計算機アーキテクチャ 134−10
（1999. 8. 2）

アセンブリ言語仕様に基づくCPUの設計

柳澤 秀明、森 秀樹、上原 稔
東洋大学工学部情報工学科

半導体集積回路 VLSI の集積度の発展は目覚しく現在でも増加している。VLSI の集積度の向上が進み、複雑な機能を持ったものでも１つのチップ上にシステムを搭載 (SO C:System On Chip) できるようになってきている。しかし、システムの設計が大規模で、複雑、困難になったにもかかわらず、今まで以上に短期間で開発することが求められている。そこで、我々は設計時間を短縮するために、CPU の開発を目的とした言語レベルによる設計を行ない、ハードウェアの生成と同時に、目的とする CPU のソフトウェア開発環境を同時に生成し、いままでの HDL による CPU の設計よりもより簡単で、短期間に行なうことを目的とする。

Design of CPU based on language level

Hideaki YANAGISAWA, Hideki MORI, Minoru UEHARA
Department of Infomation and Computer Sciences, Toyo University

The development of VLSI system is advanced rapidly. So that high performance hardware can be implemented easily. As long as VLSI density is increased, it's software development is going to be complicated and difficult in short term. In the decade of SOC(System On Chip),it is difficult to design one VLSI chip from begining. In this paper to solve the problem above mentioned, we are developing C'(C-like Design Automation Shell) aiming to generate CPU hardware and software both. In our design, using C', both of the CPU hardware and the software are generated simultaneously. We show how our design is performed easier and implemented in shorter time compared with the conventional CPU design by HDL

1 はじめに

半導体集積回路（VLSI）の集積度の向上はとても速く、この傾向は今後も続き、ますます増加すると思われる。VLSI 集積度の向上にともないハードウェアの性能は向上し、VLSI の集積度の向上、ハードウェアの動作速度の高速化、ダイサイズの小型化が進んでいる。また、台サイズの小型かに伴い製品コストの抑制、低消費電力化がなされている。

これらの理由により、ますます利用範囲が広がり、様々な種類の CPU を短期間で開発することが求められている。半導体集積度の向上にともない、システム開発が複雑、困難になってきているにも関わらず今まで以上に開発時間を短縮することが求められ設計の再利用をする為に VSI(Virtual Socket Interface) 化が行なわれたり、IP(Intellectual Property) の有効利用が進められている。

1.1 HDL による設計

現在のハードウェアの設計は動作レベルで記述、検証することができるハードウェア記述言語 (HDL : Hardware Description Langage) を用いて行なわれている。

HDL の設計は、以下のような手順で行なわれる (図 1)。

1. 設計仕様の決定。
   - ソフトウェアインタフェース設計。
   - 外部インタフェース設計。
   - アーキテクチャ設計。

2. 動作レベル記述。
3. テスト。
4. RTL 記述。
5. テスト。
6. 論理合成。
7. テスト。
8. ハードの実現。

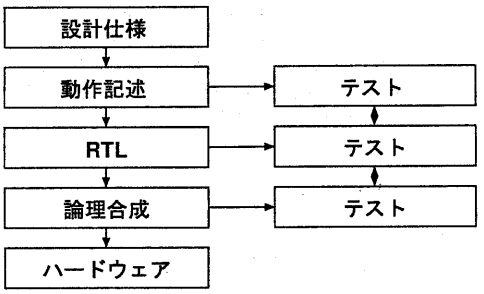

図 1: HDL を用いた設計

ハードウェアの機能テストは動作レベル、RTL、ゲートレベルの各段階で、考えられる全ての bit パターンをテストパターンとして用意し動作確認を行わなければならない。しかし、大規模化、複雑化しつつある回路のテストパターンを用意して期待通りの動作をしているかを確認することは困難であり、テストパターンを用いた検査では間違えに気付きにくく完全に思われるテスト後でもハードウェアの設計ミスが発見されることもある。開発時間を短縮化するためにはできるだけ後戻りを少なくすることが必要であり、実機テストの段階で設計ミスが見つかると大きな無駄が生じる。

1.2 ソフトウェアの開発

ソフトウェア開発環境の構築は実機でのテストが中心に行なわれている。ハードウェアが完成してからソフトウェア開発環境を構築するためハードウェア設計からすぐに製品化することができないでいる。ハードウェアである CPU は、ソフトウェアにより制御する必要があるが、大規模、複雑化しつつあるハードウェアを制御するようなプログラムを作るのはとても時間がかかり、大きな負担になっている。CPU の開発時間を短縮するためには、制御プログラムを生成をする為の負担の割合が大きくなり、制御プログラムを簡単に書くためのプログラミング環境の開発を短期間で行なうことが求められる。しかし、機械語はそれぞれの CPU によって異なるためにそれぞれの CPU に対してソフトウェア開発環境（アセンブラ）を作ることはとても大変である。また、変化の早い市場に対応するために設計仕様の変更を必要とすることもある。仕様変更が行なわれた場合、ハードウェアの変更に合わせソフトウェアを変更しなければならない。

本研究では、多種多様な高性能 CPU を言語レベルでの設計によりハードウェアの開発とソフトウェアの開発環境生成を同時に行なう。この時、ソフトウェアの論理的機能検証を行ない正しく動作ができることを確認してから HDL 記述に変換することで目的とする動作を最低限保証したハードウェアの開発を行なうことを目的としている。

2 言語レベル設計

言語レベルによる設計では、設計しようとしている CPU のアセンブラの仕様記述に基づき、HDL によるハードウェアの設計と、プログラミング環境であるアセンブラ、逆アセンブラ、ソフトウェアシミュレータの開発を同時に行なう。

従来の HDL を用いたハードウェアの設計では、設計仕様書に基づき、ハードウエアの設計とソフトウェア開発を独立して行なっている。ハードウエアの設計とソフトウェアの設計を独立して行なっているために仕様変更が行なわれると、ハードウェアの変更を行ない、変更に合わせてソフトウェア開発環境の変更も必要になり、ハードウェアとソフトウェア開発環境をそれぞれ作り直さなければならないので、とても大きな負担となる。(図 2)

言語レベル設計では、アセンブラの仕様を記述しハードウェア（HDL）とソフトウェア開発環境の生成を同時に行なうので、仕様変更を行なう時でも簡単に対応することができる。(図 3)

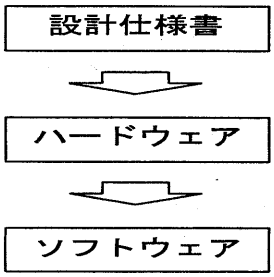

図 2: HDL を用いた開発

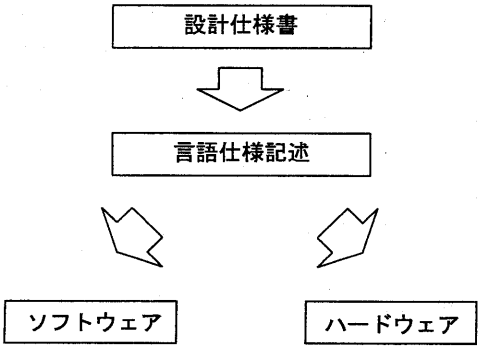

図 3: 言語レベルでの開発

3 C'

3.1 概要

C'(C-like Design Automation SHell) は、CPU の開発を目的とした設計自動化システムである。VLSI は、HDL(Hardware description language) により設計が行なわれている。また、ソフトウェア開発環境は、ハードウェア製造後、実機でのテストを中心に開発が行なわれている。(図 2)

VLSI 製造技術が向上するにつれて、ソフトウェアの開発にかかる負担の割合が大きくなってきている。CPU をより短期間で開発するためには、ソフトウェアの開発時間を短くすることが求められている。

C' ではビット列とニモニックを対応させアセンブラの仕様を C 言語に似た文法で記述することで、アセンブラ、逆アセンブラ、ソフトウェアシミュレータの自動生成を可能にしている。また、アセンブラの仕様記述から HDL への変換を行なう。(図 4)

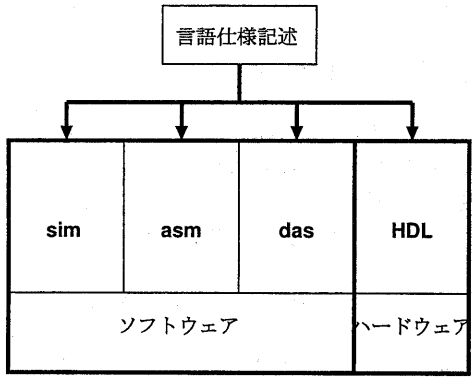

図 4: C' の概要

3.2 C' での設計

C' での設計手順を示す。(図 5)

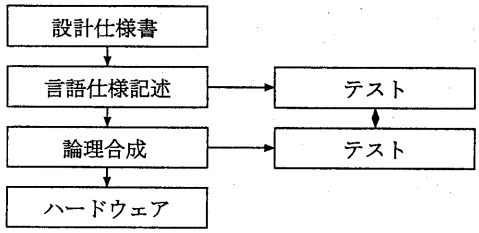

図 5: 言語レベルでの CPU 設計

1. 設計仕様の決定。
2. 言語仕様記述。
3. テスト。
4. 論理合成。
5. テスト。
6. ハードの実現。

アセンブラの仕様記述より、アセンブラ、逆アセンブラ、ソフトウェアシミュレータを生成し、ア

センブリ言語で記述したプログラムにより、目的とするハードウェアの動作確認を行なうことができる。そのために望みの動作を確認しながら設計を進めることができる。

3.3 C'コマンド

C'はスクリプト言語である Tcl/Tk を用い、Tcl/Tk コマンドの集まりを C'コマンドとして定義して使う。

C'コマンドは以下の種類に分類される。

- シミュレータコマンド (instruct)
- アセンブラコマンド (asm)
- 逆アセンブラコマンド (das)

各コマンド（instruct、asm、das）内では、C 言語に似た記述 (while,if,else、…) で動作を記述する。また、この他に C' の記述を補助するコマンド群がある。

3.4 組み込み関数

以下の組み込み関数を instruct コマンド内で使用できる。

finish() 実行プロセスを終了する。

subbit(x,n,m) レジスタ x の n ビットから m ビットの間の n-m+1 ビットの値を返す。ビットは下位から 0 で始まる。

catbit(a,as,ae, b,bs,be) 二つのレジスタ a, b の部分ビットを連結して得られる as-ae+bs-be+2 の長さのビット列を返す。

setbit(x,s,e,v) レジスタ x の部分ビットにレジスタ v のビットパターンを当てはめる。

3.5 記述例

記述例として 8080 の add 命令を示す。

シミュレータ

instruct $\underbrace{add}_{\text{ニモニック}}$ $\{\underbrace{10000fff}_{\text{ビット列}}\}\{$

```
    if (f==7) a=a+a;
    if (f==0) a=a+b;
    if (f==1) a=a+c;
    if (f==2) a=a+d;
    if (f==3) a=a+e;
    if (f==4) a=a+h;
    if (f==5) a=a+l;
    if (f==6)
        a=a+m[catbit(h,7,0,1,7,0)];
}
```

a,b,c,d,e,h,l は、レジスタ、f は機械語となるビット列と対応する部分ビットであり、レジスタ番号を示す。レジスタ番号によってどのレジスタとの加算かを決める。

アセンブラ

asm $\underbrace{add}_{\text{ニモニック}}$ $\{\underbrace{< label >< opcode >< reg >}_{\text{書式}}\}\{$

```
    f=number($3);
}
```

アセンブラ生成コマンドではニモニックの書式を定義する。label はアセンブラ記述で使われるラベルを表し、opecode はニモニック、reg は、レジスタを表す。

逆アセンブラ

disasm $\underbrace{add}_{\text{ニモニック}}$ {

```
        printf("%s\t%f",opcode,f);
}
```

逆アセンブラ記述では、printf 関数で、ニモニックと、レジスタを出力する。

4 ライブラリ

C' ではライブラリ化された部分があるので、Verilog での動作記述をあらかじめ用意しておきそれを必要に応じて書き出す。現在ライブラリ化するしている関数として次のようなものがある。

apply_reset ハードウェアの初期化。

prog_load 実行ファイルをメモリに読み込む。

fetch プログラムカウンタが指すメモリの内容を ir レジスタセットしプログラムカウンタを進める。

decode 命令の解析。

execute 命令の実行。

write_result 実行結果を書き出す。

main_process fetch,execute,write_result を呼び出す。

5 Verilog記述の生成

言語仕様記述を Verilog 記述に変換してハードウェアの生成を行なう。(図 6)

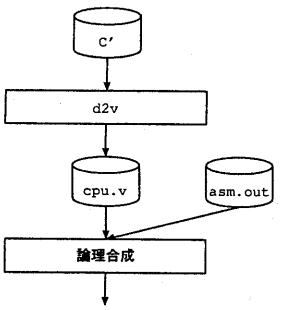

図 6: d2v による変換

5.1 宣言部の変換規則

- module risc ⇒ module sisc ;
- reg ir 32; ⇒ reg [31:0] ir;
- ram m [W][L] ⇒ reg [W-1:0] m [L] にする。

Verilog の記述では、システム定数の定義を parameter を使って行ないそれ以外の定数の定義を 'define を使って行なうことで区別している。

C' では、fetch,execute 部分をライブラリ化し、自動で書き出しすので、ニモニックの動作を記述するだけですむ。

5.2 動作部を変換規則

- subbit(reg,S,D) ⇒ reg[S:D]。
- c-func int ⇒ function [31:0]。
- int in ⇒ input [31:0] in;。
- report() ⇒ report;。
- ブロックを begin と end で囲む。
- main_process を書き出す。

Verilog 記述では、function の戻り値が何ビットになるかを指定しなければならない。関数の引数は、input で宣言して、使う変数が何ビットかを指定する。

Verilog では、関数宣言をするのに function と task があり、戻り値があるもの (function) と戻り値がない (task) で使い分けている。

予め C 言語で使われる関数名 (if,for,int$\cdots$) や固有の関数名や変数名を記述して d2v.l とする。また、C ' から Verilog-HDL への変換規則を構文解析プログラム d2v.y として作り、C ' から、Verilog への変換プログラム d2v を作る。

C ' では、c-func を使い C 言語でのプログラムをサブルーチン化して使うことができる。また、Verilog 固有の記述を行ないたい時は、v-task または、v-func を使うことで記述を行なうことができる。c-func で使われる関数名と、同じ名前のときは v-task または、v-func を優先させて出力する。v-func,v-task は、Verilog-HDL の function,task を記述するために使い、変換せずにそ出力する。

```
c-func void c-func_name {int } {
    C 言語の記述。
}
v-task task_name {input [31:0]test} {
    Verilog の task 記述。
}
v-func func_name {input [31:0]test} {
    Verilog の function 記述。
}
```

6 評価

簡単などうさのCPUを定義して実際に記述しどのような結果になるかを調べた。

今回行なったCPUの記述は

評価するためのサンプルプログラムとして簡単な11個の命令セットからなるRISC CPU[3, 5]とCASL[6],8080を用いC'記述での設計を行なった。

表 1: 評価表

	RISC	CASL	8080
C' 記述	234	264	690
シミュレータ	297	447	1120
Verilog-HDL 記述	435	758	1843

7 結論

7.1 まとめ

VLSIの集積度が上がり、ハードウェアの設計が複雑で、困難になる中でVSI化が進められたり、IPの利用が行なわれている。設計が複雑になるにつれハードウェアの設計は最初から作るのではなく、以前に設計されたものを再利用して短期間で開発するが考えられている。

本研究では、複雑で、困難になったハードウェアの設計をより簡単に行なうために、言語レベルでのCPUの設計を考えている。言語レベルによるCPUの設計ではのアセンブラの仕様を基にCPUを簡単に定義し、ソフトウェアシミュレータ、アセンブラ、逆アセンブラを生成し、ソフトウエア開発環境を整え、目的とするCPUの動作をアセンブリ言語で記述し、論理的に正しく動作することを確認できる。

また、あらかじめ用意しておいたライブラリを必要に応じて選択しアセンブラの仕様記述からHDL記述に変換することでハードウェアの自動生成を行う。自動生成されたHDL記述のシミュレーションを行ない、HDLでのシミュレーション結果とソフトウェアシミュレータでのシミュレーション結果を比較して確かめることで必要な動作を最低限保証したCPUをより効率的に開発を行なうことができる。

7.2 今後の課題

今後の課題として高速化や、より効率的に開発するためにソフトウエア開発環境をなどがある。

- Cコンパイラーの生成。
- 各ニモニックのステップ数を解析しパイプライン化を行なう。
- スーパースカラに対応。
- VLIWに対応。

参考文献

[1] R.Lipsett, C.Schaefer and C.Ussery 著杉山尚志/増田洋一郎/新妻靖明/金沢彰 訳、「VHDL: 言語記述によるハードウェア設計へのアプローチ」マグロウヒル出版株式会社、1990.

[2] 中村行宏/小野定康「ULSIの効果的な設計法」オーム社、1994.

[3] E.Sternheim/R.Singh/R.Madhavan/Y.Trivedi 著井上博史/鈴木隆 訳「Verilog-HDLによるトップダウン設計」QC出版、1995.

[4] 小林優 著「入門Verilog-HDL記述」QC出版、1996.

[5] 桜井至 著「HDLによるデジタル設計の基礎」テクノプレス、1997.

[6] 重井芳春 著「計算機工学の基礎」近代化学社、1990.

[7] 深山正幸/北川章夫/秋田純一/鈴木正國 著「HDLによるVLSI設計」共立出版株式会社、1999.