2000 年度 卒業論文

ハードウェア記述言語を用いた専用デバイスの設計

電気通信大学 電気通信学部 電子工学科

9610069　清水 信貴

指導教官 齋藤 理一郎 助教授

提出日 平成 13 年 2 月 8 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、山岡寛明氏様、ホー・フィ・クー様、沼知典様に多大なる感謝をいたします。特に沼知典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、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浮谷光明様をはじめ、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 では数日以上かかり、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。

我が研究室では昨年まで行列の対角化を使った行列の固有値や固有ベクトルを求める専用プロセッサの作製に取り組んで来た。しかし昨年の沼 [6] 現在、行列の次数が上がると正しくない値が帰ってくる。さらに、動作クロックを早く (10MH-20MHz) すると動作しなくなるという問題がでている為、専用プロセッサの完成までは至っていない。

1.2 前年度までの研究成果

本研究は、当初画像技研（株）との共同研究として、科学計算を高速に行なうために、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専用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、本研究室で行なわれている量子力学における分子軌道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および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ていたからである。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成するための設計方法を決めた。研究室で設計を行なうために、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 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 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・配線ツールとして MAX+plusII の無償提供を受けた。そして、同社から FLEX10K 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) いったメモリが搭載可能な基板を設計、製作した。そして、PC とこの基板間でデータの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 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作成した。

'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 との間でデータの通信を行なうための VHDL を設計し、 PC と FPGA 間の通信を行なった。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(Static Random Access Memory) に記憶させることができた。

それから、固有値計算を行なうための準備として、積和器の設計を行なった。この積和器は行列の計算を行な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逆反復法などの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、メモリー量に制限がある。

'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成

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成した。 DRAM は SRAM に比べてメモリー量が 8 倍と大容量であるが、 SRAM にアクセスするより 3 倍ほどクロックが必要であり、またリフレッシュと呼ばれる電荷を補充する動作が必要になる。

1.3 目的

本研究の目的は昨年まで取り組んできたベクトルの固有値と固有ベクトルを求める専用プロセッサの完成である。しかし昨年までの結果では完成には至っておらずまた問題点も明確に指摘されていない。よって、各種演算を実装し各モジュールの動作を検証していき、また新たに導入した各種ソフトウェアを本研究で使用するためにカスタマイズする。

1.4 本論文の構成

第 3 章において本研究を行う上での各機材、ソフトウェアの説明を記す。第 4 章では行列計算の大元であるベクトルの内積に付いて説明する。第 5 章においては行列の乗算について述べる。第 6 章では行列の対角化を説明する。また、付録としてこの研究で使用する全てのソフトウェア使用法を詳しく解説する。さらに教育用の FPGA 搭載ボードの使用法も併せて解説する。

1.5 用語説明

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 Descripton Language の略であり、米国国防省において 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。その理由はよりグローバルな研究をする上では世界標準が重要であるという考えによるものである。

- FPGA
  Field Programmable Gate Array の略であり、書き換えが可能なゲート素子の事である。本研究で使用する FPGA はアルテラ社製の FLEX10K100 である。これはゲート数が 10 万あると言うことである。

- PC
  Personal computer の略であり本研究で使っている PC のスペックは
  CPU PentiumIII 1GHz
  Memory 512MByte
  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図を 2.1に示す。まず PeakFPGA という VHDL エディタ で回路機能 を記述し、 HDL ファイル (*.vhd) を作製する。もし、 VHDL 構文の記述がおかしければ、コンパイル中にエラーメッセージが出る。その場合は正しい文章に書き直し、再度コンパイルする。そして、この機能検証を実行するために、テストベンチをおこなう。これは、デバイスの入力ポートにデータを入力する動作を VHDL で記述したものをシミュレーションでデバイス内の各信号の動作を検証することである。このシミュレーションで信号の動作が正しくなければ、 VHDL ファイルを書き直しシミュレーションに戻り、信号の動作が正しくなるまで書き直す。

本研究の場合には、 FPGA でのピンの配置も VHDL ソース作成の時点で指定する必要がある。この場合には、 entity の部分で port 文で入出力ピンを指定し、 attribute 文でピン番号を指定する必要がある。

そして、その回路の動作が正しければ、 PeakFPGA で論理合成を行なう。論理合成とは HDL によって記述された回路機能を AND や OR などの論理回路の形に変換することである。これにより HDL ファイルは EDIF(Electronic Design Interchange Format) ファイル (*.edf) に変換される。この EDIF ファイルはデジタル回路をテキストファイルで表したファイルで、回路を実際のデバイスに実装するための情報が含まれている。この EDIF ファイルを目的のデバイスにコンフィグレーションする為には、そのデバイスに合う形に変換しなければならない。そこで Max+-Plus2 というソフトウェアが必要になってくる。この Max+Plus2 でデバイスを指定して再びコンパイルする。すると評価基盤上にある FPGA、 FLEX10K に乗る TTF(True Type Format) ファイル (*.ttf) が作製される。この TTF(True Type Format) ファイルを松尾 [3] が作製した C

プログラムで FPGA にコンフィグレーションする。コンフィグレーション後に実際に動作するか検証する必要がある。その為に、 C++Builder を使って評価基盤を制御して検証をする。作製の流れは図 2.1に示す。

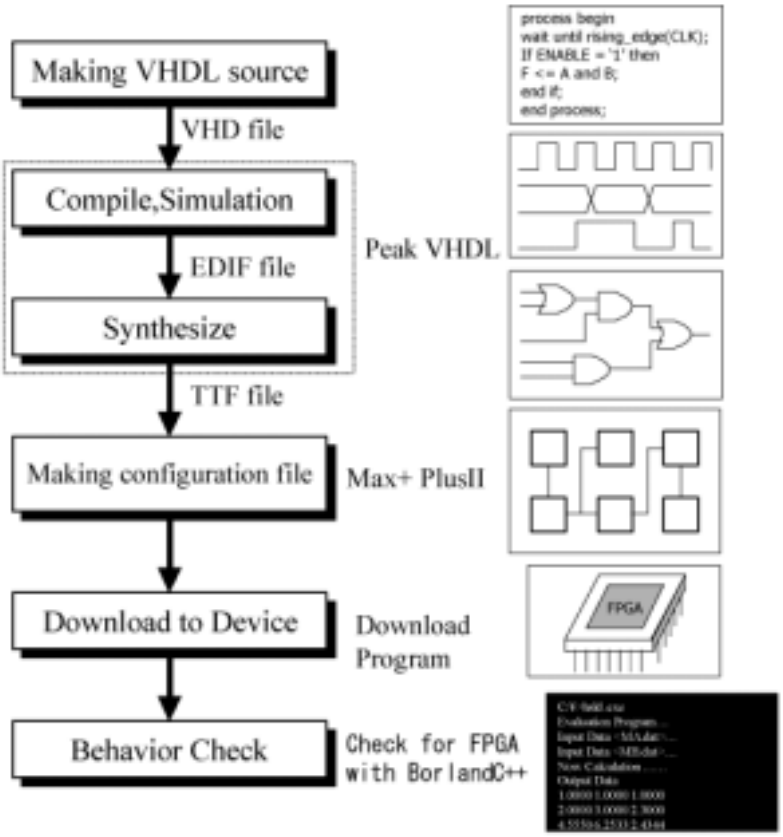


図 2.1: 本研究におけるデジタル回路設計の流れ [1]

2.2 ハードウェア

2.2.1 評価基盤

本研究で使う評価基盤は松尾 [3] によって作られた。この評価基盤の構成は、 Field Progrmmable Gate Array(以降 FPGA) は FLEX10K が 2 つ、 DRAM が各 FPGA に 2 個。 SRAM が各 FPGA に 8 個。インターフェイスは 50pin フラットケーブル。評価基盤上の FPGA は SRAM なので電源が供給されないと、データが消えてしまう。よって毎回コンフィグレーションをする必要がある。また評価基盤上には DRAM(Dynamic Random Access Memory) が各 FPGA の両サイドにあり、片方のメモリーの容量は SRAM が 521KByte、 DRAM が 16MByte ある。 PC から送る

[1] ファイル名:u00simi/eps/flow1.eps

単精度浮動小数点型は 32bit(4Byte) なので SRAM では 131072 個のデータ (512KByte) 格納出来る。

	SRAM	DRAM
アクセス速度	数 ns 以下	60ns
アクセス手続き	2 回	6 回
データ容量 (片側)	512KByte	16MByte
記憶出来るデータ (4Byte) 数	131072 個	4194304 個
リフレッシュ	不要	必要

表 2.1: SRAM と DRAM のスペック

ここでは SRAM 使うので SRAM のアクセス方法について簡単に述べる。まず SRAM にアクセスする為に必要な信号は $\overline{SCS}$、$\overline{SWE}$、$\overline{SOE}$ がある。書込む場合は SRAM のアドレスをセットした状態で $\overline{SCS}$ を立ち下げる。
これで SRAM のアドレスが指定されて、その後 (1 クロック後)$\overline{SWE}$ を立ち下げると指定したアドレスにデータが書込まれる。読み込む場合は書込みと同様に $\overline{SCS}$ を立ち下げてアドレスを指定する。それと同時に $\overline{SOE}$ を立ち下げることで SRAM からデータを読み込む事ができる。

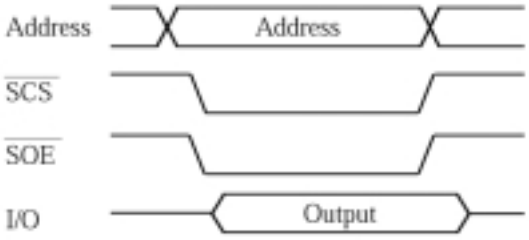


図 2.2: SRAM からの読み込み[2]

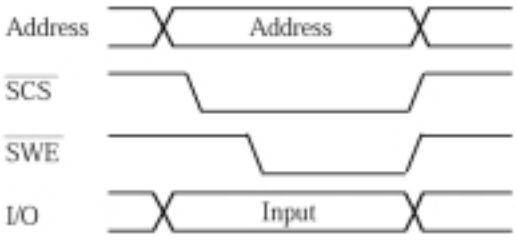


図 2.3: SRAM への書込み[3]

- ハイインピーダンス

SRAM とのデータバス (信号名:DATA) は in:out 両方兼用である。SRAM からデータが送られる時、このデータバスをハイインピーダンスに設定しなければならない。PeakFPGA ではハイ

[2]ファイル名:u00simi/eps/Image3.eps

[3]ファイル名:u00simi/eps/Image4.eps

インピーダンスを ‘Z’ で表現している。また PC と評価基盤との通信も A ポートが in:out 両方兼用なのでここも PC から評価基盤にデータを送る時、 A ポートをハイインピーダンスに設定する。
例 (VHDL 記述)

---

process begin
if OE = ’1’ then
DATA = “ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ”;
end if;
end process

評価基盤の詳しい内容は松尾 [3] の卒業論文に記述されている。

2.2.2 PC- 評価基盤間の通信インタフェイス

評価基盤のコンフィグレーションと評価基盤の制御を行う為に PC と評価基盤の通信が必要になる。そこで松尾 [3] が ISA バス専用の通信インタフェイスを作製した。通信インタフェイスの詳しい内容は松尾 [3] の卒業論文に記述されている。

2.3 ソフトウェア

2.3.1 PeakVHDL

VHDL を記述するにあたり、シミュレーション、論理合成、シミュレーションが出来るエディタが必要になる。そこでインターリンク社より PeakFPGA を購入した。現在のバージョンは PeakFPGA5.20c である。

2.3.2 Max+Plus2

VHDL エディタにより論理合成して出来た EDF ファイルを目標の FPGA にコンフィグレーションするためにコンパイルする必要がある。そこで Altera 社の MAX+Plus2 を使って行うこととする。現在のバージョンは Max+Plus2 10.0 である。

2.3.3 C++Builder

評価基盤を制御する為に Boland 社の C++Builder を使って制御する。制御とは評価基盤上の FPGA に信号を出力、 PC と FPGA 間の伝送路の制御がある。詳しい使い方は付録参照。

2.3.4 pc Anywhere を使った遠隔操作でのコンフィグレーション

PC と評価基盤との通信には松尾 [3] が作製した ISA バス専用のインターフェイスを用いているが、最近の PC には ISA バスが搭載されている PC は皆無に等しい。そこで ISA バス搭載 PC を評価基盤制御 PC として独立させ、その PC を外部の PC から通信で制御させることとする。
詳しい使い方は付録参照。

第 3 章

ベクトルの内積

3.1 目的

行列の乗算とは行、列各ベクトルの内積計算を複数回行う事によって出来る。そこで一昨年度の山岡 [5] と沼 [6] により作製された加算器、乗算器、 SRAM メモリーコントローラを用い、さらに、それらを制御するプログラムを自分で作製し、内積の計算を行うプログラムを作製する。

3.2 方法

3.2.1 実験方法

PC より N 次元のベクトルを 2 つ合計 2N 個のデータを評価基盤上の SRAM に記憶させる。データは単精度浮動小数点型 (32bit) とする。そして内積計算ルーチンを実装した FPGA でそのデータを使いベクトルの内積を計算させ、結果を PC に返す。昨年の沼 [6] は加算器と乗算器を併せた積和器を作製したが動作確認が出来ていないので別々に別けて考えた。結果はスカラーなので 1 個になる。その結果を PC で計算させた結果と比較する。

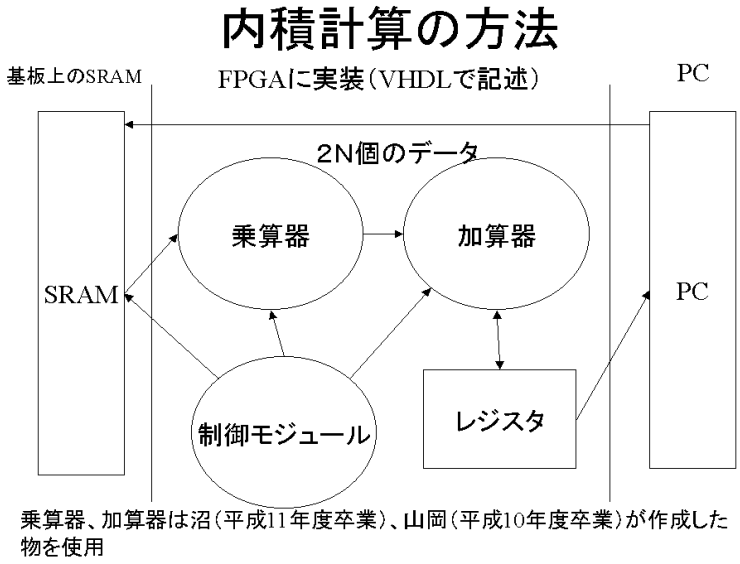


図 3.1: 内積の計算略図[1]

3.2.2 作製

内積の計算を行うには加算器、乗算器、 SRAM コントローラ、それらを制御させる制御モジュールが必要である。しかし制御モジュール以外は以前に作られているので制御モジュールのみを作れば良い。そこで、この制御モジュールで加算器、乗算器、 SRAM コントローラのデータ、 SRAM のアドレス制御、 PC からのデータの入出力を司れば良い。これらを考慮に入れながら制御モジュールを作製した。

3.3 モジュール説明

3.3.1 浮動小数点加算器

この加算器は山岡 [5] によって作製された。この加算器は 3 ステージより成り立っており、各ステージはクロックに同期しているので、データを加算器に送ってから 2 クロック後に結果が帰ってくる。

[1] ファイル名:u00simi/eps/naiseki.eps

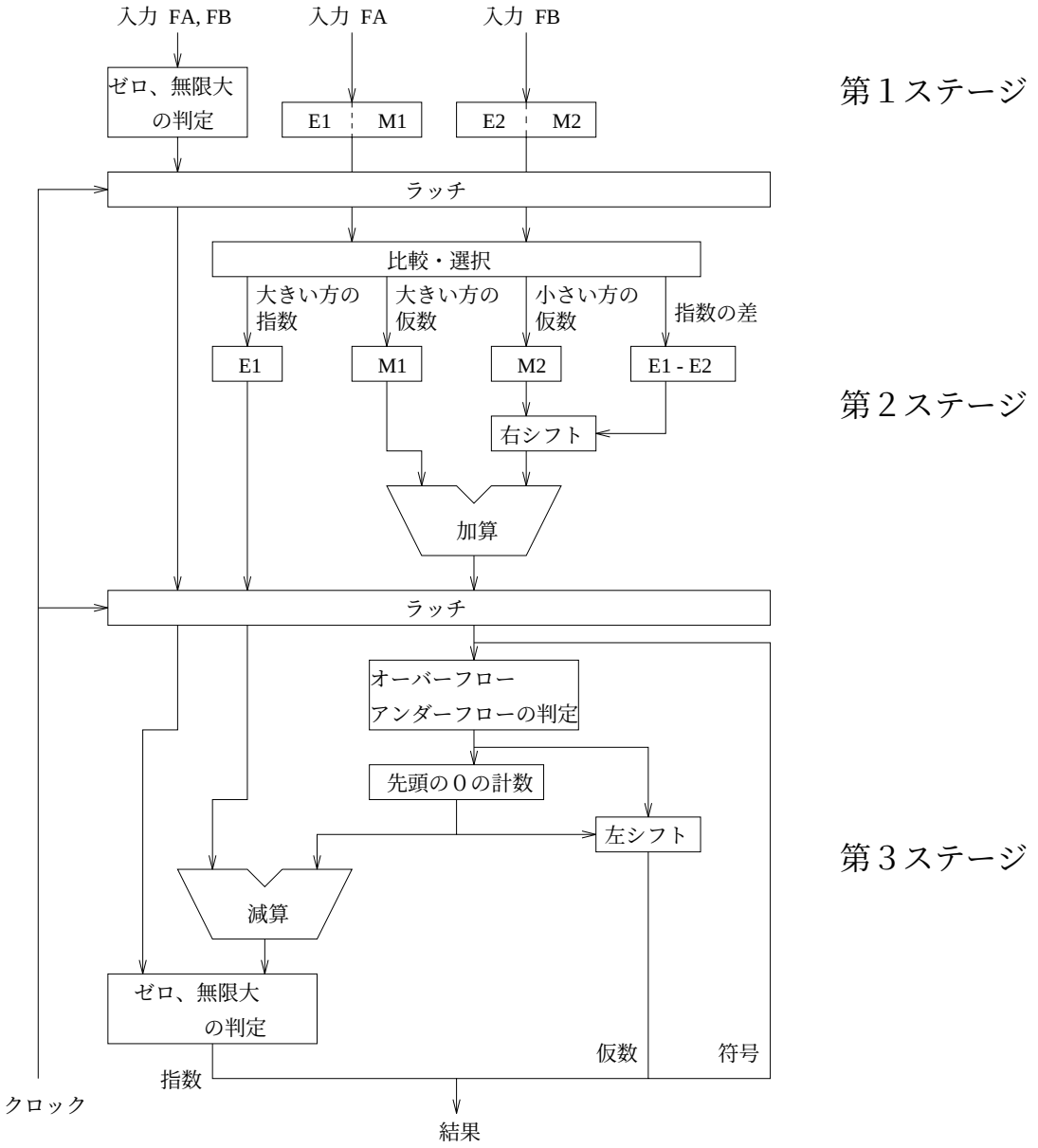


図 3.2: 加算器のステージ図[2]

このステージ構成を図 3.2に示す。第 1 ステージに入る前に、入力するデータ FA,FB はそれぞれ指数部 E1,E2 と仮数部 M1,M2 に分離され、演算が行なわれる。この図に示してあるラッチによって、各ステージはシステムクロックに同期して、独立して動作するようになっている。最初に第 1 ステージでは、 FA と FB のゼロおよび無限大の判定を行なう。指数部がすべて'0' ならばゼロ、すべて'1' ならば無限大とみなす。そして、第 2 ステージでは、 FA と FB の比較を行ない、絶対値の小さい方の数の仮数部 (M2) を指数部の差 (E1-E2) だけ右ビットシフトし、 M1 と M2 の加

[2]ファイル名:u00simi/eps/add-flow.eps

算を行なう。最後に第 3 ステージでは、その加算結果のオーバーフロー、アンダーフローの判定、仮数部の正規化、指数部の調整を行ない、計算結果 Q を出力する。またこの加算器の入出力を 3.3に記す。

信号名	ビット数	方向	用途
FA	32	in	加算する要素 1
SA1(FA)	1	in	FA の符合部
EA1(FA)	8	in	FA の指数部
MA1(FA)	23	in	FA の仮数部
FB	32	in	加算する要素 2
SB1(FB)	1	in	FB の符合部
EB1(FB)	8	in	FB の指数部
MB1(FB)	23	in	FB の仮数部
Q	32	out	加算された結果

表 3.1: 加算器の入出力配線

3.3.2 浮動小数点乗算器

この乗算器も一昨年の山岡 [5] によって作製された。加算器と同様に 3 ステージより成り立ち、各ステージもクロック同期である。

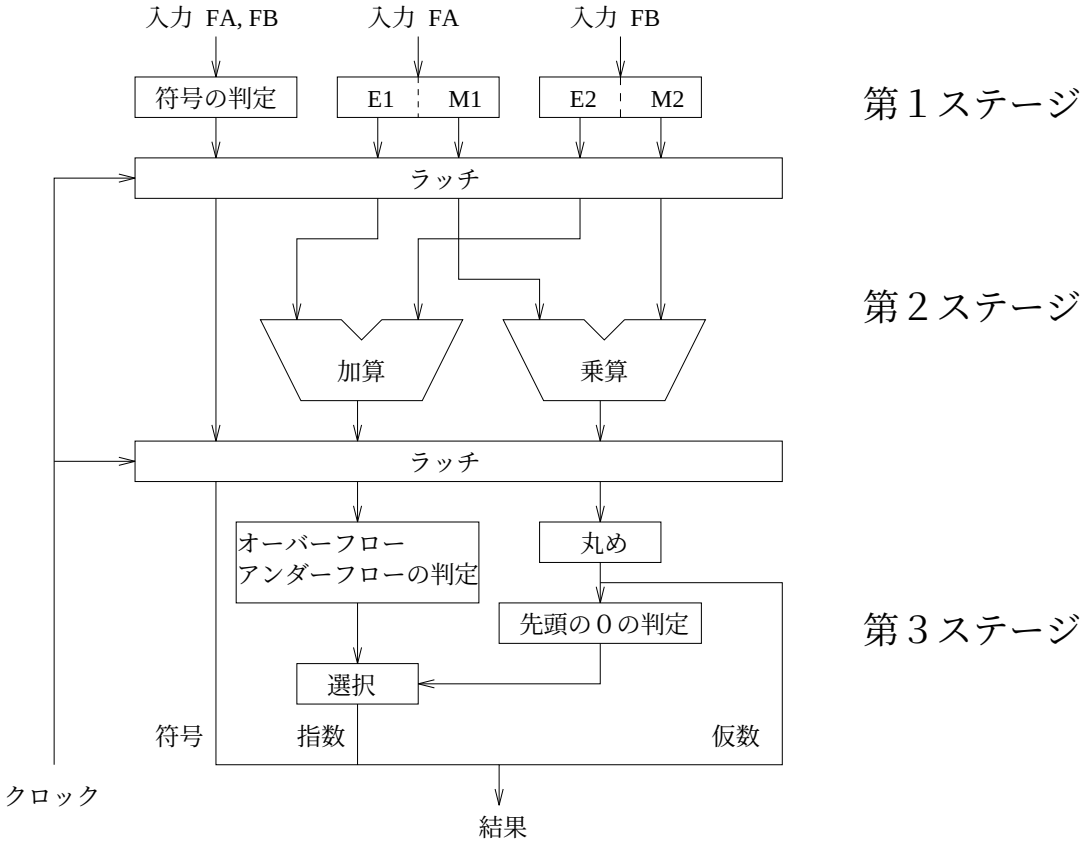


図 3.3: 乗算器のステージ図[3]

図 3.3に乗算器のステージ図を示す。最初に第 1 ステージでは、積の符号の判定をする。そして、第 2 ステージでは、指数部の加算とともに、仮数部の乗算を行なう、そして、次のステージの計算に不要な乗算結果の下位 22bit を切り捨てる。第 3 ステージでは、指数部の結果のオーバーフローとアンダーフローの判定、仮数部の乗算結果の丸めを行ない、最後に正規化して計算結果 Q を出力する。またこの乗算器の入出力を 3.2に記す。

[3]ファイル名:u00simi/eps/mul-flow.eps

信号名	ビット数	方向	用途
FA	32	in	乗算する要素 1
SA1(FA)	1	in	FA の符合部
EA1(FA)	8	in	FA の指数部
MA1(FA)	23	in	FA の仮数部
FB	32	in	乗算する要素 2
SB1(FB)	1	in	FB の符合部
EB1(FB)	8	in	FB の指数部
MB1(FB)	23	in	FB の仮数部
Q	32	out	乗算された結果

表 3.2: 乗算器の入出力配線

3.3.3 SRAM メモリーコントローラ

この SRAM メモリーコントローラは完全クロック同期型であり、 SRAM への書込みが一連の動作が終了 (次にまた書込みができる状態) するまで 3 クロック。 SRAM への読み込みも一連の動作が終了 (次にまた書込みができる状態) するまで 3 クロックで動作する。メモリーに書込むステートをこの SRAM メモリーコントローラは一昨年の山岡 [5] が作製したものを再構築したものである。以前はメモリーコントローラの中のステートが作動中に別のステートの信号が入ってきたらそのステートが作動してしまうという事になっていたが、今動作しているステートが終了するまで他は受け付けないという方が誤作動する確率が減少すると思われるので変更を加えた。具体的には付録: SRAM メモリーコントローラの中に CNT という信号を加えた事である。この信号は READ、もしくは WRITE が動作中は '1' になっている。この状態では外部からの信号 (ステートの状態変更信号) は受け付けないのである。よってステートが STOP の所に来るまで何も受け付けなく、動作が安定するのである。

信号名	ビット数	方向	用途
CLK	1	in	クロック
ADRS	17	out	SRAM へのアドレス指定バス
ADRS_MUX	17	in	制御モジュールからのアドレス指定
DATA	32	in	SRAM と FPGA とのデータバス
DATA_BUF	32	out	DATA バスに送るデータの一時バッファ
WRITE_DATA_REG	32	in	書込むべきデータがあるレジスタ
READ_DATA_REG	32	out	読み込んだデータを受け取るレジスタ
SCS	4	out	SRAM にアドレスを読み込ませる信号
SOE	1	out	SRAM からデータをはき出させる信号
SWE	1	out	SRAM にデータを書込ませる信号
OE	1	out	ハイインピーダンスを設定する信号

表 3.3: SRAM メモリーコントローラの入出力配線

3.3.4 制御モジュール

この制御モジュールの役割を以下に簡潔に述べる。

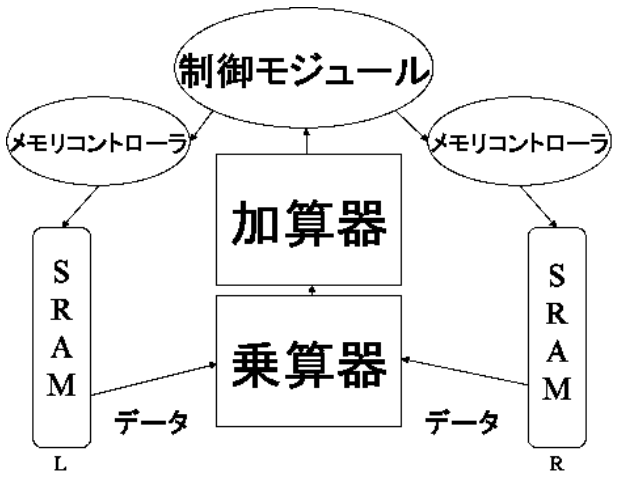


図 3.4: 制御モジュールとデータの流れ[4]

[4]ファイル名:u00simi/eps/naiseki_flow.eps

- PC とのデータ転送を制御

  PC からデータを転送する時、 PC へデータを転送する時にインターフェイスの信号を解析し信号の制御を行う。

  まず、付録: 内積計算制御モジュールの No2 で、 OBF というのがインターフェイスボードからの信号である。この OBF はデータが PC からのデータが インターフェイスボードにラッチされると ‘11‘になる (図 3.5: ア)。この信号がでたら、データをレジスタ、 WRIYE_DATA_REG に入れる。 PC からのデータは 32bit だが、インターフェイスボードのバス幅が 16bit なので 2 回に別けている。そこで 1 回目のデータを下位 16bit、 2 回目のデータを上位 16bit に入れている。 PC からのデータ転送は、内積をとる 2 つのベクトル A、 B とすると、 A を先に全て送る。この A ベクトルは右側の SRAM に記憶される。 A のベクトルを全て送ったら PC から制御信号 ‘01000000‘ を BL に送り、 R_L_CHG を ‘1‘にする。そして次の B ベクトルのデータは左側の SRAM に記憶させる。この OBF は ACK_BUF を経て ACK に代入される用に記述されている (図 3.5: イ)。 ACK とはインターフェイスボードの受取り信号であり、この信号が ‘11‘になるとインターフェイスボードが OBF に ‘00‘ を返す (図 3.5: ウ)。これにより ACK にも ‘00‘ が入って (図 3.5: エ) 初期状態に戻る。

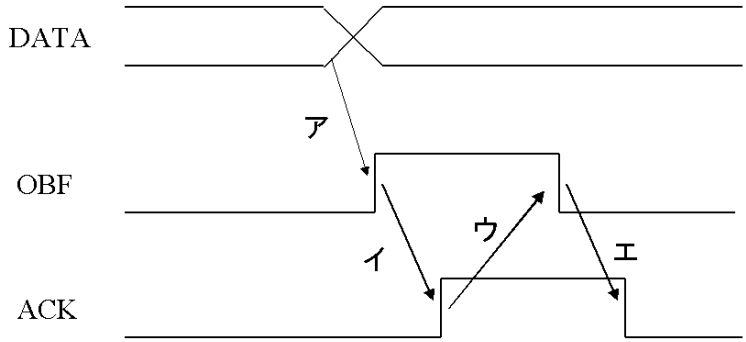


図 3.5: PC からのデータ転送タイミング図[5]

結果を PC に返すプロセスは付録: 内積計算制御モジュールの No5 で行う。結果を返すとき、 PC から制御信号 ‘00000010‘ に続き、 ‘00000000‘を BL に連続で出す。 FPGA では BL の 2 ビット目 (BL(1)) が下がるのを確認して、計算結果があるレジスタ、 QQ の下位 16bit を PC に返す。ここで、 OUT_ CNT が ‘1‘になり、続いて同じ信号が出た時今度は上位 16bit が PC に返される。ここでインターフェイスボードの信号の STB(STB_BUF) を ‘00‘にしないと (図 3.6:1) インターフェイスボードにラッチされない。またこの STB は ‘00’ になるとインターフェイスボードの信号、 IBF を ‘11‘ (図 3.6:2) にする。このタイミングを

[5]ファイル名:u00simi/eps/input_frompc.eps

見計らい、再び STB を'11' に戻す (図 3.6:3)。 IBF は自動に'00'(図 3.6:4) に戻るのでこの信号は再び考慮する必要はない。

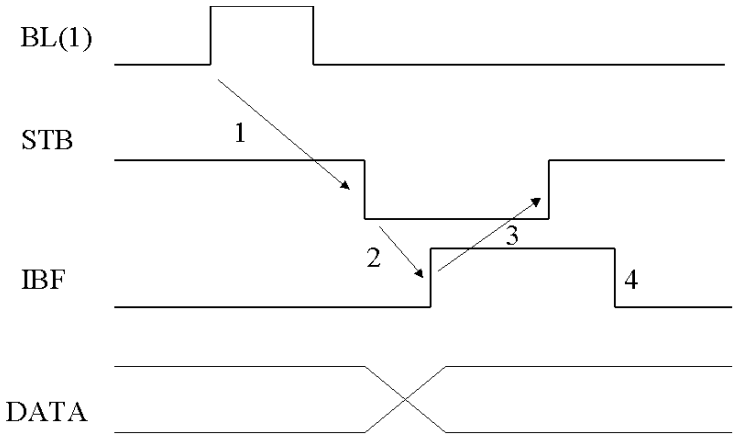


図 3.6: PC へのデータ転送タイミング図[6]

- メモリーのアドレス制御とデータ制御 SRAM にデータを入出力する時のアドレスを設定、メモリーコントローラの動作を制御する。

  付録: 内積計算制御モジュールの No8 で SRAM へのアクセスを制御している。まず PC からデータが出力されたとき (付録: 内積計算制御モジュール No2)、上位 16bit が評価基盤に入って来ると、 WRITE_DATA_ACTIVE が '1' になる。すると付録: 内積計算制御モジュールの No8 で MEM_STATE_SEL を WRITE モードに設定する。これで PC から入ってくるデータが SRAM に格納される。逆に SRAM からデータを取り出す場合は、付録: 内積計算制御モジュールの No9 で指定する。ここは内積計算の一連の動きをステートで表現している。まずここで、 READ_CYCLE2 を '1' にする。すると付録: 内積計算制御モジュールの No8 で MEM_STATE_SEL READ モードに設定する。これで SRAM からデータを読みこむ事が出来る。また SRAM は左右 2 つあるのでそれらを並列に動作させている。なお、 MEM_STATE_SEL は付録: SRAM メモリーコントローラの中で実際のアクセス制御に使用している。

  付録: 内積計算制御モジュールの No4 で SRAM に書込むアドレスを設定している。 NEXT_MEM_CYCLE という信号は付録: SRAM メモリーコントローラの中で定義されている。 SRAM メモリーコントローラステートでステートが動作すると NEXT_MEM_CYCLE が '1' に上がる。それを付録: 内積計算制御モジュールの No4 でそれと同期して ADRS_MUX が上がる用に設定する。 ADRS_MUX は 17bit で定義されているが、これは SRAM のアドレス番地が '0000000000000000' から '1111111111111111' まで定義されている為である。

[6]ファイル名:u00simi/eps/output_topc.eps

これはアドレスのデータ容量と一致している。

- 加算器、乗算器、 SRAM の制御加算器、乗算器にデータを送ったり SRAM からデータを読み込みを制御する。
  先ほど述べた付録: 内積計算制御モジュールの No9 で制御している。これは一連の動作をステートで表現している。ステートで表現する利点として、全ての信号がこのステートで制御するので設計者が分かりやすくなり、また拡張性も優れる。よって第 3 者が見ても分かりやすい。しかし余計なステートを書くのでプログラム自体が重くなってしまうという欠点がある。まず PC から計算されるデータが送り終わったら、 PC から制御信号を評価基盤に送って制御ステートを開始させる。このステートはクロックに同期させている。

  - START
    ステートの始まりである。ここから内積計算が始まる。
  - SRAM_READ1
    READ_CYCLE2 を ‘1‘ にして付録: 内積計算制御モジュールの No8 に送る。これで SRAM を連続 READ モードにする。また V_D_SIGANL を’1’ に立ち上げる。次のステートで立ち下げる事により付録: 内積計算制御モジュールの No8.5 で SRAM から呼び出した回数を計算させる。
  - WAIT1
    V_D_SIGANL を’0’ に立ち下げ次に SRAM から呼び出す回数をカウントする
  - SRAM_READ2
    V_D_SIGANL を’1’ に立ち上げる
  - WAIT2
    V_D_SIGANL を’0’ に立ち下げ次に SRAM から呼び出す回数をカウントし、 SRAM_READ1 で読んだデータを MUTI_CNT を’1’ にすることにより乗算器に入力する。
  - SRAM_READ3
    連続呼び出しによりこのステートでは SRAM からデータを読んでいる。 V_D_SIGANL も’1’ に立ち上げる
  - WAIT3
    SRAM_READ2 からでてきたデータ 2 つを乗算器に出力する。さらに WAIT2 で乗算

器に送ったデータの結果を加算器に送る為 ADDER_DATA_CNT を'0' に立ち下げる。SRAM_READ3 と WAIT3 を繰り返す事により計算を連続で行う。

- WAIT4
  ベクトルの次数分 SRAM データは全て呼び出したが計算はこの段階では乗算が一回、加算が 二回残っているのでさらに次のステートへ計算を続ける為移行する。
- LAST_MULTI
  最後に呼び出したデータを乗算器へ入力しその前の乗算器からの出力を加算器に入力する。
- WAIT5
  計算の待ち時間
- LAST_ADD
  最後に乗算器に入力したデータの結果を加算器に入れる。
- STOP
  ここで計算が終了される。

これらのステートは図 3.7の用に同時にメモリーアクセス、演算を行うので計算時間の短縮が期待できる。

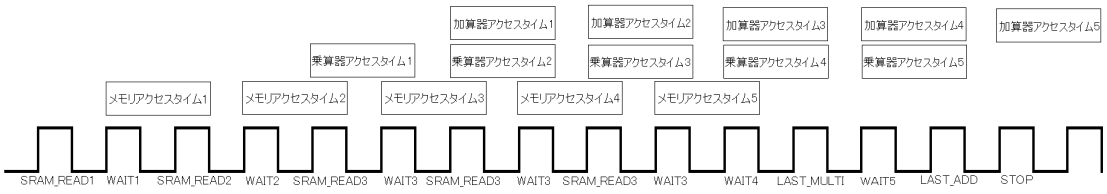


図 3.7: 制御ステートのクロック同期図[7]

3.4 結果

PC でランダムに作製した単精度浮動小数点を評価基盤に転送して、内積計算させた結果と同じデータを PC で計算させた結果と 10000 次数の内積計算速度の周波数特性を記す。PC の結果

[7]ファイル名:u00simi/eps/naiseki_hakei.eps

は 10000000 次数の内積結果である。計算されるベクトルのデータを評価基盤上の SRAM に転送する為に必要な経過時間も記す。誤差率は PC の値を真の値として S、計測値を評価基盤上の値として K とすると

$$\frac{|K-S|}{S}$$

で計算した。

ベクトルの次数	評価基盤の結果	PC の結果	誤差率 (%)
10	21197.982421875	21197.982421875	0
100	231081.171875	231081.171875	0
1000	2615403.5	2615403	$\ll 0$
10000	25448740	25448728	$\ll 0$
100000	253739344	253737584	$\ll 0$

表 3.4: 内積計算の結果

基盤周波数	必要クロック数	秒数	MFLOPS	MFLOPS/1MHz
4MHz	20000	$5.0*10^{(-3)}$	4.0	1.0
8MHz	20000	$2.5*10^{(-3)}$	8.0	1.0
10MHz	20000	$2.0*10^{(-3)}$	10.0	1.0
20MHx	30000	$1.5*10^{(-3)}$	13.3	0.67
CPU 周波数	CPU 名	秒数	MFLOPS	MFLOPS/1MHz
450MHz	PentiumII	0.8	33.2	0.073
1GHz	PentiumIII	0.4	66.4	0.0664

表 3.5: 内積計算の結果

次数	データ数	時間 (s)
100	200	0.001
1000	2000	0.01
10000	20000	0.1
100000	200000	1

表 3.6: PC から評価基盤へのデータ転送時間

3.5 結論

結果より評価基盤の動作クロックに比べるとPCのクロック周波数は遥かに高いが性能は周波数の割りには引けをとらなかった。計算能力も単精度浮動小数点の桁落ちの時の処理がPCとは違うので高次数、単精度浮動小数点の仮数部のデータ数が多くなるとPC の結果とは違いが生じる。しかし誤差率の結果より高次数 (10000 次) での誤差率は $10^(-6)$ というほとんど 0 である。よって正確な計算結果であると言える。
動作クロックに関しては 20MHz の時に計算に必要なクロック数が多くなる。これはSRAM の動作速度が高クロックに対応仕切れていない為である。

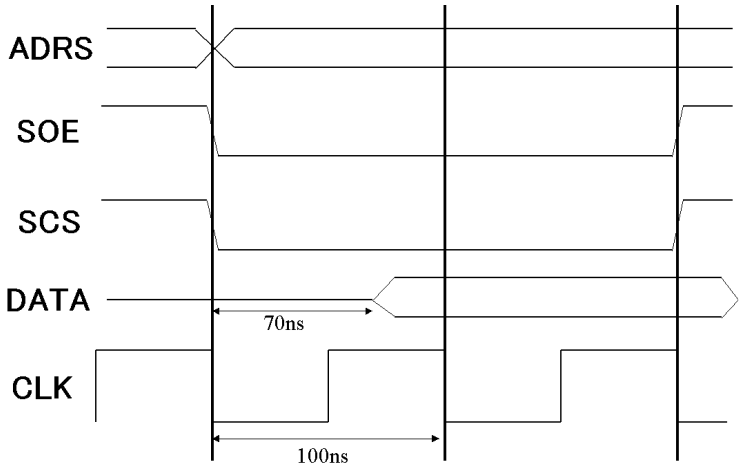


図 3.8: 10MHz での SRAM 動作図[8]

図 3.8は 10MHz での SRAM 動作図である。まずSRAM メモリーコントローラの READ ステートの 1 番目で SOE,SCS を下げ 2 番目で出てくるデータを読む。ここで SCS を下げてから DATA が有効になる時間は約 70ns である。 10MHz の 1 周期の時間は 100ns なので十分許容範囲にある。

[8] ファイル名:u00simi/eps/READ_2CLK.eps

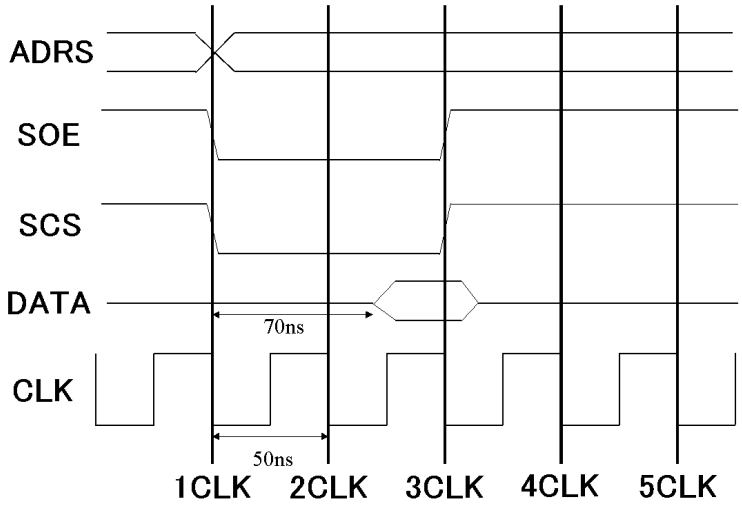


図 3.9: 20MHz での SRAM 動作図 (不完全)[9]

図 3.9は 20MHz での SRAM 動作図である。 2CLK で動作させると、図の通りデータを読み込むべき 2 クロック目ではまだデータが読み込み可能ではない。これは 20MHz の 1 周期の時間が 50ns の為である。そこで読み込むタイミングを 1 クロックずらせば良いのである。

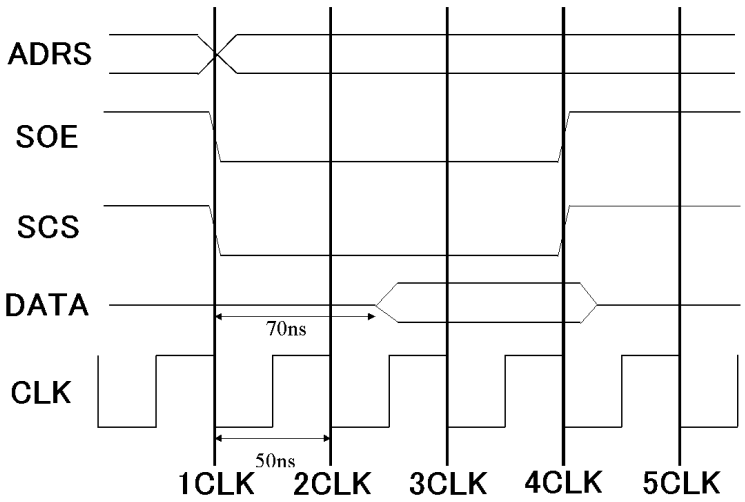


図 3.10: 20MHz での SRAM 動作図 (完全)[10]

図 3.10は正しく動作する 20MHz での SRAM 動作図である。まず 2 クロックで DATA を読み込む所を 3 クロック目で読み込む。このために必要クロック数が 少し多くなってしまう。また更なる高次ではさらに DATA を読み込むタイミングを遅くしなければいけない。
結論で述べている時間は計算の実時間だけであって計算されるデータを送る時間は含まれていな

[9]ファイル名:u00simi/eps/READ_4CLK.eps

[10]ファイル名:u00simi/eps/READ_5CLK.eps

い。データの転送スピードを計測した結果、 1 個のデータを送る為に必要な時間は 5(μs) である。1 つの内積計算時間は実質 2 クロック (0.2μs (動作クロック 10MHZ の時)) なので約 25 倍時間を費やす。以下に 1 億の次数の内積計算をさせた場合の計算時間を出してみる。

クロック周波数	データ転送時間 (s)	計算時間 (s)	総時間 (s)
10MHZ	900	20	920
20MHz	900	10	910
40MHz	900	5	905
1GHz	0	4	4

表 3.7: 1 億の次数の内積計算シミュレート結果

表より評価基盤での総計算時間の大半がデータの転送速度に占められている。これは現在のインターフェイスではこれ以上の時間短縮は望めないのでインターフェイスの改良が望まれる。

第 4 章

行列の乗算

4.1 目的

内積計算の結果においては最新の PC と同等の計算能力を示したがその計算スピードという点では遠く及ばない。そこで行列乗算計算という内積計算より計算回数が多いプロセッサではどのような結果がでるのかを実際にプログラムを作製して結果を比べてみる。内積計算では N 次元のベクトルの場合、乗算回数は N 回、加算回数は (N-1) 回である。しかし行列乗算計算では、 N 次元の正方行列では乗算回数は N^3 回、加算回数は N^{2*}(N-1) である。 N が大きくなると (N-1) を N と見なして、内積計算の 3 乗もの計算回数を必要とする。

4.2 方法

4.2.1 実験方法

前節においてベクトルの内積計算の設計を行い動作を確認した。この原理を行かして行列の乗算を行う専用基盤の作製を行い、内積計算同様、 PC での結果と専用基盤の結果を比較する。 VHDL で行列計算のプロセスを設計し、そのプログラムを評価基盤の FPGA にコンフィグレーションする。その後、行列データを評価基盤の SRAM に格納し行列の乗算を行う。行列計算を行う N 次元の正方行列を A、 B とする。この 2 つの行列のデータ数は 2*(N*N) 個ある。単精度浮動小数点型は 4Byte(32bit)、 SRAM の容量は 512KByte なので N の最大次数 は 362 次である (512KByte/4Byt
しかし行列計算の結果を記憶する場所を考えると最大で 256 次である。そして計算終了後、結果を PC に返してその結果を PC と比較する。

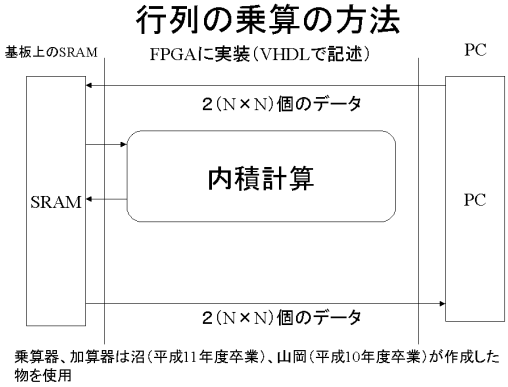


図 4.1: 行列乗算の流れ図[1]

4.2.2 作製

行列の乗算は内積計算の制御と原理は一緒である。よって加算器、乗乗算、SRAM メモリーコントローラはそのまま使用する。しかし制御モジュールは内積計算と異なる。まず N 次元の行列の場合、計算を制御するループが 3 個必要になる。まず行の制御、列の制御、そして次元の要素の制御である。

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & \cdots & a_{1N} \\ a_{21} & a_{22} & a_{23} & a_{24} & \cdots & a_{2N} \\ a_{31} & a_{32} & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ a_{N1} & a_{N2} & a_{N3} & a_{N4} & \cdots & a_{NN} \end{pmatrix} \quad \boldsymbol{B} = \begin{pmatrix} b_{11} & b_{12} & b_{13} & b_{14} & \cdots & b_{1N} \\ b_{21} & b_{22} & b_{23} & b_{24} & \cdots & b_{2N} \\ b_{31} & b_{32} & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ b_{N1} & b_{N2} & b_{N3} & b_{N4} & \cdots & b_{NN} \end{pmatrix}$$

まず a_{11} から a_{1N} と b_{11} から b_{N1} までの内積をとる。ここで内積計算のループをループ 3 とする。

次にこの計算が終わったら A の行ベクトルは変えずに B の列ベクトルを b_{12} から b_{N2} に変更して再びベクトルの内積をとる。この B の列ベクトルを変更するループをループ 2 とする。ループ 2 が N までカウントしたらループ 2 をリセットし、A を a_{21} から a_{2N} に変更して内積をとる。このループをループ 1 とする。全てのループが N になったら計算終了として、PC に結果を返す。結果を C とすると C の行列データを全て PC に返すのでデータの数は N^2 である。

[1]ファイル名:u00simi/eps/gyouretu_sekkei.eps

4.3 モジュール説明

加算器、乗算器、 SRAM メモリーコントローラは前節で述べたのでここでは割愛する。

4.3.1 行列乗算制御モジュール

この制御モジュールの動作の主な物を列挙する。機能は内積制御モジュールと基本的な部分は同じである。

- PC とのデータ転送の制御
  内積計算では N 次のベクトルでも結果は 1 個だが、 N 次の行列乗算では結果も N 次であり、PC に返すデータ数は N^2 個ある。よってその全てを返す為には PC と同期を取り返す事が必要になる。
  まず PC からのデータ入力は前節で述べたのでここでは PC への出力について説明する。
  N 次の行列乗算の結果は次のセクションで述べるが評価基盤の右側 SRAM に保管してある。その番地は 10 進数では、 N^2 番地から 131072 番地に入れてある。

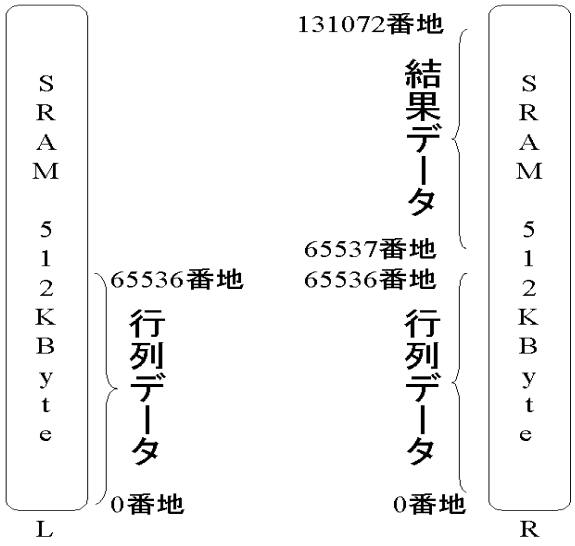


図 4.2: SRAM のアドレス割り付け (256 次行列の場合)[2]

計算終了後、 PC から制御信号を出す。この信号を受けるのが付録: 行列計算乗算モジュールの No8 である。制御信号 BL(1) の立ち下がりで結果の下位 16bit を PC に出力。その時

[2]ファイル名:u00simi/eps/adoresu_banchi.eps

OUT_CNT を‘1‘ にセットする。再び制御信号 BL(1) が出力されたら OUT_CNT が‘1‘なので 結果の上位 16bit が PC に出力される。この繰り返しをデータの数、つまり N^2 回繰り返す事によりデータを PC に返す。結果を読み出す為に必要なアドレスは PC からの制御信号 BL(3) の立ち下がりにより変化させている。つまり BL(1) を 2 回下げたの後 BL(3) を下げる。この繰り返しにより目的のアドレスからデータを読み出す事が出来る。

- メモリのアドレス制御とデータ制御
  SRAM にデータを入れる時に計算がしやすい用にデータが格納される事が大事な要因の 1 つである。

$$\boldsymbol{A} = \begin{pmatrix} a_1 & a_{N+1} & a_{2N+1} & a_{3N+1} & \cdots & a_{(N-1)N+1} \\ a_2 & a_{N+2} & a_{2N+2} & a_{3N+2} & \cdots & a_{(N-1)N+2} \\ a_3 & a_{N+3} & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ a_N & a_{N+N} & a_{2N+N} & a_{3N+N} & \cdots & a_{N^2} \end{pmatrix}$$

$$\boldsymbol{B} = \begin{pmatrix} b_1 & b_{N+1} & b_{2N+1} & b_{3N+1} & \cdots & b_{(N-1)N+1} \\ b_2 & b_{N+2} & b_{2N+2} & b_{3N+2} & \cdots & b_{(N-1)N+2} \\ b_3 & b_{N+3} & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ b_N & b_{N+N} & b_{2N+N} & b_{3N+N} & \cdots & b_{N^2} \end{pmatrix}$$

上の添字は SRAM に入れる順番である。この A、B の積をとる時 は A の行ベクトル、B の列ベクトルの内積をとれば良い。よって、B は順番通りに読み込めばいいが、A の方は a_1 から a_{N+1} 、a_{2N+1} の様に N ずつ増やさなければならない。この制御を行うのが付録: 行列計算乗算モジュールの No4、No7 である。No4 は右の SRAM のアドレスを制御、No7 は左の SRAM のアドレスを制御する。左の SRAM は内積制御モジュールと同じで SRAN メモリーコントローラの NEXT_MEM_CYCLE 信号の変化でアドレスの値を 1 ずつを変化させる。No4 の右側 SRAM の制御は SRAM メモリーコントローラの NEXT_MEM_CYCLE 信号の変化で N ずつ変化させる。さらに乗算した結果も SRAM に書込むので右の

SRAM の N 番地を初期値として 1 ずつ増やしていく。
SRAM に読み書きするデータの処理は付録: 行列計算乗算モジュールの No2 で振り別ける。もし MEM_WRITE_KEKKA_CYCLE が ‘1‘ の時はループ 3 のベクトル内積結果がでているので WRITE_DATA_REG_R(メモリーに書込むデータの一時バッファ) に QQ(計算結果のあるレジスタ) を代入する。

- 加算器、乗算器、 SRAM の制御この制御モジュールの核であり、全てのモジュールの制御を行う。
  ここもステートで記述しているので各ステートごとに説明する。
  - START
    PC から計算開始信号がでたら、まずここのステートから始まる。
  - SRAM_READ1
    READ_CYCLE2 を ‘1‘ にして MEM_STATE_SEL を連続 READ モードに設定しする。さらに LOOP3_SIGNAL を’1’ に立ち上げる。
  - WAIT1
    LOOP3_SIGNAL を’0’ に立ち下げて内積計算数をカウントする。 LOOP3 は付録: 行列計算乗算モジュール の No13 でカウントする。
  - SRAM_READ2
    LOOP3_SIGNAL を’1’ に立ち上げる。ここでも連続読み込みで次のデータを読んでいる。
  - WAIT2
    LOOP3_SIGNAL を’0’ に立ち下げて内積計算数をカウントして、 MULTI_CNT を’1’ に立ち上げて次のステートで SRAM_READ1 で読み込んだデータを乗算器に送る。
  - SRAM_READ3
    LOOP3_SIGNAL を’1’ に立ち上げる。ここでも連続読み込みで次のデータを読んでいる。 MULTI_CNT を’0’ に立ち下げて乗算器を動作させる。
  - WAIT3
    LOOP3_SIGNAL を’0’ に立ち下げて内積計算数をカウントする。さらに MULTI_CNT と ADDER_DATA_CNT を ’1’ に立ち上げて計算器にデータを送る。 SRAM_READ3

と WAIT 3 を繰り返す事により連続で計算していく。終了条件は LOOP3_COUNT が行列の次数と等しく なった時である。

- WAIT4
  WAIT3 で終了条件を満たした時にこのステートに来る。 LOOP3_REFRESH を'1' に立ち上げて LOOP3_COUNT を次のループでも正常にカウントさせる為に初期化する。初期化は付録: 行列計算乗算モジュールの No13 で行う。

- LAST_MULTI
  余っている乗算と加算の計算をさせる。 ALL_SIGNAL と LOOP2_SIGNAL を立ち上げて次で下げると ALL_COUNT と LOOP2_COUNT に 1 が加わる。この LOOP2_COUNT が 行列の次数と等しくなると LOOP2 が終り LOOP1_COUNT に 1 が加わる。 LOOP2 _COUNT は付録: 行列計算乗算モジュールの No12,14 で制御され、 LOOP1_COUNT は No11.5 と 15 で制御される。 ALL_COUNT は結果を書込むアドレスを制御する。付録: 行列計算乗算モジュールの No16 で制御される。

- WAIT5
  LAST_MULTI で使用した信号を元に戻すステートである。

- LAST_ADD
  最後に乗算器で計算したデータを加算器に送るステートである。 MEM_WRITE_KEKKA_CY とは計算結果データを SRAM に書込むデータに設定する制御信号である。

- RESULT_WRITE
  SRAM に結果を書込む準備のステート。 MEM_STATE_SEL を WRITE にして SRAM を WRITE モードにする MEM_WRITE _KEKKA を制御する。

- WAIT6
  KEKKA_ADRS_CNT を'1' に立ち上げて次に結果を書込む時のアドレスをセットする。

- WAIT7
  CLEAR_FFAG 信号で結果を SRAM に書込んだ時に使った全てのデータ、アドレス情報をリセットする。計算が終了していない時は再び START に戻る。

- STOP
  全ての計算が終了した時のみこのステートに来る。

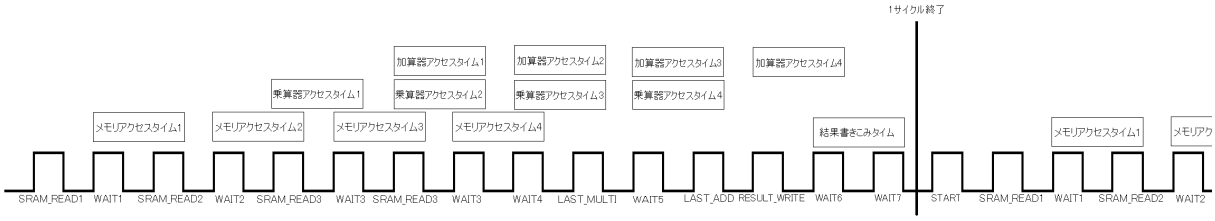


図 4.3: 制御ステートのクロック同期図[3]

4.4 結果

PC でランダムに作った単精度浮動小数点で 16 次の正方行列を作り、評価基盤上の動作クロックを 10MHz で計算させた結果を以下に記す。

$$\boldsymbol{A} = \begin{pmatrix} 3.5 & 23.25 & 8 & 20.25 & 9.75 & 23 & 23.5 & 23.5 \\ 5.5 & 22.75 & 22.75 & 5.25 & 7.75 & 14.25 & 24.75 & 5.25 \\ 24 & 13.5 & 19 & 11 & 20 & 8.5 & 24.25 & 10.5 \\ 12.75 & 12 & 18.5 & 2.5 & 24 & 8.5 & 20.75 & 24 \\ 8.75 & 11.25 & 18.25 & 23 & 6.25 & 0.75 & 9.25 & 14.25 \\ 4.75 & 23 & 17 & 24.5 & 17.25 & 23.25 & 24.5 & 11 \\ 21.25 & 13.25 & 14 & 18.25 & 23.75 & 19 & 5.75 & 20 \\ 4.5 & 6.75 & 16 & 17 & 0.75 & 1 & 1 & 14.5 \end{pmatrix}$$

[3]ファイル名:u00simi/eps/gyouretui_hakei.eps

$$\boldsymbol{B} = \begin{pmatrix} 1.5 & 22.75 & 5.5 & 23.5 & 8 & 6 & 9 & 18.75 \\ 15.75 & 3 & 16.25 & 13.75 & 6.25 & 15 & 21 & 19.5 \\ 24.5 & 4 & 16 & 19.25 & 3.75 & 14.5 & 19 & 18 \\ 21.5 & 21 & 23 & 24 & 19.25 & 12.25 & 11 & 6.75 \\ 20 & 7.25 & 9.75 & 13 & 16 & 1.5 & 4 & 11 \\ 19.25 & 0.75 & 23 & 4.25 & 19.75 & 7.75 & 23.75 & 7.25 \\ 17.75 & 17.5 & 5.25 & 5.5 & 17 & 9 & 16.25 & 12.5 \\ 14.5 & 8.25 & 18.5 & 19 & 3 & 9 & 16 & 5 \end{pmatrix}$$

$$\boldsymbol{AB(FPGA)} = \begin{pmatrix} 2398.4375 & 1981.5625 & 2096.9375 & 2075.0625 & 1642.1875 & 2699.5625 & 2208.75 & 1132.8 \\ 1299.6875 & 937.9375 & 1555.875 & 1194.0625 & 1114.125 & 1421.5625 & 1414.5 & 686.9 \\ 2173 & 1515.0625 & 1620.4375 & 1601.0625 & 1442.3125 & 2270.4375 & 2044.6875 & 1085. \\ 1842.1875 & 1403.1875 & 2008.375 & 1799 & 1669.6875 & 2009.9375 & 2190.1875 & 1209.5 \\ 1673.375 & 1214.5 & 1491 & 1271.125 & 966.3125 & 1901.8125 & 1569.625 & 557.6 \\ 1349.6875 & 1160.5 & 1165.375 & 1060 & 994.3125 & 1445.6875 & 1167.4375 & 716.8 \\ 2237.625 & 1872.875 & 1825.4375 & 1764.8125 & 1335.875 & 2313.5625 & 1895.9375 & 948. \\ 1484.9375 & 1515.875 & 1766.75 & 1527.9375 & 1128.25 & 1728.5 & 1602.875 & 719. \end{pmatrix}$$

行列乗算機能を実装させた評価基盤出の計算結果

$$\boldsymbol{AB(PC)} = \begin{pmatrix} 2398.4375 & 1981.5625 & 2096.9375 & 2075.0625 & 1642.1875 & 2699.5625 & 2208.75 & 1132.8125 \\ 1299.6875 & 937.9375 & 1555.875 & 1194.0625 & 1114.125 & 1421.5625 & 1414.5 & 686.9375 \\ 2173 & 1515.0625 & 1620.4375 & 1601.0625 & 1442.3125 & 2270.4375 & 2044.6875 & 1085.25 \\ 1842.1875 & 1403.1875 & 2008.375 & 1799 & 1669.6875 & 2009.9375 & 2190.1875 & 1209.5625 \\ 1673.375 & 1214.5 & 1491 & 1271.125 & 966.3125 & 1901.8125 & 1569.625 & 557.6875 \\ 1349.6875 & 1160.5 & 1165.375 & 1060 & 994.3125 & 1445.6875 & 1167.4375 & 726.875 \\ 2237.625 & 1872.875 & 1825.4375 & 1764.8125 & 1335.875 & 2313.5625 & 1895.9375 & 948.25 \\ 1484.9375 & 1515.875 & 1766.75 & 1527.9375 & 1128.25 & 1728.5 & 1602.875 & 719.25 \end{pmatrix}$$

PC 上での演算結果

次に 100 次の行列計算の結果について記す。 PC の結果は 300 次での計算結果を表示する。

基盤周波数	必要クロック数	秒数	MFLOPS	MFLOPS/1MHz
4MHz	1100000	0.25	8.0	2.0
8MHz	1100000	0.138	16.0	2.0
10MHz	1100000	0.1	20.0	2.0
CPU 周波数	CPU 名	秒数	MFLOPS	MFLOPS/1MHz
1GHz	PentiumIII	1.20	54	0.054

表 4.1: 行列計算の結果

4.5 結論

PC 上での行列乗算計算結果と行列乗算機能を持った評価基盤での計算結果は等しくなった。これは行列の次数が低くさらに全ての計算されるデータの仮数部が $10^{(}-2)$ の重みがある付近までしか数値が存在しない為、計算による単精度浮動小数点の桁落ちが無かった為である。計算性能は等しい結果になった。計算 PC でランダムに作った単精度浮動小数点を桁落ちしないようにしたので結果は全く同じになった。計算時間も PC に比べれば遅いがクロック当りの MFLOPS では勝っている。

第 5 章

考察と今後への提言

各演算モジュール、 SRAM メモリーコントローラを使った演算機能を持つデバイスは完成し、、高クロック (10Mhz) 中でも演算装置は動作した。この研究の最終目的である行列の対角化を用いた行列の固有値、固有ベクトルを求める専用基盤の完成までには至らなかったが昨年度の沼 [6] によって作製された行列の乗算器の問題点である高次数、高クロックで動作する専用デバイスは完成した。

今後の課題はインターフェイス部分の改善と PC と評価基盤の通信速度の短縮である。現在の ISA バスを介しての通信だとバス幅が 16bit の為、単精度浮動小数点 (32bit) では 2 回、倍精度浮動小数点 (64bit) では 4 回 もの通信が必要になってくる。さらに最近の PC には ISA バスが標準に搭載されている PC は少なくなってきた。そこで PCI バスをインターフェイスとする新しい基盤の作製に取り組むべきではないだろうか。最近では FPGA 搭載の PCI ボード開発キットなる物が販売されているのでそちらを購入し、最終的には評価基盤をそのまま PCI ボードに移行するのが好ましいだろ。また PCI ボードをバスマスタ DMA(ダイレクトメモリアクセス) 方式する事により CPU の処理を介する事なく直接 PC のメモリーにアクセスする事が出来る。よって PCI ボード上にメモリーを乗せる事が必要なくなるので PCI ボード 上 でもコンパクトな 評価基盤ボード作製する事が可能である。

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] 塚越一雄,“すぐ分かる C++Builer” , 技術評論者

付録 A

プログラムソース

A.1 加算器

```
------------------------------------------------------------
-- Floating Point Number Adder  (FLEX10k)
-- < fpadd.vhd >
-- 1998/11/17 (Tue)
-- yamaoka@tube.ee.uec.ac.jp
------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity fpadder is
  port ( CLK  : in std_logic;
         KA   : in std_logic_vector(31 downto 0);
         KB   : in std_logic_vector(31 downto 0);
         QQ    : out std_logic_vector(31 downto 0)
      );

end fpadder;
architecture RTL of fpadder is

signal INF1, INF2, INF3, ZA1, ZA2, ZA3, ZB1, ZB2, ZB3 : std_logic;
signal SA1, SA2, SA3, SB1, SB2, SB3                   : std_logic;
signal EA1, EA2, EB1, EB2                             : std_logic_vector(8 downto 0);
signal EA3, EA4, EQ, ED1, ED2                         : std_logic_vector(7 downto 0);
signal MA1, MA2, MA3, MB1, MB2, MB5                   : std_logic_vector(22 downto 0);
signal MB3                                            : std_logic_vector(23 downto 0);
signal MA4, MB4                                       : std_logic_vector(25 downto 0);
signal MQ1, MQ2                                       : std_logic_vector(25 downto 0);
signal MQ3, MQ4                                       : std_logic_vector(24 downto 0);
signal MQ5                                            : std_logic_vector(22 downto 0);
signal V0, V1, V2, V3, V4, V5                         : std_logic_vector(24 downto 0);
signal VV0, VV1, VV2, VV3, VV4                        : std_logic_vector(24 downto 0);
signal VES1, VES2                                     : std_logic_vector(8 downto 0);

begin

INF1 <= '1' when KA(30 downto 23) = "11111111" or KB(30
downto 23) = "11111111" else '0';-- どちらかの指数部が MAX な
ら INF に 1 を入力
ZA1 <= '1' when KA(30 downto 23) = "00000000" else '0';-- KA
の指数部が MIN なら ZA1 に 1 を入力
ZB1 <= '1' when KB(30 downto 23) = "00000000" else '0';-- KB
の指数部が MIN なら ZB1 に 1 を入力
```

```
EA1 <= '0' & KA(30 downto 23);-- EA1 に KA の指数部を 9bit にし
たのを入力
EB1 <= '0' & KB(30 downto 23);-- EB1 に KB の指数部を 9bit にし
たのを入力

process begin
wait until rising_edge( CLK );-- クロック同期で実行
SA1 <= KA(31);-- KA の符合 bit を入力
SB1 <= KB(31);-- KB の符合 bit を入力
EA2 <= EA1;-- KA の指数 bit 列を入力
EB2 <= EB1;-- KB の指数 bit 列を入力
MA1 <= KA(22 downto 0);-- KA の仮数 bit 列を入力
MB1 <= KB(22 downto 0);-- KB の仮数 bit 列を入力
INF2 <= INF1;-- 指数部の最大値情報を入力
ZA2 <= ZA1;-- KA の指数部の最小値情報を入力
ZB2 <= ZB1;-- KB の指数部の最小値情報を入力
end process;

VES1 <= EA2 - EB2;-- KA、 KB の指数部の差を VES1 に入力
VES2 <= EB2 - EA2;-- KB、 KA の指数部の差を VES2 に入力

SA2 <= SA1 when VES1(8) = '0' else SB1;-- 数値が大きい方の符
合を SA2 に入力
SB2 <= SB1 when VES1(8) = '0' else SA1;-- 数値が小さい方の符
合を SB2 に入力

EA3 <= EA2(7 downto 0) when VES1(8) = '0' else EB2(7 downto
0);-- 大きい方の指数部を入力
ED1 <= VES1(7 downto 0) when VES1(8) = '0' else VES2(7
downto 0);-- 指数部の差を入力

MA2 <= MA1 when VES1(8) = '0' else MB1;-- 数値が大きい方の仮数部を入力
MB2 <= MB1 when VES1(8) = '0' else MA1;-- 数値が小さい方の仮数部を入力

ZA3 <= ZA2 when VES1(8) = '0' else ZB2;-- 数値が大きい方の指
数部情報を入力
ZB3 <= ZB2 when VES1(8) = '0' else ZA2;-- 数値が小さい方の指
数部情報を入力

V0 <= "1" & MB2 & "0" when ED1(0) = '0' else "01" & MB2(22 downto 1) & "0";
V1 <= V0 when ED1(1) = '0' else "00" & V0(24 downto 2);
V2 <= V1 when ED1(2) = '0' else "0000" & V1(24 downto 4);
V3 <= V2 when ED1(3) = '0' else "00000000" & V2(24 downto 8);
V4 <= V3 when ED1(4) = '0' else "0000000000000000" & V3(24 downto 16);
V5 <= V4 + "00000000000000000000000001";
MB3 <= V5(24 downto 1) when ED1(7 downto 5) = "000" else
"000000000000000000000000";
-- 指数部の差だけ仮数部をシフトチェンジ

MA4 <= "00000000000000000000000000" when ZA3 = '1' else
       "001" & MA2 when SA2 = '0' else
       "00000000000000000000000000" - ("001" & MA2);

MB4 <= "00000000000000000000000000" when ZB3 = '1' else
       "00" & MB3 when SB2 = '0' else
       "00000000000000000000000000" - ("00" & MB3);
-- 指数部が計算範囲を下回っていたならば 0 を入れて、符合によっ
てビット列を反転させる
MQ1 <= MA4 + MB4;-- 仮数部の加算

process begin -- クロック同期
wait until rising_edge( CLK );
EA4 <= EA3
MQ2 <= MQ1;
INF3 <= INF2;
end process;

MQ3 <= MQ2(24 downto 0) when MQ2(25) = '0' else
       "00000000000000000000000000" - MQ2(24 downto 0);
-- 計算結果が負の結果なら反転させる
```

```
MQ4 <= MQ3 + "00000000000000000000000001";

ED2 <=  "00000000" when MQ3(24) = '1' else
        "00000001" when MQ3(23) = '1' else
        "00000010" when MQ3(22) = '1' else
        "00000011" when MQ3(21) = '1' else
        "00000100" when MQ3(20) = '1' else
        "00000101" when MQ3(19) = '1' else
        "00000110" when MQ3(18) = '1' else
        "00000111" when MQ3(17) = '1' else
        "00001000" when MQ3(16) = '1' else
        "00001001" when MQ3(15) = '1' else
        "00001010" when MQ3(14) = '1' else
        "00001011" when MQ3(13) = '1' else
        "00001100" when MQ3(12) = '1' else
        "00001101" when MQ3(11) = '1' else
        "00001110" when MQ3(10) = '1' else
        "00001111" when MQ3( 9) = '1' else
        "00010000" when MQ3( 8) = '1' else
        "00010001" when MQ3( 7) = '1' else
        "00010010" when MQ3( 6) = '1' else
        "00010011" when MQ3( 5) = '1' else
        "00010100" when MQ3( 4) = '1' else
        "00010101" when MQ3( 3) = '1' else
        "00010110" when MQ3( 2) = '1' else
        "00010111" when MQ3( 1) = '1' else
        "00011000" when MQ3( 0) = '1' else
        "10000000";

VV0 <= MQ3 when ED2(0) = '0' else MQ3(23 downto 0) & "0";
VV1 <= VV0 when ED2(1) = '0' else VV0(22 downto 0) & "00";
VV2 <= VV1 when ED2(2) = '0' else VV1(20 downto 0) & "0000";
VV3 <= VV2 when ED2(3) = '0' else VV2(16 downto 0) & "00000000";
VV4 <= VV3 when ED2(4) = '0' else VV3( 8 downto 0) & "0000000000000000";
MQ5 <= VV4(23 downto 1) when MQ4(24) = '0' else MQ4(23 downto 1);
-- 計算結果によって指数部の調整をして最適にする
EQ <= "11111111" when INF3 = '1' else
      EA4 - ED2 + "00000001" when ED2(7) = '0' else
      "00000000";
-- 最終結果がオーバーフロー、もしくは桁落ちならばその結果を
指数部に入れる
QQ <= MQ2(25) & EQ & MQ5;-- 最終結果
end RTL;
```

A.2 乗算器

```
------------------------------------------------------------
-- Floating Point Number Multiplier (FLEX10k)
-- < fpmult.vhd >
-- 1999/01/19 (Tue)
-- yamaoka@tube.ee.uec.ac.jp
-- (addition)
-- 2000/8/25
-- nobutaka@tube.ee.uec.ac.jp
------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity fpmult is
  port (   CLK   : in std_logic;  -- クロック
    FA   : in std_logic_vector(31 downto 0);-- 乗算要素
           FB   : in std_logic_vector(31 downto 0);-- 乗算要素
           Q    : out std_logic_vector(31 downto 0)-- 計算
結果
     );

end fpmult;
architecture RTL of fpmult is
```

```
signal MA1, MA2, MB1, MB2 : std_logic_vector(23 downto 0);
signal EA1, EA2, EB1, EB2 : std_logic_vector(9 downto 0);
signal MQ1, MQ2, MQ3      : std_logic_vector(25 downto 0);
signal EQ1, EQ2, EQ3, EQ4 : std_logic_vector(9 downto 0);
signal EQ5, EQ6, EQ       : std_logic_vector(7 downto 0);
signal MQ                 : std_logic_vector(22 downto 0);
signal S1, S2, SQ ,ZERO_FLAG_A1,ZERO_FLAG_A2,
ZERO_FLAG_A3,ZERO_FLAG_B1,ZERO_FLAG_B2,ZERO_FLAG_B3: std_logic;

begin

MA1 <= '1' & FA(22 downto 0);-- 仮数部 A の入力
MB1 <= '1' & FB(22 downto 0);-- 仮数部 B の入力
EA1 <= "00" & FA(30 downto 23);-- 指数部 A の入力
EB1 <= "00" & FB(30 downto 23);-- 指数部 B の入力
S1  <= FA(31) xor FB(31);符合を排他的論理和で取る
ZERO_FLAG_A1 <= '1' when FA(22 downto 0)
= "00000000000000000000000" and FA(30 downto 23) = "00000000" else
             '0';-- 計算要素がゼロならば結果もゼロになるの
でそのチェック
ZERO_FLAG_B1 <= '1' when FB(22 downto 0)
= "00000000000000000000000" and FB(30 downto 23) = "00000000" else
             '0';-- 計算要素がゼロならば結果もゼロになるの
でそのチェック
process begin-- クロック同期
wait until rising_edge( CLK );
MA2 <= MA1;
MB2 <= MB1;
EA2 <= EA1;
EB2 <= EB1;
S2  <= S1;
    ZERO_FLAG_A2 <= ZERO_FLAG_A1;
    ZERO_FLAG_B2 <= ZERO_FLAG_B1;

end process;

process( MA2, MB2 )
variable TMQ1 : std_logic_vector(47 downto 0);--数値
定義
begin
TMQ1 := MA2 * MB2;仮数部の乗算結果
MQ1 <= TMQ1(47 downto 22);-- 結果の畳み込み
end process;

EQ1 <= "0011111111" when EA2(7 downto 0) = "11111111" or EB2(7 downto 0) = "11111111" else
       "0000000000" when EA2(7 downto 0) = "00000000" or EB2(7 downto 0) = "00000000" else
       EA2 + EB2 - "0001111111";

EQ2 <= "0011111111" when EA2(7 downto 0) = "11111111" or EB2(7 downto 0) = "11111111" else
       "0000000000" when EA2(7 downto 0) = "00000000" or EB2(7 downto 0) = "00000000" else
       EA2 + EB2 - "0001111110";
-- 指数部が計算可能範囲かをチェックする。それ以外はオフセッ
ト分引く
process begin-- クロック同期
wait until rising_edge( CLK );
SQ <= S2;
MQ2 <= MQ1;
EQ3 <= EQ1;
EQ4 <= EQ2;
    ZERO_FLAG_A3 <= ZERO_FLAG_A2;
    ZERO_FLAG_B3 <= ZERO_FLAG_B2;


end process;

MQ3 <= MQ2 + "00000000000000000000000001" when MQ2(25) = '0' else
       MQ2 + "00000000000000000000000010";
-- 丸め込みをする
EQ5 <= "00000000" when EQ3(9 downto 8) = "11" else
       "11111111" when EQ3(9 downto 8) = "01" else
```

```
       EQ3(7 downto 0);

EQ6 <= "00000000" when EQ4(9 downto 8) = "11" else
       "11111111" when EQ4(9 downto 8) = "01" else
       EQ4(7 downto 0);
-- 計算可能範囲なら数値はそのまま
MQ <= "00000000000000000000000" when ZERO_FLAG_A3 = '1' or ZERO_FLAG_B3 = '1' else
      MQ3(23 downto 1) when MQ3(25) = '0' else
      MQ3(24 downto 2);
-- 計算する数値が 0 なら結果も 0 にする
EQ <= EQ5 when MQ3(25) = '0' else
      EQ6;
-- 指数部の最上位ビットがなら指数部をシフトチェンジする
Q <= SQ & EQ & MQ;-- 結果の入力
end RTL;
```

A.3 SRAM メモリーコントローラ

```
------------------------------------------------------------
-- SRAM mem_ctrl(FLEX10k)
-- < mem_ctrl.vhd >
-- 1999/01/19 (Tue)
-- yamaoka@tube.ee.uec.ac.jp
-- (addition)
-- 2000/8/25
-- nobutaka@tube.ee.uec.ac.jp
------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

library metamor;
use metamor.attributes.all;
 entity mem_ctrl is
 port ( CLK  : in std_logic;-- クロック
        ADRS : out std_logic_vector(16 downto 0);--SRAM に送るアドレス設定バス
        ADRS_MUX : in std_logic_vector (16 downto 0);-- 制御モジュールから送られるアドレス信号
        DATA : in std_logic_vector(31 downto 0);--SRAM からのデータバス
        DATA_BUF : out std_logic_vector(31 downto 0);--SRAM に送るデータの一時バッファ
        WRITE_DATA_REG : in std_logic_vector(31 downto 0);--PC から読み込んだデータを一時保管する
レジスタ
        READ_DATA_REG : out std_logic_vector(31 downto 0);--SRAM から読み込んだデータを一時保管す
るレジスタ
        SCS : out std_logic_vector(3 downto 0);--SRAM の内部信号
        SOE : out std_logic;--SRAM の内部信号
        SWE : out std_logic;--SRAM の内部信号
        OE  : out std_logic;--SRAM の内部信号
        MEM_STATE_SEL : in std_logic_vector(1 downto 0);-- メモリステートを管理する
        NEXT_MEM_CYCLE : out std_logic;-- アドレスを上げる信号
        V_D_FLAG : in std_logic-- 内積計算中に SRAM に連続アクセ
スする為の信号

       );

  end mem_ctrl;

architecture RTL of mem_ctrl is-- 以下は信号の定義
   type STATE_TYPE is
   (STOP, WRITE1, WRITE2,  READ1, READ2, TOCHUU);
   signal CURRENT_STATE : STATE_TYPE;
   signal NEXT_STATE : STATE_TYPE;
   signal BH_BUF : std_logic_vector(7 downto 0);
   signal CNT  : std_logic:='0';
```

```
  constant STOP2 : std_logic_vector(1 downto 0):= "00";
  constant WRITE : std_logic_vector(1 downto 0):= "11";
  constant READ : std_logic_vector(1 downto 0):= "10";

begin

  process ( CLK,MEM_STATE_SEL,CNT,CURRENT_STATE,V_D_FLAG )begin
--NO1
--SRAMにアクセスするプロセス
  if falling_edge( CLK ) then--クロックの立下りで
   case CURRENT_STATE is--カレントステートが

        when STOP =>--STOPの時
             SOE <= '1';
             SWE <= '1';
             SCS <= "1111";
             OE <= '1';
             CNT <= '0';
             NEXT_MEM_CYCLE <= '0';


        when WRITE1 =>
             SCS <= "0000";
             ADRS <= ADRS_MUX;--アドレスをSRAMに入力
             DATA_BUF <= WRITE_DATA_REG;---SRAMにデータを書き込む
             OE <= '0';
             NEXT_STATE <= WRITE2;
             CNT <= '1';

        when WRITE2 =>
             SWE <= '0';
             NEXT_STATE <= STOP;
             NEXT_MEM_CYCLE <= '1';--制御モジュールのアドレスを変更

        when READ1 =>
             SCS <= "0000";
             SOE <= '0';
             ADRS <= ADRS_MUX;--アドレスをSRAMに設定
             OE <= '1';
             NEXT_STATE <= READ2;
             CNT <= '1';
             NEXT_MEM_CYCLE <= '1'
        when READ2 =>
             READ_DATA_REG <= DATA--SRAMから読み込んだデータをレジスタに保管
             NEXT_MEM_CYCLE <= '0';--制御モジュールのアドレ
スを変更
             if V_D_FLAG = '0' then-- もし連続読み込
みが終了なら

                NEXT_STATE <= STOP;-- ステートの終了
             else-- そうでなければ
                NEXT_STATE <= READ1;-- 続行
             end if;
  end case;
 end if;
--No2
--制御モジュールからのステート信号を振り分ける
 if rising_edge( CLK ) then
   case MEM_STATE_SEL is
          when STOP2 =>
               CURRENT_STATE <= STOP;
          when WRITE =>
               CURRENT_STATE <= WRITE1;--書き込みサイクルに設定
          when READ =>
               CURRENT_STATE <= READ1;--読み込みサイクルに設定
          when others =>
               CURRENT_STATE <= STOP;--ステート終了
       end case;
      end if;
      if CNT = '1' then--ステート動作中なら
         CURRENT_STATE <= NEXT_STATE;--そのステートを優先
      end if;
```

```
    end process;
 end RTL;
```

A.4 内積計算制御モジュール

```
-----------------------------------------------------------------
-- naiseki seigyo module  (FLEX10k)
-- < test4.vhd >
-- 2000/8/17 (Tue)
-- nobutaka@tube.ee.uec.ac.jp
-----------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

library metamor;
use metamor.attributes.all;
 entity test is

  port (
         CLK : in std_logic;-- クロック
         A   : inout std_logic_vector (15 downto 0);--A レジスタ（2 方向）
         BL  : in std_logic_vector ( 7 downto 0);--B レジスタ（入力）
         BH  : out std_logic_vector ( 7 downto 0 );--B レジスタ（出力）
         CL  : in std_logic_vector ( 5 downto 0);--8255 インタフェイスの信号
         OBF : in std_logic_vector ( 1 downto 0 );--8255 インタフェイスの信号
         IBF : in std_logic_vector ( 1 downto 0 );--8255 インタフェイスの信号
         ACK : out std_logic_vector ( 1 downto 0 );--8255 インタフェイスの信号
         STB : out std_logic_vector ( 1 downto 0 );--8255 インタフェイスの信号
   --Right Memory
         DATA_R : inout std_logic_vector ( 31 downto 0 );-- アドレスを変更させるレジスタ
         ADRS_R : out std_logic_vector ( 16 downto 0 );--SRAM にに出すアドレスレジスタ
         SCS_R : out std_logic_vector ( 3 downto 0);--SRAM の内部信号
         SWE_R : out std_logic;--SRAM の内部信号
         SOE_R : out std_logic;--SRAM の内部信号
   --Left Memory
         DATA_L : inout std_logic_vector ( 31 downto 0 );-- アドレスを変更させるレジスタ
         ADRS_L : out std_logic_vector ( 16 downto 0 );--SRAM にに出すアドレスレジスタ
         SCS_L : out std_logic_vector ( 3 downto 0);--SRAM の内部信号
         SWE_L : out std_logic;--SRAM の内部信号
         SOE_L : out std_logic--SRAM の内部信号

 );

  attribute pinnum of A   : signal is
 "BC23,BB24,BC25,BB26,BC27,BB28,BC29,BB30,BC31,BB32,BC33,BB34,BC35,BB36,BC37,BB38";
  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 CLK : signal is  "D22";
  attribute pinnum of ADRS_R : signal is
"T38,W37,Y38,AA37,AB38,AC37,AD38,AE37,
AF38,AJ37,AK38,AL37,AM38,AN37,AP38,AR37,AT38";
  attribute pinnum of DATA_R : signal is
"F42,G43,H42,J43,K42,L43,M42,N43,T42,U43,V42,W43,Y42,AA43,AB42,AC43,AF42,AG43,AH42,AJ43,AK42,
AL43,AM42,AN43,AT42,AU43,AV42,AW43,AY42,BA43,BB42,BC43";
  attribute pinnum of SCS_R : signal is "AG39,AH40,AJ39,AM40";
  attribute pinnum of SWE_R : signal is "AN39";
  attribute pinnum of SOE_R : signal is "AP40";
  attribute pinnum of ADRS_L : signal is
"T6,W7,Y6,AA7,AB6,AC7,AD6,AE7,AF6,AJ7,AK6,AL7,AM6,AN7,AP6,AR7,AT6";
  attribute pinnum of DATA_L : signal is
"F2,G1,H2,J1,K2,L1,M2,N1,
T2,U1,V2,W1,Y2,AA1,AB2,AC1,AF2,AG1,AH2,AJ1,AK2,AL1,AM2,AN1,AT2,AU1,AV2,AW1,AY2,BA1,BB2,BC1";
  attribute pinnum of SCS_L : signal is "AG5,AH4,AJ5,AM4";
  attribute pinnum of SWE_L : signal is "AN5";
  attribute pinnum of SOE_L : signal is "AP4";
```

```
  attribute pinnum of CL  : signal is "AU23,AV24,AU25,AU33,AV34,AU35";
  attribute pinnum of OBF : signal is  "AV18,AV28";
  attribute pinnum of IBF : signal is  "AV20,AV30";
  attribute pinnum of ACK : signal is  "AU19,AU29";
  attribute pinnum of STB : signal is  "AU21,AU31";
end test;

architecture RTL of test is

component mem_ctrl is
port (  CLK  : in std_logic;-- クロック
        ADRS : out std_logic_vector(16 downto 0);--SRAM に出すアドレスレジスタ
        ADRS_MUX : in std_logic_vector (16 downto 0);-- アドレスを変更させるレジスタ
        DATA : in std_logic_vector(31 downto 0);-- データを SRAM に出すレジスタ
        DATA_BUF : out std_logic_vector(31 downto 0);-- データを SRAM に出すレジスタバッファ
        WRITE_DATA_REG : in std_logic_vector(31 downto 0);--PC から読み込んだデータを一時保管する
レジスタ
        READ_DATA_REG : out std_logic_vector(31 downto 0);--SRAM から読み込んだデータを一時保管す
るレジスタ
        SCS : out std_logic_vector( 3 downto 0);--SRAM の内部信号
        SOE : out std_logic;--SRAM の内部信号
        SWE : out std_logic;--SRAM の内部信号
        OE  : out std_logic;--SRAM の内部信号
        MEM_STATE_SEL : in std_logic_vector( 1 downto 0);-- メモリステートを管理する
        NEXT_MEM_CYCLE : out std_logic;-- アドレスを上げる信号
        V_D_FLAG : in std_logic-- 内積計算中に SRAM に連続アクセ
スする為の信号

         );
    end component;
component fpadder is
port(
         CLK  : in std_logic;-- クロック
         KA   : in std_logic_vector(31 downto 0);-- 加算器の 1 つめのデータレジスタ
         KB   : in std_logic_vector(31 downto 0);-- 加算器の 2 つめのデータレジスタ
         QQ    : out std_logic_vector(31 downto 0)-- 計算結果
      );
end component;
component fpmult is
port(
            CLK  : in std_logic;-- クロック
        FA   : in std_logic_vector(31 downto 0);-- 乗算器の 1 つ目のレジスタ
            FB   : in std_logic_vector(31 downto 0);-- 乗算器の 2 つ目のレジスタ
            Q    : out std_logic_vector(31 downto 0)-- 計算結果
      );

end component;


signal BH_BUF1 : std_logic;--BH を制御する信号
signal BH_BUF2 : std_logic;--BH を制御する信号
signal A_REG2 : std_logic_vector ( 15 downto 0);--PC に出力するレジスタ
signal DATA_BUF_R  : std_logic_vector ( 31 downto 0 );-- データのバッファ
signal DATA_BUF_L  : std_logic_vector ( 31 downto 0 );-- データのバッファ
signal ACK_BUF : std_logic_vector ( 1 downto 0 );--8255 インタフェイスのバッファ
signal STB_BUF : std_logic_vector (1 downto 0 );--8255 インタフェイスのバッファ
signal WRITE_DATA_ACTIVE :std_logic;-- メモリに書きこむデータの有無を知らせる信号
signal WRITE_DATA_ACTIVE_BUF :std_logic;-- メモリに書きこむデータの有無を知らせる信号
signal ADRS_MUX_R : std_logic_vector(16 downto 0);-- アドレスのレジスタ
signal ADRS_MUX_L : std_logic_vector(16 downto 0);-- アドレスのレジスタ
signal OE_R  : std_logic;--SRAM の内部信号
signal OE_L  : std_logic;--SRAM の内部信号
signal WRITE_DATA_REG_R : std_logic_vector(31 downto 0);--PC から読み込んだデータを一時保管するレ
ジスタ
```

```
signal WRITE_DATA_REG_L : std_logic_vector(31 downto 0);--PC から読み込んだデータを一時保管するレジスタ
signal WRITE_CNT : std_logic;
signal READ_DATA_REG_R :std_logic_vector(31 downto 0);--SRAM から読み込んだデータを一時保管するレジスタ
signal READ_DATA_REG_L :std_logic_vector(31 downto 0);--SRAM から読み込んだデータを一時保管するレジスタ
signal MEM_STATE_SEL_R : std_logic_vector(1 downto 0);-- ステートの変更をする信号
signal MEM_STATE_SEL_L : std_logic_vector(1 downto 0);-- ステートの変更をする信号
signal NEXT_MEM_CYCLE_R : std_logic;-- アドレスを上げる信号
signal NEXT_MEM_CYCLE_L : std_logic;-- アドレスを上げる信号
signal V_D : std_logic_vector(30 downto 0);-- メモリから計算の為に呼び出した個数をカウントするレジスタ
signal V_D_BUF1 : std_logic_vector(30 downto 0);--PC からのデータ個数をカウントするレジスタ
signal V_D_BUF2 : std_logic_vector(30 downto 0);--PC からのデータ個数をカウントするレジスタ
signal V_D_BUF1_FLAG : std_logic;
signal V_D_FLAG : std_logic;-- 連続読み込みを制御する信号
signal SIGNAL1 : std_logic;--SIGNAL2 を制御するプログラム
signal SIGNAL2 : std_logic;--READ_CYCLE を制御するプログラム
signal READ_CYCLE2 : std_logic;-- ステートをストップさせる信号 READ_CYCLE2 を制御する信号
signal IN_CNT : std_logic;--PC からのデータをレジスタに振り分ける信号
signal OUT_CNT : std_logic;--PC に出力させるデータを 32 ビットにするための信号
signal START_CNT : std_logic;
signal FA : std_logic_vector(31 downto 0);-- 乗算器の 1 つ目のレジスタ
signal FB : std_logic_vector(31 downto 0);-- 乗算器の 1 つ目のレジスタ
signal Q : std_logic_vector(31 downto 0);-- 計算結果
signal KA : std_logic_vector(31 downto 0);-- 加算器の 2 つめのデータレジスタ
signal KB : std_logic_vector(31 downto 0);-- 加算器の 2 つめのデータレジスタ
signal QQ : std_logic_vector(31 downto 0);-- 計算結果
signal DATA_CNT : std_logic;-- 乗算器に出力するデータの順番
signal MULTI_CNT : std_logic;
signal ADDER_DATA_CNT : std_logic;
signal R_L_CHG : std_logic;
signal MULTI_DATA_SET : std_logic;
signal V_D_SIGNAL : std_logic;--V_D を増やすための信号
signal STOP_SIGNAL : std_logic;-- ステートをストップさせる信号


constant STOP2 : std_logic_vector(1 downto 0):= "00";
constant WRITE : std_logic_vector(1 downto 0):= "11";
constant READ : std_logic_vector(1 downto 0):= "10";

type STATE_TYPE2 is
     (STOP,SRAM_READ,WAIT1,WAIT2,OUTPUT_DATA,WAIT3,WAIT4,OUTPUT_ADDER_DATA,START);
signal CURRENT_STATE2 : STATE_TYPE2;
signal NEXT_STATE2 : STATE_TYPE2;

begin
mem_r : mem_ctrl port map (CLK=>CLK, ADRS=>ADRS_R,ADRS_MUX=>ADRS_MUX_R,
DATA=>DATA_R, DATA_BUF=>DATA_BUF_R,WRITE_DATA_REG=>WRITE_DATA_REG_R,
READ_DATA_REG=>READ_DATA_REG_R, SCS=>SCS_R, SOE=>SOE_R,
SWE=>SWE_R, OE=>OE_R,
MEM_STATE_SEL=>MEM_STATE_SEL_R, NEXT_MEM_CYCLE=>NEXT_MEM_CYCLE_R);
mem_l : mem_ctrl port map (CLK=>CLK, ADRS=>ADRS_L,ADRS_MUX=>ADRS_MUX_L,
DATA=>DATA_L,
DATA_BUF=>DATA_BUF_L,WRITE_DATA_REG=>WRITE_DATA_REG_L,
READ_DATA_REG=>READ_DATA_REG_L, SCS=>SCS_L, SOE=>SOE_L,
SWE=>SWE_L, OE=>OE_L,
MEM_STATE_SEL=>MEM_STATE_SEL_L, NEXT_MEM_CYCLE=>NEXT_MEM_CYCLE_L);
add : fpadder  port map (CLK,KA,KB,QQ);
mul : fpmult   port map (CLK,FA,FB,Q);
--No0.5


--No1
-- メモリに書きこむ際のハイインピーダンス状態の変更を制御するプロセス
process ( OE_R )begin
```

```
 if OE_R = '1' then-- OE は No の SRAM の信号
    DATA_R <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ"; -- ハイインピーダンスに設定
 else
    DATA_R <= DATA_BUF_R;-- メモリから読み込む
 end if;
end process;
process ( OE_L )begin
 if OE_L = '1' then-- OE は No の SRAM の信号
    DATA_L <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ"; -- ハイインピーダンスに設定
 else
    DATA_L <= DATA_BUF_L;-- メモリから読み込む
 end if;
end process;
--A レジスタのハイインピーダンス設定
A <= "ZZZZZZZZZZZZZZZZ" when ACK_BUF = "00" else A_REG2;
ACK_BUF <= OBF;--used in No4
ACK <= ACK_BUF;--used in No4
STB <= STB_BUF;--used in No4
--No2
--PC からデータを FPGA に送りそれを 32 ビットにまとめるプロセス
process ( BL(0),BL(6),OBF,IN_CNT) begin
if  OBF'event and OBF = "11" then--OBF は PC からデータを FPGA に出すと真になる

       if IN_CNT = '0' then
         IN_CNT <= '1';
         WRITE_DATA_ACTIVE_BUF <= '0';-- used in No7
         START_CNT <= '1';
         if R_L_CHG = '0' then
            WRITE_DATA_REG_R(15 downto 0) <= A;-- 下位 16 ビットをレジスタに代入
         else
            V_D_BUF1_FLAG <= '0';
            WRITE_DATA_REG_L(15 downto 0) <= A;-- 下位 16 ビット
         end if;
       elsif IN_CNT = '1' then
         IN_CNT <= '0';
         WRITE_DATA_ACTIVE_BUF <= '1';--used in No7
         START_CNT <= '0';
         if R_L_CHG = '0' then
            WRITE_DATA_REG_R(31 downto 16) <= A;-- 上位 16 ビットをレジスタに代入
         else
            V_D_BUF1_FLAG <= '1';--PC からのデータ個数をカウントする used in 3.2
            WRITE_DATA_REG_L(31 downto 16) <= A;-- 上位 16 ビット
         end if;
       end if;
   end if;
if BL(0) = '1' then-- 初期化信号,
   IN_CNT <= '0';
   R_L_CHG <= '0';
   WRITE_DATA_ACTIVE_BUF <= '0';--used in No7
elsif BL(6) = '1' then
     WRITE_DATA_ACTIVE_BUF <= '0';
     R_L_CHG <= '1';
end if;
end process;
--No3
--WRITE_DATA_ACTIVE をクロックに同期させるプロセス
process (CLK) begin
  if rising_edge(CLK) then
     WRITE_DATA_ACTIVE <= WRITE_DATA_ACTIVE_BUF;
  end if;
end process;
--No4
-- アドレスを NEXT_MEM_CYCLE に応じて変化させるプロセス
process(BL(0),NEXT_MEM_CYCLE_L) begin
 if falling_edge(NEXT_MEM_CYCLE_L)  then
     ADRS_MUX_L <= ADRS_MUX_L + '1';-- アドレスを 1 ずつ増やす --used in MEM_CTRL
 end if;
 if BL(0) = '1' then-- 初期化信号
      ADRS_MUX_L <= "0000000000000000";--used in MEM_CTRL
 end if;
end process;
process(BL(0),NEXT_MEM_CYCLE_R) begin
```

```
 if falling_edge(NEXT_MEM_CYCLE_R)  then
      ADRS_MUX_R <= ADRS_MUX_R + '1';-- アドレスを 1 ずつ増やす --used in MEM_CTRL
  end if;
 if BL(0) = '1' then-- 初期化信号
       ADRS_MUX_R <= "0000000000000000";--used in MEM_CTRL
 end if;
end process;

--No5
--PC に計算結果を返すプロセス
process ( IBF, BL(0),BL(1),OUT_CNT) begin

          if BL(0) = '1' then -- 初期化信号
            OUT_CNT <= '0';
            A_REG2 <= "0000000000000000";--used in the top
            STB_BUF <= "11";--8255 インタフェイスの制御信号

          elsif  falling_edge( BL(1) ) then--PC からの信号
            if OUT_CNT = '0' then
              OUT_CNT <= '1';
              STB_BUF <= "00";--8255 インタフェイスの制御信号
              A_REG2 <= QQ(15 downto 0);-- 内積の結果の下位 16 ビットの出力
            else
              OUT_CNT <= '0';
              STB_BUF <= "00";-- インタフェイスの制御信号
              A_REG2 <= QQ(31 downto 16);-- 内積の結果の上位 16 ビットの出力
            end if;
          end if;
       if IBF = "11" then
            STB_BUF <= "11";-- インタフェイスの制御信号
    end if;
 end process;
--No6
-- 掛け算をするためにメモリからのデータを振り分けるプロセス
process begin
wait until CLK'event and CLK = '0';
  if MULTI_CNT = '1' then
        FA <= READ_DATA_REG_R;-- 掛け算の 1 つ目の要素 --used in FPMULTI
        FB <= READ_DATA_REG_L ;-- 掛け算の 2 つ目の要素 --used in FPMULTI
  end if;
  if BL(0) = '1' then-- 初期化信号
     DATA_CNT <= '0';
     FA <= "00000000000000000000000000000000";--FA の初期化 used in FPMULTI
     FB <= "00000000000000000000000000000000";--FB の初期化 used in FPMULTI
   end if;
  end process;

--No7
-- 足し算をするために乗算器からデータを振り分けるプロセス
process begin
  wait until CLK'event and CLK = '0';
  if ADDER_DATA_CNT = '1' then

      KA <= Q;-- 乗算器の結果を代入 used in ADDER
      KB <= QQ;-- 加算器の結果を再び代入 used in ADDER

  end if;
   if BL(0) = '1' then-- 初期化信号
      KB <= "00000000000000000000000000000000";--KB の初期化 used in ADDER
      KA <= "00000000000000000000000000000000";--KA の初期化 used in ADDER
   end if;
 end process;
--No8
-- メモリコントローラの制御をするプロセス
 process begin
  wait until CLK'event and CLK = '0';
    if WRITE_DATA_ACTIVE = '1' then-- もし PC からデータが入力されたなら
        if WRITE_CNT = '1' then-- もし上位 16bit 目が入ってきたら
           if R_L_CHG = '0' then
              WRITE_CNT <= '0';
              MEM_STATE_SEL_R <= WRITE;-- メモリコントローラをライトにする used in MEM_CTRL
           else
```

```
            WRITE_CNT <= '0';
            MEM_STATE_SEL_L <= WRITE;
         end if;
      elsif WRITE_CNT = '0' then-- もし下位 16bit 目が入ってきたら
         if R_L_CHG = '0' then
            MEM_STATE_SEL_R <= STOP2;-- 一度 WRITE にしたらもう書きこまない
         else
            MEM_STATE_SEL_L <= STOP2;
         end if;
      end if;
    elsif START_CNT = '1' then
          WRITE_CNT <= '1';

    elsif READ_CYCLE2 = '1' then-- もし計算命令が出たなら
        MEM_STATE_SEL_R <= READ;-- メモリコントローラをリードにする  used in MEM_CTRL
        MEM_STATE_SEL_L <= READ;-- メモリコントローラをリードにする  used in MEM_CTRL


     else
        MEM_STATE_SEL_R <= STOP2;-- メモリコントローラをストップにする  used in MEM_CTRL
        MEM_STATE_SEL_L <= STOP2;-- メモリコントローラをストップにする  used in MEM_CTRL

     end if;
    if BL(0) = '1' then
        MEM_STATE_SEL_R <= STOP2;-- メモリコントローラをストップにする  used in MEM_CTRL
        MEM_STATE_SEL_L <= STOP2;-- メモリコントローラをストップにする  used in MEM_CTRL

    elsif BL(6) = '1' then
        WRITE_CNT <= '0';

    end if;
end process;
--No8.5
--V_D(内積の要素の数) をカウントするプロセス。
 process(V_D_SIGNAL,BL(6)) begin
  if falling_edge(V_D_SIGNAL) then
     V_D <= V_D - '1';
  end if;
  if BL(6) = '1' then
     V_D <= V_D_BUF1;
  end if;
end process;
--No9
--READ_CYCLE2 を制御するプロセス

process (CLK,V_D,V_D_FLAG,BL(7),BL(0),CURRENT_STATE2)begin
  if rising_edge(CLK) then
       case CURRENT_STATE2 is-- 制御モジュール開始
            when START =>
                 NEXT_STATE2 <= SRAM_READ;
            when SRAM_READ1 =>
                 READ_CYCLE2 <= '1';
                 V_D_SIGANL <= '1';
                 NEXT_STATE2 <= WAIT1;
            when WAIT1      =>
                 V_D_SIGNAL <= '0';
                 NEXT_STATE2 <= SRAM_READ2;
            when SRAM_READ2 =>
                 V_D_SIGNAL <= '1';
                 NEXT_STATE2 <= WAIT2;
            when WAIT2      =>
                 V_D_SIGNAL <= '0';
                 MULTI_CNT <= '1';
                 NEXT_STATE2 <= SRAM_READ3;
            when SRAM_READ3 =>
                 V_D_SIGNAL <= '1';
                 MULTI_CNT <= '0';
                 ADDER_DATA_CNT <= '0';
                 NEXT_STATE2 <= WAIT3;
            when WAIT3      =>
                 V_D_SIGNAL <= '0';
                 MULTI_CNT <= '1';
                 ADDER_DATA_CNT <= '1';
                 NEXT_STATE2 <= SRAM_READ3;
```

```
            when WAIT4 =>
                 MULTI_CNT <= '0';
                 ADDER_DATA_CNT <= '0';
                 READ_CYCLE2 <= '0';
                 NEXT_STATE2 <= LAST_MULTI;
            when LAST_MULTI =>
                 MULTI_CNT <= '1';
                 ADDER_DATA_CNT <= '1';
                 NEXT_STATE2 <= WAIT5;
            when WAIT5 =>
                 MULTI_CNT <= '0';
                 ADDER_DATA_CNT <= '0';
                 NEXT_STATE2 <= LAST_ADD;
            when LAST_ADD =>
                 ADDER_DATA_CNT <= '1';
                 NEXT_STATE2 <= STOP;
            when STOP =>
                 ADDER_DATA_CNT <= '0';
                 NEXT_STATE2 <= STOP;
                 if STOP_SIGNAL = '1' then
                    BH_BUF2 <= '1';
                 else
                    BH_BUF2 <= '0';
                 end if;
            when others =>
                 STOP_SIGNAL <= '0';
                 NEXT_STATE2 <= STOP;
      end case;
    end if;
  if falling_edge(CLK) then
     if V_D = ''00000000000000000000000000000000'' then
           READ_CYCLE2 <= '0';
        if V_D_FLAG = '1' then
           CURRENT_STATE2 <= WAIT4;
           V_D_FLAG <= '0'
        else
           CURRENT_STATE2 <= NEXT_STATE2;
        end;
     elsif BL(7) = '1' then
        CURRENT_STATE2 <= START;
     elsif BL(0) = '1' then
        CURRENT_STATE2 <= STOP;
        V_D_FLAG <= '1'
     else
        CURRENT_STATE2 <= NEXT_STATE2;
     end if;
   end if;
end process;


end RTL;
```

A.5 行列計算制御モジュール

```
-------------------------------------------------------------
-- gyouretu seigyo module  (FLEX10k)
-- < main.vhd >
-- 2000/11/17 (Tue)
-- nobutaka@tube.ee.uec.ac.jp
-------------------------------------------------------------

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

library metamor;
use metamor.attributes.all;
 entity main is

  port (
         CLK : in std_logic;--クロック
```

```
        A   : inout std_logic_vector (15 downto 0);--A レジスタ（2 方向）
        BL  : in std_logic_vector ( 7 downto 0);--B レジスタ（入力）
        BH  : out std_logic_vector ( 7 downto 0 );--B レジスタ（出力）
        CL  : in std_logic_vector ( 5 downto 0);--8255 インタフェイスの信号
        OBF : in std_logic_vector ( 1 downto 0 );--8255 インタフェイスの信号
        IBF : in std_logic_vector ( 1 downto 0 );--8255 インタフェイスの信号
        ACK : out std_logic_vector ( 1 downto 0 );--8255 インタフェイスの信号
        STB : out std_logic_vector ( 1 downto 0 );--8255 インタフェイスの信号
   --Right Memory
        DATA_R : inout std_logic_vector ( 31 downto 0 );-- アドレスを変更させるレジスタ
        ADRS_R : out std_logic_vector ( 16 downto 0 );--SRAM にに出すアドレスレジスタ
        SCS_R : out std_logic_vector ( 3 downto 0);--SRAM の内部信号
        SWE_R : out std_logic;--SRAM の内部信号
        SOE_R : out std_logic;--SRAM の内部信号
   --Left Memory
        DATA_L : inout std_logic_vector ( 31 downto 0 );-- アドレスを変更させるレジスタ
        ADRS_L : out std_logic_vector ( 16 downto 0 );--SRAM にに出すアドレスレジスタ
        SCS_L : out std_logic_vector ( 3 downto 0);--SRAM の内部信号
        SWE_L : out std_logic;--SRAM の内部信号
        SOE_L : out std_logic--SRAM の内部信号

 );

  attribute pinnum of A   : signal is
"BC23,BB24,BC25,BB26,BC27,BB28,BC29,BB30,BC31,BB32,BC33,BB34,BC35,BB36,BC37,BB38";
  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 CLK : signal is  "D22";
  attribute pinnum of ADRS_R : signal is
"T38,W37,Y38,AA37,AB38,AC37,AD38,AE37,AF38,AJ37,AK38,AL37,AM38,AN37,AP38,AR37,AT38";
  attribute pinnum of DATA_R : signal is
"F42,G43,H42,J43,K42,L43,M42,
N43,T42,U43,V42,W43,Y42,AA43,AB42,
AC43,AF42,AG43,AH42,AJ43,AK42,AL43,AM42,
AN43,AT42,AU43,AV42,AW43,AY42,BA43,BB42,BC43";
  attribute pinnum of SCS_R : signal is "AG39,AH40,AJ39,AM40";
  attribute pinnum of SWE_R : signal is "AN39";
  attribute pinnum of SOE_R : signal is "AP40";
  attribute pinnum of ADRS_L : signal is
"T6,W7,Y6,AA7,AB6,AC7,AD6,AE7,AF6,AJ7,AK6,AL7,AM6,AN7,AP6,AR7,AT6";
  attribute pinnum of DATA_L : signal is
"F2,G1,H2,J1,K2,L1,M2,N1,
T2,U1,V2,W1,Y2,AA1,AB2,AC1,
AF2,AG1,AH2,AJ1,AK2,AL1,AM2,AN1,
AT2,AU1,AV2,AW1,AY2,BA1,BB2,BC1";
  attribute pinnum of SCS_L : signal is "AG5,AH4,AJ5,AM4";
  attribute pinnum of SWE_L : signal is "AN5";
  attribute pinnum of SOE_L : signal is "AP4";
  attribute pinnum of CL   : signal is "AU23,AV24,AU25,AU33,AV34,AU35";
  attribute pinnum of OBF : signal is  "AV18,AV28";
  attribute pinnum of IBF : signal is  "AV20,AV30";
  attribute pinnum of ACK : signal is  "AU19,AU29";
  attribute pinnum of STB : signal is  "AU21,AU31";
end main;

architecture RTL of main is

component mem_ctrl is
port (  CLK  : in std_logic;-- クロック
        ADRS : out std_logic_vector(16 downto 0);--SRAM に出すアドレスレジスタ
        ADRS_MUX : in std_logic_vector (16 downto 0);-- アドレスを変更させるレジスタ
        DATA : in std_logic_vector(31 downto 0);-- データを SRAM に出すレジスタ
        DATA_BUF : out std_logic_vector(31 downto 0);-- データを SRAM に出すレジスタバッファ
        WRITE_DATA_REG : in std_logic_vector(31 downto 0);--PC から読み込んだデータを一時保管する
レジスタ
        READ_DATA_REG : out std_logic_vector(31 downto 0);--SRAM から読み込んだデータを一時保管す
るレジスタ
        SCS : out std_logic_vector( 3 downto 0);--SRAM の内部信号
```

```
        SOE : out std_logic;--SRAM の内部信号
        SWE : out std_logic;--SRAM の内部信号
        OE  : out std_logic;--SRAM の内部信号
        MEM_STATE_SEL : in std_logic_vector( 1 downto 0);-- メモリステートを管理する
        NEXT_MEM_CYCLE : out std_logic-- アドレスを上げる信号

         );
    end component;
component fpadder is
port(
         CLK  : in std_logic;-- クロック
         KA   : in std_logic_vector(31 downto 0);-- 加算器の 1 つめのデータレジスタ
         KB   : in std_logic_vector(31 downto 0);-- 加算器の 2 つめのデータレジスタ
         QQ    : out std_logic_vector(31 downto 0)-- 計算結果
      );
end component;
component fpmult is
port(
            CLK  : in std_logic;-- クロック
        FA   : in std_logic_vector(31 downto 0);-- 乗算器の 1 つ目のレジスタ
            FB   : in std_logic_vector(31 downto 0);-- 乗算器の 2 つ目のレジスタ
            Q    : out std_logic_vector(31 downto 0)-- 計算結果
      );

end component;


signal BH_BUF1 : std_logic;--BH を制御する信号
signal BH_BUF2 : std_logic;--BH を制御する信号
signal A_REG2 : std_logic_vector ( 15 downto 0);--PC に出力するレジスタ
signal DATA_BUF_R  : std_logic_vector ( 31 downto 0 );-- データのバッファ
signal DATA_BUF_L  : std_logic_vector ( 31 downto 0 );-- データのバッファ
signal ACK_BUF : std_logic_vector ( 1 downto 0 );--8255 インタフェイスのバッファ
signal STB_BUF : std_logic_vector (1 downto 0 );--8255 インタフェイスのバッファ
signal WRITE_DATA_ACTIVE :std_logic;-- メモリに書きこむデータの有無を知らせる信号
signal WRITE_DATA_ACTIVE_BUF :std_logic;-- メモリに書きこむデータの有無を知らせる信号
signal ADRS_MUX_R : std_logic_vector(16 downto 0);-- アドレスのレジスタ
signal ADRS_MUX_L : std_logic_vector(16 downto 0);-- アドレスのレジスタ
signal OE_R  : std_logic;--SRAM の内部信号
signal OE_L  : std_logic;--SRAM の内部信号
signal WRITE_DATA_REG_R : std_logic_vector(31 downto 0);--PC から読み込んだデータを一時保管するレ
ジスタ
signal WRITE_DATA_REG_RL : std_logic_vector(31 downto 0);--PC から "G み込んだデータを一時保管する
レジスタ
signal WRITE_DATA_REG_L : std_logic_vector(31 downto 0);--PC から読み込んだデータを一時保管するレ
ジスタ
signal WRITE_CNT : std_logic;--
signal READ_DATA_REG_R :std_logic_vector(31 downto 0);--SRAM から読み込んだデータを一時保管するレ
ジスタ
signal READ_DATA_REG_L :std_logic_vector(31 downto 0);--SRAM から読み込んだデータを一時保管するレ
ジスタ
signal MEM_STATE_SEL_R : std_logic_vector(1 downto 0);-- ステートの変更をする信号
signal MEM_STATE_SEL_L : std_logic_vector(1 downto 0);-- ステートの変更をする信号
signal NEXT_MEM_CYCLE_R : std_logic;-- アドレスを上げる信号
signal NEXT_MEM_CYCLE_L : std_logic;-- アドレスを上げる信号
signal V_D : std_logic_vector(30 downto 0);-- メモリから計算の為に呼び出した個数をカウントするレジ
スタ
signal V_D_BUF1 : std_logic_vector(30 downto 0);--PC からのデータ個数をカウントするレジスタ
signal V_D_BUF2 : std_logic_vector(30 downto 0);--PC からのデータ個数をカウントするレジスタ
signal V_D_BUF1_FLAG : std_logic;
signal SIGNAL1 : std_logic;--SIGNAL2 を制御するプログラム
signal SIGNAL2 : std_logic;--READ_CYCLE を制御するプログラム
signal READ_CYCLE2 : std_logic;-- ステートをストップさせる信号 READ_CYCLE2 を制御する信号
```

```
signal IN_CNT : std_logic;--PC からのデータをレジスタに振り分ける信号
signal OUT_CNT : std_logic;--PC に出力させるデータを 32 ビットにするための信号
signal START_CNT : std_logic;
signal FA : std_logic_vector(31 downto 0);-- 乗算器の 1 つ目のレジスタ
signal FB : std_logic_vector(31 downto 0);-- 乗算器の 1 つ目のレジスタ
signal Q : std_logic_vector(31 downto 0);-- 計算結果
signal KA : std_logic_vector(31 downto 0);-- 加算器の 2 つめのデータレジスタ
signal KB : std_logic_vector(31 downto 0);-- 加算器の 2 つめのデータレジスタ
signal QQ : std_logic_vector(31 downto 0);-- 計算結果
signal DATA_CNT : std_logic;-- 乗算器に出力するデータの順番
signal MULTI_CNT : std_logic;--1 の時乗算器にデータを出力する
signal ADDER_DATA_CNT : std_logic;--1 のとき加算器にデータを出力する
signal R_L_CHG : std_logic;-- 基板上の左右のメモリを選択する
signal V_D_SIGNAL : std_logic;--V_D を増やすための信号
signal V_D_REFRESH : std_logic;--V_D をクリアする信号
signal STOP_SIGNAL : std_logic;-- ステートをストップさせる信号
signal GYOURETU_KEISAN_START : std_logic;-- 行列計算のスタートを知らせる信号
signal LOOP3_END : std_logic;--LOOP3 が終了したことを知らせる信号
signal LOOP2_END : std_logic;--LOOP2 が終了したことを知らせる信号
signal LOOP1_END : std_logic;--LOOP1 が終了したことを知らせる信号
signal ALL_COUNT : std_logic_vector(16 downto 0);--LOOP1 のカウ・ト
signal LOOP1_COUNT : std_logic_vector(16 downto 0);--LOOP1 のカウント
signal LOOP2_COUNT : std_logic_vector(16 downto 0);--LOOP2 のカウント
signal LOOP3_COUNT : std_logic_vector(16 downto 0);--LOOP3 のカウント
signal LOOP2_SIGNAL : std_logic;--LOOP3 の終了を前もって知らせる LOOP2 を増やす
signal ALL_SIGNAL : std_logic;
signal LOOP1_SIGNAL : std_logic;
signal LOOP3_SIGNAL : std_logic;--LOOP3 と LOOP1 を増やす
signal LOOP3_REFRESH : std_logic;--LOOP3 をクリアする
signal LOOP3_FLAG : std_logic;--1 つの行列計算の終了を制御す
る信号
signal MEM_WRITE_KEKKA : std_logic;--LOOP3 の結果をメモリーに書き込む
signal MEM_WRITE_KEKKA_CYCLE : std_logic;-- 上のサイクルを知らせる信号
signal KEKKA_ADRS : std_logic_vector(16 downto 0);--LOOP3 で出た結果をメモリーに書きこむアドレスを
保持する
signal READ_ADRS : std_logic_vector(16 downto 0);--PC に読み出す時のアドレスを保持する
signal CLEAR_FLAG : std_logic;--LOOP3 が終了したなら加算器、乗算器をクリアする
signal KEKKA_ADRS_CNT : std_logic;--KEKKA_ADRS を増やす
constant STOP2 : std_logic_vector(1 downto 0):= "00";
constant WRITE : std_logic_vector(1 downto 0):= "11";
constant READ : std_logic_vector(1 downto 0):= "10";
constant DATA_COUNT : std_logic_vector(16 downto 0):="10000000000000000";-- 行列の次数を定数化
constant YOUSO : std_logic_vector(16 downto 0):="00000000100000000";-- ベクトルの要素を定数化

type STATE_TYPE2 is
     (STOP,START,SRAM_READ1,WAIT1,SRAM_READ2,WAIT2,SRAM_READ3,
WAIT3,WAIT4,LAST_MULTI,WAIT5,LAST_ADD,RESULT_WRITE,WAIT6,WAIT7);
signal CURRENT_STATE2 : STATE_TYPE2;
signal NEXT_STATE2 : STATE_TYPE2;

begin
mem_r : mem_ctrl port map (CLK=>CLK, ADRS=>ADRS_R,
ADRS_MUX=>ADRS_MUX_R, DATA=>DATA_R, DATA_BUF=>DATA_BUF_R,
WRITE_DATA_REG=>WRITE_DATA_REG_R,READ_DATA_REG=>READ_DATA_REG_R,
SCS=>SCS_R, SOE=>SOE_R, SWE=>SWE_R, OE=>OE_R,
MEM_STATE_SEL=>MEM_STATE_SEL_R, NEXT_MEM_CYCLE=>NEXT_MEM_CYCLE_R);
mem_l : mem_ctrl port map (CLK=>CLK, ADRS=>ADRS_L,
ADRS_MUX=>ADRS_MUX_L, DATA=>DATA_L, DATA_BUF=>DATA_BUF_L,
WRITE_DATA_REG=>WRITE_DATA_REG_L,READ_DATA_REG=>READ_DATA_REG_L,
SCS=>SCS_L, SOE=>SOE_L, SWE=>SWE_L, OE=>OE_L,
MEM_STATE_SEL=>MEM_STATE_SEL_L, NEXT_MEM_CYCLE=>NEXT_MEM_CYCLE_L);
add : fpadder  port map (CLK,KA,KB,QQ);
mul : fpmult   port map (CLK,FA,FB,Q);
```

```
--No1
-- メモリに書きこむ際のハイインピーダンス状態の変更を制御するプロセス
process ( OE_R )begin
 if OE_R = '1' then-- OE はの SRAM の信号
    DATA_R <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ"; -- ハイインピーダンスに設定
 else
    DATA_R <= DATA_BUF_R;-- メモリから読み込む
 end if;
end process;
process ( OE_L )begin
 if OE_L = '1' then-- OE はの SRAM の信号
    DATA_L <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ"; -- ハイインピーダンスに設定
 else
    DATA_L <= DATA_BUF_L;-- メモリから読み込む
 end if;
end process;
--A レジスタのハイインピーダンス設定
A <= "ZZZZZZZZZZZZZZZZ" when ACK_BUF = "00" else A_REG2;
ACK_BUF <= OBF;--used in No4
ACK <= ACK_BUF;--used in No4
STB <= STB_BUF;--used in No4
--No2
--PC からデータを FPGA に送りそれを 32 ビットにまとめるプロセス
process ( BL(0),BL(6),OBF,IN_CNT,MEM_WRITE_KEKKA_CYCLE) begin
if MEM_WRITE_KEKKA_CYCLE = '1' then-- もし LOOP3 の結果をメモリ
に書き込むなら
        WRITE_DATA_REG_R <= QQ;-- 書きこむデータは計算結果を書きなさい
elsif  OBF'event and OBF = "11" then--OBF は PC からデータを FPGA に出すと真になる

      if IN_CNT = '0' then -- 上位、下位を振り分ける
         IN_CNT <= '1';-- 次は上位
         WRITE_DATA_ACTIVE_BUF <= '0';-- used in No3
         START_CNT <= '1';-- メモリ書きこみ準備
         if R_L_CHG = '0' then-- 左右のメモリ変更
            WRITE_DATA_REG_R(15 downto 0) <= A;-- 下位 16 ビットをレジスタに代入
         else
            V_D_BUF1_FLAG <= '0';
            WRITE_DATA_REG_L(15 downto 0) <= A;-- 下位 16 ビットをレジスタに代入
         end if;
       elsif IN_CNT = '1' then  -- 上位、下位を振り分ける
         IN_CNT <= '0'; -- 次は下位
         WRITE_DATA_ACTIVE_BUF <= '1';--used in No3
         START_CNT <= '0';-- メモリ書きこみ準備をもう一度初めから
         if R_L_CHG = '0' then-- 左右のメモリ変更
            WRITE_DATA_REG_R(31 downto 16) <= A;-- 上位 16 ビットをレジスタに代入
         else
            V_D_BUF1_FLAG <= '1';--PC からのデータ個数をカウントする used in 3.2
            WRITE_DATA_REG_L(31 downto 16) <= A;-- 上位 16 ビットをレジスタに代入
         end if;
       end if;

 end if;
if BL(0) = '1' then-- 初期化信号,
   IN_CNT <= '0';-- 初めは下位から
   R_L_CHG <= '0';-- 初めはライトメモリから
   WRITE_DATA_ACTIVE_BUF <= '0';--used in No3
elsif BL(6) = '1' then-- もし PC からのデータ転送が終わったら
     WRITE_DATA_ACTIVE_BUF <= '0';--
     R_L_CHG <= '1';--

end if;
end process;
--No3
-- 信号をクロックに同期させる
process (CLK) begin
  if rising_edge(CLK) then
     WRITE_DATA_ACTIVE <= WRITE_DATA_ACTIVE_BUF;-- クロックに同期させる
  end if;
end process;
```

```
--No4
-- アドレスを NEXT_MEM_CYCLE に応じて変化させるプロセス
process(CLEAR_FLAG,BL(2),MEM_WRITE_KEKKA_CYCLE,LOOP2_END,BL(0),

NEXT_MEM_CYCLE_R,GYOURETU_KEISAN_START) begin
  if falling_edge(NEXT_MEM_CYCLE_R)  then-- 一つの書きこみ、読みみ込みが終わったならば
    if GYOURETU_KEISAN_START = '1' then-- もし行列計算が始まっているならば
         ADRS_MUX_R <= ADRS_MUX_R + YOUSO;-- アドレスを行列の
要素ずつ増やす --used in MEM_CTRL
    elsif  GYOURETU_KEISAN_START = '0' then
         ADRS_MUX_R <= ADRS_MUX_R + '1';-- それ以外は 1 ずつ増
やす (結果の書込み)
    end if;
 end if;
 if BL(2) = '1' then
       ADRS_MUX_R <=  READ_ADRS;--PC から計算結果を読み込むときはこのアドレスを使う
 elsif MEM_WRITE_KEKKA_CYCLE = '1' then --LOOP3 の計算結果を
メモリに書込むときは
       ADRS_MUX_R <=  KEKKA_ADRS;-- もし LOOP3 の計算結果をメモ
リに書きこむときはこのアドレスを使う

 elsif BL(0) = '1'  then-- 初期化
       ADRS_MUX_R <= "0000000000000000";--used in MEM_CTRL
 elsif CLEAR_FLAG = '1' then
       ADRS_MUX_R <= LOOP1_COUNT;
 end if;
 end process;
--No5
--PC に結果を返す時のアドレスの計算
process (BL(3),BL(0))begin
 if BL(0) = '1' then-- 初期化
     READ_ADRS <= DATA_COUNT;-- 計算結果は何も書き込んで無いところにあるので
 elsif falling_edge(BL(3)) then--PC からの制御信号
     READ_ADRS <= READ_ADRS + '1';-- アドレスを 1 つ増やす
 end if;
end process;
--No6
--LOOP3 の計算結果をメモリに書き込む時のアドレス計算
process (KEKKA_ADRS_CNT,BL(0))begin
   if falling_edge(KEKKA_ADRS_CNT) then
        KEKKA_ADRS <= DATA_COUNT + ALL_COUNT;-- 計算結果は何
も書きこんでないところに・く
   end if;
   if BL(0) = '1' then
        KEKKA_ADRS <= DATA_COUNT;-- 初期化
   end if;
 end process;

--No7
-- アドレス (LEFT) のアドレスを計算する
process(BL(0),NEXT_MEM_CYCLE_L,LOOP2_END) begin
if BL(0) = '1' or LOOP2_END = '1' then-- 初期化信号と LOOP2 の終了
      ADRS_MUX_L <= "0000000000000000";--used in MEM_CTRL
elsif falling_edge(NEXT_MEM_CYCLE_L)  then-- もし書きこみ、読み込みが終わったら

     ADRS_MUX_L <= ADRS_MUX_L + '1';-- アドレスを 1 ずつ増やす

end if;

end process;
--No8
--PC に計算結果を返すプロセス
process ( IBF, BL(0),BL(1),OUT_CNT) begin

          if BL(0) = '1' then -- 初期化信号
            OUT_CNT <= '0';
            A_REG2 <= "0000000000000000";--used in the top
            STB_BUF <= "11";--8255 インタフェイスの制御信号

          elsif  falling_edge( BL(1) ) then--PC からの信号
            if OUT_CNT = '0' then
```

```
              OUT_CNT <= '1';
              STB_BUF <= "00";--8255 インタフェイスの制御信号
              A_REG2 <= READ_DATA_REG_R(15 downto 0);-- 内積の結果の下位 16 ビットの出力
            else
              OUT_CNT <= '0';
              STB_BUF <= "00";-- インタフェイスの制御信号
              A_REG2 <= READ_DATA_REG_R(31 downto 16);-- 内積の結果の下位 16 ビットの出力
            end if;
          end if;
       if IBF = "11" then
            STB_BUF <= "11";-- インタフェイスの制御信号

      end if;
 end process;
--No9
-- 掛け算をするためにメモリからのデータを振り分けるプロセス
process begin
wait until CLK'event and CLK = '0';
  if MULTI_CNT = '1' then
        FA <= READ_DATA_REG_R;-- 掛け算の 1 つ目の要素 --used in FPMULTI
        FB <= READ_DATA_REG_L ;-- 掛け算の 2 つ目の要素 --used in FPMULTI


  elsif BL(0) = '1' or CLEAR_FLAG = '1' then-- 初期化信号


     DATA_CNT <= '0';
     FA <= "00000000000000000000000000000000";--FA の初期化 used in FPMULTI
     FB <= "00000000000000000000000000000000";--FB の初期化 used in FPMULTI
  end if;
  end process;

--No10
-- 足し算をするために乗算器からデータを振り分けるプロセス
process begin
  wait until CLK'event and CLK = '0';
   if ADDER_DATA_CNT = '1' then

      KA <= Q;-- 乗算器の結果を代入 used in ADDER
      KB <= QQ;-- 加算器の結果を再び代入 used in ADDER

   elsif BL(0) = '1' or CLEAR_FLAG = '1' then-- 初期化信号
      KB <= "00000000000000000000000000000000";--KB の初期化 used in ADDER
      KA <= "00000000000000000000000000000000";--KA の初期化 used in ADDER
   end if;
 end process;
--No11
-- メモリコントローラの制御をするプロセス
 process begin
  wait until CLK'event and CLK = '0'; -- クロックの立ち下がり
    if WRITE_DATA_ACTIVE = '1' then-- もし PC からデータが入力されたなら
        if WRITE_CNT = '1' then-- もしここに来るのが一回目なら
           if R_L_CHG = '0' then-- もし RIGHT メモリなら
              WRITE_CNT <= '0';-- もう書きこめない
              MEM_STATE_SEL_R <= WRITE;-- メモリコントローラをライトにする used in MEM_CTRL
           else-- もし LEFT メモリなら
              WRITE_CNT <= '0';
              MEM_STATE_SEL_L <= WRITE;-- リコントローラをライトにする used in MEM_CTRL
           end if;
        elsif WRITE_CNT = '0' then--2 回目なら
           if R_L_CHG = '0' then-- もし RIGHT メモリなら
              MEM_STATE_SEL_R <= STOP2;-- 一度 WRITE にしたらもう書きこまない
           else-- もし LEFT メモリなら
              MEM_STATE_SEL_L <= STOP2;-- 一度 WRITE にしたらもう書きこまない
           end if;
        end if;
    elsif START_CNT = '1' then-- 新しいデータが来たなら
          WRITE_CNT <= '1';-- 書きこみ可能にする
    elsif MEM_WRITE_KEKKA = '1' then-- もし LOOP3 の計算結果を書き込むなら
          MEM_STATE_SEL_R <= WRITE;--RIGHT メモリに書きこむ
```

```
    elsif BL(2) = '1' then-- もし PC が結果を読む命令を出したなら
          MEM_STATE_SEL_R <= READ;--READ にしなさい
    elsif READ_CYCLE2 = '1' then-- もし計算をするときにデータを欲したなら
        MEM_STATE_SEL_R <= READ;-- メモリコントローラをリードにする  used in MEM_CTRL
        MEM_STATE_SEL_L <= READ;-- メモリコントローラをリードにする  used in MEM_CTRL

     else
        MEM_STATE_SEL_R <= STOP2;-- メモリコントローラをストップにする  used in MEM_CTRL
        MEM_STATE_SEL_L <= STOP2;-- メモリコントローラをストップにする  used in MEM_CTRL


     end if;
    if BL(0) = '1' then-- 初期化
        MEM_STATE_SEL_R <= STOP2;-- メモリコントローラをストップにする  used in MEM_CTRL
        MEM_STATE_SEL_L <= STOP2;-- メモリコントローラをストップにする  used in MEM_CTRL

    elsif BL(6) = '1' then
        WRITE_CNT <= '0';

    end if;
end process;
--No11.5
-- 計算終了を制御するプロセス
process (LOOP1_SIGNAL,BL(0)) begin
   if BL(0) = '1' then-- もし初期化信号、 LOOP2 が終了したなら
     LOOP1_COUNT <= "00000000000000000";
   elsif falling_edge(LOOP1_SIGNAL) then-- もし LOOP3 が終わるなら
     LOOP1_COUNT <= LOOP1_COUNT + '1';--LOOP2_CNT に 1 加える
  end if;
 end process;
--No12
--LOOP2__CNT を計算するプロセス
process (LOOP2_SIGNAL,BL(0),LOOP2_END) begin
   if BL(0) = '1' or LOOP2_END = '1' then-- もし初期化信号、 LOOP2 が終了したなら
     LOOP2_COUNT <= "00000000000000000";
   elsif falling_edge(LOOP2_SIGNAL) then-- もし LOOP3 が終わるなら
     LOOP2_COUNT <= LOOP2_COUNT + '1';--LOOP2_CNT に 1 加える
  end if;
 end process;
--No13
--LOOP3、 LOOP1 の計算
 process(LOOP3_SIGNAL,BL(0),LOOP3_REFRESH) begin
  if BL(0) = '1' then
     LOOP3_COUNT <= "00000000000000000";-- 初期化
  elsif LOOP3_REFRESH = '1' then--LOOP3 が終わったら
     LOOP3_COUNT <= "00000000000000000";-- 初期化
  elsif falling_edge(LOOP3_SIGNAL) then-- もし LOOP3 が動いているなら
     LOOP3_COUNT <= LOOP3_COUNT + '1';--LOOP3 に 1 を加える
  end if;

end process;
--No14
-- ループ 2 を制御するプロセス
process(CLK,LOOP2_COUNT) begin--
   if rising_edge(CLK) then--
      if LOOP2_COUNT = YOUSO then
         LOOP1_SIGNAL <= '1';
         LOOP2_END <= '1';
      else
         LOOP1_SIGNAL <= '0';
         LOOP2_END <= '0';
      end if;
   end if;
end process;
--No15
-- ループ 1 を制御するプロセス
process(CLK,LOOP1_COUNT) begin
   if  falling_edge(CLK) then
       if LOOP1_COUNT = YOUSO then
          STOP_SIGNAL <= '1';
       else
```

```
         STOP_SIGNAL <= '0';
       end if;
   end if;
end process;
--No16
--計算終了をカウントするプロセス
process (ALL_SIGNAL,BL(0))begin
   if BL(0) = '1' then
          ALL_COUNT <= "00000000000000000";--初期化
   elsif falling_edge (ALL_SIGNAL) then
     ALL_COUNT <= ALL_COUNT + '1';--LOOP1に1を加える
   end if;

end process;
--No17
--メインプログラム

process (CLK,LOOP3_COUNT,LOOP3_FLAG,STOP_SIGNAL,BL(7),BL(0),CURRENT_STATE2)begin
  if rising_edge(CLK) then
       case CURRENT_STATE2 is
       when  START =>
             CLEAR_FLAG <= '0';
             NEXT_STATE2 <= SRAM_READ1;
       when  SRAM_READ1 =>
             READ_CYCLE2 <= '1';
             LOOP3_SIGNAL <= '1';
             NEXT_STATE2 <= WAIT1;
       when  WAIT1 =>
             LOOP3_SIGNAL <= '0';
             NEXT_STATE2 <= SRAM_READ2;
       when  SRAM_READ2 =>
             LOOP3_SIGNAL <= '1';
             NEXT_STATE2 <= WAIT2;
       when  WAIT2 =>
             LOOP3_SIGNAL <= '0';
             MULTI_CNT <= '1';
             NEXT_STATE2 <= SRAM_READ3;
       when  SRAM_READ3 =>
             LOOP3_SIGNAL <= '1';
             MULTI_CNT <= '0';
             ADDER_DATA_CNT <= '0';
             NEXT_STATE2 <= WAIT3;
       when  WAIT3 =>
             LOOP3_SIGNAL <= '0';
             MULTI_CNT <= '1';
             ADDER_DATA_CNT <= '1';
             NEXT_STATE2 <= SRAM_READ3;
       when  WAIT4 =>
             LOOP3_REFRESH <= '1';
             MULTI_CNT <= '0';
             ADDER_DATA_CNT <= '0';
             READ_CYCLE2 <= '0';
             NEXT_STATE2 <= LAST_MULTI;
       when  LAST_MULTI =>
             LOOP3_REFRESH <= '0';
             MULTI_CNT <= '1';
             ADDER_DATA_CNT <= '1';
             ALL_SIGNAL <= '1';
             LOOP2_SIGNAL <= '1';
             NEXT_STATE2 <= WAIT5;
      when   WAIT5 =>
             MULTI_CNT <= '0';
             ADDER_DATA_CNT <= '0';
             ALL_SIGNAL <= '0';
             LOOP2_SIGNAL <= '0';
             NEXT_STATE2 <= LAST_ADD;
      when   LSAT_ADD =>
             ADDER_DATA_CNT <= '1';
             MEM_WRITE_KEKKA_CYCLE <= '1';
             NEXT_STATE2 <= RESULT_WRITE;
      when   RESULT_WRITE =>
             ADDER_DATA_CNT <= '0';
             MEM_WRITE_KEKKA <= '1';
             NEXT_STATE2 <= WAIT6;
      when   WAIT6 =>
```

```
              MEM_WRITE_KEKKA <= '0';
              KEKKA_ADRS_CNT <= '1';
              NEXT_STATE2 <= WAIT5;
              MEM_WRITE_KEKKA_CYCLE <= '0';
       when   WAIT7 =>
              KEKKA_ADRS_CNT <= '0';
              CLEAR_FLAG <= '1';
              NEXT_STATE2 <= START;
       when   STOP =>
              CLEAR_FLAG <= '1';
              NEXT_STATE2 <= STOP;
       when   others =>
              NEXT_STATE2 <= STOP;

       end case;
     end if;
  if falling_edge(CLK) then
     if LOOP3_COUNT = YOUSO then-- もし 1つの行列計算が終了
なら
        READ_CYCLE2 <= '0';-- SRAM 連続読み出し終了
        if LOOP3_FLAG = '1' then もし1度もここに来ていないな
ら
           CURRENT_STATE2 <= WAIT4; 残りのデータを処理しない
さい
           LOOP3_FLAG <= '0';
        else
              CURRENT_STATE2 <= NEXT_STATE2;
        end if;
     if BL(7) = '1' then--PCからの計算開始シグナル
        GYOURETU_KEISAN_START <= '1';-- 行列計算があらゆるところで優先される
        CURRENT_STATE2 <= START;
     elsif BL(0) = '1' or STOP_SIGNAL = '1' then-- 初期化
        LOOP3_FLAG <= '1';
        CURRENT_STATE2 <= STOP;
        GYOURETU_KEISAN_START <= '0';
     else
        LOOP3_FLAG <= '1';
        CURRENT_STATE2 <= NEXT_STATE2;
     end if;
   end if;
end process;


end RTL;
```

付録 B

回路設計をする上でのソフトウェアの使用法

ここでは、回路設計をする上でのソフトウェアの使用法して、 Accolade 社の PeakFPGA と Altera 社の Max+PLUSII の使用方法と、 FPGA のダウンロード方法について説明する。以下は沼 [6] の修士論文から要約した物に、さらに必要な事を付け加えたものである。

B.1 PeakFPGA

B.1.1 VHDL ファイル作成における注意点

この研究で PeakFPGA を使用する場合、 VHDL ファイルの作成方法について説明する。

まず最初に、

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

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

を記述し、 VHDL のライブラリを呼び出す。また、 metamor のライブラリを呼び出すときには、 PeakFPGA で Synopsys library を呼び出すようにしなければならない。

そして、次の entity という部分で、 FPGA の入出力ポートとピンの設定をする。

```
entity count is
    port ( CLK : in std_logic;
           A   : inout std_logic_vector(15 downto 0);
           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 A   : signal is ‘‘BC23,BB24,BC25,BB26,BC27,BB28,BC29,BB30, ..
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;
```

まず、 entity 文でモジュール名を指定する。これは FPGA にダウンロードするときのファイル名にもなるので、長い名前はつけない方が良い。そして、 port 文で FPGA のポートを指定する。 FPGA にデータを入力する場合には in、データを出力する場合には out、両方の場合には inout を指定する。

そして attribute 文でピン番号を指定する。各ポートのピン番号は?? 章に示す。

```
architecture RTL of count is

signal CLK_CNT : std_logic_vector(7 downto 0);
signal RESET   : std_logic;

begin

end RTL;
```

次に architecture 文で回路の設計要素を記述する。 count の部分は entity の名前と同じにする。 begin 文の前で、内部信号 signal を宣言し、 begin 文以降は回路の動作を記述する。

B.1.2 VHDL ファイルの作成

まず、 VHDL ファイルの作成方法について説明する。

- 最初に PeakVHDL を起動する。

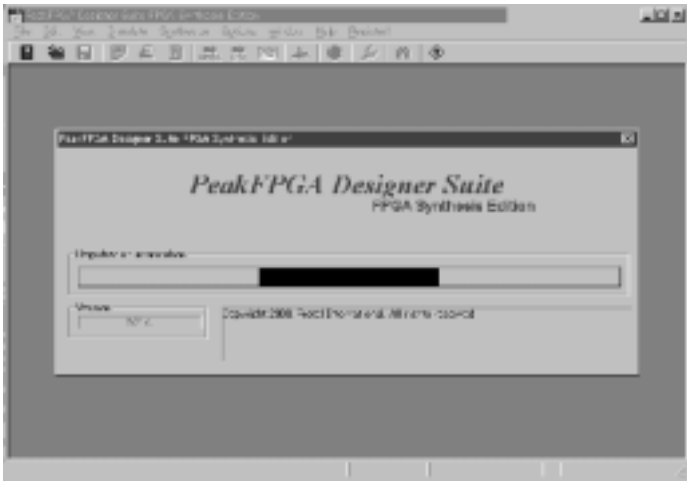


図 2.1: PeakFPGA を開く[1]

- “File” にある “New Project” を選択。

図 2.2: “メニュー” の “ファイル” を選択[2]

- 下のウインドウが表われたら、右クリックを押して”Add Module” を選択、ダイアログボックスが表われたら、”The Project has not been saved. Save it now?” と出てくるので、”OK” を押す、するとダイアログボックスが表われるので、 ACC(*.acc) ファイルに名前をつけて保存する。この ACC ファイルはプロジェクトのファイルなので、例えば内積のプログラムを作る時は naiseki1.acc 等、分かりやすい名前を付けることが必要である。

[1] ファイル名:u00simi/eps/peakfpga.eps

[2] ファイル名:u00simi/eps/select.eps

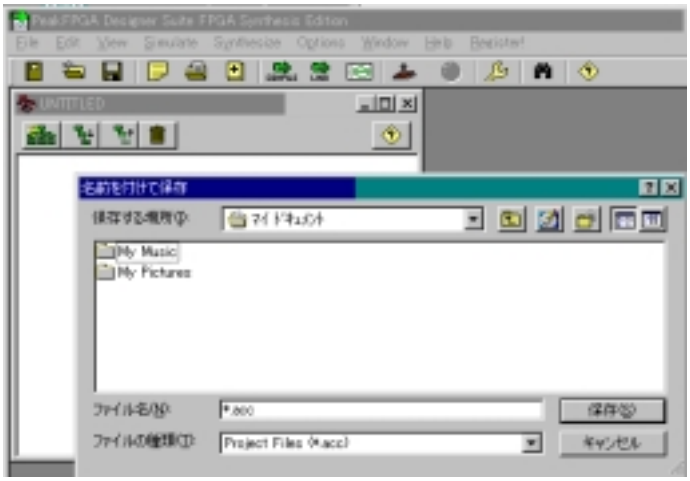


図 2.3: “開くを選択”[3]

- すると、今度は “開く” というダイアログボックスが表われるが、まだ VHDL ソースファイルを作っていないのでキャンセルを押す。そして、再び “File” を選択し、その中の “New Module” を選ぶ。すると、図 2.4が現れ、その中の “Cleate Blank Module” を選択する。すると名前を求めてくるので、そこにモジュール名を記述する。これによって VHDL が記述出来る VHD(*.vhd) ファイルが作製される。再び追加したい時も、同じ事を繰り返せば良い。

図 2.4: “NewModule” を選択[4]

- VHDL で記述した VHDL ファイルを構文解析するには、上部のメニューにある”COMPILE” というボタンを押す。するとコンパイルが始まり、中央にウインドウが表われる。構文に間違いがあったらここにエラーの原因とその行が示されるので、先の VHDL が書かれているウインドウから間違いを直す。

[3]ファイル名:u00simi/eps/open.eps

[4]ファイル名:u00simi/eps/newmodule.eps

- コンパイルが正しければ、次は論理合成を行なう。メニューの”Option” から”Synthesize” を選ぶ。すると、下のウインドウが表われる。ここで、右側にある”Device Family” から”Altera all Devices (EDIF)” を選択する。そして、左下にある”Include Synopsys Library” をチェックする。

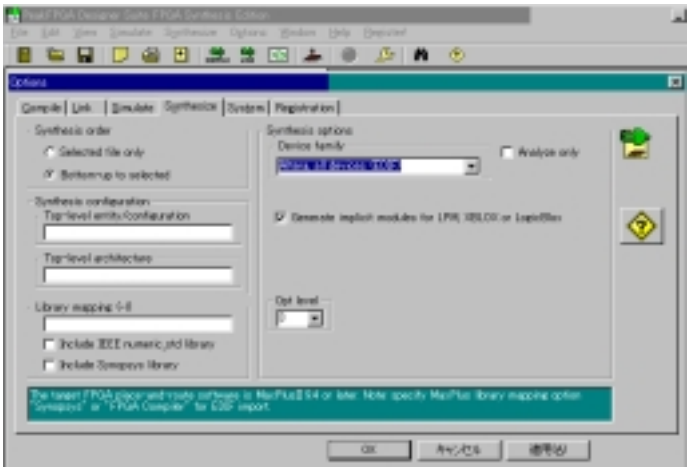


図 2.5: デバイスの選択[5]

- そして、上部のデバイスマーク、もしくは “メニュー” の “Synthesize” を選ぶ。すると、別のウインドウで論理合成が行なわれる。論理合成が正しく行なわれたならば、モジュール名のついた EDF ファイルが作成される。

図 2.6: 論理合成[6]

- このプロジェクトファイルの中で、 VHDL ファイルを 2 つ以上合成して、ひとつのモジュールを作ることができる。このとき、それぞれのファイルは component または function で

[5]ファイル名:u00simi/eps/syn.eps

[6]ファイル名:u00simi/eps/ronri.eps

ファイルがリンクできていなければならない。そのリンクができているかを確かめるには、“Rebuild Hierarchy” というボタンを押して、リンクし直し、 “Show Hieralchy” というボタンを押して component や function ができているかを確かめる。 VHDL ファイルを手直しした時は必ず “Rebuild Hierarchy” を押すことを心掛けた方が良い。

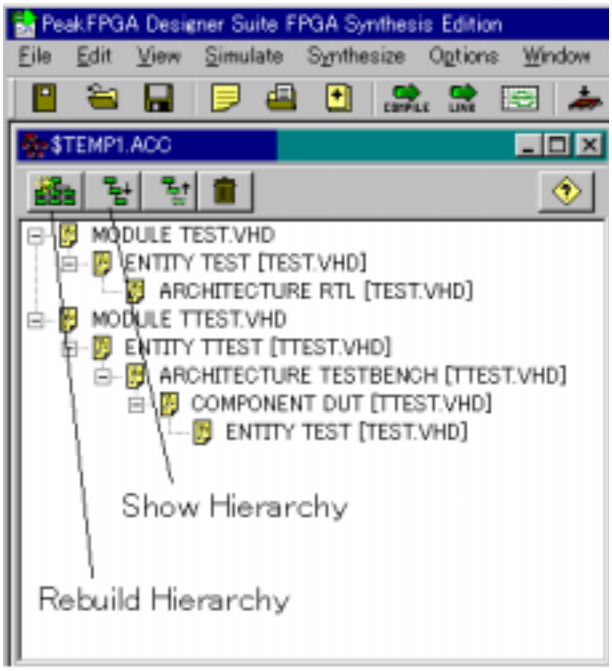


図 2.7: リンクの検査[7]

B.1.3 VHDL で記述する時の注意点

ここでは VHDL を使う時の注意点を説明する。

std_logic を使った計算

例えば

signal A : std_logic_vector(16 downto 0);

で定義した信号を作る。これに CNT という信号が ‘1‘ の時に ”00000000000000001” を足すときには、

if CNT = ’1’ then

A = A + “00000000000000001”;

[7]ファイル名:u00simi/eps/link.eps

end if;
と書きたくなるが、これでは間違いである。 CNT が ‘1‘ の時はプログラム上ではある 1 点かも知れないが、ハードウェア上では ‘1‘ になった状態の連続と見なされる。よって正しい結果は得られない。そこで CNT が ‘1‘ になった瞬間にある結果を実行させるという風に書かなければならない。
if rising_edge(CNT) then
A = A + “0000000000000001”;
end if;
CNT が立ち上がる事はハードウェア上では 1 点しかかないのでこれで正しい動作が行われる。

信号の定義

これは std_logic_vector で定義した信号の計算をするとき、信号の幅を 32bit 以上で定義した信号の計算は Max+Plus 2 のコンパイルエラーを生じる。よって先ほど述べた用な信号の計算を行う場合は 31bit 幅で定義しなければならない。
signal A : std_logic_vector(31 downto 0);

B.1.4 PeakVHDL でのシミュレーション方法

シミュレーションはプロジェクトウインドウにテストベンチ用の VHDL ファイルを挿入することでできる。その VHDL ソースには、テストベンチの対象となるソースと同じ port を宣言し、それと同じ signal も宣言する。そして、その port に使われている信号を並べた DUT 文を begin 文の後に挿入する。 proecss には入力ポートにたいする信号を記述する。このテストベンチ用の VHDL ファイルもテストするファイルにリンクしているので、必ず “Rebuild Hierarchy” を押してリンクさせる必要がある。例として、シミュレーションをする VHDL ソースとシミュレーションの VHDL ソースを以下に示す。このファイルは BL ポートの入力信号により出力信号の A,BH 信号を変化させる VHDL ソースである。

```
-----------------
-- ソースファイル
-----------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

library metamor;
use metamor.attributes.all;
 entity test is

  port (
         CLK : in std_logic;-- クロック
         A   : inout std_logic_vector (15 downto 0);--A レジスタ（2 方向）
         BL  : in std_logic_vector ( 7 downto 0);--B レジスタ（入力）
         BH  : out std_logic_vector ( 7 downto 0 )--B レジスタ（出力）


 );

  attribute pinnum of A   : signal is  "BC23,BB24,BC25,BB26,BC27,BB28,BC29,BB30,BC31,BB32,BC33,BB34,
  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 CLK : signal is  "D22";
end test;

architecture RTL of test is


begin
 process (BL,CLK) begin
 if falling_edge(CLK) then
   case BL is--BL 信号の場合別け
        when "00000000" =>-- BL が’’00000000’’ の時
             A <= "0000000000000000";--A に’’0000000000000000’’
を出力
             BH <= "00000000";--BH に’’00000000’’ を出力
        when "00000001" =>-- BL が’’00000001’’ の時
             A <= "0000000000000001";--A に’’0000000000000001’’
を出力
             BH <= "00000000";--BH に’’00000000’’ を出力
        when "00000010" =>-- BL が’’00000010’’ の時
             A <= "0000000000000010";--A に’’0000000000000010’’
を出力
             BH <= "00000000";--BH に’’00000000’’ を出力
        when "00000011" =>-- BL が’’00000011’’ の時
             A <= "0000000000000011";--A に’’0000000000000011’’
を出力
```

```
             BH <= "00000000";--BH に''00000000'' を出力
        when others =>-- BL がその他の時
             BH <= "11111111";--BH に''11111111'' を出力
    end case;
  end if;
  end process;
end RTL;

-----------------
-- シミュレートファイル
-----------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

library metamor;
use metamor.attributes.all;
 entity ttest is
 end ttest;

architecture TESTBENCH of ttest is
 component test is
  port(
         CLK : in std_logic;-- クロック
         A   : inout std_logic_vector (15 downto 0);--A レジスタ (2 方向)
         BL  : in std_logic_vector ( 7 downto 0);--B レジスタ (入力)
         BH  : out std_logic_vector ( 7 downto 0 )--B レジスタ (出力)
  );
 end component;


  signal CLK : std_logic;-- クロック
  signal A   : std_logic_vector (15 downto 0);--A レジスタ (2 方向)
  signal BL  : std_logic_vector ( 7 downto 0);--B レジスタ (入力)
  signal BH  : std_logic_vector ( 7 downto 0 );--B レジスタ
(出力)

begin

  DUT : test port map (CLK,A,BL,BH);-- 上で宣言した通りにポー
トマップを宣言
process begin-- クロックの作製、周期は 20ns
  CLK <= '0';
 wait for 10 ns;--Low は 10ns ずつ
  CLK <= '1';
 wait for 10 ns;--High は 10ns ずつ
end process;

process begin -- 以下にシミュレート内容を記述
  wait for 100 ns;
    BL <= "00000000";
  wait for 100 ns;
    BL <= "00000001";
  wait for 100 ns;
    BL <= "00000010";
  wait for 100 ns;
    BL <= "00000011";
  wait for 100 ns;
    BL <= "11111111";
  wait for 1000 ns;
end process;

end TESTBENCH;
```

シミュレーションの仕方について説明する。まず、先に述べたプロジェクトウインドウにテストベンチの VHDL ソースを追加する。そして、 Rebulid したあとに、テストベンチのファイルを選択して上部のメニューの “Simulation” から”Load Selected” を選ぶ。すると、構文解析と同じウインドウが表われ、コンパイルが始まる。コンパイルが成功すると、図 2.8のようなウインドウが表われる。

図 2.8: 信号の選択画面[8]

図 2.8はシミューレートしたい信号を全体の信号から選択するウィンドウである。図 2.8のウインドウが表われたら、シミュレーションしたい内部信号を選択し、”close” を押す。そして、 “メニュー” にあるスパナのアイコンを押して、シミュレーションする動作時間を決める。それを決めたら、 “メニュー” のボタンのような “Go” アイコンを押して、シミュレーションを開始する。シミュレーションに成功すると、図 2.9の画面となる。図 2.9より実際のソースファイルの通りに動作している事が分かる。

図 2.9: シミュレーション成功[9]

[8]ファイル名:u00simi/eps/simyu_face.eps

[9]ファイル名:u00simi/eps/logic.eps

B.2 Max+PLUS2

論理合成が終わったら、次は FPGA に搭載できるファイルに置き換えなければならない。ここではその方法について説明する。まず、 Max+PLUS2 を起動する。そして、 “File” の “Open” で論理合成した EDF ファイルを選択する。

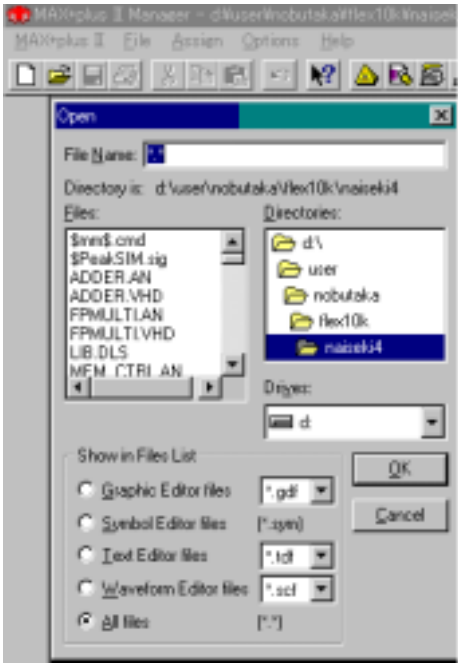


図 2.10: Max+Plus2 オープニング[10]

次に “Assign” の “Device” を選択し、ダウンロードする FPGA を選択する。本研究では、 EPF10KGC504 を使用しているので、それを選択する。そして “’Devaice Options” を選び、チェックをすべて外す。

[10]ファイル名:u00simi/eps/max_open.eps

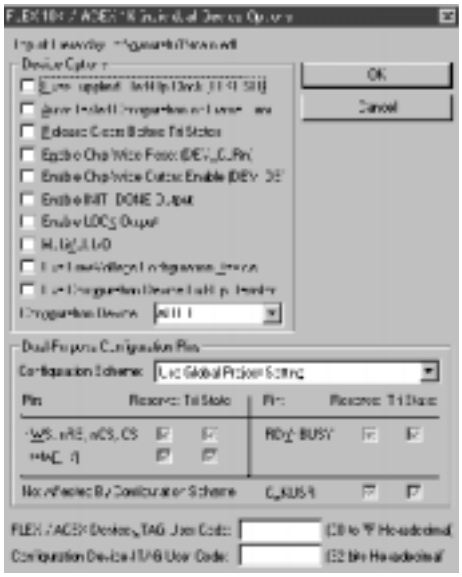


図 2.11: オプション画面[11]

次に、メニューの “MAX+plusII から “Compiler” を選択する。すると次のような画面が出てくるので、 “start” ボタンを押す。

図 2.12: コンパイル開始[12]

コンパイルが終わると、 TTF ファイルが作成されるので、あとはこれを FPGA にダウンロードすればよい。

B.3 FPGA へのコンフィグレーション

まず DOS プロンプトを開き作製した TTF ファイルのあるフォルダに移る。もし C ドライブの中の FLEX10K フォルダの naiseki フォルダの中に TTF があるなら、

cd c:\FLEX10K\naiseki(バックスラッシュは円記号の事)

[11]ファイル名:u00simi/eps/max_edit.eps

[12]ファイル名:u00simi/eps/max_com.eps

という様に入力します。そして目的のフォルダに移ったら、

flex10k ***.ttf(*** は TTF ファイルの名前)

これでコンフィグレーションが開始されます。

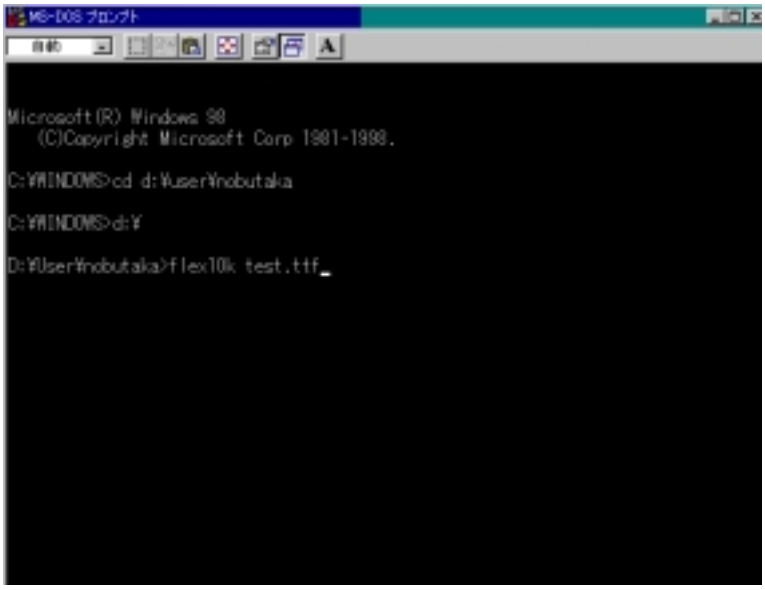


図 2.13: コンフィグレーション[13]

[13]ファイル名:u00simi/eps/dos.eps

付録 C

本研究に置ける C++Builder の使い方

C.1 使用目的

本研究において評価ボードを使用するにあたって、 FPGA を外部から制御する必要がある。通信は松尾 [3] が作製した PPI8255 を 2 個搭載した、 ISA バス専用の通信ボードを用いる (詳しくは [3] 参照)。昨年までは C++ を使用して評価基盤の制御を行って来たが、今年は C++Builder を購入したので、これで制御させる事にする。

C.2 使用方法

C.2.1 ヘッダーファイルの設定

まず C++Builder の自分の作業フォルダーに以下のファイルを入れる。

BASE8255.h
PORT95.dcu
PORT95.hpp
PORT95.OBJ
PORT95.PAS

BASE8255.h とはヘッダファイルであり、この中に基板を制御する為のポートが宣言されている。 PORT95.hpp の中に実際の基板制御関数を宣言しているのでこの 2 つのファイルを読み込まなければ実際に基板を制御出来ない。他のファイルは基板制御関数の実際の動作をパスカル等記述している。

図 3.1: C++BUilder のプロジェクトマネージャー画面[1]

次に C++Builder を起動させて、 C++Builder のメニューのプロジェクトマネージャーを開き図 3.1の様に ***.EXE に PORT95.OBJ を追加する。
最後にメインのソース、 ***.cpp に以下をヘッダーファイルとして追加する。

BASE8255.h
PORT95.hpp

これで C++Builder 上で制御命令文の使用が可能となる。

C.2.2 制御命令文

前節で行った設定で以下の命令が使用出来る

PortWiteByte(ポート名, 書込む数値)
PortReadByte(ポート名)
PortWiteWord(ポート名, 書込む数値)
PortReadWord(ポート名)

Byte は 8bit、 Word は 16bit のことである。

ポート名 ポート名には A(A_H,A_L),B_H,B_L,CTRL_H,CTRL_L がある。詳しくは巻末の PPI8255 の動作表参照。

- A(A_H,A_L)
  A は入出力両方に使用する。 16bit
- B_H
  B_H は評価基盤から PC への入力とする。 8bit

[1]ファイル名:u00simi/eps/C++_face.eps

- B_L
  B_L は評価基盤への PC からの出力とする。 8bit

- CTRL_H
  A,B 両方の High ポートのコントロールワード設定。 8bit

- CTRL_L
  A,B 両方の Low ポートのコントロールワード設定。 8bit

書込む数値　数値は 16 進数で表記する。例えば B_L に “00000010” を出力したい場合は PortWrite-Byte(B_L,0X02) と書く。‘0X’ とは 16 進数という意味である。

C.2.3　整数型と単精度浮動小数点

PC と評価基盤との通信には A ポートを使っての 16bit が最高の幅を持つ伝送路になる。 C++Builder でこの研究に使う主な数値の型には int 型 (整数型) と float 型 (単精度浮動小数点) がありどちらも 32bit である。よって伝送路の幅を通る為には、 16bit ずつ別けて通す必要がある。

- 32bit から 16bit への変換
  C のポインタを使う。
  float data1;
  data1 を float 型で定義
  ddata1= (unsigned short&*)data1;
  data1 に入っているデータがある場所のアドレスを ddata1 に返す。
  recordl1 =*ddata1;
  ddata1 に入っているアドレスが指す場所のデータを record1l に入れる。つまり 32bit の内、下位 16bit を入れる。
  recordh1 =*(ddata1+1);
  (ddata1+1) に入っているアドレスが指す場所のデータを record1l に入れる。つまり 32bit の内、上位 16bit を入れる。

図 3.2: ポインタを使った変換[2]

- 16bit から 32bit への変換

  unsigned short res[1];

  res[0],res[1] の２つの配列を unsigned short 型（16bit) で定義

  res[0]=PortReadWord(P_A);

  res[1]=PortReadWord(P_A);

  2 つの配列に分割した 16bit のデータを入力

  Memo3-Lines-Add("FPGA="+FloatToStr(*(float*)res));

  res に入ったデータを１つにつなぎ出力

- 単精度浮動少数点

  単精度浮動小数点は 32bit である。

図 3.3: 単精度浮動小数点[3]

32bit の内訳は

[2]ファイル名:u00simi/eps/divide.eps

[3]ファイル名:u00simi/eps/float.eps

- 上位 1bit
  1 の時正の数を表し 0 の時負を表す。

- 次の 8bit
  指数部を表す。この 8bit から offset と呼ばれる 8bit の数 '01111111' を引いた物が真の指数部となる。

- 残り 23bit
  仮数部を表す。ここには暗黙の 1 と呼ばれる物が付いており、この 1 に 23bit の内、左から 2^{-1}、 $2^{(-2)}$、 $2^{(-3)}$.....$2^{(-23)}$ と、重みが付いている数値を足して仮数部としている。
  例を挙げると

  '11000000111000000000000000000000'

  の浮動小数点型は

  $$\mathbf{(-1)*2^{(129-offset(127))}*(1+(1*2^{-1})+(1*2^{(-2)}))=-7}$$

  と言うことになる。 0 を表す時は特別で、 32bit 全て 0 にすればいい。これは厳密には 0 ではないが、形式的に 0 になる。

- 整数型整数型は単なる 2 進数表記で表し、負の数は補数で表現する。

付録 D

C++Builder のソースプログラム

本研究において C 言語を使いこなすことも重要な要因の一つである。そこですぐわかる C++Builder[7] という入門書を購入して学んだ。もしこの研究を行う者で C++ を理解したいならこの本を読む事をお勧めする。以下に C++ のソースの例を記す。

```
#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"
#include <stdio.h>
#include"BASE8255.h"
#include"port95.hpp"


#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
  randomize();
}


void __fastcall TForm1::Button1Click(TObject *Sender)
{
 PortWriteWord(P_CTRL_L,0xc0);// インターフェイスボードのコ
ントロール設定
 PortWriteWord(P_CTRL_H,0xc2);// インターフェイスボードのコ
ントロール設定
 PortWriteByte(P_BL,0x01);// 評価基盤への制御信号出力
 PortWriteByte(P_BL,0x00);// 評価基盤への制御信号出力
      for(j=0;j<N;j++)// 単精度浮動小数点のランダム発生 (N は
発生させる個数) のループ
         {
          a=random(100);// 100 を越えない整数をランダムに発生
させる関数

          b[j]=float(a)+float(a)/(float(a)+1);// 整数を浮動小数
点に変換
          data=(unsigned short*)&b[j];//32bit を 16bit に変換
          recordl[j]=*data;// 32bit を 16bit に変換
          recordh[j]=*(data+1);// 32bit を 16bit に変換

         PortWriteWord(P_A,recordl[j]);// 下位 16bit を評価基盤に出力
          PortWriteWord(P_A,recordh[j]);// 上位 16bit を評価基盤に出力
         }
```

```
Memo2->Lines->Add(IntToStr(j)+"="+FloatToStr(b[N-1]));//windows
画面に表示

         PortWriteByte(P_BL,0x01);// 評価基盤への制御信号出力
        PortWriteByte(P_BL,0x00);// 評価基盤への制御信号出力
}

void __fastcall TForm1::Button3Click(TObject *Sender)
{
 Close();// プログラムの終了
}


void __fastcall TForm1::Button2Click(TObject *Sender)
{
 c=0;

 PortWriteWord(P_CTRL_L,0xd0);// インターフェイスボードのコ
ントロール設定
 PortWriteWord(P_CTRL_H,0xd2);// インターフェイスボードのコ
ントロール設定

 unsigned short res[2];
 PortWriteByte(P_BL,0x01);// 評価基盤への制御信号出力
 PortWriteByte(P_BL,0x00);// 評価基盤への制御信号出力
 PortWriteByte(P_BL,0x80);// 評価基盤への制御信号出力
 PortWriteByte(P_BL,0x00);// 評価基盤への制御信号出力


 PortWriteByte(P_BL,0x02);// 評価基盤への制御信号出力
 PortWriteByte(P_BL,0x00);// 評価基盤への制御信号出力
 res[0]=PortReadWord(P_A);// 評価基盤の結果を PC に出力 (16bit)

 PortWriteByte(P_BL,0x02);// 評価基盤への制御信号出力
 PortWriteByte(P_BL,0x00);// 評価基盤への制御信号出力
 res[1]=PortReadWord(P_A);// 評価基盤の結果を PC に出力 (16bit)
 Memo3->Lines->Add("FPGA="+FloatToStr(*(float*)res));// 結果
を 32bit に変換して windown 画面に表示
 Memo3->Lines->Add("    PC="+FloatToStr(c));//windown 画面に表示

}
```

付録 E

pc Anywhere の使用法

E.1 使用目的

PC と評価基盤との通信には松尾 [3] が作製した ISA バス専用のインターフェイスを用いているが、最近の PC には ISA バスが搭載されている PC は皆無に等しい。そこで ISA バス搭載 PC を評価基盤制御 PC として独立させ、その PC を外部の PC から通信で制御して、コンフィグレーションを行うことにする。

E.1.1 遠隔操作による利点

遠隔操作を行うことによる利点は、コンフィグレーションを複数のマシーンで行うことができる点である。これはホストになった PC に複数のユーザーが遠隔操作を同時に行うことができる。これにより複数の人が評価基盤を共有できるということになった。しかし同時にそこにアクセスすることも可能だが、評価基盤は同時には使用できないため、注意が必要である。

E.1.2 使用方法

評価基盤をコンフィグレーションできる ISA　バス搭載 PC を Pc anywhere の設定画面でネットワークのホストをダブルクリックしてホストに設定する (図 5.1)。またホストのアイコンを右クリックしプロパティの中の設定で起動時にホストに設定する事も可能である。

図 5.1: ホスト設定[1]

次にリモート制御する PC において図 5.1のメニューのツールにあるネットワークのオプションを開く (図 5.2)。そこでネットワーク上にある探索するホストのネットワーク上の名前を設定する。ここでは NOBUTAKA、SUGIYAMA というホストを探すと言うことである。ここにホストに設定する PC の名前を入れて図 5.1のリモート制御を選択して NETWORK をクリックすれば遠隔操作が可能になる。

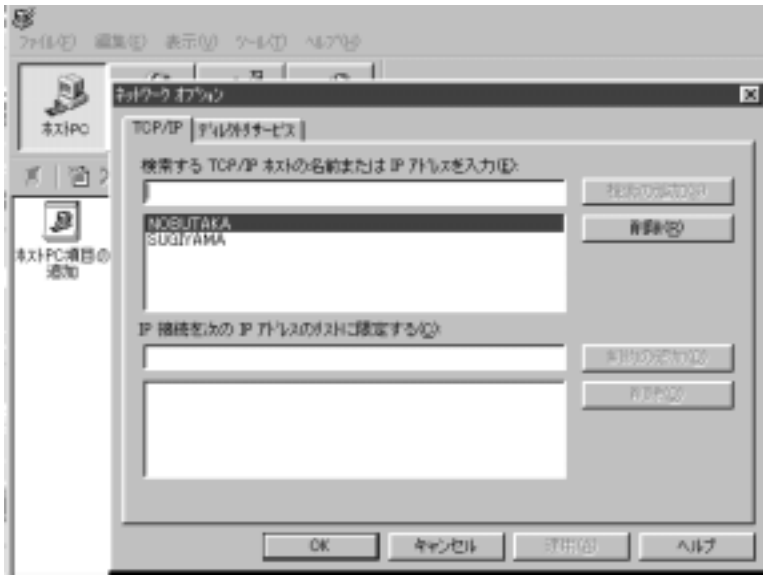


図 5.2: ホスト設定[2]

[1]ファイル名:u00simi/eps/anyware_face.eps

[2]ファイル名:u00simi/eps/ip_settei.eps

付録 F

他のボード使用法

VHDL を学ぶにあたって練習用に使ったボードの使用方、サンプルプログラムをここに表す。

F.1 cq ボード

F.1.1 使用方法

ここでは CQ ボードの使い方について述べる。

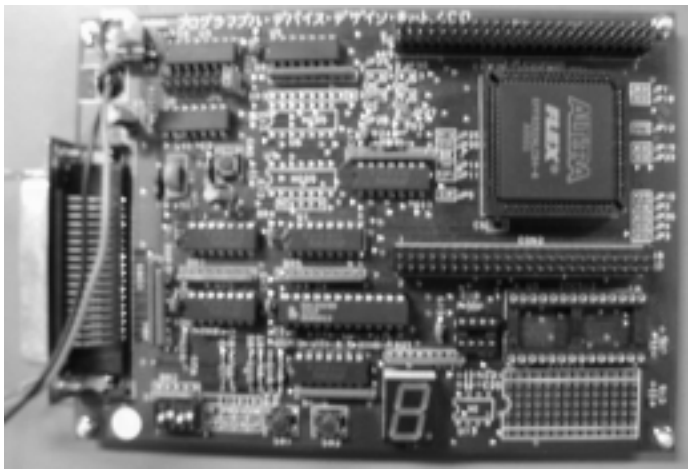


図 6.1: CQ ボード[1]

この CQ ボードは WINDOWS 95 用である。まず、図 6.1の CQ ボードと WINDOWS95 搭載の理論グループ専用の PC を CQ ボード上の FPGA にコンフィグレーションするためプリンターケーブルでつなげる。そしてこの CQ ボードは DC5V 電源を使用する為、電圧源をつなげる。そして、あらかじめ作っておいた VHDL ソースを MAX+plus2 でコンパイルする。コンパイル法は沼 [6] の修士論文の付録に詳しく書いてある。そして DOS プロンプトを開き、目的のソースがあると

[1] ファイル名:u00simi/eps/cq_face.eps

ころに移り、その後 C: flexcq ***.ttf と ttf ファイルを指定し、コンフィグレーションを開始させる。

F.1.2 ピン配線

CQ ボードのピン配線を 6.1に記す。

信号名	用途	方向	ピン数	ピン番号
CLK	クロック供給	in	1	50
SW1	SW1 の信号 負論理が on	in	1	27
SW2	SW2 の信号 負論理が on	in	1	19
LED	LED ランプの表示 正論理が点灯	out	8	a:15,b:16,c:18,d:46, e:35, f:37g:39,h:40

表 6.1: CQ ボードのピン配線

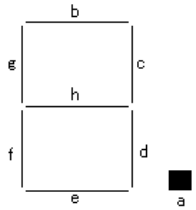


図 6.2: LED ランプの配置図[2]

F.1.3 サンプルプログラム

以下にサンプルプログラムを添付させる。このプログラムはコンフィグレーションすると、 LED が 1 ずつカウントしていき、 SW を押すと、偶数の数をカウントしていくプログラムである。

```
----------------------------------------
-- 可変速 10 進アップカウンタ (FLEX8000)
-- 2000/03/15
-- nobutaka@tube.ee.uec.ac.jp
----------------------------------------

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

library metamor;
```

[2]ファイル名:u00simi/eps/cq_LED.eps

```
use metamor.attributes.all;

entity countup3 is
port (
SW_1,SW_2,CLK : in std_logic;
CARRY : out std_logic;
LED : out std_logic_vector(7 downto 0)
     );

attribute pinnum of LED   : signal is  "15,16,18,46,35,37,39,40";
attribute pinnum of SW_1  : signal is  "27";
attribute pinnum of SW_2  : signal is  "19";
attribute pinnum of CLK   : signal is  "50";
attribute pinnum of CARRY : signal is  "45";

end countup3 ;
architecture RTL of countup3 is
signal CLK_2   : std_logic_vector(20 downto 0);
signal DCLK    : std_logic;
signal CNT     : std_logic_vector(3 downto 0);
signal CNT_1   : std_logic_vector(3 downto 0):="0000";
signal CNT_2   : std_logic_vector(3 downto 0):="0000";
signal CRY     : std_logic;
signal ST      : std_logic;

begin
process begin
  wait until CLK'event and CLK = '1';-- もしクロックが'1' に変化したら
  CLK_2 <= CLK_2+1;-- クロック 2 に'1' を加えなさい
end process;

DCLK <= CLK_2(20);-- クロックを分周ここでカウントの表示速度を
決定 CLK_2(X) の X の値を小さくすると表示速度が早くなる。

process begin
  wait until DCLK'event and DCLK = '1';-- もし分周したクロックが'1' に変化したら
  case SW_1 is--SW1 が off の時
    when '1' =>--1 の時
      if CNT_1 = "1001" then-- もし CNT1 が 9 になったら
       CNT_1 <= "0000";--CNT1 を 0 に戻してください
       CRY <= not CRY;
      else-- それ以外は
       CNT_1 <= CNT_1 + "0001";--CNT1 に 1 を足して下さい
      end if ;
     when '0' =>--SW1 が on の時
       if CNT_2 = "1000" then-- もし CNT2 が 8 になったら
        CNT_2 <= "0000";--CNT2 を 0 に戻してください
        CRY <= not CRY;
       else-- それ以外は
        CNT_2 <= CNT_2 + "0010";--CNT2 に 1 を足して下さい
       end if ;
      when others => null;
   end case;
  end process;

  process (SW_1) begin
   if ( SW_1 = '1' ) then-- もし SW1 が off なら
      ST <= '0';--ST に'0' を入れる
   elsif (SW_1 = '0' ) then-- もし SW1 が on なら
      ST <= '1';--ST に'1' を入れる
   end if;
  end process ;
    CNT <= CNT_1 when ST = '0' else--CNT を SW によって選択
           CNT_2;

   process ( CNT ) begin
     case CNT is--LED を光らせる  '0' で点灯  '1' で消灯
         when "0000" => LED <= "01111110";--0 を表示
         when "0001" => LED <= "00110000";--1 を表示
```

```
        when "0010" => LED <= "01101101";--2 を表示
        when "0011" => LED <= "01111001";--3 を表示
        when "0100" => LED <= "00110011";--4 を表示
        when "0101" => LED <= "01011011";--5 を表示
        when "0110" => LED <= "00011111";--6 を表示
        when "0111" => LED <= "01110010";--7 を表示
        when "1000" => LED <= "01111111";--8 を表示
        when "1001" => LED <= "01111011";--9 を表示
        when others => LED <= "XXXXXXXX";
    end case;
   end process;
  end RTL;
```

F.2 UP1 ボード

F.2.1 UP1 ボードの使用方法

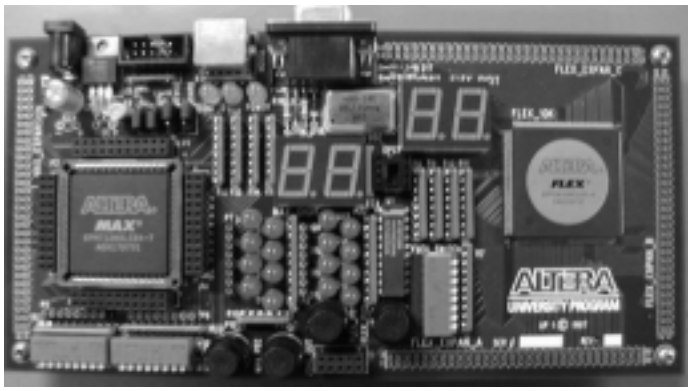


図 6.3: UP1 ボード[3]

使用する FPGA は UP1 ボード上の FPGA である、 FLEX10K20RC 240-4 を用いる。コンパイルをする時にデバイスを間違わない用注意が必要である。
MAX+plus2 コンパイル終了後図 6.4の画面の MAX+plus2 を選択して図 6.5のところで、 Programmer を選択する。

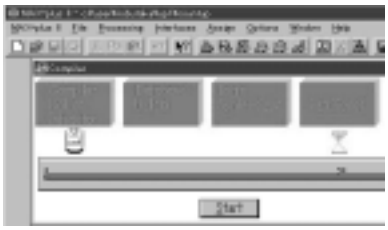


図 6.4: コンパイル終了[4]

図 6.5: MAX+plus2 選択後[5]

[3]ファイル名:u00simi/eps/UP1_face.eps

Programmer がでているところで、 Option の Hardware setup を選択して Hardware Type を ByteBlaster に変更する図 6.6。

図 6.6: ハードウェア選択[6]

その後、図 6.4の JTAG を選択。すると図 6.7でてくるので、 Multi-Device JTAG Chain をチェックする。そして JTAG の中の Multi-Device JTAG Chain Setup を選択する。

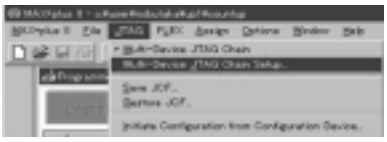


図 6.7: JTAG[7]

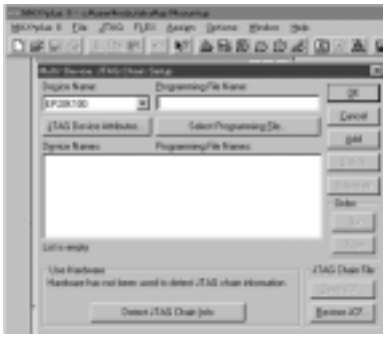
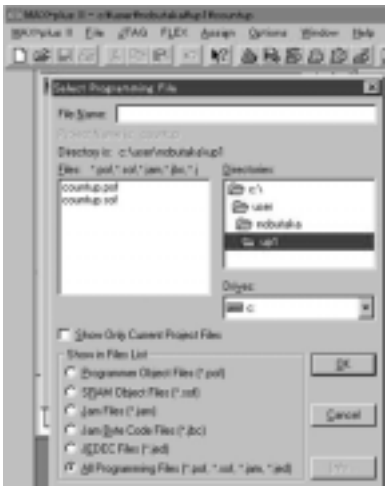


図 6.8: chain setup[8]

図 6.8でまず、 Delete All を押して、 Devaice Name を全て消す。そして Select Programming Name を押す。



[4]ファイル名:u00simi/eps/Image1.eps

[5]ファイル名:u00simi/eps/Image2.eps

[6]ファイル名:u00simi/eps/hardware.eps

[7]ファイル名:u00simi/eps/Image3.eps

[8]ファイル名:u00simi/eps/Image4.eps

図 6.9: ファイル選択[9]

すると。図 6.9がでてきて、読み込むファイルを聞いて来るので、 XXXXX.sof ファイルを選択する。後は図 6.8の ADD を押して、 Programmer の Configuration を押す。

F.2.2 ピン配線

UP1 ボードに割り当てられている FPGA のピン配線を??に記す。

信号名	用途	方向	ピン数	ピン番号
CLK	クロック供給	in	1	91
SW1	FLEX_PB1 の信号 負論理が on	in	1	28
SW2	FLEX_PB2 の信号 負論理が on	in	1	29
F_SW	FLEX_SWITCH の信号 上に上げると on	in	8	41,40,39,38,36,35,34,33
LED_1	LED ランプの表示, 負論理が点灯	out	8	a:6,b:7,c:8,d:9,e:11, f:12, g:13,DecimalPoint:14
LED_2	LED ランプの表示, 負論理が点灯	out	8	a:17,b:18,c:19,d:20,e:21, f:23, g:24,DecimalPoint:25

表 6.2: UP1 ボードのピン配線

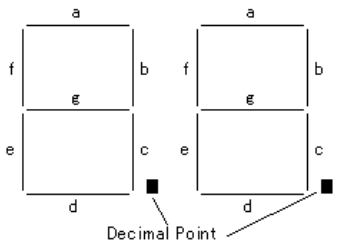


図 6.10: LED の配置図[10]

[9]ファイル名:u00simi/eps/Image5.eps

[10]ファイル名:u00simi/eps/up1_LED.eps

F.2.3 サンプルプログラム

UP1 ボード上にある 2 つの LED を 00 から 99 まで 1 ずつ光らせるプログラムである。

```
----------------------------------------
-- 可変速 2 桁 10 進アップカウンタ (UP1 ボード)
-- 2000/03/18
-- nobutaka@tube.ee.uec.ac.jp
----------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

library metamor ;
use metamor.attributes.all;
entity countup is
 port (
         SW_1,SW_2,CLK : in std_logic;
         LED_1,LED_2 : out std_logic_vector ( 7 downto 0 )
        ) ;

 attribute pinnum of LED_1  : signal is "6,7,8,9,11,12,13,14";
 attribute pinnum of LED_2  : signal is "17,18,19,20,21,23,24,25";
 attribute pinnum of CLK     : signal is "91";
 attribute pinnum of SW_1   : signal is "28";
 attribute pinnum of SW_2   : signal is "29";

end countup;

architecture RTL of countup is

 signal CLK_2 : std_logic_vector ( 20 downto 0 );
 signal DCLK  : std_logic;
 signal CNT_1 : std_logic_vector (3 downto 0 ) := "0000";
 signal CNT_2 : std_logic_vector (3 downto 0 ) := "0000";
begin
 process begin
   wait until CLK'event and CLK = '1';-- クロックが'1' になった瞬間
   CLK_2 <= CLK_2 + '1';-- クロック 2 に'1' を足す
 end process;

 DCLK <= CLK_2(20);-- クロックの分周  表示速度を決定

process begin
  wait until DCLK'event and DCLK = '1' ;-- 分周したクロックが'1' になったら
  if CNT_1 = "1001" and CNT_2 = "1001" then-- もし CNT_1 が 9 かつ CNT_2 が 9 つまり 99 になったら
     CNT_1 <= "0000";--0 に戻す
     CNT_2 <= "0000";--0 に戻す
  elsif CNT_2 = "1001" then-- もし CNT_2 が 9 つまり 1 の位が 9 になったら
     CNT_1 <= CNT_1 + "0001";--CNT_2 に 1 追加つまり 10 の位に 1 つ繰り上げ
     CNT_2 <= "0000";--CNT_2 を 0 にするつまり 1 の位を 0 の戻す
  else-- それ以外は
     CNT_2 <= CNT_2 + "0001";--CNT_2 に 1 を足すつまり 1 ずつカウントしていく
  end if;
 end process;

 process  begin  --'0' が点灯  '1' が消灯
       case CNT_1 is-- 1 の位が
           when "0000" => LED_1 <= "00000011";--0 を表示する
           when "0001" => LED_1 <= "10011111";--1 を表示する
           when "0010" => LED_1 <= "00100101";--2 を表示する
           when "0011" => LED_1 <= "00001101";--3 を表示する
           when "0100" => LED_1 <= "10011001";--4 を表示する
           when "0101" => LED_1 <= "01001001";--5 を表示する
           when "0110" => LED_1 <= "01000001";--6 を表示する
           when "0111" => LED_1 <= "00011011";--7 を表示する
           when "1000" => LED_1 <= "00000001";--8 を表示する
           when "1001" => LED_1 <= "00001001";--9 を表示する
           when others  => LED_1 <= "XXXXXXXX";
         end case;
```

```
      case CNT_2 is-- １０の位が
           when "0000" => LED_2 <= "00000011";--0 を表示する
           when "0001" => LED_2 <= "10011111";--1 を表示する
           when "0010" => LED_2 <= "00100101";--2 を表示する
           when "0011" => LED_2 <= "00001101";--3 を表示する
           when "0100" => LED_2 <= "10011001";--4 を表示する
           when "0101" => LED_2 <= "01001001";--5 を表示する
           when "0110" => LED_2 <= "01000001";--6 を表示する
           when "0111" => LED_2 <= "00011011";--7 を表示する
           when "1000" => LED_2 <= "00000001";--8 を表示する
           when "1001" => LED_2 <= "00001001";--9 を表示する
           when others  => LED_2 <= "XXXXXXXX";
        end case;
 end process;
end RTL;
```

付録: CD-ROM