卒業論文

ハードウェア記述言語による教育用
マイクロプロセッサの設計(Ⅰ)

氏名　　：　中村　央志
学籍番号　：　2210990161-3
指導教員　：　山崎　勝弘　教授

提出日　　：　2003 年 2 月 21 日

立命館大学　理工学部　情報学科

内容梗概

本研究では、Verilog HDL を用いて、マイクロプロセッサの最も基本的なアーキテクチャを理解する目的で、マルチサイクルプロセッサの設計を行った。マイクロプロセッサの設計においては、慶応大学と東京工業大学で共同に開発した実験教育用の 16 ビット幅の命令セットを持つ PICO16 があるが、これは SFL という HDL（ハードウェア記述言語）を使用している。本研究ではこのアーキテクチャを理解した上で、Verilog HDL で実際に設計を行った。

また、ハードウェア設計においてトップダウン設計が主流となっており、その設計方法である仕様作成、動作レベル HDL 作成、動作レベルシミュレーション、RTL レベル HDL 作成、論理合成、最適化、ゲートレベルシミュレーションまで行い、プロセッサの動作検証を行った。テストプログラムでは C 言語記述のシェルソートを PICO16 での記述と比較することで、ハードウェアとソフトウェアの相互の理解を目的とする。

目次

図目次

表目次

1　はじめに

世界で最初のCPUはIntel社が1971年11月に発表したマイクロプロセッサ「4004」である。その動作周波数は108KHz、トランジスタ数は2300個であった。これが30年後の現在では、Intel社より「PentiumⅣ」が発表され、動作周波数は3.06GHz、トランジスタ数は5500万個、20段のパイプラインなどが実装され、CPUの高速化が実現されている。[8]

1970年代に初めてIC（集積回路）が誕生してから、一般の人々に普及したのは電卓や時計、ゲーム機といったパーソナル機器であった。1980年代に入ると、DRAMが普及し、半導体の設計規模が増大した。また、従来は単体の機能だけを考慮して設計していたものが、家庭用電化製品や携帯電話に組み込まれているシステム全体を1Chip化するといったSoC（System On Chip）設計などに変わってきており、設計規模が非常に大きくなってきている。

LSI設計フローは、従来はセル部から設計を始め、最終的に目的の製品を作成するボトムアップ設計という手法が用いられていたが、現在では機能の検証からアプローチするトップダウン設計に移行している。ボトムアップ設計では、設計が進むにつれ、修正が困難になり、設計の最終段階まで製品の完全な動作が検証できないのに対し、トップダウン設計では、ハードウェア記述言語を用い、論理シミュレーションによって設計の初期で仕様の致命的な欠陥を発見でき、タイミングなどの考慮もする必要がなく、設計時間の短縮化など、設計が容易にできるようになった。

ハードウェア記述言語はVHDLやVerilogHDLなどがあるが、本研究に用いるVerilogHDLに関しては、1990年代にケイデンスが開発し、1995年12月にIEEE1364に認定され、業界に浸透していくこととなった。また、C言語に似ていることや、動作レベル、RTLレベル、ゲートレベルのすべての記述が同一の言語で実現できるなど、非常に扱いやすい言語であると言える。また、FPGA（field programmable gate array）の登場により、論理機能を安価に実際にチップに焼き付けることができるようになったため、開発コストの削減と、開発期間の短縮に繋がった。[1]

今回設計したPICO16は慶応義塾大学と東京工科大学で共同に開発されたマイクロプロセッサであり、16ビットRISCプロセッサである。命令長は16ビットで、I形式、R形式、J形式の３つの形式と、27個の命令を持つ。またPICO自体はSFL（Structured Functional Language）というハードウェア記述言語を用い、NTT研究所が開発したハードウェア開発環境であるPARTHENON（Parallel Architecture Refiner THEorized by Ntt Original Concept）を用いて設計されている。[3]

本研究では、PICO16をVerolog HDLを用いて設計することにより、マイクロプロセッサの基本的なアーキテクチャを理解すし、HDLによるトップダウン設計の手順を習得することを目的とする。ローム記念館の立命館大学VLSIセンターで用いられているCADツールで設計した。VLSIセンターで実際に使用したツールは、NC-Verilog、

Verilog-XL（機能設計）、Design Compiler（論理合成）である。トップダウン設計に基づいて、仕様作成、動作レベル HDL 作成、動作レベルシミュレーション、RTL レベル HDL 作成、論理合成、最適化、ゲートレベルシミュレーションまで行った。

第 2 章ではハードウェア記述言語によるマイクロプロセッサの設計について述べる。第 3 章では PICO16 マルチサイクルプロセッサのアーキテクチャについて述べる。第４章では、ローム記念館の論理設計ツールである、NC-Verilog を用いた論理シミュレーション、Design Compiler を用いた論理合成と最適化について述べる。第 5 章では C 言語記述のシェルソートを PICO 16 で記述した上で、そのテストパターンのシミュレーションによる検証、ハードウェア規模、性能などの検証と考察について述べる。

２　ハードウェア記述言語によるマイクロプロセッサの設計

２．１　ハードウェア記述言語

ハードウェア記述言語（HDL：Hardware Description Language）とはLSIやシステムなどの設計データを記述するための言語であり、ハードウェアの動作を論理ゲートレベルより抽象度の高いレベルで記述できる言語である。それにより設計時間が短縮できることや、設計の変更や再利用が容易であり、設計者がより完成度の高いシステムを設計することができる。またハードウェア記述言語はC言語に似ているため、言語の習得と言う意味でも非常に有効である。

代表的なHDLにはVHDL、VerilogHDL 、SFL（Structured Functional Language）などがあるが、ここではVHDLとVerilog HDLの2種類についてその特徴を示す。

✍✍ VHDL

－米国防総省にて開発された言語

－通信用LSI、システム設計の開発に使用

－文法的に厳格な言語

✍✍ VerilogHDL

－パブリックドメインとしてケイデンスが公開

－1995年にIEEE1364として標準化

－LSI設計の記述性を重視した言語

また、図1の全加算器における VerilogHDL での記述を図２に示す。

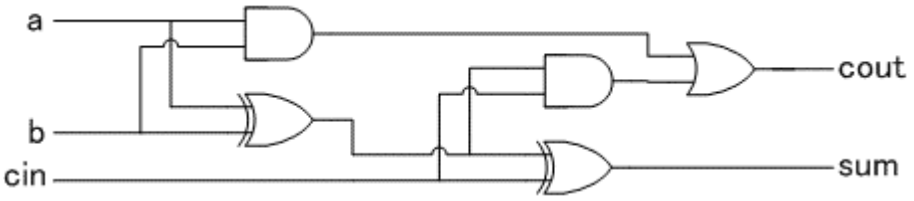

図 1：全加算器の論理回路図

```
module fadder (a,b,cin,sum,cout);
input a,b,cin;
output sum,cout;

assign cout = (a&b)|(b&cin)|(a&cin);
assign sum  = (a&~b&~cin)|(~a&b&~cin)|(~a&~b&cin)|(a&b&cin);

endmodule
```

図 2：全加算器の VerilogHDL による記述

VerilogHDLでは1行目でモジュール名の宣言と外部ポートの宣言を行う。1行目2行目のinput、outputで入出力信号の定義をする。そして、assign文で実際の出力結果と一致するようにプログラムを作成する。

２．２　トップダウン設計

ハードウェア記述言語の普及により従来のボトムアップ設計からトップダウン設計に移行した。トップダウン設計とは図３のように抽象度の高い仕様設計から機能記述、回路図を経て、実際の回路の設計に向かって徐々具体度を高めていく設計手法である。まず最初に仕様を設計し、動作レベルでの HDL を記述し、テストパターンによって動作レベルシミュレーションを行い、仕様通りの機能が得られるよう修正する。次に動作レベル HDL を RTL（Register Transfar Level）HDL に変換し、それを論理合成、最適化した上で、ネットリストを出力する。配置配線ツールなどによってゲートレベルシミュレーションを行い、FPGA（Field Programmable Gate Array）などの Chip に焼き付けて動作検証を行う。

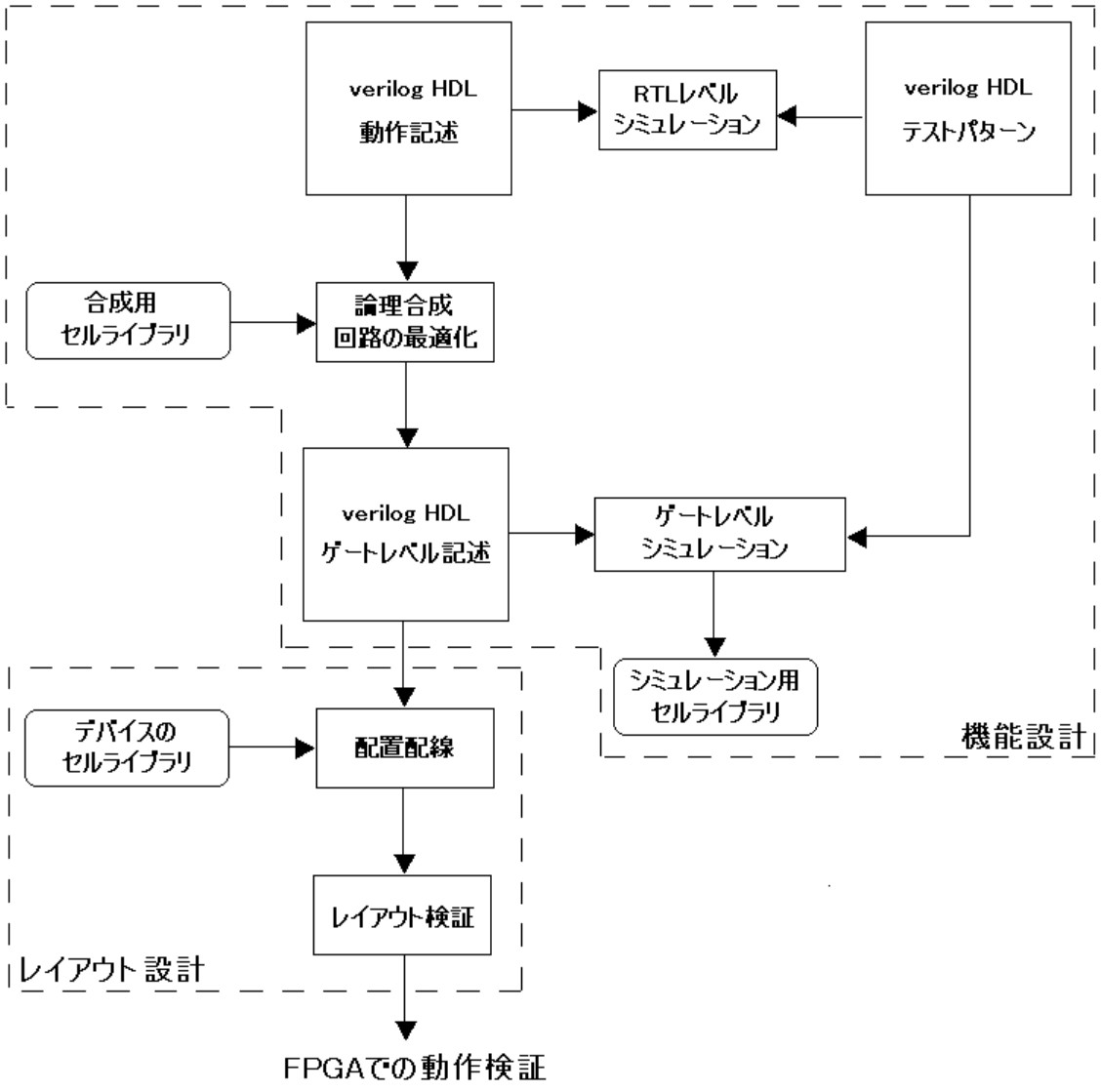

図 3：トップダウン設計のフローチャート

２．３　ＦＰＧＡ

PLD（Programmable logic device）と呼ばれるプログラム可能な LSI は、SPLD（Simple PLD）、CPLD（Complex PLD）、FPGA（Field Programmable Gate Array）の 3 種類に分類される。SPLD、CPLD は記録方式として EEEPROM 構造をしており、またゲート数は少ない。FPGA は SRAM 構造になっており、ゲート数は数百万ゲートで大規模である。FPGA はプログラム書き換え可能なゲートアレイで、トップダウン設計により設計された論理機能を焼き付けることで、その LSI が利用できる。従来のボトムアップ設計では大規模な LSI の作成に多大なコストがかかっていたが、この FPGA によって動作検証ができるようになったことでこの問題は解決した。

２．４　プロセッサアーキテクチャの分類

1971 年 11 月に Intel 社が発表したマイクロプロセッサ「4004」が世界で最初のマイクロプロセッサである。現在では Intel 社より「PentiumⅣ」が発表されている。様々なプロセッサが開発される中で命令実行方式にも違いがあり、そのアーキテクチャの分類と、その動作について述べる。

（１） 単一サイクル方式

単一サイクル方式では、一つの命令を１クロックサイクルで完了する。最も処理の長い命令に依存するため、高速化ができない。現代のマイクロプロセッサに採用されていない。また CPI は１となる。

（２）マルチサイクル方式

マルチサイクル方式では、ひとつの命令を命令フェッチ、レジスタフェッチ、EX、メモリアクセス、などの複数のステップに分け、それぞれのステップが１クロックサイクルを占める。また、単一サイクルに比べ CPI は増大するが、１クロックサイクルを短くすることが可能で、クロック周波数をあげることができる。本研究で設計した PICO16 もマルチサイクル方式である。

（３） パイプライン方式

パイプライン方式は命令の各実行ステップをステージに分割し、連続した各命令を少しずつずらして同時並列的に実行する方式である。命令のスループットが増大し、命令の全体のクロックサイクルが大幅に減少する。だが、パイプラインステージを増やせば増やすほど高速化が期待できるわけではなく、実際は次のクロックサイクルで命令が実行できないというパイプラインハザードが起こる。パイプラインハザードにはデータハザード、制御ハザードなどがあるが、これを考慮するため、制御も複雑になる。現在の Intel 社の「PentiumⅣ」には２０段ものパイプラインが実装されスーパーパイプラインとも呼ばれる。

（4） スーパースカラ方式

スーパースカラ方式とは、フェッチやデコード、実行するためのユニットをそれぞれ複数用意して、1 クロックサイクルで複数の命令を同時に実行する方式である。また命令実行時にプロセッサ内部で動的に判断され、空いている実行ユニットがあれば自動的にそれを使うようにスケジューリングが行われる。

（5） VLIW 方式

VLIW（Very Long Instruction Word）方式とは、ひとつの命令語中に、複数の命令を格納しておき、それらをすべて同時に実行する方式である。常に決まった数の命令が同時にパイプラインに投入され、同時に実行される。命令間には依存関係がないように事前にコンパイラによって最適化する。VLIW の性能を最大限に引き出すためにはコンパイラによる最適化技術が欠かせない。

3　マルチサイクルプロセッサ PICO16 のアーキテクチャ

3．1　PICO16 の命令セット

PICO16 は、慶応義塾大学と東京工科大学で共同に開発した実験教育用のマイクロプロセッサで、利用可能な FPGA のサイズに応じてさまざまなサイズの命令セットを構築することができる。PICO16 の命令セットは以下のような特徴をもつ。[3]

- Load/Store マシン（register-register マシン）である。
  計算はレジスタ間でのみ許される。計算を行うためにはメモリからレジスタにデータをロードする必要がある。
- 16 ビットの単一命令長を持つ
  命令長を固定することで、命令の作り方に苦しい部分がでてくるが、制御が圧倒的に簡単になり、パイプライン化も容易になる。
- 単純な命令セット
  単純な命令セットを持つことで、簡単な実装ができる。

PICO にはメモリアクセス命令を含むレジスタ間演算命令、immediate（即値）命令、分岐命令の３種類の命令があり、図４に PICO の命令形式を示す。

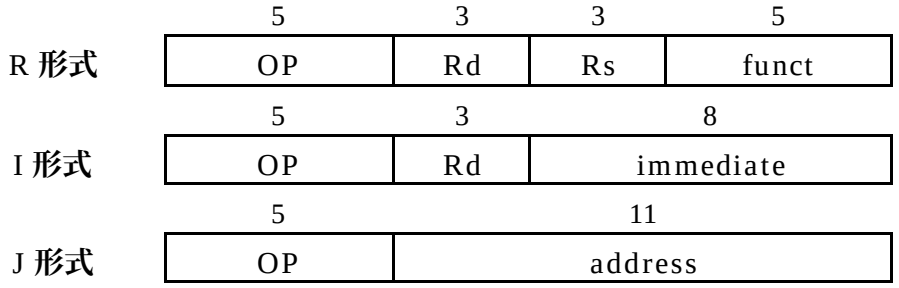

図 4：PICO の命令形式

各フィールドの意味を以下に示す

OP（opcode）	： 命令形式の判定
Rd（destination Register）	： 格納先レジスタ
Rs（source Register）	： 参照先レジスタ
Function（functional code）	： 論理演算用の補助コード
Address	： Rd に対する即値

PICO16 の命令セットはこれらの３つの命令形式をもち、I 形式（即値命令）は immidiate 命令で、条件分岐命令などに使用する 8 ビットのフィールドデータを格納する。R 形式（レジスター－レジスタ命令）は算術論理演算に利用する。J 命令（分岐命令）

はジャンプ命令、サブルーチンコールなどに使用する。命令コード以外はすべてデータに使って、できる限り長いデータを格納する。また、PICO16 の持つ命令数は 27 個である。その命令を表 1 に示す。

表 1：PICO16 の命令セット

命令形式	命令コード		実行内容
R 形式	00000ddd_sss00000	NOP	有効な操作は行わない
	00000ddd_sss00001	MV	s の値を d に格納
	00000ddd_sss00010	AND	d と s の AND を d に格納
	00000ddd_sss00011	OR	d と s の OR を d に格納
	00000ddd_sss00100	XOR	d と s の XOR を d に格納
	00000ddd_sss00101	NOT	s の NOT を d に格納
	00000ddd_sss00110	ADD	d＋s を d に格納
	00000ddd_sss00111	SUB	d－s を d に格納
	00000ddd_sss01000	LD	s で示す番地の値を d に格納
	00000ddd_sss01001	ST	d で示す番地の値に s を格納
	00000ddd_sss01100	SL	s の 1bit 左シフトを d に格納
	00000ddd_sss01101	SR	s の 1 bit 右シフトを d に格納
I 形式	00110ddd_xxxxxxxx	ADDI	d＋X を d に格納
	00111ddd_xxxxxxxx	SUBI	d－X を d に格納
	00010ddd_xxxxxxxx	ANDI	d と X の AND を d に格納
	00011ddd_xxxxxxxx	ORI	d と X の OR を d に格納
	00100ddd_xxxxxxxx	XORI	d と X の XOR を d に格納
	11100ddd_xxxxxxxx	LDLI	X を d の下位 8bit に格納
	11101ddd_xxxxxxxx	LDHI	X を d の上位 8bit に格納
	01000ddd_uuuuuuuu	JALR	戻り番地を格納して d へ分岐
	01001ddd_xxxxxxxx	BNEZ	d≠0 ならば相対分岐
	01010ddd_xxxxxxxx	BEQZ	d=0 ならば相対分岐
	01011ddd_xxxxxxxx	BMI	d<0 ならば相対分岐
	01100ddd_xxxxxxxx	BPL	d？0 ならば相対分岐
J 形式	01101_xxxxxxxxxxx	JAL	戻り番地を格納して相対分岐
	01110ddd_uuuuuuuu	JR	d の内容に相対分岐
	01111_xxxxxxxxxxx	JMP	無条件相対分岐

３．２　PICO のアーキテクチャ

本研究で作成した PICO16 のデータパスを図５に示す。

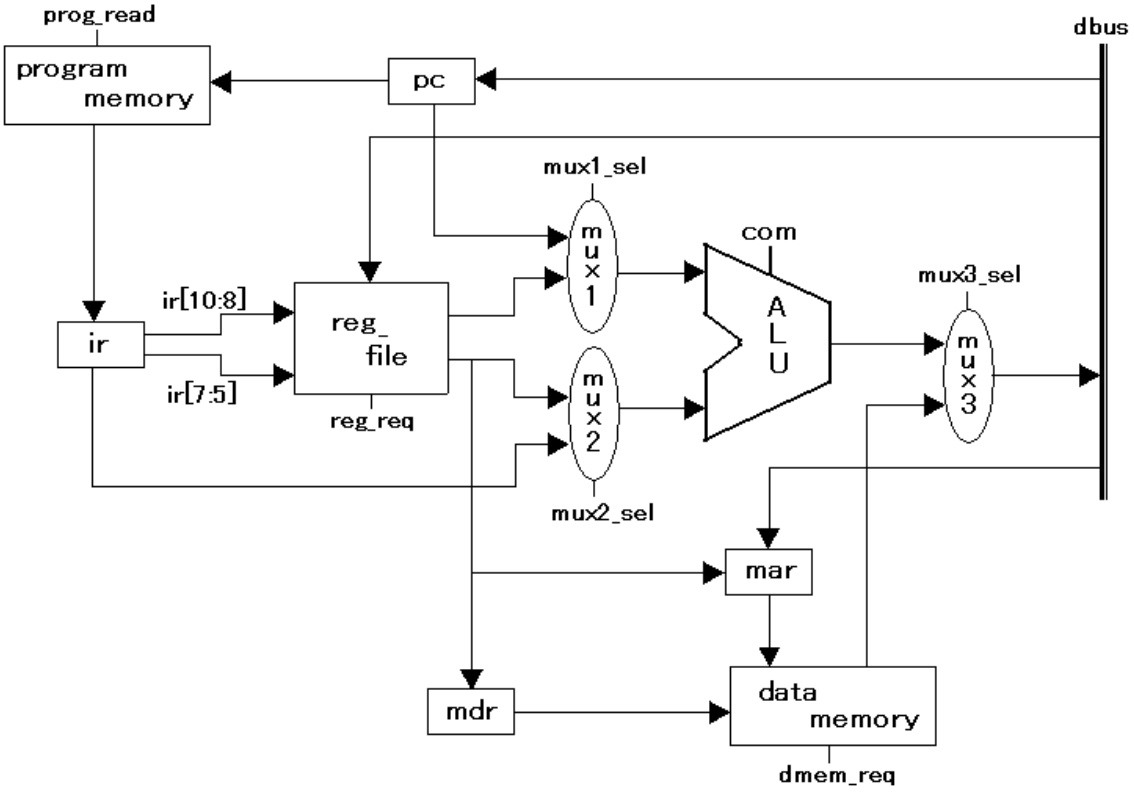

図 5：PICO16 のデータパス

PICO16 は表２に示す機能ユニットから構成されている。

表 2：PICO16 の機能ユニット

プログラムカウンタ(pc)	次に読み込む命令を格納する
命令レジスタ(ir)	プログラムメモリから命令を格納する
レジスタ(reg_file)	命令をデコードして格納する
メモリアドレスレジスタ(mar)	アクセスするメモリのアドレスを格納
メモリデータレジスタ(mdr)	メモリに書き込むデータを格納
データメモリ(data memory)	データを格納
プログラムメモリ(program memory)	命令を格納
ALU(alu)	算術演算を行う
マルチプレクサ(mux)	入力するデータを選択する

３．３　命令の実行と制御

PICO16 はマルチサイクルプロセッサであり、命令フェッチと命令実行を繰り返しながら動作する。メモリアクセス命令を含むレジスタ間演算命令、immediate（即値）命令、分岐命令の３種類の命令があり、命令フェッチ、レジスタフェッチ、EX、EX2 の 4 つのステージに分かれ状態遷移を繰り返す。メモリアクセス命令と JAL 命令のみが EX2 を含めた 4 つのステージを実行し、その他の命令は EX までの 3 つのステージで実行する。以下に４つの状態遷移を示す。

✍ IF：命令フェッチ

プログラムアドレスである PC で指定されるプログラムメモリの値を ir に格納する。

✍ RF：レジスタフェッチ

Ir のレジスタフィールド取ってきた命令中のオペランドフィールド(10〜8bit、7〜5bit)で指定されるアドレスをレジスタから読み出し、セレクタ(mux1、mux2)に値を代入する。immidiate（即値）命令では、pc_add で下位 8 ビットの値を 16 ビット（上位 8 ビットは 0）に拡張する。また 7〜5bit は immidiate の数値の一部であるため、このアドレスの値は利用されないが、読み出してもは問題はないため読み出しておく。また、このステージで次の命令を読み出すための PC の値を ALU で１加算しておく。ST 命令では読み出したレジスタの値をメモリデータレジスタ（mdr）に格納しておく。

✍ EX：実行１

メモリアクセス（LD、ST）以外の命令は EX ステージで完了する。レジスタ間演算命令では、セレクタから ALU に値を代入し、また ir の 4〜0bit は ALU のコマンド(com)となっているため、そのコマンドを用いて演算を実行し dbus を経由してからレジスタファイルに書き込む。Immidiate（即値）命令では、ir の 15〜11bit が ALU のコマンドとなっており、pc_add で符号拡張した値を演算する。LD 命令ではアクセスするメモリのアドレスをメモリアドレスレジスタ（mar）に格納する。分岐命令では、判定する条件の結果によって分岐先アドレスを pc に格納する。

✍ EX2：実行２（メモリアクセス）

LD 命令では、mar に格納したアドレスを読み出し、レジスタに格納することで命令を完了する。ST 命令では、レジスタファイルから読み出したメモリのアドレスに mdr の値を格納する。

また、各命令の実行ステップを表３に示す。

表 3：マルチサイクル方式ＰＩＣO16 の実行ステップ

	R 形式、I 形式	LD 命令	ST 命令	J 形式	JAL 命令
IF	命令フェッチ				
RF	レジスタのフェッチ、プログラムカウンタの処理				
EX	演算処理	メモリアクセス	メモリアドレスを読み込む	分岐処理	分岐元アドレスを格納
EX2		レジスタに格納	メモリに格納		分岐処理

✍✍ 演算処理：演算を実行して、その結果をレジスタに格納

✍✍ 分岐処理：分岐の条件による判断の結果を pc に格納

４　VerilogHDL によるマルチサイクルプロセッサ PICO16 の設計

４．１　CAD ツールと設計手順

本研究ではローム記念館の VLSI センターで使用されている CAD ツールを用いて設計を行った。使用したツールは以下の２つである。

✍ NC-verilog、Verilog-XL

ケイデンス社の verilog シミュレータであり、NC-verilog はコンパイラ型の超高速イベント・ドリブン・シミュレータであり、Verilog-XL はインタプリタ型の高速イベント・ドリブン・シミュレータである。Verilog-XL は業界標準シミュレータとして確立されており、多くの ASIC ベンダからサインオフ・シミュレータとして認定されている。また NC-verilog と Verilog-XL は互換性があるため、本研究では NC-verilog を用いて、RTL レベルシミュレーションとゲートレベルシミュレーションまでを行った。

✍ DesignCompiler

Synopsys 社の論理合成、最適化ツールであり、verilog による回路記述と NC-verilog による機能シミュレーションが完了したら，Design Compiler でターゲットの ASIC ライブラリに基づいて論理合成を行い、最適化済みゲートレベル記述を作成する。

本研究で行ったゲートレベルシミュレーションまでのトップダウン設計におけるツールの領域を図 6 に示す。

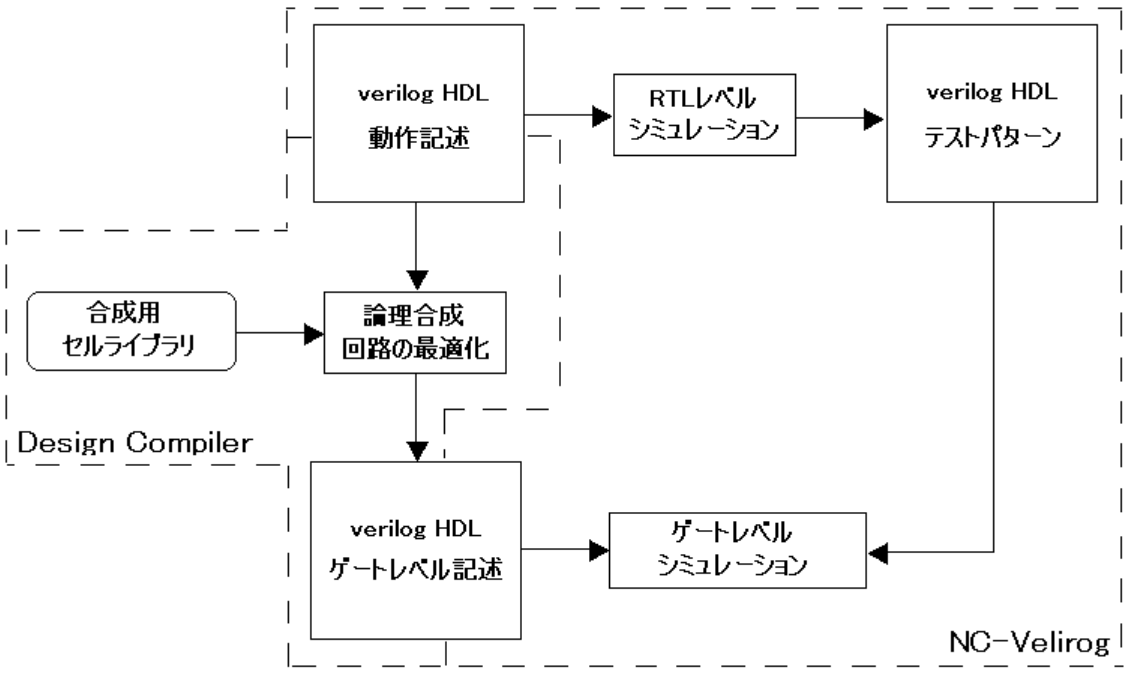

図 6：トップダウン設計におけるツールの領域

また、このツールはローム記念館の端末でのみ使用可能である。VLSI センターの端末の OS は solaris を用いている。NC-verilog による各レベルシミュレーションの手順は端末エミュレータから、

% ncverilog +access+r test_pattern.v logic_ description.v –s +gui

とコマンドを入力することで図７の GUI が立ち上がる。

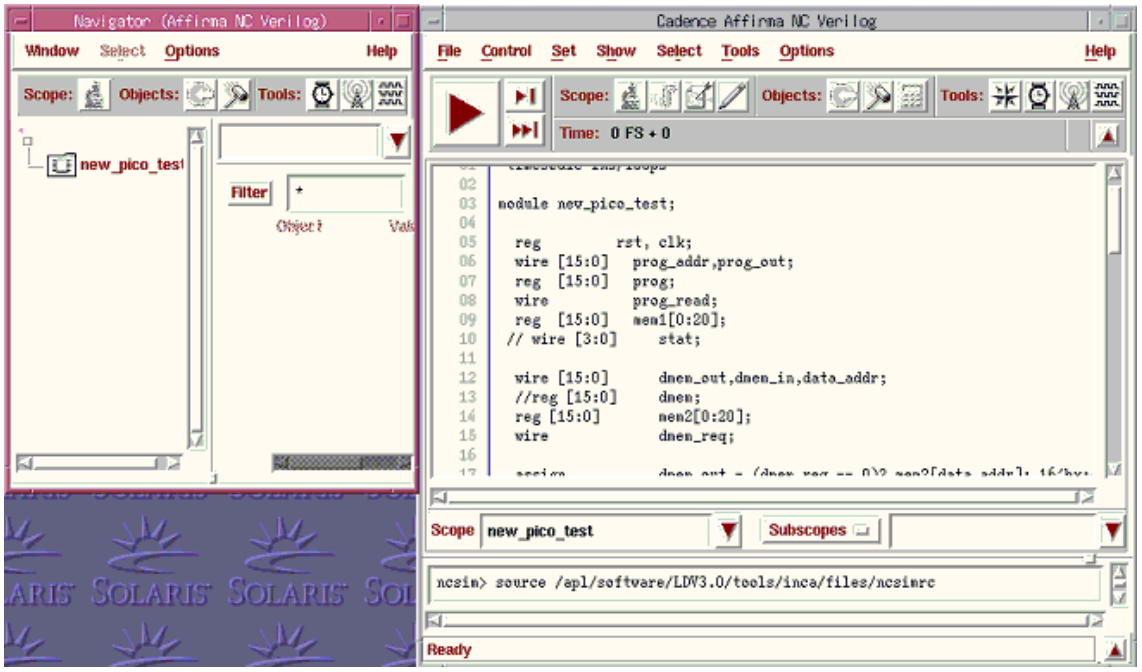

図 7：NC-Verilog

また、これから波形を表示してシミュレーション結果を確認するには、図７の左側の navigator ウィンドウで表示したい信号を選択してから右上の  ボタンを押すことで、下図のような SignalscanWaves というグラフィック波形表示ツールが起動する。図８に表示されている波形は図２の全加算器の verilogHDL 記述より用いた。

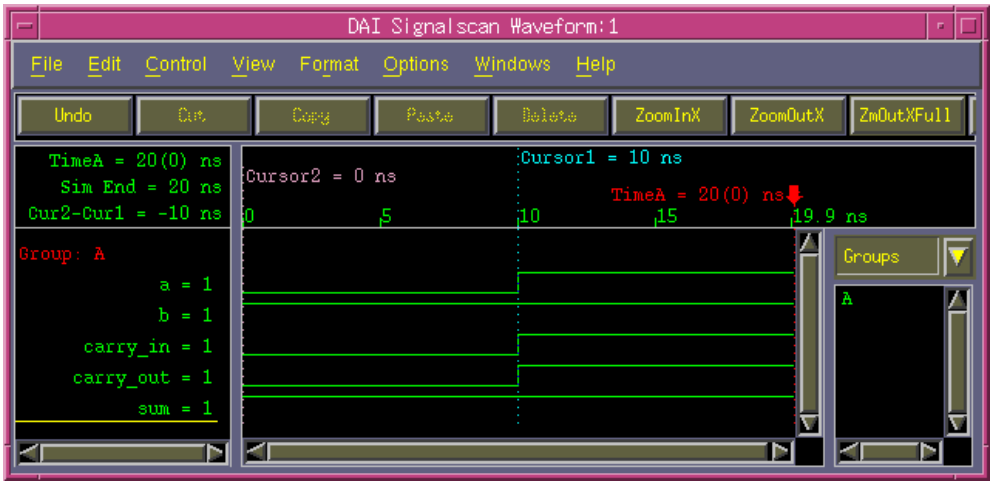

図 8：Signalscan Waves

NC-verilog と Signal Waves を用いて動作レベルの記述に問題がないことがわかれば、次に Design Compiler を用いて論理合成と最適化を行う。Design Compiler の起動は、

% design_analyser

とコマンドを入力することで、Design Compiler の GUI 版の design_analyser が起動する。ここでは「File」→「Read」から全加算器の VerilogHDL 記述ファイルを読み込んだ。そのときに、ターゲットの ASIC ライブラリに基づいて回路の合成が行われる。

最初にモジュールが２つ以上の場合は回路の平坦化、またはサブ回路の独自化を行う。回路の平坦化は「Setup」→「Command Window」から開いたウィンドウに ungroup –all –flatten と入力して実行する。サブ回路の独自化は、トップモジュールを選択して「Edit」→「Uniquify」→「Hierarchy」を実行する。

次に入出力バッファセルの挿入を行う。まず入力信号の設定は、入力信号を選択して「Attribute」→「Optimization Directives」→「Input port」から図９のような画面が開く。Port is Pad にチェックを入れ Port Pad Attributes...を押すとウィンドウが開く。そこでは Exact Pad to Use にチェックをいれ、Pad Name を VZIBUF としてセルの挿入を実行する。同様に出力信号の場合は、出力信号を選択して「Attribute」→「Optimization Directives」→「Output port」から図９と同様にして、Pad Name を VZOBUF とする。次に、「Edit」→「Insert Pads」で開いたウィンドウをそのまま OK としてこのセルの挿入が完了する。そして「Tools」→「Design Optimization」で回路の最適化を行う。

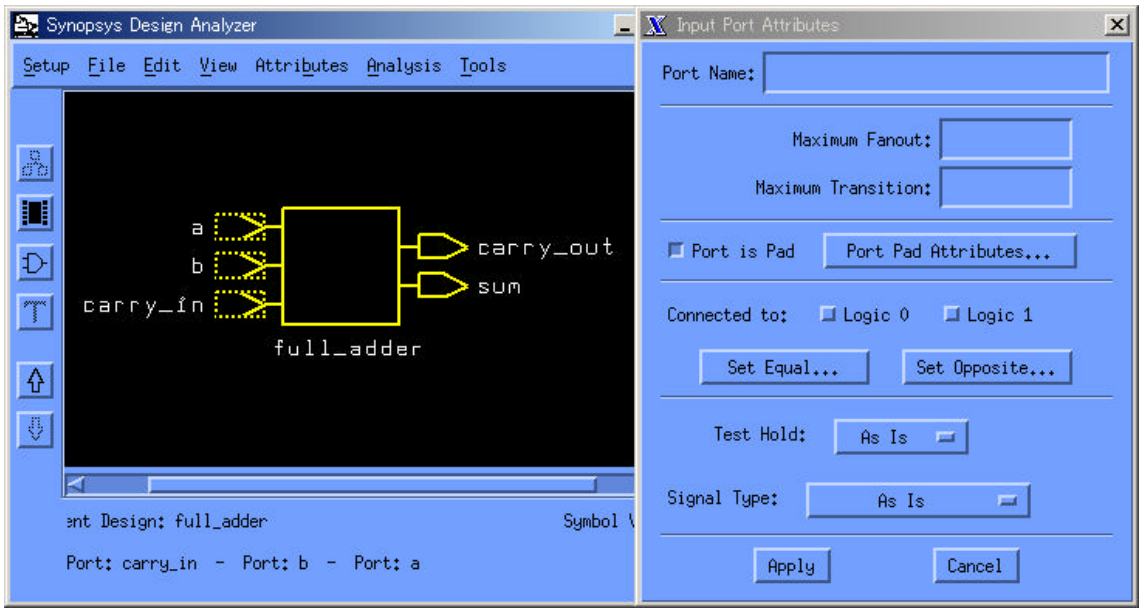

図 9：入力信号の設定

次に図９の入出力ポートを全て選択し、「Attributes」→「Optimization Constraints」→「Timing Constraints」を実行して、タイミングの制御の設定を行う。また、Clock があれば、「Attributes」→「Clocks」よりクロック周期の設定をする。今回の全加算器では Clock がないのでこの設定はしない。これらの設定が完了した上で「Tools」→「Design Optimization」から回路の最適化をもう一度実行することで、design_analyzer による回路の最適化が完了する。図 10 にファイルを読み込んだときの全加算器の回路図（左）と、手順を全て完了した後の回路図（右）を示す。

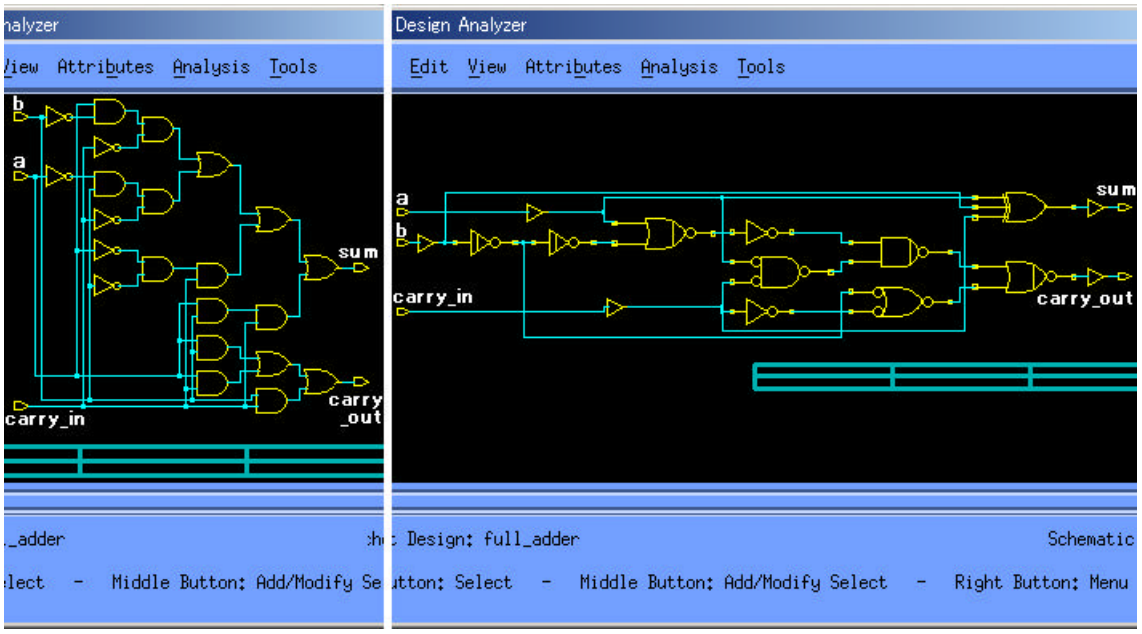

図 10：回路の最適化

４．２　モジュール構成

図５のデータパスにより設計した PICO16 にはモジュールが、reg_fire、mux1、mux2、ALU、mux3 の５つがある。以下に各モジュールの入出力と動作を説明する。

✍ reg_file（レジスタファイル）

命令を実行するためのレジスタのアドレスを受け取り各種演算命令の結果を出力する。また、その結果を再び格納する。このプロセッサにおいて最も必要なモジュールのひとつである。その入出力信号を表４に示す。

表 4：reg_fire の入出力信号

入力信号	clk(１ビット)	クロック信号、全ての動作が clk に同期する。
	rst(１ビット)	clk の立ち上がりに１ならばレジスタファイルの中身を初期化する。
	reg_req(１ビット)	レジスタから読み込むときは１、書き込むときは０を入力する。
	reg_addr1(３ビット)	演算などに用いるレジスタファイルのアドレスである ir の 10～8bit を入力する。
	reg_addr2(３ビット)	レジスタファイルのアドレスである ir の 7～5bit を入力する。
	reg_in(16 ビット)	演算の結果や、分岐元アドレスの保存時にレジスタに入力する。
出力信号	reg_dout1(16 ビット)	読み込んだ reg_addr1 のアドレスに入っているレジスタのデータを出力する。
	reg_dout2(16 ビット)	読み込んだ reg_addr2 のアドレスに入っているレジスタのデータを出力する。

✍ mux1（データセレクタ１）

mux1 では、reg_fire から受け取った reg_dout1 と pc（プログラムカウンタ）が入力され、どちらかを選択して出力する。ステージの初期値が与えられたときはクロックの立ち下がりで pc を選択する信号が入力される。入出力信号を表５に示す。

表 5：mux1 の入出力信号

入力信号	mux1_sel(1 ビット)	１ならば mux1_in1 の値を出力し、０ならば mux1_in2 の値を出力する。
	mux1_in1(16 ビット)	レジスタファイルから出力された reg_dout1 のデータを入力する。
	mux1_in2(16 ビット)	pc の値を入力する。
出力信号	mux1_out(16 ビット)	mux1_sel の信号から値を出力する。

✍ mux2(データセレクタ２)

mux2 では reg_dout2 と pc を加算する値または ir から符号拡張された値が入力され、どちらかを選択して出力する。mux1 と同様にステージの初期値が与えられたときはクロックの立ち下がりで pc を加算する値を選択する信号が入力される。

表 6：mux2 の入出力信号

入力信号	mux2_sel(1 ビット)	1 ならば mux2_in1 の値を出力し、0 ならば mux2_in2 の値を出力する。
	mux2_in1(16 ビット)	レジスタファイルから出力された reg_dout2 のデータを入力する。
	mux2_in2(16 ビット)	pc を加算する値や、immidiate 命令で加算する数値を拡張して入力する。
出力信号	mux2_out(16 ビット)	mux2_sel の信号から値を出力する。

特に mux2_out の出力が mux2_in2 の時は、表 7 のように命令によって入力される値が変わる。

表 7：各命令による mux2_in の入力

ir に入力された命令	mux2_in2(16 ビット)に入力される値
LDLI、ADDI、SUBI	ir[7]を上位 8 ビットに拡張、下位 8 ビットに ir[7:0]
LDHI	上位 8 ビットに ir[7:0]、下位 8 ビットは 0
ANDI、ORI、XORI	上位 8 ビットは 0、下位 8 ビットに ir[7:0]
JAL、JMP	上位 5 ビットに ir[10]を拡張、下位 10 ビットに ir[10:0]
BPL、BNEZ、BEQZ、BMI が条件を満たす場合	ir[7]を上位 8 ビットに拡張、下位 8 ビットに ir[7:0]
上の命令が条件を満たさない場合とステージが IF のとき	16 進数「0001」

✍ ALU（論理演算ユニット）

mux1 と mux2 のそれぞれからの出力を受け取り、ir から受け取った opcode によって決められた演算を実行する。RF ステージでは pc のカウントに使われ、EX ステージでは ir から読み取った演算を実行する。

表 8：ALU の入出力信号

入力信号	com(4 ビット)	opcode が入力され、その code によって行う演算を決定する。
	reg1(16 ビット)	mux1 の出力から入力される
	reg2(16 ビット)	mux2 の出力から入力される
出力信号	alu_out(16 ビット)	演算結果を出力する。

✍ mux3（データセレクタ 3）

alu からの出力を受け取り、また LD 命令でロードする値を受け取る。

表 9：mux3 の入出力信号

入力信号	mux3_sel(1 ビット)	1ならば mux3_in1 の値を出力し、0ならば mux3_in2 の値を出力する。LD 命令の時のみ 0となる。
	mux3_in1(16 ビット)	ALU から出力された alu_out が入力される。
	mux3_in2(16 ビット)	LD 命令の時のみ、データメモリの値が入力される。
出力信号	mux3_out(16 ビット)	mux3_sel の信号から値を出力する。

４．３ データ構成

図５のデータパスにある中で、モジュールではないユニットの動作を以下に示す。

✍ pc（プログラムカウンタ）

プログラムカウンタであり、次に読み込む命令のアドレスをプログラムメモリに送る。また、pc の加算の時には mux1 に出力される。

✍ program memory（プログラムメモリ）

pc からプログラムアドレスを受け取り、prog_read の立ち上がりに ir へデータを出力する。

✍ dbus

mux3 から入力され、値を保持するためのバス、pc を加算した値が入力され、再び pc に出力する。また ALU の演算結果も dbus に保持される。

✍ ir

プログラムメモリから受け取ったデータを 10～8bit と、 7～5bit を reg_file へ出力する。またプログラムカウンタで加算する値を mux2 に出力する。

✍ mar（メモリアドレスレジスタ）

LD 命令時に reg_file の reg_dout2 を受け取る。データメモリからロードするアドレスが入力される。また ST 命令時には dbus から格納するデータメモリのアドレスが入力される。

✍✍ mdr（メモリデータレジスタ）

ST 命令時に reg_file の reg_dout2 を受け取る。データメモリに格納するデータが入力される。

✍✍ data memory

dmem_req が１の時は、ST 命令時の mdr と mar から受け取ったデータを書き込む。
dmem_req が０の時は、LD 命令時の mar から受け取ったアドレスを dmem_out として、mux3 に出力する。

5　設計した PICO16 の検証

5．1　シェルソートによるテストプログラムシミュレーション結果

本研究で作成したテストプログラムは図 11 の C 言語記述によるシェルソートを用いた。これを PICO16 の命令セットで記述をして動作確認を行った。データ数は 5 個、20 個、50 個の場合で行った。

```
1: void shell_sort(in a[], int n)
2: {
3:   int h, i, j, t;
4:
5:   for(h = 1; h < n/9; h * 3 + 1)
6:   ;
7:
8:   for( ; h > 0; h = h/3){
9:       for(i = h; i < n; i++ ){
10:          j = i;
11:          while (j >= h && a[j-h] > a[j]){
12:              t = a[j];
13:              a[j] = a[j-h];
14:              a[j-h] = t;
15:              j = j ? h;
16:             }
17:         }
18:      }
19:  }
```

図 11：シェルソートの C 言語記述

またその C 言語の 5 行を実際に PICO16 の命令セットに変換した命令を図 12 に示す。C 言語の for 文の 1 行が PICO16 の命令だと 18 行になっている。

```
//5:    for(h = 1; h < n/9; h * 3 + 1)

1:  11100000_00010100  //LDLI r1 #20    n の値 20 を r1 に格納
2:  11100010_00000001  //LDLI r2 #1     h の初期値 1 を r2 に格納
3:  11100101_00000000  //LDLI r5 #0     r5 に 0 を格納
4:  00111001_00001001  //SUBI r1 #9     r1 - 9 → r1
5:  00110101_00000001  //ADDI r5 #1     r5 + 1 → r5
6:  01100001_11111110  //BPL  r1 #-2    r1>0 ならば命令 4 に分岐
7:  00111101_00000001  //SUBI r5 #1     r5 - 1 → r5
8:  01010101_00001010  //BEQZ r5 #10    r5=0 ならば命令 18 に分岐
9:  11100100_00000010  //LDLI r4 #2     r4 に 2 を格納
10: 00000011_01000001  //MV   r3 r2     r2 を r3 にコピー
11: 00000010_01100110  //ADD  r2 r3     r2 + r3 → r2
12: 00111100_00000001  //SUBI r4 #1     r4 - 1 → r4
13: 01001100_11111110  //BNEZ r4 #-2    r4≠0 ならば命令 11 に分岐
14: 00110010_00000001  //ADDI r2 #1     r2 + 1 → r2
15: 00000110_10100001  //MV   r6 r5     r5 を r6 にコピー
16: 00000110_01000111  //SUB  r6 r2     r6 - r2 → r6
17: 01100110_11110111  //BPL  r6 #-8    r6>0 ならば命令 9 に分岐
18: 00000111_01000001  //MV   r7 r2     r2 を r7 にコピー
```

図 12：C 言語の for 文における PICO16 命令記述

C 言語と PICO16 との命令数の比較を表 10 に示す。

表 10：シェルソートの命令数の比較（データ 20 個）

	C 言語	PICO16
静的命令数	19	55
実行命令数	422	1436

また、論理シミュレーションレベルでのクロックサイクルと実行命令数を表 11 に示す。ソート数が 5 個の時に CPI が下がっているが、これはデータの入れ替えが少なかったことで、クロックサイクルが多い LD、ST 命令が少なかったからである。シェルソートするデータの並び方によって CPI 数が変化する。

表 11：シェルソートの実行結果

データ数	5	20	50
実行命令数	226	1436	5164
クロックサイクル数	716	4574	16460
CPI	3.168	3.185	3.187

５．２ プロセッサの評価

４．１章で述べた design_analyzer を用いて最適化を行い、表 12 のような結果が得られた。入出力のポート数は変化しないが、特に配線の数が大幅に最適化されていることがわかる。

表 12：PICO16 の最適化

	最適化前	最適化後
ポート数	84	84
配線数	1486	796
セル（ゲート）数	763	671

また、最適化が完了した後のゲート遅延時間、クロック周波数を表 13 に示す。

表 13：最大ゲート遅延時間とクロック周波数

最大ゲート遅延時間	クロック周波数
38.98 ns	25.62 MHz

５．３ シミュレーション結果の考察

本研究では、マルチサイクルプロセッサである PICO16 を VerilogHDL で記述し、トップダウン設計におけるゲートレベルシミュレーションまで行った。また最適化では、配線数が大きく減少したことなどから、最適化の過程が理解できた。ゲートレベルシミュレーションは動作の確認は取れ、遅延などの情報も得ることができた。また、それを想定した VerilogHDL 記述ができるようになった。ただ、トップダウン設計の途中までしか完了していないことを踏まえると、このマイクロプロセッサの設計はまだ完了したとは言えないので、VLSI センターでのトップダウン設計を完全に理解する必要がある。

シェルソートのテストプログラムでの実行結果から、C 言語による記述と PICO16 の命令を比較して、C 言語と命令語の違いを確認することができた。

また PICO 16 では 27 命令あるが、2 つのレジスタのデータを比較して分岐する命令がないため、この命令を追加することでシェルソートなどの比較分岐が多いプログラムなどには特に有効な効果が得られると考えられる。

６　終わりに

本論文では、ハードウェア記述言語によるマルチサイクルプロセッサ PICO16 の設計について述べた。VLSI センターのツールを使用して、論理レベルシミュレーションからゲートレベルシミュレーションまでを行った。テストパターンは C 言語のシェルソートを PICO16 の命令セットに記述したことにより、ハードウェアとソフトウェアの両方に対する理解を深めることができた。

現在ハードウェア記述言語が普及してきていることから、ハードウェア設計はよりソフトウェア開発に近いものになっている。その中で我々のようにソフトウェアを学んできた者がハードウェアの知識を学ぶことで、ハードウェアとソフトウェアの双方のより深い知識が得られ、双方のことを踏まえた開発ができる能力が身につくと信じている。

今後の課題として、FPGA 上へ実際にダウンロードして動作検証を行い、トップダウン設計の完全な習得を目指したいと考えている。

謝辞

本研究の機会を与えてくださり、貴重な助言ご指導をいただきました山崎勝弘教授、小柳滋教授に深く感謝いたします。また、本研究に関して貴重なご意見をいただきました、Tran So Cong氏、池田修久氏、大八木睦氏、同じハードウェアに関する研究を行った藤原淳平氏、古川達久氏、及び様々な面で貴重な助言と励ましを下さった研究室の皆様に深く感謝いたします。

参考文献

[1]立命館大学 VLSI センター：社会人向け VLSI 設計セミナー レクチャー用マニュアル、2001

[2]John L.Hennessy,、David A .patterson 著、成田光章訳：コンピュータの構成と設計(上)(下)、日経 BP 社、1999

[3]天野英晴、西村克信 著：作りながら学ぶコンピュータアーキテクチャ、培風館、2001

[4]近藤嘉雪 著：C プログラマのためのアルゴリズムとデータ構造、ソフトバンク社、1998

[5]池田修久：ハードウェア記述言語による単一/パイプラインマイクロプロセッサの設計、立命館大学理工学部情報学科卒業論文、2002

[6]大八木睦：ハードウェア記述言語によるマルチサイクル/パイプラインマイクロプロセッサの設計、立命館大学理工学部情報学科卒業論文、2002

[7]上平 祥嗣：KITE マイクロプロセッサを用いたハードウェア/ソフトウェア・コデザイン、立命館大学理工学部情報学科卒業論文、1998.

[8] 田中義久: ハードウェア記述言語による FPGA 上への教育用マイクロプロセッサの実装、立命館大学理工学部情報学科卒業論文、1998

[9] インテル:http://www.intel.co.jp/、2002

[10] VLSI 設計技術研究専門委員会:ディジタル集積回路設計、電子情報通信学会、2002

[11]藤原純平：ハードウェア記述言語による教育用マイクロプロセッサの設計（II)、立命館大学理工学部情報学科卒業論文、2003

[12]古川達久：マルチサイクル・パイプライン方式による教育用マイクロプロセッサの設計と検証、立命館大学理工学部情報学科卒業論文、2003

付録

1　マルチサイクルプロセッサ PICO16 の VerilogHDL 記述

```
`timescale 1ns/100ps

module
new_pico(rst,clk,prog_in,pc,prog_read,data_addr,mdr,dmem_out,dmem_req);

  input            rst, clk;
  input  [15:0] prog_in,dmem_out;
  output          dmem_req,prog_read;
 output [15:0] pc,mdr,data_addr;

  wire [15:0]  reg_dout1, reg_dout2;
  wire [15:0]  dmem_out,reg_in,alu_out,
               reg1,reg2,data_addr,
               regi,prog_in;
  wire [15:0]  mux2_in1,mux2_in2;
  wire [3:0]   com;
  wire [2:0]   addr1,reg_addr2;
  wire         rst, clk;

  reg  [15:0]  pc,dbus,mar,mdr,ir;
  reg  [3:0]   stat;
  reg          mux1_sel,mux2_sel,mux3_sel,
               reg_req,dmem_req,prog_read;


  parameter      ROP = 5'b00000,
                 LDLI= 5'b11100,
                 LDHI= 5'b11101,
                 BNEZ= 5'b01001,
                 BEQZ= 5'b01010,
                 BMI = 5'b01011,
                 BPL = 5'b01100,
                 JR  = 5'b01110,
                 JMP = 5'b01111,
                 JAL = 5'b01101,
                 ADDI= 5'b00110,
                 SUBI= 5'b00111,
                 ANDI= 5'b00010,
                 ORI = 5'b00011,
                 XORI= 5'b00100,
                 HALT= 5'b11111,
                 LD  = 5'b01000,
                 ST  = 5'b01001,
                 SL  = 5'b01100,
                 SR  = 5'b01101;

  parameter      IF  = 4'b0001,
                 RF  = 4'b0010,
                 EX  = 4'b0100,
```

```
                    EX2 = 4'b1000;
alu16       alu1(.ina(reg1),
                 .inb(reg2),
                 .com(com),
                 .y(alu_out));

reg_file regf1(.reg_addr1(reg_addr1),
                 .reg_addr2(reg_addr2),
                 .reg_in(reg_in),
                 .reg_dout1(reg_dout1),
                 .reg_dout2(reg_dout2),
                 .reg_req(reg_req),
                 .rst(rst),
                 .clk(clk));

mux1         mu1(.mux1_in1(reg_dout1),
                 .mux1_in2(pc),
                 .mux1_out(reg1),
                 .mux1_sel(mux1_sel));


mux2         mu2(.mux2_in1(mux2_in1),
                 .mux2_in2(mux2_in2),
                 .mux2_out(reg2),
                 .mux2_sel(mux2_sel));

mux3         mu3(.mux3_in1(alu_out),
                 .mux3_in2(dmem_out),
                 .mux3_out(regi),
                 .mux3_sel(mux3_sel));

assign           com = ((ir[15:11] == ROP &&
                         ir[4:0] != ST &&
                         ir[4:0] != LD) &&
                         rst == 0 &&
                         stat != IF)? ir[3:0]:
                       ((ir[15:11] == ADDI ||
                         ir[15:11] == SUBI ||
                         ir[15:11] == ANDI ||
                         ir[15:11] == ORI  ||
                         ir[15:11] == XORI) &&
                         rst == 0 &&
                         stat != IF)? ir[14:11]:
                       (rst == 1)? 4'bx:
                       (stat == IF)? 4'b0110:
                       ((ir[15:11] == BPL  ||
                         ir[15:11] == BNEZ ||
                         ir[15:11] == BEQZ ||
                         ir[15:11] == BMI  ||
                         ir[15:11] == JMP) &&
                         stat != IF)? 4'b0110:
                       ((ir[15:11] == LDLI ||
                         ir[15:11] == LDHI ||
                         ir[15:11] == JAL) &&
```

```
                stat != IF)? 4'b0001: 4'b0000;

assign          mux2_in1 = (rst == 0)? reg_dout2: 16'bx;

assign          mux2_in2 = ((ir[15:11] == LDLI ||
                             ir[15:11] == ADDI ||
                             ir[15:11] == SUBI) &&
                            rst == 0 &&
                            stat != IF)?
                {ir[7],ir[7],ir[7],ir[7],ir[7],ir[7],ir[7],ir[7],ir[7:0]}:
                            (ir[15:11] == LDHI &&
                             rst == 0 &&
                              stat != IF)? {ir[7:0],8'h00}:
                           ((ir[15:11] == ANDI ||
                              ir[15:11] == ORI  ||
                             ir[15:11] == XORI) &&
                            rst == 0 &&
                             stat != IF)? {8'h00,ir[7:0]}:
                           (((ir[15:11] == BPL &&
                              reg_dout1[15] == 1'h0)||
                             (ir[15:11] == BNEZ &&
                             reg_dout1 != 16'h0000) ||
                             (ir[15:11] == BEQZ &&
                             reg_dout1 == 16'h0000) ||
                             (ir[15:11] == BMI &&
                            reg_dout1[15] == 1'b1)) &&
                             rst == 0 &&
                             stat != IF)?
                {ir[7],ir[7],ir[7],ir[7],ir[7],ir[7],ir[7],ir[7],ir[7:0]}:
                            ((ir[15:11] == JAL ||
                              ir[15:11] == JMP) &&
                            rst == 0 &&
                             stat != IF)?
                {ir[10],ir[10],ir[10],ir[10],ir[10],ir[10:0]}:
                          (((ir[15:11] == BPL &&
                              reg_dout1[15] == 1'b1) ||
                             (ir[15:11] == BNEZ &&
                              reg_dout1 == 16'h0000) ||
                             (ir[15:11] == BEQZ &&
                              reg_dout1 != 16'h0000) ||
                              (ir[15:11] == BMI &&
                              reg_dout1[15] != 1'b1)) ||
                               stat == IF)? 16'h0001:  16'bx;


assign          reg_in = (rst == 0)? dbus: 16'bx;

assign          data_addr = (ir[15:11] == ROP &&
                             ir[4:0] == ST &&
                             rst == 0 && stat != IF)? regi:
                            (ir[15:11] == ROP &&
                              ir[4:0] == LD &&
                             rst == 0 && stat != IF)? mar:
                               16'bx;
```

```
assign          reg_addr1 = (rst == 0)? ir[10:8]: 3'bx;
assign          reg_addr2 = (rst == 0)? ir[7:5]: 3'bx;


always @(posedge clk) begin                  /* STATE */

  if(rst == 1) begin
     stat <= 4'b0000;
   end

  if(stat == 4'b0000 && rst == 0) begin
     stat <= IF;
     ir <= prog_in;
   end

  if(stat == IF && ir[15:11] == HALT && rst == 0) begin
     stat <= 4'b1111;
      end

 if(stat == IF && ir[15:11] != HALT && rst == 0) begin
     stat <= RF;
   end

  if(stat == RF && rst == 0) begin
     stat <= EX;
   end

  if(stat == EX && rst == 0) begin
    if((ir[15:11] == ROP && ir[4:0] == LD) ||
         (ir[15:11] == ROP && ir[4:0] == ST) ||
         ir[15:11] == JAL) begin
        stat <= EX2;
     end
     else begin
        stat <= IF;
        ir <= prog_in;
     end
   end

  if(stat == EX2 && rst == 0) begin
     stat <= IF;
     ir <= prog_in;
   end
end


always @(negedge clk)                        /* REQUIRE */
   begin

     if(rst == 1) begin
        pc  <= 16'h0000;
        prog_read <= 1'b0;
        dmem_req <= 1'b0;
```

```
  end

  if(stat == 4'b0000 &&
     rst == 0) begin
    mux2_sel <= 1'b0;
     mux1_sel <= 1'b0;
     reg_req <= 1'b0;
    prog_read <= 1'b1;
     mux3_sel <= 1'b1;
  end

if(stat == IF && rst == 0 &&
     (ir[15] == 0  || ir[15] == 1))                  /*IF stage*/
     begin
       prog_read <= 1'b0;
        if(ir[15:11] == ADDI ||
                  ir[15:11] == SUBI ||
                  ir[15:11] == ANDI ||
                   ir[15:11] == ORI   ||
                  ir[15:11] == XORI ) begin
         dbus <= regi;
         mux2_sel <= 1'b0;
         mux1_sel <= 1'b1;
        end
      else if(ir[15:11] == LDLI ||
                  ir[15:11] == LDHI ||
                  ir[15:11] == JAL) begin
         dbus <= regi;
          end
      else if(ir[15:11] == ROP &&
                  ir[4:0] == LD)   begin
         dbus <= regi;
         mux2_sel <= 1'b1;
         mux1_sel <= 1'b1;
       end

      else if (ir[15:11] != BPL  &&
                   ir[15:11] != BNEZ &&
                   ir[15:11] != BEQZ &&
                    ir[15:11] != BMI   &&
                    ir[15:11] != JMP   &&
                    ir[15:11] != JR)   begin
         dbus <= regi;
         mux2_sel <= 1'b1;
         mux1_sel <= 1'b1;
         reg_req <= 1'b0;
       end

     end
```

```
if(stat == RF && rst == 0) begin                    /*RF stage*/

    if (ir[15:11] == BPL  ||
        ir[15:11] == BNEZ ||
        ir[15:11] == BEQZ ||
        ir[15:11] == BMI  ||
        ir[15:11] == JMP  ||
        ir[15:11] == JR) begin
      dbus <= regi;
    end
    else if(ir[15:11] == JAL) begin
      reg_req <= 1'b1;
    end
    else if(ir[15:11] == ADDI ||
            ir[15:11] == SUBI ||
            ir[15:11] == ANDI ||
            ir[15:11] == ORI  ||
            ir[15:11] == XORI) begin
      reg_req <= 1'b1;
      pc <= dbus;
      dbus <= regi;
    end
  else if(ir[15:11] == ROP &&
            ir[4:0] == LD) begin
      mux3_sel <= 1'b0;
      dmem_req <= 1'b0;
      mar <= reg_dout2;
      pc <= dbus;
    end
    else if(ir[15:11] == ROP &&
            ir[4:0] == ST) begin
      mdr <= reg_dout2;
      pc <= dbus;
    end
    else begin
      pc <= dbus;
      dbus <= regi;
      reg_req <= 1'b1;
    end
end // if (stat == RF)


if(stat == EX && rst == 0)  begin                   /*EX stage*/
    if(ir[15:11] == ROP) begin
      if(ir[4:0] == LD ) begin
       // pc <= dbus;
        dbus <= regi;
        reg_req <= 1'b1;
       end
      else if(ir[4:0] == ST) begin
        pc <= dbus;
        dmem_req <= 1'b1;
```

```
        end
        else begin
          mux1_sel <= 1'b0;
          mux2_sel <= 1'b0;
           reg_req   <= 1'b0;
           prog_read <= 1'b1;
        end
      end
      else if(ir[15:11] == ADDI ||
                ir[15:11] == SUBI ||
                ir[15:11] == ANDI ||
                ir[15:11] == ORI   ||
                ir[15:11] == XORI)
        begin
          mux2_sel <= 1'b0;
          mux1_sel <= 1'b0;
          reg_req <= 1'b0;
          prog_read <= 1'b1;
        end
      else if(ir[15:11] == BNEZ ||
                ir[15:11] == BEQZ ||
                ir[15:11] == JMP ||
                ir[15:11] == JR   ||
                ir[15:11] == BPL ||
                ir[15:11] == BMI)
        begin
          pc <= dbus;
          mux1_sel <= 1'b0;
          mux2_sel <= 1'b0;
          prog_read <= 1'b1;
        end
      else if(ir[15:11] == JAL) begin
        reg_req <= 1'b0;
        dbus <= regi;
      end
      else if(ir[15:11] == LDLI ||
                ir[15:11] == LDHI) begin
        reg_req <= 1'b0;
        prog_read <= 1'b1;
        mux2_sel <= 1'b0;
        mux1_sel <= 1'b0;
      end
    end  // if (stat == EX && rst == 0)


if(stat == EX2 && rst == 0)begin                    /*EX2 stage*/
      prog_read <= 1'b1;
    if(ir[15:11] == ROP &&
         ir[4:0] == LD) begin
        mux2_sel <= 1'b0;
        mux1_sel <= 1'b0;
        mux3_sel <= 1'b1;
        reg_req <= 1'b0;
```

```
            end
            else if(ir[15:11] == ROP &&
                    ir[4:0] == ST) begin
              dmem_req <= 1'b0;
              mux1_sel <= 1'b0;
              mux2_sel <= 1'b0;
            end
            else if(ir[15:11] == JAL) begin
              mux1_sel <= 1'b0;
              mux2_sel <= 1'b0;
               reg_req  <= 1'b0;
              pc <= dbus;
            end
          end
      end // always @ (negedge clk)

endmodule


module alu16(ina, inb, com, y);

   input   [15:0] ina, inb;
    input   [3:0]   com;
   output [15:0] y;

   wire [15:0]      ina, inb, y;
    wire [3:0]      com;

   assign           y = (com == 4'b0000)? ina:
                        (com == 4'b0001)? inb:
                       (com == 4'b0010)? ina & inb:
                       (com == 4'b0011)? ina | inb:
                       (com == 4'b0100)? ina ^ inb:
                        (com == 4'b0101)? ~inb:
                       (com == 4'b1100)? inb << 1:
                       (com == 4'b1101)? inb >> 1:
                       (com == 4'b0110)? ina + inb:
                        (com == 4'b0111)? ina - inb:16'bx;

endmodule


module mux3(mux3_in1,mux3_in2,mux3_out,mux3_sel);

   input   [15:0] mux3_in1,mux3_in2;
    input           mux3_sel;
   output [15:0] mux3_out;

   wire [15:0]      mux3_out,mux3_in1,mux3_in2;
    wire            mux3_sel;

   assign           mux3_out = (mux3_sel == 1'b1)? mux3_in1:
                               (mux3_sel == 1'b0)? mux3_in2: 16'bx;
```

```
endmodule // mux3


module mux2(mux2_in1,mux2_in2,mux2_out,mux2_sel);

   input   [15:0] mux2_in1,mux2_in2;
   input          mux2_sel;
  output [15:0] mux2_out;

  wire [15:0]     mux2_out,mux2_in1,mux2_in2;
  wire            mux2_sel;

  assign          mux2_out = (mux2_sel == 1'b1)? mux2_in1:
                             (mux2_sel == 1'b0)? mux2_in2: 16'bx;

endmodule // mux2

module mux1(mux1_in1,mux1_in2,mux1_out,mux1_sel);

   input   [15:0] mux1_in1,mux1_in2;
   input          mux1_sel;
  output [15:0] mux1_out;

  wire [15:0]     mux1_out,mux1_in1,mux1_in2;
  wire            mux1_sel;

  assign          mux1_out = (mux1_sel == 1'b1)? mux1_in1:
                             (mux1_sel == 1'b0)? mux1_in2: 16'bx;

endmodule // mux1


module reg_file (reg_addr1, reg_addr2, reg_in, reg_dout1, reg_dout2,
reg_req,rst,clk);

   input   [2:0]  reg_addr1, reg_addr2;
   input   [15:0] reg_in;
   input          reg_req,rst,clk;
  output [15:0] reg_dout1, reg_dout2;

   reg   [15:0]    register0, register1,
                   register2, register3,
                   register4, register5,
                   register6, register7;
   wire            reg_req,rst,clk;
  wire [2:0]       reg_addr1, reg_addr2;
  wire [15:0]      reg_dout1, reg_dout2,reg_in;


// read req=0
  assign     reg_dout1 = (reg_addr1 == 3'b000 &&
                          reg_req == 0)? register0:
                         (reg_addr1 == 3'b001 &&
```

```
                                reg_req == 0)? register1:
                               (reg_addr1 == 3'b010 &&
                                reg_req == 0)? register2:
                               (reg_addr1 == 3'b011 &&
                                reg_req == 0)? register3:
                               (reg_addr1 == 3'b100 &&
                                reg_req == 0)? register4:
                               (reg_addr1 == 3'b101 &&
                                reg_req == 0)? register5:
                               (reg_addr1 == 3'b110 &&
                                reg_req == 0)? register6:
                                (reg_addr1 == 3'b111 &&
                                reg_req == 0)? register7:16'bx;

  assign     reg_dout2 = (reg_addr2 == 3'b000 &&
                                reg_req == 0)? register0:
                               (reg_addr2 == 3'b001 &&
                                reg_req == 0)? register1:
                               (reg_addr2 == 3'b010 &&
                                reg_req == 0)? register2:
                               (reg_addr2 == 3'b011 &&
                                reg_req == 0)? register3:
                               (reg_addr2 == 3'b100 &&
                                reg_req == 0)? register4:
                               (reg_addr2 == 3'b101 &&
                                reg_req == 0)? register5:
                               (reg_addr2 == 3'b110 &&
                                reg_req == 0)? register6:
                               (reg_addr2 == 3'b111 &&
                                reg_req == 0)? register7:16'bx;

//write reg_req=1
  always @(posedge clk)
     begin
       if(reg_req) begin
          case (reg_addr1)
            3'b000: register0 = reg_in;
            3'b001: register1 = reg_in;
            3'b010: register2 = reg_in;
            3'b011: register3 = reg_in;
            3'b100: register4 = reg_in;
            3'b101: register5 = reg_in;
            3'b110: register6 = reg_in;
            3'b111: register7 = reg_in;
          endcase // case(reg_addr1)
        end

       if(rst == 1) begin
          register0 = 16'b00000000_00000000;
          register1 = 16'b00000000_00000000;
          register2 = 16'b00000000_00000000;
          register3 = 16'b00000000_00000000;
          register4 = 16'b00000000_00000000;
          register5 = 16'b00000000_00000000;
```

```
            register6 = 16'b00000000_00000000;
            register7 = 16'b00000000_00000000;
          end
        end


endmodule // reg_file
```

２　シェルソートテストプログラム

```
 0:11100000_00010100    //LDLI r0 #20
 1:00000001_00000001     //MV     r1  r0
 2:11100010_00000001    //LDLI r2 #1
 3:11100101_00000000    //LDLI r5 #0
 4:00111001_00001001     //SUBI  r1,#9
 5:00110101_00000001    //ADDI r5 #1
 6:01100001_11111110    //BPL    r1,#-2
 7:00111101_00000001     //SUBI  r5,#1
 8:01010101_00001010    //BEQZ  r5 #10
 9:11100100_00000010    //LDLI r4 #2
10:00000011_01000001     //MV     r3  r2
11:00000010_01100110     //ADD    r2,r3
12:00111100_00000001     //SUBI  r4,#1
13:01001100_11111110     /BNEZ r4 #2
14:00110010_00000001    //ADDI r2 #1
15:00000110_10100001     //MV     r6  r5
16:00000110_01000111     //SUB    r6,r2
17:01100110_11110111     //BPL    r6,#-8
18:00000111_01000001     //MV     r7  r2
19:00000010_11100001     //MV     r2  r7
20:01010010_00100011     //BEQZ r2 #35
21:01111000_00001001     //JMP    #+9
22:11100110_00000000    //LDLI r6 #0
23:11100001_00000011    //LDLI r1 #3
24:00000010_00100111     //SUB    r2,r1
25:00110110_00000001    //ADDI r6 #1
26:01100010_11111110     //BPl    r2,#-2
27:00111110_00000001    //SUBI  r6,#1
28:00000010_11000001     //MV     r2  r6
29:01100010_11110111     //BMI    r2,#-8
30:00000011_01000001     //MV     r3  r2
31:00000001_00000001     //MV     r1  r0
32:00000001_01100111     //SUB    r1  r3
33:01011001_11110101     //BMI    r1  #-11
34:01010001_11110100     //BEQZ  r1 #-12
35:00000100_01100001     //MV     r4  r3
36:00000101_10000001     //MV     r5  r4
37:00000100_01000111     //SUB    r4  r2
38:01011100_00001111     //BMI    r4  #15
39:00000110_10101000     //LD     r6  (r5)
40:00000111_10001000     //LD     r7  (r4)
```

```
41:00000001_11100001    //MV     r1 r7
42:00000001_11000111    //SUB    r1 r6
43:01011001_00001010    //BMI    r1 #10
44:01010001_00001001    //BEQZ r1 #9
45:00000001_11000001    //MV     r1 r6
46:00000110_11100001    //MV     r6 r7
47:00000111_00100001    //MV     r7 r1
48:00000101_11001001    //ST    (r5) r6
49:00000100_11101001    //ST    (r4) r7
50:00000100_10100001    //MV     r4 r5
51:00000100_01000111    //SUB    r4 r2
52:01111111_11110000    //JMP    #16
53:00110011_00000001    //ADDI r3 #1
54:01111111_11101001    //JMP    #23
55:11111111_11111111    //HALT
```