。アクションの引数の値を一定に保つために、１つの下位のリソースに対して１クロックの間に２度以上のアクションの呼び出しを行うと、エラーとなる。また、出力の値を１クロックの間一定の値に保つために、出力はリソースへの入力（アクション）から時間的に分離されており、入力の影響が出力に現れるのは次のクロック以降になる。

(2)レジスタ

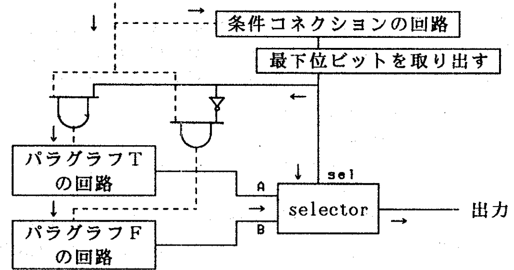
階層の最下位に位置するリソースは最も基本的なモジュールであるレジスタの実体である。

レジスタは語長を意味する構造数のパラメータを持つモジュールであり、出力には常に保持している値が返される。コネクションの引数を読み込むアクションを持ち、保持する値の切り替えはアクションが呼び出された直後のクロックの更新と同時に行われる。

(3)アクションにおけるコネクション式の解釈

アクションはロジックと同様にパラグラフの構文で記述される。ただし、内部リソースの出力の参照とアクションの呼び出しをコネクション式として記述することができる。アクションの呼び出しが副作用を持つため、onやdecode、パラグラフのハードウェアとしての解釈がロジックの場合とは異なる。具体的にはアクションの呼び出しの際に、コネクションの引数の他にトリガとなる線を渡す。レジスタでは、トリガが有効になった場合に、引数を読み込む。パラグラフとonやdecodeのコネクション式では、このトリガを内部のコネクション式に渡すことによってアクションを伝える。ただし、内部にアクションの呼び出しを含まないコネクション式やパラグラフではトリガは不要であり、ロジックにおけるコネクション式やパラグラフのハードウェアと同じ解釈となる。

分岐後のパラグラフがアクションを含むようなonコネクション式のハードウェアの解釈を例として次に示す。



破線はトリガの配線を表し、実線はデータのための配線を表している。また、条件コネクション式とパラグラフT、Fのいづれかがアクションの呼び出しを含まない場合には、そのコネクション式またはパラグラフへのトリガ線は不要である。

4.3 モジュールの記述

(1)モジュールの定義

モジュールは次の構文で記述される。

```
モジュール名〈構造数の仮引数の列〉
    リソースの宣言
    出力の記述
    アクションの記述
end
```

モジュール名に続いて宣言する構造数の仮引数は、このモジュールの定義全体の中で有効である。また、この構造数を演算した構造数に名前を付けて参照する場合には、続く宣言や記述の前に

```
define 構造数名 = 構造数式
```

として定義することもできる。ここで宣言された構造数名は、その宣言以降、endまで有効であり、同じ名前が２度定義された場合には、後で宣言された方の値が優先する。

(2)リソースの宣言

モジュールの記述の中のリソースの宣言は展開する下位のモジュールとそれがリソースとして展開された時の名前を宣言するものであり、

```
resource: register〈16〉 pre_state final_state,
          memory〈16 20〉 xfer_table,
```

のように記述する。この例では構造数パラメータがnであるモジュールregisterを２個用意し、それぞれをリソースpre_stateとfinal_stateとして使用することを表す。registerはレジスタであり、この例ではnビットの語長を持つ。次に、モジュールmemoryを１個用意し、リソースxfer_tableという名前で使用する。モジュールmemoryはregisterを用いて記述されたライブラリの１つであり、メモリを表す。構造数パラメータとしてはアドレスのビット幅とデータのビット幅を指定し、この例では20ビットの語長を持った64K語のメモリである。メモリのようなモジュールを記述するためにリソースの配列を使用することができる。これは次のリソースの宣言の例の中でみられる。

```
resource: register〈wrd〉 cell[adr_width],
```

register〈adr〉 adr_latch

この例では、wrdビットのレジスタの配列をadr_width個用意しており、その各々は0〜adr_width-1の添え字で区別される。

また、リソースの出力がビットフィールドを持つ場合には次の例のメモリのように宣言しておけば、出力やアクションの記述の中でフィールド名で参照することができる。

```
define tbl_a=state_w+input_w
define tbl_w=out_w+state_w+1
resource:register〈state_w〉   pre_state crnt_state,
         memory〈tbl_a tbl_w〉
              xfer_table(il out[out_w] nxt[state_w]),
```

この例では、defineで定義されたtbl_wのビット幅の出力を持つメモリxfer_tableを宣言しており、その出力が、それぞれ1,out_w,state_wのビット幅を持つil,out,nxtのフィールドに分けられることを表している。

(3)出力の記述

モジュールの出力は内部のリソースの出力の関数であり、次の形で定義される。

output: パラグラフ

ロジックの記述と異なるのは、ロジックがパラグラフの中でコネクション式として引数を記述できたが、ここでは引数は存在しない。そのかわり、内部のリソースの出力を次のコネクション式で参照することができる。

リソース名.出力名

出力名は、リソースの宣言の際に指定したビットフィールドの名前であり、省略された場合には全てのビットを参照する。また、リソースの配列の出力を取り出す場合には、リソース名の直後に'[…]'で添え字を指定する。その際の添え字は構造式であり、コネクション式ではない。

(4)アクションの記述

アクションは次の例のように記述する。アクションの記述を繰り返すことによって1組のリソースに対して複数の動作を記述することができる。

```
action : transfer(input[input_w]),
 xfer_tbl.read(on(xfer_tbl.il) (input pre_state.)
                         other (input xfer_tbl.nxt))
 pre_state.load(xfer_tbl.next)
```

この例では、transferをビット幅input_wの引数inputを持ったアクションとして定義している。引数はロジックの定義の場合と同様に、一般には複数個指定でき、アクションの本体はパラグラフの構文で記述される。このパラグラフは出力の記述で使用できるコネクション式に加えて、内部リソースのアクションの呼び出しを記述することができる。この例の中で下位のリソースのアクションを呼び出しているのはxfer_tbl.read(…)とpre_state.load(…)であり、xfer_tbl,pre_stateはリソース名、read,loadはアクション名である。pre_stateはレジスタとして宣言されていることを想定しており、loadは引数のコネクションを読み込むアクションである。

(5)アクションオペレータ

アクションの呼び出しはレジスタなどの低いレベルのリソースでは頻繁に用いられるため、関数呼び出しの形式による記述は読み易さや書き易さの点から好ましくない。そこで、アクションオペレータと呼ぶ概念を導入し、これをプログラマが定義して用いることによって記述を単純にしている。関数呼び出しの形式によるアクションの呼び出しは次の構文で記述される。

リソース名.アクション名(引数のコネクション式)

これと等価な記述を行うために、まず、次の構文でオペレータを宣言する。

define オペレータ アクション名

オペレータは記号列の字句構文で記述されるトークンであり、この宣言によってオペレータが呼び出すアクションを定義する。先のアクションの呼び出しの構文と等価なアクションの呼び出しを行う構文を次に示す。

コネクション式 オペレータ リソース名.

例えば次のようなアクションオペレータの宣言が記述されているとする。

```
define -〉 load
define 〉# read
```

これは、'-〉'をloadと等価なアクションオペレータ、'〉#'をreadと等価なアクションオペレータとして宣言したものであり、この場合、(4)のアクションの宣言の記述例は次のように記述することができる。

```
action : transfer(input[input_w]),
  on(xfer_tbl.il) (input pre_state.)
          other (input xfer_tbl.nxt), 〉# xfer_tbl.
  xfer_tbl.next -〉 pre_state.
```

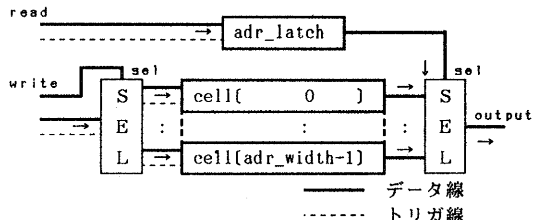
4.4 記述例

次にメモリをモジュールとして定義した例の全体を示す。ただし、';'から行の終端までは注釈である。

```
define -〉 load
memory〈adr wrd〉  ;`adr`bit address `wrd`word memory
  define adr_width =  2**adr
  resource: register〈wrd〉 cell[adr_width],
            register      adr_latch
  output : decode(adr_latch.)
                 =[adr_width-1.n.0] cell[n].
  action: read(address[adr]),
        address -〉adr_latch.

  action: write(address[adr] data[wrd]),
        decode(address)
                 =[adr_width-1.n.0] data -〉cell[n].
end
```

このように定義されたメモリをハードウェアで実現した場合の概略を次に示す。



モジュールmemoryはメモリの本体となるレジスタの配列

cellと出力するアドレスを保持するレジスタadr_latchを内部に持ち、adr_latchによって指定されたアドレスのcellの内容を出力する。アクションreadは、引数で指定したアドレスをadr_latchに転送する。その結果、次のクロックでadr_latchの出力が変わり、指定されたcellの出力がメモリの出力となる。アクションwriteは、引数にアドレスとそのアドレスの新しい内容を指定し、そのアドレスの指すcellの内容を更新する。adr_latchの指定するcellのアドレスを指定した場合には、次のクロックで出力が更新されるが、それ以外の場合には、出力は変化しない。

5．記述単位のソフトウェアによる実現

モジュールやロジックをシミュレーションする際に、内部のリソースやロジックを展開し、プログラム変換を用いて最適な記述に変換することによって効率の良いシミュレーションを行うことができる。また、言語の持つ参照の透明性のために、これらの展開や変換は比較的容易である。しかし、現在の処理系では、処理の簡単さから、こういった展開や変換は行わず、記述したロジックやモジュールに対応した関数や手続きを生成し、１つのロジックやモジュールから展開された実体はすべて同じ関数や手続きによってシミュレーションされる。このような処理系では、展開や変換によって最適化する処理系に比べ、実行と修正の繰り返しが効率よく行える。以下に、現在の処理系がモジュールに対応して生成する関数や手続きについて述べる。

5.1 ロジックのソフトウェアによる実現

ロジックはコネクションの引数と構造数の引数を持ち、評価値となるコネクションを返す関数を生成する。その関数の中はコネクション式に対応したルーチンが記述される。コネクション式に対応したルーチンは静的な変数を含まないため、生成した関数は複数のロジックの実体で共有することができる。

5.2 モジュールのソフトウェアによる実現

リソースは、ソフトウェアとしては、データとそれに付随した手続きとして解釈される。一般のモジュールは関数性を保つために、静的なデータを持たない。実際に静的なデータを持つのはレジスタのみである。このことから、データに付随した手続きはデータ領域毎に用意する必要はない。１つのモジュールから展開されたリソースはデータ領域によって区別され、出力・アクションの処理やその他の必要な処理を行う関数・手続きは全て共有される。現在の処理系では、一般のモジュールのリソースに対して、次の領域D1～D4を確保する。

D1．構造数パラメータを格納する領域
D2．内部のリソースのデータ領域へのポインタを格納する領域
D3．出力のコネクションを格納する領域
D4．アクションが引き起こされたことを表すフラグ

また、処理系は、モジュールの定義から次の手続きP1～P4を生成する。

P1．D1～D4の領域を確保・初期化し、そのアドレスを返す。
P2．出力を計算し、D3を更新する。
P3．アクションを引き起こし、D4を更新する。

P1はシミュレーションの最初の段階で呼び出される。この手続きの内部では、D2の値を得るために、内部のリソースに対応する領域を確保、初期化する。D2に格納されたアドレスはP2とP3の中で、下位のリソースを参照・操作する際に引数として渡される。

P2は出力の記述に対応して生成され、１クロックのシミュレーションが終了した時点で呼び出される手続きである。まず、D4のフラグを用いることによって、出力の更新が必要であるかどうかを判断し、必要であれは、内部リソースの出力を更新した後で、自分自身の出力を計算・更新する。そして最後に、D4のフラグをリセットする。D4による判断は、大規模な木構造を持つリソースでは、実行効率の改善に役立つものと考えられる。

P3はアクション毎に生成され、他のアクションの評価の副作用として呼び出される関数である。まず、D4のフラグを検査し、以前に呼び出されていればエラーを出す。呼び出されていなければ、D4をセットする。そして、引数のコネクションのビット幅を調節し、アクションの本体の記述に対応した処理を行う。この関数の評価値は常にD3の値となる。

6．現在の実現と問題点

現在、"Aιδηξの処理系はモジュール、ロジックの記述をＣの関数へ変換する変換系として実現されている。この変換系が"Aιδηξの記述から生成するプログラムのアルゴリズムの概略については言語仕様の説明の中でソフトウェアとしての解釈という形で述べた。この変換系によって生成されたプログラムをコンパイルし、ランタイムルーチンとライブラリをリンクすることによってシミュレータを作成する。Ｃが分割コンパイルの機能を持つため、"Aιδηξもファイル毎に分割して記述・コンパイルすることができ、モジュール毎の記述を容易に行うことができる。また、引き起こされたアクションを表示しながら実行するプログラムを生成する機能を持たせたり、Ｃのprintf()と同程度の表示能力を持つ部品をライブラリに用意するなど、通常のプログラム言語の持つ便利ないくつかの機能を"Aιδηξの枠組みの中で取り込んでいる（図6-1）。

これまでに述べた"Aιδηξの言語仕様と処理系の実現には多くの問題点が残されている。以下に、これらの問題点について述べる。

・構造数はハードウェアの構造を表している。しかし、現在の言語仕様では、繰り返しの回数しか表現していない。これは、数のみで構造を表現することと、構造数を構造に反映する仕様が不足していることによる。この解決策としては、構造数を数だけではなく、ロジックやモジュール名のような、コネクション式を表現できるように拡張し、そのパラメータとしての受け渡しを許すことである。また、構造数の条件による条件分岐の構文を設け、それをパラグラフの中だけではなく、引数やリソースの宣言にも利用することを許せば、モジュールやロジックがハードウェアの構造をより柔軟に表現できるようになる。

・次の問題点はコネクションが単純なビットの列で全てを表現していることから、受渡しするコネクションの数が多くなった場合にビット幅が長くなり、誤りが入り易くなることである。これは、コネクションにある種のデータ構造を持たせ、引数や出力名、コネクション名のバインディングに利用することによって解決できる。こうすれば、単純なビットフィールドよりも誤りが入り難くなり、たとえ誤った場合にも、他のフィールドへの影響が小さくなる。

・端末やファイルとシミュレータの間で起こる入出力はデ

バッグやデータ収集に欠かせないものである。しかし、こういった入出力は呼び出された順序関係に依存した副作用を引き起こし、言語の関数性を保つ障害となる。現在の処理系では、入出力はロジックPrとSc（Cのprintfとscanfに対応している）をライブラリとして用意し、その使用による副作用の考慮についてはプログラマに任せている。

・モジュールの記述は、関数性を保つための制限の他に、ハードウェアとしての解釈を単純で現実的なものとするための数多くの制限を含んでいる。しかし、デバッグやデータ収集のための記述はハードウェアとして実現する必要がないことから、これらの制約は無意味な記述性の低下をもたらす。そのため、ハードウェアとして実現する必要のないコネクション式がアクションや出力の中に記述でき、その記述が他の部分に影響を与えないことを検査することが望まれる。

・木構造の階層構造を持つリソースは、複数のリソースが一個のリソースを共有することができない。これは、モジュールの局所性を保つために必要である。しかし、この制限はモジュールの記述を必要以上に複雑なものにしている。

・現在の処理系は記述の構文レベルの最適化をまったく行わずに記述を変換する。従って、実行順序が記述と同一であり、入出力やトレースが一見して理解できる。しかし、少し大きなモジュールの記述では、シミュレーションの速度が遅く、確保する記憶領域も大きい。このため記述を構文レベルで最適化する処理系が実用的なシミュレーションを行うためには必要である。

7．おわりに

"Αιδηξの言語仕様はハードウェアによる実現の容易さだけではなく、ソフトウェアでの実行の容易さも考えて決定されている。また、言語の持つ関数性と局所性は記述の部分的な修正と部分的な実行を容易にする。今後の"Αιδηξの処理系の開発は、これらのプログラム言語としての特徴を生かし、プログラミング環境の整備へと向けられる。ハードウェアに近い水準の設計システムでは、テキストによる回路の表現が適切ではないことから、ビットマップディスプレイやポインティングデバイスなどによって高度なユーザインタフェースが提供されている。それに対して、従来のレジスタ転送レベルやアルゴリズムレベルのハードウェア記述言語では、記述の入力やシミュレーションが伝統的なプログラム開発のスタイルを取っており、非常に使いにくい。そのため、高度なユーザインタフェースを持ち、実行を行いながら記述ができるような高度なプログラミング環境の整備がレベルの高いハードウェア記述言語の今後の発展に役立つと考えられる。

参考文献

〔1〕M.R.Barbacci, "A Comparison of Register Transfer Languages for Computers and Digital Systems" IEEE Trans.Comput.,Vol.c-24,pp.137-150,Feb.1975.

〔2〕T.Uehara and M.Barbacci(ed.), "Computer Hardware Description Languages and their applications"IFIP-1983,North-Holland Publishing Company,1983.

〔3〕"Hardware Description Languages" IEEE Computer, vol.18,No.2,Feb.1985.

〔4〕中村敦司,"ハードウェア記述言語"Αιδηξ(情報学類卒業論文)" テクニカルノート HLLA-183,筑波大学・電子情報工学系,1987.

〔5〕中村敦司,板野肯三,"ハードウェア記述言語"Αιδηξ" 情報処理学会第34回全国大会論文集,pp.1923-1924,1987.

〔6〕S.R.Vegdahl, "A Survey of Proposed Architectures for the Execution of Functional Langeuages" IEEE Trans.Comput.,vol.c-33,pp.1050-1071,Dec.1984.

〔7〕D.P.Friedman and D.S.Wise"Aspects of Applicative Programming for Parallel Processing" IEEE Trans. Comput.,vol.c-27,pp.289-296,Apr.1978.

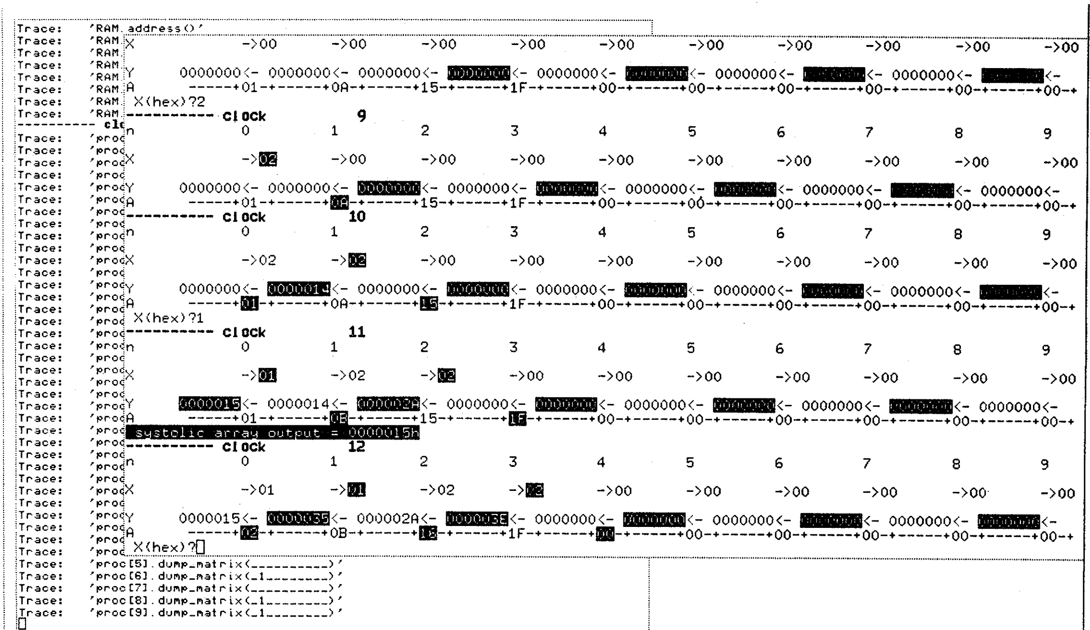


図6-1 生成したシミュレータによる実行の様子