SING 制御プロセッサ、GRAPE 版について

牧野淳一郎

May 4, 2006

Abstract

本稿は SING の制御プロセッサのミニマル版である CP-GRAPE について、その構成と動作記述を与える。

Contents

1 変更履歴

1.1 2004/12/31

書き始める。もとは「VPM ボードの概念」。制御プロセッサについてはこちらの記述がオーバーライドする。

1.2 2005/1/20

第 0.1 版。まだ ホスト API はプロセッサ仕様と完全にはコンシステントでない。
第 2 節。

1.3 2005/6/1

メモリインターフェースの変更に対応して色々見直し。

1.4 2005/6/5

メモリインターフェース変更のためハードウェア全面的に変更
ソフトウェア変更はまだ

1.5 2005/6/6

ODP-HIO に簡易版仕様を追加した。

1.6 2005/6/7

API 変更
新規関数

```
SING_EM_alloc
SING_EM_free
SING_EM_areas
SING_EM_send
SING_EM_BM_transfer
SING_BM_alloc
```

無くなる関数

```
SING_EM_write
SING_EM_read
SING_EM_BM_transfer_register
```

実際のハードウェアでは、 LM への転送も全て EM を通すことになるがこの機能は未実装。また、現在は BM 上でワークエリアと変数領域の区別はないので、RRN 実行や LM 転送実行の後 BM 上の変数内容は保証されない。

1.7 2005/6/17

EM-IDP 転送にの概要が古いままだったので修正。

1.8 2005/6/19

内積計算モデル対応の新しい仕様への変更作業始める。

1.9 2005/7/9

```
SING_iregister
SING_execute
```

仕様変更

iregister: itrans をなくした。 命令列の中自体に idp が書かれているため。

execute: areaid をなくした。これも命令列の中自体に書いてある。

`final_id` を追加

1.10 2005/8/2

ODP-HIO 転送回路の入力に UN を追加。これに対応してコード出力回路からの出力にも追加。

2 制御プロセッサ

SING を実際に使うためには、

- メモリインターフェース
- ホスト計算機インターフェース
- 命令供給

を行う外付けの何かが必要である。ここではそれを制御プロセッサと呼ぶ。

図 3 にここで考える制御プロセッサのモデルを示す。基本的にはホスト (H)、ホストインターフェース (HI)、制御プロセッサ (CP)、外部メモリ (EM) と SING チップからなる。

実際の実装としては、ボード 1 枚に複数の SING チップを実装することになる。このため、 CP, EM, SING からなるブロック (計算ブロック) が複数ホストインターフェースに接続されることになる。通信、計算方式によっては、 CP の機能の一部が HI 側に移動する。これについては別に議論することにし、この文書では計算ブロックが一つの場合について述べる。

3 制御プロセッサの機能

制御プロセッサの基本的な仕事は以下の通りである。

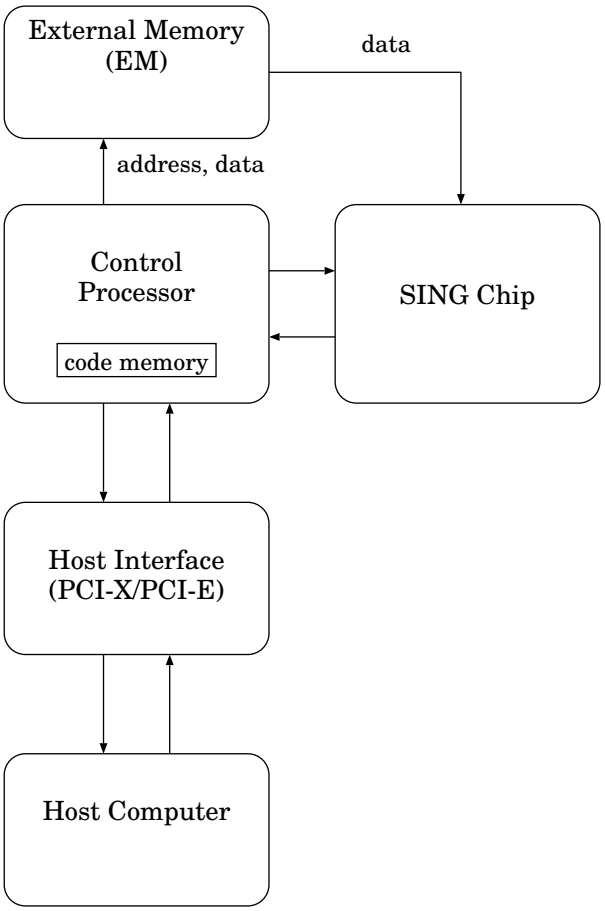


Figure 1: 制御プロセッサのモデル

1. ホストからのデータを外部メモリに格納する。
2. ホストからのデータ、または外部メモリのデータを放送メモリに格納する。
3. ISP, IDP から SING にデータを送り込む。
4. SING の ODP からのデータを受け取り、必要ならば加工してホストに送り返す。

これらは、ややこしい具合に並列に動く必要がある。一般の場合にどうなるかは良くわからないので、以下単純な重力計算等の場合について考える。で、それに対して最低限必要な関数を準備することにする。他の演算、特に線形計算については別途検討する。

3.1 重力計算の時の制御プロセッサの動作

例えば単純な重力計算なら、実際の計算の前にまず外部メモリ (以下 EM) に j 粒子を書く。
一般に N 個の自由度 (粒子) の間の相互作用計算として、

$$f_i = \sum_j f(q_i, q_j) \tag{1}$$

みたいなものを考える。ここで f_i は i 番目の粒子の受ける相互作用 (加速度とかポテンシャルとかそういうもの)、関数 $f(q_1, q_2)$ は粒子 2 から粒子 1 への相互作用、 q_i は粒子 i の物理データ (位置、質量等) である。
一セットの粒子への相互作用計算はまず i 粒子を書き、j 粒子の数だけ以下の作業

1. j 粒子を外部メモリから放送メモリに配分する
2. 放送メモリからローカルメモリに j 粒子をコピーし、重力計算をする

を、並列に行う必要がある。で、計算が終わったら結果がでてくる。
もうちょっと作業を分解すると、
j 粒子については

1. 外部メモリにデータ転送する
2. 外部メモリから放送メモリに、多分データ変換しながらデータ転送する

の 2 つの機能が必要である。上は計算前に、下は計算中に行う。
i 粒子については、以下の作業

1. BM に、データ変換しながら書き込む
2. 書いたものを PE に振り分ける

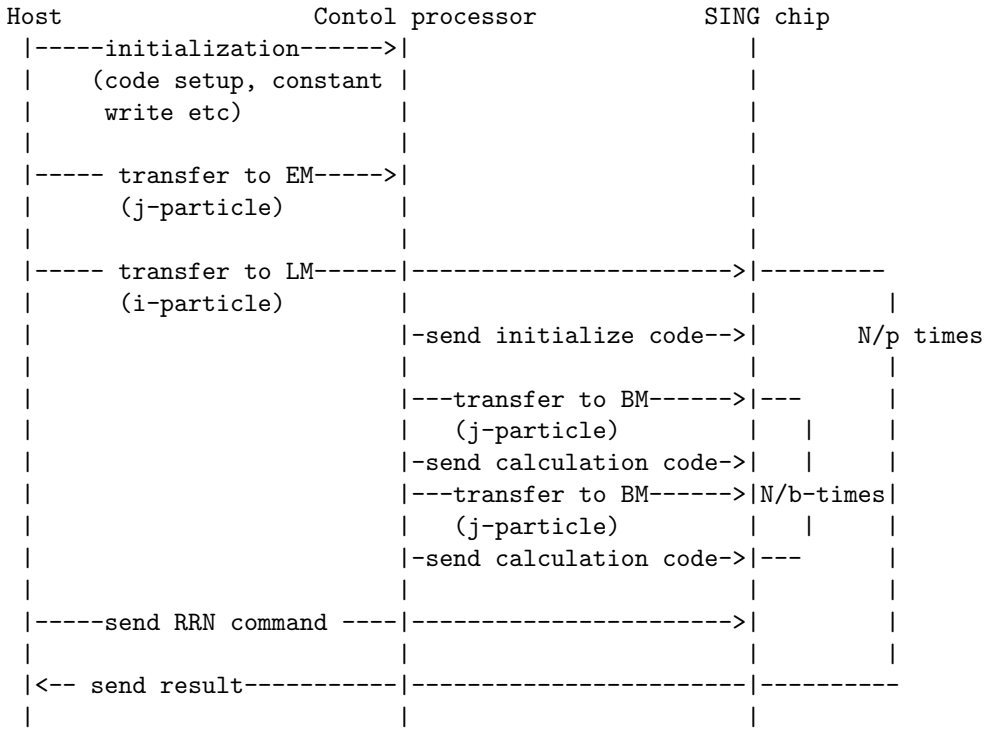
を PE の数だけする必要がある。
力の計算そのものについては、上に書いたように

1. j 粒子を外部メモリから放送メモリに配分する
2. 放送メモリからローカルメモリに j 粒子をコピーし、重力計算をする

を並列に行う。
結果の回収は

1. RRN コマンドパケットを発行する
2. でてきたデータを DMA かなにかで回収する
3. なんかデータ変換する

という感じ。
従って、基本的にこれらの機能に対応する関数が必要。
整理しておくと、こんな感じ。上から時間が流れるとして。ここで N は粒子の数、p はプロセッサの数 (放送ブロック内物理数値 × ベクトル長)、 b は放送ブロックの数である。簡単のために粒子数は放送ブロックの数でもプロセッサの数でもわり切れるとした。そうでない場合の面倒はアプリケーションホストのプログラムのほうで見る。



これから、 CP は以下の機能を実行することがわかる。

1. ホストから SING の実行コードを受け取り、自分の内部メモリに格納する。
2. ホストから定数データを受け取り、それを SING ローカルメモリに格納する。
3. ホストから j 粒子データを受け取り、 EM に格納する。
4. ホストから i 粒子データを受け取り、 SING ローカルメモリに格納する。
5. 計算本体の前の初期化コードを SING に送る。

6. EM の j 粒子データを SING に送る。
7. 計算本体コードを SING に送る。
8. 結果回数のための RRN コマンドを SING に送る。
9. SING から結果を受け取り、それをホストに転送する。

実際には、これらの機能をそれぞれ別個に実装するわけではなく、より単純な機能の組合せで実現する。以下に、単純な機能がどのようなものかについてまとめる。

3.2 CP の I/O 定義

機能の前に、 CP の I/O を定義する。

CP の IO は、すでに述べたように

1. ホストインターフェースとの I/O
2. SING との I/O
3. メモリ (EM) I/O

の 3 つである。以下、これらのそれぞれについて仕様を規定する。但し、電気的な仕様はここでは規定しない。また、これはチップ間接続の仕様であるので、チップ内での接続仕様はこれに従う必要はない。

3.2.1 ホストインターフェース I/O (HIIO)

ホストインターフェース I/O は、独立の入力ポートと出力ポートを持つ。入出力はクロック同期の単純なポートであり、それぞれ以下の信号を持つ。

信号名	本数	説明
CLK	1	クロック
DATA	64	データ
DEN	1	データイネーブル線
AEN	1	アドレスイネーブル線
RESET	1	リセット線

CLK は基本的に常時供給され、DATA, DEN, AEN はどれもクロックの立ち上がりでサンプルされる。RESET も同期サンプルとする。アドレス、データはマルチプレクスされ、 AEN, DEN が同時に出てはいけない。なお、DEN, AEN, RESET は全て low active である。

アドレスがでて以降、 DEN active と同時のデータが有効データと考えられる。なお、アドレスは 64 ビットワードアドレスであり、 32 ビットワードや byte を直接アクセスする手段はインターフェースレベルでは提供されない。また、連続しない場合でも、ある DEN の次の DEN では、その間に AEN やリセットが入らない限りアドレスは 1 インクリメントされる。

RESET がアサートされるとアドレスは 0 に初期化される。従って、 RESET 後アドレスアサート (アドレスフェーズ) なしに DEN をアサートすることはエラーではない。

また、 RESET によらずとも内部的に、あるいは別のポートからの操作でアドレスが初期化されることがある。

ホストインターフェースへの出力の、もっとも単純な実装では AEN, RESET の両方を省略し、 CLK, DATA, DEN だけという実現も可能である。また、CLK については、 CP, HIIO に共通クロックが外部から供給されて

いて、入出力ポートにはクロック線が存在しないという実装も可能である。さらに、FPGA チップ内の場合など、信号線数に余裕がある場合にはアドレス・データノンマルチプレクスな実装も許す。このため、データを受ける側は連続サイクルでのランダムアクセスに対応できる必要があるものとする。この場合には、ポートは

信号名	本数	説明
CLK	1	クロック
DATA	64	データ
ADDR	(32)	アドレス
DEN	1	データイネーブル線
RESET	1	リセット線

となる。なお、この部分の動作クロックは、 SING 動作クロックを 1, 2, 4, 8 等の単純な整数で分周したものである。

3.2.2 SING I/O

SING I/O はチップの規定に従う。基本的には

- ISP 出力。10x ビット、1/4 クロックで 4 サイクル毎に出力
- IDP 出力。72 ビット、データとイネーブルからなる
- ODP 入力。72 ビット、1/4 クロックで 4 サイクル毎に出力

である。しかし、 IDP データは EM から直接出るため、制御プロセッサから出るのはクロックとストローブだけとなる。

以下、信号名定義を与える。

3.2.2.1 ISP

ISCLK[0:3]	出力ソースクロック
ISDATA[0:31]	出力データ
ISPPHASE[0:3]	フェーズ信号

ISDATA を 4 クロック分くっつけた後のものを IS[0:127] と呼ぶ。これは 9 ビット毎に 1 ビットパリティを持つ。パリティは 10 ビット目、つまり IS[10n] になる。

4 サイクルのデータは、下から出るものとする。つまり、

```
Cycle 0  IS[0:31]
Cycle 1  IS[32:63]
Cycle 2  IS[64:95]
Cycle 3  IS[96:127]
```

の順番ででる。

3.2.2.2 IDP

データ、クロックはメモリから出るので、制御プロセッサから出て SING チップに入るのは

```
IDPPHASE[0:8] フェーズ信号
DS0           立ち上がりクロックデータストローブ
DS1           立ち下がりクロックデータストローブ
DSCLK         データストローブクロック
```

となる。DS, DSCLK は IDPPHASE から見て位相ずれがないことを保証するものとする。

メモリからは

```
IDCLK[0:8]    出力ソースクロック
IDDATA[0:79]  出力データ
```

が入る。80 ビットは 16x4+8 または 36x2+8 で構成する。8 ビット品の代わりに 9 ビット品を使う場合には単に 1 ビット使わないことにする。

IDPPHASE がメモリクロックとは独立にでてしまうので、複数あることに意味がないように見えるかもしれないが、これはそうではなくてメモリからはのクロックと FPGA からのフェーズ信号の位相を調整可能にすることでフェーズ信号が有効なタイムウィンドウを大きくする。

3.2.2.3 ODP

```
ODCLK[0:2]    入力ソースクロック
ODDATA[0:40]  入力データ
```

ODDATA を 2 クロック分くっつけた後のものを OD[0:81] と呼ぶ。これは 9 ビット毎に 1 ビットパリティを持つ。パリティは 10 ビット目、つまり OD[10n] になる。

OD[81] はストローブである。 2 サイクルのデータは、下から出るものとする。つまり、

```
Cycle 0  OD[0:40]
Cycle 1  OD[41:81]
```

の順番ででる。

3.2.3 メモリ I/O

論理的には

```
Signal       I/O  説明
DATAO[0:79]  Out  データ出力
ADDR[0:31]   Out  アドレス
WEN          1    Out  ライトイネーブル
```

データパリティはつける。

QDR メモリのために必要なピンは以下の通り。以下、メモリは 36x2+9 の場合を想定する。

データ出力

```
D0[0:35]          書き込みデータ出力 0
D1[0:35]          書き込みデータ出力 1
D2[0:7]           書き込みデータ出力 2
```

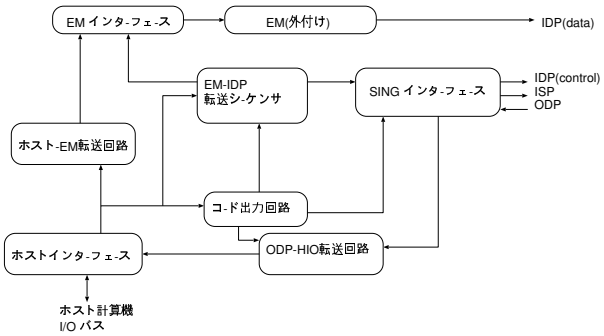


Figure 2: 制御プロセッサのブロック図

制御線。全てメモリチップ毎に独立にあるものとする。

ADDR[0:22]	アドレス出力。
MEMK	メモリ書き込みクロック正相
MEMKB	メモリ書き込みクロック逆相
WPS	書き込みバースト開始信号
BWS[0:3]	バイトセレクト。実際にはバースト内でのワード指定に使う
RPS	読み出しバースト指定。これはメモリクロック立ち上がり毎に反転させる
MEMC	メモリ読み出しクロック正相
MEMCB	メモリ読み出しクロック逆相

3.3 CP の構成

図 2 に CP のブロック図を示す。

CP は以下のユニットからなる。

- EM-IDP 転送シーケンサ
- ホスト-EM 転送回路
- コード出力回路
- ODP-HIO 転送回路
- EM インターフェース

この他に

- ホストインターフェース
- SING チップインターフェース

がある。ホストインターフェースはここでは定義しない。 SING インターフェースはここではほぼ素通しである。これの基本的な機能は物理的な信号レベルの変換や、シリアル/パラレル変換だけで、プロトコルレベルの変換はおこらない。

以下、順に入出力と機能を記述する。

3.3.1 EM-IDP 転送シーケンサ

3.3.1.1 概要

EM-IDP 転送シーケンサは、指定されたコマンドに従って IDP パケットヘッダを組み立て、それと EM からのデータによって IDP パケットがでるように EM を制御する。コマンドは以下の 2 つのケースがある。

- ホストインターフェースからくる
- コード出力回路からくる

コマンドを受け取ると、後は自動的に IDP パケットを生成する。

3.3.1.2 入出力

```
HDI      [0:63]   In    ホストからのデータ入力
DEN               In    ホストからのデータライトイネーブル
AEN               In    アドレスイネーブル
START             In    転送起動
IDPID    [0:5]    In    IDP 転送シーケンス
RRNSTART          In    RRN コマンドスタート
RRNDATA  [0:41?]  In    RRN コマンド
BMSADR   [0:11]   In    BM 開始アドレス

BUSY              OUT   転送動作中アサートされる
EMADDR [0:31]     Out   EM アドレス
EMDATA0[0:71]     Out   EM への書き込みデータ 0
EMWRITE0          Out   EM への書き込みストローブ 0
EMDATA1[0:71]     Out   EM への書き込みデータ 1
EMWRITE1          Out   EM への書き込みストローブ 1
IDPEN             Out   IDP ストローブ
RSTR              Out   最上位アドレスデータ復帰命令
```

3.3.1.3 機能

EM-IDP 転送シーケンサはホストインターフェースから書き込み可能な以下のレジスタを持つ

```
Address          Name     Function
0x0[x]0,4,..     EMSADR   EM 読み出し開始アドレス
0x0[x]1,5,..     EMIADR0  EM 読み出し増分アドレス ループ間
0x0[x]2,6,..     EMIADR1  EM 読み出し増分アドレス BM 間
0x0[x]3,7,..     COMM     コマンドワード
0x80             CCLEAR   ループカウンタクリア
```

x は 0 から 7 までとし、これにより 32 個の IDP シーケンスを格納する。

コマンドワードは以下のサブフィールドを持つ

```
0:11       BMSADR BM 開始アドレス
12:19      NBM    BM の数
20:31      LEN    転送シーケンス語数
```

32:35	MODE	転送モード
35:62	未使用	
63	START	転送開始

START の立ち下がりまたは コマンドワードレジスタへの START フィールドが 1 の書き込みで転送が起動される。START の立ち下がりでは、パラメータは IDPID によって指定されるレジスタ 4 ワードからとられる。コマンドワードレジスタの START フィールドによる起動ではそのワードが含まれる 4 ワードからとられる。

EM の実際の開始アドレスは、初期値が EMSADR であり、2 回目以降は起動の度に EMIADR0 だけ増える。但し、CCLEAR に書き込みがあると初期値に戻る。

START フィールドによる起動の場合には常に EMSADR からである。また、 コマンドワードに書き込みが起こると そのレジスタ 4 ワードに対応するアドレスカウンタは初期値に戻る。

転送は以下のように進む。

1. IDP パケットを組み立てて、 EM の最上位アドレスに書き込む。
2. EM からパケットヘッダを出力させる。
3. 4 以下を モード 2 なら NBM で指定された回数繰り返す。そうでなければ 1 度だけ実行。
4. EM へのアドレスカウンタを EMSADR(に EMIADR0 の増分を加えたもの) で初期化
5. LEN だけの語数 EM にアドレスを送り、データがでるタイミングに合わせて IDPEN をアサートする。
6. 指定された語数の処理が終わったら、まだ BM が残っていたら 4 に戻る。この時に、アドレスカウンタ は EMIADR だけ増やす。
7. 途中で ホスト-EM 転送回路に最上位ビット復帰信号を送る。

転送モードは以下の通り

0	BM 一つへの書き込み。NBM は書く BM の ID。
1	BM への放送モード。 NBM は無視。
2	複数 BM への転送モード。 NBM は繰り返し回数 (0 は全 BB の数)

RRNSTART がアサートされた時には、EMSADR、 EMIADR、 BMSADR を MSB からまとめたものが RRN データとして、IDP ヘッダと共に EM に書き込まれ、これらだけが EM から出力される。

3.3.2 ホスト-EM 転送回路

3.3.2.1 概要

ホスト-EM 転送回路は、ホストから送られてきたデータを変換テーブルによって変換しながら EM に書く。

3.3.2.2 入出力

HDI	[0:63]	In	データ入力
DEN		In	データライトイネーブル
AEN		In	アドレスイネーブル
RSTR		In	最上位アドレスデータ復帰命令
EMADDR	[0:31]	Out	EM アドレス
EMD	[0:71]	Out	EM データ
WEN		Out	ライトイネーブル

3.3.2.3 機能

基本的には AEN がアサートされた時にアドレスを記憶し、その後 DEN がくる度にそのデータをメモリに書き込み、書き込みアドレスをインクリメントする。但し、 EM への書き込みの時には以下に説明するような変換を行う。

ホスト-EM 転送回路のアドレスマップは以下の通り。

```
Address    Name   Function
0x0        CS     変換テーブル開始アドレス
0x400-7FF  CONV   変換指定
0x8000000000000000-  EM データエリア
```

CONV は 8 ビットのフィールドであり、上位、下位の 4 ビットは下のどれかの値を取る。

```
0 flt64to72   64bit 浮動小数点 ->  72bit 浮動小数点
1 flt64to36   64bit 浮動小数点 ->  36bit 浮動小数点 この時はつめる
2 flt32to36   32bit 浮動小数点 ->  36bit 浮動小数点
3 fix64to72l  64bit 固定小数点 ->  72bit 固定小数点 下に拡張
4 fix64to72rs 64bit 固定小数点 ->  72bit 固定小数点 上に拡張 符号あり
5 fix64to72ru 64bit 固定小数点 ->  72bit 固定小数点 上に拡張 符号なし
6 fix64to36   64bit 固定小数点 ->  36bit 固定小数点 上をカット
7 fix32to36l  32bit 固定小数点 ->  36bit 固定小数点 下に拡張
8 fix32to36rs 32bit 固定小数点 ->  36bit 固定小数点 上に拡張 符号なし
9 fix32to36ru 32bit 固定小数点 ->  36bit 固定小数点 上に拡張 符号あり
```

但し、上位フィールドが以下の値

```
flt64to72
flt64to36
fix64to72l
fix64to72rs
fix64to72ru
fix64to36
```

であった場合には、下位フィールドの値は無視される。上位、下位がいずれも

```
2 flt32to36
7 fix32to36l
8 fix32to36rs
9 fix32to36ru
```

のどれかであった場合には、 64 ビットの入力データの上位、下位 32 ビットがそれぞれ指定された変換を受けて、72 ビットデータになる。EM への書き込みは以下のようになる。

1. AEN できたアドレスが EM エリアであった時点で CONV 読み出しアドレスカウンタが CS レジスタの値で初期化される。 EM 書き込みアドレスカウンタは AEN で来たアドレスの値で初期化される。
2. 次にデータがきた時、変換タイプが flt64to36 と fix64to36 以外の場合には、変換結果をそのまま EM 書き込みに出力する。 EM のアドレスはインクリメントされる。

3. 変換タイプが flt64to36 または fix64to36 の場合には、これらが 2 ワードくる度にそれらをまとめて出力する。つまり、奇数番目の時には変換結果を上位 36 ビットに格納し、出力しない。偶数番目の時には下位 36 ビットに格納し、出力する。出力した時には EM アドレスもインクリメントする。
4. いずれの場合でも CONV アドレスはインクリメントする。

最上位アドレス復帰信号がくると、保存されている最上位アドレスデータ (4 ワード) を EM に書き込む。この機能はいらないかもしれない。

3.3.3 コード出力回路

3.3.3.1 概要

コード出力回路は

- ホストから受け取った機械語命令を内部メモリに拡張し
- 指定された範囲の命令を指定された回数 ISP から送る。この時に、EM-IDP
  - 可能になった時点で次のイテレーションの EM-IDP 転送を起動し、
  - 次のイテレーションは EM-IDP の終了まで待つ

というような動作を行う。

3.3.3.2 入出力

```
HDI      [0:63]   In    データ入力
DEN               In    データライトイネーブル
AEN               In    アドレスイネーブル
BUSY              OUT   コード出力動作中アサートされる
WAIT              In    コード出力の次のイテレーションにウェイトを掛ける
ISPD   [0:31]     Out   ISP データ出力

START             Out    転送起動
RRNSTART          Out    RRN コマンドスタート
IDPID     [0:5]   Out    IDP シーケンス番号
RRNDATA  [0:41?]  Out    RRN コマンドデータ
```

3.3.3.3 機能

コード出力回路はホストインターフェースから書き込み可能な以下のレジスタを持つ

```
Address      Name   Function
0x0          RESET  リセット
0x1          CMR    コマンドレジスタ
0x4000-7FFF  DATA   コードメモリ
```

リセットは、出力動作中の場合にもそれを止めてカウンタ群を初期化する。

コマンドレジスタは以下のフィールドを持つ。

Name	Location	説明
SADDR	[0:12]	命令開始アドレス
NINST	[13:25]	命令語数
NITER	[26:38]	ループ繰り返し数

コードメモリの書き込みは 64 ビットワードアドレスで行うが、 CMD0 のフィールドは 256 ビットワードアドレスで行う。256 ビットワードのアドレス 0, 1 の 128 ビット (1 の下位は未使用フィールド) に ISP ビット列 2,3 の 128 ビットに IDP または RRN フィールドが入る。

コマンドレジスタに書き込むとコード出力動作が始まる。但し、 EM-IDP 転送回路から ISP には ISP シーケンス番号を送るだけである。RRN についても同様で、EM-IDP 転送回路に RRN データとして送るだけである。

指定された語数まで送った時に、 WAIT がアサートされてネゲートされていたらそのまま命令の先頭に戻ってまた送り始める。WAIT がまだネゲートされていなかったらネゲートされるまで待つ。

命令語を送るのは NITER 回繰り返す。

IDP/RRN フィールドの内容は以下の通り

Name	Length	説明
Valid	1	0 ならなにもしない
Type	1	0 なら IDP, 1 なら RRN

以下、 IDP の場合

IDPID	5	IDP シーケンス ID

RRN の場合

RRNDATA	41?	RRN コマンドデータ
CONV	3	変換タイプ
UN	1	非正規化出力フラグ

RRN データの詳細はチップ記述書の通りとする。以下は古いかもしれない。

ADR[0:12]	アドレス初期値
N [0:7]	語数
BBADR[0:4]	アクセスする BB 番号
REDUC	縮約モードかどうか。 1 なら縮約、 0 なら PE 選択。
WL	ワード長 1:長語、 0:短語
FSEL	出力選択。1 なら FADDSUB の出力を取る。 0 なら IALU
NORMALA	ポート A の入力を正規化数とみなす
NORMALB	ポート B の入力を正規化数とみなす
SIGNB	B の符号を反転する (減算)
ROUND	出力を 25 ビットに丸める
NORMALO	出力を正規化する
IALUOP[0:4]	ALU 自体の命令コード
UNSIGNED	符号なし演算を指定 (1 の時に)
ODPOE	ODP から出力する
SREGEN	ステータスレジスタに書き込む

RRN データは EM-IDP 転送回路に直接送るが、N と CONV は ODP-HIO 転送回路に送られる。また、ODP-HIO 回路の UN 入力は RRN データの NORMALO を反転したものになる。

3.3.4 ODP-HIO 転送回路

3.3.4.1 概要

ODP-HIO 転送回路は、 ODP から戻ってきた結果に指定された変換を適用しながらホストに送り返す。どのような変換をするべきかの指定はコード出力回路からくる。

3.3.4.2 入出力

N	[0:12]	In	語数
CONV	[0:2]	In	変換タイプ
UN		In	unnormal フラグ
CS		In	コマンドストローブ
BUSY		OUT	変換動作中アサートされる
ODP	[0:71]	In	ODP データ入力
ODPEN		In	ODP ストローブ
HDO	[0:63]	Out	データ出力
OEN		Out	データ出力ストローブ

3.3.4.3 機能

ODP-HIO 転送回路はホストインターフェースから書き込み可能な以下のレジスタを持つ

CONV は開始アドレスから LBM の数だけ使われる。CONV の 1 語は以下のフィールドを持つ

CONV [0:2] CP での変換タイプ

0	flt72to64	72bit 浮動小数点	->	64bit 浮動小数点 つめて丸める
1	flt36to64	36bit 浮動小数点	->	64bit 浮動小数点
2	fix72to64l	72bit 固定小数点	->	64bit 固定小数点 下をカット
3	fix72to64r	72bit 固定小数点	->	64bit 固定小数点 上をカット
4	fix72to64w	72bit 固定小数点	->	36bit 固定小数点 2 語。36 ビットを下に

である。今のところ 3 ビットだけど、増えるかもしれないので 4 ビットとっておく。

UN (unnormal) フラグは変換タイプ 0 の時にだけ有効で、このフラグが 1 になっている時には入力を非正規化数と解釈する。但し、ハードウェアが実装困難であればこの機能は省略してよい。

ここで、変換タイプ 4 の場合にだけ入力 72 ビット 1 語が 36 ビット 2 語になることに注意して欲しい。タイプ 1 の変換では形式は短語だが長語の上半分しか使ってないので、2 語にはならない。タイプ 4 では 2 語になる。これは、この変換ユニットが 1 語きたら 1 語しまうようにできていたら、タイプ 4 の場合にだけ出力速度が足りないということを意味する。

まともにこれに対応するには、どこかでバッファリングするとかそういうことが必要になるが、これは面倒なのであまりしたくない。従って、タイプ 4 の場合には単純に、SING から来た語のうち奇数番目は無視することにする。

RRN 命令は 1 語発行すると、そこで指定された語数掛ける BB あたり PE の数だけの結果が SING から出力される。従って、次の RRN 命令は、少なくともそれだけのデータがくるのに必要なクロック数だけ待つ必要がある。このため、RRN 命令を発行する回路では命令の必要な箇所をデコーダしてウェイトサイクルを入れる。

結果の変換のほうも、返ってくるデータを数えて処理が終わったら次の変換を見る、というふうにする必要がある。RRN 命令を出してから最初のデータが返ってくるまでのウェイトはわかっているはずなので、それだけ待って切換えてもいい。

ちょっとここでややこしいのは、 RRN 命令を送ってから SING からの答が返ってくるまでのレイテンシが、RRN 命令の発行間隔よりも短い場合を考慮するかどうかである。例えば PE が一つしかないとかいう場合にはこれが問題になる。実チップではそういうことはないが、PE 一つからしか読まないということはありえる。また、 FPGA でのシミュレータでもそういうことがある。従って、サイクル数はわかっているものとして、それだけのパイプラインレジスタというかリングバッファというかそういうものを命令を出すロジックと結果を変換するロジックの間に入れる。この遅延はハードコードするとデバッグが面倒なので、レジスタで設定可能にする。これが LAT レジスタである。

3.4 実装上の注意

一つ大きな問題は、上の仕様レベルでの CP の動作は最新の FPGA でもとても実現できないほどの高速であるということである。つまり、 IDP のデータレートは 1GHz で 8GB/s (まあ、SING が 1GHz で動いたとしてだけどさ) を想定しているので、 これは論理的には、少なくとも EM-IDP は 1GHz で動く必要があるということを意味する。実際にはそんなクロックで動くはずがないので、データレートを落とすことなく動作クロックを下げるためにデータ幅を増やすとかそういうことをする必要がある。

3.5 GRAPE 機能の実現

ここでは、先に述べた GRAPE として使う時に必要な機能が、どのようにして上の基本的な制御回路で実現できるかを述べる。

3.5.1 ホストから SING の実行コードを受け取り、自分の内部メモリに格納する。

これはコード出力回路のレジスタに実行コードを書くだけ。

3.5.2 ホストから定数データを受け取り、それを SING ローカルメモリに格納す

る。

これはまず EM に書き込み、それをさらに BM に転送した後で、必要な実行コードを生成 (ホスト側で) して、コード出力回路に書き込み、実行もさせる。

3.5.3 ホストから j 粒子データを受け取り、 EM に格納する。

これはホスト-EM 転送回路そのまま。

3.5.4 ホストから i 粒子データを受け取り、 SING ローカルメモリに格納する。

これはまず EM に、そこから BM に転送した後で、必要な実行コードを生成 (ホスト側で) して、コード出力回路に書き込み、実行もさせる。

3.5.5 計算本体の前の初期化コードを SING に送る。

コード出力回路に登録したコードを実行するだけ。

3.5.6 EM の j 粒子データを SING に送る。

EM-IDP 転送シーケンサがコード出力回路からキックされる。

3.5.7 計算本体コードを SING に送る。

コード出力回路がループ動作する。

3.5.8 結果回数のための RRN コマンドを SING に送る。

ホストから RRN コマンドを受け取り、 EM に転送した上で EM から転送する。

3.5.9 SING から結果を受け取り、それをホストに転送する。

ODP-HIO 転送回路の機能。

3.6 まとめ

ここでは制御プロセッサの機能の概要をまとめた。

4 ホスト API

以下では、制御プロセッサに対するホスト側の API を与える。

4.1 制御プロセッサモデル

図 3 に制御プロセッサのモデルを示す。複数のプロセッサとかメモリとかはとりあえず考えないで、単純にホスト、制御プロセッサ、外部メモリと SING チップからなるものとする。

4.2 制御プロセッサの動作

制御プロセッサの動作は以下の通りである。

1. ホスト計算機から受け取ったデータを変換シーケンスに従って EM に格納する
2. ホスト計算機から受け取ったデータを EM を通して BM に格納する。
3. ホスト計算機から命令コードを受け取って、自分のコードメモリに格納する。
4. ホスト計算機から EM から BM への転送シーケンスを受け取って、これもコードメモリに格納する。
5. ホスト計算機から繰り返し数を受け取って、上の命令コードをその回数だけ送る。
6. それと並行して転送シーケンスのほうも繰り返し実行する。
7. ホスト計算機から LM からの結果回収シーケンスを受け取って、それを SING ISP/IDP に送り、さらに SING ODP から来たデータをホスト計算機に送り返す。

以下、このそれぞれについて、

1. ホスト API
2. 動作

の詳細を述べる。

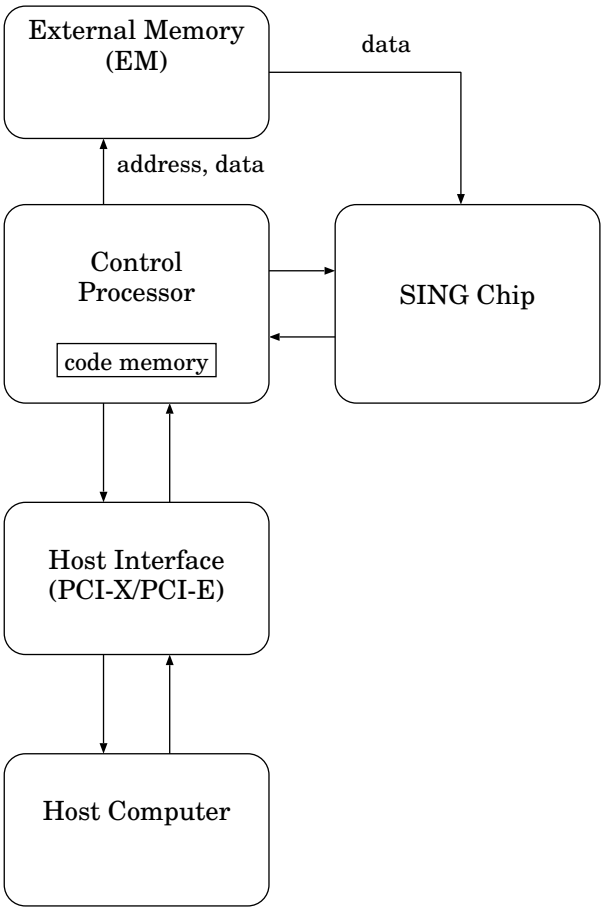


Figure 3: 制御プロセッサのモデル

4.3 制御プロセッサ API と実装の関係

制御プロセッサの物理実装は 1 種類とは限らない。例えば、 EM は 64 ビットワードの実装と 72 ビットワードの実装が考えられ、そのどちらかでどの時点でデータ型変換が起こるかが変わる。また、 ODP の実装にも、

- 何もかもハードウェアがやってくれる
- ホストが色々面倒を見る
- その中間

と何通りもありえる。従って、ここでの API は、制御プロセッサの物理実装によらずにやりたいことが同じ形で表現できるものにしたい。

それができるかどうかを検討する。

ホストから EM への転送

ここで、型変換を明示的に指定できるためには、構造体にしてその要素毎に型変換を指定するのが簡単そうな気がする。

問題は、複数の構造体をどうするか。

簡単には、malloc みたいなものを導入する。つまり、

- データ変換配列
- 変換後のサイズ

によって、 EM 上の構造体を定義して、 malloc みたいな関数で EM 上の領域を確保する。で、これはスタックベースにして最後にアロケートしたものから逆順に解放できるようにする。

現状で、 SING_LM_write_regconv, SING_LM_write はインターフェースとしては問題ないので、これを EM_write で実現するようなことをしたい。

BM にもうちょっとなんかいるかな？これもアロケートができる必要あり？

4.4 初期設定、ユーティリティ関数

実際の転送用関数群の他に、最低限の初期化、デバイス獲得、解放をする関数といくつかのパラメータ (PE,BB の数等) を返す関数が必要である。ここではそれらの仕様を与える。

```
int SING_open()
```

シミュレータでは今のところ何もしないけど、なんかいりそうなのでつけておく。実際のハードウェアをいじるコードではハードを占有する (対応するデバイスファイルをオープンする) などの操作が起きるはず。

```
int SING_close()
```

これも上のと同じ。

```
int SING_number_of_pe()
```

BB あたり PE の数を返す

```
int SING_number_of_bb()
```

チップ (または論理的なユニット、ボードとか) あたり BB の数を返す。

4.5 EM 上での論理的なメモリアロケーション

EM の実装は色々ありえるが、とりあえずアプリケーション側からは、 EM はデータ変換列によって指定される構造体みたいなもの (以下、単に構造体と呼ぶ) の配列を複数おけるものとする。 以下の関数によって構造体を定義し、それに割り当てられるメモリを EM 上に確保する。

```
int SING_EM_alloc(int   length,
  int*  conversions,
                int   size,
                int*  err)
```

length	conversions の語数
conversions	データ変換を指定する配列
size	アロケートする構造体の数
err	エラーコード

conversions は以下の値が語数 length 分はいった整数配列である。

0	flt64to72	64bit	浮動小数点 ->	72bit	浮動小数点
1	flt64to36	64bit	浮動小数点 ->	36bit	浮動小数点 この時はつめる
2	flt32to36	32bit	浮動小数点 ->	36bit	浮動小数点
3	fix64to72l	64bit	固定小数点 ->	72bit	固定小数点 下に拡張
4	fix64to72rs	64bit	固定小数点 ->	72bit	固定小数点 上に拡張 符号あり
5	fix64to72ru	64bit	固定小数点 ->	72bit	固定小数点 上に拡張 符号なし
6	fix64to36	64bit	固定小数点 ->	36bit	固定小数点 上をカット
7	fix32to36l	32bit	固定小数点 ->	36bit	固定小数点 下に拡張
8	fix32to36rs	32bit	固定小数点 ->	36bit	固定小数点 上に拡張 符号なし
9	fix32to36ru	32bit	固定小数点 ->	36bit	固定小数点 上に拡張 符号あり

ホスト上のデータはこのテーブルに従って変換されて EM に格納される。従って、実際に EM に転送されるバイト数、つまり構造体一つが EM で占める領域の大きさは、 length が同じでも conversions の中身によって変わる。

帰り値は 0 または正の時にはエリア番号 areaid であり、 負の時にはエラーである。エラーの詳細は err の値であり

1	メモリ不足
2	変換テーブルの間違い。長語が長語境界で始まっていない
4	登録数の上限を超えた
8	終点がテーブルサイズを超えた
16	変換テーブルにありえない数字がはいった
32 and higher	未定義

である。areaid は他のいくつかの関数で EM 上のエリアを指定するのに使われる。

```
int SING_EM_free()
```

これは、最後にアロケートされたエリアを解放する。帰り値は残っている登録されたエリアの数である。負の時はエラーである。但し、何も登録されていない時には帰り値は 0 になる。

```
int SING_EM_areas()
```

現在登録されているエリアの数を返す。
なお、システム側で利用する関数として

```
int SING_EM_alloc_system
int SING_EM_free_system
```

を準備する。これらの引数は system がつかないものと同じである。これらはアプリケーション側から呼んではならない。

4.6 ホストから EM への転送

これを以下では HE 転送あるいは HET (Host-to-External memory transfer) と呼ぶ。

4.6.1 API

```
int SING_EM_send(int areaid,
                 int start,
                 int ndata,
         void* hostp)

areaid          エリア id。SING_EM_alloc の帰り値
start           エリア内のストアする領域の先頭番号
ndata           ストアするデータの数
hostp           元データがある領域の先頭を指すポインタ


返り値
  0: 正常終了
  1: areaid が範囲外
  2: start が範囲外
  4: start+ndata が範囲外
 それ以外 未定義
```

これも system がつくものを持つ。
これはホストメモリのある領域を変換して EM のある領域にコピーする。

4.7 ホスト計算機から S-LM への転送

これを以下では HL 転送、 HLT (Host-to-local memory transfer) と呼ぶ

4.7.1 API

```
int SING_LM_write_regconv(int   length,
  int*  conversions

    length          語数
    conversions     データ変換を指定する配列
```

```
int SING_LM_write(void* hostp,
  int   lmoffset,
  int   peeach)

    hostp          ホストメモリ先頭ポインタ
    lmoffset       LM の開始アドレス (短語)
    peeach         0 でなければその数の PE に別のデータ送る
                   0 の時は全 PE に同一のデータを送る

int SING_LM_write_to_onepe(void* hostp,
   int   lmoffset,
           int   ipe,
                          int   ibb)

    hostp          ホストメモリ先頭ポインタ
    lmoffset       LM の開始アドレス (短語)
    ipe            PEID
    ibb            BBID
```

使いかたとしては、まず SING_LM_write_regconv で変換を登録し、SING_LM_write で実際のデータを送る。書き直しをしない限り SING_LM_write_regconv で登録した変換が繰り返し使われる。SING_LM_write_to_onepe は ipe, ibb で指定した PE 1 つにだけ書き込む。

これらは古い版との互換性のために用意されている。

length, conversions は SING_EM_alloc と同じである。

peeach は、0 であれば全 PE に同じデータを書くが、 0 以外の時には各 PE 毎に違うデータを書く。この時、ホスト側では PE の数 (SING_number_of_pe の返り値) だけのデータを hostp から始める連続領域に用意する必要がある。

読み出し関数は別に用意されることに注意。名前は SING_LM_read だが引数の考え方は write とはだいぶ違う。

4.8 ホスト計算機からの命令コードの受け取り

これを I 登録、 IR (Instruction register) と呼ぶ。

4.8.1 API

```
int SING_iregister(int ninsts,
                   char ** insts)

 ninsts 命令語数、
 insts  命令 (機械語、ISP 文字列とデバッグ命令の文字列の配列)
```

とする。返り値は登録されたルーチンの ID で、負の時にはエラーとする。

```
int SING_iflush()
```

登録したルーチンを全てクリアする。

```
int SING_ipop()
```

最後に登録されたものをクリアする。

4.9 BM 上でのメモリアロケーション

```
int SING_BM_alloc(int   areaid,
                  int   size,
                  int*  err)

    areaid          構造体 id (SING_EM_alloc の帰り値)
    size            アロケートする構造体の数
    err             エラーコード

int SING_BM_free()
```

4.10 EM-BM 転送

```
int SING_EM_BM_transfer(int areaid,
                        int emstart,
                        int bmstart,
                        int ndata,
                        int mode,
                        int bbid)
```

BM の指定されたエリアの部分領域に EM の対応エリアの部分領域をコピーする

```
areaid  構造体 id (SING_BM_alloc の帰り値)
emstart EM の開始インデックス
bmstart BM の開始インデックス
ndata   転送される数
mode    転送モード
bbid    転送モード 0 の時に、 0 または正なら書く bb 番号
```

転送モードは以下の通りである

```
0   EM 上のデータが全 BM に放送される。 ndata の数だけ書かれる。
1   EM の k 番目のデータが BM k に書かれる。
```

帰り値はエラーコードで

```
0 正常終了
1 emstart 不正
2 bmstart 不正
4 ndata   不正
8 mode    不正
```

4.11 計算実行

これは SING 実行 SE, (SING Execution) という名前ということで

4.11.1 API

```
int SING_execute(int init_id,
                 int loop_id,
 int final_id,
 int loop_count,
 void * loop_out,
 void * final_out)
```

init_id	初期化ルーチンの番号
loop_id	繰り返しルーチンの番号
final_id	終了処理ルーチンの番号
loop_count	loop 繰り返し回数
loop_out	loop_id によって指定されたコード実行中にでる結果を格納する領域
final_out	final_id によって指定されたコード実行中にでる結果を格納する領域

返り値は 0 以外だったらなんかエラー。実行は、

1. まず初期ルーチンが実行される。
2. 初期化ルーチンが終了したら、繰り返しルーチンを起動する。
3. 繰り返しルーチンは loop_count 回実行される。
4. 繰り返しルーチン実行中にでる出力は loop_out によって指定された領域に格納される。
5. 繰り返しルーチンが終わったら終了処理ルーチンを起動する。これの実行中にでる出力は final_out によって指定された領域に格納される。

というものになる。

繰り返しルーチン、終了処理ルーチンの両方とも、前の初期化なり繰り返しで起動された EM-BM 転送があれば、それが終了するまで待ってから次を起動する。

4.12 結果回収

これは LH 転送、LHT (Local-memory-Host transfer) ということにする。

4.12.1 API

基本的には

```
int SING_LM_read_regconv(length,
 int* conversions,
 int rlength,
 RRNINST* rints,
 int lmoffset,
 int pereduce,
 int bbindex)
```

```
    length          語数
    conversions     データ変換を指定する配列
    rlength         RRN コマンドの数
    rinsts          チップ上での RRN コマンドを指定
    lmoffset        LM の開始アドレス (短語)
    peeduce         1 なら縮約している。
    bbindex         -1 なら全部 (縮約していれば無視)、それ以外の時は
                    指定した BB のデータだけを読む

int SING_LM_read(void* hostp)

    hostp           ホストメモリ先頭アドレス
```

というようなものでいいはず。

SING_LM_write と同様に、変換テーブルを登録するルーチンと、実際に転送を要求するルーチンに分ける。

conversions の中身は以下の通り

```
0 flt72to64   72bit 浮動小数点 ->  64bit 浮動小数点 つめて丸める
1 flt36to64   36bit 浮動小数点 ->  64bit 浮動小数点
2 fix72to64l  72bit 固定小数点 ->  64bit 固定小数点 下をカット
3 fix72to64r  72bit 固定小数点 ->  64bit 固定小数点 上をカット
4 fix72to64w  72bit 固定小数点 ->  36bit 固定小数点 2 語。36 ビットを下に
```

length はこの conversions の長さを表すので、もどってくる語数は conversions の中身による。Rinsts は RRN コマンド列を与える。 RRN コマンドはテキスト形式機械語で与えられる。 つまり、 RRN で始まって 15 個のフィールド全てを指定する必要がある。

あ、こういうインターフェースでは、 BM に結果が入りきらない時に対応できないですね。ありゃ。えーと、どうしましょうか。

1. 1 語 (ベクトル) ずつ PE から転送しては RRN を発行する。
2. 入りきらない時は諦める。

とりあえず 2 でもいいような気もするけど、いくらなんでも美しくない。とすると、ベクトル 1 語毎に LM から BM にコピーして RRN を発行するのが簡単。問題はこれが上の指定で可能かだが、conversions を 4 語毎に見れば OK。というより、 conversions をそもそもベクトル変数単位で与えれば良い。

- RRN コマンドのほうでは開始アドレスは常に 0
- RRN コマンドの数は常に length (conversions の語数) と同じ。従って指定不要

となる。もう一度ちゃんと書くと

```
int SING_LM_read_regconv(length,
 int* conversions,
 char** rinsts,
 int lmoffset,
 int pereduce,
 int bbindex)
```

length	語数。ベクトル変数単位
conversions	データ変換を指定する配列
rinsts	チップ上での RRN コマンドを指定
lmoffset	LM の開始アドレス (短語)
peeduce	1 なら縮約している。
bbindex	縮約していれば無視、それ以外の時は 指定した BB のデータだけを読む

である。 bbindex の -1 (全 BB を順番に読む) はとりあえず実装しないことにする。

積算は unnormal でするのが標準と考えられる。BB をまたがった縮約も unnormal のままなので、誰かがどこかで変換する必要がある。

考え方は以下の 3 通りくらい

- 制御プロセッサの中で normalize してから 64 ビットにつめる。
- 制御プロセッサの中では無理矢理 64 ビットにつめて、ホストでオフセットを補正する。
- 制御プロセッサからホストまで 72 ビットで戻す。

最も正確で高速なのは最初の制御プロセッサの中で変換するものだが、これは制御プロセッサの中に大きなシフタが必要になる。入らないことはないと思うがそういうコードを書くのは面倒である。従って、必要に応じて残りの 2 つを使いわけることにする。

5 アセンブラとのインターフェース

ユーザープログラムとの API は決めたとして、そこで指定する

- Host-¿LM データ変換指定
- EM-¿LM データ変換指定
- 命令列
- RRN 命令列
- LM-¿ ホストデータ変換列

を人間は書くのは嫌である。従って、これらはアセンブラからなるべく自動に近い方法で生成できる必要がある。現在のアセンブラは

- IDP パケット
- ISP 命令
- RRN 命令

を基本的にはそのまま指定するだけ。これらの、部分的には暗黙な拡張が必要。

データ転送については、全てローカルメモリに置くとして、

- どれを送るか

- 変換をどうするか

だけ指定すれば必要な情報は全てある。従って、これらから API に必要なデータを生成できるはずである。
アセンブラについては、どこが初期化でどこがループかという指定があれば十分。フックの位置は判断できるはず。

6 付録

6.1 メモリインターフェース動作の概要

当初の予定では、制御プロセッサは通常のマイクロプロセッサに対する North Bridge 的な位置付けであり、メモリ制御は全て制御プロセッサが行い、 SING チップは制御プロセッサを通してメモリデータを受け取る構成であった。
しかし、この構成は

- 制御プロセッサのピン数が非常に大きくなる
- 制御プロセッサが高速動作する必要がある

という 2 つの大きな問題がある。商用の North Bridge であればダイサイズが小さくてピンが多いものを作ればいいのでそんなに問題ではないが、 FPGA を使う場合には I/O ネックのためにダイサイズの大きい高価なものを使う必要が発生して非常に望ましくない。この問題を解消するため、メモリの I/O 独立な QDR-II SRAM を用いる。
これはもちろんメモリのビット単価の上昇をもたらすが、これは容量を小さくすることでごまかす。量産時には 144Mbit 品とかが入手可能であると期待すると、余裕があるとは言えないもののなんとかなる容量である。

QDR-II には 4 ワードバーストと 2 ワードバーストの 2 種類がある。この 2 つ

の違いは内部構造として 4 バンクか 2 バンクかであり、4 ワードバーストの場合は

- 最初のクロック立ち上がりで read address を入れる。そのあと 2 サイクル DDR で 4 ワードバースト
- 次のクロック立ち上がりで write address を入れる。で、同様に 4 ワード書き込み

という動作が可能である。言い換えると、 read, write の制御回路はそれぞれメモリクロック速度の半分の速度で動くのでもかまわない。特に、 read burst ではデータを書いてしまうが、 write burst では書き込みするかどうかをワード毎に制御できるのでホストからランダムにデータがきてもそれに対応するのはあまり難しいわけではない。
ちょっと面倒になるのは、 IDP パケットヘッダが出なくなることである。
これには以下のような対応が考えられる。

1. パケットヘッダ用の専用ポートをつける
2. パケットヘッダを一旦メモリに書いてそれを読み出す
3. パケットヘッダを ISP から入れる

どれがよいかは難しい。それぞれメリット・デメリットがあるからである。
専用ポートメリット:

- 回路設計が簡単
- コマンド出力の起動オーバーヘッドが小さい
- コマンド出力とデータ出力をオーバーラップできる

デメリット

- ピン数が増える
- SING チップ仕様が変わる

メモリから出すメリット

- ピン数が増えない
- SING チップ仕様変更なし

デメリット

- 制御プロセッサの動作がちょっと複雑
- コマンド出力の起動オーバーヘッドが大きい

ISP から出すメリット

- ピン数が増えない
- コマンド出力の起動オーバーヘッドが小さい

デメリット

- SING チップ仕様が変わる
- 性能には多少インパクトある。転送起動と計算が並列にならないため

ピン数の増加は大したことはないので、作業時間に問題がなければコマンドポートを別に設けるのが望ましいが、今回は SING チップの設計作業の手間を増やしたくないので (2) を取りたい。この場合の動作は

1. メモリのどこか固定アドレスにパケットヘッダを書く
2. データ転送シーケンサは、必ず最初にパケットヘッダを出してから指定されたデータを送るようにする

と、基本的にはこれだけ。問題は、メモリの全ワードを使いたい時にどうするかである。

まあ、最後のワードだけなので、以下のような動作も可能と思われる。

1. 最後のワードについては制御プロセッサの中で憶えておく
2. 「最後の 4(x2 とかでもかまわない) ワードだけを転送」というのは禁止にする
3. 最後のワードを含む転送の時には転送開始してから最後のワードのあるべきデータを書き戻す

問題は、書いたデータが読めるのはいつかということ。Cypress のデータシートでは write data をフォワードする仕掛けがあるとのことで、 write した次のサイクルで同じアドレスから read をかけてもちゃんとそのアドレスがでるらしい。なかなか良く出来ている。

うーん、というか、 SING でもそれつければいいのか。どうするかな？

アセンブラで書くならあんまり性能向上にはならないかもしれないけど。

とりあえず今回はしない方針で。

6.1.1 QDR メモリへの書き込みについて

FPGA の動作速度をどうするかという問題でもあるが、 PCI-Ex インターフェース側はともかくそれ以外ののところはあまりクロックをあげたくない。具体的には、チップに対して 1/4 動作にしたい。

で、書き込みのところで DDR で 500MHz 動作とかしないといけないのはなんだか嫌だし、そんなに速く書けてもしょうがないので、とりあえず書き込みは最大 1 GB/s ということで 125 MHz 64(72) ビット書き込みが出来ればいいことにする。

4 ワードアドレスバーストは 4 ワードバウンダリから始まるんだっけ？

そうみたい。とすると、 有効ワードを 1 にしてもデータ有効ウィンドが足りないので、うーん、どうすればいいのかな？

4 クロックで 2 ワード転送を基本にするってのはどうかしら？

```
CLK ____====____====____====____====____
DATA  <=0><=1><=2><=3><=4><=5><=6><=7>
```

あ、そうではなくて

```
CLK ____====____====____====____====____====____====____
DATA  <=0>----<=2>----<=4>----<=6>----
DATA  ----<=1>----<=3>----<=5>----<=7>
DATA  <=0>----<=2>---------<=5>----<=7>
DATA  ----<=1>----<=3>-----------------<=0>----<=2>----
```

と、奇数ワード書き込みから偶数ワード書き込みに切り替わる時には wait state を入れる、というのでいいはず。

少しややこしいが、何をどこに書くかというのがはいったメモリをもっていて、書けるものを書く、という動作。

7 メモ

ISP のチェイン実行を許すべきか？

- オーバーヘッド削減の意味からは望ましい
- 具体的に必要なのは
  - まず初期化を実行
  - 次にループ本体を実行
  - 最後に結果を BM に移すシーケンスを実行。これと並行して RRN を発行

  というようなもの。

チェイニングを実装するためには、デスクリプタ的なものを作るのがもっとも簡単。

つまり、

- CMR フィールドに継続ビットを付ける
- CMR 書き込みで起動ではなく、別にする

まあ、そういう変更は大して面倒ではないので、後で追加すればよい。

7.1 2005/6/4

i 粒子の書き込みとか定数データを書くのも EM を通すことになる。メモリ割り付けをどうするか？
というよりも、それはソフトウェアのほうに任せてハードウェアはそういうのを意識しないですますべき。
EM-IDP 転送シーケンサの機能をもうちょっと整理したい。
あと、間接アドレスの可能を考えるかどうか。 cell index ならオンチップのメモリくらいでなんとかなるかな？
EM の 2 次元領域を BM に転送するだけ。
パラメータ

- 転送語数
- 転送モード (BM1 つ、放送、全 BM に別データの 3 種類。これはモードビット)
- BM ID
- EM 開始アドレス
- BM 開始アドレス
- EM ストライド

I 粒子転送と J 粒子転送ではこれは結局全部違う。
とりあえずレジスタはあんまりまとめたりしない方針で。
HII-IDP 転送回路 はなしになったのでメモのところに残しておく

7.1.1 HII-IDP 転送回路 (現在廃止)

7.1.1.1 概要

この回路は EM-IDP 転送シーケンサと同様に、64 ビットデータを指定された変換テーブルをみながら 72 ビットに変換して送る。 EM-IDP との違いは

- データが HII から来る
- 繰り返しの時に全 BB に書き込み、 BB アドレスもインクリメントするモードがある

の 2 点である。

7.1.1.2 入出力

HDI	[0:63]	In	データ入力
DEN		In	データライトイネーブル
AEN		In	アドレスイネーブル
BUSY		OUT	変換動作中アサートされる
IDPD	[0:71]	Out	IDP データ出力
IDPEN		Out	IDP ストローブ

7.1.1.3 機能

HII-IDP 転送回路はホストインターフェースから書き込み可能な以下のレジスタを持つ

Address	Name	Function
0x0		データカウンタリセット
0x1	BMSADR	BM 書き込み開始アドレス
0x2	NBM	BM の数
0x3	LBM	転送シーケンス BM 語数
0x4	LHII	転送シーケンスホスト入力語数
0x5	BMALL	全 BM に同一データを書き込み
0x100-1FF	CONV	変換指定
0x200-3FF	DATA	データ受け取りエリア

BMSADR、NBM、LBM、LHII、CONV は EM-IDP 転送シーケンサの対応するレジスタと同じ (LHII は LEM と) 機能を持つ。EM-IDP では EM 開始アドレスを指定する必要があったが、 HII-IDP では書き込むデータは HDI から供給されるので必要ない。

ホストの動作としては、レジスタに必要なデータを書いた後で、 DATA エリア (アドレスはこの範囲で任意) に必要な語数のデータを書くことになる。DATA エリアにバースト書き込みをした時は、アドレスはインクリメントされない、つまり、新しくアドレスフェーズがくるまでは DATA エリアへの書き込みとみなす。

送られるべきデータの語数は NBM × LHII となる。 BMALL が 1 の時には、NBM だけ転送シーケンスを繰り返すのは通常動作 (EM-IDP と同じく、書き込む BM 番号をインクリメントする) のと同じだが、書き込みは全 BB に行い、BM 書き込みアドレスを 1 回のシーケンスの後で戻さない。

なお、少なくとも通常動作の場合には、BM 一つに書いたあとで IDP パケットヘッダを送る必要があるのでダミーサイクルが入り、 HDI から連続でデータが来た場合には対応できない。しかし、現実には HDI からのデータレートは IDP のデータレートよりも必ず低いので、これは問題にならないことに注意して欲しい。

7.1.2 ODP-HIO 転送回路 (古い版)

7.1.2.1 概要

ODP-HIO 転送回路は、 ODP から戻ってきた結果に指定された変換を適用しながらホストに送り返す。

7.1.2.2 入出力

HDI	[0:63]	In	データ入力
DEN		In	データライトイネーブル
AEN		In	アドレスイネーブル
BUSY		OUT	変換動作中アサートされる
ODP	[0:71]	In	ODP データ入力
ODPEN		In	ODP ストローブ
HDO	[0:63]	Out	データ出力
OEN		Out	データ出力ストローブ

7.1.2.3 機能

ODP-HIO 転送回路はホストインターフェースから書き込み可能な以下のレジスタを持つ

Address	Name	Function

```
0x0         RESET   データカウンタリセット
0x0         START   起動
0x2         NPE     PE の数
0x3         LBM     転送シーケンス BM 語数
0x4         LHII    転送シーケンスホスト入力語数
0x5         LAT     SING レイテンシ
0x100-1FF   CONV    変換指定
```

CONV は開始アドレスから LBM の数だけ使われる。CONV の 1 語は以下のフィールドを持つ

```
CPCONV [0:3]  CP での変換タイプ


0 flt72to64   72bit 浮動小数点 ->  64bit 浮動小数点 つめて丸める
1 flt36to64   36bit 浮動小数点 ->  64bit 浮動小数点
2 fix72to64l  72bit 固定小数点 ->  64bit 固定小数点 下をカット
3 fix72to64r  72bit 固定小数点 ->  64bit 固定小数点 上をカット
4 fix72to64w  72bit 固定小数点 ->  36bit 固定小数点 2 語。36 ビットを下に
```

である。今のところ 3 ビットだけど、増えるかもしれないので 4 ビットとっておく。

ここで、変換タイプ 4 の場合にだけ入力 72 ビット 1 語が 36 ビット 2 語になることに注意して欲しい。タイプ 1 の変換では形式は短語だが長語の上半分しか使ってないので、2 語にはならない。タイプ 4 では 2 語になる。これは、この変換ユニットが 1 語きたら 1 語しまうようにできていたら、タイプ 4 の場合にだけ出力速度が足りないということを意味する。

まともにこれに対応するには、どこかでバッファリングするとかそういうことが必要になるが、これは面倒なのであまりしたくない。従って、タイプ 4 の場合には単純に、SING から来た語のうち奇数番目は無視することにする。

RRN 命令は 1 語発行すると、そこで指定された語数掛ける BB あたり PE の数だけの結果が SING から出力される。従って、次の RRN 命令は、少なくともそれだけのデータがくるのに必要なクロック数だけ待つ必要がある。このため、RRN 命令を発行する回路では命令の必要な箇所をデコーダしてウェイトサイクルを入れる。

結果の変換のほうも、返ってくるデータを数えて処理が終わったら次の変換を見る、というふうにする必要がある。RRN 命令を出してから最初のデータが返ってくるまでのウェイトはわかっているはずなので、それだけ待って切換えてもいい。

ちょっとここでややこしいのは、 RRN 命令を送ってから SING からの答が返ってくるまでのレイテンシが、RRN 命令の発行間隔よりも短い場合を考慮するかどうかである。例えば PE が一つしかないとかいう場合にはこれが問題になる。実チップではそういうことはないが、PE 一つからしか読まないということはありえる。また、 FPGA でのシミュレータでもそういうことがある。従って、サイクル数はわかっているものとして、それだけのパイプラインレジスタというかリングバッファというかそういうものを命令を出すロジックと結果を変換するロジックの間に入れる。この遅延はハードコードするとデバッグが面倒なので、レジスタで設定可能にする。これが LAT レジスタである。

7.1.3 ODP-HIO 転送回路 (簡易版)

7.1.3.1 概要

これはテストボード等用の ODP 回路で、フル回路に比べて以下の点で簡略化されている

- 変換テーブルが 1 語であり、ループ動作しない

7.1.3.2 入出力

HDI	[0:63]	In	データ入力
DEN		In	データライトイネーブル
AEN		In	アドレスイネーブル
BUSY		OUT	変換動作中アサートされる
ODP	[0:71]	In	ODP データ入力
ODPEN		In	ODP ストローブ
HDO	[0:63]	Out	データ出力
OEN		Out	データ出力ストローブ

7.1.3.3 機能

ODP-HIO 転送回路はホストインターフェースから書き込み可能な以下のレジスタを持つ

Address	Name	Function
0x0	RESET	データカウンタリセット
0x0	START	起動

START レジスタは

0:3 変換タイプ
4:8 受け取る語数 (NPE に相当)

となる。