CPU設計言語C'によるスタックマシン記述

3W−05

柳澤秀明　上原稔　森秀樹
東洋大学工学部

1. はじめに

CPUを開発するためには、HW（Hardware）を設計するだけでなくSW（Software）を開発するためのSW開発環境を用意する必要があり、開発のための負担が大きくなっている。システムLSIなどの開発では、開発時間を短縮するために既存のCPUを利用しているが、現在のFPGA/PLDなどの集積度の向上などから、目的に合ったCPUの開発ができれば、より効果的な利用が可能であると考えCPUの開発を目的とした設計言語C'（C-Like Design Automation Shell）の開発を行っている。
本論文では、C'でのスタックマシンの記述について述べる。

2. C'（C-Like Design Automation Shell）

C'（C-Like Design Automation Shell）[1][2]では、命令セットアーキテクチャ(ISA)レベルでCPUコアの設計を行う。C'では、図1のように一つの定義ファイルからHW部であるHDL（Verilog-HDL）記述とSW部であるアセンブラ、シミュレータ、逆アセンブラを自動生成する。

C'では、ひとつの定義ファイルからHWとSW開発環境の自動生成が可能であるためSW開発をすぐに開始することができる。また、HWとSWをひとつの定義で行っているのために仕様変更などに対応しやすくなっている。

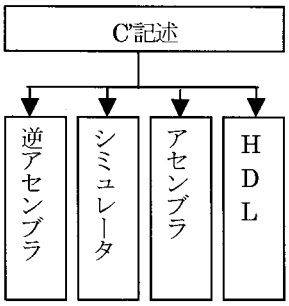

図1　C'の概要

2.1. C'でのCPU設計

C'でのCPU設計は、CPUの命令セットのニーモニックとビット列とその動作を記述して定義する。また、必要に応じてレジスタとメモリの宣言を行う。記述例を図2に示す。

```
module stack_cpu
reg pc 10;              #レジスタ宣言
ram mem 24 1024;                #メモリ宣言

instruct load {ビット列の定義}{
        動作定義
}
asm load {<label>, <opecode>, <label.address>}{
  d=address($5);      #d が数値であることを示す
}

endmodule
```

図2　C'でのCPU記述

2.2. 動作確認

C'での動作確認は、アセンブリ言語でのプログラムの作成を行い、C'から生成されたアセンブラとSWシミュレータ（C言語で生成される）でシミュレーションを行い動作確認した後、HWシミュレーション（Verilog-HDL記述）で実際のHWの動作確認を行う。

3. スタックマシン記述

スタック機構では、ランダムアクセスが不可能であるために効率のよいコード生成が困難であることや、スタックがボトルネックとなり、効率のよいハードウェアの実現が困難であること、また、データを読み込むとき、そのデータはスタックから取り除かれるため、複製を行わなくてはならないなどの問題点がある。しかし、スタックのトップを暗黙のオペランドとしているために、コード中のオペランド数を減らすことができるので、コードサイズを小さくすることができる。スタックマシンの短形式コード命令によりネットワークからダウンロードしてプログラムを実行するようなけ形態に適していること

からJavaChipの実装などで利用される。

スタックマシンのloadとadd命令のC'記述を図3に示す。この記述では、命令長24ビットの固定長であり、load命令のビット列は、オペコード8ビット（11010110）とdで16ビット長のデータを表している。

```
instruction load {11010110 dddddddd dddddddd}
{
  FD:{}
  EX1:{stack[sp+1]=d;sp=sp+1;
       goto FD;}
}
instruction add {00010011 00000000 00000000} {
  FD:{}
  E1:{tmp1=stack[sp];sp=sp-1;
       goto EX2;}
  E2:{tmp2=stack[sp];sp=sp-1;
       goto EX3;}
  E3:{stack[sp+1]=tmp1+tmp2;
      sp=sp+1;goto FTDC;}
}
```

図3　スタックマシンの記述例

3.1. パイプライン化

C'でのパイプライン化は、FSM命令でステージの宣言（FD、E1,E2,E3）とステージごとの動作を記述する。各ステージは1クロックで動作を完了する。図3の記述では、FDは命令フェッチとデコードを行うステージであり全ての命令に共通となる動作である。各ステージでの基本動作はFSM命令で記述し、命令ごとに記述する必要はない。E1,E2,E3は実行ステージであり命令ごとに異なるため各命令の記述の中に個別に動作を記述する。FSMの記述例を図4に示す。

```
FSM {
  FD : {fetch(ir,pc,mem);decode(ir);
        if(add||.....) fd_en=0;}
  E1 : {}
  E2 : {fd_en=1;}
  E3 : {}
}
```

図4　FSM記述

load命令は、FDステージでフェッチとデコードを行い、E1ステージでスタックにデータを保存し、スタックポインタをインクリメントする。add命令では、E1とE2ステージでスタックから値を取り出し、E3ステージで演算を行い、結果をスタックにもどす。

3.2. パイプラインハザード

パイプライン化を行うためには、パイプラインハザードの回避を考えなければならない。C'では、FSM命令を使いハザードの回避を記述する必要がある。例に上げ2つの命令では、先に実行中の命令がスタックに書きも込む命令（add）であり、スタック書き込みが終了するまで後続命令を停止させる記述を行わなければならない。図5の様に動作させるためには、図4の様に記述する必要がある。

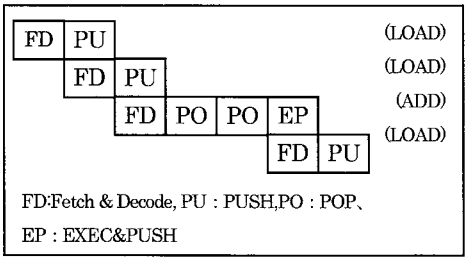

図5　命令実行の流れ

4. まとめ

本論文では、C'によるスタックマシン記述について述べた。C'では、シミュレータ、アセンブラが自動生成されるので動作確認が簡単に行える。今後の課題としてパイプラインハザードの自動検出やスタックマシンの高速化がある。

参考文献

[1] 柳澤秀明，森秀樹，上原稔，“Design Automation Shell”, 機能集積情報システム研究会発表論文集 II, FIIS-99-57, pp.211-217, (1999.6)

[2] 上原稔, 柳澤秀明, 森秀樹, “ハードウェア設計ツール SpecC の自動生成”, 機能集積情報システム研究会発表論文集 III, FIIS-01-90, pp.273-279, (2001.6)

[3] デイビッド・パターソン, ジョン・ヘネシー著, 富田眞治, 村上和彰, 新實治男 訳,”コンピュータ・アーキテクチャ”, 日系BP社, (1992)