社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2006－DPS－128 (17)
2006－GN－ 61 (17)
2006－EIP－ 33 (17)
2006／9／15

Webベースプロセッサ開発環境のためのテスト環境の構築

柳澤 秀明、上原 稔、森 秀樹
東洋大学工学部情報工学科
mist@ds.cs.toyo.ac.jp, uehara@cs.toyo.ac.jp, mori@cs.toyo.ac.jp

プロセッサを開発するためには、ハードウェア（HW：Hardware）の設計のみならず、アプリケーションを開発するためのソフトウェア開発環境（SWDE：Software Development Environment）の構築が必要になる。このため、単一の仕様記述からHW/SWDEの自動生成を行うプロセッサ開発環境が必要となる。我々は、Webベースプロセッサ開発環境であるSSC-DASH(Server Side C-DASH)の開発を行っている。本論文では、Webベースプロセッサ環境におけるテストファースト開発について述べる。テストファースト開発を行うことで、従来のテストパターンを用意する方法と比べ、効果的に開発が行えるようになる。

Constructing Test Environment for Processors Development Environment

Hideaki Yanagisawa, Minoru Uehara, Hideki Mori
Department of Information and Computer sciences, Toyo University

In order to develop processors, development of HW (Hardware) and SWDE (Software Development Environment) to develop application programs, are required. And automatic generation of HW/SWDE for designing processors based on single specification language is necessary. We developed Web-based processor development environment SSC-DASH (Server Side C-DASH). In this paper, we describe test first development for web-based processor development environment. By test first development, processor development becomes easy and rapidly than traditional test pattern development.

1. はじめに

組み込みシステムは、様々なIP (Intellectual Property)を組み合わせて構築する。それらのIPで中心的な役割を果たすのがプロセッサIPであが、単一の目的のために開発されることが多い組み込みシステムでは、汎用的なプロセッサIPを利用した場合、一度も実行されることの無い命令が存在し、不必要なリソースをシステムの中に含むことになる。

コストパフォーマンスを改善するためには、アプリケーション処理に特化したプロセッサである ASIP (Application Specific Instruction set Processor)の開発が必要である。

しかしながら、プロセッサを開発するためには、プロセッサのHW(Hardware)の開発と同時に、アプリケーションを開発するためのSWDE (Software Development Environment)の構築が必要となる。

このため、我々は、プロセッサの命令セット定義記述からプロセッサのHWとSWDEの自動生成を行うことが出来るプロセッサ開発環境（PDE；Processor Development Environment）として C-DASH（C-like Design Automation Shell）[1][2]の開発を行い、さらに、Webベース PDE（WPDE：Web-based Processor Development Environment）である SSC-DASH（Server Side C-DASH）[3][4]の開発を行ってきた。

SSC-DASHでは、プロセッサの命令セット定義から、プロセッサのHWとして論理合成可能なHDL記述(Verilog-HDL)の生成を行い、SWDEとして、シミュレータ、アセンブラ、逆アセンブラ、コンパイラの生成を行うことが出来る。

SSC-DASH を利用したプロセッサ開発では、あらかじめプロセッサの仕様を決め、定義予定の命令セットの中から、1)優先度に従い、一つずつ命令を定義、2)SWDE（シミュレータ、アセンブラ、逆アセンブラ）の生成、3)SWDE のダウンロード、4)アセンブリ言語による、定義命令の動作を確認するプログラムの作成、5)アセンブル、6)シミュレーションの実行（実行結果の保存）、7)実行結果の比較（命令の動作を変更するたびに、以前の動作と）の手順が必要となる。

SSC-DASH では、命令セットの定義ファイルを更新するたびに定義ファイルをアップロードし、定義ファイルに基づき生成された SWDE を毎回ダウンロードしなければならならない。テスト結果は、レジスタやメモリの値を人が確認しなければならない。また、アセンブリ言語を利用したテストであるため、テストプログラムの開発が必要であること、テストプログラム自体にバグが潜むことがあること、バグを見つけるのが困難であることなど、テスト環境の効率化が必要である。

この問題を解決するため、本論文では、WPDE である、SSC-DASH を利用したプロセッサ設計のためのテストファースト開発について述べる。SSC-DASH では、シミュレーションの実装を Java で行っている。そこで、Java 開発で利用されるテストファースト環境である JUnit[12]を利用したテスト環境の構築を行い、動作確認を効率的に行えるようにする。

本論文の構成は以下の通り、2 章で関連研究、3 章で SSC-DASH について述べる。4 章でテストファースト開発について述べ、5 章で従来のアセンブリ言語でのテスト方法とテストファースト開発での比較を述べる。そして 6 章で本論文のまとめを述べる。

2. 関連研究

プロセッサ開発環境は、（1）命令セットの動作のみを定義する（nML[5][6]）、（2）プロセッサのアーキテクチャを定義する（MIMOLA[7][8]）、（3）命令セットの動作とアーキテクチャを記述する（ASIP Meister[9]、LISA[10][11]）の 3 つのタイプに分けることができる。

（1）では、パイプライン段数などハードウェアの仕様を定義することは出来ないが、動作レベルのシミュレータを生成する場合などには適している。（2）は、ハードウェアの仕様を細かく定義する場合に適しているものの、（1）と比べると記述が複雑になり、理解し難くなる。（3）は、（1）と（2）を組み合わせたようなものであり、命令セットの動作記述とハードウェアのアーキテクチャの詳細記述の両方を行い、動作定義は、クロックベースで行われる。

これらの開発環境では、ローカルで開発を行い、テストパターンを用意してテストを行っている。

本研究では、Web ベースでプロセッサの開発を行う。このため、Web ベースでのプロセッサ開発のためのテストファースト開発環境の構築を行う。テストファースト開発によりテストプログラムを作り、テストプログラムを実行しながら開発を行うことで、命令セットの動作が分かりやすくなり、動作確認が効率的に行えるようになる。

3. SSC-DASH

SSC-DASH は、PDE である C-DASH を Web ベースで利用するためのサーバとして Java サーブレットにより実装されている。

プロセッサ開発を、WPDE とすることで、ブラウザとテキストエディタさえあれば、開発者のローカル環境に依存することなく、どこからでも開発に参加できる。また、リソースをサーバで集中管理しているため、いつでも、最新リソースにアクセスすることができる。

SSC-DASH は C-DASH を呼び出すことで、アップロードされたプロセッサの定義ファイルからシミュレータ、アセンブラ、逆アセンブラ、コンパイラ、プロセッサの HDL 記述などを提供する。

3.1 SSC-DASH の機能

C-DASH サーバは、ユーザ管理、プロジェクト管理、サブプロジェクト（開発者が自由に使える空間を提供）、ファイル管理（CVS を利用）、アクセスコントロール、情報管理（開発、バグ、修正情報など）、ライブラリ管理、ファイルリリース（シミュレータ、ソースファイルの公開を行う）などである。

3.2 C-DASH

C-DASH は、プロセッサを記述するための専用言語 CHDL（C-DASH HDL）を用い ISA(Instruction Set Architecture)を基本としたプロセッサの定義を行う。CHDL 記述でのプロセッサ開発では、リソースの宣言（レジスタやメモリ）と命令セットの定義を行う。

C-DASH では、Behavior（動作記述、HW の生成は行われない）、Sequential（クロックベース記述、生成される HW は、フェッチ、デコード、実行、ライトバックの 4 ステージで構成）、Pipeline（クロックベース記述、及び、ステージ定義）の 3 つの記述レベルをサポートしている。

C-DASH は、CHDL でのプロセッサ記述から、HW（論理合成可能なプロセッサの Verilog-HDL 記述）と、生成したプロセッサのための SWDE（シミュレータ、アセンブラ、逆アセンブラ、コンパイラ）の自動生成が可能である。C-DASH を利用することで、単一のプロセッサ記述から、HW と SWDE を自動生成することができるためプロセッサ設計者の開発負担を減らすことができる。

また、ISA でプロセッサを定義しているため、命令セットの変更を簡単に行うことができる。

3.3 プロセッサ定義

CHDL で、プロセッサを定義するためには、リソース（メモリ、レジスタなど）の定義と命令の動作定義が必要である。ビヘイビア記述でのプロセッサ定義例として、加算命令（ADDA）の動作記述を図 1 に示す。

```
reg pc 16;
reg gr 16 16;
ram m 16 65536;
instructin m pc;

instruct adda {0010 0000 dddd ssss} {
    gr[d] = gr[d] + gr[s];
}
asm adda {<label> <opcode> <reg> , <label.address>} {
    d = number($3);
    s = number(&5);
}
```

図 1　加算命令の動作記述

CHDL では、"reg" でレジスタのビット幅や個数を定義する。また、"ram"によりメモリのビット幅とサイズを定義し、命令メモリとプログラムカウンタを"instructin"により明示する。

各命令の動作は、"instruct"命令により定義する。CHDL では、ニーモニック名（adda）、ビットパターン（0010・・・・）と命令の動作を定義し、命令のフォーマットを"asm"命令により定義する。"<label>"は、アセンブリ言語のプログラム中で使われるラベルを表し、"<opcode>"でニーモニック名を表す。"<reg>"でオペランドがレジスタであることを表し、"<label.address>"でオペランドが即値または、ラベルであることを表す。"asm"中の"d","s"は、"instruct"のサブビット"dddd"、"ssss"をそれぞれ表し、"number()"でレジスタ番号を"address()"で即値やアドレスをサブビットに埋め込むことを表している。"$N"（N=3,4…）は、","も含めた番号を表している。

3.4 ISA シミュレータ

ISA シミュレータの実行速度は、記述レベルに依存する。詳細な記述を行うシミュレーションほど、実装速度は遅くなる。このため開発レベルに応じたシミュレータの生成とテスト環境が必要である。

C-DASH では、インタプリタ型 ISA レベルシミュレータ（Java）とアーキテクチャレベルシミュレータ（SpecC）の生成が出来る。また、コンパイル型シミュレーションを可能にするためのシミュレーションコンパイラの自動生成も行うことが出来る。

このため、命令セットの動作確認や、アプリケーション開発のデバッグでは、インタプリタ型、アプリケーションの実行確認では、実行速度の速いコンパイル型と用途に応じて使い分けることが出来る。

本研究では、インタプリタ型シミュレータを利用し Web ベースでのテスト環境を実現する。インタプリタ型シミュレータの実装例を図 2 示す。

```
/**
*     machine code 1010 0000 dddd ssss
*     mask         1111 1111 0000 0000
*/

void fetch() {
        ir = mem[pc];
        pc = pc + 1;
}
void exec () {
        if((ir & ADDA_MASK) == ADDA_OP) {
                d=getSubbit(ir,7,4);
                s=getSubbit(ir,3,0);
                adda(d,reg_file[d],reg_file[s]); }
        if((ir & INST_N_MASK)==INST_N_OP) {
                INST_N(); }
}

void adda (int d. int src_1, int src_2) {
        //
        alu_out = src_1 + src_2;
        flag_set(alu_out,src_1,src_2);
        reg_file[d] = alu_out;
}
```

図 2　インタプリタ型シミュレータ

シミュレータは、fetch-exec モデルで実装している。fetch では、pc が示すアドレスか

ら次に実行する命令を取り出し、ir レジスタにセットし、pc を増加させている。

exec では、ir と ADDA_MASK の and をとり、ADDA_OP と等しいときサブビットを取り出しオペランドをセットし命令の動作を呼び出している。

3.5 SSC-DASH でのテスト環境

SSC-DASH でプロセッサを開発するためには、命令の定義ファイルの作成、定義ファイルのアップロード、SWDE の生成、SWDE のダウンロード、定義した命令を利用したアセンブリ言語でのテストプログラムの作成、アセンブル、シミュレータを利用した動作確認を繰り返し行う。命令の動作を修正した場合も、この処理を繰り返し行う必要がある。

SSC-DASH のテスト環境では、以下の問題点がある。

- アセンブリ言語でテストプログラムを開発しなければならない
- アセンブリ言語でプログラムを記述し、テストデータを生成しているため、テストプログラムにバグが存在する場合がある
- アセンブリプログラムに含まれるバグを発見することは難しい
- プログラムとしての動作確認が必要であり、命令ごとの動作確認が難しい（例、RISC アーキテクチャでは、加算命令の動作確認を行うために、ロード命令で、レジスタの値をセットした後に加算命令の実行を行う必要があり、加算命令だけをテストすることができない）
- プログラムの実行結果の確認は、レジスタやメモリの値を一つ一つ人が確認する
- このため、複数の命令の動作を確認することが難しい
- 定義ファイルの更新のたびにシミュレータをダウンロードしなければならない

この問題点を解決するための必要なテスト機能として以下を考慮する必要がある。

- 命令単位での動作確認
- 複数の命令を実行したときに影響を及ぼさない
- テストの自動化

4. テストファースト開発

C-DASH では、シミュレーションの実装を Java で行っている。このため、テストファースト環境である JUnit を利用することで WPDE のためのテストファースト開発環境の構築を行う。

4.1 SSC-DASH でのテストファースト

テストファースト開発環境における開発の流れを図 3 に示す。

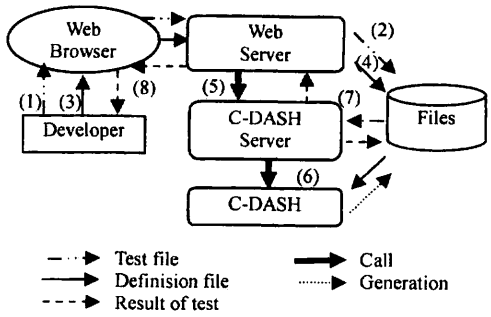

図 3　テストファースト開発の流れ

開発者は、(1)各命令の仕様の確認が行えるテストプログラムをアップロードする。(2)アップロードされたテストプログラムを保存する。(3) 開発者がプロセッサ定義ファイルをアップロードする。(4)プロセッサ定義ファイルを保存する。(5)C-DASH サーバを呼び出す。(6)C-DASH を実行し、定義ファイルからシミュレータを生成。(7)生成されたシミュレータとテストプログラムからテストを実行。そして、(8)テスト結果の確認を行う。それぞれの命令のアップロードを一つ一つ行っていく。

4.2 テスト記述

本研究では、JUnit を利用したテストファースト開発環境の実装である。このためテスト記述は、入力データ、期待値、失敗時のメッセージを行う記述とする。JUnit を利用した加算 ADDA のテスト記述を図 4 に示す。

```
/**
*       format      adda reg_1, reg_2
*       behavior    reg_1 = reg_1 + reg_2
*/
test adda {
        //
        gr[1]=3;            gr[2]=4;
        //
        adda(1,gr[1],gr[2]);
        //
        assertEquals("Error Message",7,gr[1]);
}
```

図 4　テスト記述

表 1　テスト環境の比較

	アセンブリ言語	テストファースト
テスト方法	シミュレータのダウンロード	テストパターンのアップロード
テスト単位	シミュレータ全体	命令メソッド
テスト記述レベル	低水準	高水準
動作確認のしやすさ	難しい	易しい
実行結果	レジスタ、メモリの値の変化	期待値との比較結果
実行確認	人がレジスタ、メモリの値を確認	真 Yes・偽 No
定義ミスの発見	難しい	易しい
テストプログラムにおけるバグ	高	低
テストプログラムにおけるバグの見つけやすさ	難しい	易しい
実行結果	レジスタ、メモリの値	期待値との比較結果
実行確認	人がレジスタ、メモリの値を確認	真 Yes・偽 No
ドキュメント性	低	高

テストプログラムでは、レジスタ"gr[1]"に3、"gr[2]"に 4 をセットし"adda"の命令を呼び出している。1 番目の引数の"1"は実行結果の出力先のアドレスを指定しており、実行結果が"gr[1]"にセットされることを表す。"assertEquals"により実行結果である"gr[1]"の値が期待値"7"と等しい場合は成功、それ以外のときは、エラーメッセージを出力する。

プロセッサ定義自体には、コメントをつけず、テスト記述に詳細なコメントをつけることで、仕様書として利用することが出来る。

5. 比較

今までのアセンブリ言語を用いたテスト開発と本論文で述べたテストファースト開発での比較を述べる。

テストパターンの開発の違い、アセンブリ言語を利用したテストパターン開発では、アセンブリ言語のプログラムを記述、アセンブルし実行コードの生成、シミュレータの実行、レジスタやメモリの変更を調べながら実行結果の確認、の作業を繰り返し行わなければならなかった。

テストを実行するときの単位は、アセンブリ言語でのテスト開発では、シミュレータ全体のテストになってしまうが、テストファースト開発では、定義命令単位でテストを行うことが出来る。このため、命令の定義ミスが発見しやすい。

アセンブリ言語でのテストプログラム開発は、テストプログラムにバグが潜む可能性がある。また、バグを発見が困難であった。

テストファースト開発では、テストパターンを高級言語で記述する。テスト結果の期待値を設定するため、実行結果が真・偽で判断できる。テストプログラムにバグがあったとしても発見しやすい。

アセンブリ言語で記述されたテストパターンでは、内容を即座に理解できない場合が有るが、高級言語を利用することでテストデータの可読性が向上した。細かなコメントをつけることで仕様書としての利用が可能になる。

これまで、述べたことを表 1 にまとめる。

今までは、定義した命令の動作を確認するために、定義ファイルをアップロードし、SWDE の生成を行い、シミュレータをダウンロードし、テストの実行を行っていた。

確認したい命令が有っても、その命令だけを単体でテストすることが出来ない場合場もあり、テストに時間がかかってしまうこともあった。

テストファースト開発環境とすることで、命令ごとの動作確認が行えるようになって、実行結果が真、偽で判断できるようになりテスト結果の判断が分かりやすくなった。

6. まとめ

本論文では、Web ベースプロセッサ開発環境（WPDE：Web-based Processor Development Environment）である SSC-DASH(Server Side C-DASH)における、テストファーストプロセッサ開発について述べた。

今までの SSC-DASH では、Web ブラウザ

を用いサーバにアクセスし、命令セット定義ファイルをアップロードし、SWDE の生成とダウンロードを行い、アセンブリ言語で記述したテストプログラムを用いプロセッサの開発を行っていた。テスト結果の確認は、実行ログのレジスタやメモリの値を人の目で、一つ一つ確認していた。

アセンブリ言語によるテストプログラム開発では、プロセッサの動作確認以外にもプログラム自体の動作に間違いが無いのかも注意しなければならなかった。また、アセンブリ言語で記述したテストプログラムの中からバグを見つけるのは難しい。

この問題を解決するため、本研究では、JUnit を利用したテストファースト開発手法を取り入れプロセッサの開発に利用した。

テストファースト開発を行うことで、今までは、実行結果を一つ一つ人がチェックしなければならなかったものを実行結果の期待値を設定することで、実行結果の真偽を自動的に判定できるようになった。

今後の課題として、テストファースト開発では、テストプログラムを仕様書のように利用することができるので、テストプログラムからマニュアルの自動生成を行いたい。

参考文献

[1] Hideaki Yanagisawa, Minoru Uehara, Hideki Mori, "A Processor Development Environment C-DASH", International Journal of Computer Science and Network Security (IJCSNS), Vol. 6, No. 1, pp.227-233, 2006.

[2] Hideaki Yanagisawa, Minoru Uehara, Hideki Mori, "Development Methodology of ASIP based on Java Byte Code Using HW/SW Co-Design System for Processor Design", In Proc. of The 1st IEEE International Workshop on Embedded Computing Systems (ECS'04), pp. 831-837, 2004.

[3] Hideaki Yanagisawa, Minoru Uehara, Hideki Mori, "Web-based Collaborative Development Environment for Designing Processors", GESTS International Transactions on Computer Science and Engineering, Vol. 30, No. 1, pp.121-131, 2006,4

[4] 柳沢秀明, 上原稔, 森秀樹, "ISA シミュレータ開発のための Web ベース共同開発環境", 情報処理学会 マルチメディア, 分散, 協調とモバイル (DICOMO2006) シンポジウム論文集, pp.33-36, 2006,7.

[5] A. Fauth, J. Van Praet, M. Freericks, "Describing Instruction Set Processors Using nML", In Proc. of IEEE, European Design and Test Conf., 1995.

[6] Mark R. Hartoog, James A. Rowson, Prakash D. Reddy, Soumya Desai, Douglas D, "Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign", In Proc. of ACM Design Automation Conference, 1997.

[7] R. Leupers and P. Marwedel. "Retargetable code generation based on structural processor descriptions", Design Automation for Embedded Systems, 1998.

[8] Steven Bashford, Ulrich Bieker, Berthold Harking, Rainer Leupers, Peter Marwedel, Andreas Neumann, Dietmar Voggenauer, "The MIMOLA Language Version 4.1", University of Dortmund, September 1994.

[9] Makiko Itoh, Yoshinori Takeuchi, Masaharu Imai, Akichika Shiomi, "Synthesizable HDL Generation for Pipelined Processor from a Micro-Operation Description", IEICE Trans. Fundamentals Vol. E83-A, No.3, pp.394-400, March 2000.

[10] A. Hoffmann, O. Schliebusch, A. Nohl, G. Braun, O. Wahlen and H. Meyr, "A Methodology for the Design of Application Specific Instruction Set Processors (ASIP) Using the Machine Description Language LISA", In Proc. of International Conference on Computer Aided Design (ICCAD), November 2001.

[11] A. Hoffmann, A. Nohl, S. Pees, G. Braun, and H. Meyr, "Generating Production Quality Software Development Tools Using a Machine Description Language", In Proc. of International Conference on Design Automation and Test in Europe Conference (DATE), 2001.

[12] JUnit, http://www.junit.org/index.htm