修士論文

システム LSI のオンチップテストを実行するためのプログラム環境の構築

村田泰亮

東京大学大学院　工学系研究科　電子工学専攻

学籍番号 46411

指導教員　池田 誠 助教授

目次

図目次

表目次

第１章

序章

1．1　研究の背景

近年 LSI の高集積化が進んでおり、フィールドでの LSI テストの重要性は高まって来ている。また最近ではプロセッサや専用回路、メモリなどの様々な機能ブロックを半導体チップ上に集積化し、一つの LSI でシステムの機能を実現できるシステム LSI による製品も多数登場している。システム LSI では LSI の複雑さが大幅に増すため、LSI の開発においてテストは非常に重要な意味を持っている。

従来使われてきたテスト手法はインサーキットテストと呼ばれる手法である。この手法は、LSIの外部入出力ピンに直接プローブを接触させピンの値を読み取ることによってテスト行うものである。しかし、デバイスの高集積化によるピン数の増加、パッケージの小型化によるピン間隔の減少により、ピン間隔がプローブの先端径に比べて小さすぎプローブを接触させることが困難になる、という問題が発生している。さらに、多くのピン全てにプローブを接触させ、全ての入出力ピンへの値の設定、読み出しをすることが非現実的になる、といった問題も発生している。また、BGAP(Ball Glid Array Package)等のプローブを当てられないパッケージも存在している。このように、現在のLSIテストにおいてインサーキットテスト法は物理的に限界を迎えている。

そのため現在用いられている手法は、テストの実行に必要な回路を内部コア回路と共に一つのチップ上に集積し、外部からテスト制御信号を入力することでテストを実行する、オンチップテスト手法である。

1．2　オンチップテスト手法

オンチップテストとは、テストの実行を行う回路をチップ上に集積し、内部コアのテストを行うテスト方法である。現在LSIテストに広く用いられている手法に、バウンダリスキャンテストという手法がある。バウンダリとは境界のことであり、LSIのシリコンチップと入出力ピンの境界を意味する。バウンダリスキャンテストとは、シリコンチップと入出力

ピンの境界を走査して行うテストである。

バウンダリスキャンテストにおいてはテスト対象の内部コアロジックと外部入出力ピンとの間に、バウンダリスキャンレジスタを配置する。バウンダリスキャンレジスタの構造はシフトレジスタになっており、セルと呼ばれる。バウンダリスキャンテストに対応したデバイスの図を、図 1.1 に示す。バウンダリスキャンテストの中でも現在最も広く採用されているのが、IEEE1149.1[1]によってスタンダード化されている、JTAG テスト[2]である。また、JTAG テストにおけるバウンダリスキャンレジスタの構造を、図 1.2 に示す。

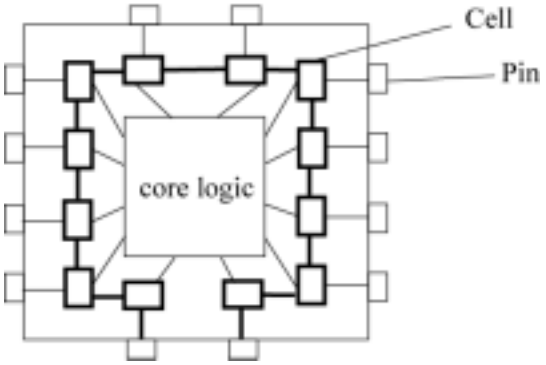


図 1.1　バウンダリスキャンテスト対応デバイス

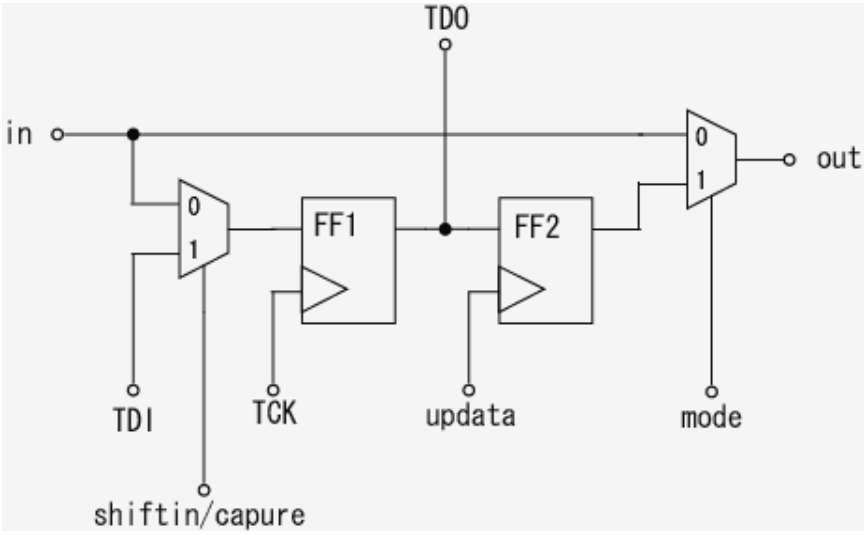


図 1.2　バウンダリスキャンレジスタ

バウンダリスキャンレジスタは、mode の値が 0 の時チップは通常動作となり、in から out へと、内部コアから外部 I/O ピン、またはその逆へと信号がセルを通過する。mode の値が 1 のときは、セルはテストモードで動作する。このときのセルの動作は、capture ステート、shift ステート、update ステートに分けられる。

capture ステートは、shiftin/capture=0 のときで、in から入力された信号値を TCK の立ち上がりで FF1 に獲得する。shift ステートは、shiftin/capture=1 としたときで、FF1 の内容が TDO にシフト出力され、新しい値が TDI からシフト入力される。TDI と TDO はそのセルの前後のセルに接続され、シフトレジスタによるスキャンパスが形成され、データはこのスキャ

ンパスをシフトされて伝達される。update ステートは、update=1 となったときで、FF1 の内容が FF2 にラッチされて、その値が out から出力される。

バウンダリスキャンテストにおけるコアのファンクショナルテストの手順について説明する。まず、内部コアの入力となるセルに、テストパターンがセットされる。テストパターンは、外部に接続されている I/O ピンから入力され、スキャンパスをシフトインしてセットされる。そして、セットされた値が内部コアに入力として与えられる。

そして、内部コアにテストパターンを入力した結果が、出力側のセルに出力される。出力結果は、スキャンパスをシフトアウトされ、I/O ピンから外部に出力される。このようにして、内部コアロジックのファンクショナルテストが実行される。

バウンダリスキャンレジスタはまた、I/O ピンと内部コアを切り離す役割もある。すなわち、外部入力ピンから入力された値を内部コアへと出力ラッチする、内部コアからの出力をラッチする、セルにセットされた任意の値を入力側は内部コアのみに、出力側は外部 I/O ピンのみに出力する、ということが可能である。これにより、チップと外部ロジックとの接続のテスト(インターコネクトテスト)や、外部ロジックのテスト、ということが可能となっている。

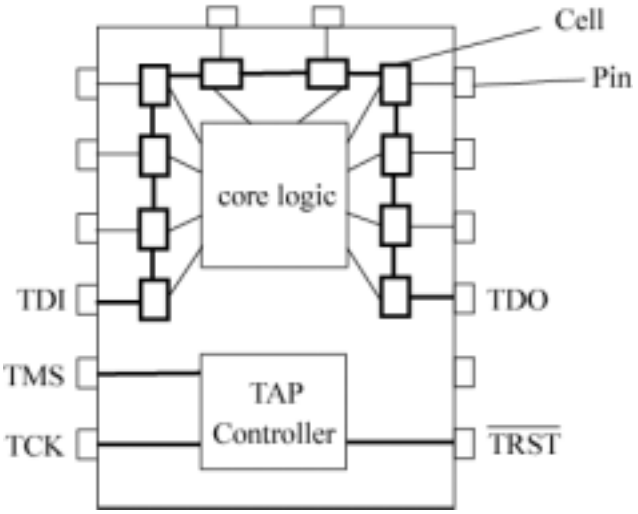


図 1.3　JTAG 対応デバイス

セルとその動作をコントロールする TAP(Test Access Port)コントローラによって構成される。必要な外部接続ピンの本数は、４本または５本である。図 1.3 に、JTAG 対応デバイスの例を示す。JTAG テストを実行する時は、外部から信号線を通して TAP コントローラに制御信号やテストデータを与え、TAP コントローラはそれに応じてバウンダリレジスタを動作させて、テスト結果を外部に出力する。

しかし、システム LSI のテストを行うためには、この JTAG テストにはいくつか不十分な点が存在する。一つにはテストを実行する際にデバイスを接続するホストコンピュータに専用のソフトウェアが必要となる、という点である。さらに、セルに値をセットするには

シフトレジスタによる 1 本のスキャンチェーンをシフトさせる必要があり、個々のセルとパラレルに値をセットできない、という点も挙げられる。すなわり、JTAG では一つのセルだけに連続して値をセットしたり、連続して値を読み出したり、ということができない。また、JTAG テストはファンクショナルテストにのみ対応しており、遅延やノイズなどのアナログ値の測定ができない、という問題点もある。

アナログ信号のバウンダリスキャンテストを行うためのスタンダードとして、IEEE1149.4[3]や、P1500[4]なども定められているが、信号波形を簡単な入出力により取得する、ということは不可能である。

システム LSI のテストにおいては、1 や 0 のデータだけ測定するのでは不十分であり、これらノイズや温度などのパラメータを測定することが必要となってくる。しかし、それらの測定回路を個別に設置してもコントロールが難しい。

オンチップテストに関する最近の研究例をいくつか示す。

バウンダリスキャンテストに関しての最近の研究としては、JTAG テストを拡張し、JTAG では不可能なパラメータをテストする手法、というのがいくつか提案されている。[5]では、コア間の接続におけるノイズや遅延などのシグナルインテグリティを、受信側のセルに ILS(Integrity Loss Censor)回路という回路を設け検出する、という研究が行われている。また[6]では、バウンダリスキャンセルを用いて信号のモニタリングを行っている。同期モードと非同期モードの二つのモードをもち、同期モードではセルはスキャンチェーンを形成する。このときスキャンチェーンを一本のシリアルのチェーンではなく、パラレルの複数のチェーンとして扱うことも可能である。非同期モードでは、セルがバイパスされ、クロック無しで信号の転送を行い、信号の実時間モニタリングを実行する。

波形取得をオンチップテスト回路で実行している例としては、[7]では回路内の信号波形を観測するために、回路内にサンプリングオシロスコープ回路を設けと、アナログの I/O を用いずに、信号波形の取得を行っている。

インタフェースの問題については、[8]では、JTAG テストを使用するデバッガを USB でホストコンピュータにより接続し、ユーザインタフェースがホストシステムに USB ポートを通してインストールされ、より汎用な環境で JTAG を動作させる、という研究が行われている。

1. 3　研究の目的

前節で挙げた JTAG テストでは不十分な点を改良し、より簡単かつ柔軟な LSI のテストを可能にするため、CPU 及びその上で動作するインタプリタを搭載した、オンチップテストプラットフォームについての研究を行った。このオンチップシステムの概略は図 1.4 に示されている。ハードウェア構成は、CPU、バウンダリレジスタ、オンチップテストインタフェース(バウンダリレジスタの制御等を行う)、シリアル I/O、ROM/RAM によって構成される。

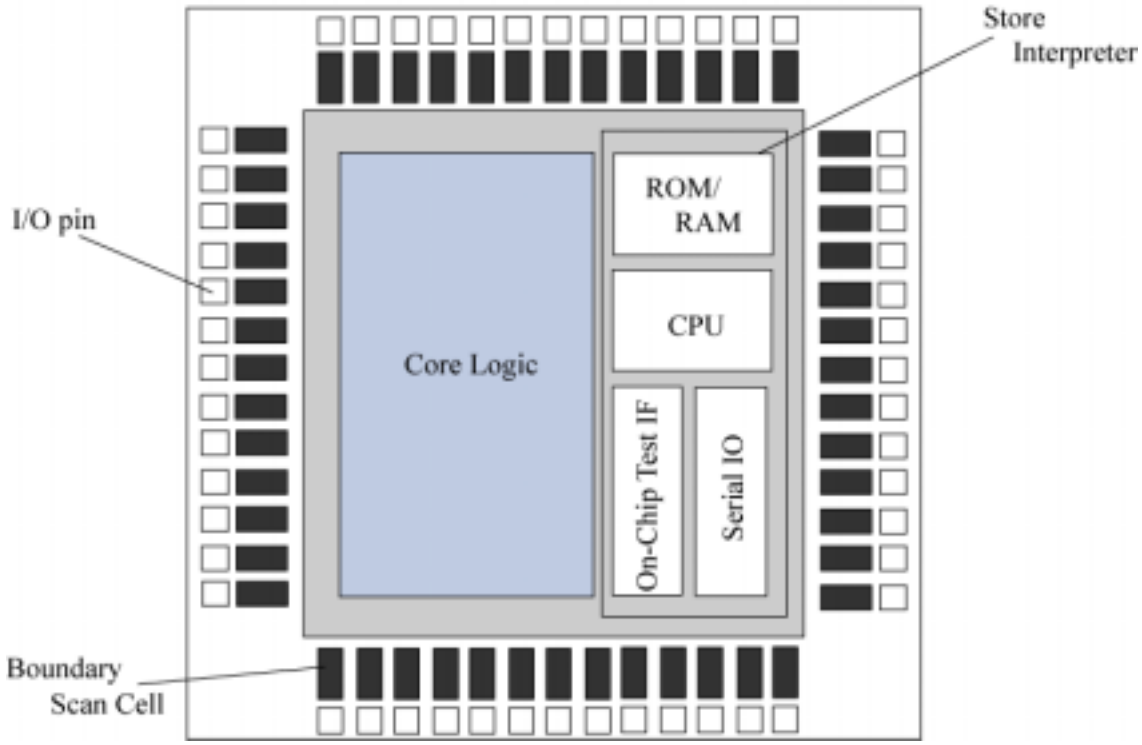


図 1.4　オンチップテストシステムの概略

インタプリタは ROM/RAM のメモリ領域に格納され、CPU 上で動作する。インタプリタは BASIC 言語をベースにした言語仕様のものとし、ユーザにプログラム環境を提供する。ファンクショナルテストの実行は、インタプリタに外部から命令が入力され、その命令に従い CPU がバウンダリレジスタを動作させ、行われる。また、本研究ではパラメトリックテストとして信号波形の取得を行うことを考えている。遅延生成回路と、各セルに電圧比較回路を追加することで、ある時点での信号の電圧波形が取得でき、遅延やノイズを調べることが可能となる。パラメトリックテストの実行も、インタプリタに命令を入力することにより行われ、セットするパラメータや出力結果もデジタル値として、インタプリタ上で扱える。

JTAGテストでは、ホストコンピュータ側に専用ソフトウェアが必要となるが、本テストシステムでは、LSI側にインタプリタを搭載することで、専用のソフトウェアは必要とならない。チップとホストコンピュータとの通信はRS-232Cによって行われるため、ホストコンピュータ側にはRS-232Cに対応した端末ソフトウェア(Windowsではハイパーターミナルなど)があれば充分であり、LSIに電源をつなぎ、普通のパソコンとRS-232Cで接続するだけで、すぐにLSIのテストを行うことが可能になる。

チップ上のプロセッサを用いてオンチップテストを行う、という研究例はいくつか存在するが[9][10]、チップ上に CPU とインタプリタをもち、プログラム環境を実現する、という研究はこれまで行われていない。チップ上の CPU によりプログラム環境を実現する利点

は、第一に機能の追加が容易である、という点が挙げられる。波形取得回路や温度測定回路等の測定回路を設けても、コントロールの問題があるが、この方法ならば CPU から制御制御信号を与えて回路をコントロールし、データを読み出して出力することで、テストが実行できる。このように新しい機能を追加しても、ユーザインタフェースの設計は、インタプリタをソフトウェア的に修正するだけでよく、容易である。近年プロセスの微細化が進んでおり、CPU を搭載することによるハードウェアオーバーヘッドが小さくなっている点もあり、これらチップ上のテスト用回路を総括してコントロールするためにはオンチップの CPU を設け、その CPU によるプログラム環境を実現してテストを行うことがシステム LSI のテストに適している。

本研究では、バウンダリレジスタからのデータの読み出し、データの書き込み、さらには信号波形の取得を行うテストシステムを構築する。

ハードウェアの設計は、本研究室によって既に行われている。一つのチップ上に集積可能なハードウェア使用量は上限があるため、オンチップテストシステムのハードウェアオーバヘッドが大きくなると、回路の規模を制限することになってしまう。そのため、オンチップテスト実行に必要な機能を実現する最小限の規模のメモリ容量、回路構成を求める必要がある。

本研究の目的は、このオンチップテストシステム上で動作するインタプリタの仕様を決定した上でハードウェアに実装し、動作を確認する。また、バウンダリからの値の読み出し命令、バウンダリへの値の書込み命令など、LSI テストにおいてどのようなテスト専用命令が必要になるかを調べ実装する。実装した命令は、プログラムのワード数、実行ステップ数、使用メモリ量などを測定し、評価する。さらに、パラメトリックテスト実行用のアナログ回路を設計し、それを制御し波形取得を行う。

1．4　本論文の構成

本論文の構成は以下のようになっている。まず、第2章では今回インタプリタを動作させ、テストを行うために使用したハードウェアについて説明する。次に第3章で、作成したインタプリタの具体的な言語仕様、動作、設計方法について述べる。続いて第4章で、ハードウェアをFPGA上に実装し、ホストコンピュータと接続してインタプリタの実行を行った結果と、パラメトリックテスト用にFPGAに接続したハードウェアの構成について示し、この回路で波形取得を行った結果を示す。さらに第5章において使用する記憶領域や実行ステップ数の評価を行った。最後に第6章で結論を述べる。

第２章

ハードウェアの構成

2. 1　CPU

本テストシステムでは、インタプリタを実行し、テスト回路の制御を行うために、チップ上にテスト専用の CPU を設ける。今回用いた CPU は、本研究室によって設計された簡易 16 ビットマイクロプロセッサである。この CPU の大まかな仕様は、以下のようになっている。

- レジスタは16bit のレジスタが16 本(R0～R15)。ユーザが自由に使えるのはR1～R15の15本。
- プログラムメモリとデータメモリを別に持つハーバードアーキテクチャ。
- プログラムメモリ、データメモリともに1 ワード16bit のワードアドレッシング。
- I/O 専用の命令はない。入出力はメモリマップトI/O で行う。

このCPUの回路は、表2.1に示したモジュールにより構成されている。より具体的な構造を図2.1に示す。

このCPUの命令フィールド構成は図2.2の通りである。OPはオペコード、RA, RB, RCはレジスタ、IMMは8bitの即値、DISPはジャンプ先を表す12bitの即値である。PCは現在のプログラムカウンタの値を表している。また、キャリーフラグとゼロフラグの二つのフラグを持ち、減算比較命令subfの結果によりセットされる。減算の結果が負ならキャリーフラグが、ゼロならゼロフラグがセットされる。条件分岐命令BRCではキャリーフラグの値に応じて、BRNZではゼロフラグの値に応じて条件分岐を行う。

具体的な命令セットは表2.2の通りである。命令の総数は15個である。

表 2.1　CPU のブロック構成

PC	プログラムカウンタ。命令の読み込み、ジャンプ･条件分岐命令などを処理する。
DIO	データ入出力回路。データメモリのアクセス(LOAD、STORE)を制御する。
ALU	演算回路。算術演算、ロジック演算などを行う。
REGFILE	レジスタファイル回路。レジスタのアクセスを制御する。

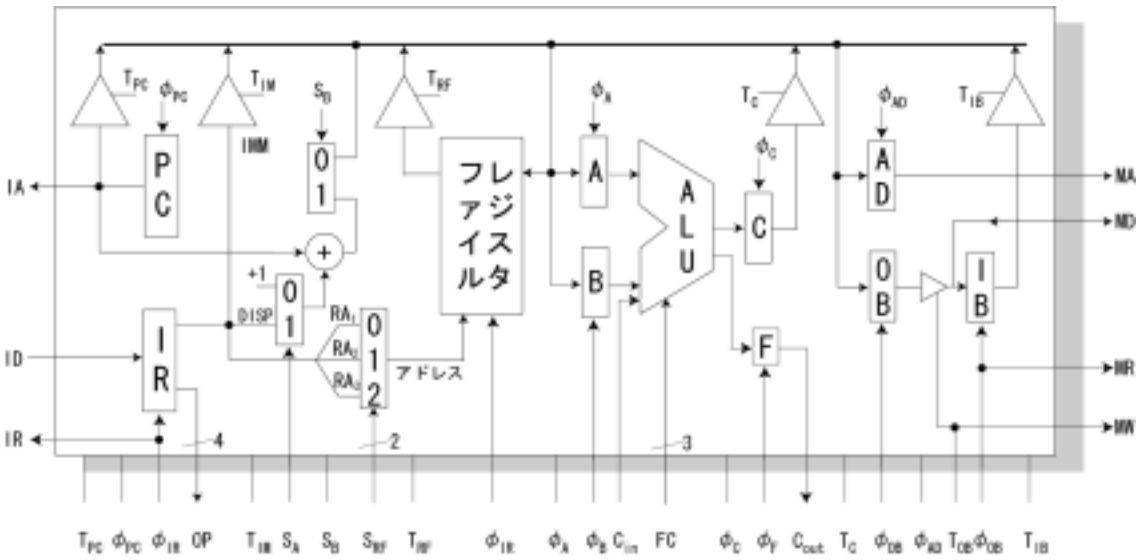


図 2.1　CPU のデータパス構造

図 2.2　命令フィールド構成

表 2.2　命令セット

命令		OP	機能
ADD	RA, RB, RC	0	加算命令。RA と RB の内容を加算し、結果を RC にセットする。
SUB	RA, RB, RC	1	減算命令。RA と RB の内容を減算し、結果を RC にセットする。
SHR	RA, RC	2	論理シフト命令。RA の内容を 1bit 右にシフトし、結果を RC にセットする。
SUBF	RA, RB, RC	3	減算比較命令。RAからRBの内容を減算し、結果をRCにセットする。RA>RBのときキャリーフラグがセットされ、RA = RBのときゼロフラグがセットされる。
AND	RA, RB, RC	4	AND 命令。RA と RB の内容を AND し、結果を RC にセットする。
OR	RA, RB, RC	5	OR 命令。RA と RB の内容を OR し、結果を RC にセットする。
XOR	RA, RB, RC	6	XOR 命令。RA と RB の内容を XOR し、結果を RC にセットする。
XNOR	RA, RB, RC	7	XNOR 命令。RA と RB の内容を XNOR し、結果を RC にセットする。
ST	RB, [RA]	8	データライト命令。RB の内容を RA が示すメモリアドレスにストアする。
LD	[RA], RC	9	データリード命令。RA が示すメモリアドレスのデータを RC にロードする。
JS	RA, RC	A	ジャンプ命令。RAにセットされている命令アドレスにジャンプし、次の命令アドレスをRBにセットする。
BRC	IMM	B	条件分岐命令。キャリーフラグがセットされているとき、PC+IMMにジャンプする。それ以外のときは次の命令を実行する。
SET	IMM, RC	C	レジスタセット命令。8bitの数値IMMをRC にセットする。RCの上位8bitは0にクリアされる。
SETHI	IMM, RC	D	レジスタセット命令。8bitの数値IMMをRCの上位8bitにセットする。RCの下位8bitは保存される。
BRNZ	IMM	E	条件分岐命令。ゼロフラグがクリアされているとき、PC+IMMにジャンプする。それ以外のときは次の命令を実行する。

2. 2　バウンダリレジスタ

本システムで用いるバウンダリレジスタ回路について説明する。今回 FPGA への実装を行ったバウンダリレジスタ回路を図 2.3 に示す。この回路は当研究室にて設計された。

バウンダリレジスタ回路は、コアからの出力信号(A)、コアへの入力信号(Y)、コアからの制御信号(En)の 3 つの信号を扱う。バウンダリレジスタの内部には各信号に 2 つのレジスタがあり、バウンダリレジスタから内部コア、または外部 I/O ピンへと出力する値がセットされる。バウンダリレジスタは、この 3 つの信号それぞれに対応するデータバスを 1 本もち(ipYBus, ipABus, ipEBus)、このバスによりオンチップテストインタフェース回路とのデータの入出力が行われる。扱う値は各信号ともに 1bit のデジタル信号であり、ピン一つごとにこのバウンダリレジスタが設置される。

バウンダリレジスタの制御は、オンチップテストインタフェース回路により生成された制御信号が入力されることで行われる。iadrs は操作を実行するバウンダリレジスタを指定する。testmode はバウンダリレジスタをテストモードか通常モードのどちらかにセットする。通常モードではコアの入出力(corey, corea, coreen)と外部入出力ピン(ioy, ioa, ioen)がそのまま繋がり、チップは通常動作を行う。テストモードではコアの入出力と外部入出力ピンが切り離され、読み出し操作、書き込み操作を実行可能になる。ipRW、padwriteck はバウンダリレジスタの入出力を制御する。

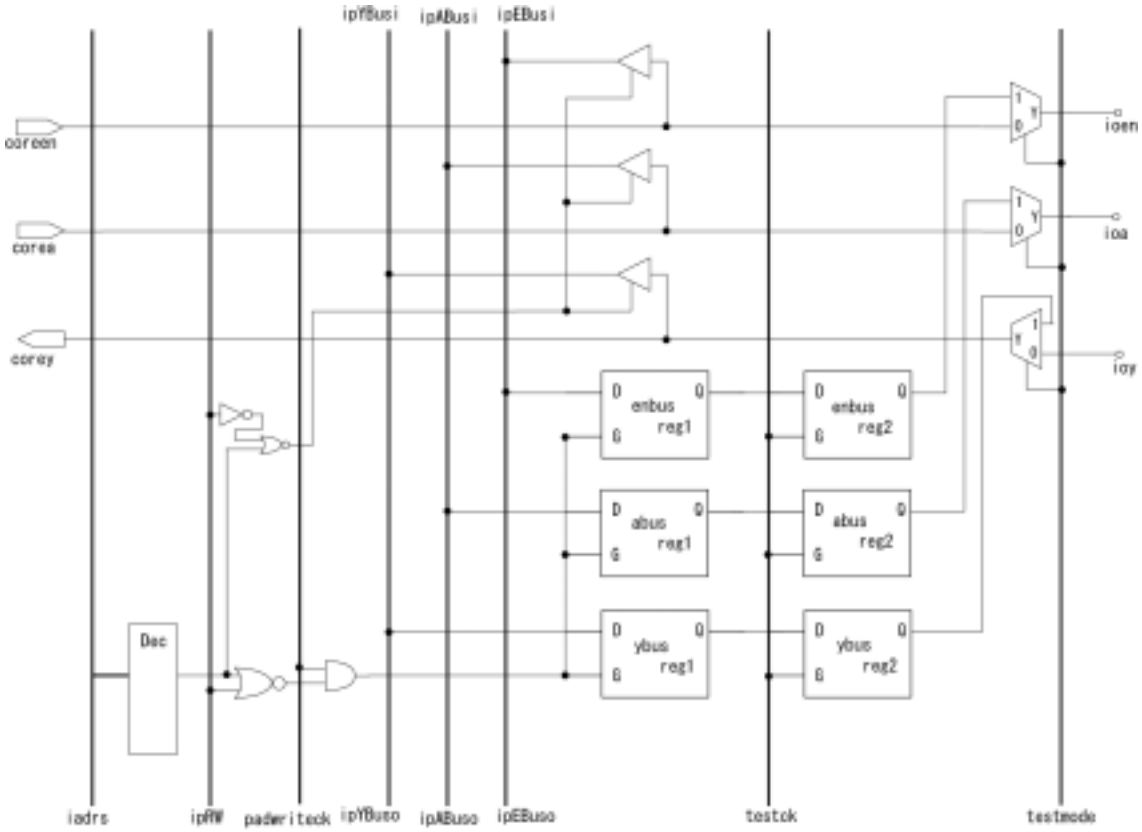


図 2.3　実装したバウンダリレジスタ回路

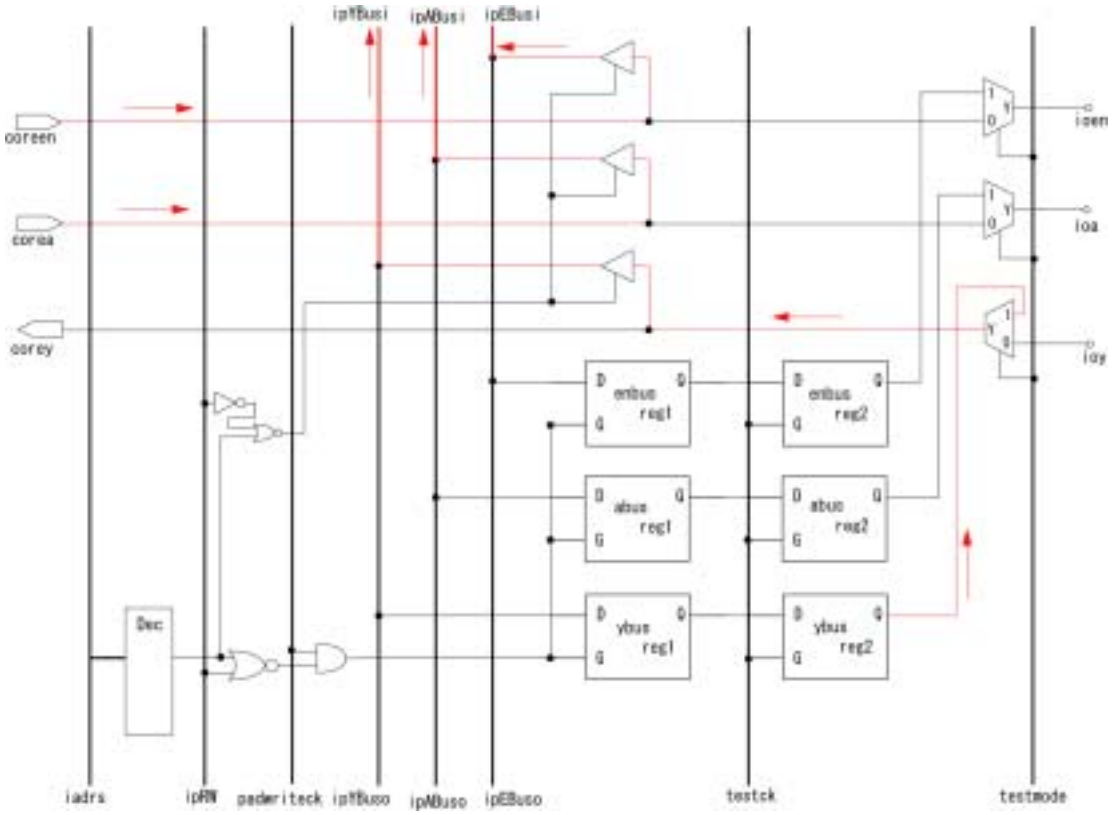


図 2.4　読み出し時の信号の流れ

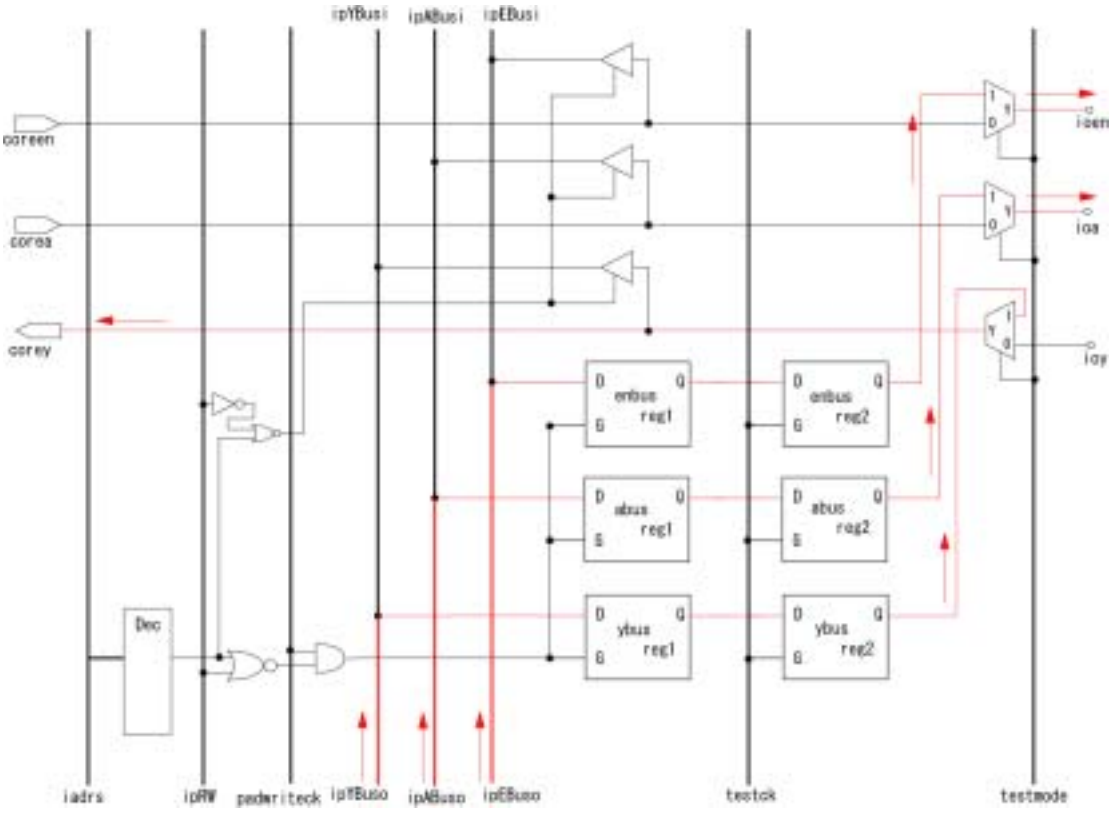


図 2.5　書き込み時の信号の流れ

以下、読み出しと書き込みの動作について説明する。図 2.4 はバウンダリレジスタから値を読み出す時の信号の流れを表したものである。コアから出力された信号が、バスにより出力される。

図 2.5 にはバウンダリレジスタへ値を書き込む時の信号の流れを示してある。バスを通してバウンダリレジスタに入力された信号は、左側のレジスタにラッチされる。そしてラッチされた信号はテストクロック(testck)の立ち上がりで右側のレジスタに伝わり、書き込み信号がコアに入力される。

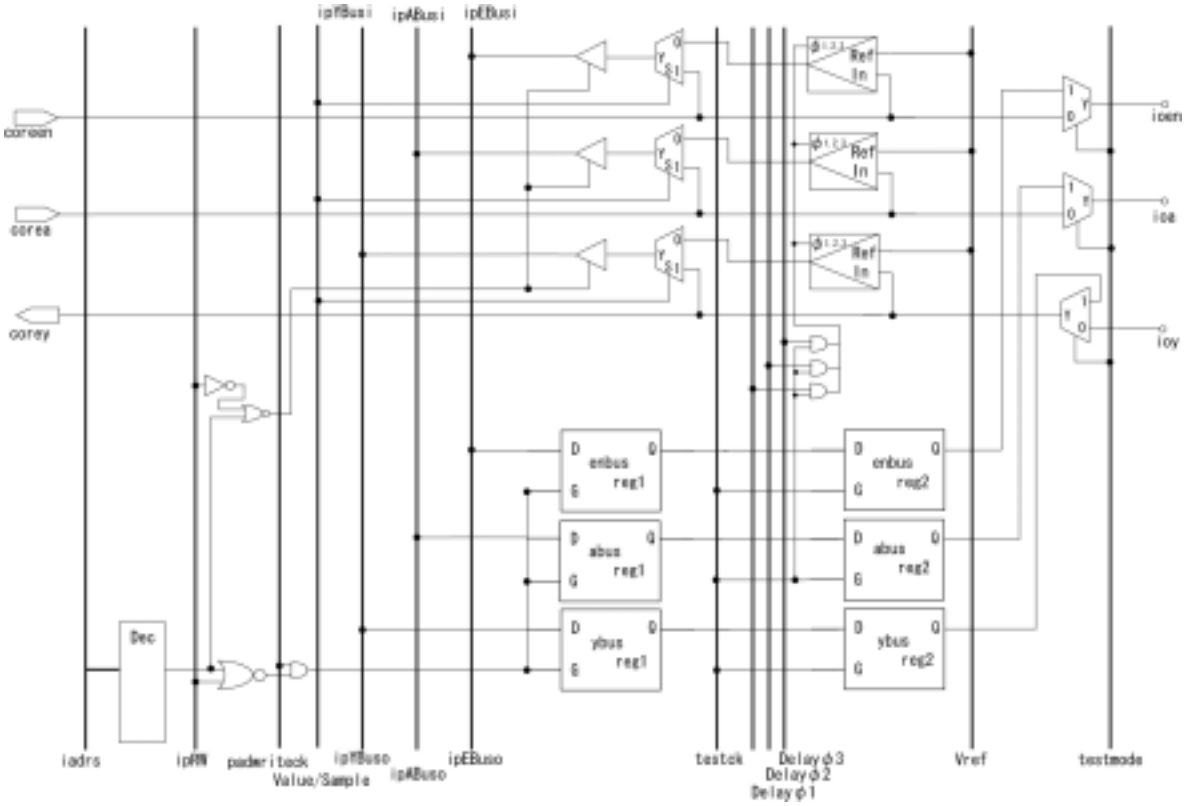


図 2.6　パラメトリックテスト実行回路も含めたバウンダリレジスタ

また、本テストシステムにおいては、パラメトリックテストを行い、信号波形の波形を取得することも考えている。そのための、パラメトリックテスト実行回路も含めたバウンダリレジスタ回路を、図 2.6 に示す。この回路では、Y,A,En の 3 つの信号線に、それぞれ 1 つずつ電圧コンパレータが接続され、信号値を読み出すかコンパレータの比較結果を読み出すかを選択可能になっている。コンパレータは目的の信号の電圧値と参照電圧値とを比較する。このとき比較対象となる参照電圧値は、チップ上に参照電圧生成回路を設け発生させ、Vref によりコンパレータに入力される。コンパレータは、コアに入力するシステムクロックの立ち上がりから指定時間後に動作する。指定時間は、チップ上に遅延時間生成回路を設け、生成する。そして、Delayφ1, φ2, φ3 により遅延時間がコンパレータに伝わり、コンパレータは指定時間後に動作する。出力が信号値なのかコンパレータの比較結果

なのかは Value/Sample の値によって指定される。

この図 2.6 のバウンダリレジスタ回路は今回実装していない。

2. 3　オンチップテストインタフェース

オンチップテストインタフェース回路の概略図を図 2.7 に示す。この回路は、テスト用回路のクロック信号やリセット信号の生成、バウンダリレジスタの制御信号の生成、データの送受信、を実行する。

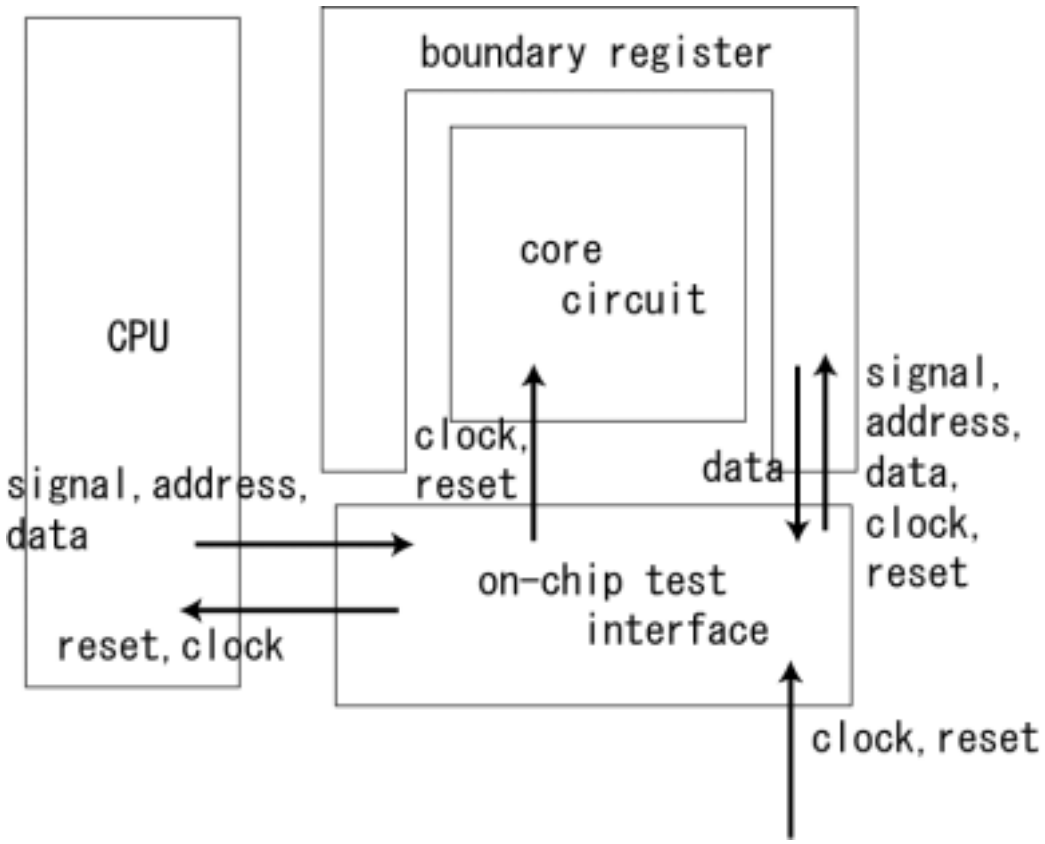


図 2.7　オンチップテストインタフェース回路の概略図

オンチップテストインタフェース回路の制御は、CPU からオンチップテストインタフェース回路のレジスタへとデータを書き込むことによって行われる。また、バウンダリレジスタからデータを読み出すときは、バウンダリレジスタから読み出されたデータはオンチップテストインタフェース回路のレジスタに書き込まれ、CPU はその値をロードすることによって行われる。CPU からレジスタへのストア、ロードはメモリマップド I/O によって行われる。オンチップテストインタフェース回路のレジスタ一覧を、表 2.3 に示す。

この回路により内部コアに入力するシステムクロックとバウンダリレジスタ用クロックが生成されるが、クロックのパラメータもレジスタにセットされるデータにより設定される。システムクロックの波形パラメータはレジスタ iCKS, iCK1, iCK0, iCKT の値によって決定され、バウンダリレジスタ用クロック波形パラメータはレジスタ iTCS, iTC1 の値によっ

て決定される。

また、レジスタ iREF と iDLY、iDCS は、パラメトリックテスト実行時のパラメータをセットするためのレジスタである。レジスタ iREF は生成する参照電圧値を決定し、レジスタ iDLY は遅延時間を決定する。iDCS は iEXE が 1 になってから遅延クロックが立ち上がるまでのクロック数を決定する。

表 2.3　オンチップテストインタフェース回路のレジスタ一覧

アドレス	bit 幅	略称	機能
0xfff0	3	iDAT	データ：0: IO 制御、1:内→外、2:外→内 書き出し:1:test モード: 外部に任意信号出力、2:test モード:コアに任意の信号を入力 読み込み:1：コアから出力されている信号、2:外部から入力されている信号
0xfff1	8	iADR	アドレス
0xfff2	2	iFLG	フラグ:iFLG[0]:RW 1:バウンダリからのデータ読み出し、0:バウンダリへのデータ書き込み、iFLG[1]:VorW 1:値、0:パラメトリック
0xfff3	8	iDLY	遅延値レジスタ
0xfff4	8	iREF	参照電圧値レジスタ
0xfff5	8	iCK1	コアクロック正時間（クロック数）
0xfff6	8	iCK0	コアクロック負時間（クロック数）
0xfff7	8	iCKS	コアクロックセットアップ時間（クロック数）
0xfff8	8	iCKT	コアクロック終了時間（クロック数）
0xfff9	8	iTCS	TestCKセットアップ時間(クロック数)
0xfffa	8	iTC1	TestCK 1持続時間（クロック数）
0xfffb	1	iEXE	実行フラグ:1にすることで、内部コア用クロック、テストクロックが生成される
0xfffc	1	iMOD	testmodeレジスタ、このレジスタを１にすることでtestmodeが1になる(コアクロックをクロック生成回路から生成、0の場合に、コアクロックは外部から供給される)
0xfffd	1	iRST	内部リセットレジスタ。１にすると内部リセット信号が生成
0xffff	8	iDCS	パラメトリックテスト用遅延クロックセットアップ時間（クロック数）
0xffe0	1	iPOL	内部クロックの極性（１：アクティブハイ、０：アクティブロー）
0xffe1	1	iBRW	バウンダリラッチに書き込み：１にすることで、バウンダリラッチ書き込みクロック生成

2．4　RS-232C インタフェース

本システムに対して、外部からホストコンピュータを通してコマンドを入力し、そのコマンドの実行結果を外部に返すために、外部とのデータの入出力を行う部分が、このRS-232Cインタフェースである。RS-232Cは調歩同期方式と呼ばれる、非同期伝送方式である。

RS-232Cによる通信の設定について、表2.4に示す。

表2.4　RS-232Cの通信設定

同期方式	調歩同期(非同期)
通信速度	9600bps
データビット	8bit
パリティビット	なし
ストップビット	1bit

この設定で通信を行うときの、通信フォーマットは図2.8に示すようになる。RS-232C では、データを送信していないときは常に1を送信する。送信するデータの前に、まず0を1bit送信する。このビットはスタートビットであり、送信開始の合図となる。スタートビットを送信したそこから、データは最下位ビットから1bitずつ8bit送信される。データビットの送信が終了したら、1を1bit送信する。このビットをストップビットといい、送信完了の合図となる。

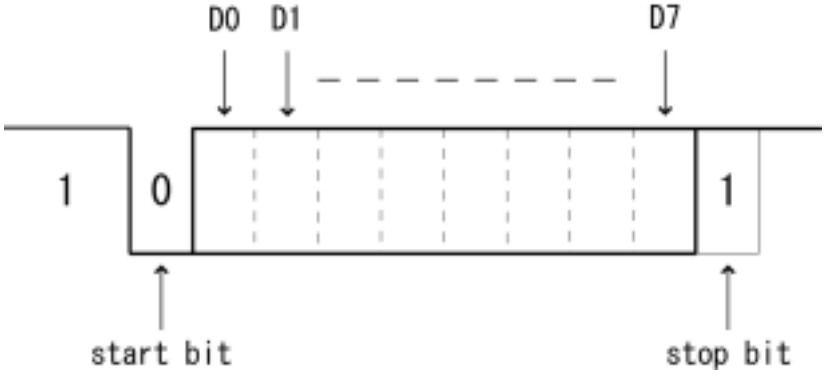


図2.8　RS-232Cの通信フォーマット

受信側では、スタートビットを受信したその時点から受信信号のタイミングをとり始める。そして、その後ビットを一定間隔で識別する。そして、ストップビットを受信するとデータの受信は終了となる。受信側では、通信速度の16倍の周波数のクロック(9.6×16=153.6kHz)を使って入力を監視し、スタートビットを検出したら、そこから8クロック待つことでスタートビットの中心点を定め、そこから16クロックごとにデータをサンプリングすることで、データを正しく受信するという方法がとられている。

送信モジュールは、送信完了前に新たなデータを与えると、送信を中断せず新たに与えられたほうのデータを破棄する。

RS-232CインタフェースとCPU はメモリマップトI/Oで接続した。アドレスの割り当てを

表2.5に示す。

送信するときは、以下の手順で行われる。まず、送信完了フラグの値を確認する。送信完了フラグの値は、アドレス0xC000の値をロードすることで得られ、送信が完了していれば1、完了していなければ0となる。この値が1であれば0になるまで待機し、0になるとアドレス0xC000へ送信データをストアする。これにより、データが送信される。

受信は、アドレス0xC001の値をロードすることによって行われる。受信したデータがないときは、0xffffという値が返されるので、そのときはデータが受信されるまで待機し、データの受信を行う。

アドレス0x002h に対して任意のデータを書き込むと、RS-232C インタフェースがリセットされる。データを送信中の場合は送信が中止され、受信したデータがあるときは破棄される。

表2.5　RS-232Cインタフェースのアドレス割り当て

アドレス	読み込み	書き込み
0xC000	送信完了フラグ	送信データ
0xC001	受信データ	―
0xC002	―	リセット

第３章

インタプリタの仕様

3. 1　インタプリタの言語仕様

作成したインタプリタの言語仕様について説明する。本システムでは、テスト用ハードウェアのコントロールをインタプリタにより行うが、その仕様は、メモリ領域の大きさなどのハードウェアの要求を満たし、さらにオンチップテストに特化したものでなければならない。

本システムは、テスト用ハードウェアを、テスト対象の回路と同一のチップ上に集積することを目標としている。一つのチップ上に集積可能なハードウェア量は上限があり、オンチップテストシステムのハードウェアオーバヘッドが大きくなると、チップ本来の機能を持つ回路の規模を制限することになってしまう。よって、インタプリタを格納するためのメモリ使用量は可能な限り小さなものとする必要がある。そのため、インタプリタはテストを実行するプログラム環境を実現するための最低限の命令と構造を有すること、を目標として言語仕様は決定された。

インタプリタの文法は、BASIC 言語を元として作成を行った[11]。BASIC 言語で用いられる文のうちシステム LSI のテストに最低限必要な文のみを採用し、さらに BASIC 言語には存在しないがテスト環境を実現するのに必要となる文を追加した。

作成したインタプリタの具体的な文法について説明する。以下に、インタプリタの言語仕様の概要を示す。文の実行方式は、１文入力されるごとに 1 文ずつ逐次実行する逐次実行と、入力された文をすぐに実行せずに記憶領域に格納しておき、後で一括して処理を行う一括実行との二つの実行方式がある。

- すべての文は一行で一文
- 文は一文ごとに逐次実行
- 行番号をつけて入力することで、プログラムを格納し、後で一括実行が可能
- 行番号だけを入力すると、その行番号の文を削除する

・ 扱える定数は、#0000~#FFFF の 16 進数と%0~%1111111111111111 の２進数
・ 変数は$a~$z の 26 個のスカラと、@[#0000]~@[#00FF]の 256 個の配列
・ 定数、変数の後に操作するビットを指定しての処理が可能
・ 演算子は+,-,の 2 個の算術演算子と、=,!,<,<=,>,>=,の 6 個の関係演算子が使用可能
・ 使用可能な文は、代入文、print 文、out 文、in 文、if 文、list 文、run 文、read 文、write 文、para 文、pset 文

各文の動作と文法について説明を行う。

代入文は、　＜変数＞ = ＜数＞　の形式で、左辺の＜変数＞へ、右辺の数値を代入する。右辺は、数値だけではなく加算、減算の数式も可能で、計算結果を左辺の＜変数＞に代入する。

print 文は、　print ＜数＞　の形式で、＜数＞の値を出力する。

out 文は、　out ＜アドレス＞ ＜データ＞　の形式である。

in 文は、　in ＜アドレス＞ ＜変数＞　の形式である。in 文と out 文の二つの文は、アドレスを指定してのチップ内のメモリへの値の書き込み、アドレスを指定してのメモリからの値の読み出しを行う文である。オンチップテストインタフェース回路の入出力はメモリマップド I/O で行われるので、この 2 文によりテストインタフェース回路のレジスタの値の設定、読み出しも可能である。out 文は、＜アドレス＞が示すアドレスのメモリに＜データ＞の値を書き込み、in 文は＜アドレス＞が示すアドレスのメモリから値を読み出して＜変数＞に格納する。

if 文は、　if ＜条件式＞ ＜行番号＞　の形式である。
＜条件式＞の形式は、　＜数１＞ ＜関係演算子＞ ＜数２＞　である。＜条件式＞が真ならば次に実行する行を＜行番号＞の行とし、偽ならば次の文を実行する。

list 文は、現在格納されているプログラムリストを、行番号順に表示する。

run 文は、格納されている文を一括して行番号順に実行を行う。

read 文、write 文、para 文、pset 文はテスト専用命令である。

read 文は、バウンダリレジスタからの値の読み出しを行い、結果を出力して表示する。出力結果は、＜ピン番号＞:＜バウンダリレジスタの値＞　である。ピン番号は、読み出しを行うバウンダリレジスタのアドレスである。＜バウンダリレジスタの値＞は、3bit の 2 進数で表示され、各ビットの意味は上位ビットから、コアへの書き込み信号の値、コアからの読み出し信号の値、コアからの制御信号の値、である。

write 文は、バウンダリレジスタへの書き込みを行う命令である。書き込まれるバウンダリレジスタのアドレスと書き込まれるデータを、配列から読み込んできてバウンダリレジスタへとセットする。書き込むデータは、数値の下位 3bit が書き込まれるデータとなる。この 3bit はの各 bit の意味は、read の出力結果と同様の意味である。

para 文、pset 文はパラメトリックテスト実行に関する命令である。パラメトリックテスト

は、内部コアを任意の内部状態に設定したときの信号波形の取得を行う。内部コアの内部状態の設定方法は、クロックを必要回数内部コアに与え、各クロックごとに内部コアの全ての入力に対応するバウンダリレジスタにユーザが設定した信号値をセットし、コアへその値を入力することによって行われる。

pset 文は、このときのバウンダリレジスタにセットされる値をクロックごとに設定する命令である。pset 文の形式は、pset ＜クロック数＞　である。＜クロック数＞は、設定を行うのがクロック何回目の値であるかを示す。書き込む値は、全てのピンにつき一つずつ配列を割り振ってあり、配列に格納されている値がこのクロックでのピンの値として、メモリに格納される。ピンの割り振りは@[＜ピンのアドレス＞]という方式である。

para 文は、para ＜クロック数＞　の形式であり、パラメトリックテストを実行する命令である。参照電圧と遅延時間を変化させながら信号波形との比較を繰り返し行う。参照電圧と遅延時間を変化させる度に pset 文で設定された値を使用して内部コアの状態設定が行われる。＜クロック数＞は、内部コアの状態設定に必要なクロックの総数である。結果は、遅延時間ごとの電圧値の形式で表され、配列に格納される。配列@[#0018]から@[#0047]にはコアへの書き込み信号、@[#0048]から@[#0077]にはコアからの読み出し信号、@[#0078]から[@#00a7]にはコアからの制御信号の結果が格納される。

図 3.1 に、インタプリタ言語の例を示す。行番号がない文は逐次実行される。行番号つきの文は記憶領域に格納されて、後で run が入力されたときに一括実行される。

```
$a = #0000
out #0010 $a
in #0011 $b
print $b
10 $c = #0000
20 @[$c] = #0000
30 $c = $c + #0001
40 if $c < #000F 10
50 $c = #0000
60 out #0010 @[$c]
70 $c = $c + #0001
80 if $c < #000F 50
run
```

図 3.1　インタプリタ言語の例

各文の具体的な動作と文法については、付録 A.1 の言語の仕様書に詳細に記載する。

3．2　インタプリタの動作

3．2．1　処理の流れ

作成したインタプリタの動作は、文の入力、字句解析、構文解析、実行に分けられ、この順で処理が行われる。

インタプリタは、ホストコンピュータから入力された文字列を入力として受信し、この文に対して解析･実行を行う。ホストコンピュータから入力された文字列は一文字ずつ受信され、記憶領域に格納される。文の入力は、リターンが入力されると終了となり、記憶領域に格納された文字列の後には区切り文字として'¥0'が格納される。そして、入力された文には、そのまま解析･実行が開始される。

文字の入出力は、2.4 節で説明した RS-232C の受信、送信手順を実行するサブルーチン getchar、putchar によりそれぞれ一文字ずつ行われる。

字句解析とは、入力された文字列を、定数や変数、演算子などの要素に分け、その各要素をトークンと呼ばれる記号に変換することである。トークンとは言語の文法上の役割を

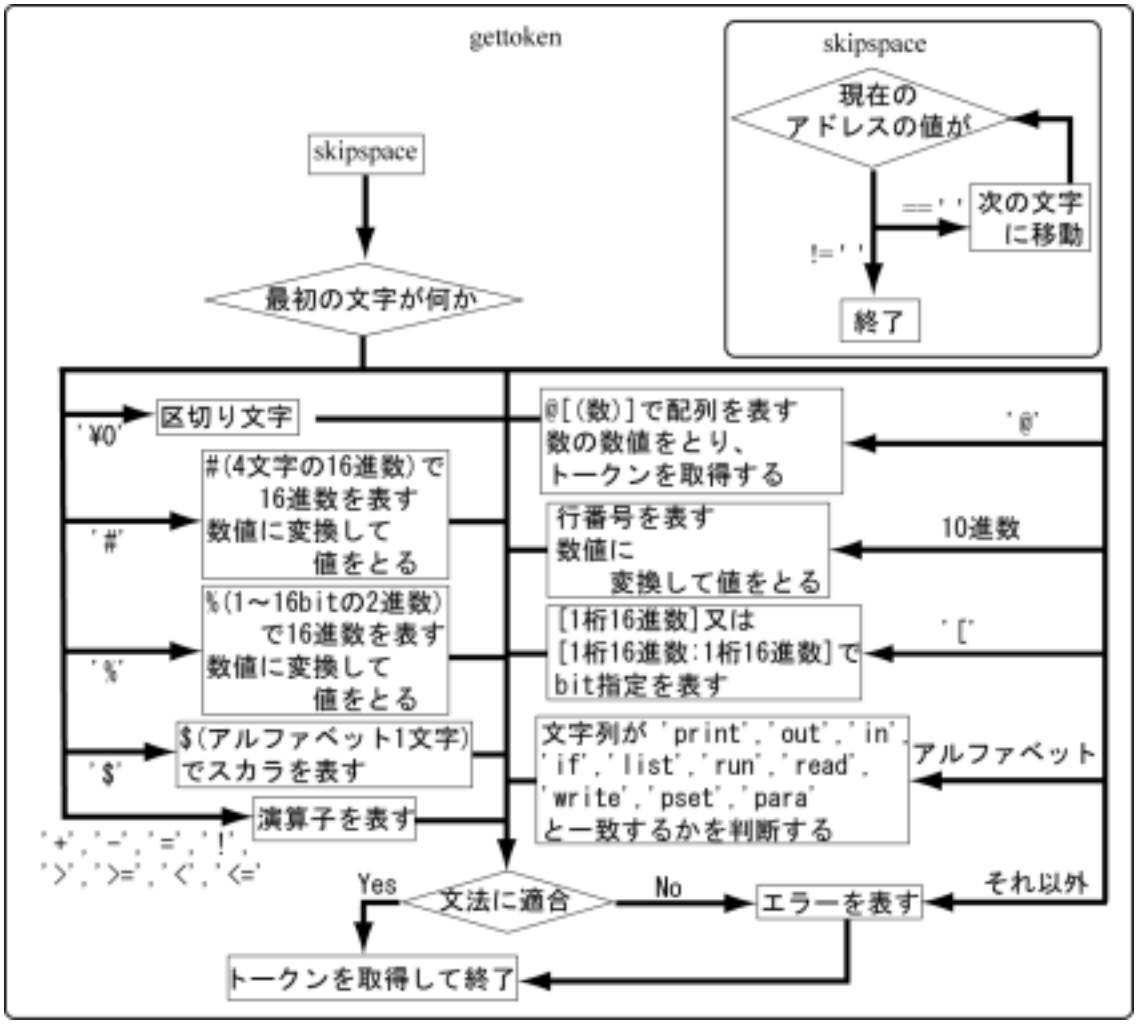


図 3.2　字句解析のフローチャート

示す記号のことである。

図 3.2 に、字句解析のフローチャートを示す。字句解析は、gettoken というサブルーチンにより行われる。まず、スペースは意味を持たないので、スペースのスキップを行う。次に、文字列の最初の文字が何かを判断する。最初の文字が何であるかによって、その文字列のトークンの種類が判断可能である。それから、その後に続く文字列の読み込みを続けて行い、文字列がトークンごとに定義された形式に適合しているかを判断する。適合している場合は、その文字列に対応した tokentype（トークンの種類）と tokendata（トークンの数値）を取得して、字句解析は終了となる。それ以外の場合は、その文字列はエラーのトークンとなる。

トークン列の並びを調べ、文がどのように構成されているかを調べるのが、構文解析である。字句解析によって得られたトークン列の構造を解析し、入力された文の種類を、最初のトークンが何であるかにより判断する。さらに、文の構造が定義された文法に一致しているものであるかを判定する。

そして、構文解析の結果に従った各文ごとの処理がなされ、入力されたプログラムの実行が行われる。

図 3.3 に、構文解析のフローチャートを示す。構文解析は、parse_commandline というサブルーチンにより行われる。構文解析では、最初のトークンが何であるかによって現在解

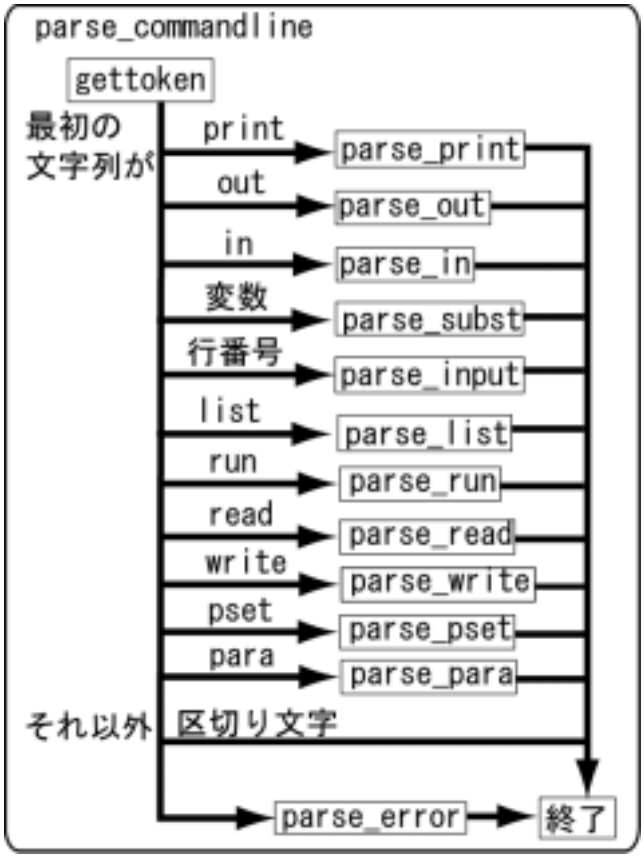


図 3.3　構文解析のフローチャート

析中の文の種類が決定される。そして、各文ごとに対応した構文解析･実行を行うサブルーチンが存在しており、そのサブルーチンで処理が行われる。

3．2．2　テスト専用命令の動作

作成したインタプリタは、テスト専用命令として read 文、write 文、pset 文、para 文を設けてある。以下に、これらの文の構文解析･実行部の動作について説明する。

read 文

read 文は、バウンダリレジスタに格納されている値を連続して読み出し、その値を標準出力へと出力を行う。その動作は読み出しを行う回数の取得、バウンダリからのデータの読み出し、読み出した値の出力、の順で行われる。バウンダリからのデータの読み出しは、オンチップテストインタフェース回路のレジスタへのデータのストア、ロードを以下の手順で行うことで行われる。

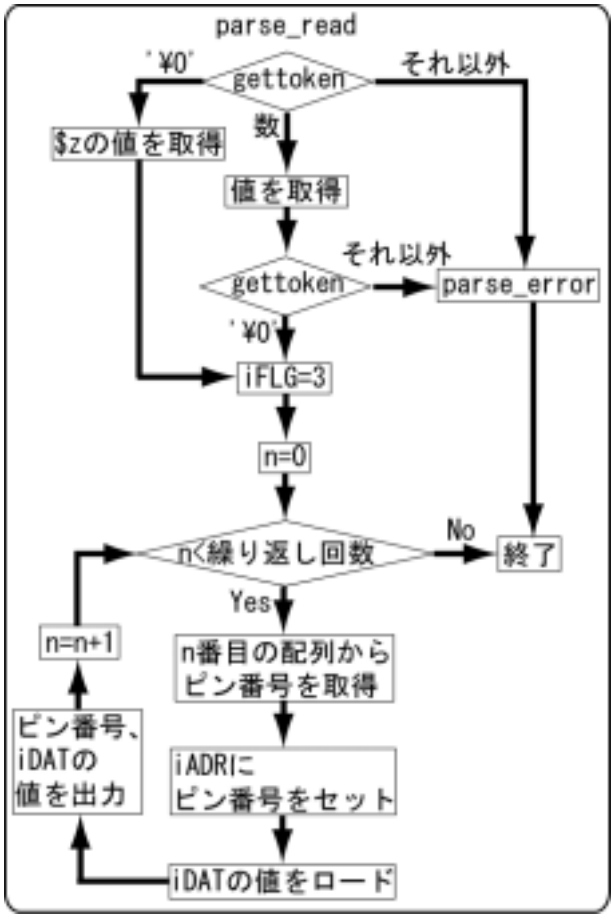


図 3.4　parse_read のフローチャート

1. iFLG=3 とすることでバウンダリへの読み出しモードに設定される。
2. iADR に読み出しを行うピン番号をセットする。
3. ピンの値が iDAT にセットされるので、iDAT の値を読み出す。
4. 2.3 を必要数繰り返す

read 文の構文解析･実行は parse_read というサブルーチンで行われる。このサブルーチンのフローチャートを図 3.4 に示す。繰り返し読み出しを行うときのピン番号は、1 回目に読み出されるピン番号は@[#0000]に、2 回目は@[#0001]、現在の繰り返し回数を n としたとき n 回目は@[(n-1)]というように、配列の先頭から読み出し回数順にあらかじめ格納しておき、その値を読み出して使用する。

write 文

wrirte 文は、バウンダリレジスタへの値の書き込みを実行する。その動作は書き込みを行う回数の取得、バウンダリへのデータの書き込み、の順で行われる。バウンダリからのデータの読み出しは、オンチップテストインタフェース回路のレジスタへのデータのストア、ロードを以下の手順で行うことで行われる。

1. iFLG=2 とすることでバウンダリへの書き込みモードに設定される。
2. iADR にデータ書き込みを行うピン番号をセットする。
3. iDAT に書き込みを行うデータをセットする。
4. iBRW=1 とすることでバウンダリへの書き込みパルスが生成される。
5. 2.3.4 を必要数繰り返す
6. iEXE=1 とすることで、バウンダリレジスタの値がコアに書き込まれる。

write 文の構文解析･実行は parse_write というサブルーチンで行われる。このサブルーチンのフローチャートを図 3.5 に示す。繰り返し書き込みを行うときのピン番号とデータは、1 回目に書き込まれるピン番号は@[#0000]でデータは@[#0080]、2 回目はピン番号が@[#0001]でデータは@[#0081]、現在の繰り返し回数を n としたとき n 回目はピン番号が@[(n-1)]でデータが@[(n+127)]というように、それぞれピン番号とデータを配列@[#0000]と@[#0080]を先頭として、書き込み回数順に格納されている値を読み出して使用する。

配列からの読み出し回数は全てのピンに書き込みを行うときに最大となり、ピン数が 128 までならば上記の配列の割り当てで全てのピンに対して書き込みを行うことができる。しかし、ピン数がそれ以上ならば、配列の数を増やし割り当てを変える必要がある。

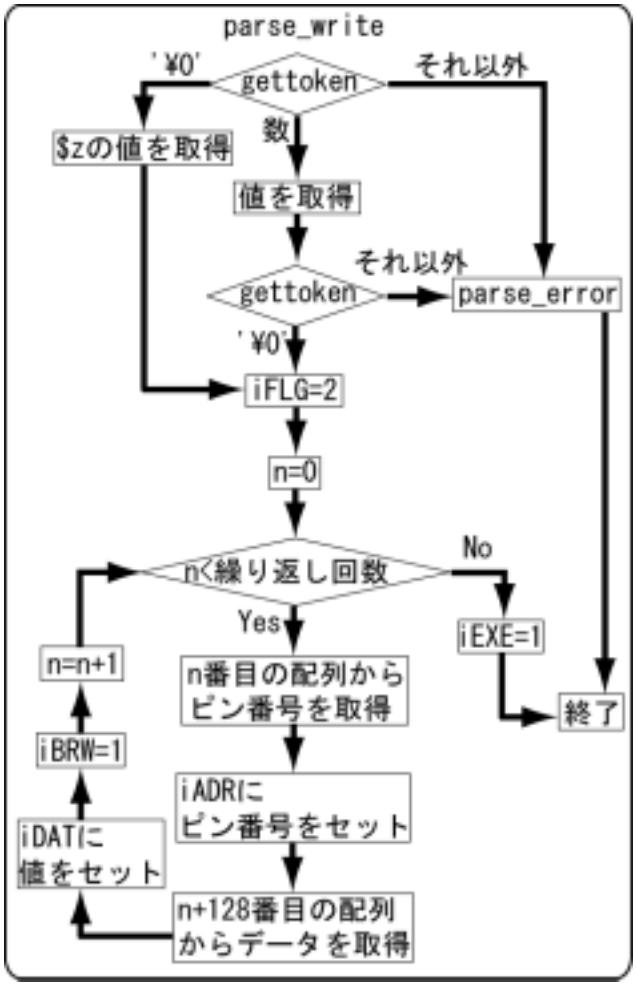


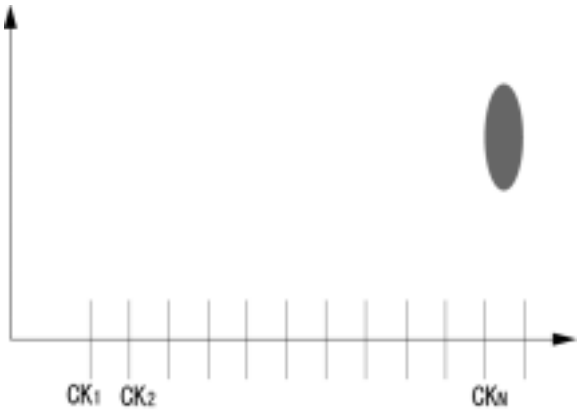
図 3.5　parse_write のフローチャート

pset 文、para 文

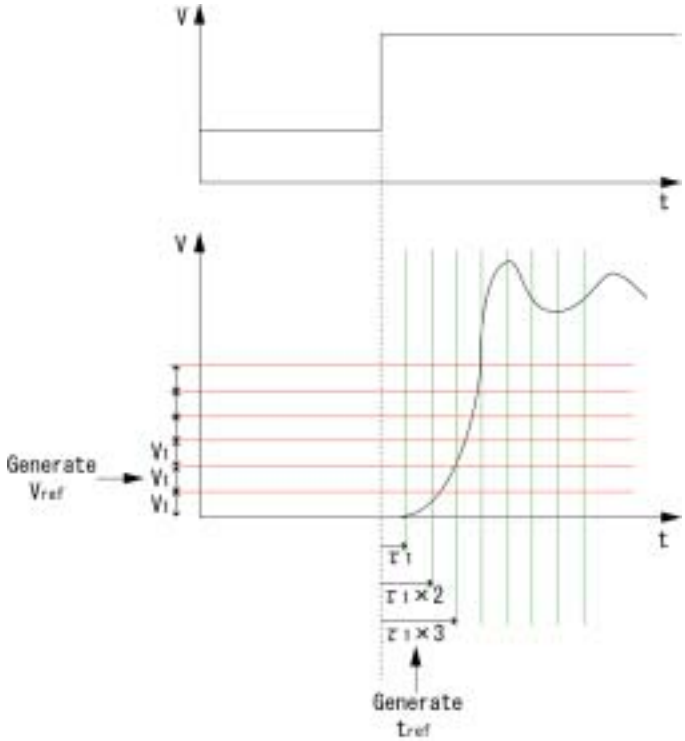
pset 文、para 文は、パラメトリックテストを行い、信号の波形を取得する。

パラメトリックテストの手順について説明する。パラメトリックテストは、図 3.6 に示す手順で波形の取得が行われる。まず、(a)観測したいイベントを発生させる。内部コアにシステムクロックをイベントが発生させるために必要な数だけ入力する。このときに、各クロックにおいて内部コアの全てのピンに対して値を書き込み、コアへの入力として与えなければいけない。このようにしてコアの内部状態を設定し、観測する信号波形を発生させる。

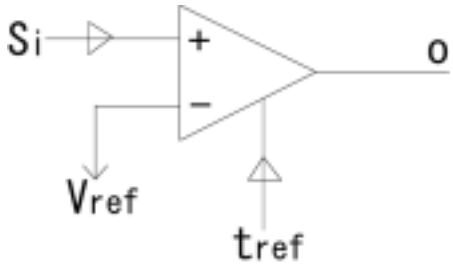
観測する信号波形に対して、比較する信号を生成するために、(b)クロックの立ち上がりからの遅延時間と参照電圧を生成する。そして、(c)クロックの立ち上がりから遅延時間後の信号波形の電圧値を参照電圧の値と比較する。比較結果は、その遅延時間における信号波形の電圧値が参照電圧値より大きければ 1､そうでなければ 0 で表される｡以上(a)~(c)を、遅延時間、参照電圧値を変化させながら繰り返し行う。その結果、(d)遅延時間、参照電圧



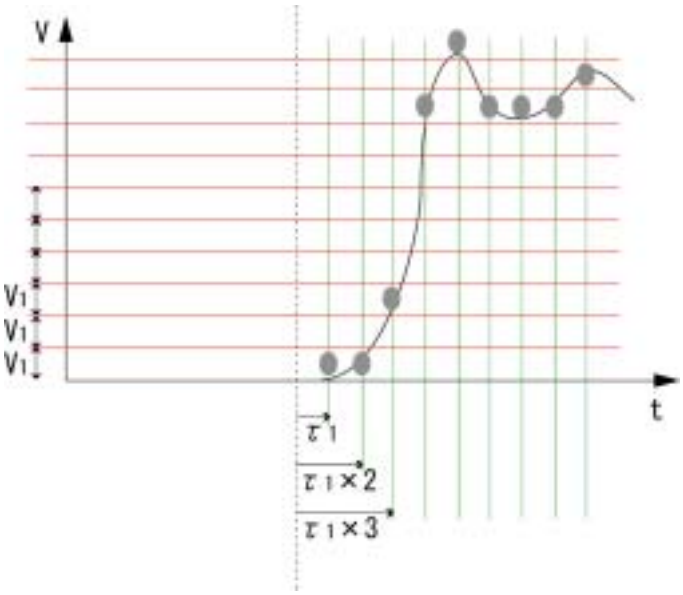
(a) 観測したいイベント発生させる



(b)遅延時間、参照電圧を生成する



(c)遅延時間後の電圧値を参照電圧と比較



(e)比較結果より波形取得

tref	Vref	o
τ1	0	1
τ1	V1	0
τ1	V1X2	0

tref	Vref	o
τ1X2	0	1
τ1X2	V1	0
τ1X2	V1X2	0

tref	Vref	o
τ1X3	0	1
τ1X3	V1	1
τ1X3	V1X2	1
τ1X3	V1X3	0
τ1X3	V1X4	0

(d)遅延時間、参照電圧ごとの比較結果

図 3.6　パラメトリックテスト実行手順

値ごとの比較結果が得られる。この比較結果をプロットすることによって、(e)比較結果より波形の取得が行われる。

pset 文は、(a)のコアの内部状態を設定する過程において、各クロックごとにコアに入力される全ピンの値を、パラメトリックテスト実行に先立って設定し、その値をメモリに格納する、という動作を行う。pset 文の動作は、クロック数の取得、ピンの値の記憶領域への書き込み、の順で行われる。

pset 文の構文解析･実行は parse_pset というサブルーチンで行われる。このサブルーチンのフローチャートを図 3.7 に示す。書き込むデータは、各ピンに一つずつ配列を割り当て、そこから値を読み出す。読み出された値は、メモリにピン数に応じた全てのピンの値を格納する一定の大きさの記憶領域を、クロックごとに確保してあり、そこにピンに書き込む値が格納される。

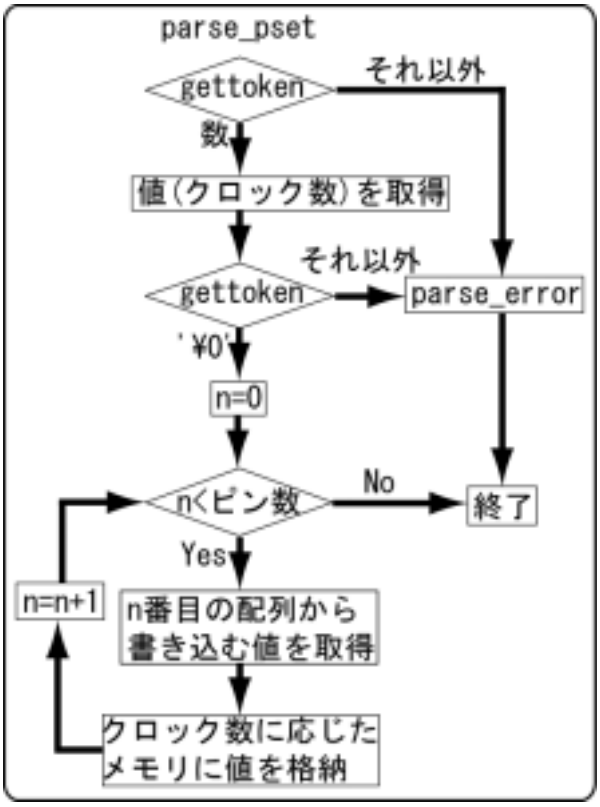


図 3.7　parse_pset のフローチャート

para 文は、上記(a)~(e)の手順を実行し、パラメトリックテストを行う文である。その動作は波形取得を実行するピン数の取得、内部状態を設定するために必要となるクロック数の取得、参照電圧と遅延時間の値を変えながら信号電圧値との比較、結果の配列への格納、の順で行われる。パラメトリックテストの実行は、オンチップテストインタフェース回路のレジスタへのデータのストア、ロードを以下の手順で行うことで行われる。コアの内部

状態の設定をについては pset 文によりセットされたクロックごとのピンの値を使用して行う。

1. iFLG=0 とすることでパラメトリックモードに設定される。
2. iADR に波形取得を行うピン番号をセットする。
3. iDAT[1]=1 とする。
4. iBRW=1 とすることでバウンダリ書き込みパルスが生成される。
5. iREF に発生させる参照電圧の値を設定する。
6. iDLY に発生させる遅延時間の値を設定する。
7. iEXE=1 とすると、delayck の立ち上がりでコンパレータが動作する。

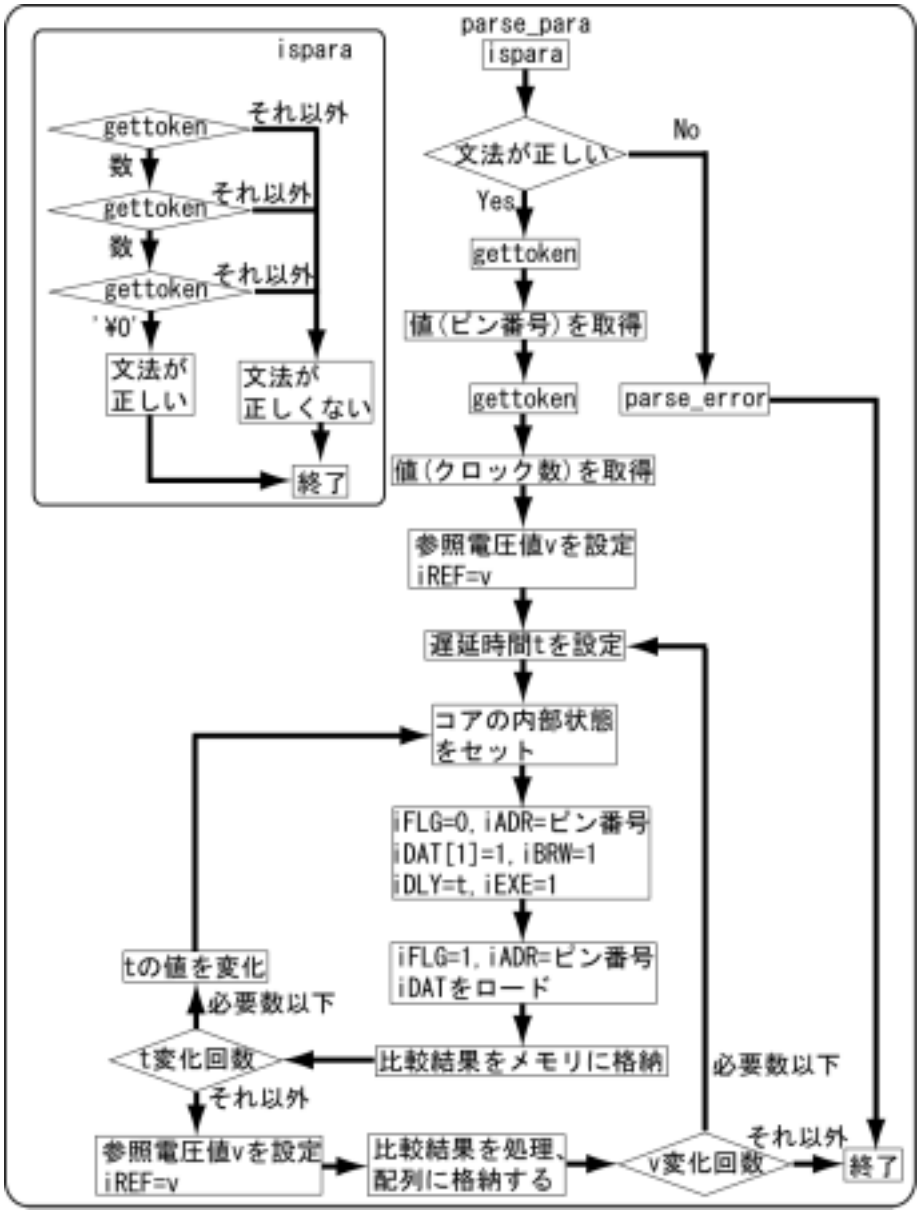


図 3.8　parse_para のフローチャート

8. iFLG=1 とする。
9. iADR に波形取得を行うピン番号をセットする。
10. iDAT に比較結果が書き込まれるので、それを読み出す。
11. 遅延時間の値を変化させ、6~10 を必要回数繰り返す
12. 参照電圧の値を変化させ、5~11 を必要回数繰り返す。

para 文の構文解析･実行は parse_para というサブルーチンで行われる。このサブルーチンのフローチャートを図 3.9 に示す。一つのピンにつき、コアへの書き込み信号、コアからの出力信号、コアからの制御信号の 3 つの信号があるが、parse_para では、その 3 つ全ての信号についてパラメトリックテストを実行する。

パラメトリックテストの実行結果の出力方法を以下に説明する。データの扱い方を、図 3.9 に示す。参照電圧値を m 点、遅延時間を n 点とるとし、参照電圧値を V_0~V_{m-1}、遅延時間を t_0~t_{n-1} とする。比較を繰り返し行い、その比較結果を処理して得られる最終的な結果は、3 つの信号それぞれについて、遅延時間ごとの電圧値の形で得られる。これを格納する配列領域を、コアへの書き込み信号、コアからの読み出し信号、コアからの制御信号のそれぞれを、配列 1、配列 2、配列 3 とする。配列領域の長さは 3 つとも遅延時間の変化回数 n と等しい。

比較結果は、一度比較を行うごとにこの 3 つの信号が 3bit の数として得られる。一度の比較結果は 3bit の値であるが、3 つの信号それぞれの波形を取得するので、3bit を bit ごとに別々に処理した方が処理は簡単になる。比較は、遅延時間と電圧値を変化させながら複数回行うので、比較回数×3 個の比較結果が最終的に得られ、一つの信号につき(a)のように参照電圧、遅延時間ごとの比較結果が得られる。この全ての結果を配列に出力するとする。このとき、必要な配列の個数は、

$$n \times m + 3 \times n$$

となる。n=128、m=128 とすると、16768 個となってしまい、非常に大きい値となってしまう。

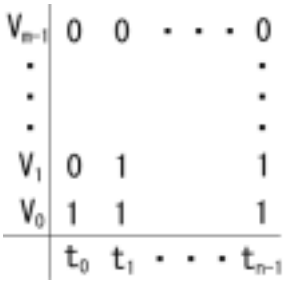
そこで、出力結果を圧縮する。すなわち、1 つの信号につき、1 度の比較で 1bit の比較結果が得られるのだが、これをそのまま 1bit ずつ格納するのではなく、配列は 16bit であるので、比較 16 回の結果を配列 1 つに格納する。para 文では、遅延時間を変化させるループを参照電圧を変化させるループの内側にしている。これは、値を設定し終えるまでの時間が参照電圧生成回路の方が遅延時間生成回路よりも遅いからである。すなわち、データはある電圧値ごとの出力結果がまとまって出力されるので、(b)のようになる。このとき必要な配列数は、

$$3 \times n \times m/16 + 3 \times n$$

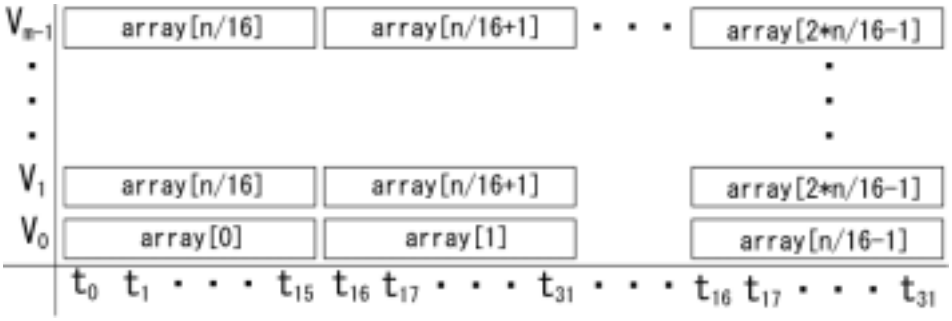
となり、n=128、m=128 のとき、3456 個と減少するが、まだかなり大きい。

ここで、ある遅延時間における波形の電圧値の算出法について考える。参照電圧値が信号の電圧値より小さければ出力結果は 1 となり、大きければ出力結果は 0 となる。よって、

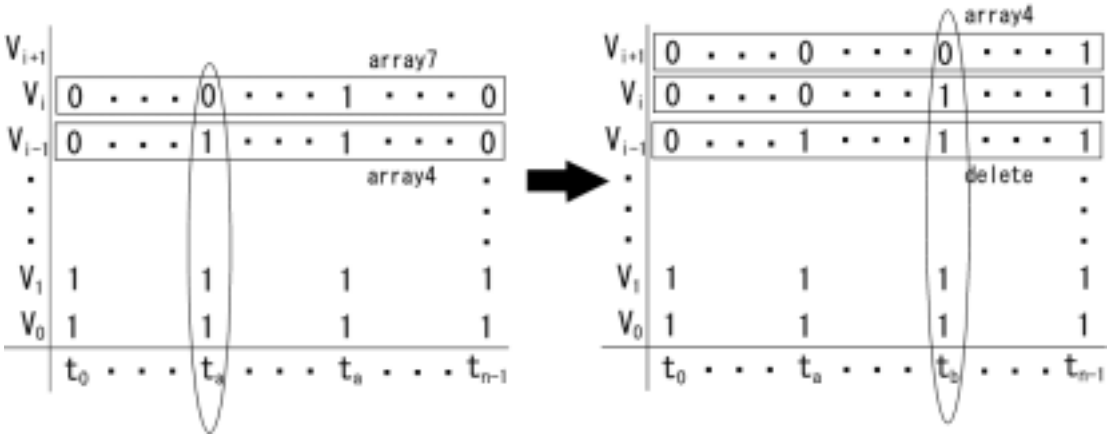
比較結果が 1 から 0 に変化するときの参照電圧の値がその遅延時間での電圧値となる。つまり、比較結果のうち、参照電圧値が一つ大きくなったときに、比較結果が 1 から 0 に変わった遅延時間を検出すればよい。このときデータの処理に必要な比較結果は、連続する参照電圧 2 つ分である。配列 4、5、6 にはそれぞれ 3 つの信号のある参照電圧値の比較結果を格納し、次の参照電圧値の比較結果はそれぞれ配列 7、8、9 に格納するとする。(c)にはコアへの書き込み信号の波形の比較結果を示している。ある点の参照電圧の比較結果が全て得られたら、それらを配列 4 に格納し、次の比較結果を配列 7 に格納する。配列 4 と配列 7 の値を比較し、1 から 0 に変化した遅延時間を検出する。この場合、t_a のときの電圧値が v_{i-1} となり、配列 1 内の t_a に割り振ってある配列に v_{i-1} の値を格納する。この結果の比



(a)一つの信号についての比較結果



(b)一つの信号の比較結果 16 回分を一つの配列に格納



(c)使用するデータは参照電圧二つ分

図 3.9　比較結果の処理

較が終われば、古い参照電圧の値が不要になるので、次の参照電圧値の結果をそこに格納する。すなわち、配列 4 の値は不要となり、次の参照電圧の比較結果は配列 4 に上書きして格納される。この方法であれば、必要な配列数は、

$$3 \times 2 \times n/16 + 3 \times n$$

となり、m の値にはよらず、n=128 のとき、432 で済む。para の出力は、この方法で行うことにした。

n の値は、今回の配列の数は 256 個であるので、それに合わせて n=64 とした。n の数を増やしたければ、配列の個数を増やす必要がある。各配列領域の割り当ては、配列 1、すなわちコアへの書き込み信号が格納されるのが@[#0018]~@[#0057]、配列 2、すなわちコアからの読み出し信号が格納されるのが@[#0058]~@[#0097]、配列 3、すなわちコアからの制御信号が格納されるのが@[#0098]~@[#00d7]、配列 4 が@[#0000] ~@[#0003]、配列 5 が@[#0004] ~@[#0007]、配列 6 が@[#0008] ~@[#000b]、配列 7 が@[#000c] ~@[#000f]、配列 8 が@[#0010] ~@[#0013]、配列 9 が@[#0014] ~@[#0017]である。

3．3　インタプリタの設計

以上までで仕様と動作を説明したインタプリタの設計を行った。

まず C 言語によりインタプリタの設計を行った。これにより、インタプリタの仕様を決定し、動作確認を行い正常に動作することを確かめた。この C 言語により作成されたインタプリタの行数は 1544 行となった。

次に、C 言語により記述したインタプリタを、アセンブラ言語に変換した。使用したアセンブラは、フリーソフトとして公開されている汎用マクロアセンブラ AASM[12]である。表 2.2 に示した CPU の命令セットを定義したマクロを用いてインタプリタの記述を行った。なお、拡張命令は使用しなかった。図 3.10 に、アセンブラ言語による記述の例を示す。これは、out 文の構文解析･実行を行うサブルーチンである parse_out の記述の一部である。アセンブラにより作成したインタプリタのワード数は、3981 となった。

表 3.1 にレジスタの機能の割り当てを示す。この表に記載されていないレジスタは、特定の役割を与えず、各サブルーチンで自由に使用している。より多くのレジスタが使用したい場合に、レジスタ r1、r10 の値をスタックに保存して一時的に使用する場合もあった。

また、表 3.2 に作成した関数一覧を示す。

```
parse_out:  st r3,r14
            add r14,r12,r14
            st  r4,r14
            add r14,r12,r14
            st r5,r14
            add r14,r12,r14
            st r6,r14
            add r14,r12,r14
            st r7,r14
            add r14,r12,r14
            st r15,r14
            add r14,r12,r14
            set gettoken&0xff,r5
            sethi gettoken>>8,r5
            js r5,r15
            set issuu&0xff,r5
            sethi issuu>>8,r5
            js r5,r15
            subf r11,r5,r13
            brnz 7
```

図 3.10　アセンブラ言語による記述の例

表 3.1　レジスタの割り当て

レジスタ	機能
r1	解析中の文字列のアドレスを指すポインタ
r8	RS-232C 出力ポートのアドレス
r9	RS-232C 入力ポートのアドレス
r10	文格納時の格納リストのポインタ
r11	常に 0
r12	常に 1
r13	一時的なデータのためのレジスタ
r14	スタックポインタ
r15	サブルーチンからリターンするときの戻り値

表 3.2　作成した関数一覧

関数名	機能
getchar	RS-232C から 1 文字受信
putchar	RS-232C に 1 文字送信
puts	RS-232C に 1 文字送信
tolower	大文字を小文字に変換
isalpha	アルファベットかどうかを判定
isidigit	10 進数かどうかを判定
isxdigit	16 進数かどうかを判定
ishensuu	変数(スカラ、配列)かどうかを判定
issuu	数(16 進数、2 進数、変数)かどうかを判定
strcmp	文字列比較
h2i	16 進数文字 1 文字を数値に変換
h2i	16 進数文字文字列を数値に変換
i2h	数値を 1 桁の 16 進数文字に変換
i2h	数値を 4 桁の 16 進数文字列に変換
skipspace	字句解析時にスペースをスキップする
gettoken	字句解析を実行する
parse_command;ine	構文解析を実行する
parse_factor	数のトークンの数値を取得する
parse_bit	bit 指定トークンから数の数値を bit 指定
parse_print	print 文の構文解析･実行を行う
parse_out	out 文の構文解析･実行を行う
parse_in	in 文の構文解析･実行を行う
parse_subst	代入文の構文解析･実行を行う
parse_expr	代入文の右辺の構文解析･実行を行う
issubst	代入文の文法が正しいかどうかを判定
parse_input	行番号つきの文の構文解析･実行を行う
parse_store	行番号つきの文の格納を行う
parse_sort	行番号･文の先頭アドレスリストのソーティングを行う
parse_del	格納されている文の消去を行う
parse_gc	文の格納領域のガーベージコレクションを実行する
parse_list	list 文の構文解析･実行を行う
parse_run	run 文の構文解析･実行を行う
parse	一括実行時に構文解析を実行する
parse_if	if 文の構文解析･実行を行う
parse_jump	if 文の条件分岐でのジャンプ操作を行う
isif	if 文の文法が正しいかどうかを判定
parse_read	read 文の構文解析･実行を行う
parse_write	write 文の構文解析･実行を行う
parse_pset	pset 文の構文解析･実行を行う
parse_para	para 文の構文解析･実行を行う
ispara	para 文の構文解析･実行を行う
parse_error	エラー処理を行う

3．4　シミュレーション

アセンブラ言語により記述されたインタプリタのシミュレーションを行った。アセンブラ言語により記述されたインタプリタを機械語に変換し、当研究室により製作されたエミュレータを用いてパソコン上でシミュレーションを行った。このエミュレータの図を、図 3.11 に示す。

エミュレータは、RS-232C によるデータ通信および CPU とメモリの動作のエミュレートを行うことができる。エミュレータの入力は機械語であり、エミュレータ上でデバッグコマンドを入力することでエミュレータは動作し、インタプリタのシミュレーションを行うことができる。

このエミュレータを用いてインタプリタのシミュレーションを行った。シミュレーションは、インタプリタに適当な長さのテストプログラムを入力として与え正常な動作をするかどうかを確認するという方法で行い、作成したインタプリタが設計通りの動作をするか確認した。

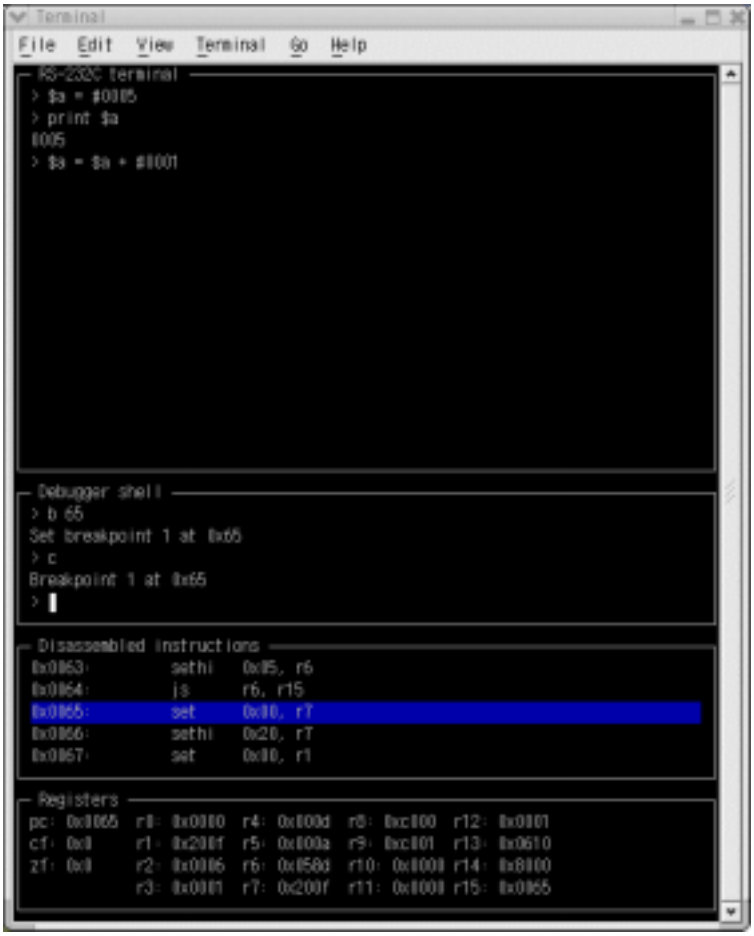


図 3.11　エミュレータ

まず逐次実行動作、一括実行動作の動作確認を行うために、１０行程度の簡単なテストプログラムをそれぞれ用意した。その結果、どちらの動作も正常に実行されることが確認できた。

次に、５０行程度の長めの行数をもつ、自然対数の底 e の値を計算するプログラムを記述し、これによりインタプリタの入力として与えシミュレーションを行った。このテストプログラムは、if 文による複数の条件分岐とループ構造を含む、やや複雑な構造となっている。実行結果は、図 3.12 に示す通り、小数点以下 20 桁まで正確な値を表示した。以上のことから、このテストプログラムに対しても正確な動作が確認できた。また、プログラムカウンタレジスタの値が正確に変化すること、記憶領域への値の書き込み、読み出しが正常に行われていることも確認した。以上のことから、作成したインタプリタの動作が正常であると、確認できた。

エミュレータは、バウンダリ回路やオンチップテストインタフェースなどのハードウェアを含まないので、テスト専用命令についてはパソコン上で動作を確認することはできない。

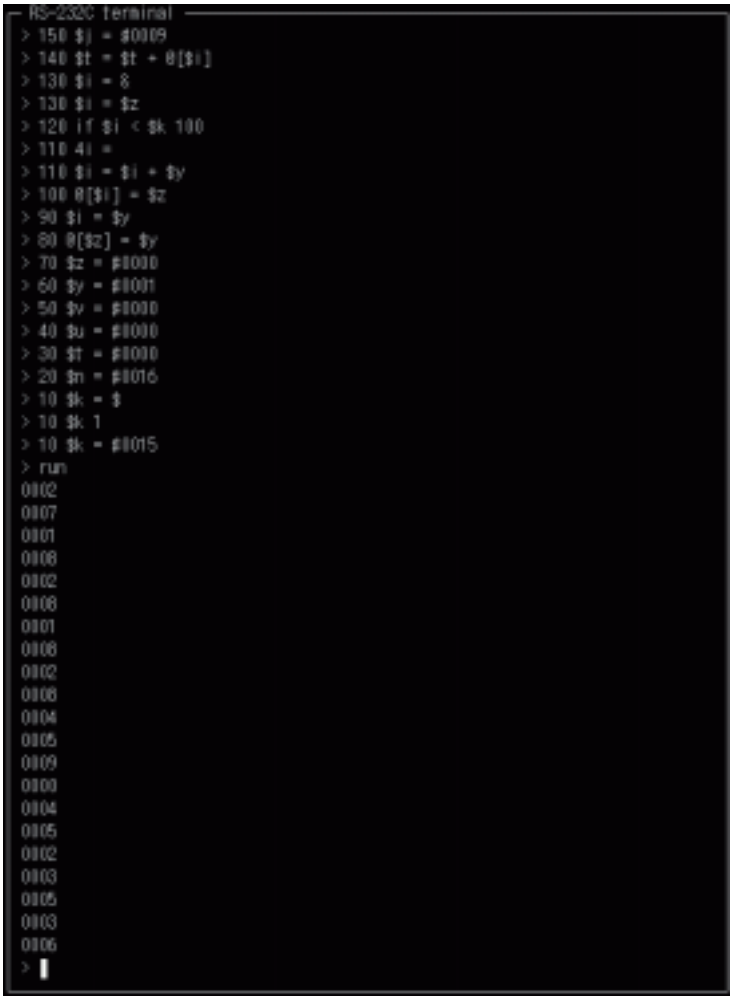


図 3.12　インタプリタ上での e を計算するプログラムの実行結果

第４章

FPGA による実装

4. 1　実装環境

第 2 章で説明した CPU、ROM/RAM 等全てのハードウェアを、FPGA に実装した。用いた FPGA は三菱電機マイコン機器ソフトウエアの PowerMedusa MU200-SX であり、回路を FPGA にダウンロードするために使用したツールは日本アルテラの Quartus Ⅱである。実装環境を図 4.1 に示す。

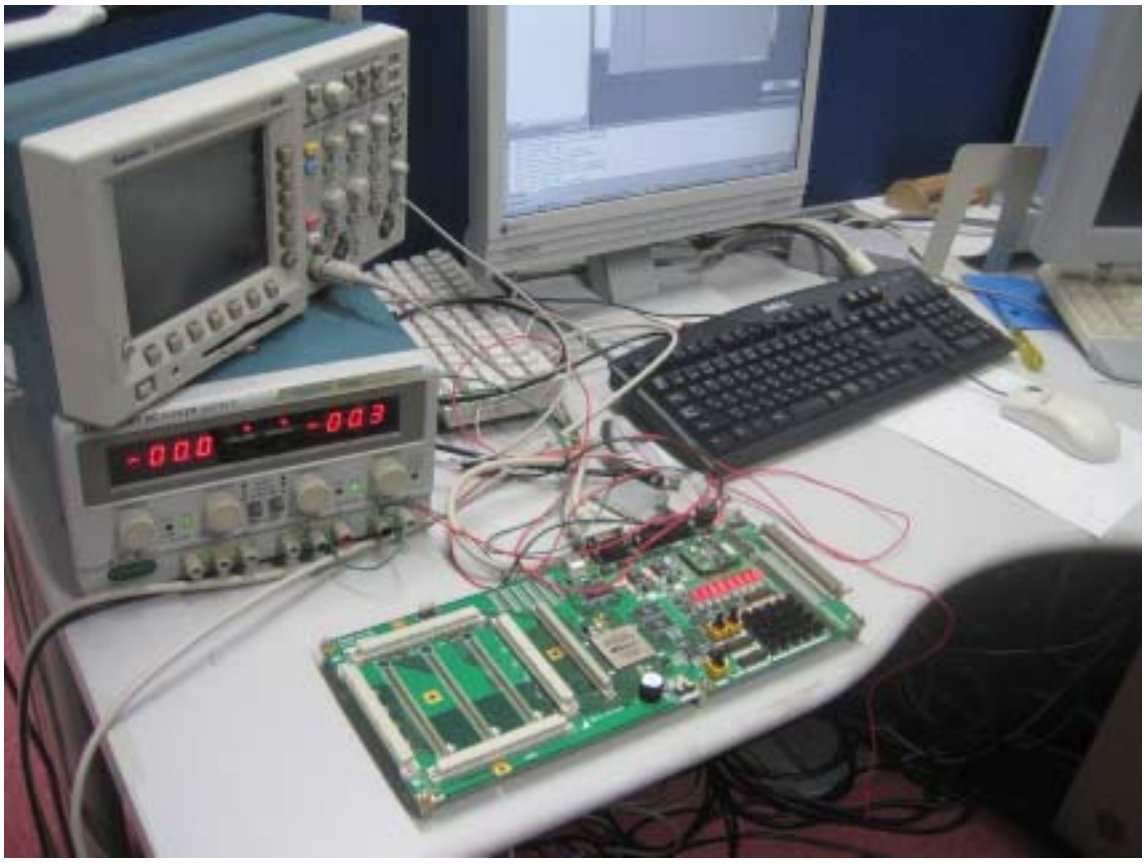


図 4.1 実装環境

第 3 章で説明したインタプリタは、アセンブラ言語で記述したものを機械語へと変換し、機械語を mif ファイルへと変換したものを使用した。この mif ファイルは、メモリ初期化ファイル(Memory Initialization File)であり、FPGA にダウンロードするメモリの内容を設定するファイルである。機械語から mif ファイルへの変換を行うプログラムは、当研究室で C

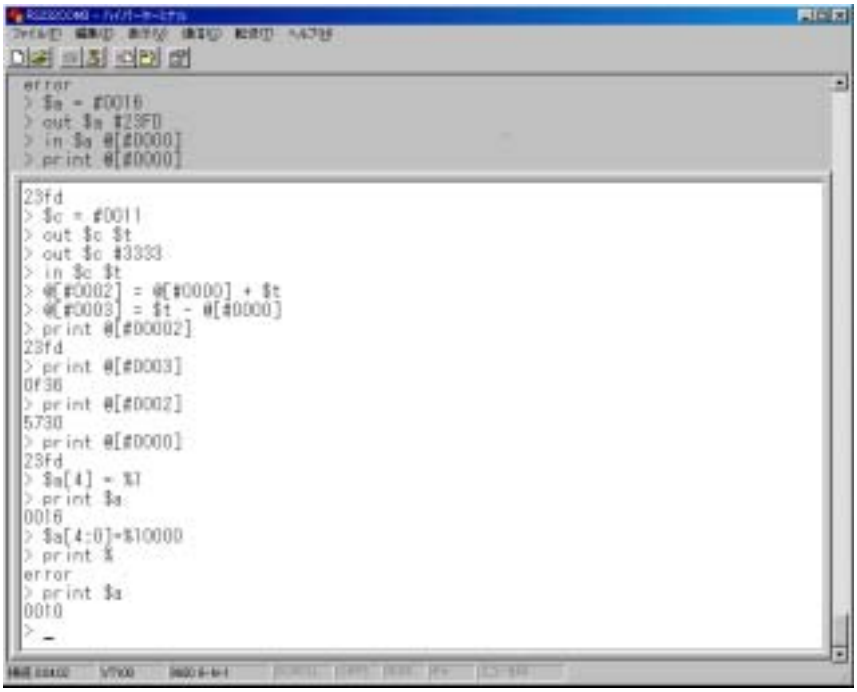


図 4.2　ハイパーターミナル上での FPGA に実装したインタプリタの動作の様子

言語により作成されたものを使用して行った。mif ファイルのインタプリタを ROM の中身として FPGA 上に実装した。

使用するクロックは、FPGA 搭載のクロックジェネレータ 80MHz を使用する。RS-232C 通信用のクロックも、このクロックを分周して使用する。

FPGA ボードは RS-232C によりホストとなるコンピュータと接続した。ホストコンピュータの OS は Windows XP であり、RS-232C との通信を行うソフトウェアは、ハイパーターミナルである。

この実装環境でインタプリタを動作させた。ホストコンピュータのハイパーターミナル上での、FPGA に実装されたインタプリタの動作の様子を、図 4.2 に示す。機械語の段階でエミュレータにより動作確認を行った、自然対数の底 e の値を計算するテストプログラムを入力として与えたところ、図 4.3 のように小数点以下 20 桁まで正確に表示され、実装したインタプリタが正常に動作することが確認できた。

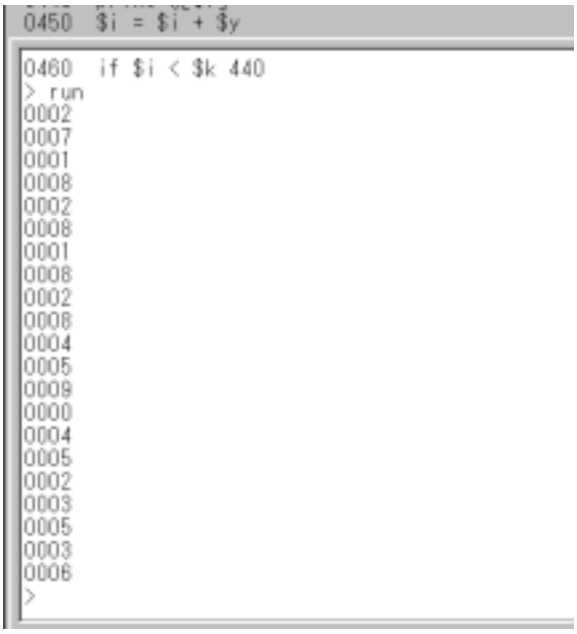


図 4.3　e の値をプログラムによるインタプリタの動作確認

実装した回路のハードウェア規模を表 4.1 に示す。この結果から、チップ上にこれらのハードウェアを集積した場合のトータルのゲート数は数万程度になり、全体のハードウェア量からすると許容範囲であるといえる。メモリはデータメモリが 65536、プログラムメモリ 8192 とした。

表 4.1　回路のハードウェア規模

ハードウェア	LE 数
CPU	669
バウンダリレジスタ(内部コア含む)	30
オンチップテストインタフェース	316
RS-232C I/O	131

4. 2　read 文、write 文の動作

実装したハードウェアには、内部コアに相当する、テスト対象となる回路として、4bit のカウンタが含まれている。図 4.4 には、このカウンタの回路図を示す。5bit 目の入力がプリセット信号となっている。acorey0~corey4 はバウンダリレジスタから入力される信号であり、corea0~corea3 はバウンダリレジスタへと出力される信号である。このカウンタに接続されるバウンダリレジスタの個数は、入力 bit 数と同じ 5 個である。ここに記されていないバウンダリレジスタに出力されるピンの制御信号と、入力信号 corey4 に対応するコアからの出力信号については、全て 1 として扱うようになっている。ピンのアドレスは、下位 bit から 0x0000~0x0004 になっており、オンチップテストインタフェース回路の iADR レジスタ(0xfff1)にこのアドレスの値をストアすることで、ピンのアドレスは指定される。

テスト専用命令である read 文、write 文について、この回路、及びこれに接続されている

バウンダリレジスタ回路を正しく動作させられるかを見ることで、この 2 つのテスト専用命令の動作を確かめる。

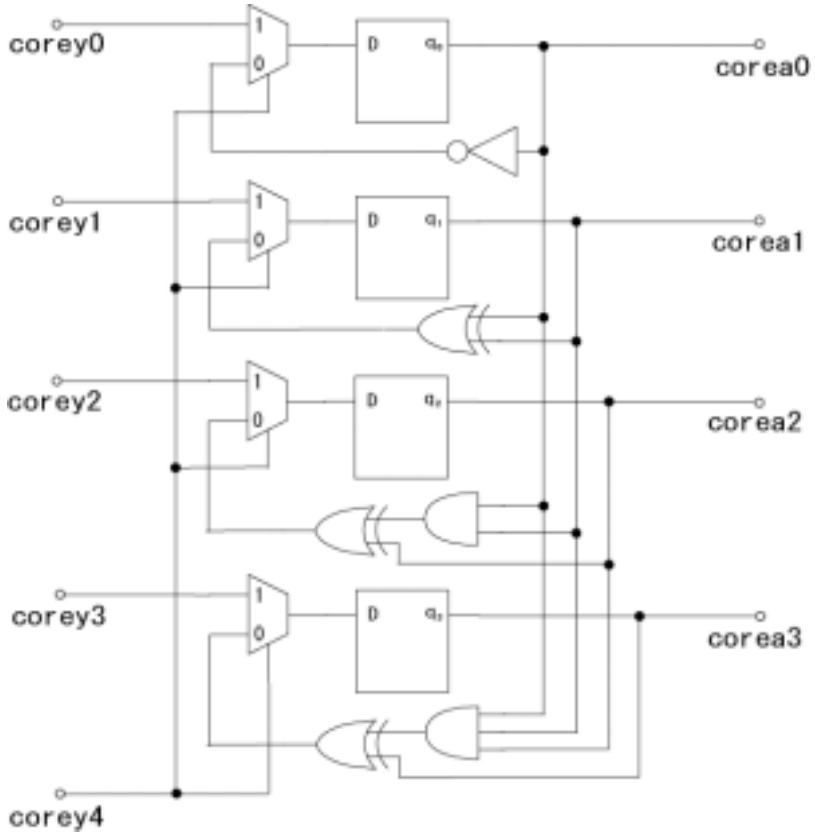


図 4.4　テスト対象となるカウンタ

read 文

read 文は、バウンダリレジスタからの値の読み出しを、任意の回数連続して実行する命令である。図 4.5 には、read 文を実行した結果を示す。

read 文を実行するときは、読み出しを行う回数と、そのとき読み出すピンのアドレスを最初に指定する。読み出し回数は、read 文の引数としても指定できるが、$z に値をセットすることでもできる。今回の回路ではピンの数は 5 なので、$z に#0005 をセットする。そしてこのとき、読み出すピンのアドレスは順に配列@[#0000]~@[#0004]の値が使用されるので、ピンのアドレスを配列に格納する。

値のセットを行い、ピンの値の読み出しを一度だけ行った結果が図 4.5(a)である。出力形式は 3 章で述べたように、＜ピン番号＞:＜バウンダリレジスタの値＞であり、＜バウンダリレジスタの値＞は 3bit の 2 進数で表示され、上位ビットからそれぞれコアへの書き込み信号の値、コアからの読み出し信号の値、コアからの制御信号の値である。アドレス 0x0001 のピンからの出力値が 1 で、それ以外のピンの出力値は 0 となっている。

次に、内部コアにシステムクロックを与え、カウンタを進めたときのピンの値を読み出

```
error
> print $a
0010
>
```

```
k
error
> 10 $i = ¥0000
> 20 @[$i] = $i
> 30 @[$i] =
> 30 $i = $i + ¥0001
> 40 if $i < #0005 20
> list
0010  $i = #0000
0020  @[$i] = $i
0030  $i = $i + #0001
0040  if $i < #0005 20
> run
> $z = #0005
> read
0000:000
0001:010
0002:000
0003:000
0004:000
>
```

(a)読み出し回数とピンのアドレスを指定し read を実行

```
0003:000
0004:000
> 10 out #fffb #0001
> 20 read
```

```
> 30
> 40
> list
0010  out #fffb #0001
0020  read
> run
0000:010
0001:010
0002:000
0003:000
0004:000
> run
0000:000
0001:000
0002:010
0003:000
0004:000
> run
0000:010
0001:000
0002:010
0003:000
0004:000
> _
```

(b)コアにクロックを与えながら繰り返し読み出し

図 4.5　read 文の実行結果

す、という操作を連続して行った。その結果が図 4.5(b)である。コアに与えられるシステムクロックは、テストインタフェース回路の iEXE レジスタに 1 をセットすると生成される。つまり、out #fffb #0001 としてレジスタに値をセットすると、システムクロックが 1 クロックコアに入力される。コアからの出力信号値を見ると、システムクロックを 1 つ進めるごとにカウンタの値が 1 ずつ増えていることがわかり、この回路がカウンタとして正しく動作していることがわかる。

これにより、バウンダリレジスタの値を読み出してコアの動作を見ることが、read 文によって行われているといえる。

write 文

write 文は、バウンダリレジスタへの値の書き込みを、任意の回数連続して実行する命令である。図 4.6 には、write 文を実行した結果を示す。

write 文を実行する際には、書き込みを行う回数と、書き込みを行うピンのアドレス、書き込みを行うデータ、をセットしなければいけない。書き込み回数とピンのアドレスについては、read 文での設定がそのまま使用可能であるので、ここではピンに書き込むデータの設定のみを行う。ピンに書き込むデータは配列@[#0080]以降にセットする。書き込み時には、1 回目に書き込まれるピン番号は@[#0000]でデータは@[#0080]、2 回目はピン番号が@[#0001]でデータは@[#0081]となるので、ピンのアドレス 0x0000~0x0004 に対応するアドレスが@[#0000]~@[#0004]であり、データが@[#0080]~@[#0084]となる。書き込むデータの形式は、read 文の出力結果と同様である。ここでは、アドレス 0x0001 と 0x0003 のピンの書き込み信号の値を 1 とし、それ以外のピンには 0 を書き込む事にした。

値を設定し、write を実行した結果が図 4.6(a)である。バウンダリレジスタに書き込まれた値を read 文によって見ると、アドレス 0x0001 と 0x0003 のピンの書き込み信号が 1 になっていることがわかる。

今回のコア回路は、プリセット信号を 1 にしないと値がセットされない。そのため、このままではコアには値が伝わらない。そこで、write 文によりアドレス 0x0004 のピンの書き込み信号に 1 を書き込んだ結果が図 4.6(b)である。コアからの出力信号に、コアに書き込んでいる信号の値が現れており、書き込んだ値が内部コアに伝わっていることがわかる。さらに、全ての書き込み信号の値を 0 にしたところ、コアのカウンタは書き込まれた値から動作を開始した。

このことから、write 文によりバウンダリレジスタに値を書き込み、その信号を内部コアへの入力とすることが可能である。

以上のように、テスト専用命令 read 文と write 文の動作確認を行った。この 2 つの命令によりコアからの値の読み出し、コアへの値の書き込みが、インタプリタ上で可能であることを確かめた。

```
0001:000
0002:010000
0003:010
0004:00000

0002:000
0003:000
0004:000
> 20
> list
0010  out #fffb #0001
0030  read
> @[#0080] = %0
> @[#0081] = %100
> p@
error
> @[#0082] = %0
> @[#0083] = %100
> @[#00840 = %0
error
> @[#0084] = %0
> write
> read
0000:010
0001:100
0002:000
0003:100
0004:000
>
```

(a)ピンに書き込む値を設定し書き込む

```
0004:000
> 20fal:
> list
0010  out ¥fffb #00010

0003:100
0004:000
> @[#0084] = %100
> write
> read
0000:000
0001:110
0002:000
0003:110
0004:110
> list
0010  out ¥fffb #0001
0030  read
> @[#0081] =%0
> @[#0083] = %0
> @[#0084] = %0
> write
> read
0000:010
0001:010
0002:000
0003:010
0004:000
>
```

(b)書き込んだデータが内部コアに伝わる。

図 4.6　write 文の実行結果

4. 3　パラメトリックテスト実行用回路

パラメトリックテストを行うための回路は、アナログ回路が必要となるため、FPGA では実装ができない。そのため、今回のハードウェアでパラメトリックテストを実行するためには、FPGA と外部のアナログ回路とを接続して行わなければいけない。

パラメトリックテスト実行用回路の概略を、図 4.7 に示す。レジスタ iREF の値を入力として D/A コンバータにより参照電圧を生成し、入力された信号との電圧比較を行う。遅延時間は、レジスタ iDLY の値を入力として遅延時間生成回路により作成する。入力されたクロックの遅延時間後に遅延クロックが出力される。FPGA からの出力信号が入力となる。出力は、電圧比較結果と遅延クロックを FPGA へと入力し、遅延クロックの立ち上がりでの電圧比較結果の値を取り込む。これにより、参照電圧と遅延時間を変えての電圧比較が実行可能となる。

この回路のうち、参照電圧生成回路と電圧比較回路を外部のユニバーサル基板上で IC を接続して、作成した。作成した回路の回路図を図 4.8 に示す。使用したコンパレータは NJM319 であり、D/A コンバータの出力と、対象となる FPGA から入力される信号を比較する。電源電圧は+5V であり、単電源動作である。応答時間は 80ns である。コンパレータの出力は、そのまま FPGA へと入力されるので、出力電圧は V_{OH} が 3.3V になるように調節した。使用した D/A コンバータは AD558 であり、電源電圧 15V ならば、出力範囲は 0~10V となる。FPGA の動作電圧は 3.3V であるので、必要な電圧範囲は 10V も必要がなく、また、コンパレータの差動入力電圧の定格が±5V であるので、MSB を接地して、出力範囲を 0~5V とした。

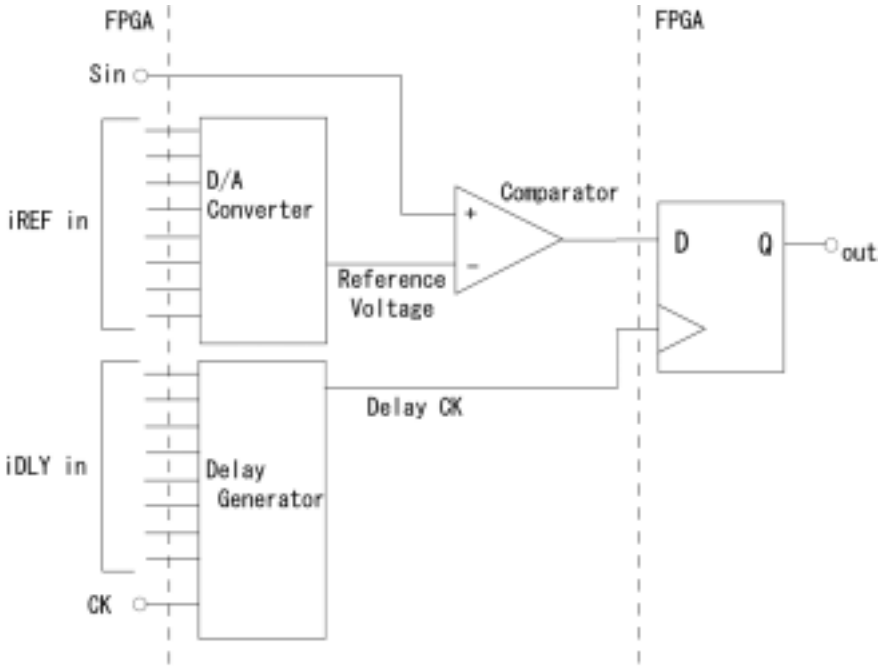


図 4.7　パラメトリックテスト実行用回路の概略図

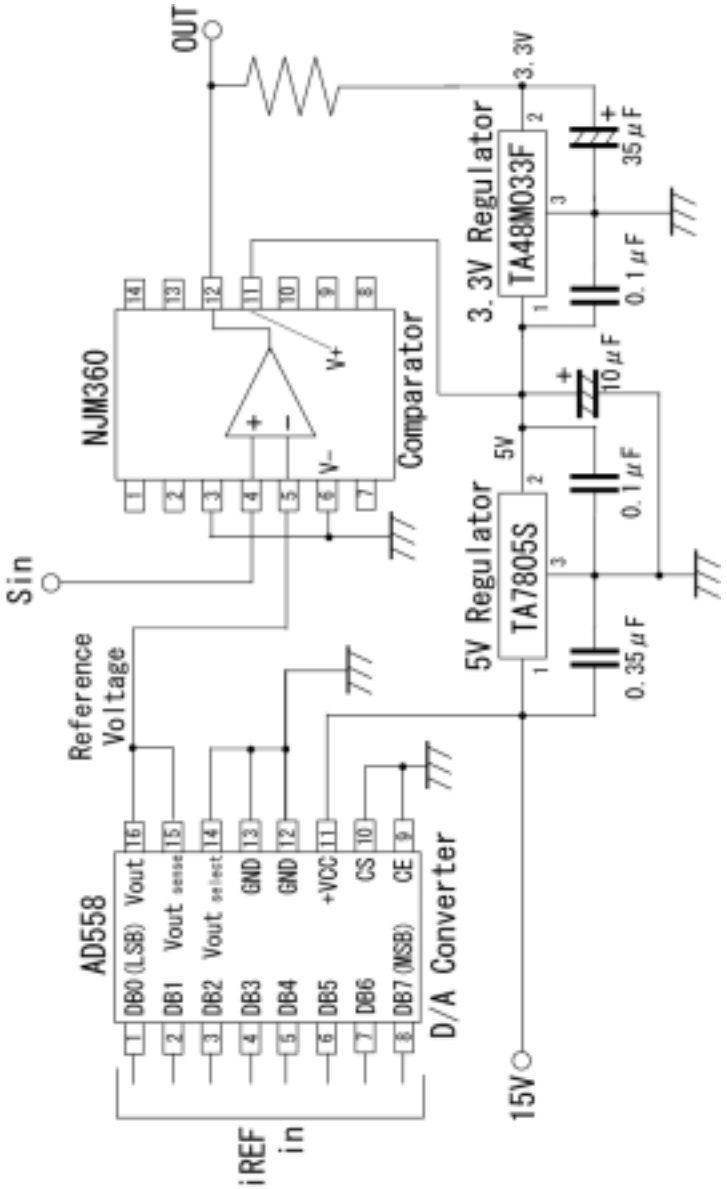


図 4.8　作成した参照電圧生成回路と電圧比較回路の回路図

入力する電源電圧は、D/A コンバータの電源電圧に合わせて 15V とし、この 15V からコンパレータの電源電圧 5V をレギュレータにより作成し、さらにこの 5V から、コンパレータの出力調整用電圧 3.3V を作成した。使用したレギュレータは、15V 入力から 5V の出力を作成するのが TA7805S、5V 入力から 3.3V の出力を作成するのが TA48M033F である。また、GND は全て共通になっている。FPGA の電源は、この回路の電源とは別の電源から 5V を入力し、GND は共通である。

この回路に電源電圧と出力信号を与え、動作確認を行った。D/A コンバータの入出力特性、および参照する信号値を 3.3V で固定し D/A コンバータの出力電圧を変化させたときのコンパレータの入出力特性を図 4.9 に示す。D/A コンバータは入力に比例して出力電圧が増加している。コンパレータは D/A コンバータの出力が 3.3V より低いときは 3.3V を出力してい

るが、3.3　V を越えると出力が 0.5V まで下がっている。よって、D/A コンバータ、コンパレータともに正常に動作していることがわかる。

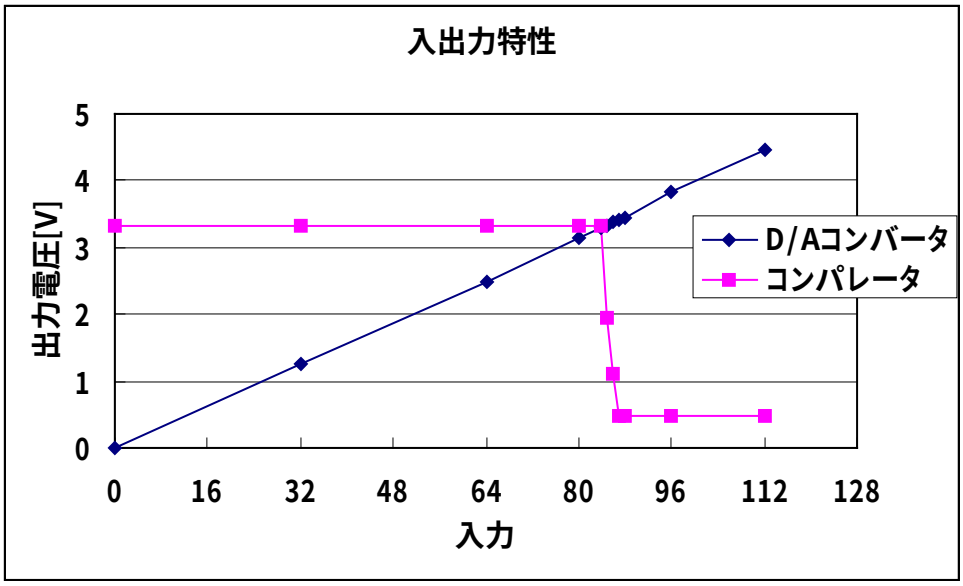


図 4.9　D/A コンバータ、コンパレータの入出力特性

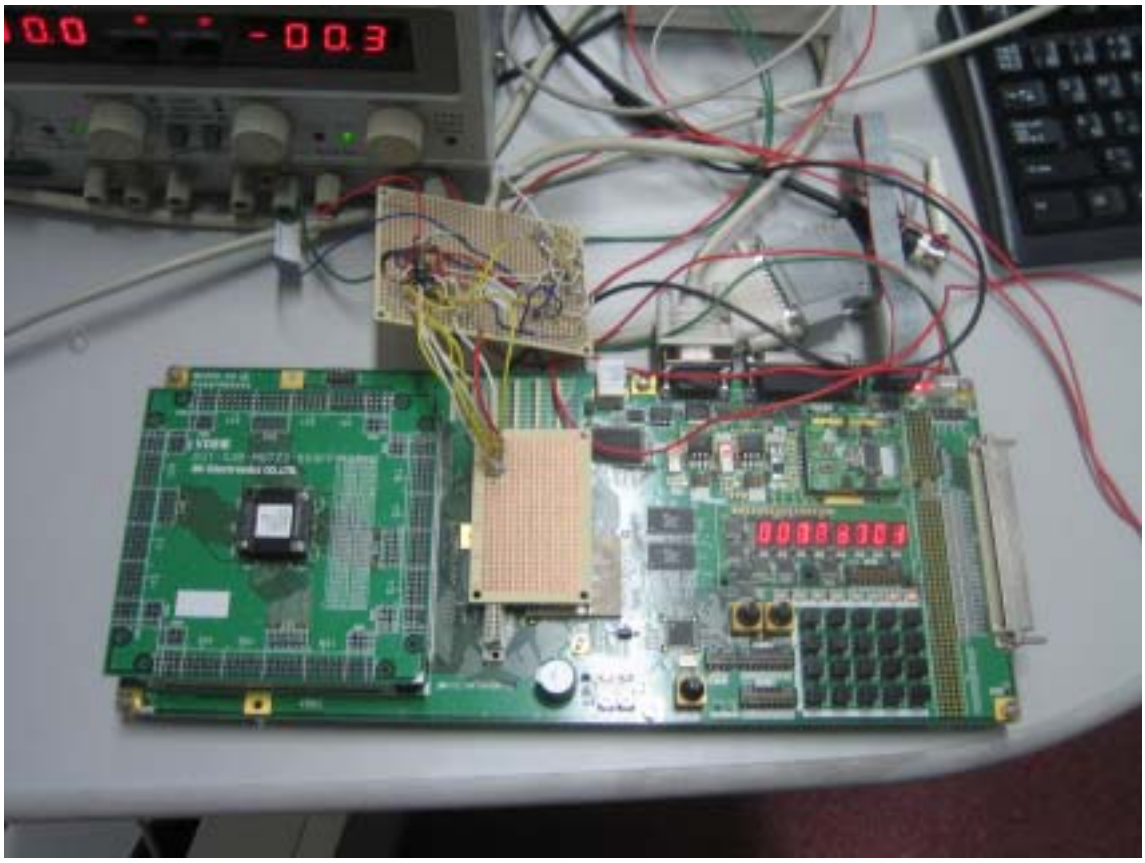


図 4.10 波形取得回路を接続した実装環境

遅延生成回路は、本研究室で製作されたチップを用いる。このチップは、入力されたクロックを、遅延時間後に出力する。遅延時間は、7bit の入力信号により決定され、入力が 1 増加すると、200psec 増加する。

以上の回路を FPGA へと接続し、遅延クロックの立ち上がりでコンパレータからの出力信号を取得し、その結果をインタプリタにより読み出せるようにした。波形取得回路を接続した実装環境を図 4.10 に示す。FPGA 左側のサブボードで遅延生成回路のチップを接続し、中央部の回路が作成した電圧生成回路、コンパレータ回路である。入力される信号については内部コアからの読み出し信号を一つ外部に出力するように回路構成を変更した。

4. 4　インタプリタによる波形取得

以上の回路を用いて、波形の取得を行った。測定は、カウンタの値が 0101 から 0110 に変化するときの 2bit 目の立ち上がり信号を取得することにした。

参照電圧、遅延時間はオンチップテストインタフェース回路のレジスタに値をセットすることで生成される。参照電圧は iREF レジスタに値をセットして生成する。遅延時間は、iDCS レジスタと iDLY レジスタにセットする値によって決まる。iDCS レジスタは、遅延クロックセットアップクロック数を決定する。これは、iEXE が 1 となり実行フラグが 1 になってから何クロック目に遅延生成回路に入力を与えるか、という値であり、50nsec 単位で遅延時間を設定する。iDLY レジスタの値は、遅延回路に入力クロックに入力が与えられてから何 sec 後に遅延クロックを出力するか、という値を決定する。この値が 1 増加すると、遅延時間は 200psec 増加する。この iDLY によって生成できる遅延時間は、最大 25.4psec である。

参照電圧と遅延時間を変化させながら、比較結果の読み出しを行った。このときオシロスコープを用いて用いて得られた波形とインタプリタにより得られた波形を比較した。オシロスコープのプローブを、信号を観測する信号線に接触させて、波形を観測した。このとき、インタプリタによる波形取得の結果を、図 4.11 に示す。オシロスコープによる影響を調べるために、プローブをつけずに波形を取得したときとプローブをつけて波形を取得したときの両方の場合で波形取得を行った。175nsec~200nsec については、iDCS による最小間隔が 50nsec であり、iDLY による最大間隔が 25nsec であるため、遅延時間が設定できず、値が得られていない。オシロスコープにより得られた波形は図 4.12 である。図 4.11 と図 4.12 は、横軸の時間軸を合わせてある。黄線は波形取得を行う信号波形であり、青線は遅延時間 150nsec で立ち上がる遅延クロック生成クロックであり、信号波形のトリガーとして使用した。コンパレータの遅延時間があるため、インタプリタによる波形はオシロで観測した波形よりも遅れたものとなる。

図 4.11 を見ると、オシロがある場合はオシロの負荷容量の影響を受けているように見える。

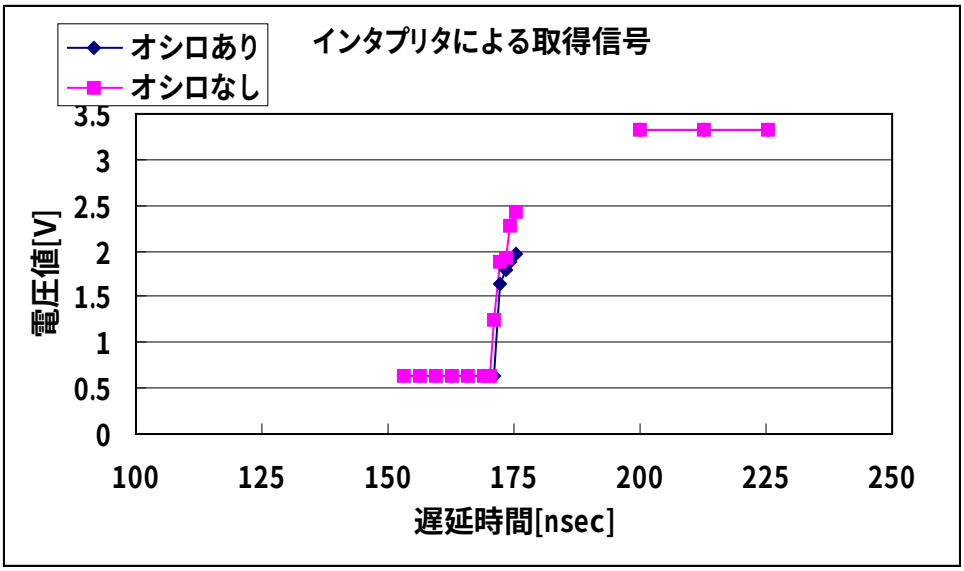


図 4.11 インタプリタにより取得した信号波形

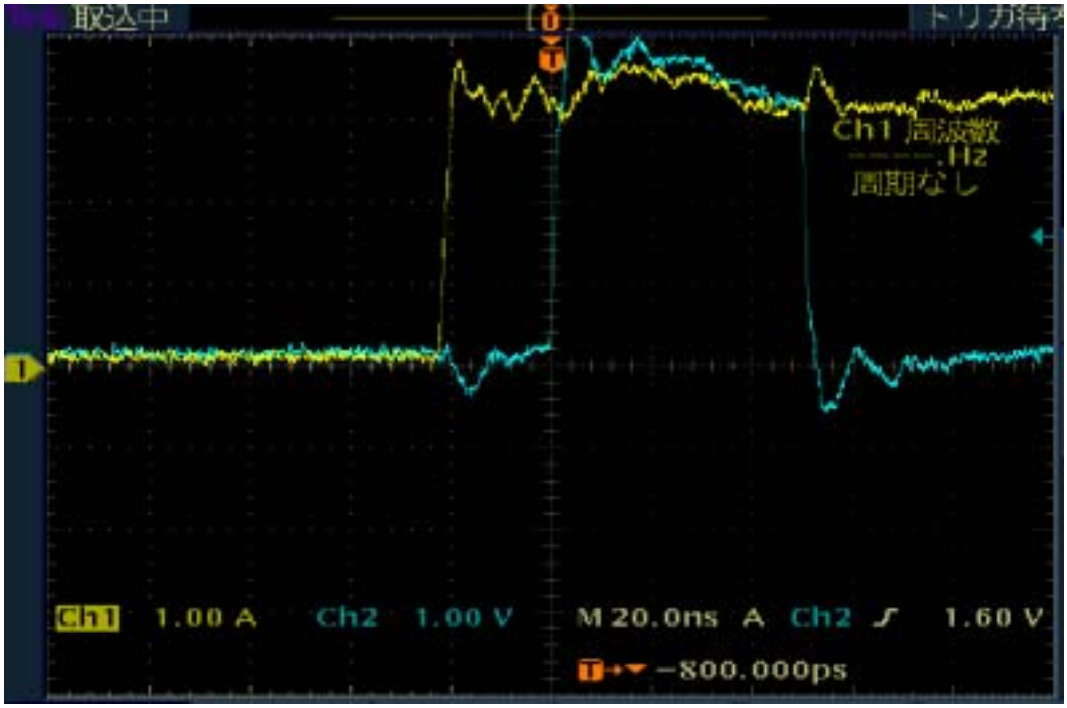


図 4.12 オシロスコープで観測した信号波形

以上のように、インタプリタによって信号の波形を取得することが可能である。

第５章

インタプリタの評価

5．1　記憶領域

作成したインタプリタが使用するメモリのうち、データメモリについて表 5.1 に示す。メモリは、インタプリタによるプログラム環境を実現し、オンチップテストを実行するために必要な領域が確保されていなければいけない。しかし、メモリ量が多くなるとハードウェアオーバーヘッドが大きくなってしまうので、本システムにおいて最適なメモリ量を求める必要がある。以下、表 5.1 に示すそれぞれのデータメモリについて、その機能と必要なメモリ量について説明する。

表5.1　データメモリの割り当てと機能

名前	アドレス	長さ	機能
cmdbuf	0x2000	128	入力された文字列を格納する
tokbuf	0x2080	64	字句解析時にデータを格納する
strbuf	0x20c0	6	字句解析時に文字列比較用の文字列を格納する
varbuf	0x20c6	26	スカラの値を格納する
arrbuf	0x20e0	256	配列の値を格納する
parbuf	0x21e0	16	出力時に文字列を格納する
progbuf	0x21f0	2048	プログラムを格納する
proglist	0x29f0	128	格納されたプログラムの先頭のアドレスを格納する
gyou	0x2a70	128	格納されたプログラムの行番号を格納する
parabuf	0x2af0	1028	パラメトリックテストでコアにセットする値を格納する
prog_p	0x2ef0	1	progbuf 上を指すプログラムポインタ
stack	0x6000	64	スタック

cmdbuf

この cmdbuf は、入力された文を格納するための記憶領域である。入力された文字は一文字ずつ受信され、この記憶領域に格納される。リターンが入力されると、入力された文字列はこの記憶領域より呼び出され、そのまま解析· 実行が行われる。

この記憶領域は、入力可能な一文の長さの上限を決定する。今回作成したインタプリタの文法では、cmdbuf の長さは 128 ワードであれば、オーバーフローすることはないといえる。

tokbuf

この tokbuf は、字句解析時にデータを格納するための記憶領域である。16 進数や行番号のトークンを字句解析する際はそのトークンの文字列をこの記憶領域に格納した後、数値へと変換する。bit 指定のトークンを解析するときは、一時的なデータの格納先として使われる。out 文、read 文等のアルファベットの文字列により構成される文字列を格納する場合は、そのトークンの文字列が tokbuf に格納され、比較用文字列と比較が行われる。

16 進数、行番号の場合は、5 桁目以上は無視するような仕様になっており、必要な長さは区切り文字の格納も含めて 5 ワード。bit 指定の場合は使用するのは 3 ワード。アルファベットによる文字列の場合は、最大の長さの文字列が write、print の 5 文字であるので、区切り文字も含めて 6 ワード必要である。よって、この記憶領域は最低 6 ワードの長さが必要であり、トークンの追加がしやすいように余裕をもたせて 64bit とした。

strbuf

この strbuf は、字句解析時にアルファベットによる文字列の、比較用の文字列を格納するための記憶領域である。

これらのトークンの長さは、本インタプリタにおいては 5 文字以下にするようにしたので、必要な長さは 6 ワードである。この記憶領域への文字列のセットは、ユーザではなくインタプリタにより自動的に行われるので余裕を持たせる必要はなく、長さは 6 ワードでよい。

varbuf、arrbuf

スカラの値を格納する記憶領域が varbuf であり、配列の値を格納する記憶領域が arrbuf である。

スカラは$a~$z までの 26 個としたので、varbuf の長さは 26 ワードである。そして、配列は、@[#0000]～@[#00ff]までの 256 個としたので、長さは 256 ワードである。

配列は、read 文、write 文、pset 文ではピン番号や書き込むデータの設定にも使用される。pset 文では、一つのピンにつき一つの配列が割り当てられるため、配列の長さはピン数と同じだけ必要である。今回の長さだと 256 ピンまで対応できるが、それ以上のピン数であれ

ば配列の個数を増やす必要がある。また、write 文ではピン番号とデータにそれぞれ配列を使用するので、全てのピンに一度に書き込みを行うためにはピン数×2 の長さが必要となる。さらに、第 3 章で説明を行った通り、para 文の結果は配列に出力され、遅延時間の変化回数を n として、必要な配列の個数は

$$3\times 2\times n/16+3\times n$$

となる。今回は配列の個数を変更する必要がないように、遅延時間を 64 点とることにしたが、遅延時間をより多く変更し波形取得の精度を上げるためには、それに対応した配列の記憶領域を確保しなければいけない。

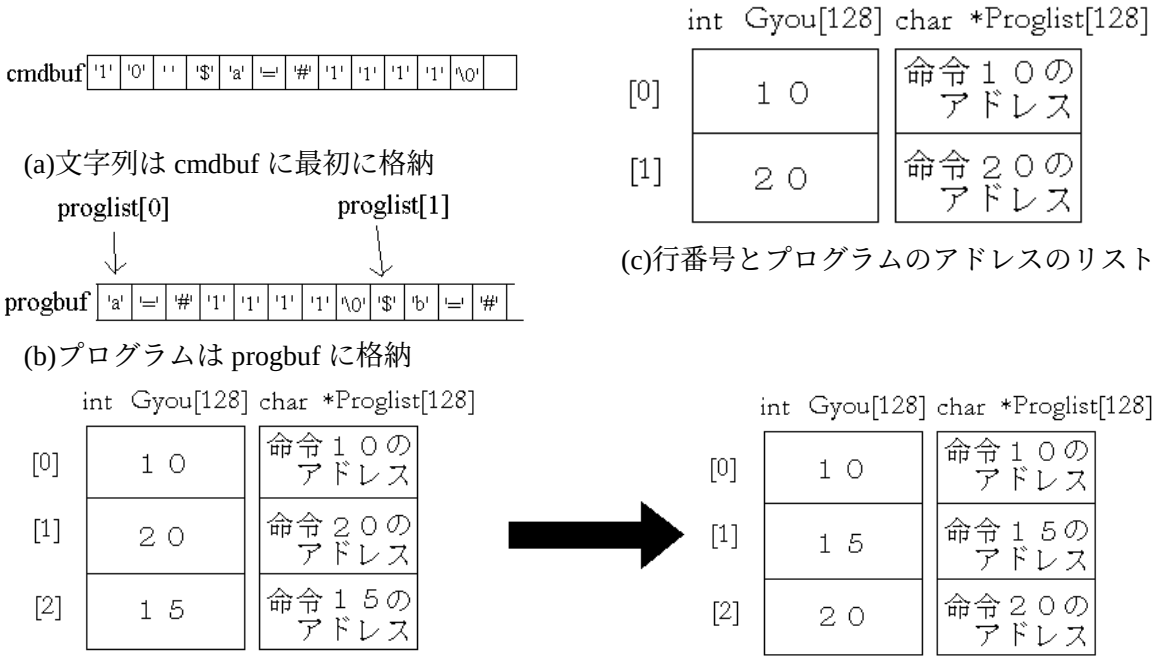
parbuf

parbuf は、出力時に出力する文字列を格納するための記憶領域である。数値や”error”等の文字列を格納する際は、その文字列は parbuf に格納され、一文字ずつ出力される。

長さは 16 ワードとしたが、本インタプリタでは read 文で結果を出力するときに 9 ワード使用するのが最大である。

progbuf、proglist、gyou、prog_p

これらは、行番号つきで入力された文を格納し、文の追加、削除や、list 文による表示、run 文による一括実行を行うための記憶領域である。文の格納は、図 5.1 に示すように行われる。



(d)1 文入力されるごとにソーティング

図 5.1　文の格納

入力された文字列は、文の種類によらず最初は cmdbuf に格納される(a)。入力された文の最初のトークンが行番号ならば、その文は格納される。格納するときは、行番号を除いた文字列が記憶領域 progbuf に格納される(b)。文は記憶領域の先頭から一文字ずつ格納され、1 文の入力が終了すると、区切り文字として'¥0'が格納される。次に文を格納するときは、区切り文字の後から続けて入力していく。次に文の入力が開始される progbuf 内のアドレスの値が格納されるのが prog_p である。このとき、その文の行番号と progbuf に格納された文字列の先頭のアドレスが、それぞれリストとして格納される(c)。行番号を格納する記憶領域が gyou、プログラムのアドレスを格納するのが proglist である。

文が格納される順番は、文字列の progbuf への格納は入力された順に先頭から行われる。gyou、progkist は、1 文の入力が終わるごとに行番号の大きさによってソーティングが行われる(d)。同じ行番号の文が入力されると、古い文が消去され、新しい文が上書きされる。また、行番号だけが入力されたときは、以前入力された文に同じ行番号の文が無ければ何も格納されず、有ればその文の消去を行う。消去された文の文字列は progbuf からは消去されず、文のアドレスが書き換えられるだけである。

記憶領域 progbuf には、消去された意味の無い文字列が残ってしまう。そこで、この progbuf を有効に使用するために、progbuf の使用量が progbuf の長さの 9 割を超えたときは、ガーベージコレクションを実行するようにした。ガーベージコレクションの動作は、図 5.2 に示す通りである。図の長方形は progbuf 内の文字列を示し、区切り文字により区切られている。数位は文字列に先頭から順番をつけたものである。白い領域 2、3、5 が意味のある文字列であり、それ以外の領域がガーベージである。

1. 最初の文字列の先頭のアドレスを見る(上側の矢印)。proglist 内のデータと比較を行い、同じアドレスが proglist 内に存在していなければガーベージと判断される。この場合、文字列 1 はガーベージとなる。
2. ガーベージであった場合は、何の操作も行わず、次の文字列の最初の文字のアドレスを見る。最初の文字は、区切り文字'¥0'を検出し、その次の文字が次の文字列の最初の文字である。そして、ガーベージかどうかを判断する。文字列 2 はガーベージではないと判断される。
3. ガーベージで無かったら、その文字列を前方に移動させる。ガーベージではない文の格納先を下の矢印は示しており、最初の位置は progbuf の先頭である。よって、文字列 2 が移動され、次の文字列 3 のガーベージかどうかの判定が行われる。次の文字列の格納先は、文字列 2 の後になる。
4. 以下、同様の操作を繰り返す。文字列 4 はガーベージなので、何も操作は行われない。
5. 文字列 5 はガーベージではないので、前方にシフトされる。
6. 以上の手順を progbuf の文字列の最後尾まで繰り返す。文字列の最後尾は、prog_p が指すアドレスの一つ前である。よって、現在参照しているアドレス(上側の矢印)の値

と prog_p の値とを比較し、等しければガーベージコレクションは終了となる。

gyou、proglist の長さによって格納可能な文の数が決まり、progbuf の長さにより格納可能な文の文字数が決まる。今回作成したインタプリタでは gyou、proglist の長さは 128 ワードとし、progbuf の長さは 2048 ワードとした。

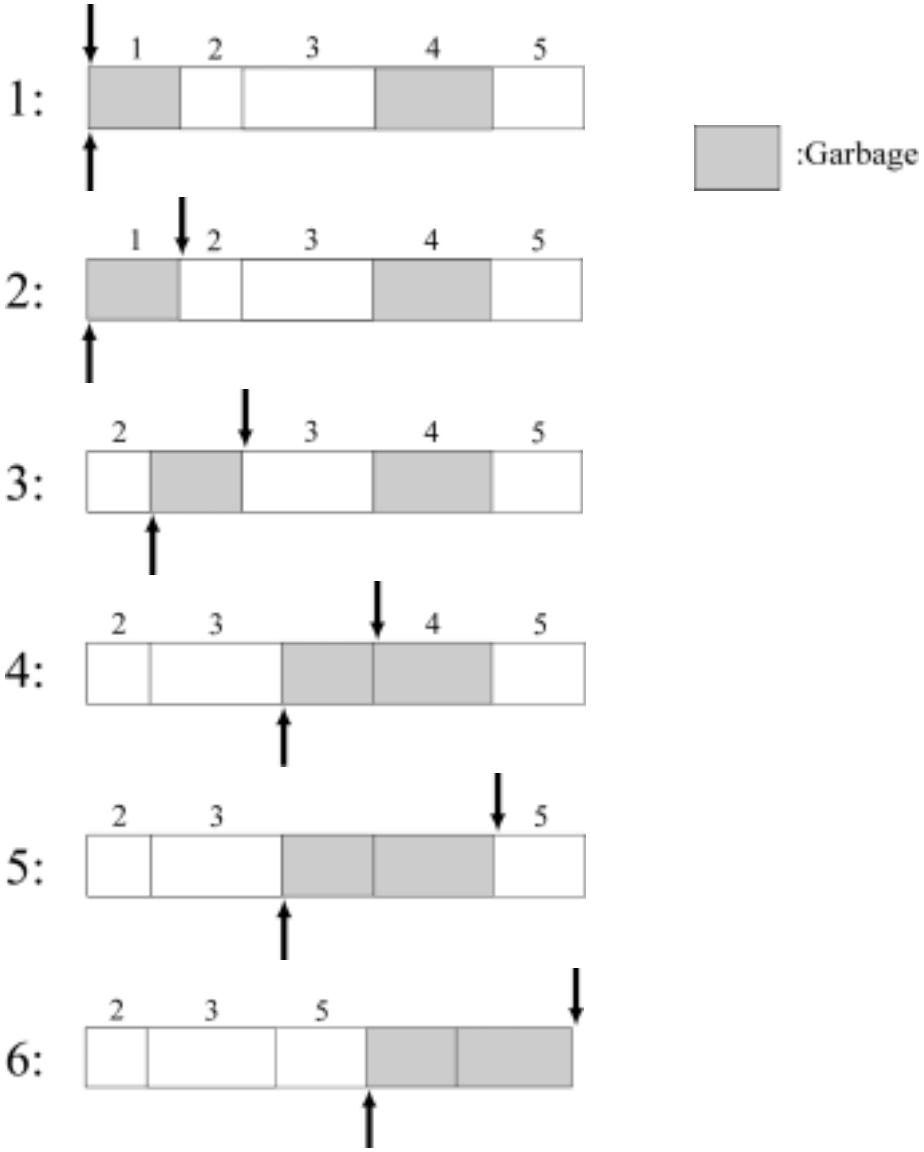


図 5.2　ガーベージコレクションの動作

parabuf

parabuf は、パラメトリックテスト実行時に、内部コアの状態設定のために、クロックごとに全てのピンにセットする値を格納するための記憶領域であり、pset 文によって書き込まれる。

各ピンに書き込む値を、ピン一つにつき 1 ワード割り振り、それを全てのピン、さらに

それをクロックごとに書き込むとすると、ピン数を pin、書き込みを行うクロック数を ck、とすると必要なワード数は pin*ck となる。ピン数を 128、書き込みクロック数を 128 とすると、ワード数は 16384 と、非常に大きくなってしまう。

そのため、データの圧縮を行う。コアの状態設定のためにピンに書き込む信号で必要なのは、コアに書き込む信号だけである。すなわち、ピン一つにつき 1bit でよい。1 ワードは 16bit であるので、1 ワードにつき 16 ピンのデータが格納可能である。よって、必要なワード数は

$$pin/16 * ck$$

となり、ピン数を 128、書き込みクロック数を 128 とすると、ワード数は 1024 となる。

今回の実装では記憶領域 parabuf の長さは 1024 ワードとしたが、テストするチップのピン数と、内部コアの状態設定に必要なクロック数によって必要なワード数は変わるので、この記憶領域の長さもテストするチップに合わせて変更する必要がある。

stack

インタプリタの動作中、サブルーチンにジャンプするときは、必要なレジスタの値を保存する必要がある。この値は、サブルーチンから戻ってくるときに呼び出され、元のレジスタに値が戻される必要がある。そのため、レジスタの値を格納するためのスタックを設けている。

必要なスタックの長さを調べるために、逐次実行と一括実行、さらにテスト専用命令を実行したときに使用される最大スタック使用量を測定した。それぞれの結果をグラフにしたのが、図 5.3~図 5.5 である。スタック使用量は文の実行時にサブルーチン内でサブルーチンを呼び出すことで増加し、サブルーチンから戻ると、呼び出し前の値まで減少する。

これらの結果を見ると、スタック使用量は 30 弱程度のワード数となっている。スタックはサブルーチンの最初に格納が行われる。あるサブルーチン内で呼び出されるサブルーチンは決まっているおり、サブルーチンの再起呼び出しは、配列を字句解析する時意外行われない。そのため、文の構造によって使用されるスタックの使用量はほぼ決まる。よって、スタックの使用量はこの結果から大きく値が変わることがないといえる。

今回は、スタックの長さに特に上限を設けてはいないが、以上のことから 64 程度の長さがあればいいといえる。

以上が今回割り当てたデータメモリであり、インタプリタを格納するプログラムメモリも含めて必要な記憶領域全体のワード数を表 5.2 に示す。

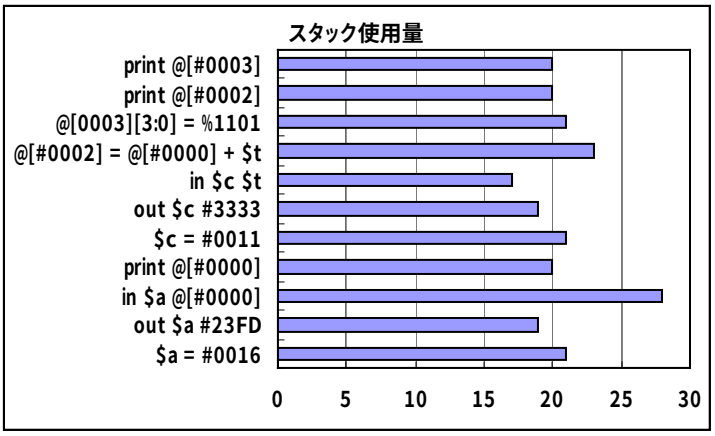


図 5.3　逐次実行時の最大スタック使用量

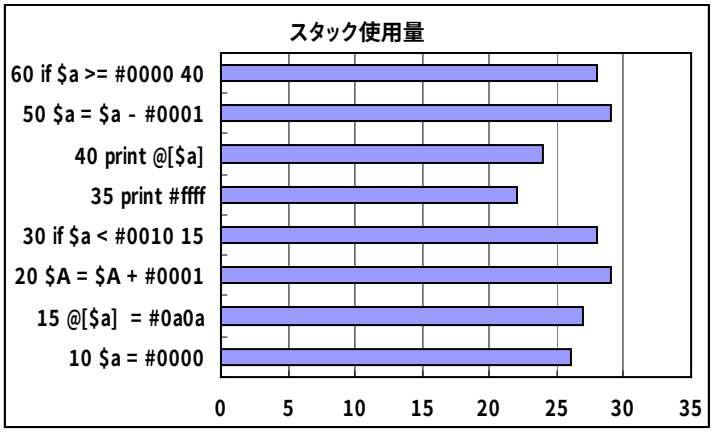


図 5.4　一括実行時の最大スタック使用量

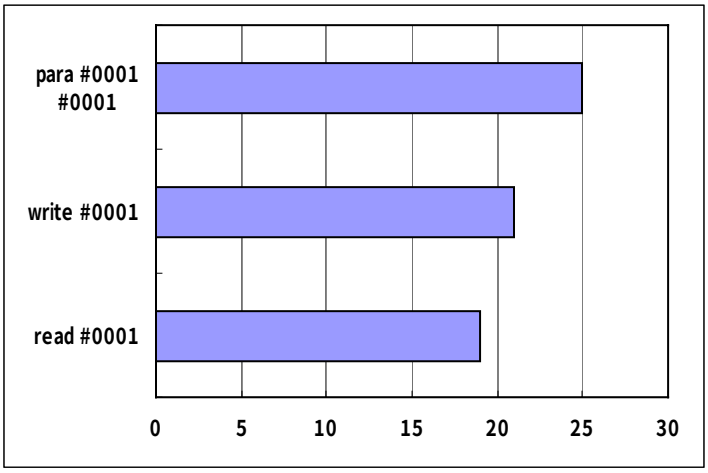


図 5.5　テスト専用命令の最大スタック使用量

表 5.2　必要な記憶領域全体のワード数

記憶領域	ワード数
データメモリ	3893
プログラムメモリ	3981

5．2　実行ステップ数

文を 1 文実行するときの実行ステップ数について、測定を行った。実行ステップ数は、文を実行するときに必要となるクロック数を表し、実行ステップ数にクロック周波数を掛けたものが実行時間となる。そのため、テスト実行時間を減少させるためには、実行ステップ数はより小さい方がよい。

文を逐次実行した時のステップ数を測定した結果を図 5.6 に示す。測定は、10 行程度の適当な文を入力として与え、文の実行開始から実行終了までのステップ数をエミュレータ上でカウントした。

逐次実行では、1 文の実行ステップ数は 800~3000 程度となった。このとき、実行ステップ数の内訳を調べると、大部分が字句解析部である gettoken によるものであった。全体の実行ステップ数のうち、字句解析による割合を表したものが図 5.7 であり、全体の 8 割以上を占めていることがわかる。

そこで、字句解析のみの実行ステップ数を、字句の種類ごとに調べた。その結果を図示したものが図 5.8 である。文字数が長くなるごとにステップ数が多くなる傾向があり、特に 16 進数やそれを含んだ構造の配列、print などの長い文字列で多くなっている。実行ステップ数を減少させ、テストを高速化するためには、字句解析部のアルゴリズムを改良し、より少ない実行ステップ数で字句解析が行えるようにすることが効果的である。

次に、一括実行についても実行ステップ数の測定を行った。このときの結果を図 5.9 に示す。逐次実行可能な文については、逐次実行でも一括実行でも、ほとんど同じステップ数となった。また if 文については 3900 ワード程度と他の文に比べてやや長いが、これはトー

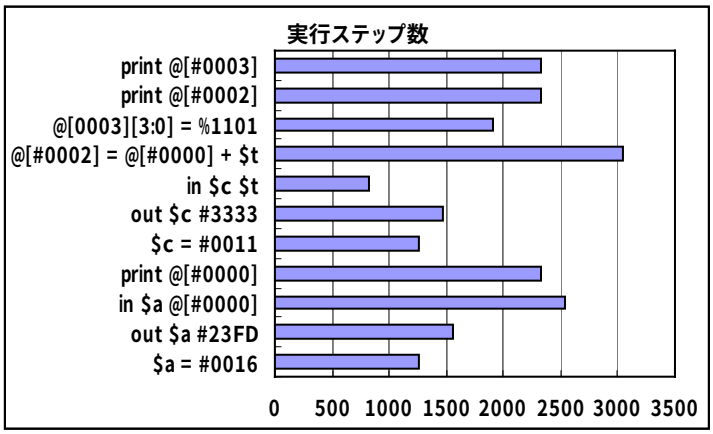


図 5.6　逐次実行時の実行ステップ数

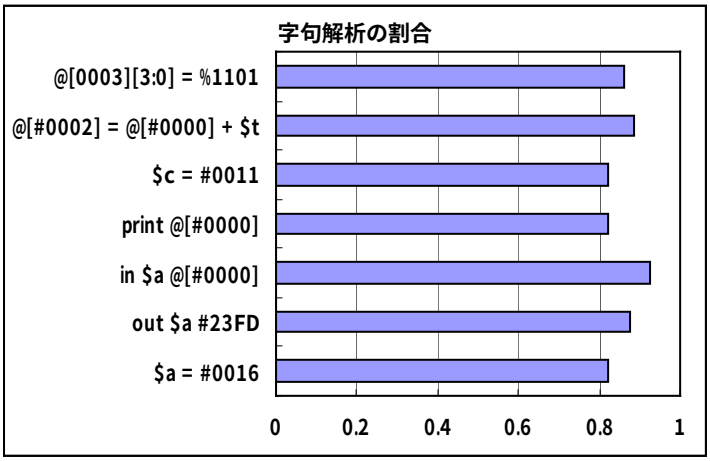


図 5.7　実行ステップの字句解析の割合

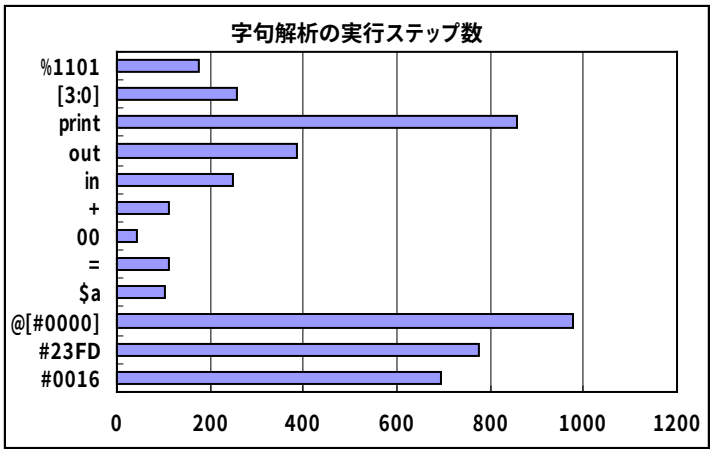


図 5.8　文字列の種類による字句解析の実行ステップ数

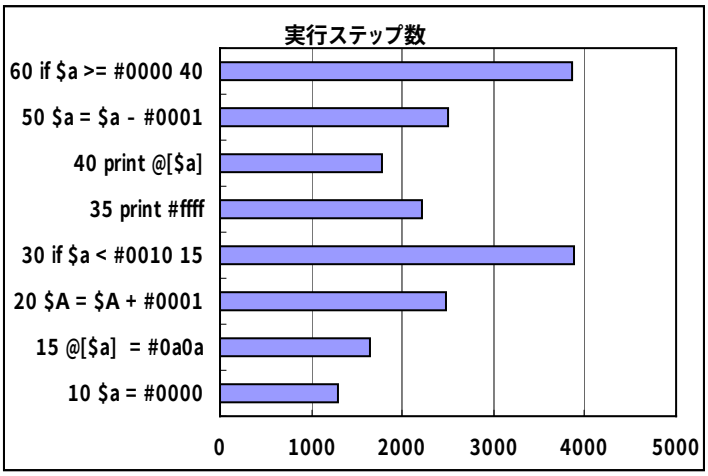


図 5.9　一括実行時の実行ステップ数

クンが多いためである。

また、テスト専用命令についても実行ステップ数の測定を行った。図 5.10 には read 文実行時の、図 5.11 には write 文実行時のステップ数について示す。read 文と write 文は、$z の値を変えて、読み出し又は書き込みを行う回数を変えながら何度か測定を行った結果を示している。read 文の実行ステップ数が write 文に比べて大きいが、これは read 文は結果の出力を含んでいるためである。

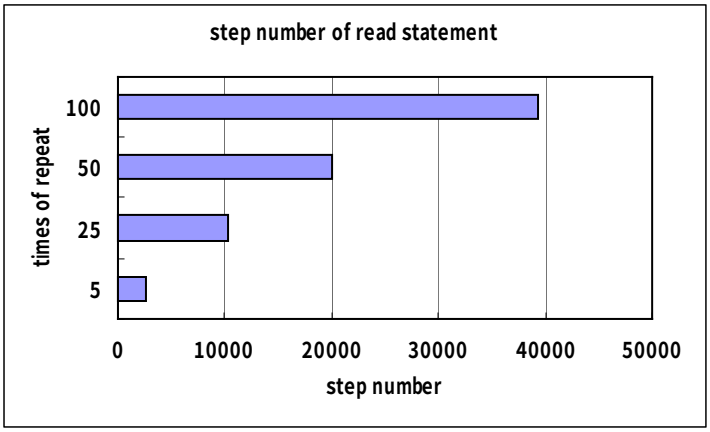


図 5.10　read 文の実行ステップ数

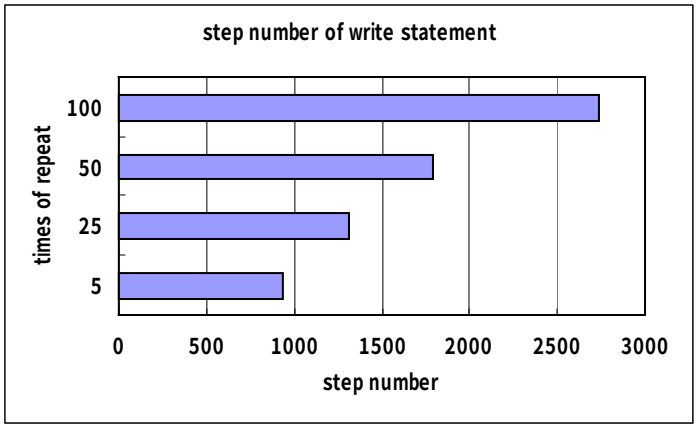


図 5.11　write 文の実行ステップ数

図 5.12 には、para 文の実行ステップ数を測定した結果を示す。比較回数による実行ステップ数の変化を見るために、遅延時間の変化回数を作成したインタプリタでは 64 回であるが、48 回としたときの結果も合わせて示している。この結果を見ると、para 文はかなり実行ステップ数が多い。これは、para 文の内部では比較を繰り返し 6144 回又は 8192 回行っているためと、比較結果を元に結果を算出するアルゴリズムがデータメモリの使用量を最小にすることを目的としたものになっているためである。

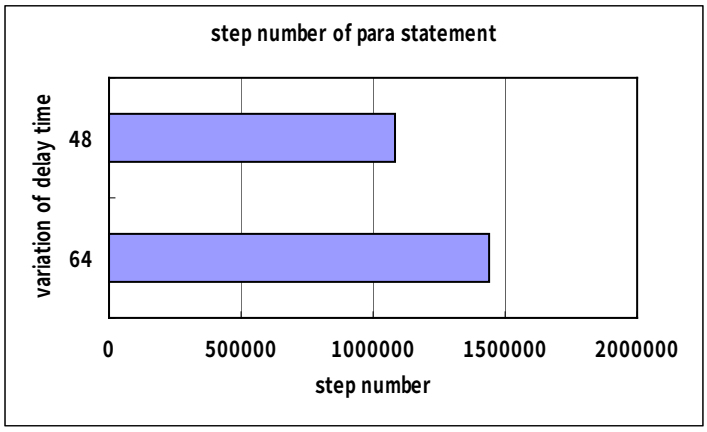


図 5.12　para 文の実行ステップ数

第６章

まとめ

6．1　結論

本論文では、システムLSIのオンチップテストのための、チップ上にCPUを搭載し、そのCPU上でインタプリタを動作させることによりプログラム環境を実現するテストシステムについて行った。

ハードウェアは、CPU、インタプリタを格納するメモリ、バウンダリレジスタ、オンチップテストインタフェース回路、RS-232C インタフェース回路であり、本研究室で設計されたものを用いた。

搭載するインタプリタは設計を行い、それをアセンブラにより記述を行い、機械語に変換してエミュレータにより動作確認を行った。インタプリタの言語仕様は BASIC 言語を元にしたものとし、逐次実行とプログラムを格納しての一括実行の両方が可能である。使用可能な文は、ハードウェア規模を最小にするため、必要最低限な文のみを持つものとした。さらに、オンチップテストを行うための、テスト専用命令をもつものとした。

テスト専用命令は、バウンダリからの値の読み込みを行う read 文と、バウンダリへの値の書き込みを行う write 文、パラメトリックテストを実行する para 文、パラメトリックテスト実行の際にセットするコアの内部状態を設定するための pset 文である。

必要なハードウェアを FPGA 上に実装した。インタプリタは、メモリの中身として FPGA にダウンロードした。そしてホストコンピュータと RS-232C により接続し、ホスト上の RS-232C に対応したターミナルを用いて動作させた。その結果、作成したインタプリタのテスト専用命令以外の文についてはエミュレータ上で動作確認したのと同様の動作をすることが確認できた。

FPGA に実装したハードウェアの規模を調べたところ、チップ上に実装することを考えれば許容範囲内であった。

テスト専用命令についても動作確認を行った。テスト用の内部コアとして、4bit のカウンタを設けておき、コアにシステムをクロックを与えカウンタを進めながら、バウンダリレ

ジスタによりコアの値の読み込み、コアへの値の書き込みを行い、作成した read 文と write 文によって行うことが可能である、という結果が得られた。

FPGA 上ではアナログ回路が実装できないため、FPGA の外部に接続するパラメトリックテスト用回路を作成した。これにより、インタプリタで信号波形の取得を行うことが可能であった。

また、作成したインタプリタの評価を、記憶領域と実行ステップ数の点で行った。このハードウェアは全てチップ上に実装されることを想定しており、そのためには必要な回路規模を見積もることが必要である。今回のインタプリタに必要なプログラムメモリとデータメモリの算出を行い、インタプリタの動作に必要最低限なメモリの大きさを決定した。テストの実行速度は、実行ステップ数により決定される。この実行ステップ数をエミュレータを用いて測定した。

このオンチップテストシステムは、ユーザにプログラム環境を提供することで、ユーザはより柔軟かつ多機能なテストを実行することができる。システム LSI のテストにおいては、ノイズや温度など、1 と 0 だけではないパラメータに関してのテストが必要になってくる。本研究では波形取得を行ったが、他のテスト回路を追加してインタプリタをそれに合わせた仕様にすることで、容易にコントロールすることができる。この CPU を用いてプログラム環境を実現する方法ならば、ユーザに対してテスト回路のコントロール容易性を与えることが可能となる。

付録 A.1 インタプリタの言語仕様書

基本型
======
16 進数：#0000 から#FFFF までの 16bit。16 進数は、16 より大きいビット数を入力すると、
上位 16bit の値をとる。
ビット数が 16bit 未満の数は入力を受け付けない。
2 進数：%0 から%1111111111111111 までの最大 16bit の可変長。2 進数は、16 より大きい
ビット数を入力すると、上位 16bit の値をとる。
定数：16 進数または 2 進数。
スカラ：$a から$z までの 26 個。#0000 から#FFFF までの定数を格納。
配列：@[数]。数の下位 8bit を値としてとる。
上位 8bit は無視される。
#0000 から#FFFF までの定数を格納。
変数：スカラまたは配列。
数：定数または変数。
bit 指定：数の後に[0]から[F]。操作する数の 0 から 15bit のうち 1bit を指定できる。
[F:0]と:で区切ると複数の bit を指定する。:の左の bit が右の bit より大で
ないとエラーとなる。
変数の値の参照時は変数に格納されている値に対して bit 指定操作を行う。
変数への値の格納時は、その変数の値を格納するメモリの何 bit 目に格納する
かが指定指定される。
行番号：0 から 9999 までの可変長最大 4 桁の１０進数。5 桁以上入力された場合は、上位
4 桁をとる。
算術演算子：+, - の２個。
関係演算子：=, !, <, >, <=, >= の６個。

文
==
すべての文は 1 行で 1 文。余分なスペースとタブは無視される。
文は一行入力されるごとに逐次実行される。
以下の形式に合致しない文が入力された場合は、エラーとなる。
エラーになると、標準出力に error と出力され、その文は何も実行されずに処理が終了となる。逐次実行では次の文の入力待ちの状態に移行する。行番号による格納で run 文による一

括実行では、次の文を実行する。

<左辺> = <右辺> (代入)
<左辺> = <右辺１> + <右辺２> (加算)
<左辺> = <右辺１> - <右辺２> (減算)

右辺の演算を行った後、左辺への代入を行う。
<左辺>は変数型、<右辺>は数型

例：$a = $b
@[#0010][6] = %1
$c = #0045 + $b

in <アドレス> <格納先>

<アドレス>により指定されるアドレスのメモリからデータを読み込み、
<格納先>に格納する。
<アドレス>は数型、<格納先>は変数型。
<アドレス>を bit 指定して操作すると、<アドレス>の数値に対してではなく、
<アドレス>が示すメモリのデータに対して bit 指定操作を行う。

例：in #0023 $a
in #00a0[3:0] @[$f][7:4]
in $c $d

out <アドレス> <データ>

<アドレス>により指定されるメモリのアドレスに、<データ>の値を格納する。
<アドレス>、<データ>ともに数型。
<アドレス>を bit 指定して操作すると、<アドレス>の数値に対してではなく、
<アドレス>が示すメモリの何 bit 目に書き込まれるかが指定される。

例：out #0072 $a
out $a #2513

print <数>

<数>の値を標準出力に出力する。
出力は 16bit の 16 進数で表示される。

例：print $c
print #10A5

read
read <読み出し回数>

バウンダリからの値の読み出しを行う。
<読み出し回数>は読み出しを行う回数で、数型。
引数無しの場合はスカラ$z の値が読み出しを行う回数となる。
読み出しを行うピンのアドレスは、配列の先頭から順に一つずつ読み出す。
読み出し 1 回目では先頭の@[#0000]からアドレスを読み出し、2 回目では
次の@[#0001]から読み出す、と読み出し回数分行われる。
読み出し結果は、標準出力に出力される。
出力結果は、読み出し 1 回あたり
<ピン番号>:<2 進数 1><2 進数 2><2 進数 3>
の形式で出力される。<2 進数 1>はコアへの書き込み信号、<2 進数 2>はコアからの読み出し信号、<2 進数 3>はコアからの制御信号の値を表す。

出力例：0000:110
0015:010

例：read #0005

write
write <書き込み回数>

バウンダリへの値の書き込みを行う。
<書き込み回数>は書き込みを行う回数で、数型。
引数無しの場合はスカラ$z の値が書き込みを行う回数となる。
書き込みを行うピンのアドレスは、配列の先頭から順に一つずつ読み出し、
書き込む値は@[#0080]から順に一つずつ読み出す。
書き込み 1 回目では先頭の@[#0000]からアドレスを、@[#0080]から書き込む値を読み出し、二回目では次の@[#0001]からアドレスを、@[#0081]から書き込む値を読み出す、と書き込み回数分行われる。

書き込む値は、数の下位 3bit 目がコアへ書き込む値、下位 2bit 目がコアから読み出す値、下位 1bit 目が制御信号の値を表す。
それ以外の bit は、無視される。

例：write #000f

pset <クロック数>

パラメトリックテスト実行時にコアの内部状態をセットするために、あるクロックでコアに書き込む値を設定する。
<クロック数>は数型。何クロック目に書き込む値を設定するかが決定される。
全てのピンにつき一つずつ配列を割り振り、配列の値がそのクロックでのピンに書き込む値としてメモリに格納される。
配列の割り振りは、@[<ピンのアドレス>]となる。
書き込む値は下位 1bit だけをとり、コアへの書き込み信号の値とする。それ以外の bit は無視される。

例：pset $e

para <ピン番号> <クロック数>

パラメトリックテストを実行し、任意のコアの状態での信号波形を取得する。
<ピン番号>は数型。信号波形を取得するピンのアドレスを設定する。
<クロック数>は数型で、コアの状態をセットするために必要なクロック数を設定する。
各クロックごとに、pset 文で設定されたピンに書き込む値を読み出し、コアへの書き込みを行う。
出力結果は、遅延時間ごとの電圧値が配列に格納される。
配列@[#0018]から@[#0057]にはコアへの書き込み信号の結果、@[#0058]から@[#0097]にはコアからの読み出し信号の結果、@[#098]から[@#00d7]にはコアからの制御信号の結果が格納される。
@[#0000]から@[#0017]は、para 文実行時にデータ処理のため使用されるので para 文実行以前に格納されていた値は失われる。

例：para #0O1F $e
　　para $a #0010

行番号　文

文の実行を行わずに記憶領域にその文を格納する。
run が入力されたときに一括で行番号が小さい文から順番に実行する。
同じ行番号の文が入力されたときは、以前蓄えられていた文は新たに入力された文に置き換えられる。

例：10 out $a #2513
　　20 print $a

行番号

行番号のみが入力されたときは、それ以前に入力されていた文に同じ行番号の文があれば、その文が消去される。
同じ行番号の文が無ければ、何もしない。

if <数> <関係演算子> <数> <行番号>

関係式<数> <関係演算子> <数> が真のときに、<行番号>の示す行へ制御を移す。
偽ならば次の文を実行する。
if 文は、行番号による入力で格納され、run で実行されたときのみ動作する。
逐次実行ではエラーとなる。

例：if $a <= #1000 50
　　if $b ! $c 100

list

現在行番号による入力で蓄えられている文を、行番号の小さい順に表示する。
list 文は、逐次実行のみで動作する。
行番号によりこの文を格納し run で実行すると、エラーとなる。

run

行番号による入力で蓄えられている文を行番号の小さい順に一括実行する。
run 文は、逐次実行のみで動作する。
行番号によりこの文を格納し run で実行すると、エラーとなる。

例

==

アドレス#0016 のメモリに#23FD という値を格納する。その値を配列@[#0016]に入れて、配列@[#00a0]にはその値から#1000 を引いた値を入れる。配列@[#00a0]の値を表示する。

```
>$a = #23FD
>$b = #0016
>out $b $a
>in $b @[$b]
>@[#00a0] = @[$b] - #1000
>print @[#00a0]
13FD
```

配列@[#0000]から配列@[#0040]まで一つおきに%1 の値を入れる。次に配列@[#0001]から配列@[#003f]に一つおきに%10 の値を入れる。

```
>10 $a = #0000
>20 @[$a] = %1
>30 $a = $a + #0002
>40 if $a <= $b 20
>15 $b = #0040
>run
>10 $a = #0001
>20 @[$a] = %10
>run
```

参考文献

[1]IEEE 1149.1-2001, “Standard Test Access Port and Boundary-Scan Architecture”, IEEE Standards Board, 2001

[2]坂巻佳壽美、「JTAG テストの基礎と応用」、CQ出版社、1998

[3]IEEE 1149.4-1999, “Standard for a Mixed Signal Test Bus”, IEEE, USA, 2000

[4]IEEE P1500, “http://grouper.ieee.org/groups/1500”

[5] Tehranipour. M.H, Ahmed. N, Nourani. M, “Testing SoC Interconnects for Signal Integrity Using Extended JTAG Architecture,” Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, Vol:23, Issue:5, pp:800-811, May 2004

[6] Leon van de Logt, Frank van der Heyden, Tom Waayers,“An extension to for at-speed debug on a system,” Proc of., ITC International Test Conference, Vol.2, pp:123-130,2003

[7] M. Takamiya, M. Mizuno, ”A Sampling Oscilloscope Macro toward Feedback Physical Design Methodology”, Symposium on VLSI Circuits, pp:240-243,2004

[8] Ingeol Chun, Chaedeok Lim, ”ES-debugger: the flexible embedded system debugger based on JTAG technology” Advanced Communication Technology, ICACT 2005. The 7th International Conference on, Volume 2, pp:900 – 903, 2005

[9] Sungbae Hwang, Abraham, J.A, ” Test data compression and test time reduction using an embedded microprocessor” IEEE Transactions on, Very Large Scale Integration (VLSI) Systems, Volume 11, Issue 5, pp:853 – 862, Oct, 2003

[10] Kuen-Jong Lee, Chia-Yi Chu, Yu-Ting Hong, ”An Embedded Processor Based SOC Test Platform”, Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on, pp:2983-2986, 2005

[11] ドナルド M.モンロ著、高澤嘉光、吉川久男補訳、「ベーシック BASIC」、近代科学社、1981

[3-2] 寒河江忠男，AASM Version 3.60，
”http://www9.plala.or.jp/sagae_/soft/index.html”

本研究に関する発表

1．村田泰亮, 吉田浩章, 池田　誠, 浅田　邦博.” SoC のオンチップテストのためのプログラム環境”, 電子情報通信学会総合大会, 2006 年 3 月(発表予定)

謝辞

本研究を進めるにあたり、池田誠助教授には本研究を行う上で常に多くのご意見、ご指導を頂きまた貴重な時間を割いて研究活動を助けて頂き_ 心から大変感謝しております。また、適切なご指導を頂き、研究する上で良好な設備を与えて下さいました浅田邦博教授に深く感謝致します。

浅田研究室の助手鄭若丹彡氏には、FPGA 等の装置の使い方を丁寧に教えて頂き、感謝の念を強く抱いております。

さまざまな場で数多くの御助言、御支援を下さいました浅田研究室の助手佐々木昌浩氏、大学院生の Mohamed Abbas Abdel-Rday 氏、吉田浩章氏、飯塚哲也氏、谷内出悠介氏、新宅宏彰氏、三瓶真弘氏、Dia Kin Hooi 氏、田島貴明氏、山内裕史氏、山本一統氏、金允王景氏、梁 志成氏、風間大輔氏、橋本紘和氏、門馬太平氏、石井健氏、技官の鈴木真一氏、秘書の横地順子氏、丸山由香子氏、卒論生の井上拓郎氏、栗原健一郎氏、曽我部拓氏および金雄鉉氏に深く感謝します。