ソフトウェア工学 68－3
（１９８９．９．２６）

ハードウェア動作記述からのシミュレータ自動生成

井上勝博　　三原幸博

株式会社　東芝　　システム・ソフトウェア技術研究所

ソフトウェアシミュレータは、マイクロコンピュータ組み込みシステムのソフトウェア開発において有効なツールである。本稿では、そのシミュレータを対象のハードウェア毎に生成するシステムについて述べる。本システムでは、ハードウェアの変更に対し効率よく対応するため、シミュレータの内部構造を、オブジェクト指向により設計し、ハードウェアのアーキテクチャに依存する部分と依存しない部分を明確にすると共に、ハードウェア記述言語を設計することにより、この言語からの自動生成を実現した。本システムにより、従来のプログラミング言語での記述に比べ、１／８程度の記述でシミュレータを開発することが可能である。

AUTOMATED GENERATION OF A SOFTWEAR－SIMULATOR FROM A HARDWARE BEHAVIOR DESCRIPTION

Katsuhiro Inoue　　Yukihiro Mihara

System and Software Engineering Laboratory, TOSHIBA Corporation
70 Yanagi-cho Saiwai-ku Kawasaki, 210 Japan

A software simulator is an effective tool for debugging software for micro-computer built-in systems. In this paper, we discuss a system to generate the simulator for each target hardware. In this system, for efficient generation, we have adopted an object-oriented approach to design the simulator construction and we have divided it into architecture dependent parts and independent parts. In addition, we have designed a hardware behavior description language, and have implemented the automated generation of the simulator from it. This system allows development of simulators to be done in 1/8 description.

1．はじめに

半導体技術の進歩により、マイクロコンピュータの高性能化、低価格化が進み、マイクロコンピュータを組み込んだ多くの製品が開発されている。それに伴いこのような組み込み型のマイクロコンピュータ用のソフトウェアの開発量も年々増加の傾向にあり、その開発効率を向上させることは重要な課題となっている。

我々は、ソフトウェア統合開発支援システムＩＭＡＰ(Integrated software MAnagement and Production support system)[1]開発の一環として、組み込み型マイクロコンピュータのソフトウェアの品質向上を目指して、特に、ソフトウェアのデバッグ／テストの強化を行なっている。組み込み型マイクロコンピュータ用のソフトウェアは、通常の汎用のコンピュータ用のアプリケーションのように、開発環境が整っていないことが多い。このようなソフトウェアの開発には、クロス環境での開発が有効であると考え、ソフトウェアの論理デバッグ／テストの支援ツールであるソフトウェアシミュレータを開発し、ソフトウェアのクロス開発を進めてきた。[2]

ソフトウェアシミュレータは、ターゲットの動作を汎用のコンピュータ上で、シミュレートすることにより、ソフトウェアのデバッグ／テストを行えるようにしたツールであるが、これは、ターゲットの動作をシミュレートするため、ターゲットが変わると使用できない。最近では、マイクロコンピュータの多種多様化、短寿命化により、ターゲットの変化が激しく、開発支援ツールもこれに対応する必要がある。

本稿では、このソフトウェアシミュレータを各種のハードウェア（ターゲット）に対して、柔軟かつ迅速に対応する方式に付いて述べる。本方式では、各種ターゲットに対して迅速に対応するため、シミュレータ内部構造をオブジェクト指向の考え方を用いてモデル化し、ターゲットの動作に依存する部分と依存しない部分に分け、依存する部分は、ターゲット毎に作り、依存しない部分は、再利用することによりシミュレータを開発する。また、新規開発部を効率的に開発するため、ハードウェア動作記述言語を設計し、この言語からの自動生成を実現した。[3]

2．論理デバッグ／テスト

組み込み型マイコンの分野では、ソフトウェアとハードウェアは、並行して開発されることが多く、通常の汎用コンピュータ用のソフトウェアのようにその開発時において実行環境が整っていないことが多い。このため、ソフトウェアのデバッグ／テストは、ソフトウェアと並行して開発されるハードウェア（実機）上で行なうか、または、それに相当するデバッグ／テスト環境を作りその上で行なう、または、実機と同等のＣＰＵを搭載したエミュレータを利用する等の方法により行なってきた。

しかし、これらの方法では次のような理由から有効なソフトウェア開発が行えない。

- ソフトウェアのデバッグ／テストは、その実行環境であるハードウェアが必須であり、ハードウェアの開発が、ソフトウェアの開発に影響を及ぼす。
- ソフトウェアの開発とともに、ハードウェアも並行して作られるため、発生したバグが、ソフトウェアによるものなのか、ハードウェアによるものなのかが明確に区別できない。
- ハードウェアを用いた動作環境は、ソフトウェアの内部の細かい動作を確認することが困難な場合が多い。

そこで、これらの問題点を解決するため、ソフトウェアのデバッグ／テストにおいて、ソフトウェアとハードウェアの分離が必要であると考え、その工程を論理デバッグ／テスト工程と物理デバッグ／テスト工程に分けた。（図２．１）

論理デバッグ／テスト工程は、専用のハードウェア（実機、エミュレータ等）を用いずに、ソフトウェアのみのデバッグ／テストを行なう工程である。これに対し、物理デバッグ／テスト工程は、論理デバッグ／テストの終了後に行なう工程で、従来の実機、エミュレータ等を用いてソフトウェアのテストを行なう。論理デバッグ／テスト工程で、ソフトウェアの論理的（アルゴリズム等）な誤りをなくし、物理デバッグ／テスト工程では、タイミング等のハードウェアに関する

部分のみをテストする。このようにするとソフトウェアのより厳密なデバッグ／テストが行える。

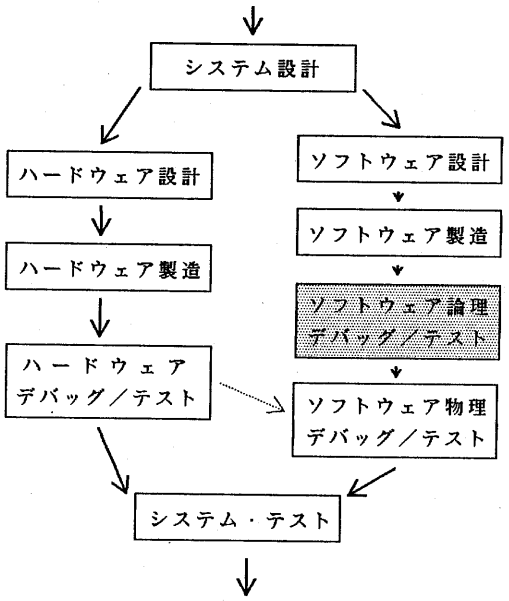

図２．１　論理デバッグ／テスト

論理デバッグ／テストを行なうためには、その支援環境が必要である。この環境は、実際の実行環境からソフトウェアを切り離した汎用のコンピュータ上（クロス環境）で、ソフトウェアの設計からデバッグ／テストまでを行えることが望ましい。このようなクロス環境でのデバッグ／テストツールとしてシミュレータがある。我々は、このような開発方法の有効性を確認するため、当社製の４ビットマイコンを対象としたシミュレータを開発し、製品開発に適用したところ、マイコン内部動作の可視化、操作性、バグ発見効率、開発環境移動のロスがない、開発環境がコンパクト等の点に関して有効であり、結果的に開発期間を約３０％程度短縮できることが確認できた。

このようなシュミュレータは、通常、ターゲット毎に作り、ターゲットが変わると使用することが出来ない。最近では、マイクロコンピュータの多種多様化、短寿命化が進行し、ターゲットの変化が激しくなっている。このような状況から、マイコンのソフトウェアを効率的に開発するためには、開発支援ツールに関してもこれに迅速に対応することが必要である。

このような状況を考慮し、ソフトウェアシミュレータを各種のターゲットに対して柔軟に対応する方式について以下に述べる。

３．シミュレータの実現方式

論理デバッグ／テストツールとしてのシミュレータは、ソフトウェアから見たハードウェアの動作を正確にシミュレートする機能とデバッグ／テストを行なうための機能を持つ。例えば、ハードウェアの動作をシミュレートする機能としては、

・ハードウェア構成要素（リソース）の実現
・目的プログラムの実行
・割り込み処理

等があり、デバッグ機能としては、

・ブレークポイント設定
・実行トレース
・ハードウェア内部状態の表示／変更

等がある。ここではこれらの内、シミュレーション機能の実現方式について主に述べる。デバッグ／テスト機能は、シミュレーション機能をベースとして上位に実現する方針で設計する。このようにすることにより、ターゲットが変わっても汎用的に使用できるデバッグ／テスト機能や、ユーザインタフェースを実現する。

シミュレーション機能を実現する方法としては、ソフトウェアで全ての動作をシミュレートする方法とハードウェアを利用する方法がある。ハードウェアを用いるとより高速で、正確な実行が出来るが、ハードウェア内部の可視化が困難で、機能拡張性が乏しく、環境が大きくなる。また、近年のハードウェアの高速化にともなってエミュレーションは困難になってきている。これに対し、ソフトウェアで実現する方式は、速度的には劣るが、開発期間、コスト、機能拡張性等の面で優れている。

ここでは、設計からコーディング、デバッグ／テストまでを汎用のコンピュータ上で行えるコンパクトな開発環境を実現することを目指し、また、豊富なデバッグ／テスト機能の組み込みの可能性の点から、ソフ

トウェアのみで実現する方法を用いる。

ターゲットのコンピュータの動作をシミュレートするには、ハードウェアのアーキテクチャの切り口としてマイクロプログラムレベル、高級言語レベルなどいろいろな切り口が考えられる。[4] 低水準になるほどハードウェアのより細かい動作をシミュレートする必要があり、高水準になるほどハードウェアの動作は抽象的でよい。ここでは、組み込み型マイクロコンピュータのソフトウェア開発を対象としているため、ハードウェアの切り口をアセンブリ言語レベルとしてシミュレータの実現方式を考える。

アセンブリ言語レベルでハードウェアを考えたときハードウェアの動作は、基本的に、メモリ、I／Oポート、レジスタ等のハードウェアリソース間のデータ転送で実現できる。プログラムは、ハードウェアをレジスタ、メモリ等に対するデータ操作により制御し、また、周辺装置は、I／Oポートに対するデータの入出力によって制御されると考えられる。したがって、プログラマにとっては、ハードウェアが物理的にどのように構成されていようと、レジスタ、メモリ、I／Oポート等のデータがプログラムの実行によってどの様に変化するかが見えればよい。

したがって、ハードウェアのアーキテクチャは、多種多様に存在するが、基本的には、ハードウェアのリソースの構成と、プログラムによりリソース間のデータ転送がどの様に行なわれるかという情報によりシミュレータは実現できる。これらの特徴をいかして各種のターゲットに対応するシミュレータの実現方式を提案する。

異なるターゲットに対応するシミュレータを開発する方法として、ターゲットの変更に対して柔軟に対応できる1つのシミュレータを作る方法と、ターゲット毎にシミュレータを生成する方法の2つが考えられる。前者の方法は、各種のターゲット毎の機能をシミュレータ内に持ち込まなければならないため、大規模になり、性能的に問題が出る可能性がある。後者の方法は、ターゲットの毎に作成するため、よりコンパクトで、性能の良いものを実現できる。このため、ここでは、後者の方法を取る。

この方法の実現のポイントとして、

・シミュレータ内部構造のターゲット依存関係の抽出
・非依存部の部品化再利用、依存部の新規開発
・依存部開発の効率化
・シミュレータ開発の自動化

を挙げ、シミュレータ開発効率の向上を目指した。

ターゲット依存関係の抽出と、非依存部の部品化再利用に関しては、オブジェクト指向を取入れ、依存部開発の効率化に関して、ハードウェア記述言語を設計し、この言語からの自動生成を実現した。

4．オブジェクト指向による内部構造設計

シミュレータの内部構造の設計にあたり、オブジェクト指向[5] の考え方を用い、内部構造のモデル化、モジュール化を行い、再利用を図った。

シミュレータは、前章で述べたようにハードウェアのリソースとそれらの間のデータ転送で実現できる。したがって、シミュレータとして実現しなければならないのは、リソースの実現とリソース間のデータ転送手順の組み込みである。以下にそれぞれの実現方式について述べる。

（1）ハードウェア・リソースの実現

ハードウェアのリソースは、一般に、レジスタ、メモリ、I／Oポート等があり、アーキテクチャが異なってもリソース名とサイズなどが異なるだけで、これらは存在し、同じような性質を持っている。したがって、これらのリソースをそれぞれクラスとして定義し、インスタンスとして生成できるようにして、それらをリソースとして扱えるようにすれば、より自然な形で実現できる。（図4．1）

具体的には、レジスタ、メモリ、I／Oポート等のクラスを設け、そのクラスに属するインスタンスがリソースになるようにする。各クラスはターゲットに関

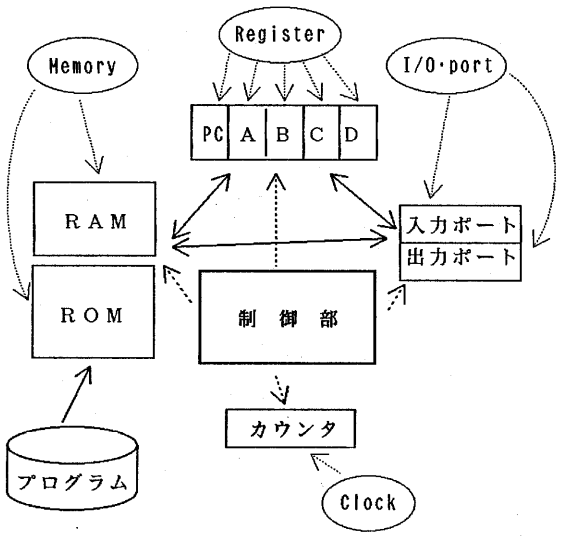

図４．１　リソースの実現

係しない固有の性質を持つ。たとえば、メモリに関しては、アドレス空間がある、プロテクト機能がある等の性質があり、レジスタに関しては、ビットサイズがある、フラグの処理に影響する等の性質がある。したがって、これらの性質をクラス内に含め、ターゲットに依存しない共通の性質として利用する。また、ターゲットに依存するリソースの名前、サイズ等に関しては、インスタンスの生成時に属性として与えることによりターゲット毎のリソースを実現できるようにした。

この名前、属性等に関しては、ハードウェア記述言語で宣言的に記述することにより、この言語から自動的に生成するようにしている。

（２）リソース間のデータ転送手順

リソース間のデータ転送手順、すなわち、ハードウェアの実行順序に関しても、ターゲットに依存するものと依存しないものがある。ターゲットに依存しない一般的なものについては、あらかじめ組み込んでおくことにより、処理手順の枠組みとして利用する。依存するものに関しては、次の章で述べるハードウェア記述言語で、予め宣言しておいたリソース名を用いて、オブジェクト間のデータ転送として、手続き的に記述する。

ハードウェアに依存しない一般的な処理手順とは、プログラムカウンタ参照による命令フェッチと命令実行の繰り返し、及び、割り込みの発生の判断処理等があること及びその処理手順であり、依存する機能とは、命令コードに対する動作、割り込み要因とその処理等の手順である。

このように、シミュレータの機能のターゲットへの依存関係に着目して処理手順も分類できる。これを利用して、シミュレータの内部構造をより一般的なものにするため、シミュレータの機能にも着目して、クラス分けする。これにより、モジュール化、再利用を推進する。シミュレータ内部のクラス構成とクラス間の操作の関係を図４．２に示す。この関係図は、基本的なデバッグ機能も含んでいる。

このように設計すると、ターゲットの依存関係が明確になり、モジュール化、再利用に有効であるとともに、各クラス間のインタフェースが明確になるり、ターゲットが変わってもこの構成は汎用的に利用できる。

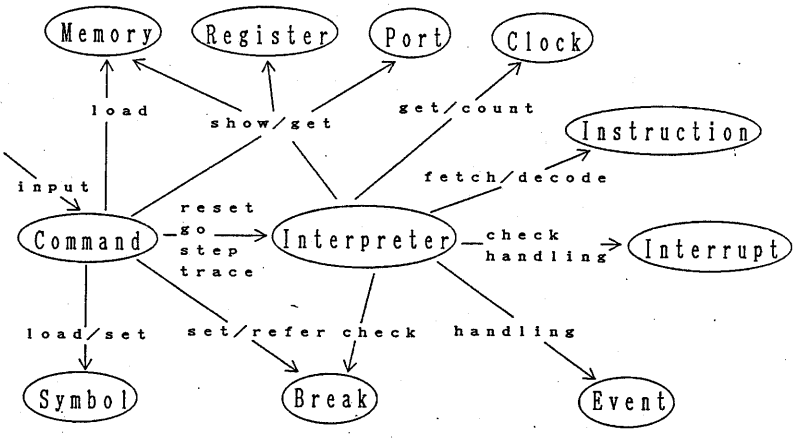

図５．４　シミュレータ　クラス構成

5．ハードウェア動作記述言語

シミュレータの新規開発部を効率的に生成するため、ハードウェア動作記述言語を設計し、この言語からシミュレータを自動生成する。この言語は、リソースの構成の宣言的記述、ターゲット依存動作の手続き的記述を基本として記述する。ハードウェア動作記述言語の設計に当たり、以下の点を考慮した。

・記述容易性（記述量が少なく簡潔である）
・理解性（読み易い）
・記述能力（ハードウェアの動作を確実に記述できる）
・形式性（シミュレータへの変換ができること）

具体的には、次のような記述能力を持つ。

・記述すべき項目を明確にする。
・ターゲットに依存しない機能は記述しない。
・各種のターゲットに対応したリソースを、属性を指定することにより宣言的に記述する。
・プログラムの命令コードと動作、ニーモニックの対応を、マイクロプロセッサの命令一覧表のイメージで簡潔に記述する。
・割り込み処理は、割り込み要因とその処理の対応を表形式で記述する。
・個々の処理の具体的な動作については、手続き型言語を用いて手続き的に記述する。

表５．１に記述項目の一覧表を、図５．１に記述例を示す。

記述項目	開始ステートメント	記述内容
リソース	resource	ハードウェアの持つリソース（メモリ、レジスタ、Ｉ／Ｏポート等）の構成を記述する。
初期値	initial	リソース記述で記述したリソースの初期値を指定する。
実行制御順序	execution	１命令の実行される順序（命令フェッチ、実行、割り込み処理等の関係）を記述する。
命令動作	instruction	個々の命令コードの動作を記述する 命令コード、動作、ニーモニックの対応で記述する。
クロック・カウント	clock	クロックのカウント方式を記述する。命令サイクルとクロックの関係を記述する。 これは、内部タイマによる割り込み処理や、入出力のタイミングを計るのに用いる。
割り込み処理	interrupt	割り込み要因とそれに対する割り込み処理動作を記述する。 割り込み要因と、割り込み処理の対応で記述する。
補助	event	命令コードの動作以外にターゲットの動作に影響を及ぼす動作を記述する。 リソースアクセス等をイベントとしてそれに対する動作を記述する。Ｉ／Ｏポートがシステムの状態に影響する特殊な機能が割り当てられているようなターゲットの動作を記述するのに用いる。
	supplement	同様に命令コードの動作に関係なくシングルステップ毎に発生するような動作を記述する。
	user	上記の記述中で、動作の記述は、基本的にＣ言語風の記述形式で記述できる。この中では、任意の関数を使用することができ、このような関数の内部をこの部分で定義する。

表５．１　ハードウェア動作記述言語記述項目一覧

```
%resource
//  主レジスタ
        Register        F(8),A(8),B(8),C(8),D(8),E(8),H(8),L(8);
        Pegister        IX(16), IY(16), SP(16), PC(16);
//  ペア・レジスタ
        Pair    AF      ( A, F ),
                BC      ( B, C ),
                DE      ( D, E ),
                HL      ( H, L ) ;
//  Fレジスタ構成
        Map     SF      F.bit(7),       // サイン・フラグ
                ZF      F.bit(6),       // ゼロ・フラグ
                IFF     F.bit(5),       // 割り込み許可フラグ
                HF      F.bit(4),       // ハーフ・キャリ・フラグ
                XF      F.bit(3),       // 拡張キャリ・フラグ
                OV      F.bit(2),       // オーバーフロー・フラグ
                PF      F.bit(2),       // パリティ・フラグ
                NF      F.bit(1),       // 加/減算フラグ
                CY      F.bit(0) ;      // キャリ・フラグ
//  メモリ
        Memory  MEM( 0x0000, 0xffff, 8 ) ;
%end
%initial
{ // メモリ・マップ
        MEM.Assign( 0x0000, 0x1fff, RO ) ;      // 内蔵ROM
        MEM.Assign( 0x2000, 0xfebf, RW ) ;      // 外部メモリ
        MEM.Assign( 0xfec0, 0xffbf, RW ) ;      // 内蔵RAM
        MEM.Assign( 0xffc0, 0xffef, RW ) ;      // Ｉ／Ｏポート
        MEM.Assign( 0xfff0, 0xffff, RW ) ;      // 外部メモリ
}
%end
```

```
%instruction
        var     Register r(0,2)         { B, C, D, E, H, L, A } ;
        var     Pair     qq(0,2)        { BC, DE, HL, , IX, IY, SP };
        var     Pair     qq(0,2)        { BC, DE, HL, , IX, IY, AF };
        Program_Memory          MEM ;
        Program_Counter         PC ;

E0+qq:28+r,             "LD     r,(qq)",
{
        r = MEM[qq] ;
        cycle(6) ;
} ;
50+qq,                  "PUSH   qq",
{
        SP = SP - 2 ;
        MEM[SP] = qq ;
        cycle(8) ;
} ;
F8+g:60,                "ADD    A,g" ,
{
        A = A + g ;
        S = SIGN ;
        Z = ZERO ;
        H = HALF ;
        V = OVERFLOW ;
        N = 0 ;
        X = C = CARRY ;
        cycle(4) ;
} ;
%end
```

図５．１　ハードウェア動作記述言語記述例

このように、ハードウェア動作記述言語はリソースの宣言部とハードウェアの動作の記述部よりなり、リソース部は、宣言形式で明確にリソースの構成を定義し、ハードウェアの動作は、記述項目が設定することにより記述漏れを明確にすると共に、項目毎にその動作手順を手続き的に簡潔に記述できる。

6．実　現

ハードウェア動作記述言語からシミュレータを自動生成する手順を図６．１に示す。図のように、シミュレータをターゲット依存部と非依存部に分け、依存部に付いては、ハードウェア動作記述より生成し、非依存部は、共通利用モジュールとして汎用的に利用する。

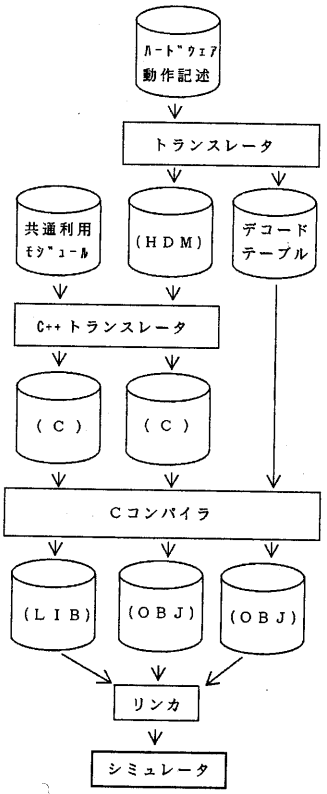

図６．１　シミュレータ生成の流れ

シミュレータ内部のオブジェクト指向の構造を効率よく実現するため、ここでは生成されたシミュレータの性能を考慮してＣ＋＋言語[6] の処理系を用いた。これにより、ハードウェア動作記述からＣ＋＋言語、さらに、Ｃ言語へのソースへと変換するトランスレータ形式とした。ソースレベルのトランスレータにより柔軟性が得られると共に、Ｃ言語の持つ移植性の特徴も利用できると考えられる。

7．評　価

本システムを用いて、当社製の８ビットマイコン(TLCS-90)のシミュレータのエンジン部を開発した。その開発効率を当社製の４ビットマイコン(TLCS-47)のシミュレータの開発経験をもとにそれと比較することにより、シミュレータの生成システムの評価を行なう。

TLCS-90 のシミュレーション・エンジン部の記述は、約4.8K 行となった。このうち、９０％が命令動作記述である。命令動作記述は、各命令コード毎の動作を記述するため、命令数に比例して大きくなる。TLCS-90 の場合約４００命令であるので、１命令あたり１０行程度で記述している。

TLCS-47 のシミュレータは、専用シミュレータとして開発した。ソースはＣ言語で記述し、シミュレーション・エンジン部約６Ｋ、デバッガ部約７Ｋの計１３Ｋステップとなった。ＴＬＣＳ－４７の命令数は、約９０命令である。したがって、ハードウェア動作記述で記述すると約９００行、命令動作記述以外は約６００行程度で記述できるため、ハードウェア動作記述は、約１．５Ｋ程度で記述できると考えられる。

デバッグ機能は、ターゲットに依存せず汎用的に使用できるものとして設計しているため、ハードウェア動作記述のみからシミュレータを生成できる。したがって、通常のプログラミング言語で記述する場合と比べて約１／８の記述でシミュレータを生成することが可能である。

さらに、ハードウェア動作記述言語は、記述項目が機能別に整理されており、機能の変更／修正なども容易に行なうことができる。また、他のターゲットへの対応に関しても、それが以前に記述したものと同一のアーキテクチャを持ったファミリであれば、以前の動作記述を再利用することが可能である。

また、ハードウェア記述言語は、個々の命令動作記

述、割り込み処理動作記述中に任意の機能を手続き的に記述することが出来る。この機能を利用すれば、命令使用頻度、メモリアクセス頻度、割り込み発生頻度等の採取や、カバレージ計測機能等を容易に組み込むことが可能である。

8．まとめ

各種のターゲットに対して、シミュレータを自動的に生成するジェネレータの実現方式について述べた。この方式を実現するに当たり、オブジェクト指向の考え方を導入するとともに、ハードウェア動作記述言語を設計し、この言語からのシミュレータの自動生成方式を考えた。これらの方式によりシミュレータをより効率的に実現することが可能である。

現在、TLCS-90を対象としたシミュレータを開発することによりシミュレータの評価を行なっているが、今後、対象を広げて行くことにより、ハードウェア記述言語の記述能力、シミュレータの開発効率、生成されたシミュレータの性能等についての定量的な評価を行なっていく。また、今回は、アセンブラ言語レベルのシミュレータとして開発したが、今後は、ＯＳシミュレーション機能の組み込みや、高級言語レベルのデバッグ／テスト機能等について検討していく予定である。

参考文献

[1] 高橋、大筆他：ＩＭＡＰシステム(1)-(10)、情報処理学会第３１回全国大会予稿集、pp.429-508

[2] 井上、貫井他：マイコンソフト開発における論理テスト環境、情報処理学会第３５回全国大会予稿集、pp.1085-1086, 1987

[3] 井上、三原：ハードウェア動作記述からのソフトウェア論理シミュレータの生成、情報処理学会第３８回全国大会予稿集、pp.1217-1218, 1989

[4] Myers,G.J.:Advances in Computer Architecture, John Wiley & Sons, 1978

[5] オブジェクト指向プログラミング，情報処理 Vol.29 No.4, 1988

[6] B.Stroustrup: The C++ Programming Language, Addison-Wesley, 1986