設 計 自 動 化 52－2
（1990．5．22）

ハードウェアシミュレーションシステム ALHARD

小島 泰三† 鶴 薫‡ 杉本 明†
† 三菱電機株式会社 中央研究所
‡ 三菱電機株式会社 コンピュータ製作所

Alhard はC言語をハードウェア記述向きに拡張した，シミュレータ作成用の問題向き言語である．本稿では，Alhard の特徴である，ビット幅付き整数型の導入とC言語演算子の拡張やオブジェクト指向言語機能，イベント処理機能，リソースに対する参照，更新を監視するデーモン機能などについて述べる．そして，Alhard によりハードウェア動作の模擬ばかりではなく，シミュレーション時のチェックや評価のための機能も簡潔に実現できることを示す．また，本稿では対話型シミュレーションのためのコマンドインタプリタとビジュアルインタフェースについても述べる．これらはデーモン機能を用いて実装しており，ハードウェア記述とユーザインタフェースの分離が行なわれている．

ALHARD: An Application-Oriented Language for Hardware Simulation

Taizo Kojima†, Kaoru Tsuru‡ and Akira Sugimoto†
† Central Research Laboratory, Mitsubishi Electric Corporation
8-1-1 Tsukaguchi-Honmachi, Amagasaki, Hyogo, 661, JAPAN
‡ Computer Works, Mitsubishi Electric Corporation
325 Kamimachiya, Kamakura, Kanagawa, 247, JAPAN

Alhard language is an application-oriented language to be used for generation of simulators, and is designed by expanding C language to be used for hardware descriptions. Its characteristics are an operation for the bit string, an object oriented feature, and demon function activated by event driven. In this system, by using the demon function, descriptions can be made independently of check function for hardware descriptions and simulators, and can be separated from user interfaces. In the simulation systems, visual user interfaces, implemented through use of the demon function, are provided.

1 まえがき

計算機ハードウェアの開発には，性能評価や検証のため様々な角度からのシミュレーションが必要である．Alhard (Application-oriented Language for HARDware) システムの目的は，設計者自身によるレジスタトランスファレベルのシミュレータ開発を支援することである．

従来から多数のハードウェア記述言語が提案され，その記述を解釈実行するシミュレーションシステムも開発されている [1, 2, 3, 4]．しかしながら，従来のハードウェア記述言語は，設計仕様を明確に定義することを目的とした設計記述言語としての役割に重点が置かれている．そのため，1) 利用できる構文や関数の種類はハードウェアの動作仕様に関するものに限られている．従って，性能評価のための統計情報を収集する手続きや，検証のためのチェック手続きなどをハードウェア記述の中に組み込むといったことができない．また，2) 自動設計ツールなどが容易に記述を解析できるよう，特定の動作モデルに従った記述を強制される言語もある．そのため，クロックスキームや動作の表現方法をシミュレーションの目的に応じて選択することができない．さらに，3) 記述言語に付随したシミュレータも，コマンドなどのユーザインタフェースが固定され，ユーザが目的に応じてカスタマイズすることが容易ではない．

このようなことから，C言語や ADA，MODULA-2 などの汎用プログラミング言語を使用してシミュレータを個別に作成することも行われている．この場合，記述の自由度は高い．しかし，任意のビット幅のデータ間の演算や，詳細なタイミングに基づく同期並行動作などの記述は一般には複雑となる．また，最初からシミュレータを作成することは大きな労力がかかり，使いやすい高レベルの対話型インタフェースを用意することは困難である．

筆者らは，ハードウェア記述言語と汎用プログラミング言語の双方の利点を生かすため，シミュレータ作成のための問題向き言語としてC言語をハードウェア記述向きに拡張した Alhard 言語を開発した．また，ユーザのカスタマイズ可能な対話型シミュレータインタフェースもあわせて用意した [9, 10, 11]．

Alhard のC言語拡張上の特徴は次の3点である．

- ビット幅付き整数型の導入：基本的なデータタイプとして任意にビット幅を指定できる整数型 (bint) を導入し，それに合わせて C言語演算子の拡張を行なった．
- オブジェクト指向機能：ハードウェア記述のモジュール性を高め，再利用が容易となるよう，オブジェクト指向のクラス，インスタンスの概念や，クラス間継承機能，メッセージ転送機能を実現した．また，ハードウェアブロック間の接続を表現するため，ポートの概念をクラスに導入した．
- イベント処理機能：詳細なタイミングに従う同期動作を模擬するためイベント処理機能を実現した．また，イベントに対する動作が簡潔に記述できるよう，制御構文として新たに when 文を導入した．

Alhard の開発にあたっては，まず様々な動作モデルを構成するための基本機能や，性能評価や検証のための枠組みの提供に重点を置き，上記機能をC言語の拡張により実現した．次にそれらの機能を組合せ，レジスタやメモリなどの標準的なハードウェアリソースを用意すると共に，リソースの参照，更新をイベントとして監視できるデーモン機能を導入し，ハードウェア記述の容易化を計った．

このように2段階に分けて拡張することにより，Alhard ではC言語レベル，基本拡張レベル，ハードウェア記述言語レベルの機能を自由に組み合わせて柔軟にシミュレータを作成することが可能となった．しかも，デーモン機能により，シミュレーション中のチェックや出力表示のための記述をハードウェア記述部分と分離できるため，ハードウェア記述だけを設計仕様ドキュメントとして利用できる．

Alhard 言語の周辺システムとして用意した対話型シミュレータインタフェースにおいても，目的や利用環境に対応できる柔軟性の実現を計った．Alhard では高速なバッチ処理，テキストターミナルによる会話型処理，ワークステーション画面上にハードウェア動作を視覚化するビジュアルシミュレーションの3つの方式を選択できる．また，シミュレータコマンドをハードウェアリソースへのメッセージ転送として扱うことにより，任意のコマンドをユーザが容易に追加できるようにした．さらに，ビジュアルシミュレーション用の画面を容易に定義できるグラフィックエディタも提供している．

なお，筆者らは，既に同様な目的によりオブジェクト指向言語 VEGAMS をハードウェア記述向きに拡張し，マイクロプログラムのビジュアルシミュレータの作成を行っている [5, 6, 7, 8]．

しかし，VEGAMS は LISP 言語をベースとしたオブジェクト指向汎用プログラミング言語としての性格を持っていた．このため，設計者自身によるシミュレータ作成には，まず LISP のシンタックスや組込み関数，プログラミング環境などに慣れる必要があった．Alhard では，C言語をベースとすることにより，ハードウェア設計者に親和性の高い表現形式を提供し，

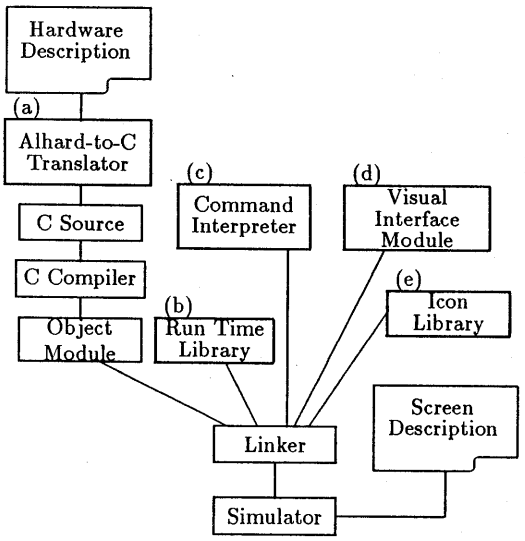

図 1: ソフトウェア構成

また Alhard システムの移植性を高めた.

本論文では，まず，Alhard システムのソフトウェア構成を示し，次に言語仕様の特長，及びシミュレータインタフェースについて説明する．そして最後にマイクロプログラムシミュレータ構築への適用結果について述べ，有効性を明らかにする.

2 ソフトウェア構成

Alhard ではシミュレーション速度の高速化や移植性の向上のため，ハードウェア記述をC言語に変換してコンパイルする方式を採用している．図 1にソフトウェア構成を示す.

Alhard によるハードウェア記述は，(a) の Alhard-to-C トランスレータによりC言語に変換し，これをコンパイルする.

また，コンパイルしたオブジェクトモジュールは，リンク時のライブラリを変更することにより，次の3つの使用形態に対処できる.

1. バッチ型シミュレーション
   コンパイルしたハードウェア記述と，標準的なリソースや任意ビット幅の演算などを定義した (b) の実行支援ライブラリとをリンクし，実行モジュールを作る．バッチ型の高速実行が可能である．しかし，データの入出力文などは Alhard で記述しておく必要がある.

2. 対話型シミュレーション
   シミュレーション時において，リソースの参照，更新や，トレース，ブレークなどを対話的に行えるよう，コマンドの解釈実行を行う (c) のコマンドインタプリタを用意している．これを同時にリンクすることにより，マイクロプログラムやハードウェア記述のデバッグを行うための対話型シミュレータが構成される.

3. ビジュアルシミュレーション
   さらに，グラフィック画面によりシミュレーション実行過程を追跡したい場合は，(d) のビジュアルインタフェースモジュールと (e) のアイコンライブラリともリンクを行う．なお，ハードウェア画面の定義は，ビジュアルインタフェースの画面編集機能を用いて作成する.

3 言語仕様

Alhard の言語仕様は，C言語をベースとしたものになっている．if や while などのC言語の制御構文はすべて使用できる．また，よりハードウェア設計者に親和性の高い制御構文として，ISPS ハードウェア記述言語 [2] で用いられている decode 文をC言語 switch 文の簡易表記法として追加した.

本章では言語仕様の主な特徴について述べる.

3.1 ビット幅付き整数

任意ビット幅のデータを表現するため，新たにビット幅付きの整数型 (bint) を導入した．bint 型データに対する演算はそのデータのビット幅に従って行われる．C言語の演算子がすべて使用できるほか，表 1に示す演算子を追加した．int 型と bint 型が自由に混在できるよう両者の型変換も定義している．なお，現在のシステムでは，255 ビット幅までの演算を実現している.

bint 型は符合無し整数として取り扱う．代入時点の符合拡張や算術シフトにはC言語とは別な演算子を用意している（表 1）．C言語の自然な拡張という点では，int 型と同様に bint 型にたいしても符合付き，符合無しの両方を用意する方が良い．しかし，ハードウェア記述上のデータ操作の観点からは，符合を考慮した演算は，演算子により区別する方が自然である．その判断から bint 型を符合無しのみとした.

3.2 オブジェクト指向機能

新たなハードウェアリソースを部品としてモジュール性高く記述できるよう，オブジェクト指向方式のクラス定義やクラス間の属性継承機能の導入をおこなった.

表 1: 拡張演算子

:=	代入 (符合拡張)
<==	遅延転送
@	結合
:>>	算術シフト (右)
<:<	ローテート (左)
>:>	ローテート (右)
( )	ビットフィールド
[ ]	メモリアクセス

クラスの記述形式の設計ではドキュメントとしての可読性を考慮した. 図 2に例を示す.

図 2は文献 [12] に示された簡単な CPU をクロックによる同期を用いて記述した例である. クラス定義は, 'class クラス名' で始まり, 'end クラス名' で終る. クラス定義内では, このクラスから生成されるオブジェクト (ハードウェアブロック) が使用するリソース, オブジェクトの動作を定義するメソッドを記述する. リソース定義の始まりは 'resource' で, また, メソッド定義の始まりは 'method' で示す. なお, 各メソッド定義は ' メソッド名 is' で始まり 'end' で終る.

図 2では, クラス CPU(行 1-5) とクラス CPU_1 (行 6-76) を定義している. クラス CPU では定数宣言とリソース定義を (行 2-4), またクラス CPU では init, step など 9 つのメソッド (行 11-75) を定義している. 行 6 は, CPU_1 が CPU を継承することを指定している.

ハードウェア記述では, 各部品の外部端子 (ポート) とポート間の接続を表現できることが必要である. Alhard では, オブジェクト間のリソースの共有によりポートを表現している. ポートとして定義されたリソースは, インスタンス生成時にパラメータとして与えられるリソースと同一化され, 部品間のリソースが共有される.

3.3 イベント処理機能

ハードウェア動作シミュレータでは, 詳細なタイミングに従う並行動作を模擬する必要がある. Alhard では, これをイベントドリブンシミュレーションにより行なう. そして, あるイベントが発生した時に行う動作を記述する機能として, event 型データと when 文の導入を行なった.

event 型データに対して, when 文により実行文を記述すると, この実行がアクションとして イベントに登録される. 後に, そのイベントが発生すると, 登録されたすべてのアクションが起動される.

図 2では, 1 サイクルを $\phi 1$ から $\phi 4$ までの 4 つに分割し, データ転送のタイミングを指定するため, 各相に対応させて event 型変数 phi$N$ を定義している (行 9).

```
1  class CPU is
2  const add =#80; adc=#81; clr=#02; ......
3  resource memory mem[256](8);
4    register pc(8),acc(8),sp(8),c,op1(8),op2(8);
5  end CPU
6  class CPU_1 is inherit CPU;
7  resource  register memd(8);
8    bus     abus(8),bbus(8),cbus(8);
9    event   phi1,phi2,phi3,phi4;
10 method
11   init is
12     pc = #00; sp = #FF; fetch;
13     end
14   step is
15     signal(phi1); signal(phi2);
16     signal(phi3); signal(phi4);
17     end
18   read(bint adr) is
19     when(phi2) {
20       abus = adr;
21       when(phi3) bbus = mem[adr]; }
22     end
23   write(bint adr,bint data) is
24     when(phi2) {
25       abus = adr; cbus = data;
26       when(phi4) mem[abus] = cbus; }
27     end
28   fetch is
29     when(phi1) {
30       read(pc);
31       when(phi4) {
32         pc += 1; op1 = bbus;
33         if(op1 & #80) fetch2;
34         else exec; }}
35     end
36   fetch2 is
37     when(phi1) {
38       read(pc);
39       when(phi4) {
40         pc += 1; op2 = bbus;
41         exec; }}
42     end
43   exec is
44     when(phi1) {
45       decode(op1) {
46         add,adc,lda,lds: read(op2);
47         push:            write(sp,acc);
48         jsr:             write(sp,pc);
49         ......
50       when(phi4) {
51         decode(op1) {
52           add,adc: memd = bbus; fetch; operate;
53           clr:     acc = 0;     fetch;
54           com:     acc = ~acc;  fetch;
55           push:    sp -= 1;     fetch;
56           jsr:     sp -= 1;     next;
57           rts:     sp += 1;     next;
58           ......
59     end
60   operate is
61     when(phi1) {
62       decode(op1) {
63         adc: when(phi4) c @ acc = acc + memd + c;
64         add: when(phi4) acc = acc + memd;}}
65     end
66   next is
67     when(phi1) {
68       decode(op1) {
69         jsr:  when(phi2) abus = op2;
70               when(phi3) pc = abus;
71         rts:  read(sp);
72               when(phi4) pc = bbus;
73         ......
74       fetch; }
75     end
76 end CPU_1
```

図 2: Alhard による記述例

Alhard では通常あるイベントによる処理の後，続けて次の別のイベントで処理を行なう場合，when 文を入れ子にして記述する．図 2では，メモリ読み出しは read メソッドに，メモリ書き出しは write メソッドに記述している．read メソッドでは，$\phi 2$ で abus にアドレスを出力し (行 20)，$\phi 3$ で値を bbus に流す (行 21)．また，write メソッドでは，$\phi 2$ でアドレスとデータをそれぞれ abus, cbus に流し (行 25)，$\phi 4$ でメモリに格納する (行 26)．一方，fetch メソッドでは，$\phi 1$ で命令コードの読み込みを指示し (行 30)，$\phi 4$ で利用可能となった命令コードを bbus から取り込む (行 32)．命令長が 1 の場合は，次に exec メソッドを呼び出し (行 34)，2 である場合，残り 1 byte の読み込みを指示する (行 33)．

Alhard では，init メソッドによる初期化と step メソッドによるステップ実行によりシミュレーションが行なわれる．図 2において，init メソッドにおける fetch メソッドの呼び出し (行 12) により最初のアクションが登録される．なお，イベントの発生は，event 型変数に対する signal 操作により起こる．図 2の例では，step メソッドにて phi$N$ に対して signal 操作が行われており (行 15-16)，これによるイベントの発生でシミュレーションが実行される．

このほか，event 型の巡回リストとして timer 型があり，順序付けられた遅延の記述に使用される．

ハードウェア動作記述上のクロックスキームには種々のものが有り得る．このため，クロックを基本データタイプとはせず，event と timer を組合わせることにより，対象ハードウェアやシミュレーションの目的に応じたクロックスキームを構成できるようにした．

なお，言語的側面から説明を加えると，アクションは when 文の実行パートを，登録時点の環境を保存した関数クロージャ [13] として切り出したものである．すなわち，実行パートで使用する変数の内，実行パートの外部で宣言された C 言語のスタック変数に対しては，登録時の値を保存し，実行時に参照できるようにしている．これにより，実行パートの記述でも C 言語のスコープ規則は保たれ，簡潔な記述が可能となっている．

3.4 ハードウェアリソースとデーモン機能

標準的なハードウェアリソースとして，レジスタやバス，メモリを用意している．また，レジスタ内のビットフィールドやフィールド間の結合を仮想的なリソースとして扱うための仮想レジスタや，バイトメモリをワードメモリとしてアクセスする場合などに使用する仮想メモリなども，ハードウェア記述の便宜のため用意した．例えば，仮想レジスタでは，リソース定義において，

```
register AH(8),AL(8),
        A(16) == AH(8) @ AL(8);
```

のように宣言しておくと，レジスタ A を AH, AL を結合した 16 bit 幅のレジスタとして扱うことができる．

ハードウェアの動作シミュレーションでは，このようなリソースに対するデータの参照，更新時に，ある種のチェックや統計情報の収集を行ないたい場合が多い．これを容易化するため，フレーム型システム [13] や LOOPS のアクティブバリュー [14] などに見られる，データの参照，更新を監視するデーモン機能を導入した．

Alhard のデーモン機能の実現では，前述したイベント処理機能を利用している．例えば，レジスタは，bint 型データと，2 種類の event 型データ (読出し時に発生する参照イベントと書込み時に発生する更新イベント) を組み合わて構成した．これらのイベントに登録したアクションが，デーモンとしてレジスタの参照時や更新時に起動される．

また，デーモン登録構文として，ifget 文や ifset 文を導入した．その結果，例えば，初期化時点で，

```
ifset (A) set_count_of_A++;
```

のようにレジスタ A に対して更新デーモンを設定しておくと，A の更新回数を数えることができる．このように，デーモン機能を用いると，統計情報の収集やチェック機能を組み込むために，レジスタ転送の記述を変更する必要がない．

また，デーモン機能により，柔軟なハードウェア動作の記述が可能となる．例えば，A,B,C をレジスタとする．

```
ifget (A) A = B + C;
```

により A の参照デーモンを設定すると，A の値が読出される毎にまず B と C から A を計算してその値が返される (on demand evaluation)．また，逆に，

```
ifset (B,C) A = B + C;
```

により B と C に更新デーモンを設定すると，B や C に値が書込まれると A の値も更新される (forward propagation)．

4 シミュレータインタフェース

2章で述べたように，Alhard では対話型シミュレータを構築するためのコマンドインタプリタと，ビジュアルシミュレーションを可能とするビジュアルインタフェースモジュールを用意している．本章では，これらを利用したシミュレータユーザインタフェースについて述べる．

表 2: シミュレーションコマンド

set	環境変数の参照と更新
do	コマンドファイルの起動
log	ログファイルの設定
help	ヘルプ
quit	終了
visual	ビジュアルモードへの移行
reset	リセット
eval	リソースの参照更新
red,green,blue	リソースの色指定
trace	トレースの設定
break	ブレークの設定
trap	メソッドに対するブレークポイントの設定
notify	メソッドトレースの設定
continue	ブレークポイントからの継続
next	次ステートメントの実行
where	メソッド呼びだしのスタックの表示
var	メソッド変数の参照更新

4.1 コマンドインタプリタ

コマンドインタプリタの基本機能は，キーボードやファイルからのコマンド指示により，Alhard で記述されたメソッドを呼び出すことである．すなわち，コマンドをユーザからのメッセージ転送として捉え，コマンドと同一名のメソッドを選択し実行する．これにより，対話型シミュレータごとに，必要なコマンドを実現することが容易となる．

例えば，ハードウェア動作の記述とは別に，デーモン機能を用いて収集した統計情報に何らかの処理を施して表示を行なったり，あるいは保存するなどの操作を，メソッドとして記述しておく．すると，それをシミュレーション時に対話的に呼び出すことができる．

一方，リソースの更新やトレースなど，一般的にハードウェアシミュレーションに必要とされるコマンドは，内部コマンドとして提供している．内部コマンドの種類を表 2に示す．Alhard では既存のソースレベルプログラムデバッガは利用できない．これに対処するため，ハードウェア記述自体をデバッグするためのコマンドも内部コマンドとして用意している．

内部コマンドでは，リソースのトレースやブレークポイントの設定にデーモン機能を用いている．そのため，柔軟な条件の指定が可能となっている．例えば，レジスタ PC の値をトレースする場合，

```
trace PC:A(12,8) >= 4
```

とすると，レジスタ A の更新時に，A の 12 ビットから 8 ビットまでの値が 4 以上となる場合に PC の値を表示する．また，

```
break =CM[100,120]
```

とすると，メモリ CM の 100 番地から 120 番地までの範囲の読出しが行われた場合に，シミュレーションを中断する．

4.2 ビジュアルインタフェース

ビジュアルインタフェースでは，アイコンやポップアップメニューによる画面インタフェースを提供している．図 3に，Alhard により作成したビジュアルシミュレータの画面例を示す．図中，中央はハードウェアを示すアイコンが表示されるウィンドウ，左下はテキストインタフェースのためのウィンドウ，右下は本シミュレータのデバッグ対象のマイクロプログラムソースが表示されるウィンドウである．シミュレーションは，画面左下のウィンドウからのコマンド入力，あるいは画面上のアイコンを用いて対話的に行なう．画面上でハードウェアを示すアイコンをマウスで指定するとデータの設定を行なうことができる．

ビジュアルシミュレーションの表示方式は，VGEMAS で導入した手法を用いている．そのため，Alhard の演算データにはマーカとして 4 ビット幅のフラグを持たせている．演算においては，オペランドのマーカの論理和を，その結果のマーカとする．ビジュアルインタフェースでは，このマーカを色として使用し，レジスタやバスの現在の値をアイコン化して表示する．このため，シミュレーションは，データの流れに従い，アイコンの表示色が変化する動画として視覚化される．なお，画面表示を前の状態に戻すブレーバック機能も用意している．

一方，視覚化の実現方法においては，VEGAMS とは異る手法を用いている．VEGAMS ではオブジェクト間の作用がメッセージ転送に統一されていた．そこで，メソッドの動的な再定義により動作の視覚化を実現していた [5]．ところが，Alhard ではC言語の構造体と同様な表記で，オブジェクト内部に直接アクセスすることも可能となっている．従ってハードウェア動作がすべてメッセージ転送により進行するわけではない．

そこで，Alhard の記述から生成されるハードウェアオブジェクトとは別個に，表示を担当するオブジェクトとしてアイコンオブジェクトを用意した．そして，アイコンオブジェクトからハードウェアオブジェクトにデーモンを設定する，アクティブバリューによる視覚化方式 [14] を採用した．

画面上へのアイコンの配置は，図 4に示す画面エディタにより行なう．マウスで画面左側のメニュをクリックしてアイコンを生成し，配置する．ハードウェアオブジェクトとアイコンオブジェクトの対応は，図 4の中央，右のように，ポップアップメニューを用い，ハードウェアリソース名を入力することにより指定する．このときシステム内部では，リソース名文字列を用いてハードウェアオブジェクトを参照し，デーモンの設定解除を行なう．画面エディタはビジュアルインタフェー

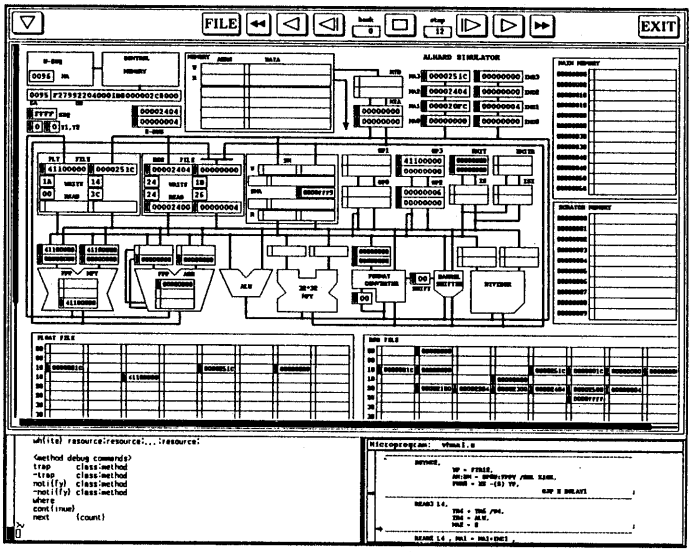

図 3: ビジュアルインタフェースの画面例

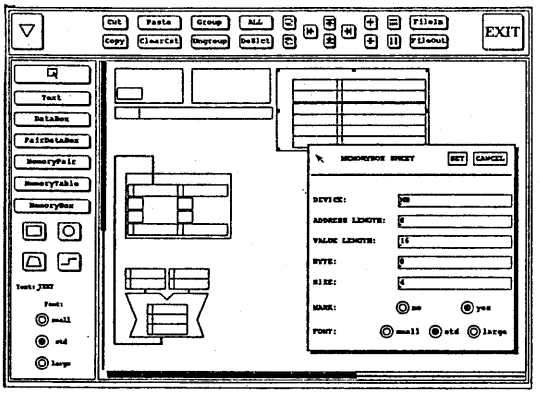

図 4: アイコン編集画面例

スモジュールの一つのモードとして実現しているため，シミュレーション実行時にも画面編集が可能である．

5 マイクロプログラムシミュレータへの適用

図 3のシミュレータは，MELCOM MX/5000 ミニコンピュータのマイクロプログラムデバッグ用に開発したものである．Alhard によるハードウェア記述量は，4500 行であり，9 個のクラスと 115 個のメソッドの定義を行なった．これは，約 17000 行のC言語ソースに変換されている．

ハードウェア記述の多くはデータ演算であり，任意ビット幅の演算機能により，ハードウェア仕様書のテーブルをそのまま Alhard 記述に変換できた．また，イベント処理機能やデーモン機能により，マイクロプログラムの並行動作や浮動小数点データのパイプライン処理，メモリアクセス時の同期操作のチェックなども容易に記述することができた．その結果，本シミュレータのハードウェア記述は 1 人で 3 週間という短期間で終了した．また，クラス機能の利用により，個々のハードウェアブロックについて再利用性の高い部品化が行なえた．

MX/5000 は，104 ビット幅の水平型マイクロプログラムを持つハードウェアである．並行性が重要な水平型マイクロプログラムにおいては，各ハードウェアの並行動作を直感的に認識できるビジュアルシミュレーションが有効である [7]．そのため，図 3 に示す画面を作成したが，4 章で述べた画面エディタを用いて 3 日で完了した．本シミュレータを用いて現在までに約 280 本のマイクロプログラム（総ステップ数は約 16000 行）のデバッグを行なっている．

本シミュレータは，MELCOM ME200 ワークステーション (MC68030 20MHz) を用いて，X-Window 上に実現されており，シミュレーション速度は，1 マイクロプログラムステップ当りテキストインタフェースの場合平均 60ms であり，画面上のアニメーション

を行うと平均400msである．この速度はハードウェアモデル，使用システムの差により単純には比較できないが，VEGAMS[8]と比較して10倍程度高速になっている．

なお，本シミュレータ以外にもAlhardシステムを用いたマイクロプログラムシミュレータの開発が現在4件行なわれている．また，マルチプロセッサボードのシミュレーションへの適用も開始している．

6 おわりに

本論文では，ハードウェア記述言語 Alhard とシミュレーションシステムについて概要を述べ，その具体例として作成したマイクロプログラムシミュレータにより，その有効性を明らかにした．

Alhardは，任意ビット幅の演算命令や同期動作動作のためのデーモン機能を持ったオブジェクト指向の問題向き言語であり，ハードウェア記述をC言語に変換して実行する．イベント処理機能，デーモン機能により，ハードウェアやシミュレーションの目的に応じて様々なハードウェアのモデル化が可能である．またハードウェアのモデル化をユーザインタフェース構築から分離できるため，シミュレータ作成が容易になる．表3にAlhardシステムのソフトウェア量を示す．

残された問題として，より上位の並行動作概念の導入がある．対象ハードウェアの内部仕様が利用可能な場合は，イベント処理機能やデーモン機能によって容易に同期並行動作を表現できる．しかしながら，命令セットなどの外部仕様のみから同期並行動作を実現する場合には，一連の動作をタスクとして表現する方が，並行動作の記述としてはより明解である．今後は，その実現手法について検討していきたいと考えている．

表 3: Alhard のソフトウェア量(行)

	C ソース	Yacc, Lex
Alhard-to-C translator	8,000	800
run-time library	3,000	
text interface module	2,500	700
visual interface module	10,000	
window toolkit	8,000	
合計	31,500	1,500

参考文献

[1] A. V. Dam,et al., "Simulation of a horizontal bit-sliced processor using the ISPS architecture simulation facility", *IEEE Trans. Comput.*, C-30, No. 7, July, 1986, pp. 513-519.

[2] M. Barbacci, "Instruction set processor specifications (ISPS): The notation and its applications", *IEEE trans. comput.*, C-30, No. 7, Jan, 1986, pp. 24-40.

[3] R. Lipsett,E. Marschner,M. Shahdad, "VHDL–The Language", *IEEE Design and Test*, Apr, 1986, pp. 28-41.

[4] A. S. Gilman, "VHDL–The Designer Enviroment", *IEEE Design and Test*, Apr, 1986, pp. 42-47.

[5] A. Sugimoto, "VEGA: A Visual Modeling Language for Digital Systems", *IEEE Design and Test*, Jun, 1986, pp. 38-45.

[6] A. Sugimoto,et al., "An Object-Oriented Visual Simulator for Microprogram Development", *Proc. of the 23rd Design Automation Conf.*, Jun, 1986, pp. 138-144.

[7] 杉本, 阿部, 黒田, 加藤, "オブジェクト指向方式による対話型マイクロプログラムシミュレータ", 信学会論文誌, Vol. J70-D, No. 2, pp 315-324, 1987.

[8] 杉本, 阿部, "オブジェクト指向言語 VEGAMS による構造レベルハードウェアのモデル化", コンピュータソフトウェア, Vol. 3, No. 3, pp 71-85, 1986.

[9] 杉本, 小島, 阿部, 鶴, 加藤, "ハードウェア動作記述言語： ALHARD(1) 概要とマイクロプログラムシミュレータへの応用", 情処 37 全大 講演論文集, 3, pp 1743-1744, 1988.

[10] 小島, 杉本, 阿部, 鶴, 加藤, "ハードウェア動作記述言語：ALHARD(2) 言語仕様とC言語への変換処理", 情処 37 全大 講演論文集, 3, pp 1745-1746, 1988.

[11] 鶴, 加藤, 杉本, 小島, 阿部, "ハードウェア動作記述言語：ALHARD(3) シミュレーションインタフェース", 情処 37 全大 講演論文集, 3, pp 1747-1748, 1988.

[12] 論理設計ＣＡＤに関する調査報告書, 電子協,61-C-528,1986.

[13] P.H.Winston,B.K.P.Horn, *LISP (Second Edition)*, Addison-Wesley, Reading, Massachusetts, 1984.

[14] M.J.Stefik,D.G.Bobrow,K.M.Kahn, "Integrating Access-Oriented Programming into a Multiparadigm Enviroment", *IEEE Software*, Jan,1986, pp. 10-18.