2001 年度 卒業論文

PCI バスを介した PC と書き換え可能な集積回路との通信の制御

電気通信大学 電気通信学部 電子工学科

9610011　稲荷 徹

指導教官 齋藤 理一郎 助教授

提出日 平成 14 年 2 月 7 日

目次

謝辞

本研究および論文作成にあたり、懇切なる御指導、を賜わりました指導教官である
齋藤理一郎助教授に心より御礼の言葉を申し上げます。本研究およびセミナー等で御指導を賜りました木村忠正教授、湯郷成美助教授、一色秀夫助手に厚く感謝の意を表します。また、本研究をするにあたり、さまざまな資産を残して頂いた八木将志様、中島瑞樹様、松尾竜馬様、
グエン・ドゥック・ミン様、山岡寛明様、ホー・フィ・クー様、沼知典様、清水信貴様に多大なる感謝をいたします。特に清水信貴様には丁寧に直接指導して頂きました。改めて感謝致します。さらに、布田将一様をはじめとする木村研究室、湯郷研究室の皆様方にも感謝致します。本研究にあたって、 Max+PlusII を無償で提供して頂きましたアルテラ・ユニバーシティプログラムマネージャー宮田喜明様をはじめ、日本アルテラ（株）にも感謝致します。

第 1 章

序論

1.1 本研究の背景

量子力学をはじめとする科学計算においては、膨大な計算量を要する。物性計算を例にとると、行列の固有値、固有ベクトルを求める必要がある。その計算量は行列の次数を N とすると $O(N^3)$、つまり次数の 3 乗に比例し、科学計算で使われる 1000 次以上の大規模な計算においては PC では数時間以上かかり、コンピュータの使用効率を下げてしまう。

この計算時間短縮の手法として、並列コンピュータを用いた計算の並列化や新しい行列計算アルゴリズムなどがあげられる。しかし、並列化の問題点としては、並列化できない演算部分があり、コンピュータ数を増やしても、その部分は計算時間の短縮はできない。また、コンピュータ間での通信にも時間がかかるため、その部分も短縮はできない。

また、新しい行列計算アルゴリズムとして、 $O(N)$ 法などがあげられるが、このようなアルゴリズムでは、計算時間の短縮が期待できるが、厳密解が得られないという問題点がある。

我が研究室では去年まで使用していた評価基板に搭載されている SRAM を使用した行列計算を行うことができた。しかし昨今のコンピュータの性能の向上により、 PC と評価基盤との通信速度の短縮が要求されてきた。

1.2 前年度までの研究成果

本研究は当初、 (株) 画像技研 との共同研究として、科学計算を高速に行うために、専用の計算機を開発しようという目的で、 1996 年度から始まった研究である。ここでは、前年度までの本研究の成果について述べる。

まず、'96 年度は、本研究室の中島 [1] と八木 [2] が、行列の固有値および固有ベクトルを求めるためのアルゴリズムであるハウスホルダ法をこの計算機に搭載するアルゴリズムとして採用し

た。このアルゴリズムを採用した理由としては、本研究室で行われている量子力学上の分子起動計算では行列計算を多用している。この計算において固有値、固有ベクトルを求めるために、多くの時間を要しているため、この計算時間を短縮するための手法として、ハウスホルダ法を採用した。さらに、ハードウェア上での三重対角化から逆反復法までの計算過程のモデルを提案した。

そして'97 年度は、松尾 [3] とグェン [4] が、計算アルゴリズムを実際に動作させるためのハードウェアを作製するための設計方法を決めた。研究室で設計を行うために、設計の容易さと開発コストを考慮しなければならない。そこで、近年ディジタル回路の設計手法として一般的になってきたハードウェア記述言語 HDL(Hardware Description Language) を採用した。そして、この言語により設計した機能をハードウェアとして動作させるために、プログラマブルデバイスである FPGA(Field Programmable Gate Array) を採用した。

本研究室では、これらを用いた開発環境を得るために、(株) インターリンク社より PeakVHDL を HDL 設計ツールとして購入し、(株) 日本アルテラ社のユニバーシティプログラムに参加し、FPGA の配置、配線ツールである Max+plusII の無償提供を受けた。そして、(株) アルティマから FLEX10-K シリーズのひとつである EPF10K100GC503-4 という FPGA を 2 個購入した。

この FPGA を使用した専用計算機を構築するためには、FPGA を搭載するための基板が必要である。そこで、松尾はこの FPGA2 個、かつ SRAM(Static Random Access Memory)、DRAM (Dynamic Random Access Memory) といったメモリを実装した基板を設計、製作した [3]。また、PC とこの基板間でのデータ通信が可能なインターフェースボードを製作した。そして、計算アルゴリズムを VHDL(VHSIC Hardware Description Language) によって記述し、シミュレーションによって、この計算アルゴリズムをハードウェアレベルで動作させるためのモデルを築いた。

'98 年度は、山岡 [5] と沼 [6] により、先に製作された基板を利用してハウスホルダ法のアルゴリズムを使い、実際に行列の固有値と固有ベクトルの計算をハードウェア上で動作させた。まず、基板と PC との間でデータ通信を行うための VHDL を設計し、PC と FPGA 間の通信を行った。そして、SRAM コントローラを設計し、計算の対象となるデータを SRAM に記憶させることができた。

次に、固有値計算を行うための準備として、積和器の設計を行った。この積和器は行列の計算を行う上で非常に重要な要素となっている。最後に、ハウスホルダ法の三重対角化からの逆反復法など 4 つのアルゴリズムを VHDL で設計し、実際に行列の固有値計算がハードウェア上で動作が可能となった。ただし、この動作には SRAM を用いているので、メモリ容量に制限がある。

'99 年度は沼 [6] により、新たに DRAM を用いて DRAM を制御するサブプログラムを作製して行列の固有値と固有ベクトルを求めるプログラムを作製した。DRAM は SRAM に比べてメモ

リ容量が 8 倍と大容量であるが、 SRAM にアクセスするより、 3 倍ほどのクロックを必要とし、リフレッシュと呼ばれる電荷を補充する動作が必要となる。

’2000 年度は清水 [7] により、松尾 [3] により製作された FPGA 搭載評価基板である PC との通信に ISA バスを使用した基板を用いての SRAM をデータメモリに指定しての高周波クロック (10MHz) での行列演算装置が開発された。クロック周波数 10MHz での行列の乗算器、高次数での乗算器が可能となった。

1.3 目的

本研究の目的は清水 [7] によって提案された PC との通信に PCI バスを使用した FPGA 搭載評価基板を (株) アリテックより購入した。前年度まで使用してきた ISA バスの通信ではデータのバス幅 16bit であったのに対し、 PCI バスはデータのバス幅は 32bit である。よってこの PCI バスを使った FPGA 搭載評価基板を使用し、前年度までの評価基板との実際の比較を行い、より高速な行列演算器 の完成させる。また、 PCI バスを使った FPGA 搭載評価基板の導入により開発に使う各種ソフトウェアの増加、性能の向上に伴い新機能の増加など各種ソフトウェアを本研究で使用できるためにカスタマイズする。

1.4 本論文の構成

第 2 章において本研究を行う上での各ハードウェア、ソフトウェアの説明を述べる。第 3 章では PC から PCI 評価基板とのデータのやり取りを １つ１つのデータを送るシングルモードによる通信、と簡単な演算について説明する。第 4 章ではデータをシングルモードではなく連続してデータを転送できるバーストモードサイクルを使用したデータの通信について説明する。最後に第 5 章に考察と今後への提言を述べる。付録には VHDL、 Visual C++ のプログラムソースと各種ソフトウェアの使用方法について述べる。

1.5 用語解説

以下において、本研究を行う上での必要最低限の用語を列挙する。

- VHDL
  VHDL とは、 VHSIC Hardware Description Language の略であり、米国防省において VHSIC プロジェクトの一環で、 1981 年にハードウェア記述言語として提案された。 HDL は他に Verilog-HDL があり、日本国内ではこちらの方が一般的であるが、本研究では VHDL

を用いている。その理由はグローバルな研究をする上では世界規格である VHDL を採用することが重要であるという考えによるものである。

- FPGA
  FPGA とは、 Field Programmable Gate Array の略であり、書き換え可能なゲート素子のことである。同じく書き可能なゲートとしては、 CPLD (Complex Programmable Logic Device) というものがある。 FPGA と CPLD はいづれも書き換え可能なゲート素子であるが各々構造の違いにより区別されるがここでは詳しくは言及しない。本研究で使用する FPGA はアルテラ社製の Flex10KE、 EPF10K200SRC240-3 という商品名である。
  Flex10KE というのは SRAM 方式で FPGA に書き込む込み、中規模なゲート素子を示したものである。 EPF10K200SRC240-3 の 200 はゲート数 20 万ゲートを示し、 240 というのはピンの数、 3 というのはそのデバイスのスピードレベルを示している。スピードレベルは小さい数ほど速いということを示している。

- PCI バス
  PCI バス とは Peripheral Component Interconnect bus の略であり、米国のインテル社が提唱し、多くの企業が参加して組織した PCISIG(PCI Special Interest Group) という団体が規格を決定している。 PCI は同期型のバスで 32 ビットのバス幅を持ちアドレスとデータを同じ信号線で時分割によりデータ転送を行う。最初のサイクルでアドレスを次のサイクルでデータを転送している。現在では 64 ビットの PCI バスもある。

- ISA バス
  ISA バスとは Industry Standard Architecture bus の略であり、データ幅 16bit である。最近の PC では搭載している PC が少なくなり、 PCI バスへと移行している。

- PC
  Personal Computer の略であり、本研究で使用している PC のスペックは
  CPU Intel PentiumIII 1GHz
  Memory 512MBytes
  である。また使用している OS(Operating System) は Windows98 である。

第 2 章

設計方法と使用装置

2.1 設計方法

本研究において VHDL による回路設計の流れを 2.1に示す。まずテキストエディタにより VHDL にて回路機能を記述する。本来ならば VHDL で回路機能を記述した後、入力ポートに信号を与えてシミュレーションを行う必要があるが、本研究にてシミュレーションはタイミングシミュレーションという形で回路機能をデバイスに実装する直前に行っている。

テキストエディタで VHDL を記述した後、コンパイル、および論理合成する必要がある。論理合成とは記述した VHDL の回路機能を AND 回路や OR 回路に置き換えることである。この時つくられるファイルは EDIF(Electron Design Interchange Format) ファイルと呼ばれテキスト形式である。この回路機能を VHDL 記述したファイルをコンパイル、論理合成するのに使用したツールは Exemplar 社の Leonardo Spectrum Altera Version である。この Leonardo は前年度まで使用していた Peak VHDL と比較してエラーメッセージが分かりやすい。また、階層設計を用いた際に、サブの回路を空のままトップの回路にサブ回路のポートを結ぶだけで、サブの回路を空にして論理合成することができる。それを利用することでのちの配置配線ツールを使用してその空の部分を補うことができる。

次に、論理合成して得られた EDIF ファイルをデバイスに配置配線する為に使用したツールはアルテラ社の QuartusII ver1.1 である。 QuartusII は前年度まで使用した Max+Plus2 と比較して大規模な FPGA に対応している他、 Leonardo で出力される EDIF ファイルのライブラリに対応している。また、 Max+Plus2 の後継ソフトとととして QuartusII が誕生し、現在となっては Max+plus2 は開発、生産を中止したことから QuartusII を採用した。

この QuartusII では、 Leonardo で出力された EDIF ファイルをコンパイルし、タイミングを計算し、タイミングシミュレーションを行う。コンパイルを行うとデバイスプログラミングで必

要となる sof(Sram Object File) が作られる。その後シミュレーションの結果がうまくいったら、実際のデバイスに対して配置配線を行う。

配置配線を行った後、 PC から PCI 評価基板を認識するためにドライバをつくる。ドライバを作るツールは Jungo 社の WinDriver を使用した。また作られたドライバの雛形から Microsoft 社の Visual C++ を使用してソフトウェアを作製する。この一連の流れを図 2.1に示す。

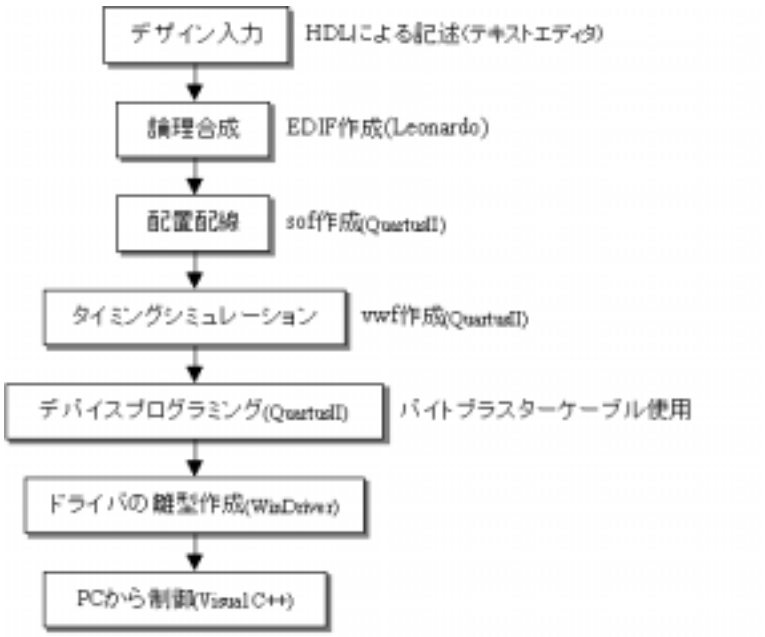


図 2.1: 本研究におけるディジタル回路設計の流れ [1]

2.2 ハードウェア

2.2.1 評価基板

本研究で使う評価基板は今年度より新たに購入された基板である。この基板は PC との通信に PCI バスを使用し、 PCI バスに PCI デバイスとして PLX 社の PCI9054 が使われている。 PCI デバイス PCI9054 のローカル側に本研究で使用する FPGA である EPF10K200SRC240-3 が取りつけられている。以下に使用する評価基板を図 2.2に評価基板のブロック図を 2.3に示す。

[1] ファイル名:u01inar/ps/flow.ps

図 2.2: PCI バス評価基板 [2]

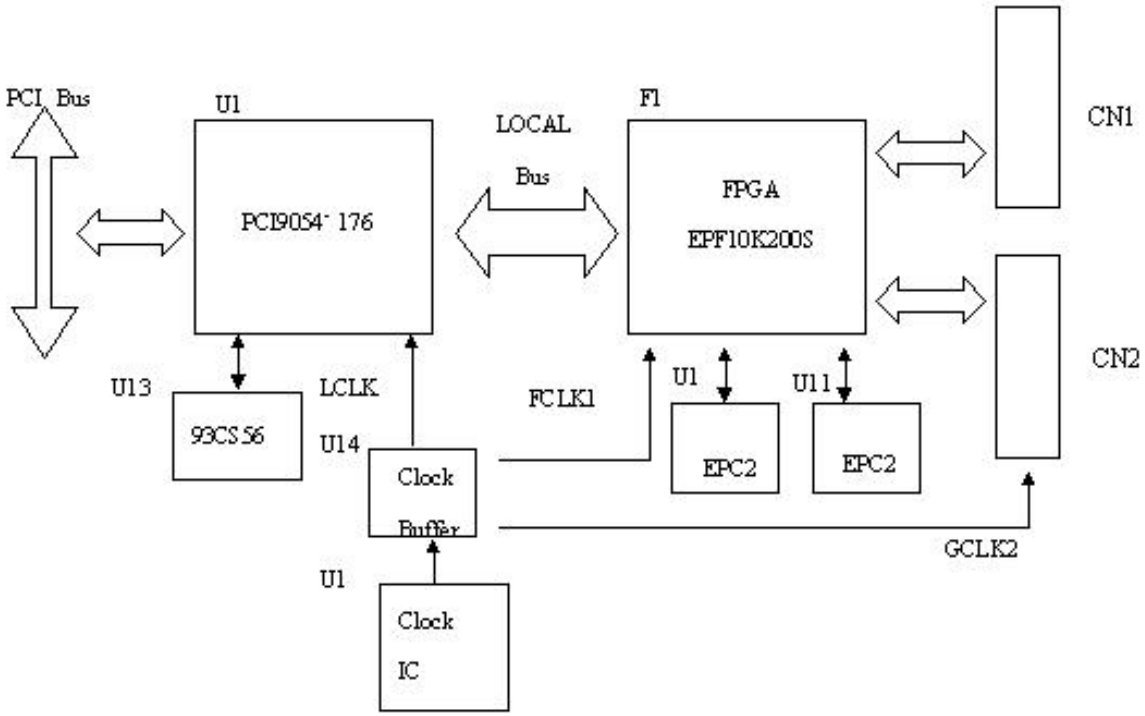


図 2.3: 評価基板ブロック図 [3]

[2]ファイル名:u01inar/ps/pci.ps

[3]ファイル名:u01inar/ps/block.ps

PCI9054 に取り付けられているクロックは 33MHz、また FPGA に取り付けられているクロックは 40MHz である。ブロック図の U1、 U11、 CN1、 CN2 はそれぞれ拡張コネクタである。 U13 は PCI9054 専用の EEPROM である。この基板にはメモリは搭載していない。そのため、設計を行うためには FPGA 内にレジスタを用意して設計を行うか、 FPGA である Flex10KE に予め EAB(Embeded Array Block) というブロックが搭載されている。この EAB を RAM として使用するマクロ関数が設計ツールに含まれている。この EAB を用いて設計を行うか、拡張バスにメモリを搭載することもできる。

本研究では FPGA 内にレジスタを用意してそのレジスタにデータを入れての演算と EAB を RAM として使用した 2 通りの方法で演算回路を設計した。

Flex10KE、 EPF10K200SRC240-3 に含まれてる EAB は 1 個あたり 4096bit(512byte) の RAM を形成することができ、この EAB が 24 個搭載されている。よって 12KByte の RAM を形成することが可能である。

- PCI バス側

  PCI バス側とは PC と PCI デバイス PCI9054 とつながっているインターフェースを PCI バス側と呼ぶ。 PCI バスはデータとアドレスを時分割で PCI9054 にデータとアドレスを転送する。

  最初にアドレスを指定し、次にデータがその信号線を通って PCI9054 へと入っていく。また PCI バスにはシングルモードとバーストサイクルモードがあり、シングルモードはその都度アドレスとデータを時分割で転送していくが、バーストサイクルモードでは最初にアドレスを指定して、以降はデータを連続して転送することができる。バーストサイクルモードを利用することで複数のデータの転送時間を短縮することもできる。

  PCI バスの転送の詳しい説明は第 3 章結果 PCI バスの理解を参照。

- ローカル側

  ローカル側とは、 PCI9054 と FPGA とを結んでいるローカル側のことをローカル側と呼んでいる。ローカル側ではアドレスとデータは 1 つのポートを時分割で使用するのではなくそれぞれのポートが与えられている。これも PCI バスと同じようにシングルモード、バーストサイクルモードに分かれており、本研究で VHDL で記述する際にはこのシングルモードとバーストサイクルモードを使い分けて FPGA に入ってくるアドレスとデータを処理する。

  ローカル側の通信については第 3 章にて詳しく述べる。

- ハイインピーダンス

  PCI9054 から FPGA 内にアドレスとデータを送る信号は双方向である。 PCI9054 から FPGA にデータが送られる時、この FPGA の入力をハイインピーダンスに設定しなければならない。 VHDL でハイインピーダンスを設定する時は'Z' で表現する。

- デバイスプログラミング
  VHDL で記述した回路をデバイスプログラミングを行い、 FPGA に回路機能をダウンロードする。この時使用している手法は JTAG(Joint Test Action Group) と呼ばれる機能を用いてプログラムを FPGA にダウンロードしている。 PC のパラレルポートからダウンロードファイル
  .sof(Sram ObjectFile) を FPGA にバウンダリスキャンによってダウンロードしていく。バウンダリとは境界のことで IC の内部と外部の境界、つまり I/O ピンの状態をスキャンしながらダウンロードを行う。

2.2.2 ハードウェアの比較

以下に前年度まで使用していた松尾 [3] が作製した ISA バスを PC との通信に利用した評価基板と PCI 評価基板との比較を表にした。

	PCI 評価基板	ISA 評価基板
データ幅	32bit	16bit
FPGA 集積度	20 万ゲート	10 万ゲート
搭載メモリ	(EAB:12Kbyte)	SRAM:512Kbyte, DRAM:16Mbyte
動作クロック	FPGA:40MHz, PCI9054:33MHz	10MHz

表 2.1: PCI 評価基板と ISA バス評価基板の比較

2.2.3 PC と評価基板の通信インターフェース

評価基板の制御と評価を行うためには PC と評価基板との通信が必要である。 PC との通信は PCI バスを使用して 32bit のデータを送ることができる。また 1 つのデータ幅 32bit のデータを１つずつ転送するシングルモードと連続して転送するバーストモードがある。

2.3 ソフトウェア

ハードウェアの性能の向上に伴い、ソフトウェアの性能も向上させる必要がある。本研究でも当初は前年度までのソフトウェアを使用していたが処理能力の向上に伴いソフトウェアを更新してきた。以下にそのソフトウェアを紹介する。

2.3.1 Leonardo Spectrum ALTERA version

VHDL をテキストエディタで記述したものを論理合成するソフトである。前年度まで PeakVHDL を使用していたが、 PeakVHDL だと階層設計において、サブ回路を空にしたままでの論理合成はできない。空のまま論理合成ができないと特に EAB を使った回路を組むことは難しい。そこで今年度よりこの Leonardo Spectrum のライセンスを取得し、このソフトウェアを使用して論理合成を行い、 EDF ファイルを作製した。
詳しい使用方法は付録 C1 参照。

2.3.2 QuartusII

この QuartusII は前年度まで使用していた Max+plus2 の後継ツールである。大規模な集積度をもつ FPGA に対してタイミングシミュレーション、デバイスプログラミングを行うために開発されたが、今では中規模、小規模な FPGA でもデバイスプログラミング、タイミングシミュレーションが行える。これは Leonardo Spectrum で作られた EDF ファイルのライブラリ全てを網羅しており、 Max+plus2 では行えないコンパイルを行うことができるため今年度よりライセンスを取得して使用した。使用しているバージョンは 1.1 である。
詳しい使用方法は付録 C2 参照。

2.3.3 WinDriver

このソフトウェアは PCI 基板を PC から制御可能にするために必要となるドライバを作製するのに使用するソフトである。このソフトを使用しようすると、ドライバに必要な雛型を簡単に作製してくれる。 PCI 評価基板を購入した (株) アリテックより薦められて採用した。
詳しい使用方法は付録 C3 参照。

2.3.4 Visual C++

このソフトは WinDriver によって作製されたドライバの雛型を用いて PC 上で PCI 評価基板を制御するソフトウェア作製するために使用した。 (株) アリテックから購入した評価基板の動作

確認するためのテストプログラムが Visual C++ で作製されていたので従来の Boland C++ builder からそのソフトウェアを採用した。

2.3.5 pc Anywhere を使った遠隔操作による評価基板の制御

PCI 評価基板は PC の内部に接続しているため、どんな PC からも制御ができるわけではない。他の PC で PCI 評価基板を制御するためには基板自体を取り付け動作確認する必要がある。その手間を省くために採用したのが、 pc Anywhere である。このソフトを利用することにより、ネットワークでつながれた PC であれば遠隔操作が可能であり、基板自体を移動することなく PCI 評価基板を制御することができる。

2.3.6 設計ツールの比較

以下に前年度まで使用してきた設計ツールと今年度から使用したツールを表にまとめた。

	設計の流れ	(今年度)PCI 基板	(前年)ISA 基板
1	VHDL 記述	テキストエディタ	PeakVHDL
2	論理シミュレーション		PeakVHDL
3	論理合成	Leonardo Spectrum	PeakVHDL
4	配置配線	QuartusII	Max+Plus2
5	タイミングシミュレーション	QuartusII	
6	デバイスプログラミング	QuartusII	Max+Plus2
7	ドライバ作製	WinDriver	
8	ソフトウェア作製	Visual C++	Builder C++

表 2.2: 設計で使用したソフトウェアの比較

表を見て分かるように、前年度まで使用してきたソフトウェアはすべて違う設計ツールに変わった。ハードウェアの進歩と共にソフトウェアの性能も進歩してきた。その進歩と共に、ソフトウェアの選択も必要である。

第 3 章

シングルモードによる FPGA への通信

3.1 目的

PCI 評価基板を用いて行列計算を行う際、まず最初に必要になってくるのが PC と FPGA との間でデータのやり取りを行うことである。本研究では、行列演算器を設計する上で必要な PCI バスインターフェース用 IC である PCI9054 の仕様を理解し、VHDL でローカル側の通信を行う回路を設計し、C++ 言語を使用して PCI バス側の通信をシングルモードで可能にする事が目的である。次に、FPAG に対して読み書きが行えた後、実際に 2 つの数の加算器を作成する。

3.2 データの読み書き

最初に PC からデータを PCI デバイス PCI9054 を介して FPGA に書き込むことができるか、書き込むことができた後その FPGA から書き込んだデータを PC から読み込むことができるかを調べる。

3.2.1 方法

シングルモードによる FPGA にたいしてデータの読み書きを行う方法を説明する。

実験方法

PC より 32bit 幅のデータ 2 つをシングルモードで評価基板上の FPGA に送り、FPGA からその値を読み込む。次に複数個データを送る。そのときにその通信時間を測定し、評価する。

作製方法

FPGA では、 PCI9054 デバイスからのローカルバスへ書き込み可能状態となったとき、データを受取り、いったん FPGA 内部に設けたレジスタにデータを入れる。またローカルバス側の通信が可能となったときにレジスタのデータを PCI9054 に送る機能をプログラムする。その後、PCI9054 の PCI バス側の設計を WinDriver 、 VisualC++ を用いて行う。

3.2.2 設計

PCI 評価基板を設計する上で重要な事に、 PCI デバイスである PCI9054 の仕様を理解することが重要である。 PCI9054 を PCI バス側と FPGA が接続されているローカル側の通信それぞれを理解する必要がある。また、 PCI9054 の PCI バスからローカルバスへと続く内部構成も理解する必要がある。

ローカル側の設計

PCI9054 のローカル側、 FPGA と接続されている方のモジュールについて説明する。 PCI9054 のローカル側に接続されている FPGA の動作を PCI9054 の仕様に合わせて設計する。 PCI9054 のローカル側のシングルモードの通信を PCI9054 Data Sheet[8]P5-44 PCI Target Single Write 、 P5-46 PCI Target Single-Cycle Read からの引用を図 3.1、図 3.2に示す。信号の後ろについている ♯ は L アクティブであり 0 になると有効である。それ以外は H アクティブであり、 1 になると有効である。

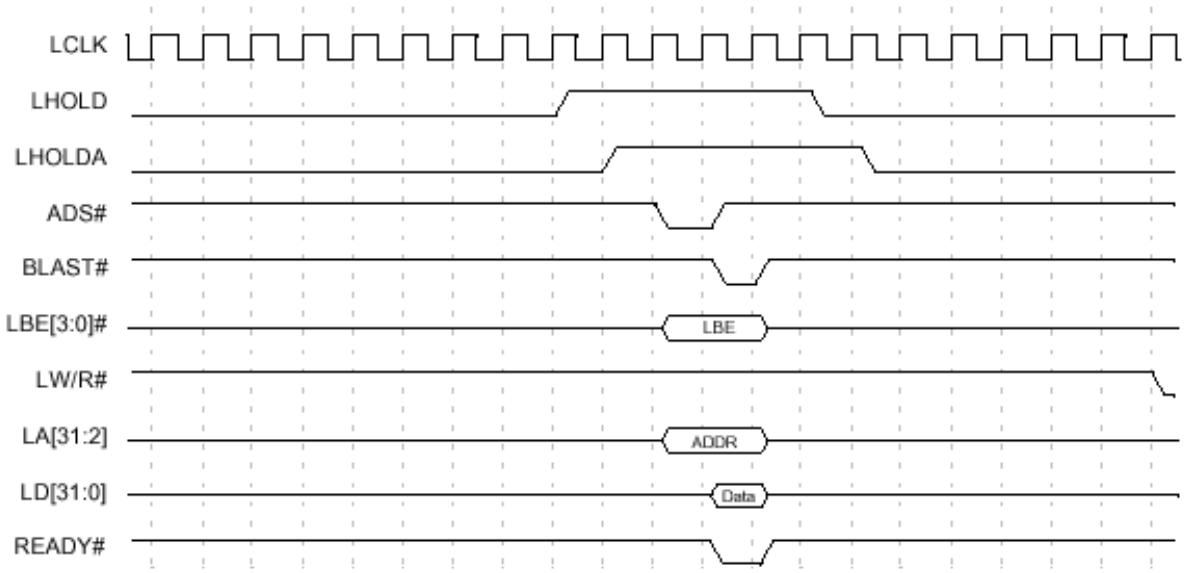


図 3.1: シングルデータの書込み [1]

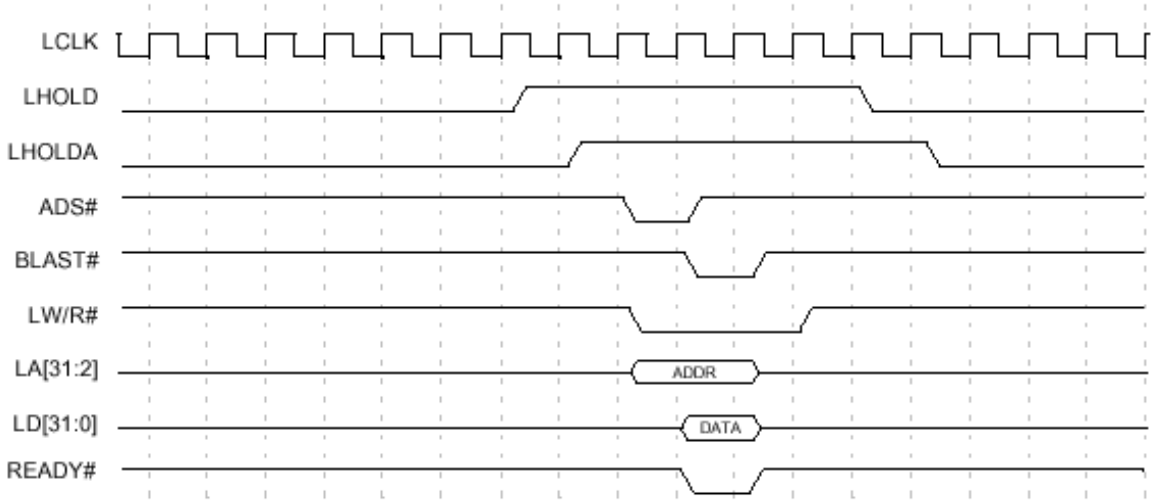


図 3.2: シングルデータの読込み [2]

それぞれ表記されている信号の役割を以下の表 3.1にまとめた。この表 3.1に表記した信号は FPGA 側に常に入ってくる信号を in とし、出て行く信号を out 、入出力信号を inout として表記している。

信号名	inout	役割
LCLK	in	FPGA に与えられているクロック信号 40MHz
LHOLD	in	ローカル側の通信可能なときに有効になる信号
LHOLDA	out	LHOLD に対する FPGA からの応答信号
ADS♯	in	この信号が有効なときにアドレスが入力される
BLAST♯	in	データ通信サイクルの終りを示す
LW/R♯	in	Write 時は 1、 Read 時は 0
LA[31..2]	inout	最高 30bit 幅のアドレス
LD[31..0]	inout	最高 32bit 幅のデータ
READY♯	out	データ通信の成功を示す

表 3.1: ローカル側の通信で最低限必要な信号

図 3.1、図 3.2の説明をする。クロック同期により、ローカル側の通信が可能になると PCI9054 はローカル側に LHOLD を有効にする。 FPGA はその応答として LHOLDA を PCI9054 に返す。

[1] ファイル名:u01inar/ps/sinwite

[2] ファイル名:u01inar/ps/sinread

ローカル側の通信が可能になると PCI9054 は ADS♯ を有効にし、アドレスの転送を行う。この時、 Write なら LW/R♯ は 1 に、 Read なら 0 になる。次に ADS♯ が無効になると同時に BLAST♯ が有効になりデータ転送が行われる。シングルモードの場合はデータが 1 個のため、 BLAST♯ と同時にデータ転送の終りを示す READY♯ も有効になる。もしこの転送が READ なら READY♯ 信号が無効になったことを確認されたら LW/R♯ を 1 にする。データ転送が行えたことが確認されたら PCI9054 は LHOLD を無効にして、その応答反応として LHOLDA を無効にする。

図 3.1、図 3.2により VHDL では状態変移マシン (ステートマシン) を使用して設計を行った。そのステートマシンを図 3.3に示す。

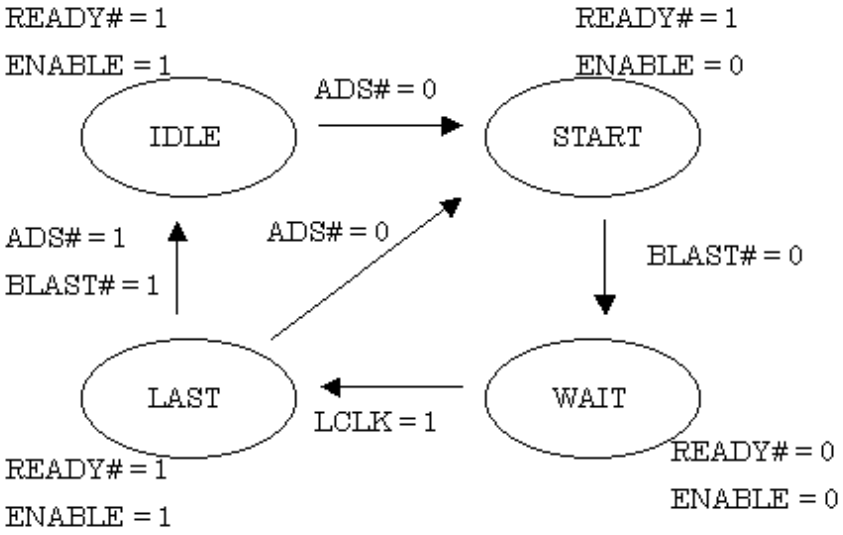


図 3.3: ステートマシン [3]

以下にステートについてそれぞれの状態についてどのような状態であるかを説明する。

- 状態 IDLE
  上のステートマシン図 3.3は状態の変化を ADS♯ と BLAST♯ で行い、それぞれの状態は IDLE が初期状態として、 READY♯、 ENABLE 信号をそれぞれ 1 にしている。ここで使用する ENABLE 信号は DATA がポート LD[31..0] を通って、 FPGA 内のレジスタに入るように制御する信号である。

- 状態 START
  状態 START は ADS♯ が有効になり、図 3.1、図 3.2ではアドレスの転送が行われるが、ここではアドレスを必要としていない。シングルモードの転送の場合、アドレスはあらかじ

[3]ファイル名:u01inar/ps/state

め考えなくてもデータが一つであるため、またレジスタにデータを格納するという考えから、アドレスは使用しなかった。もしアドレスを使用する場合では、 ENABLE 信号を利用して、ポート LA[31..2] からアドレスを FPGA 内に送ることもできる。

- 状態 WAIT
  次に BLAST♯ 信号が有効になったら状態 WAIT になる。状態 WAIT では LW/D♯ の値によってデータが FPGA に対しての入出力を行うステートである。 LW/D♯ の値が 0 なら FPGA からデータが読み込み、 1 ならデータは FPGA に入ってくる。この時、 READY♯ を 0 にしてデータ通信の成功を PCI9054 に送ってやる。

- 状態 LAST
  最後に状態 LAST にして、初期状態と同じ信号に戻す。この状態 LAST で、 ADS♯ がまだ 0 であればデータ通信が行われるため、次の状態では START。 逆に ADS♯ が 1 であれば IDLE 状態に戻る。

PCI9054 のローカル側の通信で注意しなければならないことは、データの出入りするポートである。ポート LD[31..0] は入出力で定義されている。よって出力ポートとして使用するときは ENABLE 信号によりトライステート記述しなければならない。トライステートとは ENABLE 信号が有効なときは LD[31..0] は、出力ポートとしてレジスタの値を送ることができるが、 ENABLE が有効でないときは LD[31..0] をハイインピーダンスにしなければならない。

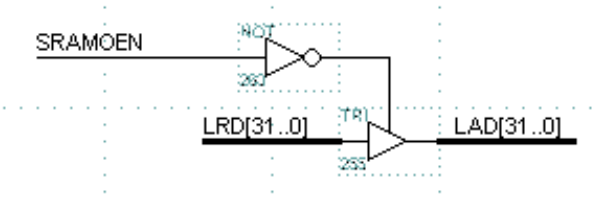


図 3.4: トライステート [4]

```
LAD <= LADOUT when EN_READ = '1'  else (others => 'Z');
```

トライステート VHDL 記述

図 3.1、図 3.2を記述したプログラムは付録 A1 に示す。

[4]ファイル名:u01inar/ps/tri

PCI デバイス PCI9054 の内部構成

PCI バス側の設計は PC からデータを送るために PC の OS に依存したドライバを作成する。ドライバの作成には WinDriver というソフトを利用して C++ 言語で記述する。
ここにその PCI9054 に対して PC からどのように PCI 9054 にアクセスしてローカル側に信号が伝わるかを PCI9054DATA BOOK[8] より説明する。 PCI9054 は動作クロック 33MHz であり、複数の内部のレジスタ、 FIFO、コントロールを持っている。 PCI9054 の一般的な接続構成と内部構成のブロック図を図 3.5に示す。 FIFO とは、データを格納したり，それを取り出して使用する場合に，格納した順番どおりに，先に格納したデータが取り出せる構成のメモリ，またはそのような記憶方法を FIFO メモリと呼ぶ。

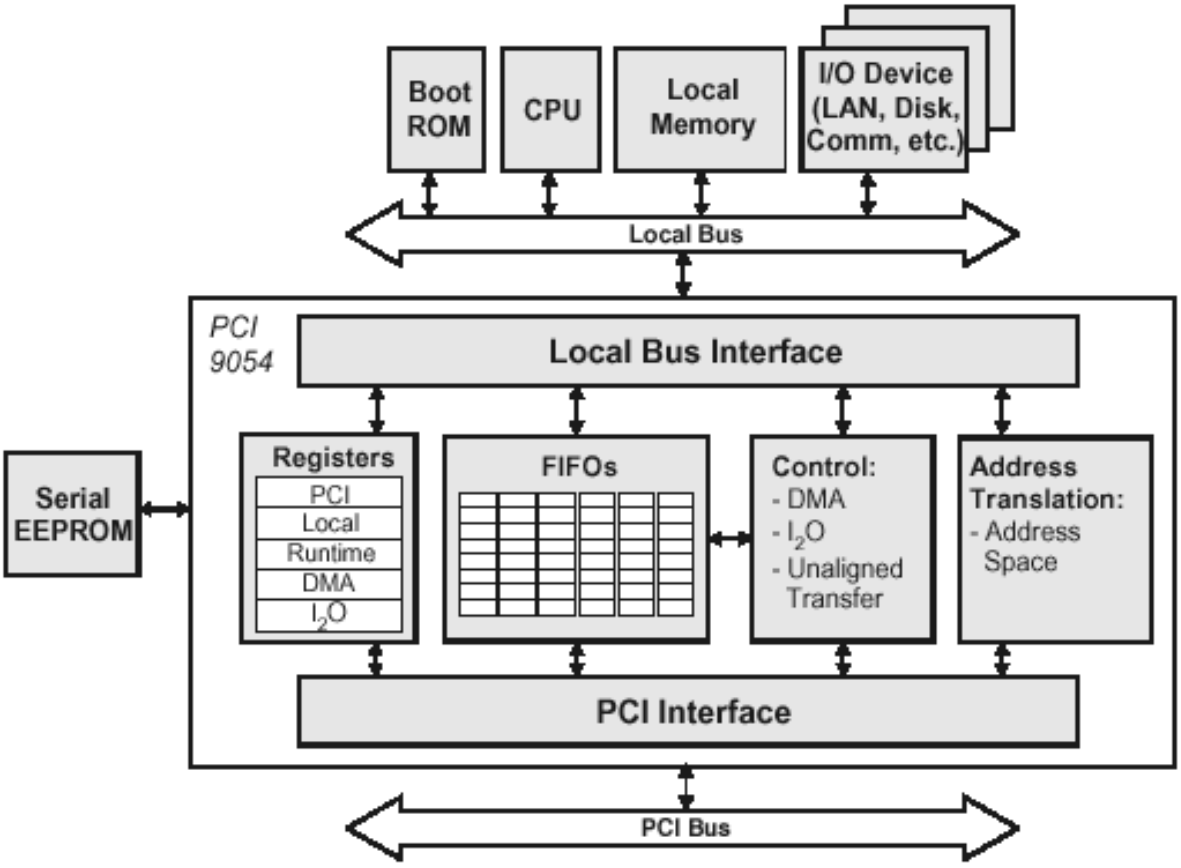


図 3.5: PCI9054 の内部ブロック図 [5]

図 3.5の説明をする。 PCI9054 の内部にはレジスタと、 FIFO、また DMA や I/O をコントロールするブロックを持っている。外付けされている Serial EEPROM はこの PCI9054 を動かすために必要な情報を予め格納することができる。本研究で使用されている評価基板にもこの Serial

[5]ファイル名:u01inar/ps/naibu

EEPROM が取り付けられている。また、ローカルバス側には様々なデバイスを取り付けることができ、それに応じて PCI9054 を動作させることができる。レジスタには I/O 空間、メモリ空間、コンフィグレーションレジスタを持っている。 PCI バスはコンフィグレーションレジスタという領域を用意し、 OS がハードウェアを認識する時にプラグ & プレイを実現する。そのために、アドレスなどをソフトウェアなどで格納できる空間としてコンフィグレーションレジスタが用意されている。 PCI9054 はコンフィグレーション時にコンフィグレーションレジスタのアドレスをローカルアドレスの先頭としてアドレスを変換する。そのコンフィグレーションレジスタを以下の表 3.2に示す。

PCI CFG レジスタアドレス	ローカルアクセス	すべての未使用ビットは0を書き込むこと bit 31 – bit 24	bit 23 – bit 16	bit 15 – bit 8	bit 7 – bit 0	PCI/ローカル書き込み可能	シリアルEEPROM書き込み可
00h	00h	デバイスID		ベンダID		N	Y
04h	04h	ステータス		コマンド		Y	N
08h	08h	クラスコード			修正ID	N	Y[31..8]
0Ch	0Ch	BIST	ヘッダ・タイプ	PCIレイテンシタイマ	キャッシュラインサイズ	Y[7..0]	N
10h	10h	PCIベースアドレス0、メモリ・マップド・コンフィグレーション・レジスタ用(PCIBAR0)				Y	N
14h	14h	PCIベースアドレス1、I/Oマップド・コンフィグレーション・レジスタ用(PCIBAR1)				Y	N
18h	18h	PCIベースアドレス2、ローカル・アドレス・スペース0用 (PCIBAR2)				Y	N
1Ch	1Ch	PCIベースアドレス3、ローカル・アドレス・スペース1用 (PCIBAR3)				Y	N
20h	20h	未使用ベースアドレス (PCIBAR4)				Y	N
24h	24h	未使用ベースアドレス (PCIBAR5)				Y	N
28h	28h	カード・バスCISポインタ(非サポート)				N	N
2Ch	2Ch	サブシステムID		サブシステムベンダID		N	Y
30h	30h	拡張ROM用PCIベース・アドレス				Y	N
34h	34h	予約			Next_Cap Pointer	Y[7..0]	N
38h	38h	予約				N	N
3Ch	3Ch	Max_lat	Min_Gnt	インタラプトピン	インタラプトライン	Y[7..0]	Y[15..8]

表 3.2: コンフィグレーションレジスタ

コンフィグレーションレジスタのレジスタは PCI バスのバス幅が 32 ビットなため、レジスタ配置も 32 ビットで、 4 バイトづつ並べてある。表 3.2はそのコンフィグレーションレジスタを示したものである。左側から PCI バス側からのアクセス、ローカル側からのアクセスのアドレスを示し、そのあとそれぞれのレジスタのビットを 1 バイトづつ示して計 4 バイト。その次は PCI バス側、ローカル側からの書き込み可能を示す Yes か No か。そして最後が PCI9054 に外付けの EEPROM に対してレジスタの書き込み可能かを Yes か No で示している。 PCI バスをコンフィグレーションするとき、このレジスタすべてを理解する必要はない。ここでは必要なレジスタについて説明する。

- デバイス ID、ベンダ ID(00h)
  ベンダ ID はPCI デバイスの製造元メーカーの ID を格納するレジスタである。 PCI9054 の製造元は PLX TECHNOLOGY であり、そのベンダ ID は 10b5 である。

デバイス ID は、デバイスの種類を示す ID であり、ベンダによって自由に決めることができる。 PCI9054 の場合は、 9054 がデバイス ID である。
このデバイス ID とベンダ ID によってメーカとデバイスが特定できるわけで、 PCI バスのプラグ & プレイを行うための重要な ID である。

- ステータス (04h)
ステータスレジスタは PCI9054 が処理中に発生したエラーの原因や信号の応答速度、そして PCI の仕様規格をサポートしているかどうかを示すレジスタである。

- コマンド (04h)
コマンドレジスタには、 PCI デバイスに割り当てられたアドレス空間や、エラー検出、拡張機能などを有効にするかどうかを設定するビットが割り当てられている。これらのビットは、リセット直後はすべてゼロクリアされる。 PC の BIOS や、 OS、 デバイスドライバなどが必要な初期化処理の後に各ビットを 1 にする。

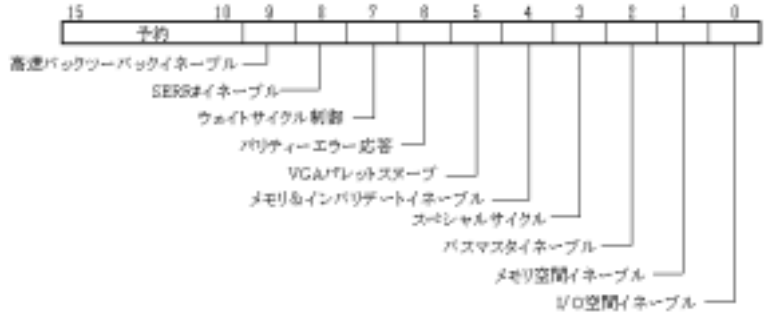


図 3.6: コマンドレジスタのフォーマット [6]

- クラスコード (08h)
クラスコードは PCI デバイス / ボードのおもな分類を示すものであり、基本クラス、サブクラス、プログラムインターフェースの 3 バイトから構成されている。このクラスコードは規格で決められている。このクラスコードは PC の OS が新しく増設された PCI ボードを認識すると、このクラスコードを読み取ってデバイスの種類と共に新しいハードウェアを認識したことを示すメッセージを表示する。その後ベンダ ID やリビジョン ID を調べて、対応しているドライバがすでに存在すればそれを自動的に読み込むがドライバが無い場合はユーザの操作を待つ。

[6] ファイル名:u01inar/ps/com

- リビジョン (修正)ID(08h)
  リビジョン ID はPCI デバイスの改変履歴やバージョンなどを表している。これもベンダが自由に規定できる。ただし、この 1 バイトの値を変えてもPCI デバイスの動作としては特に変化はない。この ID はPCI デバイスを制御するアプリケーションなどが、PCI デバイスのバージョンを識別し、それにあった最適なドライバを読み込んで制御するといった場合に用いられる。

- ベースアドレス (10h ～ 1Ch)
  ベースアドレスレジスタは、そのPCI デバイスに割り当てられた物理アドレスを保持するレジスタである。ベースアドレスレジスタは最大で 0 から 5 まで最大 6 本あり、一般的には 0 から使用する。
  図 3.7にベースアドレスレジスタのフォーマットを示す。ベースアドレスレジスタの最下位ビットは、そのベースアドレスレジスタがメモリ空間を要求しているのか、I/O 空間を要求しているのかを示す。ベースアドレスレジスタを読み出した時、このビットの値が 0 であればメモリ空間を、1 であればI/O 空間を要求している。図 3.7の (a) ではメモリ空間を要求しているため、最下位ビットは 0 であり、(b) ではI/O 空間を要求しているため最下位ビットは 1 である。
  ベースアドレスレジスタに設定されたアドレスは、メモリサイクルやI/O サイクルのアドレス転送を行う時に出力される AD バスの値と比較し、比較結果が一致した時に自分に対してのアクセスであると判定してDEVSEL♯ を有効にする。(関連項目 3.2.3 結果 PCI バスの通信参照)

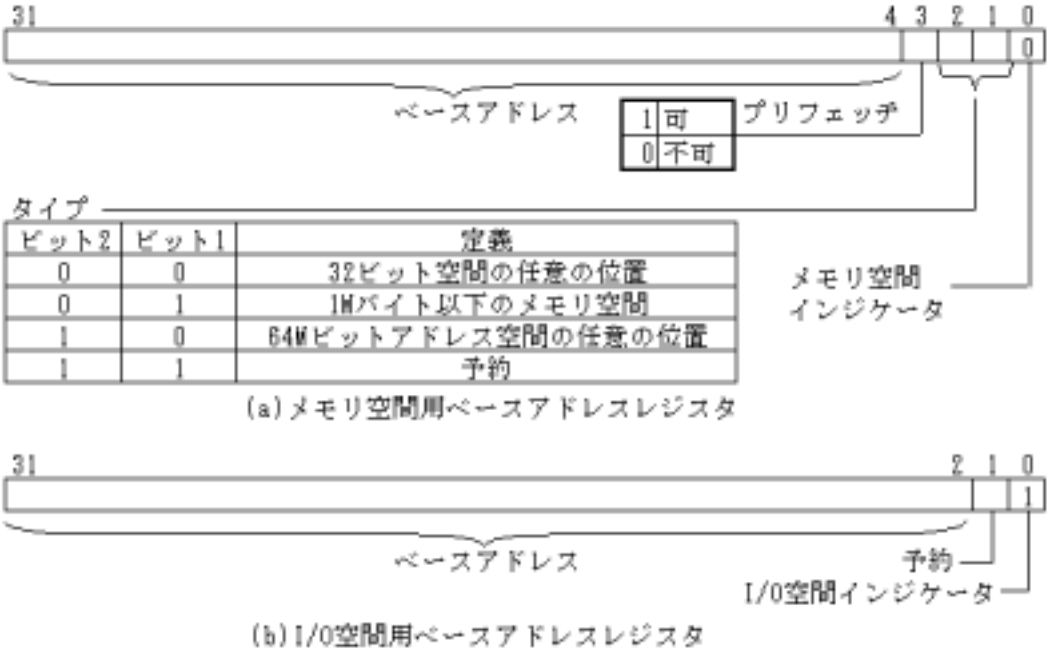


図 3.7: ベースアドレスレジスタのフォーマット [7]

必要最低限のレジスタは上の通りである。以下の図 3.44に、 WinDriver により表示される PCI9054 のコンフィグレーション空間を示す。

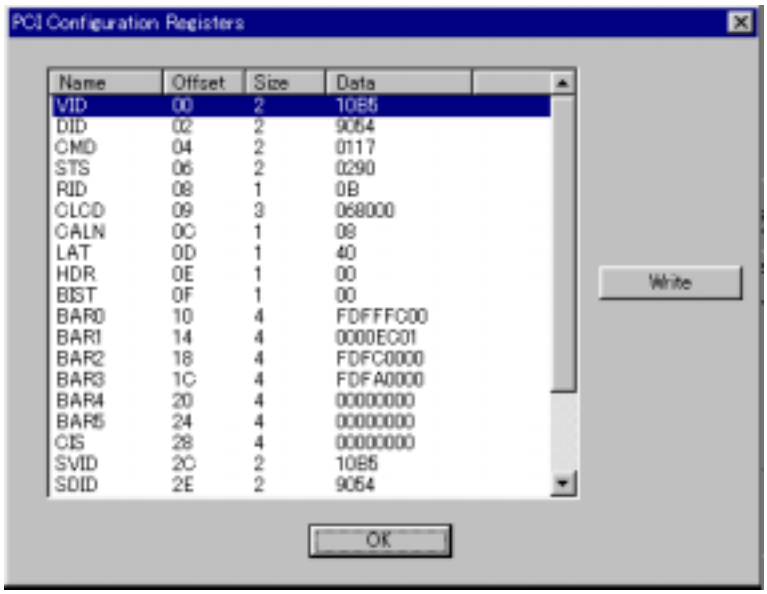


図 3.8: WinDriver により表示される PCI 9054 のコンフィグレーションレジスタ [8]

図 3.44よりベンダ ID(VID 10B5)、デバイス ID(CID 9054)、コマンド (CMD 0117)、ステータス (STS 0290)、クラスコード (CLCD 068000)、リビジョン ID(RID 0B)、ベースアドレス (BAR0

[7]ファイル名:u01inar/ps/Base

[8]ファイル名:u01inar/ps/Win

FDFFFC00,BAR1 0000EC01,BAR2 FDFC0000,BAR3 FDFA0000) の Offset と Size と Data が表示されている。 FPGA との通信にはベースアドレスを BAR0 に指定した。この BAR0 というレジスタはメモリ空間として 00 から FF まで 255 個のアドレスとデータを処理することができる。 BAR2、 BAR3 は 16 進数で FFFF 個、 10 進数では 65535 個のアドレスとデータを処理することができる。本研究ではこの PCI9054 のコンフィグレーション空間に対する処理を WinDriver というソフトを用いる。
WinDriver の使い方については付録 C3 参照。

ドライバ作成

PCI バスを使用したハードウェアに PC からアクセスするにはドライバが必要である。
WinDriver を使用してコンフィグレーション空間のベースアドレスレジスタを決め、処理を実行すると Visual C++ で処理可能な関数と共にその雛形を出力する。 PC から DATA を PCI バスに送るためにはこの雛形を使用して、どのような処理を行うことができるか、設計する必要がある。以下にその設計について説明する。

- ボードのオープン
  プログラムの起動時にボードをオープンする。オープンに成功するとボードアクセス用のハンドルが変えるので、それを変数に保存して以後のアクセスに使用する。

- レジスタの読み書き
  ダイアログボックスの read/write ボタンをクリックすると関数により書き込みと読み込を行う。この読み込んだ値をさらにダイアログボックスに表示する。

- ボードのクローズ
  プログラムの終了時に関数によりボードをクローズする。

3.2.3 結果

FPGA への読み書き

読み書きが行えた PCI9054 のローカル側の MaxPlus2 でのシミュレーション結果を図 3.9に示し、 Visual C++ で行えた読み書きのダイアログボックスを図 3.10に表示する。

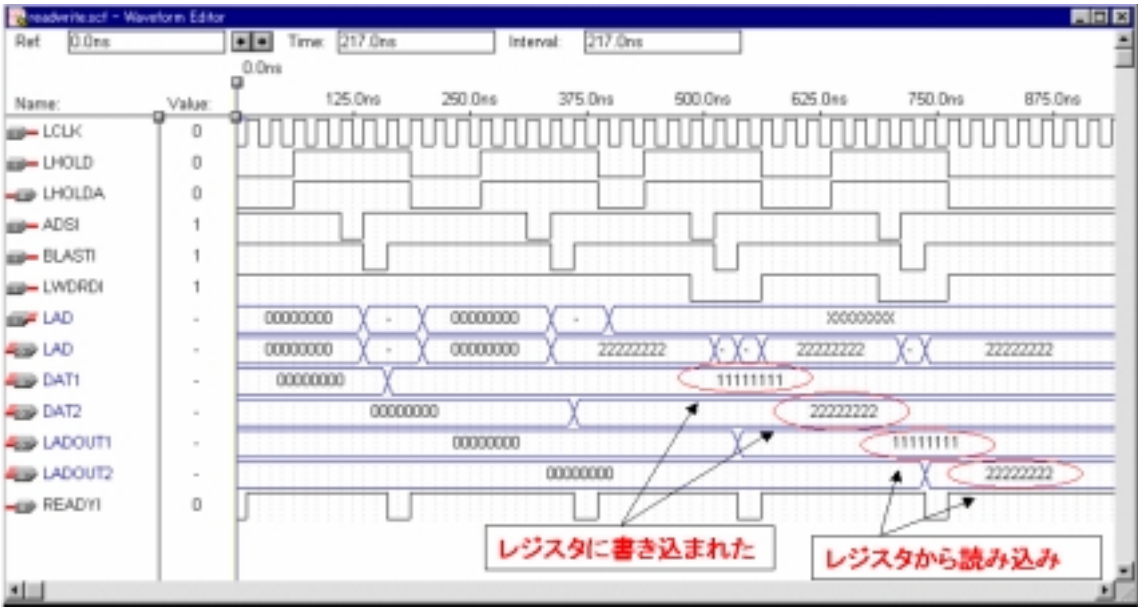


図 3.9: ローカル側のシミュレーション結果 [9]

シングルモードにおいて、ローカルバスが有効になっている時間はローカルバスの使用を示す LHOLD が 1 になって有効になったら LHOLDA が応答する。その後 LHOLD が 0 になって、その応答として LHOLDA が 0 になる。その間 LHOLD が 0 になってから LHOLDA が 0 になるまでの間がローカルバスが使用されている時間である。シミュレーションの結果よりその間の時間は 8 クロックであり 33MHz の動作クロックでは 240ns である。

図 3.10: ダイアログボックスの結果 (付録 B1) [10]

ダイアログボックスの値は 10 進数でダイアログボックスに書き込んだ値を 16 進数に変えて表示している。

PCI バスの通信

設計する上では、PCI バス側の通信を直接用いて設計する必要はない。しかし、PCI バスを理解する上では PCI バス側の通信を理解する必要がある。PCI9054 DATA BOOK[8] より、PCI

[9] ファイル名:u01inar/ps/simf

[10] ファイル名:u01inar/ps/diac

バスの通信について説明する。 PCI バスには I/O 空間とメモリ空間そしてコンフィグレーションレジスタがある。
PCI バスの動作は、メモリリード、メモリライト、 I/O リード、 I/O ライト、コンフィグレーションリード、コンフィグレーションライトと 6 つのサイクルがある。このサイクルにあわせて、 PC と PCI バスの通信を行うのに最低限必要な信号を表 3.3に示す。信号が 0 で有効となる信号には ♯ を付けて示している。また信号は PCI9054 に入ってくる信号を in とし、出て行く信号を out としてリードサイクル、ライトサイクルに分けて表記している。

信号名	名称	リードサイクル	ライトサイクル
CLK	クロック	in	in
RST♯	リセット	in	in
AD[31..0]	アドレス / データバス	out	in
C/BE♯	バスコマンド / バイトイネーブル	in	in
PAR	パリティ	out	in
FRAME♯	フレーム	in	in
IRDY♯	イニシエータレディ	in	in
DEVSEL♯	デバイスセレクション	out	out
TRDY♯	ターゲットレディ	out	out
STOP♯	ストップ	out	out
IDSEL	ID セレクト	in	in

表 3.3: PCI バス側の主な信号

表 3.3で表記した信号の役割を説明する。

- CLK
  PCI バスの動作の基準となる信号。 33MHz で動作する。リセット信号以外の信号は全てこの信号で動作する。

- RST♯
  システムの電源投入時またはシステムリセット時に、有効となる。 PCI バスのリセット信号である。

- AD[31..0]
  データバスとアドレスバスは、時分割でこの 32 本の信号線を使う。

- C/BE[3..0]
  この信号も時分割で使われ、アドレスが転送されている時はバスコマンドとして、データが転送されている時はバイトイネーブルとして動作する。

- FRAME♯
  転送を開始する時に有効になる。連続データ転送中は有効になり続ける。

- IRDY♯
  デバイスがデータ転送可能状態にあるときに有効となる。

- DEVSEL♯
  アクセスを受けたデバイスが有効となり、転送が完了するまで有効となる。

- TRDY♯
  デバイスがデータ転送可能状態にあるとき有効になる。IRDY♯ と TRDY♯ が有効になっている時データの転送が行われる。

- PAR
  PCI バスにはパリティがあり、AD[31..0] と C/BE[3..0] の合計 36 本のうち、“1” が立っているビットが偶数なら “0”、奇数なら “1” を出力する。

- STOP♯
  転送を中断してもらう時に有効となる信号である。

- IDSEL♯
  コンフィグレーション空間のアクセスには、各信号がぶつからないように制御している信号。

ここでは、6 つのサイクルの 1 例として、コンフィグレーションサイクルのリード、ライトについて説明する。コンフィグレーションリード、ライトのタイムチャートを図 3.12、図 3.11 に示す。

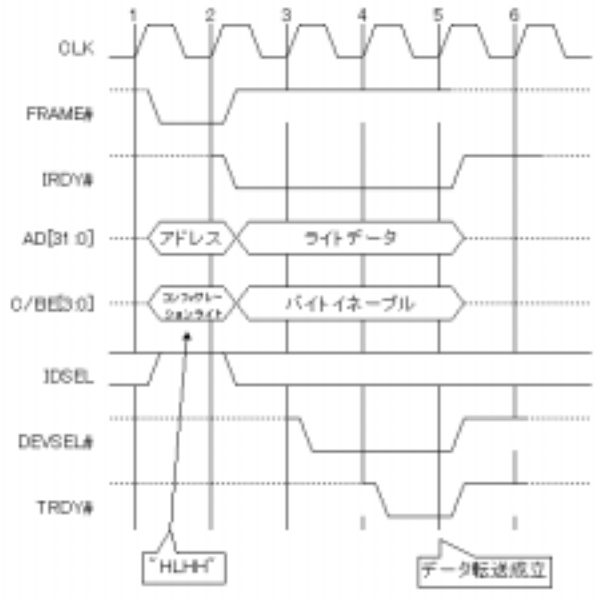


図 3.11: コンフィグレーションライトサイクル[11]

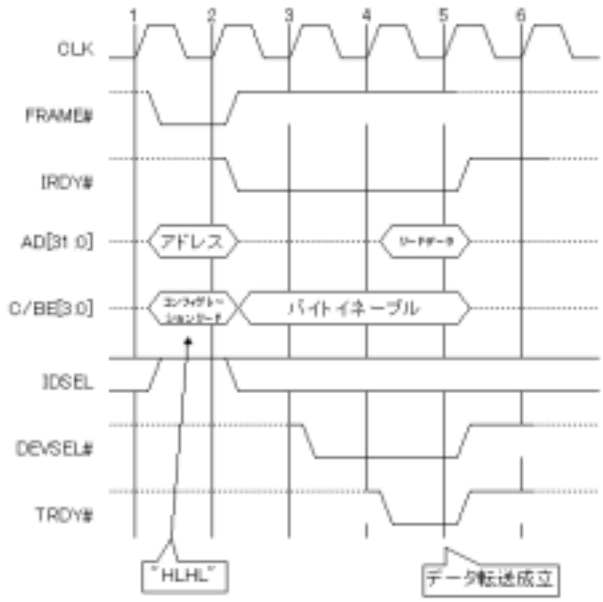


図 3.12: コンフィグレーションリードサイクル [12]

コンフィグレーションライトサイクルの動作をクロックごとに説明する。

- クロック 1
  FRAME♯ と IRDY♯ のどちらも有効になっていないのでアイドル状態を示す。ここで PCI バスのボードがオープンされると AD バスにアドレスが、C/BE バスにバスコマンドが出力され FRAME♯ を有効にする。この時、コンフィグレーションサイクルであれば IDSEL♯ の値が 1 になっていることを確認する。

- クロック 2
  FRAME が有効になるとアドレスが AD バスにあると認識され、C/BE の値をバスコマンドとして認識される。また C/BE のバスコマンドより、アドレスがライトなのかリードなのかを調べる。これは瞬間的な動作ではなく次のクロックまで待つ。そして FRAME♯ を無効にして、データが出力される準備が整い、IRDY♯ を有効にする。

- クロック 3
  デバイスにアドレスが送られてくるとライトかリードに対応できるので、DEVSEL♯ を有効にする。しかし、データが瞬時に受け取る準備はできていないので TRDY♯ は有効にしない。

[10] ファイル名:u01inar/ps/pcicw.ps

[11] ファイル名:u01inar/ps/pcicr.ps

- クロック 4
  DVSEL♯は有効になっているが、 TRDY♯が有効になっていないのでデバイスは待つ。デバイスはデータを受け取る準備ができたので TRDY♯を有効にする。しかし、 4 番目のクロックでは TRDY♯をまだ認識していない。認識するのはクロックの立ち上がり時である。ここでライトの時は IRDY♯が有効になった時に、データは常に出力しているが、リードの時は TRDY♯が有効になった時にデータの用意ができ、出力される。

- クロック 5
  デバイスは TRDY♯が有効になっているのを確認して、デバイスがデータを受け取ったことを示す。データ転送が成立したとして IRDY♯を無効にする。そして、 IRDY♯が無効であるので、データ転送が成立したとして、 TRDY♯、 DEVSEL♯が共に無効になる。

- クロック 6
  ここで常に出し続けていた信号をハイインピーダンスにする。図 3.12、図 3.11 では、点線の信号線がそのハイインピーダンスを示す。

3.2.4 結論

結果で示した通り、 PCI 評価基板の FPGA に対して通信が行うことができた。今回 FPGA 上の通信には、 PCI9054 の仕様で決められているシングルモードの Write Read の通信を行った。このシングルモードでは １つのデータを転送するときに使用する。仮にデータを複数個送る時にはそのデータ１つ１つについてシングルモードで転送する。シングルモードで転送するときはデータがローカル側に送られる度に LHOLD に対する応答信号 LHOLDA を有効にし、 ADS♯を有効にするため、データが送られるまで 3 クロック分の時間がかかる。一方、データを連続して転送するバーストサイクルモードでは最初に ADS♯を有効にした後は連続的にデータを転送する。よってバーストサイクルモードでは最初に LHOLD に対する応答 LHOLDA を返し、 ADS♯を有効にする 3 クロックの後は 1 クロックごとにデータを転送していく。行列の演算を行う上でデータの転送をより速く行う上ではバーストサイクルモードで転送を行うことが必要となってくる。次章でバーストサイクルモードについて説明する。

3.3 シングルモードでの加算

3.3.1 方法

実験方法

PC からデータをシングルモードで FPGA 内に 2 つ用意したレジスタに格納する。送るデータは 32bit である。レジスタにデータが格納されたことが確認されたら、今度は FPGA 内で加算を行う。その結果をまた PC にシングルモードで返す。

3.3.2 設計

- ローカル側の設計
  データの転送はシングルモードで行うため、データの転送については 3.2　で述べたシングルモードの通信を使用した。今回のモジュールではデータをシングルモードで 2 個のデータを送り、送ったデータを 2 つのレジストリに格納する必要がある。その 2 つのレジストリに分ける方法について述べる。

  - 2 つのレジストリにデータを分ける
    2 つのレジストリにデータを分けるのに使用した方法はシングルデータの転送に必要な信号 ADS を使用した。(3.2 参照)ADS はデータの転送を開始する際 1 から 0 になる信号である。この信号を利用してカウントを数える。つまり用意した内部信号 COUNT を ADS が最初に 0 になった時を 1 に次に ADS が 0 になったときを 2 にといった形で ADS のイベントにあわせて FPGA 内に用意した内部信号に 1 づつ足していくことにより送られてきたデータの区別をつけることができる。
    ちなみにポート LAD を入力ポートとして使用するか、出力ポートとして使用するかを区別する方法もこの COUNT という内部信号を用意した。今回の FPGA 内外からポート LAD に対してのアクセスは 3 回ある。最初の 2 回は PCI9054 から送られてくるデータの入力ポートとして使用し、3 回目は足し算の結果を PCI9054 に送る出力ポートとして使用した。 COUNT を使用して出力ポートとして LAD を使用する時は ADS のイベントに同期した CONT が 3 になった時に出力ポートとして LAD を定義した。

  加算器については、シングルモードを利用してデータを FPGA 内のレジストリに格納した後、そのレジストリの和を取り読み出しを行う (ポート LAD からデータを送る) レジスト

リに格納する。和を取るプロセスは今回は 32bit 幅のデータを整数として和を取った。
加算器のプロセスについては付録 A2 参照。

- PCI バス側の設計
  PCI バス側の設計は WinDriver というソフトを使用して、 PCI9054 に対しての書き込み読み込みを行うコンフィグレーション空間の設定をおこなう。今回使用ししたコンフィグレーション空間の BAR(Base Address Register) は BAR0 である。この BAR0 は 3.2 でも述べたが 16 進数で FF 個、 10 進数で表すと 65535 個のデータとアドレスを処理することができる。

3.3.3 結果

FPGA のタイミングシミュレーションの結果と足し算を行ったダイアロゴボックスの結果を図 3.13、図 3.14と図 3.15に示す。

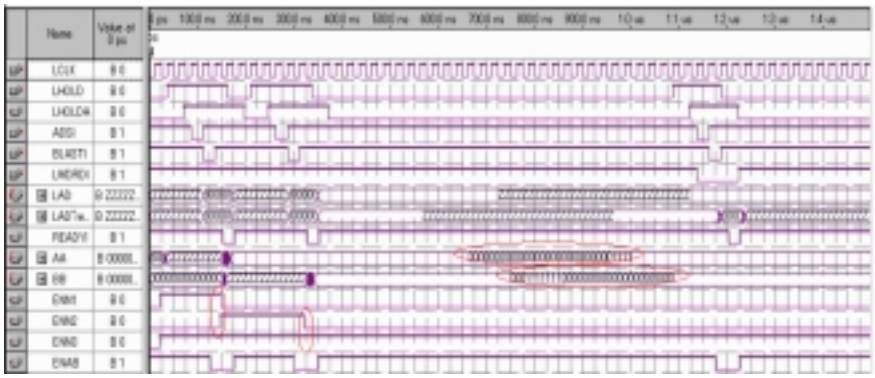


図 3.13: シミュレーション結果データ書き込み部 [13]

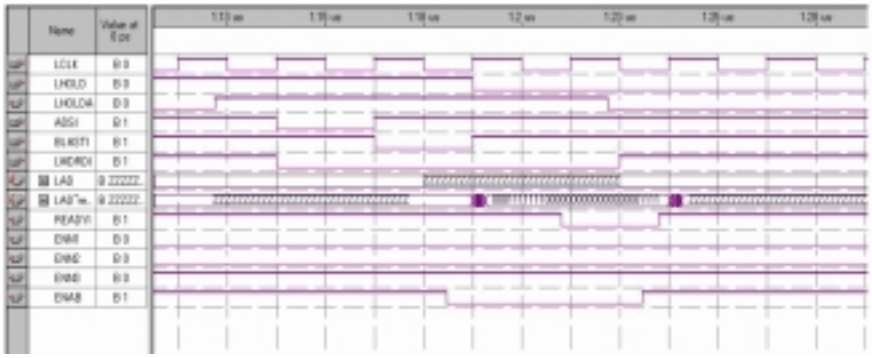


図 3.14: シミュレーション結果データ読み込み部 [14]

[13]ファイル名:u01inar/ps/ksnsimw

[14]ファイル名:u01inar/ps/ksnsimr

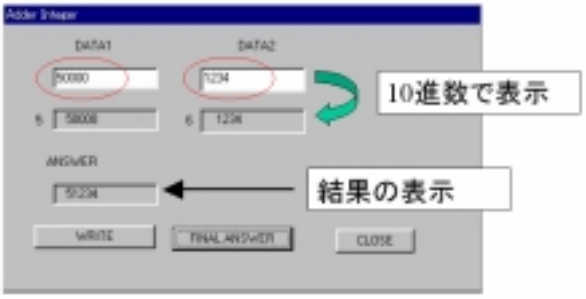


図 3.15: 加算器のダイアロゴボックス [15]

図 3.13はデータの入力部を示している。データが FPGA 内のレジスタに入る部分である。入力ポートとしての LAD からデータが送られ、 COUNT 信号により加算された値により信号 EN1,EN2 がそれぞれ有効になっている。 EN1 が有効なときは LAD からのデータは DATIN1 へ、 EN2 が有効な時は DATIN2 へとデータが格納される。このシミュレーション上では DATIN1 へのデータは”00000000000000000000000000001111” であり、 DATIN2 へのデータは ”00011111110000000000000000000000” である。
図 3.14ではデータの読み出しを行ったシミュレーション結果である。この時読み出すデータの値は足し算の結果である。よって DATIN1 と DATIN2 の加算した結果 ”00000000000000000000000000001111” +”00011111110000000000000000000000” = 00011111110000000000000000001111”
の値を読み出す。図 3.14ではポート LAD に値が性格に出力されたことを示している。
図 3.15では実際に PC からデータを送り、その結果を PC の結果に送っている。図 3.15で使用した数は 50000 と 1234 である。 ANSWER を見ると 51234 と足し算の結果が表示された。これにより、 PC から FPGA 内にデータを送り簡単な演算が行うことができることが分かった。

3.3.4 結論

2 つのデータを FPGA に送り、その和の結果を PC に表示されることはできた。このことは FPGA 内での演算を行うことができるということが示せた。しかし、転送を開始する信号 ADS に同期し、そのカウントを取り、レジストリを区別する方法だとデータの数を増やしていくとそのデータの数だけレジストリを用意し、 FPGA が出力するときもそのデータの個数に合わせたレジストリの数を用意しなければならず、未知数のデータの演算を行う時にその応用範囲は狭い。行列演算回路として機能させるためには複数個のデータのやり取りが必要となってくる。この方法ではシングルモードでのデータの読み書きは演算を行う上でも効率的とは言えない。次章でバースト

[15] ファイル名:u01inar/ps/kasandia

サイクルモードについて述べるが演算を行う上でもこのバーストサイクルモードを使用する必要がある。

第 4 章

複数のデータの転送

4.1 目的

PCI バスを利用したデータの高速転送を実現するため、バーストサイクルモードでデータのやり取りを行う。また、行列演算回路として設計するために、まず簡単な演算を行い結果を PC で読み取る。

4.2 バーストサイクル

PCI9054 DATA BOOK[8] よりバーストサイクルのタイムチャートを示す。

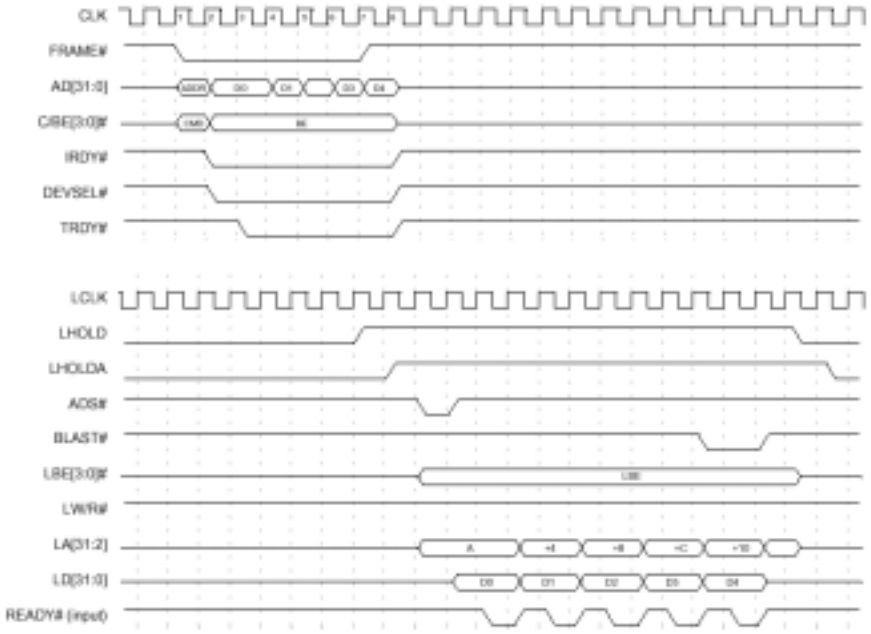


図 4.1: ローカル側のバーストライト [1]

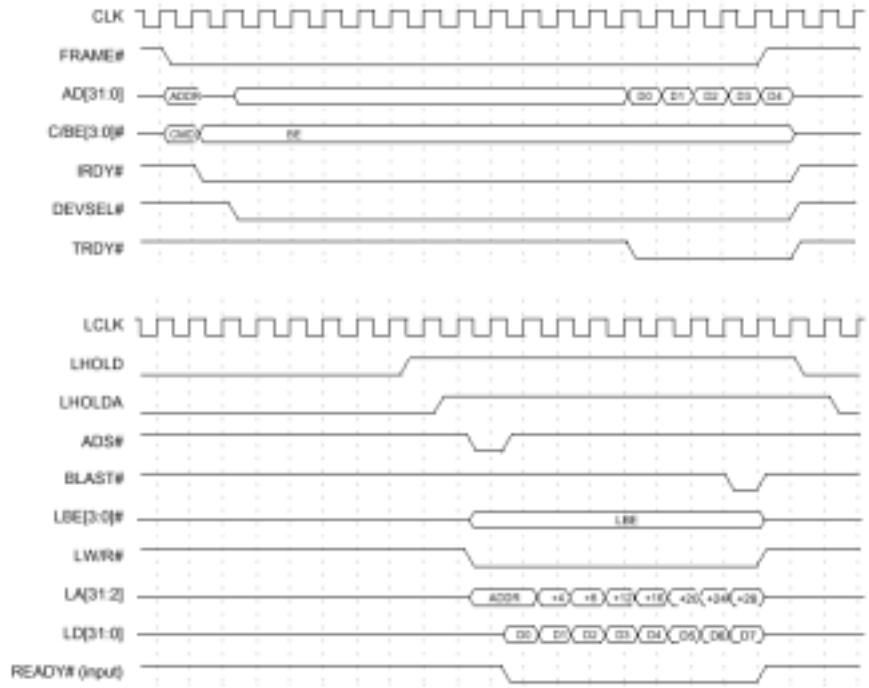


図 4.2: ローカル側のバーストリード [2]

図 4.2、図 4.1の説明をする。

- PCI バス側
  図 4.2、図 4.1 の上段の図が PCI バス側のタイムチャートである。 PCI バス側の信号の動作は、第 2 章で説明した図 3.12、図 3.11 とほぼ同様の信号の動作を示す。データは IRDY♯、TRDY♯ が同時に有効になったときにデータの転送が行われる。

- ローカル側
  FPGA 側から見た信号の動作について説明する。図 4.1、図 4.2は PCI9054 から見た信号であるため、 READY♯ が input になっている。がここでは output として説明する。
  図 4.1、図 4.2より下段の信号がローカル側の信号である。

  – バーストライト
    バーストライトの転送では、最初の 3 クロックまではシングルモードと同じである。図 4.1ではポート LA より入ってきたアドレスは 4 づつ加算されていく。そのアドレスとデータは同じタイミングでポート LD から入力されている。データが転送されると

[1] ファイル名:u01inar/ps/burstw

[2] ファイル名:u01inar/ps/burstr

READY♯が”0” になり有効になる。その後、データが転送されると、 READY♯は”1” になり、データ転送の成立を示す。この READY♯は１つ１つのデータが入力されると、その度に”0” になりデータ転送の完了を示す為に”1” を返す。
このようにしてデータが FPGA 内に入力されていく。また、最後のデータが入力されると同時に、 BLAST♯が有効になる。これは、最後のデータの転送であることを示す。この転送が完了すると LHOLD を”0” にしてローカル側の転送が終了される。

– バーストリード
バーストリードとバーストライトの違いは READY♯信号の違いである。ポート LA にアドレスが出力されると、データが出力される準備ができる。そのときに READY♯が有効になり、最初のデータが出力され、転送が行われる。データが転送され続けている間はデータとアドレスは同じタイミングで出力され、同時に READY♯も有効である信号”0” を出力し続ける。その後、最後のデータが出力されると同時に BLAST♯が有効になり、データの最後を示す。後は、バーストライトと同様 BLAST♯が”1” に戻ると LHOLD が無効になり LHOLDA を返し、データの転送が終了する。

4.3 バーストサイクルモードでの FPGA へのデータの読み書き

4.3.1 方法

実験方法

図 4.2、図 4.1 を利用して実際に FPGA に対してデータの読み書きを行う。 PC より複数個のデータをバーストサイクルモードでデータを送り、その送ったデータを再度 PC 側で読み取る。

作製

FPGA では、 PCI9054 から送られて来たデータを FPGA 内部にバーストサイクルモードで転送する。この時送られてきたデータは FPGA 内の EAB(Embeded Array Block) に格納する。EAB を一時的なメモリとして使用し、その EAB から書き込んだデータを PC に返す。

EAB について

EAB について説明する。 EAB は ALTERA 社の FPGA の製品の 1 つである Flex10K シリーズに搭載されているマクロ空間である。このマクロ空間 を RAM や ROM、 FIFO として使うことができる。本研究で使用している FPGA にはこの EAB が 24 個取りつけられており、 1 個の

EAB の容量は 512Byte である。このマクロ空間をメモリとして使用するときは、 VHDL で記述するときは階層構造設計を利用してマクロの宣言するモジュールを用意し、配置配線ツールで具体的な設定をする。 (詳しくは付録参照)
今回は、この EAB を ALTERA 社が独自に設定している LPM_RAM_DQ を使って、 RAM として EAB を使用した。

- LPM_RAM_DQ

図 4.3: LPM_RAM_DQ
[3]

LPM_RAM_DQ は、入力と出力を単方向で指定することができる。データとアドレスの幅は 32bit まで指定することができ、本研究 では 8bit のデータ、アドレス幅を持つ EAB を 4 個使用した。こうすることで、 2KByte のメモリとして利用することができる。
図 4.3 では、 inclock とは LCLK のポートを、 Data は LAD のポートと結んでいる。 Address はポート LA から結んでいるが、読み込の時に分かりやすいように、最初のアドレスを LA から読み込み、以後は 1 づつ足していった。 WE 信号 とは Write Enable 信号であり、書き込むときは常に”1” でなければならない。以下に ALTERA 社からのデータブックより EAB の仕様についてそのタイムチャートを示す。 LPM_RAM_DQ の使用方法は付録 C2 参照

[3]ファイル名:u01inar/ps/lpm

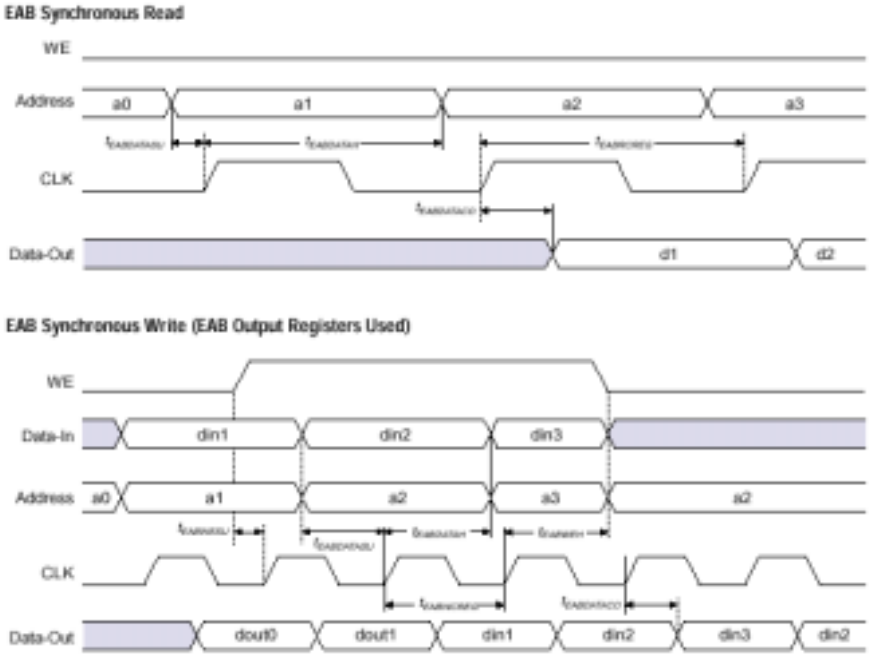


図 4.4: EAB の同期タイミング波形、 dsf10ke.pdf p62 [4]

4.3.2 設計

ローカル側の設計は図 4.2、図 4.1 より、 VHDL でデータの入力制御を行い、 FPGA 内に用意した EAB (LPM_RAM_DQ) にデータを格納する。データを読み取る場合はその格納した LPM_RAM_DQ のアドレスポートに ”1”づつ足していってデータを呼び出し出力ポートに送る。PCI バス側ではコンフィグレーションレジスタを使用して WinDriver、 VisualC++ を使用してドライバならびに、ソフトウェアを作成する。

ローカル側の設計

ローカル側の設計は FPGA に対して、 VHDL で記述する。
reffigburstr、図 4.1 より、入力信号、出力信号の定義を行う。信号の定義は第 3 章シングルデータの読み書きと同様である (第 3 章、表 3.1 参照)。バーストサイクルライトの時は ADS♯ が有効になると同時にアドレスが転送される準備ができ、次のクロックでデータが入力される。そして次のクロックで READY♯ を”0” にしてやり、次のクロックで READY♯ を 1 に戻してやる。こうして 1 つのデータが転送できた。このサイクルを繰り返して、１つ１つのデータの転送を行う。そして最後に BLAST♯ が有効になり、”1” に戻ったところでデータ転送終了を示す。バーストサイクルリードでは ADS♯ が有効になると同時にアドレスが転送される準備ができ、次のクロック

[4]ファイル名:u01inar/ps/eab

でデータが入力される。そして次のクロックで READY♯ を”0” にしてやり、 BLAST♯ が有効になり、”1” に戻ったところでデータ転送が成立する。この際バーストライトとは異なり、データ転送中は READY♯ は”0” のままである。この一連のデータのやり取りを状態変移マシン (ステートマシン) にしたものを図 4.5に示す。

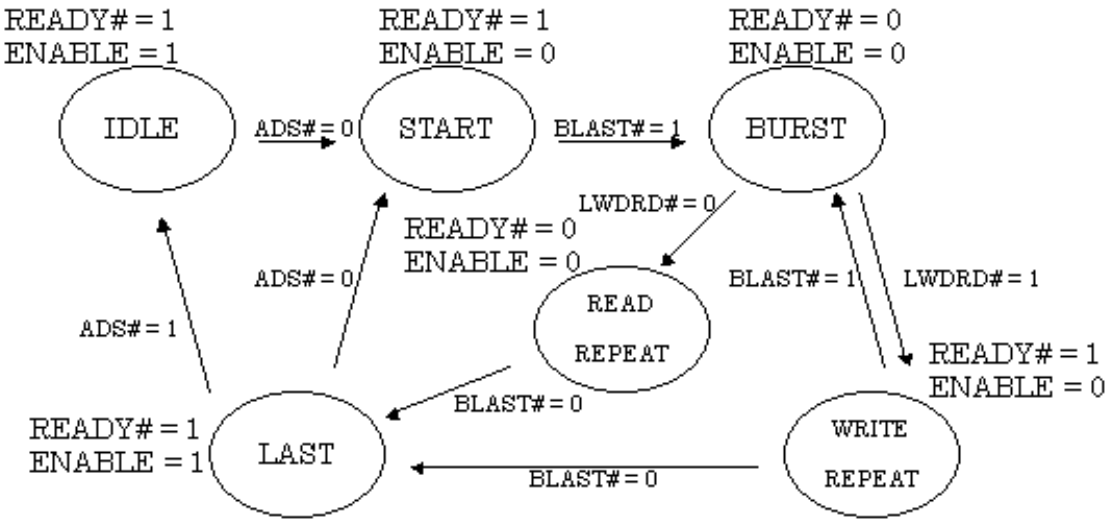


図 4.5: バーストサイクルモードの状態マシン [5]

このステートの説明をする。
用意するステートは IDLE、 START、 BURST、 WRITEREPEAT、 READREPEAT、 LAST の 6 つの状態である。 IDLE、 START、 LAST の 3 つの状態は第 3 章で述べた動作と同じである (第 3 章 3.3.1 ローカル側の設計)。よってここでは状態 BURST、状態 WRITEREPEAT、状態 READREPEAT について説明する。

- 状態 BURST
  状態 START においてもし BLAST♯ が “1” なら状態 BURST に移る。状態 BURST では、データを入出力できる状態であることを示すステートである。ここでは、 READY♯ を “0” にしてやる。この時、 FPGA 内に用意した ENABLE 信号を “0” にしている。 ENABLE 信号はデータの入出力を許可する信号であり、この信号が “0” であるときはポート LAD は入力ポートか出力ポートとしてデータを入力する。

- 状態 BURSTREPEAT
  書き込み時は、図 4.1 より、データが転送されている間、 READY♯ は常に “0” ではないの

[5] ファイル名:u01inar/ps/statew

で状態 BURST になった次のクロックで LWDRD♯ の値が “1” なら状態 WRITEREPEAT に状態を変える。状態 WRITEREPEAT は READY♯ を”1” にしてやり １つのデータが書き込まれたことを示す状態である。データが複数個存在する場合はこの状態 BURST、状態 BURSTREPEAT の行き来を繰り返してバーストサイクルモードを実現させる。その後、状態 WRITEREPEAT で BLAST♯ が “0” になったら状態 LAST に行き、シングルモードと同様にステートを変化させる。

- 状態 READREPEAT
  状態 BURST で LWDRD が “0” ならクロックに同期して、状態 READREPEAT に移る。状態 READREPEAT は複数個のデータが転送されている状態を示し、 BLAST♯ が有効になる “0” になったら LAST に移る。データが N 個あったら N-1 個のデータ転送がこのステートで行われる。

PCI バス側の設計

PCI バス側の設計は、 WinDriver を使用してまず、使用する PCI9054 のコンフィグレーション空間を指定する。この研究で使用した BAR(Base Address Register) は BAR2 である。 BAR2 は。 BAR2 は 16 進数で FFFF 個、 10 進数では 65535 個のアドレスとデータを処理することができる。この BAR2 を使用してデータの書き込みを行うドライバを作成した。

4.3.3 結果

結果の図 4.6は、データを 100 個送ってその読み込んだ値を 16 進数で表示したものである。これにより、複数個のデータの読み書きが行えたことを示す。

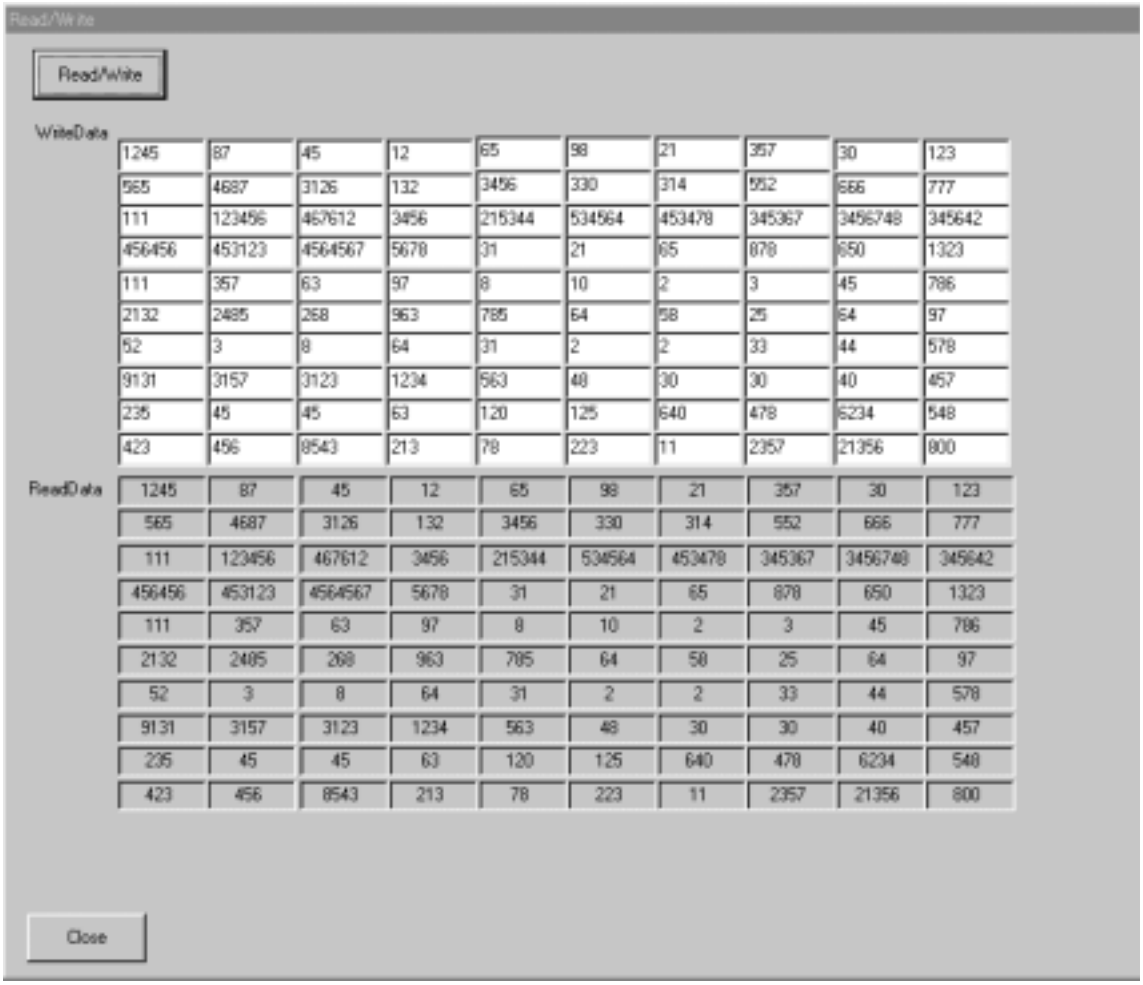


図 4.6: データを 100 個送ったプログラム [6]

4.3.4 結論

バーストサイクルモードでの FPGA へのアクセスができた。送ったデータは 100 個であり、その書き込みとよ見込みは瞬時のうちに行うことができた。正確な時間までは計測していないが、行列演算器を作成するためにはこのバーストサイクルモードを利用した方がデータの転送にかかる時間は速い。

この研究では EAB を 4 個使用している。そのため 1 つの EAB が 4096bit のメモリとして使用できることはすでに述べた。4096bit のメモリ 4 個で 32bit 幅のデータは 512word 送ることができる。次にデータを 512 個 FPGA に読み書きした結果を図 4.7表示する。

[6] ファイル名:u01inar/ps/100

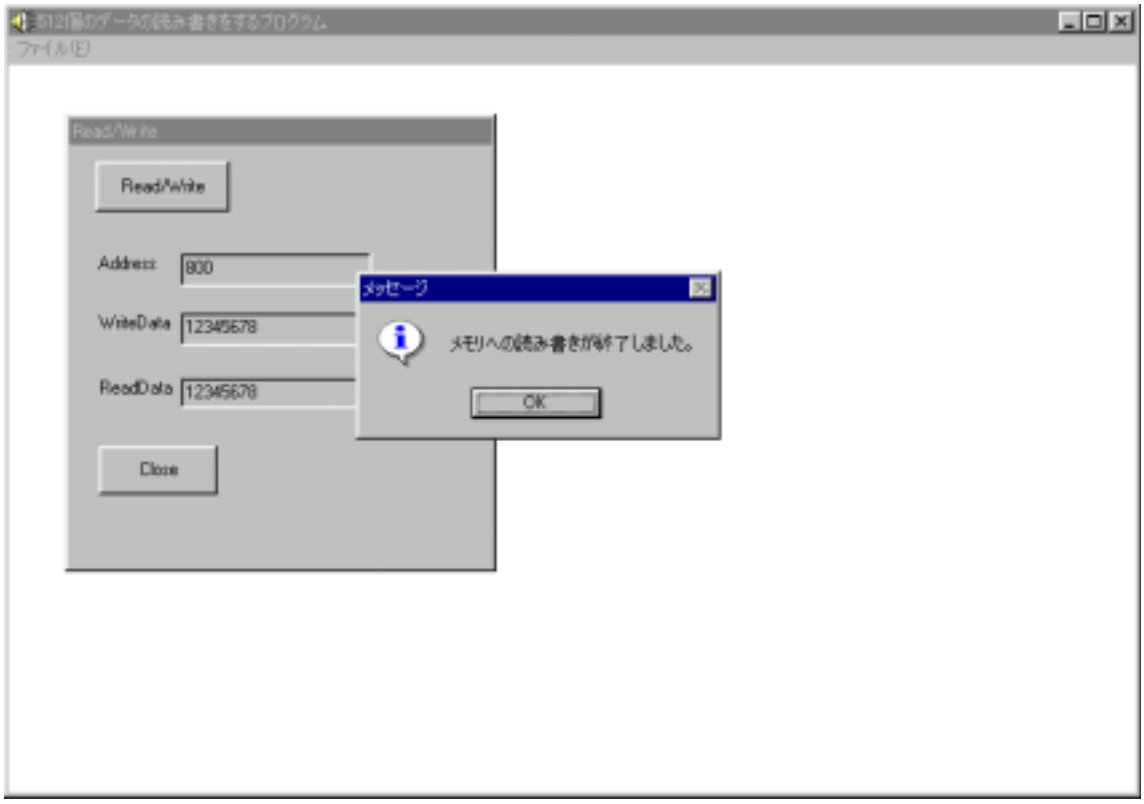


図 4.7: データを 512 個送ったプログラム [7]

このプログラムは EAB に対してシングルモードで 1 秒づつデータを読み書きしている。512 個のデータをコンフィグレーション空間の BAR2 に対して +4 づつアクセスしてデータを FPGA に書き込み読み取る。よって 16 進数表示でアドレスを 4 づつ加えたアドレスは 800 である。送ったデータは 16 進数表示で最初 12345678 を送り、1 秒ごとにその数字を反転したものを送って読み込んだ。今回の実験では、EAB を使用し、100 個、512 個の数のデータを送ることができた。この EAB は 1 つの容量が 4096bit で本研究で使用している FPGA には 24 個取り付けられているため、12Kbyte の容量を持っている。この容量全てをメモリとして使用した場合、32bit 幅のデータを 3072 個格納することができる。

[7] ファイル名:u01inar/ps/data512

第 5 章

考察と今後への提言

今年度から導入したPCIバスに対して、ハードウェアの進歩と同様にソフトウェアの進歩に伴い、大幅に設計ツールを変え、 PC から FPGA に対してのデータを書き込みのに対し、 PC からはまずPCIデバイスへPCIデバイスからFPGAへ、読み込みの時はその逆の動作を行うことのメカニズムは分かった。しかし、本研究の目的である行列演算器の設計とまでは行かなかった。また前年度まで使用していたISAバスとのデータ転送における高速性の評価も行うことができなかった。

昨今の情報技術の進歩によりPC開発環境、特にソフトウェアの進歩はめまぐるしく変わってきた。高速転送、 DMA(Direct Memory Access)、 FPGA の高集積化、性能は向上してもシステムは従来のものより小型化されていくといったハードウェアの進歩、それらのハードウェアを制御するのは今や、設計者自身が作りたいもの、制御したいものを作製していくのはソフトウェアの役目である。本研究では昨年度まで使用してきたPCとFPGAとのインターフェースがISAからPCIに変わったことで、新しいソフトウェアの導入に対しては一通り完成した。

さらに開発環境の向上性を求めるならば、行列演算器の開発設計において開発ソフトウェアの特徴として設計プロセスの短縮になるか、遅延、タイミングを考慮できるか、といったことを考慮する必要がある。例えば、設計の初期段階、本研究ではVHDLで記述し配置配線コンパイルの前にタイミングの解析を行うソフトを導入して試験するソフトを導入すれば、配置配線してタイミングシミュレーションを実行するよりは設計プロセス上時間短縮が行える。今後はソフトウェアの特徴、利点を把握することで新しいソフトウェアを導入していく設計者の眼力も今後は必要になってくる。

以上の事柄を念頭に置いた上でPCIバスをデータの通信に利用した行列演算器の設計に取り組むべきである。行列の内積や演算を行う上でのF PGA内のモジュールは山岡 [5] らによって既に完成されている。このモジュールを使って演算を行うことが可能だ。その際に必要となるメモリも

PC の CPU を介さずにメモリとのアクセスができる DMA 機能、または、直接 PCI 基板のサブボードとしてメモリを搭載することができる。

参考文献

[1] 中島瑞樹, “超高速行列演算チップの開発” , 1996 年度卒業論文

[2] 八木将志, “大行列の対角化プログラムの並列化” ,1996 年度卒業論文

[3] 松尾竜馬, “行列計算専用大規模集積回路の開発” ,1997 年度卒業論文

[4] グェン・ドゥック・ミン, “ハードウェア記述言語を用いた行列計算専用プロセッサの設計” ,1997 年度卒業論文

[5] 山岡寛明, “FPGA を用いた行列計算専用プロセッサの設計” ,1998 年度卒業論文

[6] 沼知典, “書き換え可能なゲート素子を持つデバイスを用いた行列計算専用集積回路の設計” ,1999 年度修士論文

[7] 清水信貴, “ハードウェア記述言語を用いた専用デバイスの設計” ,2000 年度卒業論文

[8] PCI9054 DATA BOOK ,PLX テクノロジー社

付録 A

VHDL プログラムソース

A.1 データの読み書き

```
--pci single readwrite

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

--Leonardo でピン定義をするときに必要なライブラリ
library exemplar ;
use exemplar.exemplar.all;


entity ReadWrite is

 port (
       LCLK   : in std_logic;
       LHOLD  : in std_logic;
       LHOLDA : out std_logic;
       ADS1   : in std_logic;
       BLAST1 : in std_logic;
       --LBE#   : in std_logic_vector(3 downto 0);
       LWDRD1 : in std_logic;
       LAD    : inout std_logic_vector(31 downto 0);
      -- LADIN  : in std_logic_vector(7 downto 0);
      -- LADOUT : out std_logic_vector(7 downto 0);
       DAT    : out std_logic_vector(31 downto 0);
       READY1 : out std_logic
      );

 attribute pin_number of LCLK   : signal is "91";
 attribute pin_number of LHOLD  : signal is "143";
 attribute pin_number of LHOLDA : signal is "144";
 attribute pin_number of ADS1   : signal is "56";
 attribute pin_number of BLAST1 : signal is "54";
 --attribute array_pin_number of LBE#   : signal is
   --                              ("113","114","115","116") ;
 attribute pin_number of LWDRD1 : signal is "117";
 attribute array_pin_number of LAD : signal is
        ("70","71","72","73","74","75","78","79","80","81","82","83","84","86","87",
"88","94","95","97","98","99","100","101","102","103","105","106","107","108","109","110","111");
 --attribute array_pin_number of LADIN  : signal is
   --                              ("70","71","72","73","74","75","78","79")  ;
 --attribute array_pin_number of LADOUT : signal is
   --                              ("80","81","82","83","84","86","87","88");
 attribute pin_number of READY1 : signal is "67";

end ReadWrite ;
```

```
architecture RTL of ReadWrite is
  signal ENABLE : std_logic;
  signal LWDRD  : std_logic;
  signal ADS    : std_logic;
  signal BLAST  : std_logic;
  signal READY  : std_logic;
  signal LADOUT : std_logic_vector(31 downto 0);
  signal DATIN  : std_logic_vector(31 downto 0);
  type STATE_TYPE is (IDLE, START, WAITSTATE1, WAITSTATE2, LAST);
  signal CURRENT_STATE, NEXT_STATE : STATE_TYPE ;

 begin

 LWDRD <= LWDRD1;
 ADS <= ADS1;
 BLAST <= BLAST1;
 READY1 <= READY;
 DAT <= DATIN;

-- 信号代入文により inout で信号を使う時必ず必要
 LAD <= DATIN when ENABLE = '0' and LWDRD = '0'  else "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";

-- ローカルバスを開く時 LHOLD に対して LHOLDA を返す
  process (LCLK) begin
    if (LCLK'event and LCLK= '1') then
       if LHOLD = '1' then
           LHOLDA <= LHOLD ;
       else
           LHOLDA <= '0';
       end if;
    end if;
  end process;

-- クロックによりステート変化
 process (LCLK) begin
 if (LCLK'event and LCLK='1') then
 CURRENT_STATE <= NEXT_STATE;
 end if;
 end process;

-- ステート定義
 process (ADS, LWDRD, BLAST) begin
   -- デフォルト出力
   READY <= '1';
   ENABLE <= '1';

   case CURRENT_STATE is

   --IDLE 初期状態 PCI バス入力待ち
   when IDLE =>
     READY <= '1';
     ENABLE <= '1';

     if ADS = '0'  then
        NEXT_STATE <= START;
     else
        NEXT_STATE <= IDLE;
     end if;

   --START BLAST 信号アサートされる
   when START =>
     READY <= '1';
     ENABLE <= '0';

     if BLAST = '0' then
        NEXT_STATE <= WAITSTATE1;
      else
        NEXT_STATE <= START;
     end if;

   --WAIT   データ保持
   when WAITSTATE1 =>
     READY <= '0' ;
     ENABLE <= '0';
```

```
     NEXT_STATE <= WAITSTATE2;

   when WAITSTATE2 =>
     READY <= '0';
     ENABLE <= '0';

     NEXT_STATE <= LAST;

   --LAST   データ転送終
   when LAST =>
     READY <= '1';
     ENABLE <= '1';

     if ADS = '1' and BLAST = '1' then
   NEXT_STATE <= IDLE ;
      elsif ADS = '0' then
   NEXT_STATE <= START;
     end if;
   end case;
  end process;

-- 書き込み、読み込みの指定 ENABLE、 LWDRD の値により ReadWrite
process(ENABLE, LWDRD) begin
if (LCLK'event and LCLK='1') then
 if ENABLE = '0' then
  if LWDRD = '1' then
     DATIN <= LAD;
  elsif LWDRD = '0' then
     LADOUT <= DATIN;
  end if;
 end if;
end if;
end process;

end RTL;
```

A.2 2 つのデータの足し算

```
--pci readwrite 2つのデータの足し算
--01/12/12

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

--Leonardo でピン定義をするときに必要なライブラリ
library exemplar ;
use exemplar.exemplar.all;


entity ReadWrite is

 port (
       LCLK   : in std_logic;
       LHOLD  : in std_logic;
       LHOLDA : out std_logic;
       ADS1   : in std_logic;
       BLAST1 : in std_logic;
       --LBE1   : in std_logic_vector(3 downto 0);
       LWDRD1 : in std_logic;
       LAD    : inout std_logic_vector(31 downto 0);
       LA     : out std_logic_vector(7 downto 0);
       --DAT1    : out std_logic_vector(31 downto 0);
       --DAT2    : out std_logic_vector(31 downto 0);
       READY1 : out std_logic
      );
```

```
 attribute pin_number of LCLK   : signal is "91";
 attribute pin_number of LHOLD  : signal is "143";
 attribute pin_number of LHOLDA : signal is "144";
 attribute pin_number of ADS1   : signal is "56";
 attribute pin_number of BLAST1 : signal is "54";
 --attribute array_pin_number of LBE#   : signal is
 --                              ("113","114","115","116") ;
 attribute pin_number of LWDRD1 : signal is "117";
attribute array_pin_number of LAD : signal is
       ("111","110","109","108","107","106","105","103","102","101","100","99","98","97",
"95","94","88","87","86","84","83","82","81","80","79","78",
"75","74","73","72","71","70");
 attribute array_pin_number of LA : signal is
       ("131","129","128","127","126","120","119","118");
 attribute pin_number of READY1 : signal is "67";

end ReadWrite ;

architecture RTL of ReadWrite is
  --PCI9054との通信に使う内部信号
  signal ENABLE : std_logic;
  signal LWDRD  : std_logic;
  signal ADS    : std_logic;
  signal BLAST  : std_logic;
  signal READY  : std_logic;
  --DATAを区別するのに使う内部信号
  signal COUNTW  : std_logic_vector(3 downto 0);
  -- 足し算の結果をこの信号に入れる
  signal LADOUT : std_logic_vector(31 downto 0);
  signal DATOUT : std_logic_vector(31 downto 0);
  --LADの値を入力する時にレジスタに振り分ける信号
  signal EN1 : std_logic;
  signal EN2 : std_logic;
  signal EN3 : std_logic;
  --2つのレジスタ、DATAをここに入れる
  signal DATIN1 : std_logic_vector(31 downto 0);
  signal DATIN2 : std_logic_vector(31 downto 0);
  -- ステートタイプ
  type STATE_TYPE is (IDLE, START, WAITSTATE, LAST);
  signal CURRENT_STATE, NEXT_STATE : STATE_TYPE ;


 begin


 LWDRD <= LWDRD1;
 ADS <= ADS1;
 BLAST <= BLAST1;
 READY1 <= READY;


-- 信号代入文によりinoutで信号を使う時必ず必要
 LAD <= LADOUT when ENABLE = '0' and LWDRD = '0'  else "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
 LA <= "00000111" when ENABLE ='0' and LWDRD = '0' else "ZZZZZZZZ"; -- アドレス7を指定して読み込
み
DATIN1 <= LAD when EN1 = '1' else "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
DATIN2 <= LAD when EN2 = '1' else "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";

-- ローカルバスを開く時LHOLDに対してLHOLDAを返す
  process (LCLK) begin
    if (LCLK'event and LCLK= '1') then
       if LHOLD = '1' then
           LHOLDA <= LHOLD ;
       else
           LHOLDA <= '0';
       end if;
    end if;
  end process;

-- クロックによりステート変化
 process (LCLK) begin
  if (LCLK'event and LCLK='1') then
     CURRENT_STATE <= NEXT_STATE;
```

```
  end if;
 end process;

-- ステート定義
 process (CURRENT_STATE, ADS, LWDRD, BLAST) begin
   -- デフォルト出力
   READY <= '1';
   ENABLE <= '1';

   case CURRENT_STATE is

   --IDLE 初期状態 PCI バス入力待ち
   when IDLE =>
     READY <= '1';
     ENABLE <= '1';

     if ADS = '0'  then
        NEXT_STATE <= START;
     else
        NEXT_STATE <= IDLE;
     end if;

   --START BLAST 信号アサートされる
   when START =>
     READY <= '1';
     ENABLE <= '0';

     if BLAST = '0' then
        NEXT_STATE <= WAITSTATE;
      else
        NEXT_STATE <= START;
     end if;

   --WAIT   データ保持
   when WAITSTATE =>
     READY <= '0' ;
     ENABLE <= '0';

     NEXT_STATE <= LAST;

   --LAST   データ転送終
   when LAST =>
     READY <= '1';
     ENABLE <= '1';

     if ADS = '1' and BLAST = '1' then
   NEXT_STATE <= IDLE ;
      elsif ADS = '0' then
   NEXT_STATE <= START;
     end if;
   end case;
  end process;

--ADS が 0 になれば COUNT に 1 づつ足す
process (ADS) begin
  if (falling_edge (ADS)) then
    COUNTW <= COUNTW + 1;
  end if;
 end process;

--READY が 1 になれば COUNTR に 1 づつ足す
--process (READY) begin
  --if (READY'event and READY='1') then
    --COUNTR <= COUNTR + 1;
  --end if;
--end process;

-- 書き込み読み込みの指定 ENABLE、 LWDRD の値により ReadWrite
--COUNT の値を利用してデータを LAD から書き込み、読み込む
process (LCLK, ENABLE, LWDRD, COUNTW) begin
if (LCLK'event and LCLK='1') then
 if ENABLE = '0' then
  if  LWDRD = '1' then
   if COUNTW = "0001" then
    EN1 <= '1';
```

```
    EN2 <= '0';
EN3 <= '0';
   elsif COUNTW = "0010"  then
    --DATIN2 <= LAD;
    EN1 <= '0';
    EN2 <= '1';
EN3 <= '1';
   else
    EN1 <= '0';
    EN2 <= '0';
EN3 <= '1';
   end if;
  end if;
 end if;
end if;
end process;

process(EN3) begin
 if EN3 ='1' then
   DATOUT <= DATIN1 + DATIN2;
 end if;
end process;

process(LCLK, ENABLE, LWDRD) begin
if (LCLK'event and LCLK='1') then
 if ENABLE = '0' then
  if LWDRD = '0' then
    LADOUT <= DATOUT;
   end if;
  end if;
 end if;
end process;


end RTL;
```

A.3 バーストサイクルモードによるデータの読み書き

```
--pci lpm_ram_dq を使用した burst 転送
--02/01/08

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

--Leonardo でピン定義をするときに必要なライブラリ
library exemplar ;
use exemplar.exemplar.all;


entity burstreadwrite is

 port (
       LCLK   : in std_logic;
       LHOLD  : in std_logic;
       LHOLDA : out std_logic;
       ADS1   : in std_logic;
       BLAST1 : in std_logic;
       LBE1   : in std_logic_vector(3 downto 0);
       LWDRD1 : in std_logic;
       LAD    : inout std_logic_vector(31 downto 0);
       LA     : in std_logic_vector(31 downto 2);
       READY1 : out std_logic;
       CONF_LED : out std_logic
       --PIC_RA : out std_logic_vector(5 downto 0);
       --PIC_RC0 : out std_logic;
       --USB_RESET : out std_logic;
       --BIGENDN : out std_logic;
```

```
       --U_D_Li : out std_logic;
       --U_D_Lo : out std_logic;
       --LINTN : out std_logic;
       --D_E  : out std_logic;
       --LRESETN : out std_logic;
       --WAITN : out std_logic;
       --BREQI : out std_logic;
       --BREQO : out std_logic;
       --LSERRN : out std_logic;
       --DP : out std_logic_vector(3 downto 0);
       --BTERMN : out std_logic
       );

 attribute pin_number of LCLK   : signal is "91";
 attribute pin_number of LHOLD  : signal is "143";
 attribute pin_number of LHOLDA : signal is "144";
 attribute pin_number of ADS1   : signal is "56";
 attribute pin_number of BLAST1 : signal is "54";
 attribute array_pin_number of LBE1   : signal is
                            ("113","114","115","116") ;
 attribute pin_number of LWDRD1 : signal is "117";
 attribute array_pin_number of LAD : signal is
        ("111","110","109","108","107","106","105","103","102","101","100","99","98",
"97","95","94","88","87","86","84","83","82","81","80","79",
"78","75","74","73","72","71","70");
 attribute array_pin_number of LA : signal is
        ("161","158","157","156","154","153","152","151","149","148","147","146","144","143",
"142","141","138","137","136","134","133","132","131","129","128","127","126","120","119","118");
 attribute pin_number of READY1 : signal is "67";
 attribute pin_number of CONF_LED : signal is "30";

end burstreadwrite ;

architecture RTL of burstreadwrite is

 --PCI9054 との通信に使う内部信号
  signal ENABLE : std_logic;
  signal LWDRD  : std_logic;
  signal ADS    : std_logic;
  signal BLAST  : std_logic;
  signal READY  : std_logic;
  signal LBE    : std_logic_vector(3 downto 0);


  --lpm_ram_dq の出力先
  signal LADOUT : std_logic_vector(31 downto 0);

  --lpm_ram_dq に入れるアドレス信号
  signal LPM_ADDR : std_logic_vector(7 downto 0);

  --lpm_ram_dq に入れるデータ
  --signal LAD_IN : std_logic_vector(31 downto 0);

  --signal LADc : std_logic_vector(31 downto 0);
  --lpm_ram_dq の Write Enable 信号
  signal BYTEN : std_logic_vector(3 downto 0);
  signal WRITE1 : std_logic;
  signal WRITE2 : std_logic;
  signal WRITE3 : std_logic;
  signal WRITE4 : std_logic;

  signal A31_28 : std_logic_vector(3 downto 0);
  signal EN_READ : std_logic;
  signal UPPER : std_logic_vector(3 downto 0);
  signal ADDRCNT : std_logic;
  signal CHIPSL : std_logic;

  -- 加算器の信号
  signal AA : std_logic_vector(31 downto 0);
  signal BB : std_logic_vector(31 downto 0);
  signal ANS : std_logic_vector(31 downto 0);
  signal ADDER : std_logic;
```

```
  -- 他と接続される未使用ピンは出力ピンに設定し、トライステート出力にする
  --signal GN : std_logic  ;
  -- ステートタイプローカル側にデータ入力
  type STATE_TYPE is (IDLE, START, BURST, WRITEREPEAT, READREPEAT, LAST);
  signal CURRENT_STATE, NEXT_STATE : STATE_TYPE ;


component lpmram
   port (
         inclock  : in std_logic;
         data : in std_logic_vector(7 downto 0);
         address : in std_logic_vector(7 downto 0);
         we   : in std_logic;
         q    : out std_logic_vector(7 downto 0)
        );
end component;


 begin

-- 他と接続される未使用ピンは出力ピンに設定し、トライステート出力にする
 --PIC_RA <= "000000" when GN ='1' else "ZZZZZZ";
 --PIC_RC0 <= '0' when GN ='1' else 'Z';
 --USB_RESET <='0' when GN='1' else 'Z';
 --BIGENDN <='0' when GN='1' else 'Z';
 --U_D_Li <='0' when GN='1' else 'Z';
 --U_D_Lo <='0' when GN='1' else 'Z';
 --LINTN <='0' when GN='1' else 'Z';
 --D_E  <='0' when GN='1' else 'Z';
 --LRESETN <='0' when GN='1' else 'Z';
 --WAITN <='0' when GN='1' else 'Z';
 --BREQI <='0' when GN='1' else 'Z';
 --BREQO <='0' when GN='1' else 'Z';
 --LSERRN <='0' when GN='1' else 'Z';
 --DP <="0000" when GN='1' else "ZZZZ";
 --BTERMN <='0' when GN='1' else 'Z';

--LPM_RAM_DQ コンポーネントと信号を接続する
MEM1: lpmram
  port map
(inclock => LCLK, data => LAD(7 downto 0), address =>
LPM_ADDR, we =>WRITE1,
q => AA(7 downto 0) );

MEM2: lpmram
  port map
(inclock => LCLK, data => LAD(15 downto 8), address =>
LPM_ADDR, we =>WRITE2,
q => AA(15 downto 8));

MEM3: lpmram
 port map
(inclock => LCLK, data => LAD(23 downto 16), address =>
LPM_ADDR, we =>WRITE3 ,
q => AA(23 downto 16));

MEM4: lpmram
 port map
(inclock => LCLK, data => LAD(31 downto 24), address =>
LPM_ADDR, we =>WRITE4,
q => AA(31 downto 24));


 LWDRD <= LWDRD1;
 ADS <= ADS1;
 BLAST <= BLAST1;
 READY1 <= READY;
 LBE <= LBE1;
 A31_28 <= LA(31 downto 28);
 --LAD_IN <= LAD;

-- 信号代入文により inout で信号を使う時必ず必要
 LAD <= LADOUT when EN_READ = '1'  else (others => 'Z');
```

```
 --LA <= (others => 'Z') when GN ='1' else (others => 'Z');


--LPM_RAM_DQ の Write ENABLE  制御信号
  BYTEN <= LBE(3 downto 0) when LWDRD ='1' and UPPER ="0010" else "1111";

--LA(31 downto 28) の値により UPPER の値を決める
  process (LCLK, LHOLD, ADS) begin
   if (LCLK'event and LCLK='1') then
    if LHOLD ='1' and ADS ='0' then
      if A31_28 ="0010" then
         UPPER <= A31_28;
      end if;
    else
       UPPER <= UPPER;
     end if;
    end if;
  end process;


-- ローカルバスを開く時 LHOLD に対して LHOLDA を返す
  process (LCLK) begin
    if (LCLK'event and LCLK= '1') then
       if LHOLD = '1' then
           LHOLDA <= LHOLD ;
       else
           LHOLDA <= '0';
       end if;
    end if;
  end process;

--LPM アドレスカウンター
 process (LCLK, ADS, BLAST, LWDRD, ADDRCNT)begin
  if (LCLK'event and LCLK ='1') then
    if ADS = '0' then
     LPM_ADDR <= LA(9 downto 2);
    elsif BLAST ='1' and LWDRD ='0' and ADDRCNT = '1' then
     LPM_ADDR <= LPM_ADDR + '1' ;
    else
     LPM_ADDR <= LPM_ADDR;
    end if;
  end if;
 end process;

-- クロックによりステート変化
 process (LCLK) begin
  if (LCLK'event and LCLK='1') then
     CURRENT_STATE <= NEXT_STATE;
   end if;
 end process;

-- ステート定義
 process (CURRENT_STATE, ADS, BLAST) begin
   -- デフォルト出力
   --READY <= '1';
   --ENABLE <= '1';
   --CHIPSL <= '1';
   --ADDRCNT <= '0'

   case CURRENT_STATE is
   --IDLE 初期状態 PCI バス入力待ち
   when IDLE =>
     READY <= '1';
     ENABLE <= '1';
     CHIPSL <= '1';
     ADDRCNT <= '0';

     if ADS = '0'  then
        NEXT_STATE <= START;
     else
        NEXT_STATE <= IDLE;
     end if;

   --START バーストサイクルモードスタート
   when START =>
```

```
      READY <= '1';
      ENABLE <= '1';
      CHIPSL <= '0';
      ADDRCNT <= '1';

      if BLAST = '1' then
         NEXT_STATE <= BURST;
       else
         NEXT_STATE <= START;
      end if;


   --BURST CYCLE START
   when BURST =>
      READY <= '0';
      ENABLE <= '0';
      CHIPSL <= '0';
      ADDRCNT <='0';

      if LWDRD = '1' then
         NEXT_STATE <= WRITEREPEAT;
      elsif LWDRD ='0' then
         NEXT_STATE <= READREPEAT;
      end if;

   --BURST WRITE REPEAT
   when WRITEREPEAT =>
      READY <= '1';
      ENABLE <='0';
      CHIPSL <= '0';
      ADDRCNT <= '0';

      if BLAST = '1' then
         NEXT_STATE <= BURST;
      elsif BLAST ='0' then
         NEXT_STATE <= LAST;
      end if;

   --BURST READ REPEAT
   when READREPEAT =>
      READY <= '0';
      ENABLE <= '0';
      CHIPSL <= '0';
      ADDCNT <= '0';

      if BLAST = '1' then
         NEXT_STATE <= READREPEAT;
      elsif BLAST = '0' then
         NEXT_STATE <= LAST;
      end if;

   --BURST CYCLE LAST
   when LAST =>
      READY <= '1';
      ENABLE <= '1';
      CHIPSL <= '1';
      ADDRCNT <='0';

      if ADS ='0' then
        NEXT_STATE <= START;
      else
        NEXT_STATE <= IDLE;
      end if;

   end case;
  end process;


--LPM_RAM  からの書き込み、読込みの指定
process(LCLK, LWDRD, ENABLE) begin
 if (LCLK'event and LCLK='1') then
  if LWDRD='0' and ENABLE ='0' then
    EN_READ <= '1';
  else
    EN_READ <= '0';
  end if;
```

```
 end if;
end process;

-- コンフィグレーション LED を光らせる
--LPM の Write Enable 信号を有効にする
process(LCLK, CHIPSL, BYTEN) begin
  if (LCLK'event and LCLK='1') then
   if CHIPSL ='0' and  BYTEN(0)='0' then
     CONF_LED <= LAD(0);
    else
     CONF_LED <= '0';
   end if;
  end if;
end process;

--lpm_ram_dq の Write Enale を有効にするプロセス
process(LCLK, CHIPSL) begin
 if (LCLK'event and LCLK='1') then
   if CHIPSL = '0' and BYTEN(0) ='0' then
     WRITE1 <= '1';
   elsif CHIPSL ='0' and BYTEN(1) ='0' then
     WRITE2 <= '1';
   elsif CHIPSL ='0' and BYTEN(2) ='0' then
     WRITE3 <= '1';
   elsif CHIPSL ='0' and BYTEN(3) ='0' then
     WRITE4 <='1';
   else
     WRITE1 <='0';
     WRITE2 <='0';
     WRITE3 <='0';
     WRITE4 <='0';
   end if;
 end if;
end process;

-- 加算を行うプロセス
process(LCLK, ADDER, AA) begin
 if (LCLK'event and LCLK='1') then
   if ADDER = '1' then
    ANS <= AA + BB ;
   end if;
   if AA = "00000000000000000000000000000000" and ADDER <= '1' then
    LADOUT <= ANS;
   else
    BB <= ANS;
   end if;
 end if;
end process;



end RTL;
```

付録 B

Visual C++ プログラムソース

B.1 データの読み書き

```
#include <windows.h>
#include "resource.h"
#include <stdio.h>
#include <time.h>
#include "testp5_lib.h"

BOOL InitApplication( HANDLE hInstance );
HWND InitInstance( HANDLE hInstance, int nCmdShow );
LRESULT PASCAL Testp5WndProc( HWND hWnd, UINT uMsg,
WPARAM wParam, LPARAM lParam );
VOID GoModalDialogBoxParam( HINSTANCE hInstance, LPCSTR
lpszTemplate, HWND hWnd,
DLGPROC lpDlgProc, LPARAM lParam );
BOOL PASCAL ReadWriteDlgProc( HWND hDlg, UINT uMsg, WPARAM wParam, LPARAM lParam );

char gszAppName[] = "テストプログラム ";
char gszTestp5Class[] = "Testp5WndClass";

// The main window loop.
// WinMain() opens a handle for TestP5, and then creates the main menu window.
int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE
hPrevInstance,
LPSTR lpszCmdLine, int nCmdShow )
{
HWND hTestP5Wnd;
    MSG msg;
TESTP5_HANDLE hTESTP5 = NULL;
    BOOL fUseInt = FALSE; // by default - do not install interrupts

    if (!hPrevInstance)
        if (!InitApplication( hInstance ))
            return FALSE;

if (TESTP5_Open (&hTESTP5, TESTP5_DEFAULT_VENDOR_ID,
TESTP5_DEFAULT_DEVICE_ID, 1, fUseInt ?
TESTP5_OPEN_USE_INT : 0 ))
    {
char msg[256];
sprintf (msg, "Error while opening TestP5 hardware:\n%s", TESTP5_ErrorString);
        MessageBox( NULL, msg, "エラー ", MB_OK | MB_ICONERROR );
        return FALSE;
    }

if (NULL == (hTestP5Wnd = InitInstance( hInstance, nCmdShow )))
return FALSE;

    while (GetMessage( &msg, NULL, 0, 0 ))
```

```
    {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }

if (hTESTP5) TESTP5_Close(hTESTP5);

    return (int) msg.wParam;
}

// Initilization. This registers information such as window classes.
BOOL InitApplication( HANDLE hInstance )
{
    WNDCLASS  wndclass;

    // register Testp5 window class
    wndclass.style =         0;
  wndclass.lpfnWndProc =   Testp5WndProc;
    wndclass.cbClsExtra =    0;
    wndclass.cbWndExtra =    0;
    wndclass.hInstance =     hInstance;
  wndclass.hIcon =         LoadIcon( hInstance, MAKEINTRESOURCE( SPEAKERICON ) );
    wndclass.hCursor =       LoadCursor( NULL, IDC_ARROW );
    wndclass.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
wndclass.lpszMenuName =  MAKEINTRESOURCE( TESTP5MENU );
  wndclass.lpszClassName = gszTestp5Class;

    return RegisterClass( &wndclass );
}

// Initializes instance specific information.
HWND InitInstance( HANDLE hInstance, int nCmdShow )
{
    HWND hTestp5Wnd;

    // create the Testp5 window
hTestp5Wnd = CreateWindow( gszTestp5Class, gszAppName,
                           WS_OVERLAPPEDWINDOW,
                           CW_USEDEFAULT, CW_USEDEFAULT,
                           CW_USEDEFAULT, CW_USEDEFAULT,
                           NULL, NULL, hInstance, NULL );

    if (NULL == hTestp5Wnd)
        return NULL;

    ShowWindow( hTestp5Wnd, nCmdShow );
    UpdateWindow( hTestp5Wnd );

    return hTestp5Wnd;
}

// WinDriver のハンドル
// インストールされている WinDriver が最新版であることを確認する
BOOL PCI_Get_WD_handle(HANDLE *phWD)
{
    WD_VERSION ver;

    *phWD = WD_Open();

    // Check whether handle is valid and version OK
    if (*phWD==INVALID_HANDLE_VALUE)
    {
MessageBox( NULL,
"Cannot open WinDriver device", "エラー ", MB_OK | MB_ICONERROR );
        return FALSE;
    }

    BZERO(ver);
    WD_Version(*phWD,&ver);
    if (ver.dwVer<WD_VER)
    {
MessageBox( NULL,
```

```
"error - incorrect WinDriver version", "エラー", MB_OK | MB_ICONERROR );
WD_Close (*phWD);
        *phWD = INVALID_HANDLE_VALUE;
        return FALSE;
    }
}

TESTP5_HANDLE TESTP5_LocateAndOpenBoard(DWORD dwVendorID, DWORD dwDeviceID, BOOL fUseInt)
{
    DWORD cards, my_card;
    TESTP5_HANDLE hTESTP5 = NULL;
char msg[256];

if (dwVendorID==0) return NULL;

    cards = TESTP5_CountCards (dwVendorID, dwDeviceID);

    if (cards==0)
    {
  sprintf (msg, "%s", TESTP5_ErrorString);
        MessageBox( NULL, msg, "エラー", MB_OK | MB_ICONERROR );
        return NULL;
    }
    else if (cards==1) my_card = 1;
    else
    {
        DWORD i;

        i = 0;
        if (i>=1 && i <=cards) my_card = i;
        else
        {
MessageBox( NULL, "Choice out of range", "エラー", MB_OK | MB_ICONERROR );
            return NULL;
        }
    }
    if (TESTP5_Open (&hTESTP5, dwVendorID, dwDeviceID,
my_card - 1, fUseInt ?
TESTP5_OPEN_USE_INT : 0 ))
{
}
    else
{
sprintf (msg, "%s", TESTP5_ErrorString);
MessageBox( NULL, msg, "エラー", MB_OK | MB_ICONERROR );
}
    return hTESTP5;
}

// データを書き込む
void TESTP5_AccessRangesWrite(TESTP5_HANDLE hTESTP5, HWND hDlg)
{
DWORD addr, write_data; // アドレス、書き込むデータ
    TESTP5_ADDR ad_sp = 0; // アドレス領域用変数を定義し、BAR0 を設定
time_t rtimer, nowtime, oldtime;
time_t *timer = &rtimer;
char msg[256]; // メッセージ用

    if(!TESTP5_IsAddrSpaceActive(hTESTP5, ad_sp)) //BAR0 が使用可能かを調査する
{
MessageBox( NULL,
"No active memory or IO ranges on board!", "エラー", MB_OK | MB_ICONERROR );
        return;
    }

write_data = GetDlgItemInt(hDlg, IDC_WRITE, NULL, FALSE); // 書込みデータ

for(addr =0; addr < 0x8 ;addr +=4) //0x00 ～ 0x8 に 32bit アクセス
{

//TESTP5_WriteDword(hTESTP5, ad_sp, addr, write_data); // データを書き込む

//read_data = TESTP5_ReadDword(hTESTP5, ad_sp, addr); // データを読み込む
```

```
UpdateWindow(hDlg); // クライアント領域を更新する

sprintf (msg, "%8d", write_data);
SetDlgItemText(hDlg, IDC_WRITEDATA, msg);

sprintf (msg, "%2X", addr);
SetDlgItemText(hDlg, IDC_WRITEADD, msg);

oldtime = time(timer);
do
 nowtime = time(timer);
 while(nowtime == oldtime);
}
}


// データを読み込む
void TESTP5_AccessRangesRead(TESTP5_HANDLE hTESTP5, HWND hDlg)
{
DWORD addr, write_data, read_data; // アドレス、読み込んだデータ, 書き込むデータ
    TESTP5_ADDR ad_sp = 0; // アドレス領域変数を定義し、 BAR0 を設定
time_t rtimer, nowtime, oldtime;
time_t *timer = &rtimer;
boolean readerror; // 読み込みエラーがある or ない
char msg[256]; // メッセージ用

    if(!TESTP5_IsAddrSpaceActive(hTESTP5, ad_sp)) //BAR0 が使用可能かを調査する
{
MessageBox( NULL,
"No active memory or IO ranges on board!", "エラー ", MB_OK | MB_ICONERROR );
        return;
    }

write_data = GetDlgItemInt(hDlg, IDC_WRITE, NULL, TRUE); // 書込みデータ

for(addr =0; addr < 0x8 ;addr +=4) //0x00 ～ 0x08 に 32bit アクセス
{

TESTP5_WriteDword(hTESTP5, ad_sp, addr, write_data); // データを書き込む

read_data = TESTP5_ReadDword(hTESTP5, ad_sp, addr); // データを読み込む


UpdateWindow(hDlg); // クライアント領域を更新する

sprintf (msg, "%8d", read_data);
SetDlgItemText(hDlg, IDC_READ, msg);

sprintf (msg, "%2X", addr);
SetDlgItemText(hDlg, IDC_READADD, msg);

if (read_data != write_data)

{
readerror = 1;
MessageBox( NULL, "読み込みエラーです。\n 読み込みは正しく行われません。 ","エラー ",MB_OK | MB_ICONERROR);

break;
}
else
readerror =0;

oldtime = time(timer);
do
 nowtime = time(timer);
 while(nowtime == oldtime);
}
    if (readerror == 0)
MessageBox( NULL, "読み込みが終了しました。 ","メッセージ ", MB_OK | MB_ICONASTERISK);
}
```

```
// ファイルメニュー
LRESULT PASCAL Testp5WndProc( HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam )
{
    switch (uMsg)
    {
    case WM_COMMAND:
        switch ( LOWORD( wParam ) )
        {
case IDM_TESTPROGRAM: //「テストプログラム」を選択した場合
            GoModalDialogBoxParam( (HINSTANCE) GetWindowLong( hWnd, GWL_HINSTANCE ),
                                   MAKEINTRESOURCE( WRITEREADBOX ), hWnd,
    ReadWriteDlgProc,
                                   (LPARAM) NULL);
            break;

        case IDM_EXIT: //[終了] を選択した場合
            PostMessage( hWnd, WM_CLOSE, 0, 0L );
            break;
       }
        break;

    case WM_DESTROY:
        PostQuitMessage( 0 );
        break;

    default:
        return DefWindowProc( hWnd, uMsg, wParam, lParam );
    }

    return FALSE;
}

//  【Read/Write】ダイアログボックス
BOOL PASCAL ReadWriteDlgProc( HWND hDlg, UINT uMsg, WPARAM wParam, LPARAM lParam )
{
    TESTP5_HANDLE hTESTP5 = NULL;
    HANDLE hWD;
BOOL fUseInt = FALSE; // by default - do not install interrupts

    switch (uMsg)
    {
    case WM_COMMAND:
        switch ( LOWORD( wParam ))
        {
        case IDD_WRITE: //WriteData を押した場合
            {
if (!PCI_Get_WD_handle(&hWD)) return 0;

if (TESTP5_DEFAULT_VENDOR_ID)
hTESTP5 = TESTP5_LocateAndOpenBoard(TESTP5_DEFAULT_VENDOR_ID, TESTP5_DEFAULT_DEVICE_ID, fUseInt);

if (hTESTP5) TESTP5_AccessRangesWrite(hTESTP5, hDlg);

if (hTESTP5) TESTP5_Close(hTESTP5);

WD_Close (hWD);

return 0;
break;
          }

case IDD_READ: //ReadData を押した場合
            {
if (!PCI_Get_WD_handle(&hWD)) return 0;

if (TESTP5_DEFAULT_VENDOR_ID)
hTESTP5 = TESTP5_LocateAndOpenBoard(TESTP5_DEFAULT_VENDOR_ID, TESTP5_DEFAULT_DEVICE_ID, fUseInt);

if (hTESTP5) TESTP5_AccessRangesRead(hTESTP5, hDlg);

if (hTESTP5) TESTP5_Close(hTESTP5);

WD_Close (hWD);
```

```
return 0;
break;
            }

        case IDD_CLOSE: // 【CLOSE】ボタンを押した場合
            EndDialog( hDlg, TRUE );
            return TRUE;
        }
        break;
    }

    return FALSE;
}

VOID GoModalDialogBoxParam( HINSTANCE hInstance, LPCSTR
lpszTemplate, HWND hWnd,
DLGPROC lpDlgProc, LPARAM lParam )
{
    DLGPROC  lpProcInstance;

    lpProcInstance = (DLGPROC) MakeProcInstance( (FARPROC) lpDlgProc, hInstance );
    DialogBoxParam( hInstance, lpszTemplate, hWnd, lpProcInstance, lParam );
    FreeProcInstance( (FARPROC) lpProcInstance );
}
```

付録 C

回路設計する上でのソフトウェアの使用方法

ここでは、回路設計を行う上で、必要となるソフトウェアの使用方法について説明する。説明するソフトウェアは Leonardo Spectrum、 QuartusII、 WinDriver、 Visual C++、である。

C.1 Leonardo Spectrum

Leonardo Spectrum は論理合成ツールである。 VHDL 記述を Leonardo Spectrum で行うことはできるが、本研究では VHDL の記述にはテキストエディタを使用した。 VHDL 記述については清水 [7] の卒業論文に述べられている。ここでは Leonardo Spectrum での論理合成の方法について述べる。

C.1.1 VHDL 記述において PeakVHDL と異なる点

前年度まで使用していた VHDL 記述エディタ PeakVHDL で論理合成を行うときと、 Leonardo Spectrum で論理合成を行うときで異なる VHDL 記述をしなければならない事が一つだけある。それは FPGA にあらかじめ決められている pin 番号と VHDL で定義した信号を結ぶときである。

PeakVHDL を使用して論理合成を行うときのピン番号の定義はライブラリを呼び出すところで

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

library metamor;
use metamor.attributes.all;
```

と記述し、 metamor ライブラリを呼び出す。これは論理合成ツールのベンダーである Synopsys library を利用している。

```
entity count is
    port ( CLK : in std_logic;
           BL  : in std_logic_vector(7 downto 0);
           BH  : out std_logic_vector(7 downto 0);
           CL  : in std_logic   );

attribute pinnum of CLK : signal is ‘‘D22’’;
attribute pinnum of BL  : signal is ‘‘BC13,BB14,BC15,BB16,BC17,BB18,BC19,BB20’’;
attribute pinnum of BH  : signal is ‘‘BC5,BB6,BC7,BB8,BC9,BB10,BC11,BB12’’;
attribute pinnum of CL  : signal is ‘‘AU23’’;

end count;
```

その後、 FPGA の外から来る信号、外に出す信号の定義を行い、 pin 配置を行う。であるのに対して、 Leonardo Spectrum を使用して論理合成を行うときのピン番号の定義を行う上での使用するライブラリは

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

library exemplar;
use exemplar.exemplar.all;
```

と記述する。これは exemplar 社のライブラリを利用すること示している。次にピン番号の指定する記述は

```
entity count is
    port ( CLK : in std_logic;
           A   : inout std_logic_vector(15 downto 0);
           BH  : out std_logic_vector(7 downto 0);
           CL  : in std_logic   );

attribute pin_number of CLK : signal is ‘‘D22’’;
attribute array_pin_number of BL  : signal is
                        (‘‘BC13’’,‘‘BB14’’,‘‘BC15’’,‘‘BB16’’,‘‘BC17’’,‘‘BB18’’,‘‘BC19’’,‘‘BB20’’);
attribute array_pin_number of BH  : signal is
                        (‘‘BC5’’,‘‘BB6’’,‘‘BC7’’,‘‘BB8’’,‘‘BC9’’,‘‘BB10’’,‘‘BC11’’,‘‘BB12’’;
attribute pin_number of CL  : signal is ‘‘AU23’’;

end count;
```

と記述する。 port で定義した信号が std_logic_vector ならピン番号は array_pin_number と記述し、番号一つ一つダブルクォーテーションで囲み、全体を括弧で囲む。 std_logic なら pin_number と記述する。

C.1.2 Leonardo Spectrum を使用した論理合成

Leonardo Spectrum を使用した論理合成を説明する。

- leonardo Spectrum を起動する。

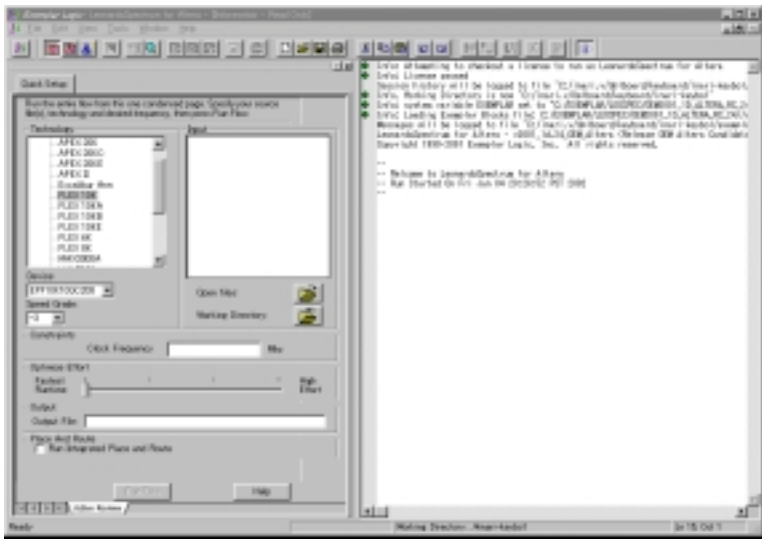


図 3.1: Leonardo Spectrum [1]

- VHDL ソースの選択

図 3.2: open file [2]

  - Open Files をクリックする。
  - 論理合成するソースファイルの選択

[1] ファイル名:u01inar/ps/kidou.ps

[2] ファイル名:u01inar/ps/input.ps

- 階層設計により複数のソースを選択する場合は最上位のソースファイルをいちばん最後に読み出す。
- 階層段階を間違えた場合は Input 画面に表示されているソースファイルをドラッグ & ドロップで移動することができる。

- ワークディレクトリの設定

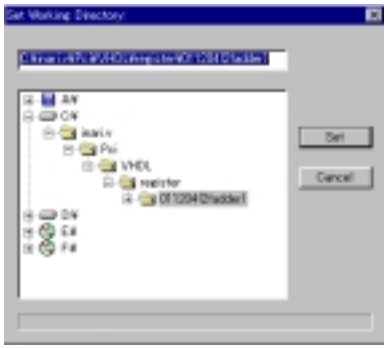


図 3.3: ワークディレクトリ [3]

- Working Directory をクリックする。
- 論理合成後の edif ファイルの出力先を指定する。
- 指定したら set をクリックする。

- デバイスの選択

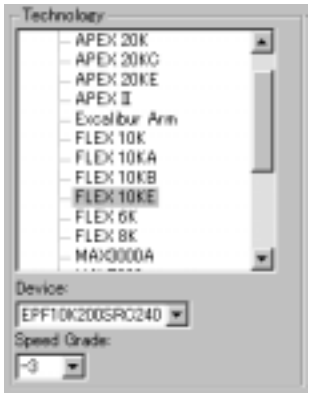


図 3.4: デバイスの選択 [4]

- Technology からデバイスファミリを選ぶ

[3]ファイル名:u01inar/ps/dir.ps

[4]ファイル名:u01inar/ps/device.ps

- Device からデバイスを指定する
- Speed Grade はデバイス名の最後のハイフン以下を指定
  - (例) EPF10K200SRC240-3 なら -3 を選ぶ
- ここでで指定したデバイス情報は配置配線ツールで使用する acf ファイルに書き込まれる。

- 最適化方法の指定

図 3.5: 最適化の設定 [5]

- Constraints の Clock Frequency は動作希望周波数を設定する項目。
- Optimaize Effort は動作希望周波数に近づける努力を設定する項目。右側に行くほど良い結果を導き出すが時間がかかる。

- 論理合成
  - 以上の設定が終わったら Run Flow ボタンをクリックする。
  - エラーがなく合成が終了すると Finished Synthesis run と表示される。
  - edif 、 acf ファイルが Working Directory にあることを確認する。

- エラーの表示
  - エラーが表示されると、エラー内容と行番号が表示されたら、その表示されたところでダブルクリックを行うと、ソースファイルのエラーにカーソルが移動する。
  - エラーメッセージに従いエラーを修正する。
  - 再度合成するときは、修正後 Run Flow をクリックする。

[5] ファイル名:u01inar/ps/saiteki.ps

C.2 QuartusII の使い方

C.2.1 QuartusII を使用した配置配線

QuartusII を使用した配置配線の方法を説明する。 QuartusII は前年度まで使用していた Max+Plus2 の後継ソフトであり、 EDIF ファイルのライブラリの多くに対応している。実際、 Leonardo Spectrum で論理合成した EDIF ファイルで Max+Plus2 で配置配線することができないものがあるが、 QuartusII では配置配線することができる。

- プロジェクトの作製
  - QuartusII を起動する

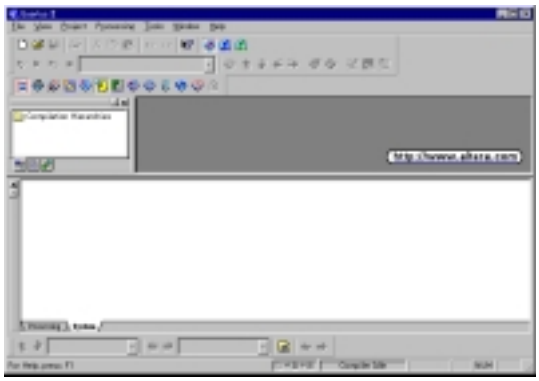


図 3.6: QuartusII の起動 [6]

  - File → New Project Wizard... を選択
  - Introduction のページが開き Next をクリックすると、最初のページが表示される。

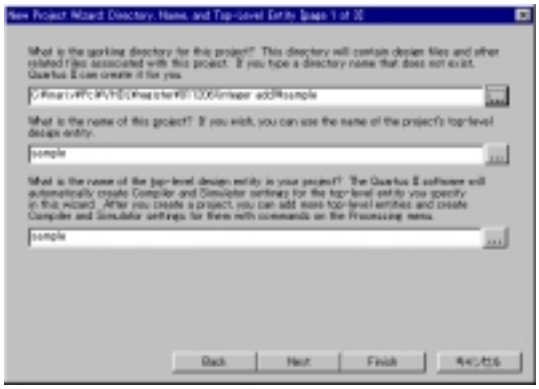


図 3.7: New Project Wizard 最初のページ [7]

[6] ファイル名:u01inar/ps/q0.ps

[7] ファイル名:u01inar/ps/q1.ps

- 作業ディレクトリ、プロジェクト名、トップの回路のエンティティ名を入力して Next を選択
- New Project Wizard の 2 ページ目が表示される

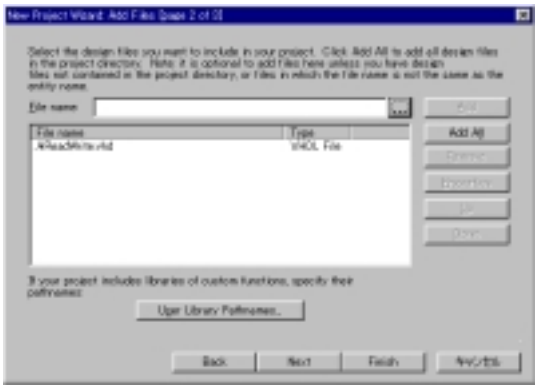


図 3.8: New Project Wizard 2 ページ目 [8]

- ここではプロジェクトに追加するファイル (edif ファイル) を選択する
- プロジェクトに入れるファイルを選択したら Next を選択
- New Project Wizard の 最後のページが表示される

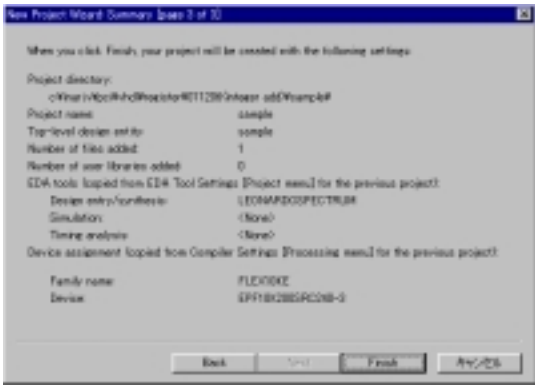


図 3.9: New Project Wizard 最後のページ [9]

- ここには作業ディレクトリ、プロジェクト名、トップの回路のエンティティ名、プロジェクトの中のファイルの数、論理合成ツール、 Leonardo Spectrum で選択したデバイスなど、プロジェクトととして定義した情報が表示される。

- MegaWizard Plug-In Manager の使い方
  EAB を使った回路設計を行うとき、その EAB を使った ALTERA 独自のマクロが定義さ

[8] ファイル名:u01inar/ps/q2.ps

[9] ファイル名:u01inar/ps/q3.ps

れている。そのマクロは EAB を RAM や ROM として使用することができ、 MegaWizard Plug-In Manager を使用して使用するマクロ、データ幅、アドレス幅の設定を行う。ここではその MegaWizard Plug-In Manager を使用しての ALTERA マクロの使用方法にていて説明する。

- ツールバー Tools → MegaWizard Plug-In Manager を選択

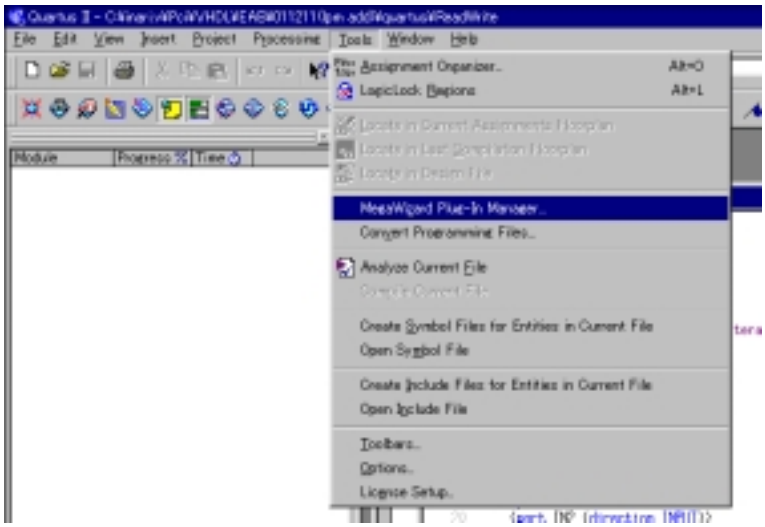


図 3.10: MegaWizard Plug-In Manager を選択 [10]

- MegaWizard Plug-In Manager ダイアログボックスが表示され、 Next を選択

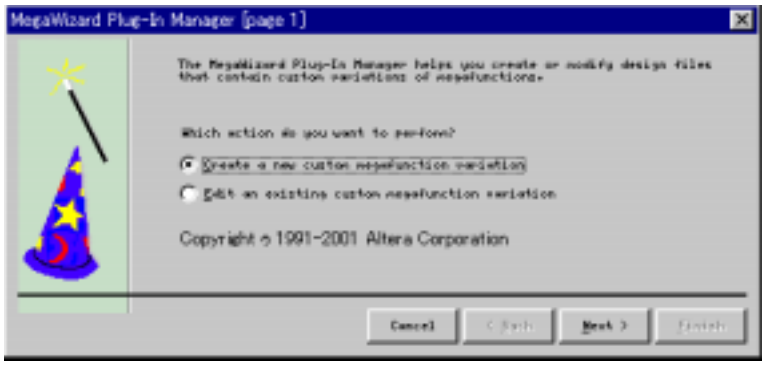


図 3.11: MegaWizard Plug-In Manager ダイアログボックス [11]

- ここでは EAB を RAM としてデータを単方向で指定できる LPM_RAM_DQ を 選択する。次に Which Type では AHDL を指定する。これは AHDL で指定すれば Quartus でコンパイルできるため VHDL よりも作業の手間が省ける。最後にファイル名を指定して Next を選択

[10] ファイル名:u01inar/ps/meg0.ps

[11] ファイル名:u01inar/ps/meg1.ps

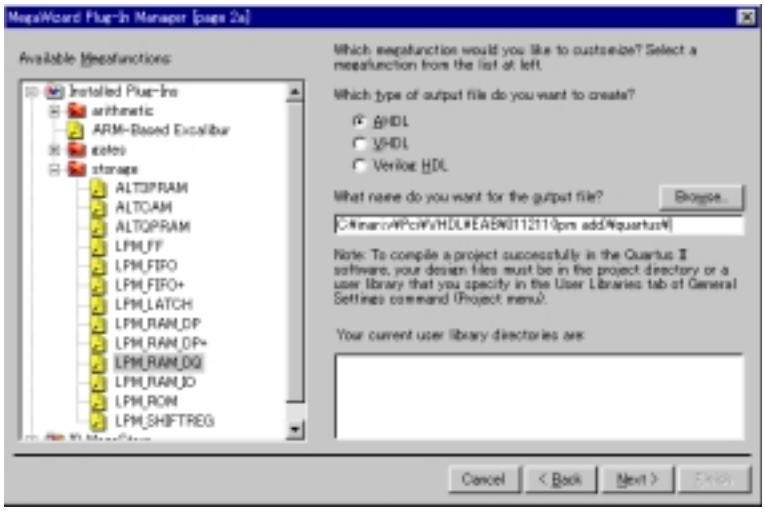


図 3.12: ALTERA マクロの選択 [12]

– アドレス幅、データ幅を指定する

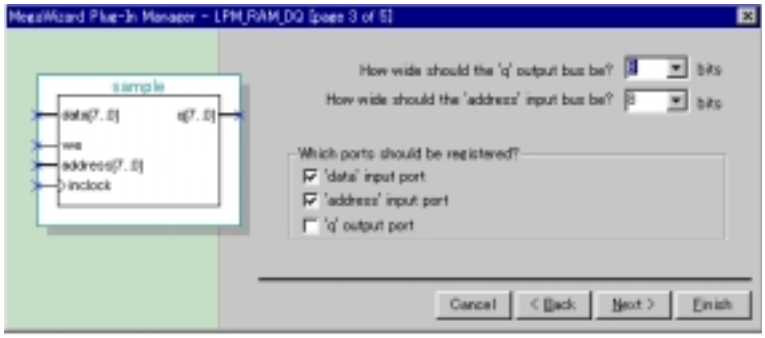


図 3.13: アドレス幅、データ幅の選択 [13]

– 最後に MegaWizard Plug-In Manager で出力するファイルが表示される

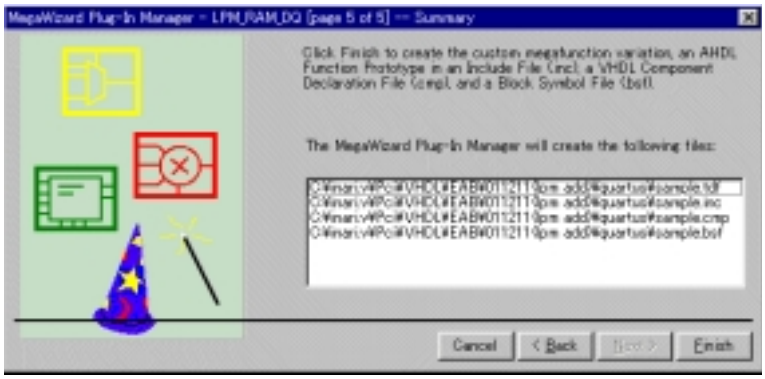


図 3.14: MegaWizard Plug-In Manager で出力するファイル [14]

[12] ファイル名:u01inar/ps/meg2.ps

[13] ファイル名:u01inar/ps/meg3.ps

[14] ファイル名:u01inar/ps/meg4.ps

– 次に、ツールバー Project → Add Files to Project を選択する。

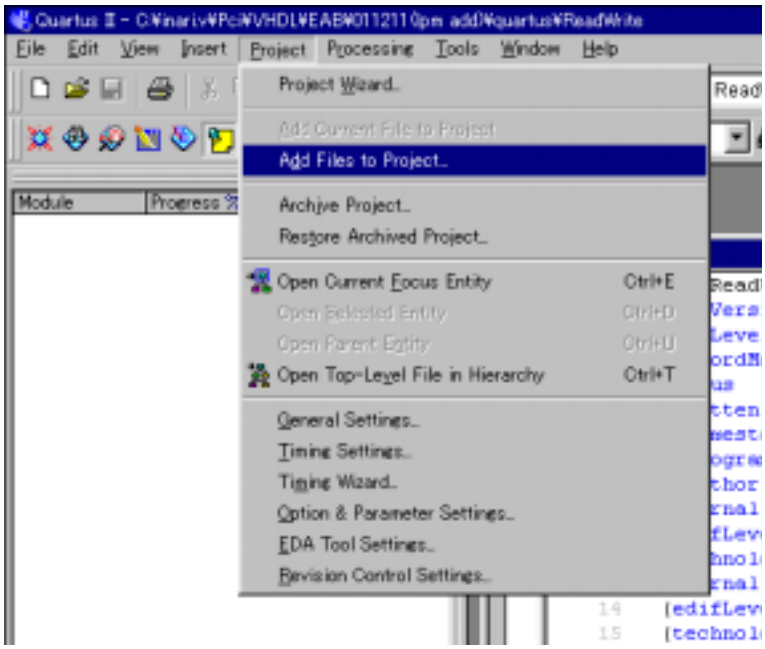


図 3.15: Add Files to Project の選択 [15]

– Add File ダイアログボックスが表示され、図 3.16に示した ◯ をクリックする

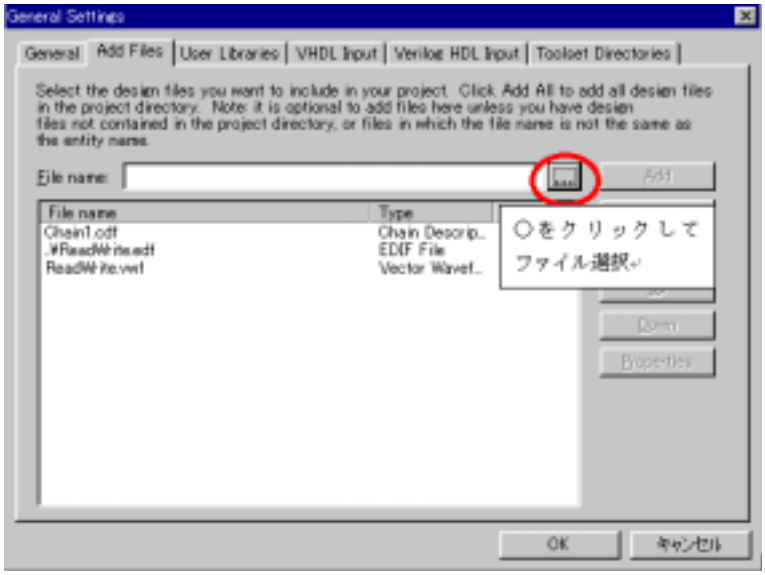


図 3.16: Add File ダイアログボックス [16]

– プロジェクトに MegaWizard Plug-In Manager で作製した AHDL ファイルを取り込む。 AHDL ファイルの拡張子は tdf である。

[15] ファイル名:u01inar/ps/meg5.ps

[16] ファイル名:u01inar/ps/meg6.ps

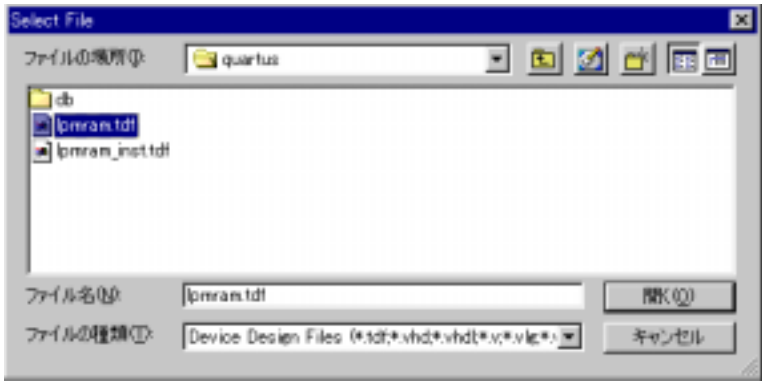


図 3.17: tdf ファイルの選択 [17]

- コンパイル
  コンパイルを実行することで、設計回路のエラーを調べ、回路と選択した FPGA との配置配線を行い、タイミングシミュレーション、デバイスプログラミング用の出力ファイルを作製する。

  – コンパイラセッティングを確認

    * Compile Mode を選択して、 Compile モードであることを確認

図 3.18: ツールバー Compile Mode [18]

      · ツールバーの Compile Mode を選択

    * ツールバー Processing より Compiler Setting を選択する

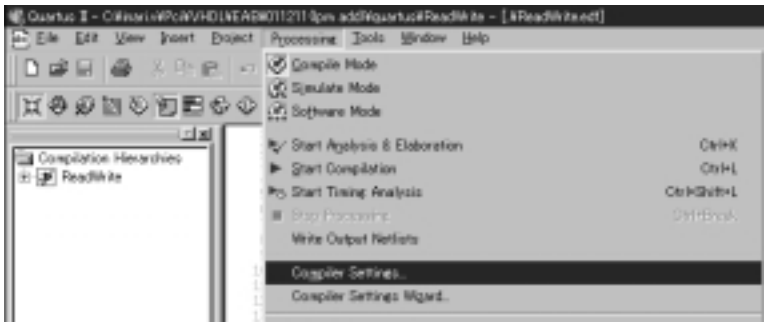


図 3.19: Compiler Setting [19]

[17] ファイル名:u01inar/ps/meg7.ps

[18] ファイル名:u01inar/ps/com0.ps

[19] ファイル名:u01inar/ps/co0.eps

* Compiler Setting を選択すると General タブが開く

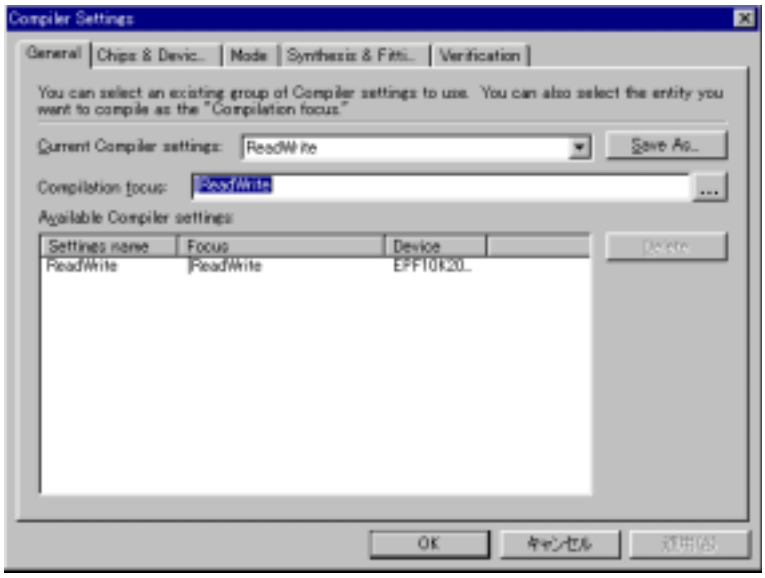


図 3.20: General タブ [20]

- General タブでコンパイラセッティング名、エンティティ名、既存のコンパイラセッティングを確認する

* Chip&Devices タブで選択されている FPGA を確認する

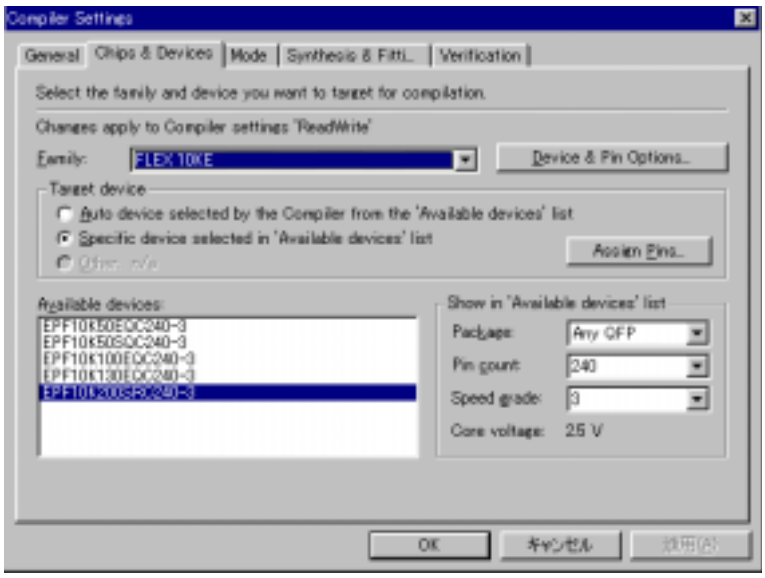


図 3.21: Chip&Devices タブ [21]

* Mode タブで Full compilation を選択する。

[20] ファイル名:u01inar/ps/co1.ps

[21] ファイル名:u01inar/ps/co2.ps

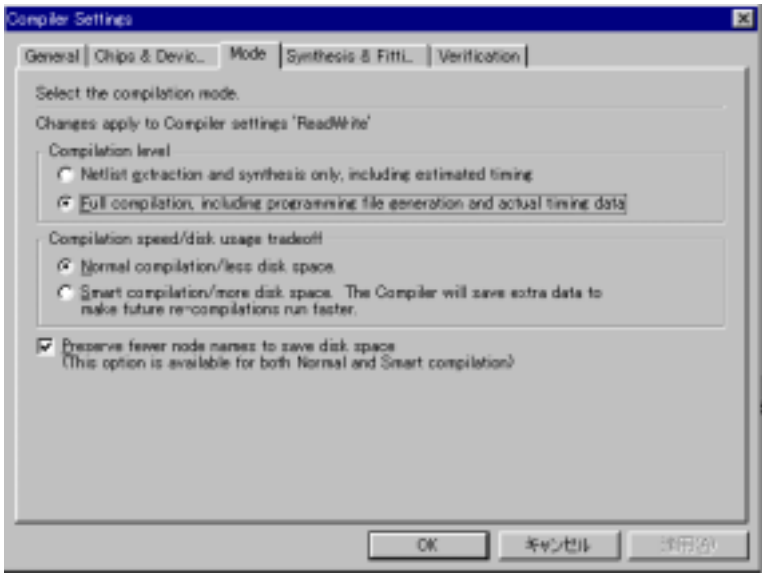


図 3.22: Mode タブ [22]

- 将来、再コンパイルするときは時間短縮するため、[Compilation speed/disk usage tradeoff] で [Smart compilation/more disk space] を選択する
- Mode タブの [Preserve fewer node names to save disk space] のチェックボックスがオンになっていることを確認する

＊ Synthesis &Fitting タブでロジック配置の最適化を Fitter に指示する

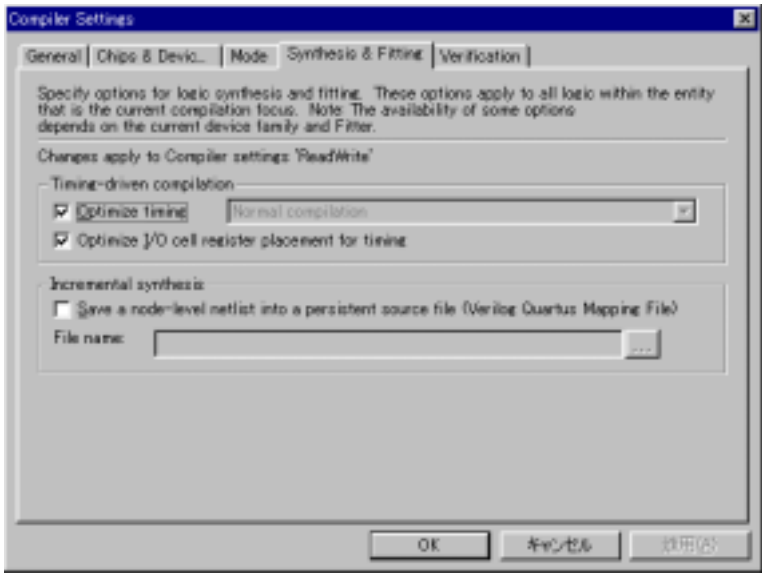


図 3.23: Synthesis &Fitting [23]

[22] ファイル名:u01inar/ps/co3.ps

[23] ファイル名:u01inar/ps/co4.ps

・ Synthesis &Fitting タブの [Use timing-driven complication to achieve performancegoals] のチェックボックスがオンになっていることを確認する。

＊ Verification タブでコンパイルプロセスの最後に自動タイミング解析やバッチシミュレーションの両方、もしくはどちらかを実行するか指定する

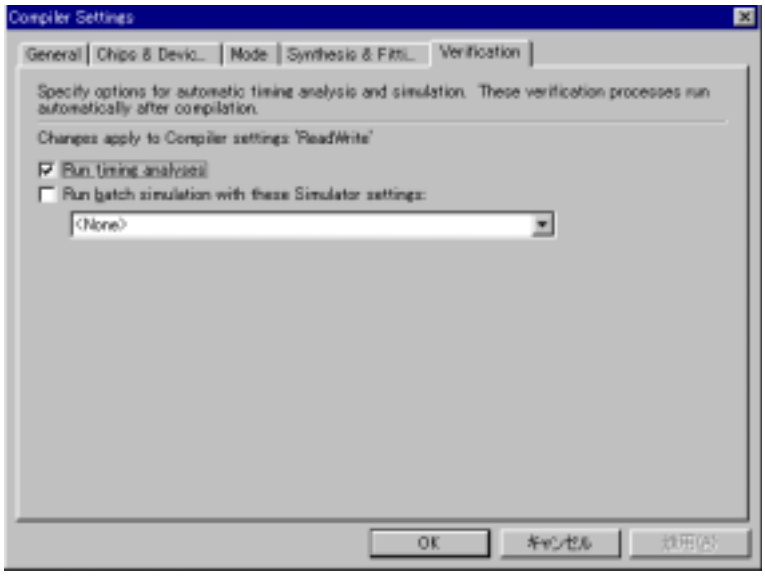


図 3.24: Verification タブ [24]

・ Verification タブの [Run timing analyses] のチェックボックスがオンになっていることを確認する。

– コンパイルの実行

＊ ツールバーコンパイルの実行を選択

図 3.25: コンパイルの実行 [25]

コンパイラを実行すると ウィンドウの左側にパーセンテージ、コンパイル全体の所要時間、各ステージごとの所要時間が表示される。

[24] ファイル名:u01inar/ps/co5.ps

[25] ファイル名:u01inar/ps/co6.ps

図 3.26: コンパイルの実行 [26]

* メッセージソースの表示

  コンパイルの間、すべてのコンパイルメッセージがウィンドウ下側の [Processing] タブに表示される。

図 3.27: メッセージソースの表示 [27]

* コンパイルレポートの表示

  コンパイルの間、右側のウィンドウにコンパイルレポートが表示される。ここには現在のコンパイルに関する情報が表示される。[Summary] セクションにはコンパイルする回路のエンティティ名、ロジックセルと FPGA で使用するピンの合計数、使用メモリの合計量などが表示される。

[26] ファイル名:u01inar/ps/co7.ps

[27] ファイル名:u01inar/ps/co8.ps

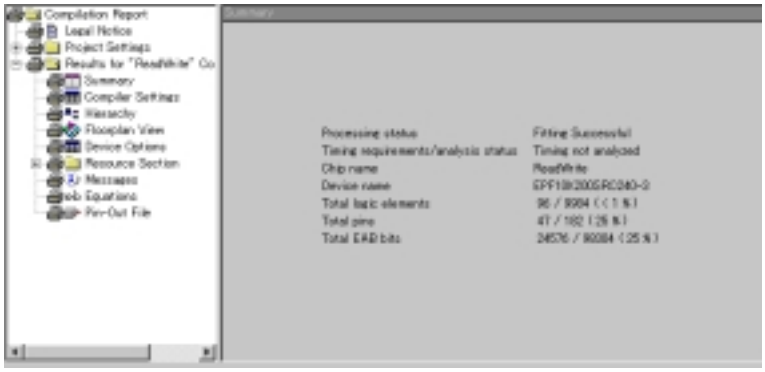


図 3.28: コンパイルレポートの表示 [28]

– エラーメッセージの表示
  メッセージソースにエラーメッセージが表示されたら、そのエラーメッセージに従い、エラーを修正する。

– コンパイラの終了コンパイルがエラーなく終了すると図 3.29のようにメッセージが表示される。

図 3.29: コンパイルの終了 [29]

C.2.2 タイミングシミュレーション

配置配線の後に実際に FPGA が思い通りに動作させることができるか確かめる方法にタイミングシミュレーションがある。タイミングシミュレーションを行うことによってデバイスプログラミングを行うかどうか判断する。うまくシミュレーションできない場合は再度 VHDL を記述し直す。

– Vector Wave File を作製する
  * ツールバーの Simulate Mode をクリックして Simulate Mode にする

[28] ファイル名:u01inar/ps/co9.ps

[29] ファイル名:u01inar/ps/co10.ps

図 3.30: Simulate Mode [30]

* File → New を選択する
* New ダイアログボックスが表示され、 Other Files タブを選択する
* そこで Vector Wave File を選択する

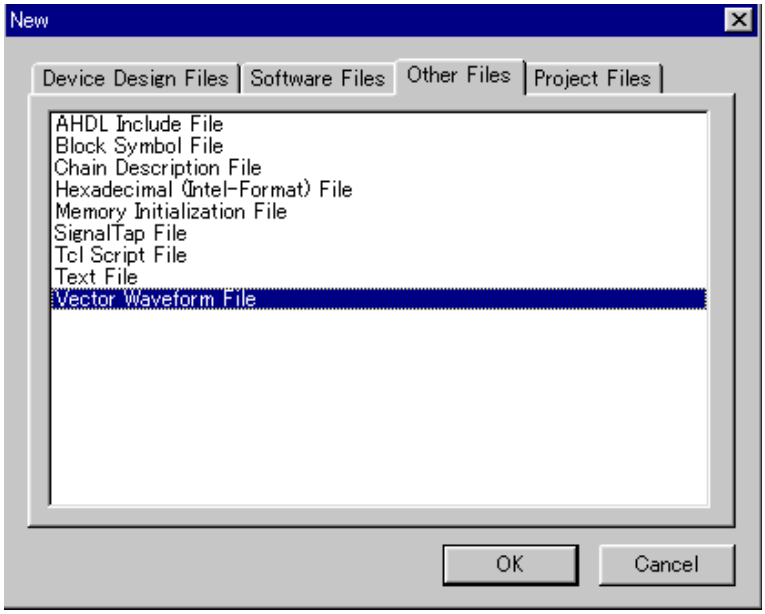


図 3.31: Other Files タブ [31]

* ツールバー Time で End Time と Grid Size を設定する。 Grid Size はクロック周期の半分に設定する
* ツールバー Save をクリックして保存する

– 入力ノードと出力ノードを追加する

* ツールバー View → Auxilialy Windows → Node Finder を選択する

[30] ファイル名:u01inar/ps/sim0.ps

[31] ファイル名:u01inar/ps/sim1.ps

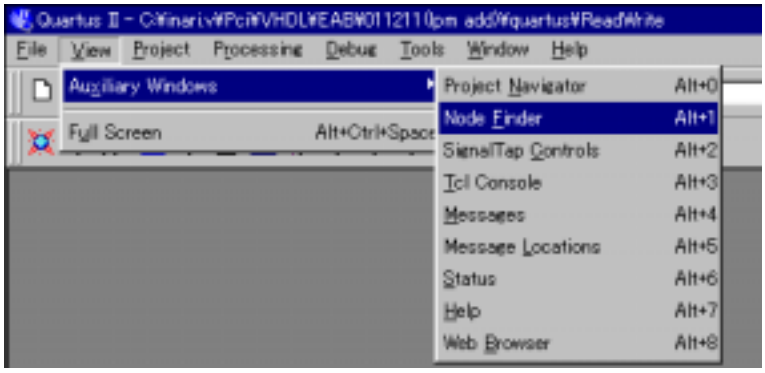


図 3.32: Node Finder の選択 [32]

* ウィンドウの左側に Node Finder が表示される
* Node Finder の Start をクリックするとポートで定義した信号が表示される。

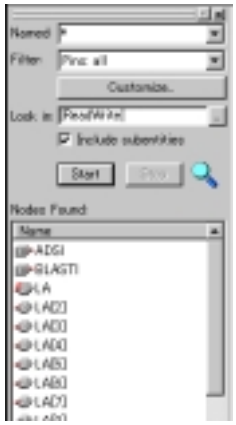


図 3.33: ポートの表示 [33]

* シミュレーションで必要な信号をドラッグ & ドロップで Vector Wave File の Name の欄に持ってくる

– 入力ノードの波形を編集する

* 入力ポートを選択する
* 波形を編集するときはツールバー に表示された波形編集ツールバーを利用して編集する。図 3.34参照

図 3.34: 波形編集ツールバー [34]

– シミュレーションの実行

[32] ファイル名:u01inar/ps/sim2.ps

[33] ファイル名:u01inar/ps/sim3.ps

[34] ファイル名:u01inar/ps/sim4.ps

* ツールバー Run Simulate をクリックする

図 3.35: Run Simulation [35]

* コンパイラの時と同様にコンパイラプロセスの各所要時間、シミュレートレポートが表示される。
* シミュレーションが行えたら出力ポートに信号が表示される

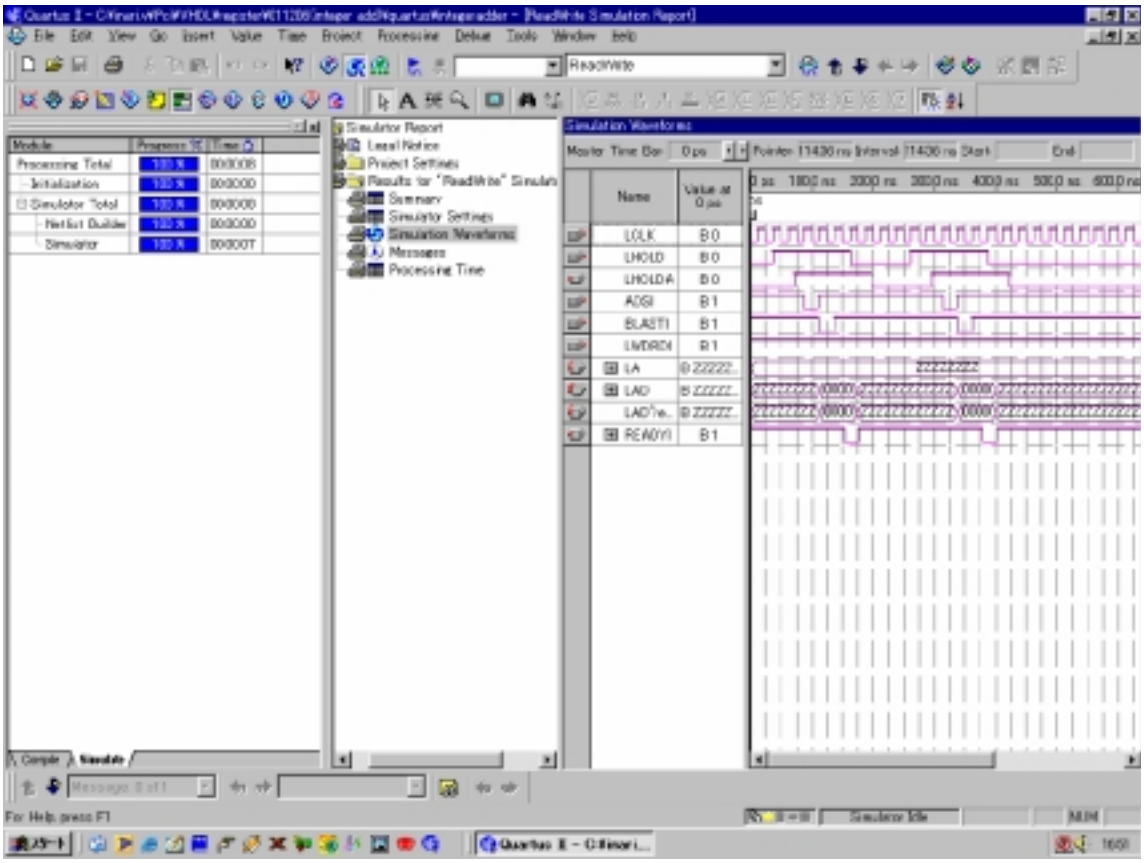


図 3.36: シミュレーション結果 [36]

C.2.3 デバイスプログラミング

- ツールバー File → New を選択
- New ダイアログボックスが表示され、 Other Files タブを選択する

[35] ファイル名:u01inar/ps/sim5.ps

[36] ファイル名:u01inar/ps/sim6.ps

– そこで Other Files タブを選択して、 Chain Description File を選択して OK をクリック

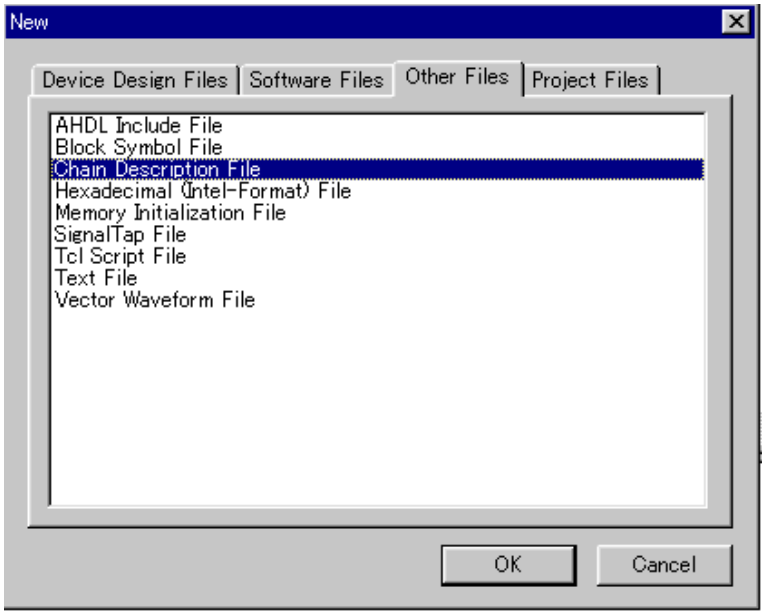


図 3.37: Chain Description [37]

– Chain ダイアログボックスが表示され、 Mode は JTAG であることを確認したら、 Add File を選択

図 3.38: Add File を選択 [38]

– Select Programing File ダイアログボックスが表示され、拡張子が sof(Sram Object File) のファイル選択する

[37] ファイル名:u01inar/ps/pr0.ps

[38] ファイル名:u01inar/ps/pr1.ps

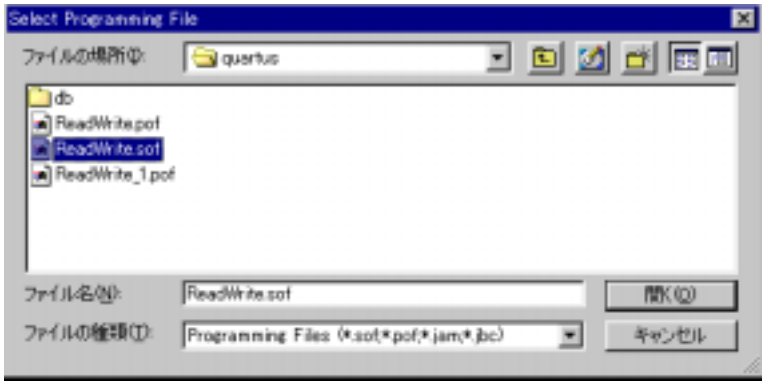


図 3.39: sof ファイルの選択 [39]

– Chain ダイアログボックスに sof ファイルが表示されたら、Program/Configure のチェックボックスを ON にする

– Chain ダイアログボックスの Start をクリック

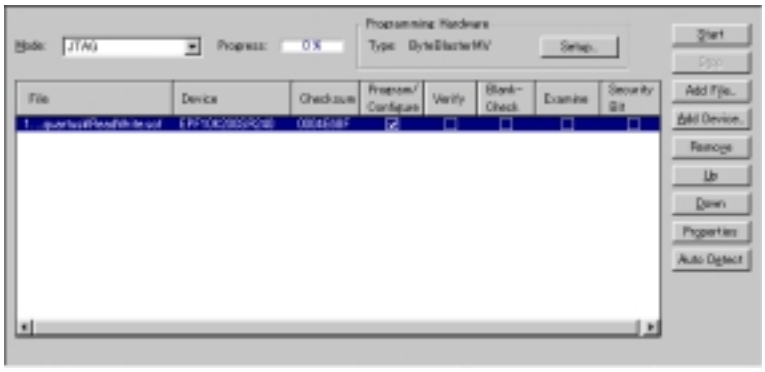


図 3.40: Start をクリック [40]

– デバイスプログラミングが終了するとメッセージボックスに終了のメッセージが表示される

[39] ファイル名:u01inar/ps/pr2.ps

[40] ファイル名:u01inar/ps/pr3.ps

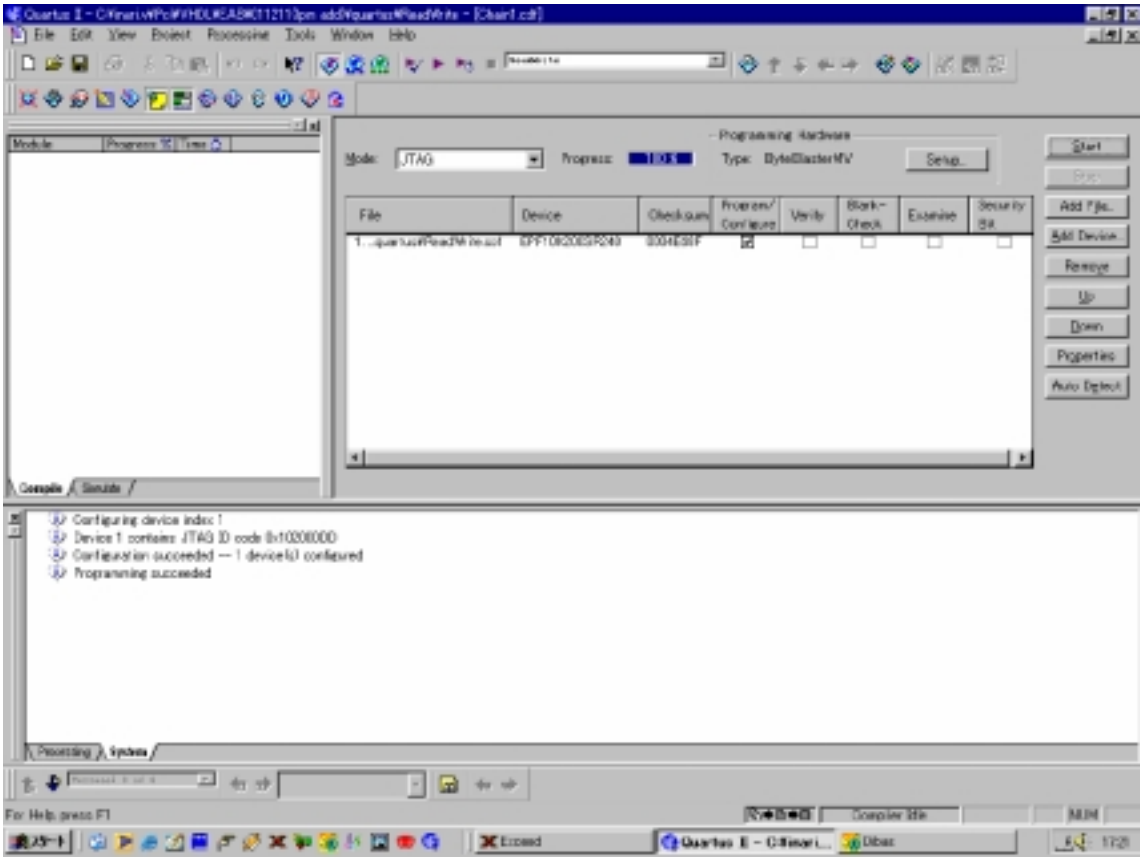


図 3.41: デバイスプログラミングの結果 [41]

C.3 WinDriver の使い方

WinDriver の使用方法について説明する。 WinDriver は PC に接続している PCI デバイスを検索、表示し、そのデバイスに対してのドライバを作成するときに使用する。
WinDriver はコンフィグレーション空間にテストしたものを元にドライバ作成の雛形を出力する。この雛形をユーザ個人が編集することによりドライバ、 PCI 評価基板を制御するソフトウェアを作成することができる。

- WinDriver の起動
  WinDriver を起動し、 Create a new driver を選択する。

[41] ファイル名:u01inar/ps/pr4.ps

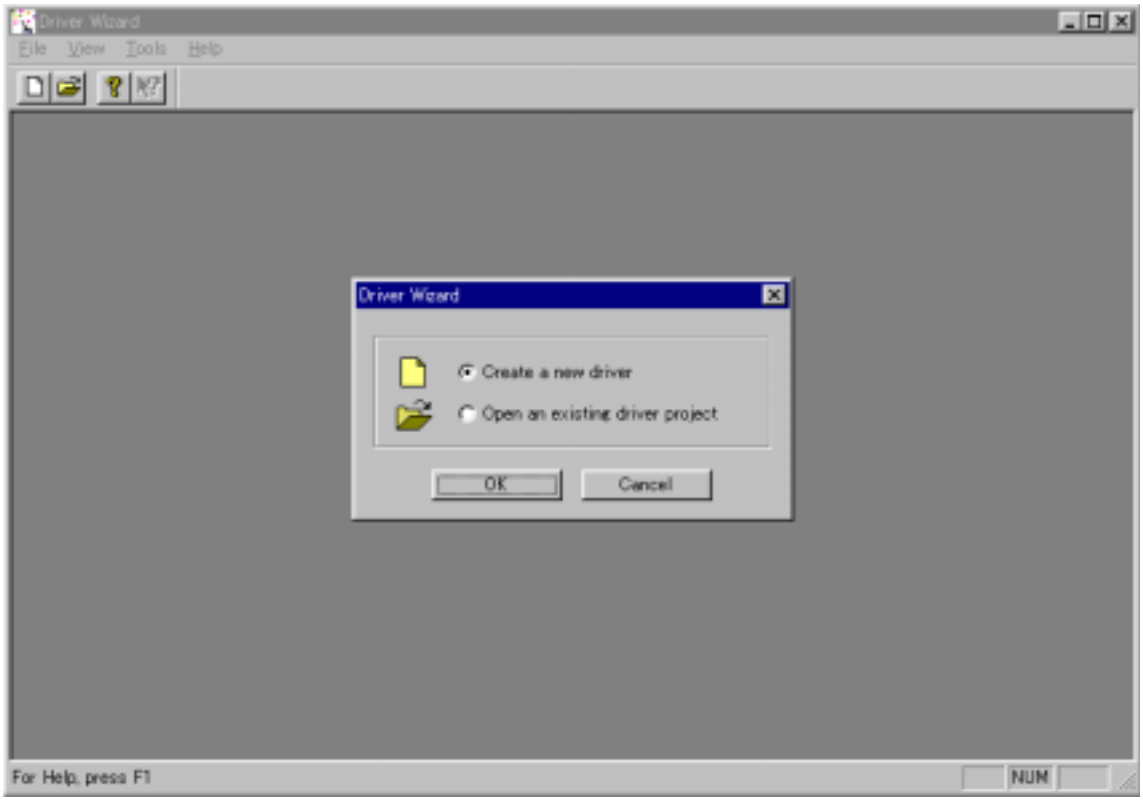


図 3.42: WinDriver の起動 [42]

- デバイスの選択

  Create a new driver を選択すると、 Card infomation が表示される。これは使用している PC のすべての PCI バスに接続しているデバイスが表示される。このカードの中から PLX DeviceID 9054 を選択し、 OK をクリックする。

[42] ファイル名:u01inar/ps/wind0.ps

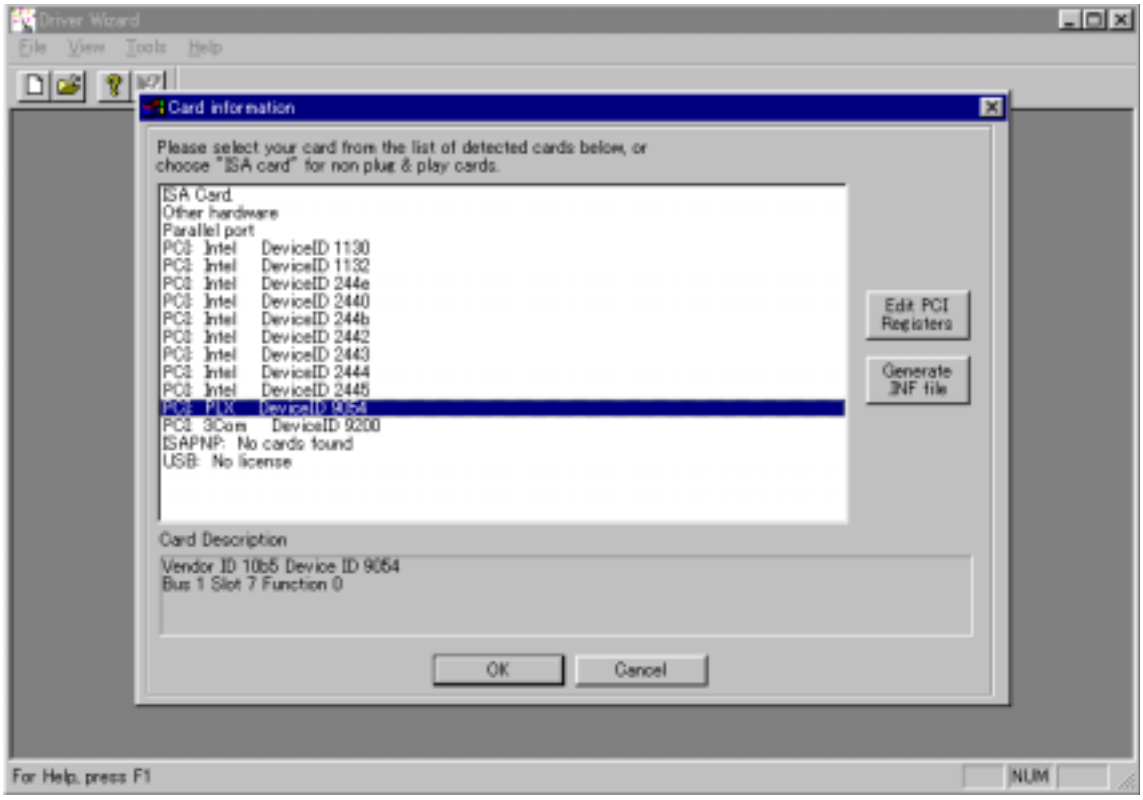
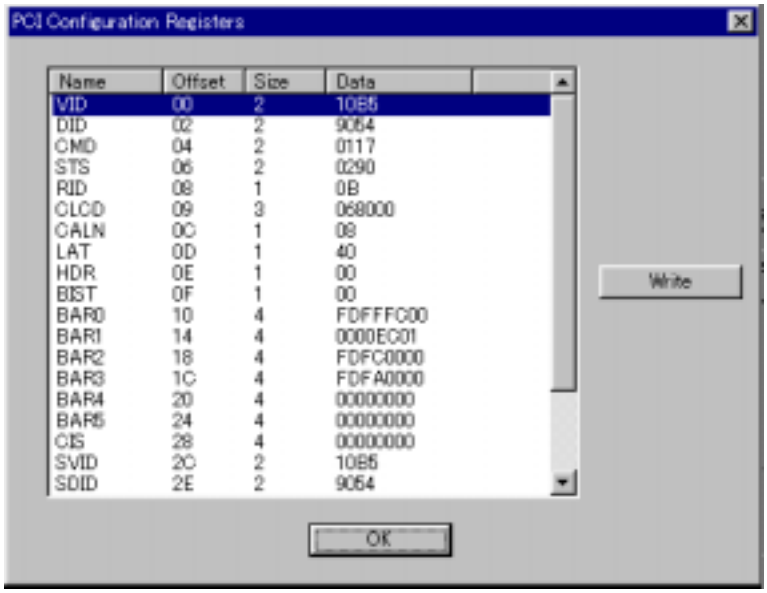


図 3.43: デバイスの選択 [43]

- コンフィグレーション空間の表示

  Card information ダイアログの画面図 3.43で Edit PCI Registers ボタンをクリックすると選択したデバイスのコンフィグレーション空間が表示される。



[43] ファイル名:u01inar/ps/wind1.ps

図 3.44: PCI 9054 のコンフィグレーションレジスタ [44]

これは編集する必要はない。

- コンフィグレーション空間のベースアドレスレジスタ (BAR) の選択

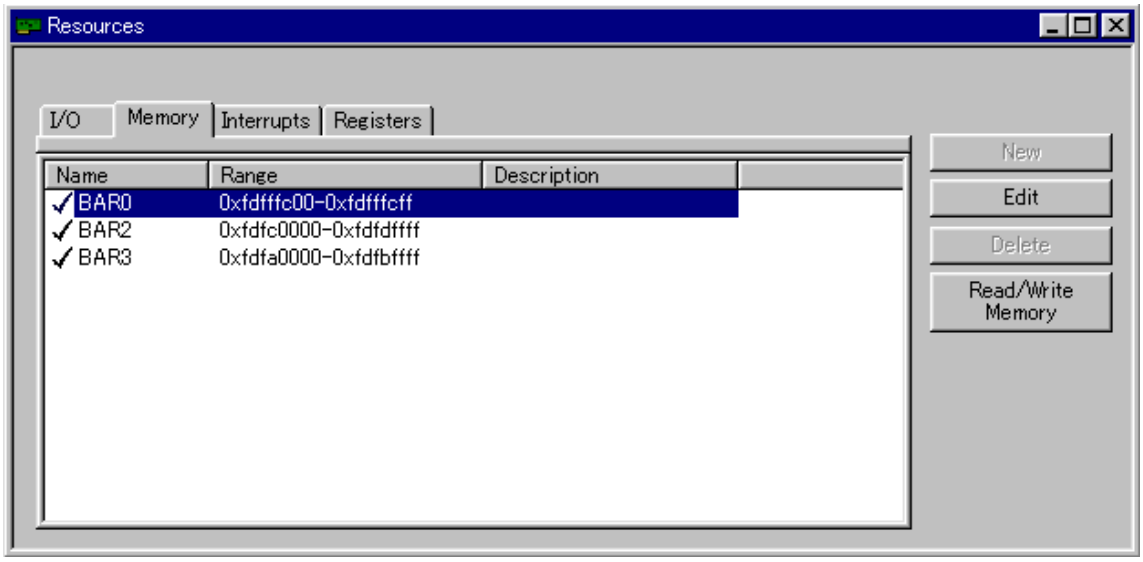


図 3.45: ベースアドレスレジスタの選択 [45]

次にコンフィグレーション空間へのアクセスを指定する。 BAR0 から BAR3 までコンフィグレーション空間で定義された機能別に別れている。 BAR1 は I/O アクセス。 BAR0、BAR2、 BAR3 はメモリアクセスとなっている。

- コンフィグレーション空間へのテスト

[44] ファイル名:u01inar/ps/Win

[45] ファイル名:u01inar/ps/wind2.ps

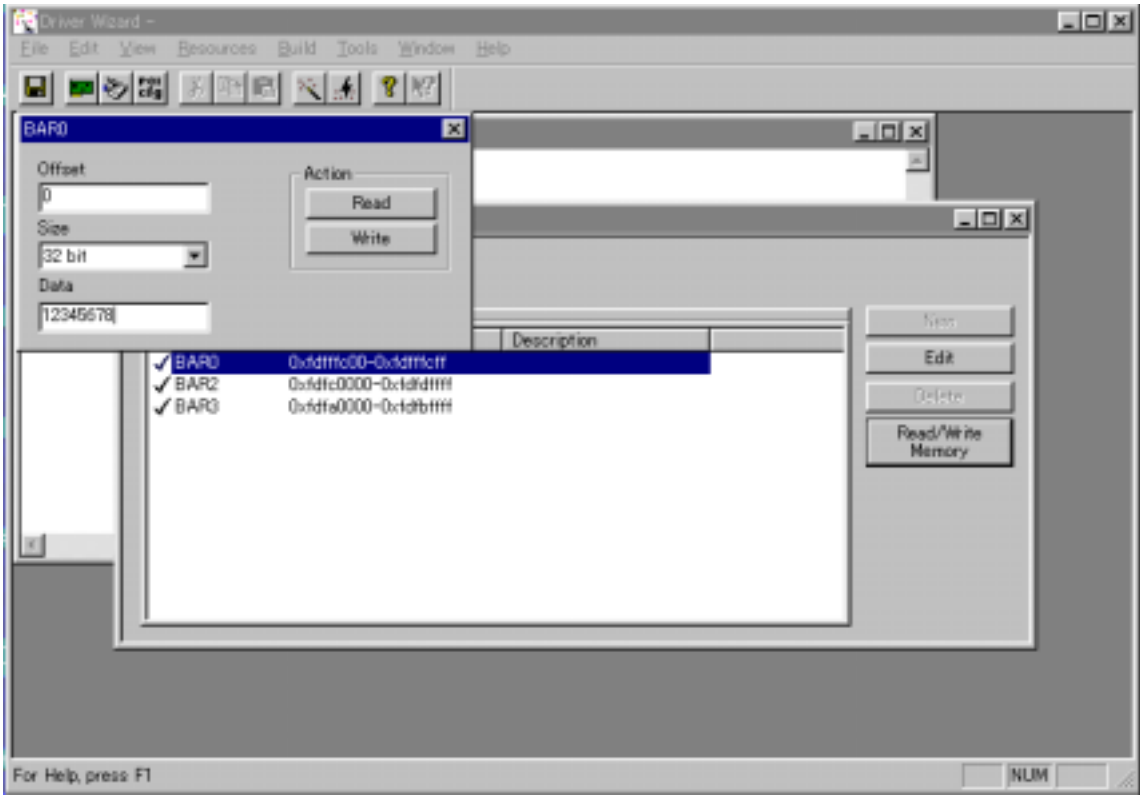


図 3.46: デバイスプログラミングの結果 [46]

BAR0 から BAR3 までの一つを選択したら Read/Write I/O または、 Read/Write Memory ボタンをクリックして選択したベースアドレスレジスタに対してテストを行う。テストでは適当にデータを 16 進数でダイアログボックスにデータを入力し、 32bit アクセスを Write、 Read の順でクリックすることにより選択したベースアドレスレジスタにたいして読み書きを行う。この一連の動作が log に表示される。 PC がフリーズすることなくベースアドレスレジスタに対して読み書きが行えたら、ドライバの雛形を出力する。

[46] ファイル名:u01inar/ps/wind3.ps

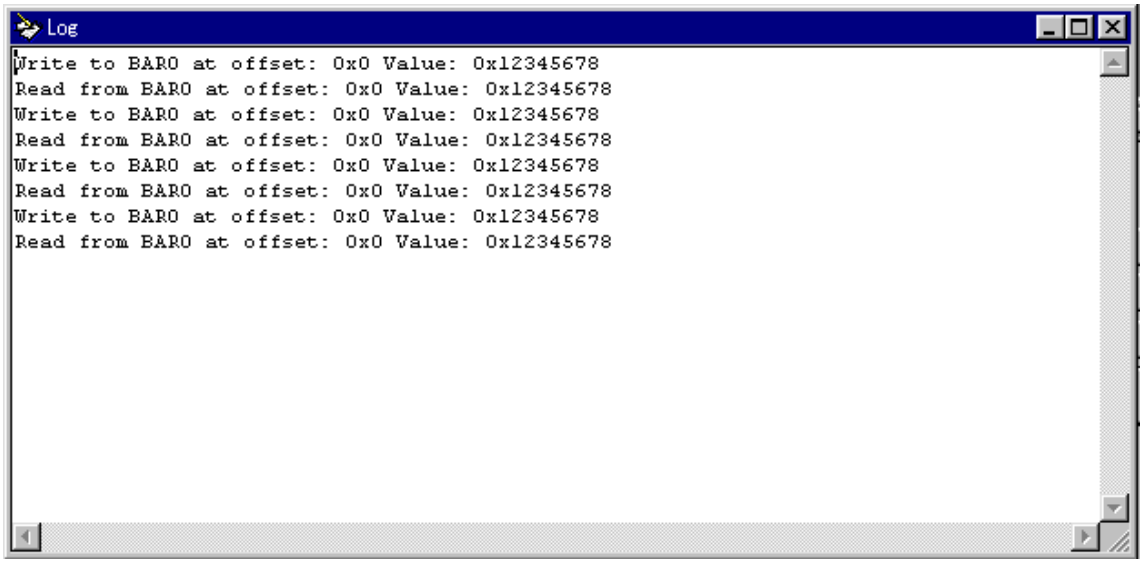
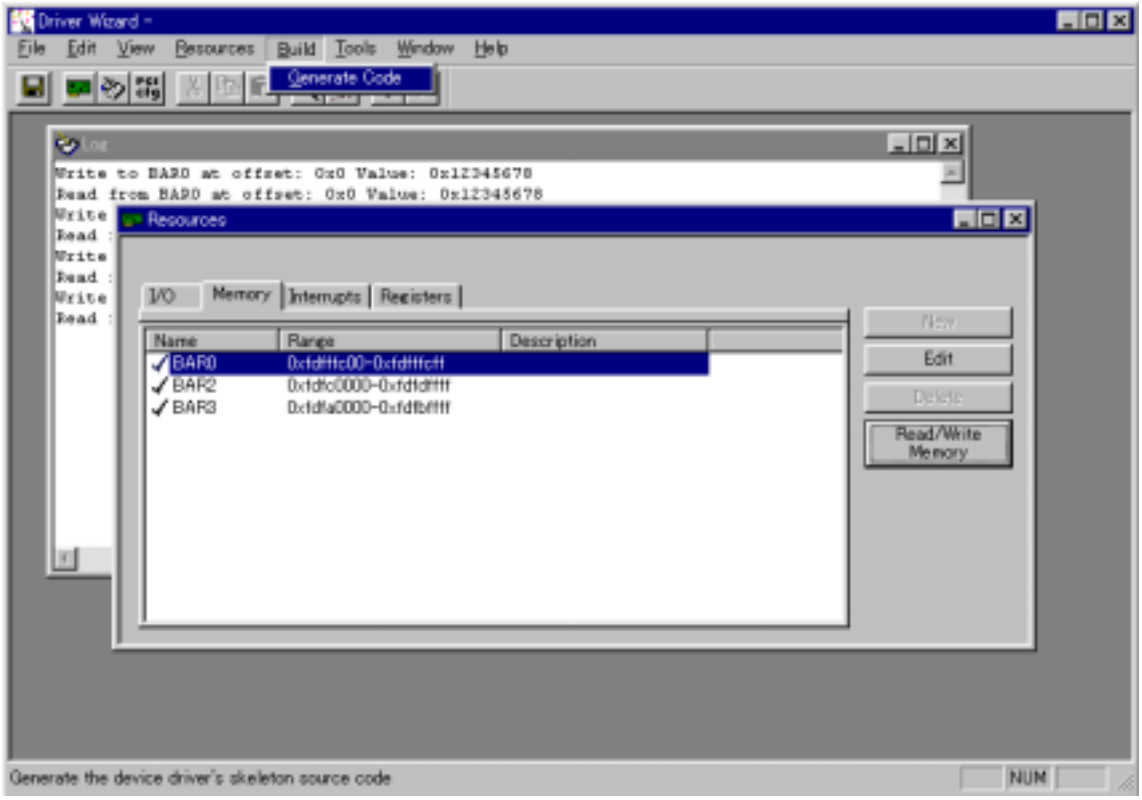


図 3.47: ログの表示 [47]

- Visual C++ のプログラムソースとして雛形を出力
  - ツールバー Build→ Generate Code を選択



[47] ファイル名:u01inar/ps/wind4.ps

図 3.48: Build→ Generate Code [48]

– 拡張子.wdp にファイル名を入力し、保存する。このファイルの保存先は Visual C++ のファイルを保存するフォルダを指定する。

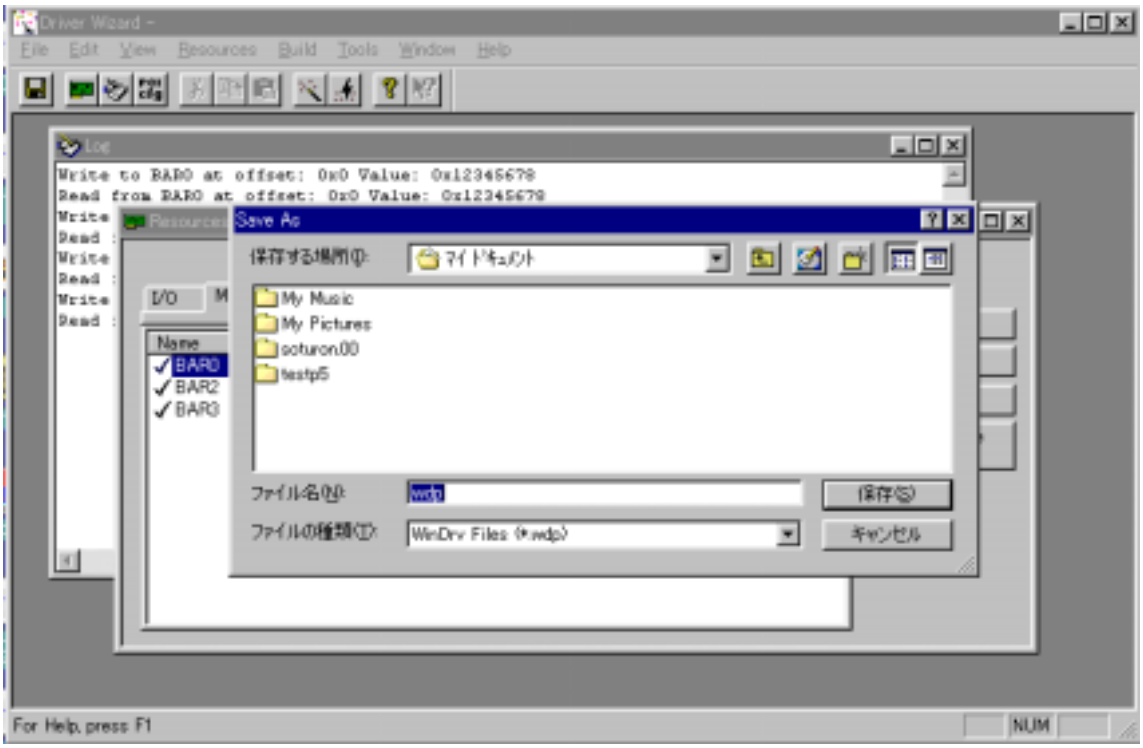


図 3.49: wdp の保存 [49]

– Generate Code ダイアログが表示され、 MS Developer Studio 6 にチェックを入れ、Generate Code ボタンをクリックする。

[48] ファイル名:u01inar/ps/wind5.ps

[49] ファイル名:u01inar/ps/wind6.ps

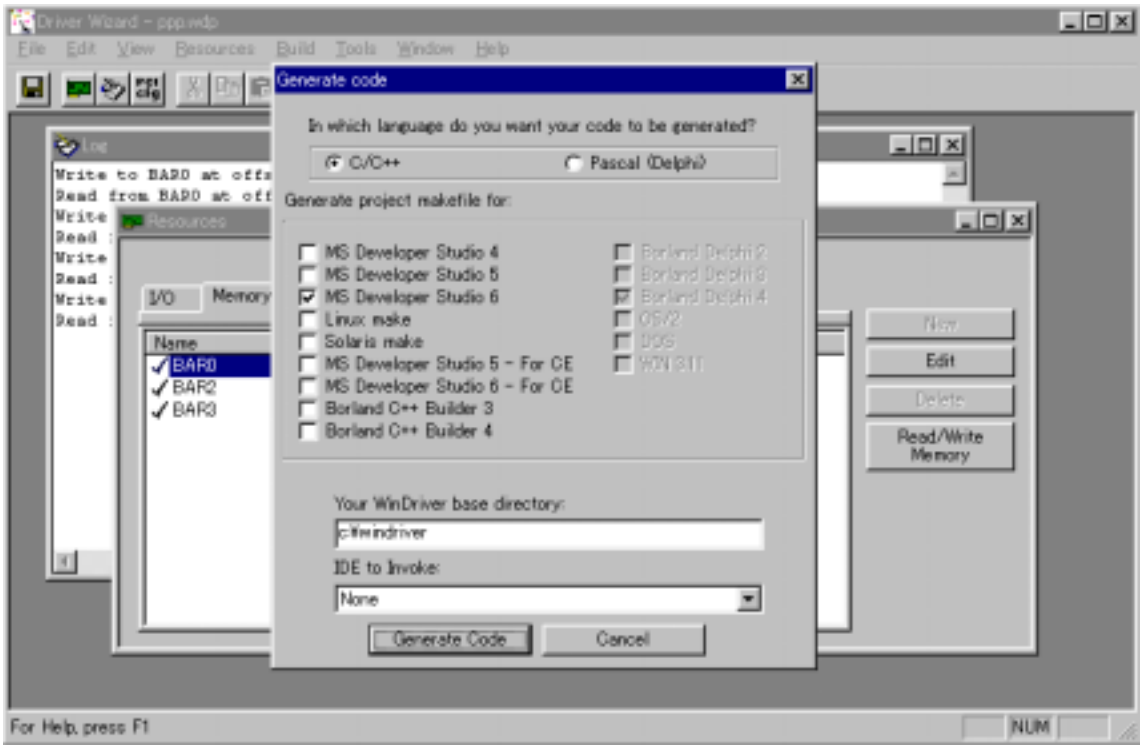


図 3.50: Generate Code を選択 [50]

[50] ファイル名:u01inar/ps/wind7.ps