GRAPE-8 デザインメモ

Ver 0.0 — 2009/12/4 Ver 0.01 — 2010/6/6

牧野淳一郎

July 22, 2011

Contents

1 知られている未修正バグ **5**

2 履歴 **7**

2.1 2009/12/4 7

2.2 2010/6/6 7

3 何を実装するか？ **9**

3.1 演算パイプライン 9

3.1.1 eASIC 固有の問題 9

3.2 原理的な疑問 10

3.3 概略仕様 10

4 詳細仕様 **13**

4.1 中心力計算パイプライン 14

4.2 入出力ポート 16

4.3 メモリマップ 17

5 詳細仕様 **21**

5.1 パイプライン 21

5.1.1 アドレスマップ 24

5.1.2 距離差計算ユニット (DD) 25

5.1.3 二乗距離計算ユニット (DS) 25

5.1.4 -1.5 乗計算ユニット (POW32) 26

5.1.5 最近接粒子ユニット (NNB) 26

5.1.6 ネイバーリストユニット (NBL) 27

5.1.7 関数補間ユニット (FIQ) 27

5.1.8 積算ユニット (ACC) 30

5.2 ブロックパイプライン 30

5.3 J 粒子メモリユニット 31

5.4 トップレベル回路 32

6 シミュレータ、エミュレータ等について 35

6.1 最上位インターフェース 35
6.2 シミュレータ 37
6.3 エミュレータデバッグ用のインターフェース 37
6.4 ブロック、チップのエミュレータインターフェース 38

7 チップ入出力等定義 41

7.1 I/O 規格 41
7.2 タイミング制約 41

8 メモ 43

8.1 pg2 でのアキュムレータ動作 43
8.2 eta 43
8.3 川井版 pg なんとかでの型変換 43
8.3.1 整数から浮動小数点 43
8.3.2 エミュレータデバッグ。 44
8.4 ハードウェアデバッグ 44
8.5 メインパイプラインについてのメモ 46
8.6 2010/9/12 74
8.7 2010/9/17 76
8.8 2010/10/29 76
8.9 2010/11/7 77
8.10 I/O 定義 81
8.11 ピン配置 82
8.12 2010/12/2 85
8.13 2010/12/8 85
8.14 2010/12/13 86
8.15 2010/12/30 87
8.16 2011/1/16 88
8.17 2011/1/29 91
8.18 テストパターンの作成方法 92
8.19 イノテックからの vcd ファイル 93
8.20 2011/2/18 95
8.21 2011/2/21 96
8.22 2011/2/25 99
8.23 2011/3/3 102

Chapter 1

知られている未修正バグ

まだない。

Chapter 2

履歴

2.1 2009/12/4

書き始める。

2.2 2010/6/6

eASIC の状況の応じてだいぶ変更。

Chapter 3

何を実装するか？

3.1 演算パイプライン

基本には従来の低精度 GRAPE だが、

- Oshino scheme に対応するため、ソフトニングだけでなく外側にもカットオフを付ける
- ツリーコードの演算量を減らすため、4 重極を計算する。(これは今回面倒なのでやめる)
- 座標入力は 32 ビット固定小数点
- 相対精度は 8-10 ビットの間。
- 積算は 64 ビット

3.1.1 eASIC 固有の問題

RAM が極めて少ないので、かなり使いづらい。512 ビットのレジスタファイルが 640 個。これは最大幅で 32 ビット。

これを 2 個使って両側にカットオフをつけることを考える。

エントリーは 16 ビットで出力は 32 ビット。カットオフなので、固定小数点 10 ビット出力あればいいかな。DLL 多項式で 2 次、16 エントリー (実際に使っているのは 8 エントリー) で、精度は最大誤差で 0.012 くらいなのでまあ十分。

この時、 0-1 を 12 ビット固定小数点で表して、3 ビットづつ落とすくらい？4 ビット落としては多分導関数がオーバーフローする？下の、指数をいれることにすれば 8 ビット、4 ビットでいいかな？

従って、12+9+6 ビットの合計 27 ビット。ちょっとあまるので指数がいれられる？乗算器は 6 ビットと 9 ビットで、大体 130 セルくらい。ちょっと多い気がするけどしょうがない。

エントリーを整数にするのは無駄なので、16 エントリーの場合 0.5 から 1 までを 8 エントリー、0.25 から 0.5 までを 4 エントリ、 0.125 から 0.25 までを 2 エントリ、 1/16 から 1/8 までを 1 エントリーにしたい。1/16 から下については、2 ー 3 ビットの精度で固定小数点に変換する。ここでは、カットオフ関数の変化は小さいはずなのであまり高い分解能はいらない。

出力は対数表現とする。

ま、これは実際にシミュレータコード書かないと駄目ですね。

力とポテンシャルと両方に 2 つカットオフをいれるとテーブル、補間多項式が 4 個必要。これは多すぎる気がする。8 エントリーで 3 次多項式にしてみる？この場合、4 次で 10 ビットなので 2 ビットづつしかきれない。10+8+6+4 で 28 ビット。この場合、両側のテーブルで同じメモリを使う。外側のスケールを s_1, 内側を s_2 とすると、補間テーブルの精度をなるべく有効に使うためにはそれぞれの対数表現の差の間を 8 等分することが望ましい。これは、割算回路をつけるということ。3 ビットとかならいけないこともない？

あ、そんなに簡単な話ではない。 s_0 がずっと小さいかもしれないから。そうすると、単純にそれぞれを 8 エントリでやる。入力レンジがオーバーラップするので、オーバーラップしていたら内側をとる。

3.2 原理的な疑問

bangling みたいな、薄いディスクの不安定性は表現できるのか？

薄いといっても厚さ 0 ではない。ディスクが座標軸と並行ならば引き算前の固定小数点で表現できるものは表現可能。傾いていると誤差が大きくなる。

3.3 概略仕様

- 座標入力は 32 ビット固定小数点
- 積算は 48 ビット
- 中間表現は対数。精度は 9 ビット。

計算するものを決める。

ソフトニングの対称化を可能にするため、 ϵ_i^2 ϵ_j^2 を両方いれて、

$$r_s^2 = r^2 + \epsilon_j^2 + \epsilon_i^2 \tag{3.1}$$

を計算するようにする。力についても、カットオフをいれるために、任意関数 $g(r)$ を使って

$$\mathbf{f}_{ij} = \frac{g(r)\mathbf{r}_{ij}}{r_s^3} \tag{3.2}$$

を計算するようにしたい。この時に、 $g(r)$ の計算方法をどうするか？

テーブル構成をどうするか？これは基本的にコンパクトサポートを持つ関数なので、テーブルへの入力は対数ではなく固定小数点形式が望ましい。といっても、原理的にはそんなに真面目に変換する必要はないはずで、指数が 1 減る度に仮数を 1 ビット切り捨ててつめていくだけでよいはず。で、これでできる値からカットオフ関数への変換テーブルを、区分多項式で補間する。2 次補間なら 32 エントリ程度でよいので、それくらいのテーブルにする。メモリは 12+8+4 で 24 ビット幅で十分なはずなので、全体で 768 ビット。 eASIC は 740 で 11520kbits あるので、パイプラインを 300 本 (こんなに本当にはいるのか？) いれてもまだ 200kbits, 全く無視できる。

これは、上のほうにちょっと書いた通り、レジスタファイルでできる範囲でやる。ポテンシャルと力でレジスタファイル 1 つづつですます。

粒子メモリについては、座標データが 32bit x 3、質量、ソフトニングがそれぞれ 16 ビットとして、合計 128 ビットである。4M ビット使うことにすると 3 万粒子になって、十分以上であろう。

メモリが内蔵なので、仮想パイプラインを考える必要はあまりない。但し、最終段の積算回路をパイプライン化したほうがよいかもしれない。

eGroup は 320 個、bRAM は 36kbits だが、最大 36 ビット幅までなので、テーブルには少し使いにくい？ただ、これは 18kbits, 36 ビット幅のメモリ 2 つとしても使えるように見える。

レジスタファイルは 640 個、 512 ビット (最大 16W 32 ビット) である。

74 万セルで 300 パイプラインとすると、2300 セル。なんか無理な気がする。最初の減算: 32 ビットでやるので、 96 セル最終の積算: 48 ビットでやることにして、ポテンシャルも合わせて 192 セル。

最終のシフタ: 8 ビットから。 1 セルが 2x2 のスイッチになる。16x4, 32, 48 なので、144 セル、4 個で 600 セル。

最初のシフタ:エンコードしてから、普通にやると 32+16*5 = 96 セル、合計 30 セル

対数への変換:テーブルは 50 セルくらいかな？ここでの指数は 5 ビットくらい。7 ビットでやることにするとデータは 16 ビット。

これは現在は単にテーブルから論理合成かけている。他に考えられる実装

- 多項式 1 つ
- 区分多項式

基本的に、3 次多項式なら十分。 2 次ともう 1 ビットくらいとか。

問題は、多項式の場合乗算器が大きいこと。まあ、固定入力なので、論理圧縮がそうとう効くはず。FA の数は大体半分になるはずだから。

問題は加算器ですね。

1+x を計算するので、 関数としては log(1+exp(x/log(2)))/log(2) だけど、 x のレンジが広い。

2 乗を浮動小数点のままでやってから加算するのはどうか？これは乗算器が必要なのがもちろん問題。まあ、川井君回路はそうなっているので、使えばいいかもしれない。損得を考えると、乗算器が 6 個増えるのと、対数への変換と逆変換を 6 個するのがどうか、という話。多項式でやるならおんなじか。でも、対数の場合には対数での加算器が余計。で、浮動小数点だとカットオフとか平方根とかが余計。なので、多分どっちでも同じくらい。

対数加算器の効率が良い実装はどんな感じか？

区分多項式でやるとして、問題は入力が小さいところ。数学的には、小さいことを使った漸近展開をするべき？例えば 9 ビットなので 1/1024 の途中つまり指数の差 10 の途中までが LSB, 9 の途中までが 2,... で、 0 から -1 までを多項式 1 つですますとして、

区分多項式でするくらいなら浮動小数点に戻しても同じ？

今回、結局全体を浮動小数点で。

粒子のデータ形式は、座標 32 ビット、質量 19 ビット (指数 8 ビットと仮数 10 ビット、符号もあり)、4 重極はトレースレスではない対称行列なので 6 成分、スケールに 8 ビットいれたブロック対数形式として、仮数部分は 5 ビット、符号いれて 6 ビットで合計 36+8=42 ビット、質量と合わせて 61 ビット。インデックスに 32 ビット、、、使いたくないけど、、、使うことにすると、全部で大体 192 ビット。ソフトニングもあるか。インデックスを 16 ビットにして、ソフトニングも 16 ビットですますことにしよう。

テーブルをなるべく小さくする、ということが重要になると思われる。そうすると、ポテンシャルが $g(r)$ だと、結局その 3 階導関数までがあれば必要なものは計算できる。この意味では、 $g(r)$ の計算自体にも 3 階導関数を使うのが自然と思われる。ポテンシャルの計算自体は対数側で補間計算をする。カットオフ関数が独立変数を対数にとった時にコンパクトサポートになるわけではなく、一般には 0 の側か無限大の側かの一方にしかカットオフがないので、

補間テーブルが $r \to 0$ のほうで無限に容量が必要になるのをどうするか、という問題がある。また、外側でも同様の問題がある。

テーブルの仕様として、まず、テーブルを使う区間の最大値と最小値を与え、その内側及び外側では、純粋な $1/r$ ポテンシャルまたは 0 を選択できるようにする。で、テーブルの範囲については独立変数は距離の対数のままで区分多項式近似をし、出力は正規化するかテーブルで対数形式に戻す。数値微分ができるようにすると精度が余計に必要なので、ポテンシャルと力は別にテーブルを持つ。4 重極に必要な項は力の 2 階導関数までから全て計算可能なはずなので、これはテーブルにしないで計算することで必要なテーブルを抑える。そうすると、2 次補間、32 エントリで十分。指数部に 8 ビット、仮数の最初に 10 ビット、あとの 2 つに 7、4 ビットとして、32 ビットに 1 つを収める。そうすると、64 ビット 32 エントリなので、レジスタファイルなら 4 個で収まる。

bRAM が 72 ビット出力できるなら、これ 1 つですますこともできる。こっちを使うと 512 エントリもあるテーブルができてしまう。それを利用してなんか考えたほうがよい？パイプライン 200 本程度だとすると、100 本くらいにはこっちを使って、残り 100 本には

フリップフロップは 44 万個ある。i 粒子データ、計算結果で大体 300 ビット。パイプライン 1000 本とかでもまだ半分あまるので、これは結構全大丈夫となる。

Chapter 4

詳細仕様

プロセッサは中心力計算パイプラインを最大 1024 個 (実際の数は利用可能なチップに収容できる数に依存する) 集積し、それらをを制御するための制御回路、計算に必要なデータを供給するデータ入力ポート、結果を出力する出力ポートをもつ。

今回、設計期間を短縮するため、パイプライン本数の、カットオフ関数以外のところは川井さんの pgpg2 ベースのものを使う。これの原型は

```
parameter ARCH         "GRAPE7M100";  // targetting hardware.
parameter NPIPE        4;             // number of pipelines to generate.
parameter JMEMSIZE     2048;          // j-particle memory depth.
parameter PREFIX       "g5";          // prefix for library functions.

int32      xj[3]        jpin;
int32      xi[3]        ipin;
int64      a[3]         fout;
log17.8    mj           jpin;
log17.8    eta          ipin;
float18.9  eps2         ipin;

dx  = (float18.9)(xj - xi);
r2  = (log17.8)(dx * dx + eps2);
r   = pg_sqrt(r2);
r3  = r2 * r;
mr3 = (float18.9)(pg_cutoff(r,eta)*mj / r3);
f   = mr3 * dx;
a  += (int64)(f << 75);
```

である。これに

- ポテンシャルを加える
- カットオフ関数を置き換える
- ネイバーリスト回路をつける
- 対数表現を廃止して浮動小数点に

という変更を加える。

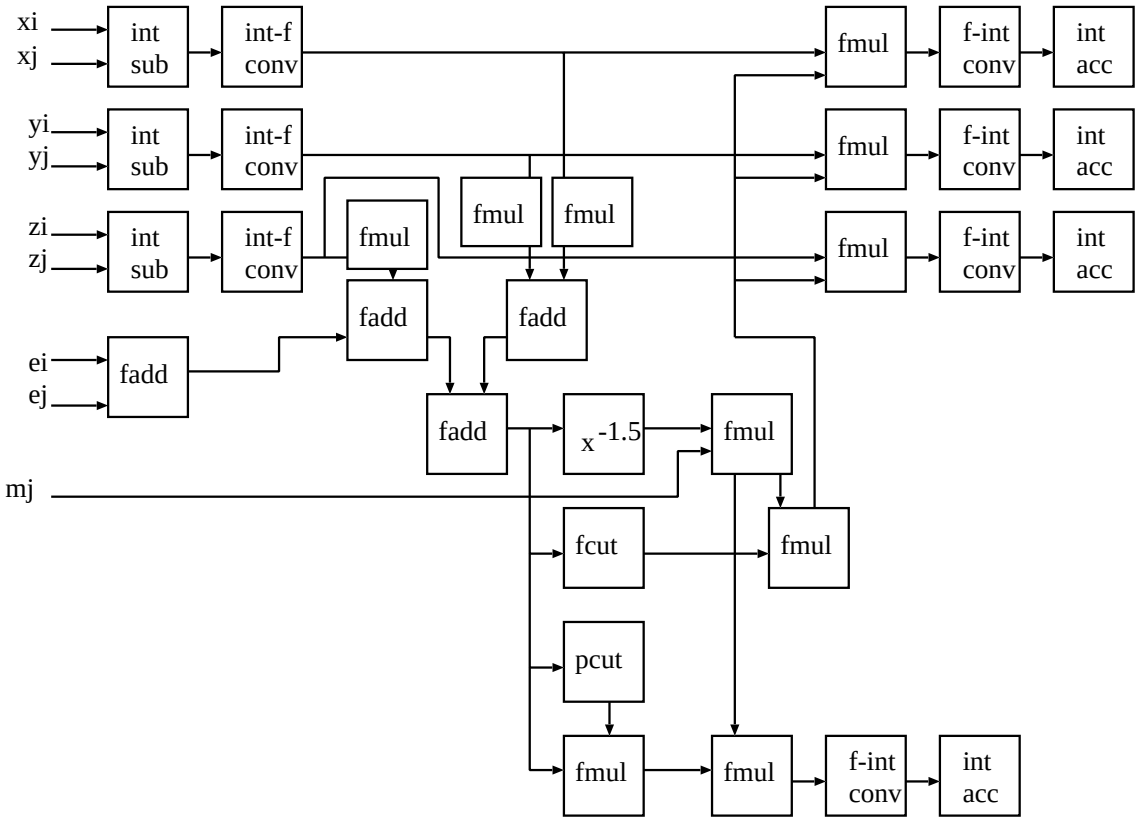

Figure 4.1: パイプラインの構成

4.1 中心力計算パイプライン

中心力計算パイプライン (以下、計算パイプライン）は、 GRAPE-8 プロセッサのうち粒子間相互作用の計算を担当する。具体的には、以下の式で与えられる粒子間相互作用をハードワイヤー化したパイプラインであり、各サイクル毎にパイプライン 1 本当り 1 つの粒子への相互作用を計算する。

$$\mathbf{a}_i = \sum_j G m_j g_f(r_{ij}/r_0) \frac{\mathbf{r}_{ij}}{(r_{ij}^2 + \epsilon_i^2 + \epsilon_j^2)^{3/2}} \tag{4.1}$$

$$\phi_i = \sum_j G m_j g_\phi(r_{ij}/r_0) \frac{1}{(r_{ij}^2 + \epsilon_i^2 + \epsilon_j^2)^{1/2}} \tag{4.2}$$

ここで

$$\mathbf{r}_{ij} = \mathbf{x}_j - \mathbf{x}_i \tag{4.3}$$

であり、 $\mathbf{x}_i$ は粒子 i の位置、 G は重力定数、m_j は粒子 j の質量である。また、 ϵ はソフトニングパラメータと呼ばれる定数であり、 $g(r)$ は単調減少なカットオフ関数、 r_0 は外場から与えるスケール長である。

粒子間重力の計算精度は、相対誤差が 2 乗平均の平方根で 1% 以下、平均値で 0.01% 以下を要求仕様とする。ここでの相対誤差は、カットオフ関数がガウシアンである時に、カットオフがない力の絶

対値で規格化したものと定義する。入力座標は 32 ビット固定小数点であり、加速度、ポテンシャルの積算は 56 ビット固定小数点で行う。

今回の設計では実パイプラインの数は 100 前後になるため、結果を縮約する回路は不要である。このため、論理的には粒子メモリは 1 つとし、全パイプラインが同じ j 粒子からの力を計算する。このため、仮想パイプライン構成はとらないことでパイプライン毎のレジスタを節約する。

アプリケーションの性能を上げるためには、 i および j 粒子データの書き込み、結果の書き出しを、計算となるべく並行して行う必要がある。このため、これらのためのバッファメモリを設け、計算は、

1. i 粒子バッファメモリから各パイプラインの入力レジスタへのデータロード
2. 実際の計算
3. アキュムレータから結果バッファメモリへの書き込み

がシーケンスになって実行されるようにする。これらと、

- i, j 粒子バッファへの書き込み
- 結果バッファメモリの読み出し

が非同期に実行できるようにすることで、可能な限り実行効率を上げる。

Nextreme2 の場合、 レジスタや小規模メモリはかなり貴重な資源であり、それに対して block ram は余裕がある。このため、i 粒子のバッファ、結果バッファともに、ある程度の幅ではあるが block ram を使うほうが現実的なような気がする。

パイプラインが 100 本くらいのような気がする。i 粒子は 1 つ 128 ビット。答は 64 ビットとして 256 ビット。

j 粒子メモリをパイプライン毎につけると幅が不足する。これは、複数のパイプラインでシェアすればよいので問題ではない。論理的にパイプライン数がどうなってないといけないかを考えてみると、計算は 500MHz で回ってパイプライン 100 本で 50Ginteractions/sec である。j 粒子の数が 1000 個くらいで性能をだしたいとしよう。さらに、通信速度が 2GB/s 程度を想定する。そうすると、 j 粒子のデータ量は 16kB なので、転送に 16 マイクロ秒かかる。この間に計算する相互作用の数は 0.8M interations になってしまうので、i 粒子の数が 800 と大きくなり過ぎることがわかる。妥協して j 粒子の数を 3000 くらいとすると、 i 粒子の数は 200 となって j 粒子に比べて十分小さい。この時には、100 マイクロ秒で 500 粒子なので、チップ 1 つで 5×10^6 粒子/秒になる。

この時、計算自体には 20 マイクロ秒くらいかかる。十分な効率を得るためには 1 マイクロ秒くらいで i 粒子データと結果のパイプラインへの load/store をしたい。これは 500 サイクルもあるということである。バッファメモリブロックが 16 個 (32 ビット幅として) くらいあると、パイプラインが 256 個としても 64 サイクルで書き込みができる計算なので、この時間とハードウェア量は全く問題にならない。

一つ考えるべきことは、 j 粒子メモリを複数ブロック持つかどうかである。i 粒子の数は少ないので、書き込み・読み出しに時間がかかったとしても複数に分けるほうが有利な状況がありえる。メモリ量は余っているので。

あ、ネイバーリストをつけるかどうかはどうしよう？ nearlest neighbour はあると嬉しい。G6 のようなネイバーリストは、あれば使うわけではあるが、、、速度を考えた時に CPU でやるのと比べたメリットはあるか？結果転送速度はノード当り 8GB/s が限界なので、32 ビットで返すとして 2G ネイバー/秒。50 ネイバーとして 4×10^7 粒子が処理できるので、実はやっぱりこっちが圧倒的に速い？では、やっぱりネイバーリストをつける。16 パイプラインで共有で 1024 ワードとか。

これは bRAM 1 つでできる。書き込み、読出し回路は適当に。

読出しは、語数がわからないのをどうするか？という問題がある。まあ、なにも考えずにバーストで出してしまってもいいが、そうするとまた色々不便なので、外から wait 信号をつけることができるようにする。

4.2 入出力ポート

入出力ポートは、双方向 1GB/s 以上の転送速度をもつものとする (内部 500MHz で動いた時に)。つまり、 1 クロックあたり 2 バイトとする。このため、ポート幅 32 ビット、データレートは内部クロックの 1/2 とする。

信号レベルはシングルエンドの LVCMOS 1.8V または 1.5V とする。但し、これは出力インピーダンスマッチングができるかどうかによる。

実際のチップの I/O 信号は以下の通りである。

CLKIN		In	データ入力クロック
CLK		In	コアクロック
WDATA	[0:35]	In	データ入力
WE		In	データ書き込みストローブ
WAEN		In	書き込みアドレスライトストローブ
RAEN		In	読出しアドレスライトストローブ
RREQ		In	読出しリクエスト
RDATA	[0:35]	Out	データ出力
RVALID		Out	データ出力ストローブ
RST		In	リセット
TESTMODE		In	テストモード
STATUS	[0:7]	Out	ステータス出力

データ、アドレスはマルチプレクスし、 WDATA ポートから入力する。 入出力とも奇数パリティであるが、パリティ位置は 32-35 であり、順番に下位バイトから上位バイトに対応する。つまり

パリティ	データ
32	0-7
33	8-15
34	16-23
35	24-31

となる。

なお、制御信号は全て正論理である。

WAEN をアサートしたサイクルで、書き込みアドレスがセットされる。 WE をアサートしたサイクルでデータがロードされる。また、この時にアドレスは自動インクリメントされる。

後で述べる j 粒子メモリ領域では、アドレスカウンタは下位 3 ビットが 6,7 をスキップする。これにより、制御チップからは連続データを送って j 粒子メモリに書き込むことができる。

RAEN をアサートしたサイクルで、読み出しアドレスがセットされる。 RREQ をアサートすることでデータがリクエストされる。リードの場合も、ライトの場合と同様にアドレスは自動インクリメントされる。なお、リードとライトで別々にアドレスカウンタをもつ。

リクエストからデータがでるまでの遅延は定義しない。 G8 チップは、RVALID シグナルによって、有効なデータがでていることを示す。

ステータスは以下の通り

ビット位置	名称	説明
0	CALC	計算中
1	IBBUSY	i 粒子バッファからパイプラインに転送中
2	RBBUSY	パイプラインから結果バッファに転送中
3	PERROR	入力パリティエラー

リセット信号は、入力パリティエラー信号をリセットする。今のところそれ以外の機能はない。

テストモードは、アサートすると入力を出力にループバックする。この時には、パリティはみないで、パリティビット位置で入力制御信号を出力する。対応は以下の通りである。

ビット位置	信号
32	WE
33	WAEN
34	RAEN
35	RREQ

4.3 メモリマップ

ここでは、チップの外からみたメモリマップについて述べる。

まず、チップに書き込まれるデータには以下の種類がある

i 粒子データ
j 粒子データ
テーブルデータ
その他コマンド・パラメータレジスタ

i 粒子データ、 j 粒子データは、それぞれブロック ram で構成されるバッファメモリエリアに書き込まれる。それぞれバッファを 2 つ持ち、計算の時にどっちを使うかはコマンドで指定する。テーブルデータは全パイプラインに放送される。

一つの i 粒子データは以下で表現される

変数名	語長 (ビット)
xi	32
yi	32
zi	32
id	32
eta	18
eps2	18
h2	18
合計	182

一つの j 粒子データは以下で表現される

変数名	語長 (ビット)

xi	32
yi	32
zi	32
m	18
eps	18
id	32
h2	18
合計	182

テーブルデータは幅 32 ビット、16 エントリである。これについては別に述べる。

j 粒子は 32 ビット 6 ワードに収まる。データ書き込みは 32 ビットワードでやるので、1 粒子は 6 ワードになる。

i 粒子は、ブロック毎に 1KW のバッファをもつ。j 粒子はアドレスマップとしては 128K 語とする。j 粒子 1 つは 6 語であり、これを 64 ビット幅のメモリ 3 個で実現する。これらは下位アドレスでセレクトする。このため、下位 3 ビットのアドレスが 6 及び 7 はメモリホールになっている。

アドレスマップは以下の通りである。

書き込みアドレス	説明
00000h-0003fh	パラメータレジスタ
00040h-0007fh	関数テーブル
04000h-043ffh	i 粒子バッファメモリ 0
04400h-047ffh	i 粒子バッファメモリ 1
07c00h-07fffh	i 粒子バッファメモリ 15 (?)
20000h-3ffffh	j 粒子メモリ

読み出しアドレス

00000h-003ffh	F0 バッファメモリ 0
00800h	ネイバーリスト長 0
00C00h-00fffh	ネイバーリスト 0
01000h-013ffh	F0 バッファメモリ 1
01800h	ネイバーリスト長 1
01C00h-01fffh	ネイバーリスト 1
0f000h-0f3ffh	F0 バッファメモリ 15
0f800h	ネイバーリスト長 15
0fC00h-0ffffh	ネイバーリスト 15

ここまでアップデートした。 2010/8/15

コマンドストローブがアサートされた時の入力データがそれ以降のデータの開始アドレス等を指定する。これは、 LSB をビット 0 として以下のビットフィールドを持つ

ビット位置	説明
0-19	開始アドレス
20-37	バースト長

パラメータレジスタは以下のアドレスマップを持つ

アドレス	名称	説明
00h	RUN	ここへの書き込みが計算開始となる

01h	ICSTART	i 粒子バッファからパイプラインブロックへコピー開始
02h	FCSTART	パイプラインブロックから fo 結果バッファへコピー開始

RUN レジスタは

ビット位置	説明
0-15	j 粒子数
16-31	開始アドレス (粒子番号)

である。

ICSTART と FCSTART は、フィールドの構成は同じでどちらも

ビット位置	説明
0-9	アドレス
16-25	ワード数 (粒子数ではないみたい)

となっている。

Chapter 5

詳細仕様

5.1 パイプライン

パイプラインは、以下のモジュールからなる

- 距離差計算ユニット (DD)
- 二乗距離計算ユニット (DS)
- 最近接粒子ユニット (NNB)
- ネイバーリストユニット (NBL)
- 関数補間ユニット (FIQ)
- 積算ユニット (ACC)

以下、順番に機能を説明する。まず、入力変数は

```
int32      xj[3]        jpin;
int32      jid          jpin;
float18.9    mj           jpin;
float18.9  eps2j        jpin;
float18.9  h2j          jpin;

int32      xi[3]        ipin;
int32      iid          ipin;
log17.8    eta          ipin;
float18.9  eps2         ipin;
float18.9  h2           ipin;
```

である。これは pg2 表記である。出力は

```
int64      a[3]         fout;
int64      p            fout;
int64      nnbinfo      fout;
```

の他にネイバーリストとなる。 a, p は加速度、ポテンシャルで、 nnbinfo は最近接粒子の情報である。

pg2 でのパイプライン定義は以下の通り

```
entity pipe is
    generic(JDATA_WIDTH    : integer;
            PIPELINE_DELAY : integer
    );
    port(p_jdata  : in  std_logic_vector(JDATA_WIDTH-1 downto 0);
         p_run    : in  std_logic;
         p_we     : in  std_logic;
         p_adri   : in  std_logic_vector(3+4 downto 0);
         p_adrivp : in  std_logic_vector(3 downto 0);
         p_datai  : in  std_logic_vector(63 downto 0);
         p_adro   : in  std_logic_vector(3+4 downto 0);
         p_adrovp : in  std_logic_vector(3 downto 0);
         p_datao  : out std_logic_vector(63 downto 0);
         p_runret : out std_logic;
         p_bcast  : in  std_logic;
         p_cid    : in  std_logic_vector(2 downto 0);
         rst      : in  std_logic;
         pclk     : in  std_logic
    );
end pipe;
```

vp って使ってるのかな？というか使ってない。アドレスも、入力 2 ビット、出力 3 ビット。少し整理すると

```
entity pipe is
    generic(JDATA_WIDTH    : integer;
            PIPELINE_DELAY : integer
    );
    port(p_jdata  : in  std_logic_vector(JDATA_WIDTH-1 downto 0);
         p_run    : in  std_logic;
         p_we     : in  std_logic;
         p_adri   : in  std_logic_vector(1 downto 0);
         p_datai  : in  std_logic_vector(63 downto 0);
         p_adro   : in  std_logic_vector(2 downto 0);
         p_datao  : out std_logic_vector(63 downto 0);
         pclk     : in  std_logic
    );
end pipe;
```

これに付け加えるべきインターフェース:ネイバーフラグ。

```
entity pipe is
    generic(JDATA_WIDTH    : integer;
            PIPELINE_DELAY : integer
    );
    port(p_jdata  : in  std_logic_vector(JDATA_WIDTH-1 downto 0);
         p_run    : in  std_logic;
```

```
          p_we     : in  std_logic;
          p_adri   : in  std_logic_vector(1 downto 0);
          p_datai  : in  std_logic_vector(63 downto 0);
          p_adro   : in  std_logic_vector(2 downto 0);
          p_datao  : out std_logic_vector(63 downto 0);
          p_nbf    : out std_logic;
          pclk     : in  std_logic
     );
 end pipe;
```

で、以下のユニットを追加するので:

```
   component  cutoff
       port (r     : in  std_logic_vector(17 downto 0);
             eta   : in  std_logic_vector(17 downto 0);
             addr  : in  std_logic_vector(4 downto 0);
             data  : in  std_logic_vector(31 downto 0);
             wen   : in  std_logic;
             scale : out std_logic_vector(17 downto 0);
             clk  : in  std_logic);
   end component;

   component nnb
       port(jid         : in  std_logic_vector(31 downto 0);
            iid         : in  std_logic_vector(31 downto 0);
            run         : in  std_logic;
            r2          : in  std_logic_vector(17 downto 0);
            result      : out std_logic_vector(63 downto 0);
            clk         : in  std_logic
            );
   end component;
```

- cutoff テーブルの書き込みインターフェースをつける必要あり
- nnb の結果に出力アドレス割り当てる必要あり

cutoff テーブルは、 i 粒子データとは別のインターフェースで書くことにする。従って

```
entity g8pipe is
   generic(JDATA_WIDTH    : integer := 182;
           PIPELINE_DELAY : integer := 40
           );
   port(p_jdata  : in  std_logic_vector(JDATA_WIDTH-1 downto 0);
        pclk     : in  std_logic;
        p_run    : in  std_logic;
        p_we     : in  std_logic;
        p_adri   : in  std_logic_vector(2 downto 0);
        p_datai  : in  std_logic_vector(31 downto 0);
        p_adro   : in  std_logic_vector(3 downto 0);
        cutoff_addr  : in  std_logic_vector(5 downto 0);
        cutoff_data  : in  std_logic_vector(31 downto 0);
        cutoff_wen   : in  std_logic;
```

```
        p_datao  : out std_logic_vector(31 downto 0);
        p_nbf    : out std_logic
        );
end g8pipe;
```

とする。 カットオフテーブルは2つあるのでアドレス1ビット増やす。元々のカットオフは

```
   component pg_cutoff_log_log_log_17_17_17
     port (r    : in  std_logic_vector(16 downto 0);
           eta  : in  std_logic_vector(16 downto 0);
           ginv : out std_logic_vector(16 downto 0);
           clk  : in  std_logic
     );
   end component;
```

で、これを

```
entity cutoff is
port (r     : in  std_logic_vector(17 downto 0);
      addr  : in  std_logic_vector(4 downto 0);
      data  : in  std_logic_vector(31 downto 0);
      wen   : in  std_logic;
      scale : out std_logic_vector(17 downto 0);
      clk   : in  std_logic);
end cutoff;
```

これは書いてみた。
パイプライン段数があってない気が、、、
まあ、とりあえずユニットだけいれこんでみた。
一応、書いた。

5.1.1 アドレスマップ

g8pipe としてのアドレスマップをまとめておく。
i 粒子入力

```
   p_adri   : in  std_logic_vector(2 downto 0);
   p_datai  : in  std_logic_vector(31 downto 0);
   address    data
         0    xi
 1    yi
 2    zi
 3    iid
 4    0:17   eta
      31:18  eps2(0:13)
 5    0:3    eps2(14:17)
      4:21   h2
```

出力

```
    p_adro   : in  std_logic_vector(3 downto 0);
    p_datao  : out std_logic_vector(31 downto 0);
    address    data
        0x0    ax low

0x1    ax high
        0x2    ay low
0x3    ay high
        0x4    az low
0x5    az high
        0x6    pp low
0x7    pp high
        0x8    nnb low (nnb index)
0x9    0:17  nnb high (r2nnb)
       18:31 unused
```

カットオフテーブル

```
    cutoff_addr  : in  std_logic_vector(5 downto 0);
    cutoff_data  : in  std_logic_vector(31 downto 0);
    address    data
    0x00-0x0f  table for f cutoff
         0x10  flag for value outside [0:1) (0:0, 1:1)
    0x20-0x2f  table for p cutoff
         0x30  flag for value outside [0:1) (0:0, 1:1)
```

5.1.2 距離差計算ユニット (DD)

DD は、自分が持つ i 粒子座標と、外部 (オンチップ) メモリから供給される j 粒子座標の差を計算し、対数フォーマットに変換する。入力座標は 32 ビットであり、出力は 「仮数」部分が 9 ビットの浮動小数点フォーマットである。仮数以外は、符号ビットが 1 ビット、指数部分が 7 ビットとなる。これは実際には pgpg2 の

```
  dx  = (float18.9)(xj - xi);
```

で実装される。

5.1.3 二乗距離計算ユニット (DS)

実装は

```
  r2f  = dx * dx + eps2+eps2j;
```

これは変更して、 r2 への変換はせず、浮動小数点表現のまま使うことにする。

5.1.4 -1.5 乗計算ユニット (POW32)

対数表現を使わないことにしたので、逆数平方根または -1.5 乗を計算する回路が必要になる。今回は -1.5 乗を計算することにする。このほうが乗算器が 1 つ少なくてすむからである。

で、それからカットオフをかけることになる。

定数との乗算、、、これは避ける？定義域を [1/2,2) に規格化してよいはず。入力は、

指数 8 ビット
仮数 9 ビット

これの、指数の最下位ビット+仮数 9 ビットの内訳、仮数上位 4 ビットまでをテーブルに入れる。上の規格化だと、0.35 から 2.8 まで。指数は -1, 0, 1, 2 の 4 種類。なので、2 ビットとっておけばよい。線形補間を想定するので、

メモ: 2 次のチェビシェフ多項式は $2x^2 - 1$ なので、0 点は $\pm\sqrt{1/2}$

テーブル入力は 5 ビット、出力は、指数 2 ビット、0 次の項 10 ビット、1 次の項は 7 ビット必要かな？これを下位の 5 ビットと乗算して、11 ビットの答の 5 ビットを切り捨てる。

これはシミュレータできた。結局、 $x^{-1.5}$ を計算するもの。

5.1.5 最近接粒子ユニット (NNB)

これは、以下の入力をとる

JID	[0:31]	In	J 粒子インデックス
IID	[0:31]	In	I 粒子インデックス
R2	[0:17]	In	距離
RUN		In	計算中信号

出力は

RESULT	[0:63]	Out	最近接粒子データ。

フォーマットは、上位 32 ビットの一部に距離、下位にインデックスとする。VHDL では一応こんな感じ

```
entity nnb is
    port(
        jid         : in  std_logic_vector(31 downto 0);
        iid         : in  std_logic_vector(31 downto 0);
        run         : in  std_logic;
        r2          : in  std_logic_vector(17 downto 0);
        result      : out std_logic_vector(63 downto 0);
        clk         : in  std_logic
        );
end nnb;
```

5.1.6 ネイバーリストユニット (NBL)

これは、以下の入力をとる

```
FLAGS   [0:15]   In    ネイバーフラグ
RUN              In    計算中信号
ADDR    [0:9]    In    読み出しアドレス
```

出力は

```
RESULT  [0:63]   Out   ネイバーリストデータ
```

ネイバーリストメモリは 1024 語、32 ビット (bRAM 1 つ) とする。
出力のアドレスマップは

```
ADDR
0x000          ネイバー数。オーバーフローしていても数は正しいとする。
0x0200-0x3ff   ネイバーデータ。上位 16 ビットはフラグ、下位 16 ビットは
               カウンタ出力。2 データを 1 語にパック。
```

とする。

5.1.7 関数補間ユニット (FIQ)

ここをどうするかが主な問題ではある。

16 エントリーの 2 次補間で、ここには基本的にレジスタファイルを使う方向。

レジスタファイルは 640 個しかない。

なので、以下のような補間が理想？

0 - r0: 等間隔 8 エントリ r0-r1 : 等間隔 8 エントリ

まず、普通の P3M なら単純に 16 エントリ使える。逆に、普通の Oshino スキームならやはり 16 エントリ。 P3M + Oshino の時にこれでいいか？あと、回路としては？ r0 が r1 よりずっと小さい時にどういう動作をするべきか？

ソフトニングと組合せてごまかす？そっちのほうが現実的だな。

なので、あまり難しいことはしないで整数化してすます。

そうすると、補間部分への入力は、整数化した 9 ビット。この上位 4 ビットがテーブルへの入力、テーブルは、

- 指数 e 本来不要？ 4 ビットつけておく。一応両側。
- 0 次仮数 m0 9 ビット
- 1 次仮数 m1 6 +1 ビット
- 2 次仮数 m2 4 +1 ビット

がでてきて、1 次、2 次仮数は符号ビットが別にあって加算。

(m2 * dx+m1)*dx + m0

川井君の pg2 は衝撃的なことに cutoff は emu 側では実装されてないっぽい。おい。
とりあえず、VHDL 側を書いてみる。
補間器としては

```
R       [0:17]  In    距離入力 f18.9 形式
ETA     [0:17]  In    最大距離の逆数 f18.9 形式
ADDR    [0:4]   In    テーブル書き込みアドレス
DATA    [0:31]  In    テーブル書き込みデータ
WEN             In    テーブル書き込みイネーブル

ADDR   0x0-0xF: 補間テーブル
       0x10:    外側の係数。1 または 0
```

出力は

```
SCALE   [0:17]   Out  スケール出力  f18.9 形式
```

補間ユニット自体:

```
R       [0:8]   In    距離入力 整数
E       [0:3]   In    指数
M0      [0:9]   In    0 次係数
M1      [0:6]   In    1 次係数
M2      [0:3]   In    2 次係数
S1              In    1 次係数符号
S2              In    2 次係数符号
```

整数への変換

```
R       [0:17]  In    距離入力 f18.9 形式
ETA     [0:17]  In    逆数最大距離 f18.9 形式
ROUT    [0:8]   Out   整数化距離
OVL             Out   R>ETA なら 1
```

floattoscaledint は作った。
次は多項式補間
書くだけは書いた。
cutoff の実装詳細。floattoscaledint によって結果を整数化。 floattoscaledint の動作は

```
入力が 1 より大きいなら ovl=1 そうでなければ ovl=0
シフト量が 8 以上なら 0 にする
```

この結果を polynomial ユニットに入れる。
polynomial.vhd の動作は以下のような感じ

```
entity polynomial is
port (r    : in  std_logic_vector(9 downto 0);
      m0   : in  std_logic_vector(9 downto 0);
      m1   : in  std_logic_vector(7 downto 0);
      m2   : in  std_logic_vector(5 downto 0);
      s1   : in  std_logic;
      s2   : in  std_logic;
      e0   : in  std_logic_vector(5 downto 0);
      dst  : out std_logic_vector(8 downto 0);
      exp  : out std_logic_vector(7 downto 0);
      clk  : in  std_logic);
end polynomial;
```

r の 4-1 ビットと m2 をかけた結果: 10 ビット。この上位 7 ビットをとる。
ここでは丸めは、、、まずは切り捨てにしてみる。
これを s2 の符号によって、m1 と加減算。上位 2 ビットは 0 で埋める。
この結果の 6-1 ビットを r の 4-0 ビットと掛ける
この結果の 10-4 ビットを force1 丸めをしてとる
これを m0 と加減算する。
この結果の 9-1 ビットを、 force1 まるめをしてとる
上のゼロを数える。
シフトしないといけないがまだしてない気がする、、、、

まず、16 エントリーで [0-1] の関数補間を浮動小数点でやるライブラリを作る。それから、それを整数化というか g8 フォーマット化する。

関数補間ライブラリは作った。一応 3 次チェビシェフ多項式の 0 点をサンプル点に使って 2 次補間する。

これを、g8 内部のフローティング形式での表に変換する。

方針:入力の内部表現は、普通に hidden bit がついてくる。

指数は 100... (MSB のみ 1) が 1 に相当

係数は、 0-2 次のどの係数も 0-1 の間。ビット位置が

```
m0  9876543210
m1     6543210
m2        3210
```

で、3 ビットづつ小さくなるので、そうなっていないとまずい。DLL function の表は

```
 8         0.9982189479        0.03391478264        -0.1194561101
 9         0.9135282803        -0.2337167299       -0.06513912647

10         0.6168039556        -0.3558489936        0.03981256097
11         0.3013819653        -0.2659774961        0.06823455323
12          0.103192763        -0.1260529814        0.04546447922
13        0.02201995288       -0.03647884242        0.01688697089
14       0.002131521373      -0.004840262539       0.002802143829
15     3.047210956e-05     -9.105055568e-05     6.258291089e-05
```

で、結構殆どの領域でそうなってないので規格化はちょっと複雑。

1. まず、区間両端での値を計算、これからスケールの候補を 1 つ決める
2. 次に、 1 次 (1/4) 2 次 (1/16) の係数の制約からスケールの候補を決めて
3. これらの候補に最大値がスケールの基準値になる。

つまり、

```
max (c0, c0+c1+c2, c1*8, c2*64)
```

を超える最小の 2^k ということ。

で、それを基準に 10 ビットで整数化。

フォーマットに対数を使わないことにして、全部浮動小数点でやる。

この場合、 r はもってないので、距離の 2 乗をそのまま補間テーブルに入れる。例えば DLL 関数の場合にはこれで問題ない。

そのため、逆数平方根の回路をつける。そのため、レジスタファイルを使う。32 エントリで 10+6 ビットで十分じゃないかな？

5.1.8 積算ユニット (ACC)

5.2 ブロックパイプライン

パイプラインの上位に、16 個のパイプライン+ネイバーフラグユニットのモジュールを置く。これをブロックパイプラインと呼ぶ。

このレベルで、データバッファリングを行う。つまり、 i 粒子の書き込み、結果の書き出しのためのバッファを用意する。このため、書き込み、読出しはこれらの RAM へのランダムアクセスとなる。

これらの RAM はデュアルポートとして、パイプライン側との転送を制御するコマンドを設ける。これは指定したアドレスから連続アクセスのみ。

i 粒子データは 6 語だが、アドレスはこれに 2 ビット使うので、こっちのアドレスカウンタは 10 の時に 0 に戻る (+2 する) カウンタにする。

結果データは 10 語で、アドレスはこれに 4 ビット使うので、こっちのアドレスカウンタは 100 の時に 0 に戻る (+4 する) カウンタにする。

バッファは大きくして、物理的な i 粒子の計算単位より多数おけるようにする。

バッファは 32 bit 幅、1024 語とする。書き込み、読出しはこのレベルではアドレスいれてアクセスするだけ。

転送は、コマンドレジスタへの書き込みで行う。コマンドは読出し、書き込み別々に用意。それぞれ

- 開始アドレス (メモリのみ)
- 転送語数

を設定。そうすると、インターフェースは以下のような感じ

```
entity g8block is
    generic(JDATA_WIDTH    : integer;
            PIPELINE_DELAY : integer
    );
    port(p_jdata  : in  std_logic_vector(JDATA_WIDTH-1 downto 0);
         pclk     : in  std_logic;
         run      : in  std_logic;
         waddr    : in  std_logic_vector(8 downto 0);
         wdata    : in  std_logic_vector(63 downto 0);
         we       : in  std_logic;
         raddr    : in  std_logic_vector(10 downto 0);
         rdata    : out std_logic_vector(63 downto 0);
 wtrig   : in  std_logic;
         wtaddr   : in  std_logic_vector(8 downto 0);
         wtlen    : in  std_logic_vector(8 downto 0);
 rtrig   : in  std_logic;
         rtaddr   : in  std_logic_vector(8 downto 0);
         rtlen    : in  std_logic_vector(8 downto 0);
         cutoff_addr   : in  std_logic_vector(5 downto 0);
         cutoff_data   : in  std_logic_vector(31 downto 0);
         cutoff_wen    : in  std_logic;
    );
end g8block;
```

read buffer の上位半分はネイバーリストメモリに書かれた語数、さらにその上位はネイバーリストメモリとする。ネイバーリストは 1024 語、つまり、アドレスマップとしては

```
0x000-0x1ff 結果バッファ
0x200-0x3ff ネイバーリスト語数 (0x200 以外はなんでもよい)
0x400-0x7ff ネイバーリストデータ
```

メモリ

5.3 J粒子メモリユニット

これは、

- 64 ビット単位で書き込みができる
- 160 ビットか何か、パイプラインが要求する幅でデータ出す

機能があればよい。計算中に別のところに書くことを可能にするため、開始アドレスを指定して計算開始するようにする。これは 4Mbits くらい使うとして、j 粒子データは 192 ビットなので、2 万粒子、64 ビットワードアドレスは 16 ビット、粒子数は 15 ビットで指定できる。

```
entity jdata_mem is
    port(pclk     : in  std_logic;
         waddr    : in  std_logic_vector(15 downto 0);
         wdata    : in  std_logic_vector(63 downto 0);
         we       : in  std_logic;
```

```
          run    : in  std_logic;
   astart : in  std_logic_vector(15 downto 0);
   jdata  : out  std_logic_vector(JDATA_WIDTH-1 downto 0);
      );
  end jdata_mem;
```

これ、192 ビットのメモリをどうやって実現する？64 ビットのメモリを 3 個並べるとして、単純に下位 2 ビットでモジュール選択して 11 は穴にすればいいか。

5.4 トップレベル回路

トップレベル回路の外部インターフェースは、書き込みと読み出しだけ。計算開始等はアドレスを割り当てる。書き込みが必要なもの:

j 粒子メモリ
i 粒子バッファ
カットオフテーブル
N（=計算開始信号）

読み出しが必要なもの:

i 粒子結果バッファ

あれ、これだけ？読み出しアドレスはブロックが 16 個として 15 ビット

書き込みアドレスマップ アドレスは 18 ビット

```
0x00000          計算開始信号
0x00001          i 粒子バッファ->パイプライン転送開始信号
0x00002          パイプライン-> 結果バッファ転送開始信号
0x00040-0x0007f カットオフテーブル
0x02000-0x021ff i 粒子バッファ 0
0x02200-0x023ff i 粒子バッファ 1
0x02400-0x025ff i 粒子バッファ 2
0x02600-0x027ff i 粒子バッファ 3
....
0x03e00-0x03fff i 粒子バッファ 15
0x10000-0x1ffff j 粒子メモリ
```

i 粒子データは実際にはパイプラインあたり 3 ビットしかアドレスがないので、上のはちょっと過剰。

チップのピン数を減らしたいので、アドレス、データはマルチプレクスする。アドレスをロードしたあと、書き込み、読出し共にアドレスは自動インクリメントとする。インクリメントは無制限に可能とする。

読み出しアドレスマップ

```
0x0000-0x07ff    ブロック 0 の i 粒子結果バッファ
....
0x7800-0x7fff    ブロック 15 の i 粒子結果バッファ
```

```
entity g8_top is
    port(pclk     : in  std_logic;
         wdata  : in  std_logic_vector(31 downto 0);
         we     : in  std_logic;
         waen   : in  std_logic;
         raen   : in  std_logic;
         rreq   : in  std_logic;
         rdata  : out std_logic_vector(31 downto 0);
         rst    : in  std_logic;
 rvalid : out  std_logic;
         status : out std_logic_vector(15 downto 0)
    );
end g8_top;
```

WDATA	[0:35]	In	データ入力
WE		In	データ書き込みストローブ
WAEN		In	書き込みアドレスライトストローブ
RAEN		In	読出しアドレスライトストローブ
RDATA	[0:35]	Out	データ出力
RVALID		Out	データ出力ストローブ
RST		In	リセット
STATUS	[0:7]	Out	ステータス出力

ステータスビットの意味は以下の通り

0: 計算中
1: バッファ -> パイプライン転送中
2: パイプライン -> バッファ 転送中
3: 入力データパリティエラー
4-7: 未定義

データは 8 ビット毎のバイトパリティ (奇数パリティ:1 が奇数個の時 0) をつける。パリティビットの場所は 32:35 とする。

Chapter 6

シミュレータ、エミュレータ等について

g8 用には、以下の 2 レベルのシミュレーションを作成する

1. 計算精度は普通の倍精度のままのもの
2. ビットレベルでパイプライン動作をエミュレーションし、結果を返すもの

以下では、前者をシミュレータ、後者をエミュレータと呼ぶ。

6.1 最上位インターフェース

これを始めから決めておくと話が速い。i 粒子転送、j 粒子転送、結果回収、計算の全部についてパイプライン的な動作が前提とすると、どういうインターフェースをもっているべきか？

基本的に multiwalk 型にするべき？

バッファリングとの関係をどうするかが問題。

lazy evaluation というわけにもいかないとすれば、単純に複数の interaction list と i 粒子グループを送って、結果を返してもらう。

それ用の構造体を作る。

```
struct g8task_struct{
   g8jparticle * jp;
   g8iparticle * ip;
   int njp;
   int nip;
}G8TASK, *G8TASKP;
```

みたいな。 f と ip で分離したほうがいいのかな？まあ、メインメモリアクセスを最小にするとかは、あとから考えればよかろう。

あと、これを非同期で、、、というのは、やるとすれば、、、、

- open/close
- カットオフ関数設定

- 上の関数

なので、必要と思われる関数は、とりあえず

```
int g8nunit();
```

g8 チップか何か、利用可能なユニットの数を返す。

```
int g8open(int unitid);
int g8close(int unitid);
```

それぞれのユニットの open/close。但し、 g8 の場合、これが返すのは物理的なデバイス、つまり、通信が並行して行えるユニットの数であって、j 粒子メモリを共有するブロックの数ではないとする。

```
void g8_setcutoff(int index, double (*func)(double));
```

カットオフ関数を指定。

```
index=0  力
index=1  ポテンシャル

int g8run(int ntask, G8TASKP taskp, int getnbl);
```

実際に計算して結果を返すところまでとりあえず全部する。ホストとの並行処理については、非同期インターフェースをつける？非同期インターフェースは、

```
int g8run_start(int ntask, G8TASKP taskp, int getnbl,
                &int expected_runtime);
int g8run_continue(int ntask, G8TASKP taskp, int getnbl,
                   &int expected_runtime);
```

g8run_continue の返り値が

```
1 まだ仕事が残っている
0 完了
<0 異常終了
```

とする。次に呼ぶまでの間隔は、「期待する次に呼ばれるまでの時間」を返すとする。単位はマイクロ秒。これは、シミュレータ、エミュレータではあまり意味がない。

ネイバーリストは、上位レベルインターフェースでは getnbl が 0 なら無視、1 なら返す。返す場所は i 粒子構造体にあるとする。粒子構造体は、 j のほうが

```
struct g8jparticle_struct{
    double xj[3];
    double mj;
    double eps2j;
    int jindex;
}
```

i のほうは

```
struct g8iparticle_struct{
    double xi[3];
    double eps2i;
    int iindex;
    double a[3];
    double p;
    double rnnb;
    int jnnb;
    int nnbl;
    int * nbl;
}
```

6.2 シミュレータ

ユーザーインターフェースはともかくとして、シミュレータのパイプライン部分のインターフェースは

void g8sim_pipeline(double xj[3], double mj, double eps2j, double h2j,

```
    int jindex,
    double xi[3], double eps2i, double h2i, double eta,
    int iindex,
    double a[3], double *p, double *rnnb, int * jnnb,
    int * nnbl, int * nbl, int run_start)
```

カットオフ関数を指定するインターフェースが必要。

```
 void g8sim_setcutoff(int index, double (*func)(double));
```

これだけあればいいのかな？

一応これを作って動くようにした。

6.3 エミュレータデバッグ用のインターフェース

g8 は g6 と違ってハードウェア側にスケール関数をもたない。えーと、これでいいんだっけ？ 32 ビットの引き算のあとの規格化は、、、浮動小数点形式の指数が必要以上に長いので、ここは考えなくても原理的にはかまわない。

なので、テスト用ルーチンでは、スケーリングを固定で与える。

エミュレータのスケーリングをメモのほうに整理しておく。

要点は、固定小数点の 1 が 指数 128、つまり内部表現の 1 に変換される、ということ。

座標については、パイプライン自体の単体テスト関数では、ホストでの浮動小数点の 1 を (1¡¡24) に変換するようにする。エミュレータのパイプライン本体のインターフェースは

```
 void g8_pipeline(UINT64 eps2, UINT64 h2, UINT64 iid, UINT64 eta,
UINT64 xi_0, UINT64 xi_1, UINT64 xi_2,
UINT64 h2j, UINT64 jid,
UINT64 xj_0, UINT64 xj_1, UINT64 xj_2,
UINT64 eps2j, UINT64 mj,
UINT64 UINT64 *a_0p, UINT64 *a_1p, UINT64 *a_2p,
UINT64 *nnb, UINT64 *pp,
UINT64 *nnbl, UINT64 *nbl,
int run_begin)
```

だが、これを

void g8emu_testpipeline(double xj[3], double mj, double eps2j, double h2j,

```
    int jindex,
    double xi[3], double eps2i, double h2i, double eta,
    int iindex,
    double a[3], double *p, double *rnnb, int * jnnb,
    int * nnbl, int * nbl, int run_start)
```

から呼べるようにする。

6.4 ブロック、チップのエミュレータインターフェース

ブロックは、

- xi を書く
- xj はサイクル毎に送る
- ネイバーリストはメモリから読む

感じ。なので、

npipe nblock

は変数として、

```
 void g8_block_singlestep(int chipid, int block_id, UINT64 h2j, UINT64 jid,
UINT64 xj_0, UINT64 xj_1, UINT64 xj_2,
UINT64 eps2j, UINT64 mj,
int run_begin)
```

がメインのインターフェース、

スケール設定

```
g8sim_set_scale(int scale_bit)
```

i 粒子設定

```
g8sim_set_i_particle(int chipid, int blockid, int pipeid,
              struct g8sim_iparticle * ip
```

// double xi[3], double eps2i, double h2i, double eta, // int iindex)
結果読出し

```
g8sim_get_result(int chipid, int blockid, int pipeid,
                  struct g8sim_result * resp)
```

ネイバーリスト読出し

```
g8sim_get_nbl(int chipid, int blockid,
       struct g8sim_nbl *resp)
```

があればいいかな？
そうすると、ついでに

```
g8sim_set_j_particle(int chipid, struct g8sim_jparticle * jp)
```

```
g8sim_run(int chipid, int nj)
```

これくらいあればいい？
g8_pipeline に、生のネイバーフラグがでるインターフェースの追加の必要が発生。
結局、

```
    void g8sim_set_scale(int scale_bit);

    void g8emu_set_i_particle(int chipid, int blockid, int pipeid,
      G8SIM_IPARTICLE * ip);
    void g8emu_get_result(int chipid, int blockid, int pipeid,
  G8SIM_RESULT * resp);
    void g8emu_get_nbl(int chipid, int blockid, G8SIM_NBL *resp);
    void g8emu_set_j_particle(int chipid, G8SIM_JPARTICLE * jp, int j);
    void g8emu_run(int chipid, int nj);
```

これだけのインターフェース関数を作った。
これらがどういうテストベクタを吐くべきか:

- カットオフ
- i 粒子データ
- j 粒子データ
- 計算開始信号
- 結果回収信号

Chapter 7

チップ入出力等定義

7.1 I/O 規格

SSTL or HSTL. やっぱり SSTL-18 で。Arria GX は SSTL だと 15 はサポートしない。

7.2 タイミング制約

クロック 200MHz (DDR 400MHz) として
入力

```
DCD max 150ps ( 2.5ns +- 150ps)
```

入力クロックに対する入力信号遅延

```
最大 1.00ns
最小 -0.100 ns
```

出力

SSTL-18 Class I の標準負荷で、入力クロックに対する遅延を X ns +- 500ps にする。 X は値が決まっていればよい。

Chapter 8

メモ

8.1 pg2 でのアキュムレータ動作

```
entity pg_inc_int_64_64 is
  port (src  : in  std_logic_vector(63 downto 0);
        dst  : out std_logic_vector(63 downto 0);
        run  : in  std_logic;
        clk  : in  std_logic);
end pg_inc_int_64_64;
```

run=1 の間にきたものが加算される。

8.2 eta

あれ、スケールは ij 対称化したい？

8.3 川井版 pg なんとかでの型変換

8.3.1 整数から浮動小数点

(src/pg2pkg/pgemu.c)

入力は

```
0x1 ->  sign:   0  exp:   128  (0x80)  man: 0x0  return: 0x10000
0x2 ->  sign:   0  exp:   129  (0x81)  man: 0x0  return: 0x10200
0x3 ->  sign:   0  exp:   129  (0x81)  man: 0x100 return: 0x10300
0x40000000 -> sign:   0  exp:   158  (0x9e)  man: 0x0 return: 0x13c00
```

なので、単純に整数の 1 が入ると指数が 128 になるように設計されている模様。

座標は本当に 32 ビットでいいか？というのはどうなんだろう？カットオフの上は 24 ビットになるので、7 桁。なので、カットオフをシステムサイズの 3 桁下にとってもまだ 4 桁あるので、まあ普通は大丈夫。

ちなみに、 10^7 体の球状星団の計算に PPPT スキーム (と呼ぼう) を使ったとして、カットオフが half mass の 4 桁下だとすると、平均粒子間距離に対してこれは 5/3 桁下。なので、等温カスプだと思うと 10%くらいの粒子にはネイバーがいて、ネイバーが 1000 を超えるような粒子が 0.3% くらいはある計算になる。内側の 1% くらいはネイバーリストをもたずにダイレクトでやって、その外側はネイバーリストをもたせるみたいな感じかな？

パイプライン自体のテスト関数では、ホストでの浮動小数点の 1 を (1¡¡24) に変換するようにする。

8.3.2 エミュレータデバッグ。

```
-1.5 乗が、
r^2
1-> 1/8
4-> 1/64
```

それ以前に 0.5 で計算できてない。これは引数がそもそも間違っていた。

pf まではそれらしく計算できている。シフト量が 0x4b = 75 元の位置が 2^-16 だった時に、ポテンシャルは 1/256 になっていて、補正シフトの後は 64 ビットを溢れてちょっと大き過ぎる。もうちょっと小さい数でテスト

1 をいれると pf は 2^-24

とりあえず、カットオフを使わない簡単なケースでは大丈夫みたい。epsi もいれてみた。epsj, h はまだ。

ernnb はおかしい？いや、これはインターフェースと、シミュレータのほうがおかしかった。

粒子数を 2 にした時の加算ができてない。

これは、呼び出し側で変数を保存していなかったため。

後はカットオフ。

カットオフ入れるともちろん答はあわない。

カットオフが定義されている範囲ではあうようになった。

外側の定義を追加した。これで大丈夫な気がする。

川井君のエミュレータの整数からフロートへの変換は、符号の扱いがなんかおかしい。まあ、 64 ビットに拡張された数字が正しくはいっていれば問題ない。多分、入力で与えたビット長ではなくて 64 ビットの MSB をみている？

まだネイバーリストとかみてないけど、ハードのチェックに進む？

8.4 ハードウェアデバッグ

そういう方針で。 cutoff から。

r3inv のテーブル

```
m0: 10 ビット
m1: 7 ビット
exp: 2 ビット
合計 19 ビット
```

```
g8linear_maketable | ruby make_linear_table.rb
```

でテーブルデータを作り、 r3inv.vhd に取り込む

入力が 1 = 0x10000 の時のエミュレータ

```
 tabid=10 exp0=ffffff7e idx =0 table outs = 3ff 58 1

 tabid=10 exp0=ffffff7e idx =0 table outs = 3ff 58 1
 1                 1         0.99902344 -0.000976562 10000 ffff
 tabid=0 exp0=ffffff7b idx =0 table outs = 2d3 3e 3
 2        0.35355339         0.35302734 -0.00148789 10200 fcd3
 tabid=8 exp0=ffffff7b idx =0 table outs = 313 2e 2
 3        0.19245009         0.19213867 -0.00161817 10300 fb13
 tabid=10 exp0=ffffff7b idx =0 table outs = 3ff 58 1
 4             0.125         0.12487793 -0.000976562 10400 f9ff
 tabid=14 exp0=ffffff7b idx =0 table outs = 2dc 33 1
 5       0.089442719        0.089355469 -0.000975489 10480 f8dc

 0.1         31.622777           31.6875 0.00204673 f933 109f6
 0.2          11.18034          11.203125 0.00203796 fb33 106cd
 0.3         6.0858062           6.09375 0.0013053 fc66 1050c
 0.4         3.9528471          3.9609375 0.00204673 fd33 103f6
 0.5         2.8284271          2.8242188 -0.00148789 fe00 102d3
 0.6         2.1516574           2.15625 0.00213444 fe66 10228
 0.7         1.7074694          1.7109375 0.00203111 fecc 1016c
 0.8         1.3975425          1.4003906 0.00203796 ff33 100cd
 0.9         1.1712139           1.171875 0.000564416 ff99 10058
```

シミュレーション結果の検証は、とりあえず 1 出力については vcd ファイルと検証パターンを比べればよい。検証パターンのフォーマットは、まず単純に 16 進で出力するだけにして、比較するプログラム群を作った。以下のような感じ

```
 r3inv.mqv: ../../emu/g8linear_makepattern.rb  ../../emu/g8linear_makepattern Makefile
ruby ../../emu/g8linear_makepattern.rb -n 1000
 testin.mqv: r3inv.mqv
cp -p r3inv.mqv testin.mqv
 check:
        getfilem.csh r3inv.vcd
ruby ../../emu/getcheckpatternfromvcd.rb -f r3inv.vcd -F r3inv.vpf -v r3invout -s 80
```

Quartus のシミュレーションで vcd ファイルを作成し、その出力と検証パターンを比較する。

同様に、カットオフのほうもテストする。

カットオフのテーブルサイズ

```
  m0: 10
  m1:  8  18
  m2:  6  24
  s1,2: 2  25 26
```

ここまでで 26 なので、指数には 6 ビットしかない。それで収まるように指数のベースを調整する。

この調整はした。入力が fc00 の時に、整数化した結果が 040 になっている。大体、最大値が 3ff までいくはずだから、全然おかしい？

これはシフト量がおかしいのを修正した。
intval= 100 tabid= 4 idx = 0, m = 1ff d 33 e,s= 21 0 1 result=1ff

```
   fc00 0 0 0     fffe
```

fc24 で間違える。

```
p1reg = ffffffff
intval= 112 tabid= 4 idx = 9, m = 1ff d 33 e,s= 21 77 0 1 result=1fe
   fc24 0 0 0     fffc
```

結果はなんだか全然違う。
p1reg2 が 0x47?

```
m0reg
dst
e0reg
e0reg2
exp
p1reg2
sum0reg3
```

ハードウェアの問題: m1 +m2 * dx の符号が m1 と変わった時の処理がない。
符号付きにしてもいいけど、それはそれで複雑なので以下の手順を考える。
(0,1) の範囲では答が合うようになった。ふう。

- 0 の時
- 1 およびそれよりも大きい時

のチェック必要。

```
  quartus_sim --read_settings_files=on --write_settings_files=off cutoff -c cutoff
```

でシミュレーション実行もできる。
テストパターン追加の必要: ランダムパターンを出す。
これは通った。ふう。

8.5 メインパイプラインについてのメモ

eps2j のパイプラインステージ
元々は 20

```
sub_int はレイテンシ 2
conv_int_to_float はレイテンシ 3
mul_float                      3
add_float                      4
```

元々は add_float が

```
(((dx + dy)+dz)+epsi)+epsj
```

となっていて、12 ステージ追加があった。 今回の計算順序では、加算結果を mul_float のステージと合わせるだけなので 4 段でいい。ここまでで r2f 計算できた。

```
            遅延

eps2sum     8
dzeps2      12  (+4)
r2f         16
rinv3       19
mrinv3      22  (rinv3+3)
cfarg       19  (r2f+3)
f/pscale    26  (cfarg+7)

cutoff の内部遅延
  floattoscaledint  2
  memory            1
  polynomial        4
```

なので合計 7 段

これ現在のコードはステージが多すぎる。減らしてみる？(8-¿7 にした)

polynomial.vhd のクリティカルパス: sum1reg と p1reg2 の間。

乗算器が 1 つあるだけか、、、速くするならここをパイプライン化

```
            遅延

f/pscale    26  (cfarg+7)
psr2        29  (pscale + 3 = r2f+10 +3)
pf          32  (psr+2)
pfshift     33  (pf+1)
pi          36  (pfshift+3)
ffact       29  (fscale+3)
f_0         32  (ffact+3)
node325_0   33  (f_0+1)
node326_0   36  (node325_0 + 3)
```

げろげろ、 g8simtest が答あわないように、、、 g8cutoff.c が rev 546 なら問題ない。561 は× 555 も × 552 は◯ 553 も◯ 554 は×

この 2 つの違いのうち、結果に影響しそうなのは以下の 3 つ。

```
@@ -131,7 +131,7 @@
     pc-> s1 = (coef[1]> 0)? 0:1;
     pc->m2 = fabs(coef[2])*1024/scale;
     pc-> s2 = (coef[2]*coef[1]> 0)? 0:1;
```

```
-    pc->e0 = 0x80+exp0;
+    pc->e0 = 0x20+exp0;

 }

@@ -145,7 +145,7 @@
     if (c0 > 1023) c0=1023;
     pc->m0 = c0;
     pc->m1 = fabs(coef[1])*1024/scale;
-    pc->e0 = exp0+1;
+    pc->e0 = exp0+1 - 64;
 }


@@ -327,7 +327,7 @@
        }
        if (result < 0) result =0;
        return convert_double_to_g8_float(ldexp((double)result,
-                                               pc->e0 - 10 - 0x80),
+                                               pc->e0 +96 - 10 - 0x80),
                                          8,9);
```

古い (正しい) 計算:

```
Enter cutoff func type (0:unity, 1: dll):ctype=1
Enter  xi, eps2i, h2i, eta, iindex:xi= 0 0 0 eps2i=0
eta=1 h2i=0 iindex=0
Continue calculation? (1/0)Enter  xj, mj, eps2i, h2j, jindex:xj= 0.4 0 0 mj=1
eps2j=0 h2j=7 jindex=1
r2s, h2ij = 0.16 7
r2s=0.16
rinv=2.5
arg to cutoff and out= 0.16 1 1
scales = 1.67772e+07  2.81475e+14  7.45058e-09  4.44089e-16
xi(0), xj(0) = 0 666666
xi(1), xj(1) = 0 0
dx0=    12d33 0   96  133     6.7092480000e+06
dx0sq=    15a8f 0   ad   8f     4.5011257262e+13
dx=  666666
dy=  0
dx_1=        0 0    0    0     0.0000000000e+00
dx1sq=        0 0    0    0     0.0000000000e+00
dx2dy2=    15a8f 0   ad   8f     4.5011257262e+13
dx2sq=        0 0    0    0     0.0000000000e+00
dz2eps2=        0 0    0    0     0.0000000000e+00
r2f=    15a8f 0   ad   8f     4.5011257262e+13
g8nnb jindex = 1
g8nbl rsqnbl=16580  rsq =15a8f
rinv3=     77ea 0   3b  1ea     3.3153398951e-21
m1, p2 = 0 0, p1reg = 0
intval= a3 tabid= 2 idx = 11, m = 200 0 0 e,s= 81 d7 1 1 result=200
```

```
m1, p2 = 0 0, p1reg = 0
intval= a3 tabid= 2 idx = 11, m = 200 0 0 e,s= 81 d7 1 1 result=200
eta=     a000 0   50    0     3.5527136788e-15
cfarg=     fa8f 0   7d   8f     1.5991210938e-01
fscale=    10000 0   80    0     1.0000000000e+00
pscale=    10000 0   80    0     1.0000000000e+00
pf=     d281 0   69   81     1.4924444258e-07
pfshift=    16881 0   b4   81     5.6382956272e+15
pi =   14080000000000 *pp=                0, run_begin=1
*pp =   14080000000000
*a_0 =         32100000
*a_1 =                0
*a_2 =                0
innb = 15a8f00000001
a= 6.25 0 0 p=-2.5
rnnb=0.16 jnnb=1 nnbl=1
bl= 0
ea= 6.25781 0 0 ep=2.50391
ernnb=0.159912 ejnnb=1 ennbl=1
bl= 0
Error:   4.0000000e-01   6.2500000e+00   1.2500000e-03   2.5000000e+00   1.5625000e-03
```

現在の版では

```
rinv3=    1f7ea 0   fb  1ea     2.0810725809e+37
```

指数が 96 ずれている。で、

```
@@ -145,7 +145,7 @@
     if (c0 > 1023) c0=1023;
     pc->m0 = c0;
     pc->m1 = fabs(coef[1])*1024/scale;
-    pc->e0 = exp0+1;
+    pc->e0 = exp0+1 - 64;
 }
```

が問題だったんだけど、ここ修正してもハードウェアの問題起きなかったのは何故だ？

と、それはともかく、cutoff.vhd でメモリの後にもう一段ステージがあったほうがよいような気がするのでいれておく。そうすると、後半のパイプラインステージがずれる。

```
cutoff: 8 stages
```

```
          遅延
node41    2
dx_0      5   (conv int to float = 3)
eps2sum   8
```

```
dzeps2     12  (+4)
r2f        16
rinv3      19
mrinv3     22  (rinv3+3)
cfarg      19  (r2f+3)
--------------------  ここから 1 ステージ増える
f/pscale   27  (cfarg+8)
psr2       30  (pscale + 3 = r2f+11 +3)
pf         33  (psr+2)
pfshift    34  (pf+1)
pi         37  (pfshift+3)
ffact      30  (fscale+3)
f_0        33  (ffact+3)
node325_0  34  (f_0+1)
node326_0  37  (node325_0 + 3)

r2fD10:  10-> 11   pscale: 27 r2f: 16           r2fD
mrinv3D4 4 -> 5    mrinv3: 22 fscale:27         mrinv3Da
mrinv3D7 3 -> 3    mrinv3D4: 27 psr2: 30        mrinv3Db
dxD27_0  27->25    dx_0: 5   ffact: 30 あれ？   dxD_0
dxD27_1  27->25                                 dxD_1
dxD27_2  27->25                                 dxD_2
jidD     16 調整必要なし
h2jD     12  調整必要なし
```

なので、 cutoff.vhd の再検証の必要あり。

これはパスした。 (testrun.csh)

pg_delay を速くしてみて、合成結果をチェックしておきたい。

パイプラインの動作チェックについては、まずシミュレータで a_0-a_2, pp, nnb_result, p_nbf を出力させて、これをチェックする。これがあえばブロックとのチェックになる。

vhdl で書いてしまおうと思ったが、 Cyclone III に入るパッケージがなくなるので止め。EP3C120 とか使えば入らないわけでもないが、フィットに時間かかるようになる可能性が高い。

```
u112: pp
u118: a0
u128: a1
u138: a2
```

1 パターンでのシミュレーション結果

- nnb の rsq は正しい。 15A8F
- 答は 0 になってしまっている。本当は

```
*pp =   14080000000000
*a_0 =         32100000
*a_1 =                0
*a_2 =                0
```

まず、ポテンシャルの途中経過を

```
r2f まではできている
r3inv: u100
   u102 : pg_mul_float_18_18_18
       port map (srca => rinv3, srcb => mjD19, dst => mrinv3, clk => pclk);

mr3inv: u102

r3inv, mr3inv は 3b:1ea になっている。:77ea これも正しい。
psr2: u107 15a8f のはず 5a8f になっている。
pf:   u109 d281 でないといけない bb:1ea =177ea とりあえず全然違う。
pi:   u111
pscale: u105 1000 がでないといけないけど、 0 になってる気が。
cfarg:  u103 fa8f のはずが、1a8x  (X は msg がないせいなので OK)
       r2f * eta なので遅延の問題ではない
```

eta が 0 のままになっている。書き込み ena がでてないから当然、、、これは修正した。でも、 cfarg がおかしい？

いや、これは optimized out されただけで、正しいような気が。とりあえず、今日はここまで。

問題は、乗算が正しくできている気がしないこと？

これだけのテストをしてみる。

単体だと、 cfarg に相当する乗算の結果はちゃんと fa8f になる。

ポートから出力させてみてもちゃんと FA8F になっている。

あ、これはやはり optimized out されていたため。あと、 cutoff table への書き込みデータが t 正しくなかった。修正したら、 pscale まではでるようになった。

```
pscale
15AF8
```

0001 0101 1010 1111 1000 17 からだと

```
0101 0110 1011 1110
5    6    B    E
```

あれ？

psr2 と mrinv3DB でずれがある。mr3Db が遅れている。

```
psr2
psr2= pscale+3        psr2= r2fD +3
pscale = cfarg + 8?
cfarg = r2f+3
```

なので、現行の psr2 = r2f+14 のはず. r2fD が 11 でないといけない？

```
mr3Db = mrinv3Da + 3
mrinv3Da = mrinv3+ 5
mrinv3 =  rinv3+3
rinv3 = r2f + 3?    14
```

```
pf=     d281 0   69   81     1.4924444258e-07
pfshift=    16881 0   b4   81     5.6382956272e+15
```

シフトしたあとの答が 16A81 になっていて、指数が 1 ずれるてる。

エミュレータではシフト量は 0x4b と書いてある。VHDL では何故か 4C だった。修正した。 a のほうは初めから 4b になっていた。何故？

よし、

```
*pp =   14080000000000
*a_0 =          32100000
```

は答が合うようになったぞ!

あとするべきテスト:まずランダムパターンでのペアテスト。これが通ったら積算。それができたら、ブロックでのテスト。

と思うと、今のテストパターン生成をちょっといじるだけか。読みこんだ最後が run_start なので、

- run_start だったら xi を出す
- 次のパターンが run_start であるか、データがなければ、出力させる

とすればよいはず。そういうふうに g8simtest_convert_pattern.rb を改修する。

テストもうひとつ

```
Enter  xi, eps2i, h2i, eta, iindex:xi= 0.551246 -0.946716 0.189126 eps2i=0
eta=1 h2i=0 iindex=0
Continue calculation? (1/0)Enter  xj, mj, eps2i, h2j, jindex:xj= 0.722476 -1.14961 0.0548372 mj=0.6
eps2j=0 h2j=1 jindex=1
r2s, h2ij = 0.0885212 1
r2s=0.0885212
rinv=3.36106
arg to cutoff and out= 0.0885212 1 1
scales = 1.67772e+07  2.81475e+14  7.45058e-09  4.44089e-16
xi(0), xj(0) = 8d1e6d b8f434
xi(1), xj(1) = ffffffffff0da402 fffffffffed9b2d9
dx0=    12abd 0   95   bd     2.8712960000e+06
dx0sq=    155bf 0   aa  1bf     8.2377472737e+12
dx=  2bd5c7
dy=  ffffffffffcc0ed7
dx_1=    32b3f 1   95  13f    -3.4037760000e+06
dx1sq=    156a3 0   ab   a3     1.1596411699e+13
dx2dy2=    15841 0   ac   41     1.9825569038e+13
dx2sq=    15451 0   aa   51     5.0938312131e+12
dz2eps2=    15451 0   aa   51     5.0938312131e+12
r2f=    158d6 0   ac   d6     2.4945170055e+13
g8nnb jindex = 1
g8nbl rsqnbl=16000  rsq =158d6
rinv3=     7a5f 0   3d   5f     8.0335781091e-21
m1, p2 = 0 0, p1reg = 0
```

```
intval= 5a tabid= 1 idx = d, m = 200 0 0 e,s= 21 77 1 1 result=200
m1, p2 = 0 0, p1reg = 0
intval= 5a tabid= 1 idx = d, m = 200 0 0 e,s= 21 77 1 1 result=200
eta=     a000 0   50    0     3.5527136788e-15
cfarg=     f8d6 0   7c   d6     8.8623046875e-02
fscale=    10000 0   80    0     1.0000000000e+00
pscale=    10000 0   80    0     1.0000000000e+00
psr2=    158d6 0   ac   d6     2.4945170055e+13
pf=     d23d 0   69   3d     1.3341195881e-07
pfshift=    1683d 0   b4   3d     5.0401613017e+15
pi =   11e80000000000 *pp=                0, run_begin=1
*pp =   11e80000000000
*a_0 =         22900000
*a_1 = ffffffffd6f00000
*a_2 = ffffffffe4c80000
innb = 158d600000001

r2f 158D6 のはずが 15966 (1.14us)
dx2dy2=    15841  これは OK (1.05us)
dz2eps2 が計算できてない。

z の引き算のあと: fffffffffdd9f44
dz               :32a27
```

あれ？

```
xi2, xj2: 306a8b e09cf

xi2 は OK
xj2 が間違っている。何故？
```

```
xi(0), xj(0) = 8d1e6d b8f434
xi(1), xj(1) = fffffffffff0da402 fffffffffed9b2d9
xi(2), xj(2) = 306a8b e09cf
dx0=    12abd 0   95   bd     2.8712960000e+06
dx0sq=    155bf 0   aa  1bf     8.2377472737e+12
dx=  2bd5c7
dy=  fffffffffcc0ed7
dx_1=    32b3f 1   95  13f    -3.4037760000e+06
dx1sq=    156a3 0   ab   a3     1.1596411699e+13
dx2dy2=    15841 0   ac   41     1.9825569038e+13
dz=  fffffffffdd9f44
dx2=    32a27 1   95   27    -2.2568960000e+06
dx2sq=    15451 0   aa   51     5.0938312131e+12
dz2eps2=    15451 0   aa   51     5.0938312131e+12
r2f=    158d6 0   ac   d6     2.4945170055e+13
g8nnb jindex = 1
g8nbl rsqnbl=16000  rsq =158d6
rinv3=     7a5f 0   3d   5f     8.0335781091e-21
```

```
m1, p2 = 0 0, p1reg = 0
intval= 5a tabid= 1 idx = d, m = 200 0 0 e,s= 21 77 1 1 result=200
m1, p2 = 0 0, p1reg = 0
intval= 5a tabid= 1 idx = d, m = 200 0 0 e,s= 21 77 1 1 result=200
eta=     a000 0   50    0     3.5527136788e-15
cfarg=     f8d6 0   7c   d6     8.8623046875e-02
fscale=    10000 0   80    0     1.0000000000e+00
pscale=    10000 0   80    0     1.0000000000e+00
psr2=    158d6 0   ac   d6     2.4945170055e+13
pf=     d23d 0   69   3d     1.3341195881e-07
pfshift=    1683d 0   b4   3d     5.0401613017e+15
pi =   11e80000000000 *pp=                0, run_begin=1
*pp =   11e80000000000
*a_0 =         22900000
*a_1 = ffffffffd6f00000
*a_2 = ffffffffe4c80000
innb = 158d600000001

xi(0), xj(0) = 8d1e6d
xi(1), xj(1) = fffffffffff0da402 ffffffffffed9b2d9
xi(2), xj(2) = 306a8b e09cf
jdata の下のほうは  e09cf ffed9b2d9 00b8f434
```

1600000000feaa00000001000e09cfffffffffed9b2d900b8f43

なので入力が間違っている。何故？

sprintf が長いのを出してた。32 ビットにマスクする必要あった。

innb がおかしい？

これは修正した

cutoff をいれると？

fscale: 10000 のはず

というわけで、 cutoff はまだおかしい。

cfarg は f8d6 at 1.2us で、これは正しい。

```
FFCC??? polyout これはなんかおかしい
       rint 5A = 101 1010 at 1.24 なので、これは OK
table 86cc35FF at 1.18 あ、そうではなくて、これは大丈夫な気が。

fscale:  1.39 で 1 になっている。もっともらしい。
mrinv3Da mrinv3 が正しいのが 1.27 なので、あれ？なんか全然変。サイク
ル 200 で回しているので、6 クロック、、、あれ？

rinv3 は 1.20(21?) ででる。
```

今日はここまで、、、、 (8/9)

現在 136.56MHz

Assignment -¿ Settings -¿ Analysis & Synthesis Settings -¿ More Settings -¿ Auto Shift Register Replacement

152MHz

なんか、パイプライン段数がまだ途中でずれてる？

```
            遅延

 node41     2
 dx_0       5   (conv int to float = 3)
 eps2sum    8
 dzeps2     12  (+4)
 r2f        16
 rinv3      19
 mrinv3     22  (rinv3+3)
 cfarg      19  (r2f+3)
 --------------------  ここから 1 ステージ増える
 f/pscale   27  (cfarg+8)
 psr2       30  (pscale + 3 = r2f+11 +3)
 pf         33  (psr+2)
 pfshift    34  (pf+1)
 pi         37  (pfshift+3)
 ffact      30  (fscale+3)
 f_0        33  (ffact+3)
 node325_0  34  (f_0+1)
 node326_0  37  (node325_0 + 3)

 r2fD10:  10-> 11   pscale: 27 r2f: 16           r2fD
 mrinv3D4 4 -> 5    mrinv3: 22 fscale:27         mrinv3Da
 mrinv3D7 3 -> 3    mrinv3D4: 27 psr2: 30        mrinv3Db
 dxD27_0  27->25    dx_0: 5   ffact: 30 あれ？   dxD_0
 dxD27_1  27->25                                 dxD_1
 dxD27_2  27->25                                 dxD_2
 jidD     16 調整必要なし
 h2jD     12  調整必要なし
```

run が入るのが 0.82 (cycle 41) fscale : 1.38 なのでサイクル 69, 28 サイクルあと mrinv3 1.26 なのでサイクル 63, 22 サイクルあと cfarg は？ 1.20 なのでサイクル 60, 19 サイクルあと

ということで、 fscale が 9 サイクル遅延あり？ cutoff での遅延段数を数え直す。

```
 cutoff の遅延               ここ    依存              トータル
  floattscaledint の遅延    2       なし               2
  rintreg  の遅延           1       floatoscaledint    3
  rintreg2 の遅延           1       rintreg            4
  rintreg3 の遅延           1       rintreg2           5
  メモリの遅延              1      floatoscaledint    3
  memout                     1       メモリ             4 ここであってない？
  ovlreg(5)                  6       f....              8
  polynomial                 4       memout and rintreg3?
```

cutoff を再チェック

1 ステージ減らしてみると？メモリが実は入出力にレジスタはいって 2 ステージだった。これを 1 ステージに減らす？

でも、 eASIC のメモリ実装のことを考えると、ここは余計にいれておいたほうが無難。なので、1 ステージ増やす。

```
cutoff の遅延              ここ   依存              トータル
 floattscaledint の遅延    2      なし              2
 rintreg  の遅延           1      floatoscaledint   3
 rintreg2 の遅延           1      rintreg           4
 rintreg3 の遅延           1      rintreg2          5
 メモリの遅延              1      floatoscaledint   3
 memout                    2      メモリ            5
 ovlreg(5)                 6      f....             8
 polynomial                4      memout            9
```

ということで、cutoff のトータル遅延は 9

```
           遅延

node41     2
dx_0       5   (conv int to float = 3)
eps2sum    8
dzeps2     12  (+4)
r2f        16
rinv3      19
mrinv3     22  (rinv3+3)
cfarg      19  (r2f+3)
---------------------  ここから 1 ステージ増える
f/pscale   28  (cfarg+9)
psr2       31  (pscale + 3 = r2f+12 +3)
pf         34  (psr+2)
pfshift    35  (pf+1)
pi         38  (pfshift+3)
ffact      31  (fscale+3)
f_0        34  (ffact+3)
node325_0  35  (f_0+1)
node326_0  38  (node325_0 + 3)

r2fD:    11->12   pscale: 27 r2f: 16          r2fD
mrinv3Da 5 -> 6    mrinv3: 22 fscale:28        mrinv3Da
mrinv3Db 3 -> 3    mrinv3Da: 28 psr2: 31       mrinv3Db
dxD_0   25->26    dx_0: 5   ffact: 31          dxD_0
dxD_1   25->26                                 dxD_1
dxD_2   25->26                                 dxD_2
jidD     16 調整必要なし
h2jD     12  調整必要なし

cutoff.vhd は検証済 (testrun.csh で)
```

で、 shift register を memory にしないようにしたら動作速度は 155 MHz までアップ

```
  遅いところ: pg_conv_int_to_float e2[0]->dst[0]
        polynomial:p1reg2[0]->e0reg3[7]
             cutoff:memout[19]-> polynomial:sum1reg[0]
```

で、 g8pipe に戻る。ffact: 1.44us のサイクルから

```
xi(0), xj(0) = 8d1e6d b8f434
xi(1), xj(1) = fffffffffff0da402 fffffffffed9b2d9
xi(2), xj(2) = 306a8b e09cf
dx0=    12abd 0   95   bd     2.8712960000e+06
dx0sq=    155bf 0   aa  1bf     8.2377472737e+12
dx=  2bd5c7
dy=  fffffffffcc0ed7
dx_1=    32b3f 1   95  13f    -3.4037760000e+06
dx1sq=    156a3 0   ab   a3     1.1596411699e+13
dx2dy2=    15841 0   ac   41     1.9825569038e+13
dz=  fffffffffdd9f44
dx2=    32a27 1   95   27    -2.2568960000e+06
dx2sq=    15451 0   aa   51     5.0938312131e+12
dz2eps2=    15451 0   aa   51     5.0938312131e+12
r2f=    158d6 0   ac   d6     2.4945170055e+13
g8nnb jindex = 1
g8nbl rsqnbl=16000  rsq =158d6
rinv3=     7a5f 0   3d   5f     8.0335781091e-21
mrinv3=     7929 0   3c  129     5.3535129244e-21
m1, p2 = 0 0, p1reg = 0
intval= 5a tabid= 1 idx = d, m = 200 0 0 e,s= 21 77 1 1 result=200
m1, p2 = 0 0, p1reg = 0
intval= 5a tabid= 1 idx = d, m = 200 0 0 e,s= 21 77 1 1 result=200
eta=     a000 0   50    0     3.5527136788e-15
cfarg=     f8d6 0   7c   d6     8.8623046875e-02
fscale=    10000 0   80    0     1.0000000000e+00
pscale=    10000 0   80    0     1.0000000000e+00
psr2=    158d6 0   ac   d6     2.4945170055e+13
pf=     d23d 0   69   3d     1.3341195881e-07
pfshift=    1683d 0   b4   3d     5.0401613017e+15
pi =   11e80000000000 *pp=                0, run_begin=1
*pp =   11e80000000000
*a_0 =         22900000
*a_1 = ffffffffd6f00000
*a_2 = ffffffffe4c80000
innb = 158d600000001

psr2 1.44us 158d6
pf
```

アキュムレータの遅延を設定してなかった。4 ついれて合計がでるのがいくつか確認する。
ポテンシャルの遅延があっている。
よし、これはあうようになった。 vcd ファイルからデータを取り出す方法を考える。
これはびっくり、エラーがでたぞ、、、

```
vector 25 differs
sim: 007b200000000000 0111800000000000 ff19c00000000000 1900000000000000 0001243300000001
```

```
emu: 007b200000000000 0111800000000000 ff19c00000000000 0000000000000000 0001243300000001
vector 26 differs
sim: 005ee00000000000 ffe0780000000000 008bc00000000000 0d80000000000000 000124af00000001
emu: 005ee00000000000 ffe0780000000000 008bc00000000000 0000000000000000 000124af00000001
vector 86 differs
sim: ff00c00000000000 005c600000000000 0186800000000000 2700000000000000 000123e100000001
emu: ff00c00000000000 005c600000000000 0186800000000000 0000000000000000 000123e100000001

3 errors found
vector 25 differs
sim: 007b200000000000 0111800000000000 ff19c00000000000 1900000000000000 0001243300000001
emu: 007b200000000000 0111800000000000 ff19c00000000000 0000000000000000 0001243300000001
vector 26 differs
sim: 005ee00000000000 ffe0780000000000 008bc00000000000 0d80000000000000 000124af00000001
emu: 005ee00000000000 ffe0780000000000 008bc00000000000 0000000000000000 000124af00000001
vector 66 differs
sim: 0000000422000000 0000002890000000 0000000dd4000000 0020700000000000 00013e4700000001
emu: 0000000064a00000 00000003db000000 0000000150800000 0003190000000000 00013e4700000001
vector 80 differs
sim: 0000001418000000 fffffffb1e000000 0000000da4000000 0011b80000000000 00013e0d00000001
emu: 00000003f1000000 fffffffff0a400000 00000002af000000 00037b0000000000 00013e0d00000001
vector 86 differs
sim: ff00c00000000000 005c600000000000 0186800000000000 2700000000000000 000123e100000001
emu: ff00c00000000000 005c600000000000 0186800000000000 0000000000000000 000123e100000001
vector 94 differs
sim: 0000018c80000000 fffffcf300000000 fffff7fc00000000 06de000000000000 00013e7d00000001
emu: 0000001138000000 ffffffde10000000 ffffffa6e0000000 004c600000000000 00013e7d00000001

6 errors found
vector 66 differs
sim: 0000000000000000 0000000000000000 0000000000000000 0000000000000000 00013e4700000001
emu: 00000003bb000000 0000002490000000 0000000c84000000 001d580000000000 00013e4700000001
vector 80 differs
sim: 0000000000000000 0000000000000000 0000000000000000 0000000000000000 00013e0d00000001
emu: 0000001018000000 fffffffc15000000 0000000af4000000 000e3c0000000000 00013e0d00000001
vector 94 differs
sim: 0000000000000000 0000000000000000 0000000000000000 0000000000000000 00013e7d00000001
emu: 0000017b80000000 fffffd1700000000 fffff85a00000000 0692000000000000 00013e7d00000001

3 errors found
```

この辺だけのを作ってみる。

```
c = 0 のものは

  5723  17:02   cd ../../emu
  5724  17:02   ruby pairtest.rb -n 1 -c 0 -v vout -n 100
```

で作って抜き出してみた。

```
G8PIPE_vector        0        0        0     a000   9ad358 ffffffffffc9c956   fb1916    16000
```

```
a= 2.57052e+08 5.71474e+08 -4.80891e+08 p=-25325.3
rnnb=1.02798e-09 jnnb=1 nnbl=1
bl= 0
ea= 2.58212e+08 5.73571e+08 -4.82869e+08 ep=800
ernnb=1.02409e-09 ejnnb=1 ennbl=1
bl= 0
Error:   3.2062144e-05   7.8988276e+08   3.9338587e-03   2.5325334e+04   9.6841108e-01
G8PIPE_vector        0        0        0     a000   876251 ffffffffff7d4bcf ffffffffff1d73eb    160
a= 1.99204e+08 -6.65603e+07 2.92305e+08 p=-12719.2
rnnb=1.24872e-09 jnnb=1 nnbl=1
bl= 0
ea= 1.98967e+08 -6.61258e+07 2.93077e+08 ep=432
ernnb=1.24965e-09 ejnnb=1 ennbl=1
bl= 0
Error:   3.5337301e-05   3.5993667e+08   2.5478307e-03   1.2719190e+04   9.6603557e-01
G8PIPE_vector        0        0        0     a000   4e1418 ffffffffff76d210 ffffffffff555aef    160
a= -5.35953e+08 1.92116e+08 8.17763e+08 p=-29939.5
rnnb=9.02792e-10 jnnb=1 nnbl=1
bl= 0
ea= -5.35298e+08 1.93724e+08 8.18938e+08 ep=1248
ernnb=9.03128e-10 ejnnb=1 ennbl=1
bl= 0
Error:   3.0046492e-05   9.9643892e+08   2.1043463e-03   2.9939494e+04   9.5831593e-01
```

全て、ポテンシャルがオーバーフローしている。オーバーフローの処理がそもそも違うのではないか？

```
ruby pairtest.rb -n 1 -c 0 -v vout -n 100 -S 25
```

```
Enter  xi, eps2i, h2i, eta, iindex:xi= 0.604787 -0.211772 0.980852 eps2i=0
eta=1 h2i=0 iindex=0
Continue calculation? (1/0)Enter  xj, mj, eps2i, h2j, jindex:xj= 0.604798 -0.211748 0.980832 mj=0.8
eps2j=0 h2j=1 jindex=1
r2s, h2ij = 1.02798e-09 1
r2s=1.02798e-09
rinv=31189.4
arg to cutoff and out= 1.02798e-09 1 1
scales = 1.67772e+07  2.81475e+14  7.45058e-09  4.44089e-16
xi(0), xj(0) = 9ad358 9ad407
xi(1), xj(1) = fffffffffc9c956 fffffffffc9cadb
xi(2), xj(2) = fb1916 fb17cf
dx0=    10ebc 0   87   bc     1.7500000000e+02
dx0sq=    11dbd 0   8e  1bd     3.0624000000e+04
dx=  af
dy=  185
dx_1=    1110a 0   88  10a     3.8900000000e+02
dx1sq=    1224f 0   91   4f     1.5129600000e+05
dx2dy2=    122c6 0   91   c6     1.8176000000e+05
dz=  ffffffffffffeb9
dx2=    3108e 1   88   8e    -3.2700000000e+02
```

```
dx2sq=    12143 0   90  143     1.0688000000e+05
dz2eps2=    12143 0   90  143     1.0688000000e+05
r2f=    12433 0   92   33     2.8825600000e+05
g8nnb jindex = 1
g8nbl rsqnbl=16000  rsq =12433
rinv3=     c979 0   64  179     6.4683263190e-09
mrinv3=     c8d1 0   64   d1     5.2459654398e-09
m1, p2 = 0 0, p1reg = 0
intval= 0 tabid= 0 idx = 0, m = 200 0 0 e,s= 21 77 1 1 result=200
m1, p2 = 0 0, p1reg = 0
intval= 0 tabid= 0 idx = 0, m = 200 0 0 e,s= 21 77 1 1 result=200
eta=     a000 0   50    0     3.5527136788e-15
cfarg=     c433 0   62   33     1.0240910342e-09
fscale=    10000 0   80    0     1.0000000000e+00
pscale=    10000 0   80    0     1.0000000000e+00
psr2=    12433 0   92   33     2.8825600000e+05
pf=     ed19 0   76  119     1.5125274658e-03
pfshift=    18319 0   c1  119     5.7141672072e+19
Warning: pg_conv_float_to_int: too large exponent (65).
conversion result would not fit in 64-bit causing overflow.
src:0x0000000000018319
pi = 1900000000000000 *pp=                0, run_begin=1
*pp = 1900000000000000
*a_0 =   7b200000000000
*a_1 =  111800000000000
*a_2 = ff19c00000000000
innb = 1243300000001

 pf : ED19 at 1.50us OK
 pshift:  18319 at 1.52 OK
 pi upper:  0 になってしまう。
```

18319 での計算

```
 exponent:  16:9 = 1 1000 001 = 1100 0001 = C1
 offset = 1 引き算の結果は C0
```

仮数: 119 に hidden bit をいれて 319

これで答えはあうようになった。

```
c=1, S=66
Enter  xi, eps2i, h2i, eta, iindex:xi= 0.762055 0.144872 -0.395135 eps2i=0
eta=1 h2i=0 iindex=0
Continue calculation? (1/0)Enter  xj, mj, eps2i, h2j, jindex:xj= 0.762338 0.147649 -0.394188 mj=0.0
eps2j=0 h2j=1 jindex=1
r2s, h2ij = 8.68816e-06 1
r2s=8.68816e-06
rinv=339.263
arg to cutoff and out= 8.68816e-06 1 1
```

```
scales = 1.67772e+07  2.81475e+14  7.45058e-09  4.44089e-16
xi(0), xj(0) = c3160d c32892
xi(1), xj(1) = 251658 25cc56
xi(2), xj(2) = fffffffffff9ad868 fffffffffff9b1677
dx0=    11851 0   8c   51     4.7440000000e+03
dx0sq=    130af 0   98   af     2.2511616000e+07
dx=  1285
dy=  b5fe
dx_1=    11ed7 0   8f   d7     4.6528000000e+04
dx1sq=    13e05 0   9f    5     2.1684551680e+09
dx2dy2=    13e0a 0   9f    a     2.1894266880e+09
dz=  3e0f
dx2=    11be1 0   8d  1e1     1.5888000000e+04
dx2sq=    137c3 0   9b  1c3     2.5244467200e+08
dz2eps2=    137c3 0   9b  1c3     2.5244467200e+08
r2f=    13e47 0   9f   47     2.4452792320e+09
g8nnb jindex = 1
g8nbl rsqnbl=16000  rsq =13e47
rinv3=     a254 0   51   54     8.2711615335e-15
mrinv3=     9591 0   4a  191     9.8987658348e-17
m1, p2 = 0 0, p1reg = 0
intval= 0 tabid= 0 idx = 0, m = 200 0 0 e,s= 21 77 1 1 result=200
m1, p2 = 0 0, p1reg = 0
intval= 0 tabid= 0 idx = 0, m = 200 0 0 e,s= 21 77 1 1 result=200
eta=     a000 0   50    0     3.5527136788e-15
cfarg=     de47 0   6f   47     8.6873769760e-06
fscale=    10000 0   80    0     1.0000000000e+00
pscale=    10000 0   80    0     1.0000000000e+00
psr2=    13e47 0   9f   47     2.4452792320e+09
pf=     d407 0   6a    7     2.4167820811e-07
pfshift=    16a07 0   b5    7     9.1303445571e+15
pi =   20700000000000 *pp=                0, run_begin=1
*pp =   20700000000000
*a_0 =        422000000
*a_1 =       2890000000
*a_2 =        dd4000000
innb = 13e4700000001
Continue calculation? (1/0)test 67

sim: 0000000422000000 0000002890000000 0000000dd4000000 0020700000000000 00013e4700000001
emu: 0000000064a00000 00000003db000000 0000000150800000 0003190000000000 00013e4700000001

pf: CD19  1100 1101 0001 1001

mrinv3: 9591: 9591 これは OK

pscale 1.38 では F9C0

cfarg: 1.20 で DE47
42: fscale
44: rint   1.24 で 247 つまり、シフトアウトできてない。
45: memout
```

これは、0 にしないといけないところの条件判定が間違っていた。
次: g8block.vhd でのテスト。これは、

- 複数のパイプラインあり
- ネイバーリストあり

なので、粒子も複数いれる。エミュレータのほうに構造体とか準備する。
testrun0.csh -n 100 はパスした。
j 粒子を複数出すようにした。 -n 200 -N 20 でテスト実行中。
test 0 で間違いみっけ。

```
vector 9 differs
sim: ffffdf9d423fc000 00002e5368481e00 ffffcbacee5be000 1cd9f58000000341 0001040000000001
emu: ffffdf9d423fc000 00002e5368481e00 ffffcbacee5be000 1cd9f58000000000 0001040000000001
vector 19 differs
sim: 057ef3e58b5e4800 0b8922138b5e7000 2e4d9d5b032b8800 f32667880000021d 0001020000000001
emu: 057ef3e58b5e4800 0b8922138b5e7000 2e4d9d5b032b8800 f326678800000000 0001020000000001
vector 152 differs
sim: 006b239820874000 03e702162a17ad61 f4a3f2d0b18db900 25f090200000029f 0001000000000001
emu: 006b239820874000 03e702162a17b000 f4a3f2d0b18db900 25f0902000000000 0001000000000001

3 errors found
```

明らかにシフトアウトの処理がまだおかしい。うーむ。

```
sim: C
emu: VHDL
```

なので、問題は emu のほう。
距離と質量をいじるだけみたいなテストをしてみよう。

```
  5958  11:59   ruby pairtest.rb -c 0 -v vout -n 10 -N 20 -S 9 > &   simout
  cd...
  5962  12:00   ruby ../../emu/g8simtest_convert_pattern.rb -d 44 < ../../emu/vout > testin.mqv


  5965  12:01   getfilem.csh g8pipe.vcd
  5966  12:01   ruby ../../emu/printcheckpatternfromvcd.rb -f g8pipe.vcd -v p_adro,p_datao -s 18 >
  5967  12:01   ruby ../../emu/readandcompareg8pipepattern.rb -f xxx -F ../../emu/vout


pfshift

VHDL 16E4D 173F7 1660D 16715 168BD 19341 17087 178B7
```

これはシミュレータと一致

pi 19341 が、上にシフトアウトできてない。

64 ビット以上のシフトの時にまともな答がでない。C の pgemu.c を修正した。

これで多分 OK? -n 200 -N 20 でテスト実行中。

これはパスした。 (8/13)

g8block.vhd が、ネイバーリストの読み出しを考えてないような気がする。あ、そうではなくて、これはそもそも結果バッファにコピーとかしないで読むのか。だから、これはこれで OK。

とりあえず、ネイバーリスト以外のところのチップレベルテストを考える。

基本的に、

- カットオフを書く
- j 粒子を書く
- i 粒子を書く
- 計算させる
- 結果回収

の順番。

j 粒子は、

- 開始アドレス: 初期 粒子 id * 8 + 0x10000
- 転送語数: 粒子数 * 6

で送るだけのように見える。

i 粒子は、ブロック単位。ブロック内は連続アドレスで、粒子数 * 6 のはず。

```
0x02000-0x021ff i 粒子バッファ 0
0x02200-0x023ff i 粒子バッファ 1
```

カットオフは、本来のアドレス + 0x40 に連続アドレスで書く

計算させるには、まず i 粒子をバッファからコピーする。これはアドレス 1 への書き込み。下位 10 ビットはバッファメモリの開始アドレス、上位 10 ビット (16 ビットから) は転送するワード数 (粒子数*6 のはず)。

で、それだけ待つ。

それから、計算させる。これは、アドレス 0 への書き込み。下位 16 ビットが粒子数、上位は開始アドレス。

それが終わったら、結果をバッファにコピー。これはアドレス 2 への書き込み。

下位 10 ビットはバッファメモリの開始アドレス、上位 10 ビット (16 ビットから) は転送するワード数 (粒子数*10 のはず)。

これも終わったら、各バッファメモリを読み出す。

バッファメモリ開始アドレスから、必要語数を、というのを、ブロック数だけやる。

jmem に書く信号はそれっぽい。

cutoff はどうか

we はでている。

i 粒子バッファは、 rp オペレーションで最初の粒子はでることを確認した。というか、2 個目がでてない気が、、、

memreadaddress が回ってない。これは入力データが間違っていた。

ip 書けているような気がする。

計算が始まっているような気はまだしない。 2010/8/13 今日はここまで

run はでている。

start シグナルがきてからの遅延:

j 粒子データと合わせた。多分大丈夫

結果のコピー。アドレス間の遅延がおかしい気がする。あと、初期値ができてない？ip も同じ問題あるはず。

ip のほうも開始アドレスを設定してみる。これは OK

最後の fo buffer の読み出しはなんか変？

raen が入力からはいったらすぐに回っている。あれ？

これは、 VHDL の間違い。単純に、fifo の出力でないものをいれていた。修正。

g8sim.c にこれ用の出力をつける。

シミュレータ出力として必要なのは、

- カットオフ 実装済み
- i 粒子
- j 粒子
- 答

なので、形式はいいことにしてとにかくこれらを 16 進で出す。

変換するプログラムも作ったので、結果をチェック。

もちろん、まずは答があわない。

xi がきているかどうかをチェック。

xi はくるようになった。

run がでていない？

確かにだしてない気が、、、

run_block は 1.87us ででるようになった。2 サイクルでるのも正しい。

jp の出力は、そこでアドレスが回るのはいいけど数字はおかしい。まあ、書き込みアドレスが間違ってましたね。

jmem への最初のデータがくるのは 1.19 us

waddr_reg が不定だ。何故？

これは、アドレスの配線が全然変だった。これは修正した。

で、今度は、 jp 1 粒子の中のデータ順が全然おかしい気が、、、

なんだから、上位と下位が逆

上位がまだおかしい。ビットずれてる？0016580000010000000000010080000000800000008000000

B2C00001..... 000000000.....

3っつめの ram の振る舞いがなんか変。

B2C000001...

1011 0010 1100 101100101100

1658

1 0110 0101 1000 1011001011000 101100101100

何故かどこかが 1 ビットずれている？

これは、 jdata 163, 181 が使われていないためっぽい。符号みてないからね、、、

a_0 までは正しく計算できている。なので、バッファへの書き込み等ができてない？

粒子一つはそれっぽい。 foselect がなんかおかしいのを直し。

パイプライン 2 本目。距離は正しく計算できてるが、ポテンシャル、力が全部 0。

カットオフ関数の出力をみてみる。

これは pipe 1 でででていない。

入力は:

カットオフテーブルがはいってない？

そうではなくて、 eta がはいってなかった。書き込みカウンタがそれだけ回ってない？

確かに、 12 をセットしないといけないのに、 10 をセットしていた。これでは駄目である。

これで、ブロック 1 つ、パイプライン 2 本ではちゃんとそれぞれ計算できること確認。

ブロック 2 つにするとまだおかしい。

ブロック 1 では eta がはいってない。 xi は？

マルチブロックのアドレス変換がおかしい。メモリは 1024 ワードあるんだから、 10 ビットはそっちで使うべき。で、ブロックを最大 16 とすれば、14 ビット必要。

なので、 we_ball をビット 14 に変更する。また、ブロックデコードを 13-10 に変更。

対応して、テストベクタも変更必要。

これで 2 ブロックで答えある。 rvalid g ずれている。

あとはネイバーリストと、大規模なテストですね。

まず ネイバーリスト。

そもそもデコードがおかしくないか？

g8top: 14-11 でブロックデコードしている g8block: 10 でデコード。

でも、ネイバーリストは 11 ビットアドレス使っている。ので、

g8top: 15-12 でブロックデコードしている g8block: 11 でデコード。

と変更する。

あと、結果の遅延が違うので、合わせる必要あり。メモリのステージ数は 2 である。あ、そうではなくて、これは同じ。バッファと並列になっているため。但し、 nbl count はおかしい気がする。

あれ、 eps2 をいれるとあわない？

あと、 t=0 であわない、、、、 この時、 2.87-2.89 で polynomial 出力が 0 オーバーフローした時の振る舞いがおかしい？

おかしいみたい。

cinfty が 0 になっていた。ちょっとおかしいけど、書き込みをちゃんと下位もデコードしたら直った。何故？

eps のステージ数は正しいか？

```
node41     2
dx_0       5   (conv int to float = 3)
```

と書いてあるのに、4 段になってた。これでは駄目である。 5 に修正。

eps2 が 15933 なのに 1933 がはいっている。0 64cca000 1 0 0 0 00 0 160005 1 0 0 0 00 15933 = 0001 0101 1001 0011 0011

```
          0001 01 01 1001 0011 0011
```

memreadaddress は 1.91 から回る。

なので、 6 個目のデータがでるのは 2.01 + 0.04

メモリからのデータが 0 になっている。書き込みに失敗？

現在は 2.07 のでるデータが 0。

連続アドレスで書くと書ける。

we_block が 1 サイクル前だった。

これを直したら、 eps2 関係の問題は解消。

epsj をふってみるとどうか？

まだあわない。なんかずれてる？

一定 epsj ならあう。G8VECTOR_FO 0 1 1 ffffffff2088800 ffffffff34b3f40 ffffffff2390140 16b84000000000 15e0500000003

最後だけ 0 にしてみると。G8VECTOR_FO 0 1 1 ffffffef5e8800 ffffffff0d53f40 ffffffeef10140 18104000000000 15d3d00000003

本当の答は上の。

一定だとあうということは、配線は間違ってないはず。

eps2j の遅延はやっぱり 4 でよかったはず。こっちに戻す。

これで大丈夫な感じ。

we_cutoff も、アドレス、データに対してずれていた気が。修正してみる。

これで、力に関しては一通りみたはず。

ネイバーも、カウンタの回り方は正しい気がする。

後は出力をチェックするコードを書く必要あり。

というか、出力の出方はおかしいね、、、パイプラインステージがあってない？

これは色々合わせたけど、まだブロック 1 の最後のワードがでていない気が。

```
     raddr   rdata
4.63 0800
4.65 0801
4.67 0C00
4.69 0C01  ABCD0004
4.71 0C02  ABCD0004
4.73 0C03  00000003
4.75 0C04  00020003
4.77 1800  00030003
4.79 1801  00000000
```

3 クロック遅れるのはいいか？これはパリティつけるのに 1 クロック使っているから。

ブロック切換えの前にダミーサイクルをいれて見るとどうか？原理的にはそういう問題ではありえない気がするけど、、、

ダミーサイクルをいれると読める。え？

あ、 g8top でのセレクションが間違ってた。 (この辺で rev 600)

ruby printcheckpatternfromvcd.rb -s 20 -f ../vhdl/g8top.vcd -v rvalid,rdata — awk '{if ($1 == 1) print}'

でなんか作れる。

1 ffbd09c9 1 fffffff 1 3ffc05d6e 1 fffffff 1 3ffb15107 1 fffffff 1 a5000000 1 2000068c0 1 3 1 600015e81 1 6fa29c800 1 fffffff 1 2faa3f800 1 fffffff 1 6f931c800 1 fffffff 1 0 1 70007ceb0 1 3 1 700015d3d 1 ffbd09c9 1 fffffff 1 3ffc05d6e 1 fffffff 1 3ffb15107 1 fffffff 1 a5000000 1 2000068c0 1 3 1 600015e81 1 6fa29c800 1 fffffff 1 2faa3f800 1 fffffff 1 6f931c800 1 fffffff 1 0 1 70007ceb0 1 3 1 700015d3d 1 dabcd0004 1 3 1 400020003 1 30003 1 400040003 1 cabcd0003 1 3 1 400020003 1 30003

nearest neighbour の何か

self interaction を neighbour list、 nnb から抜くのができてない。

パイプライン段数が多分違う。

r2f: 16

あれ、 jid がない ??? これは、なんか変なものに合成されてしまっているから？

これらは合わせた。また、 nnb, nbl とも、index を見るように修正した。

ポテンシャルのスケーリングがなんかおかしい気が。

現在 75 ビットシフトだけど、 50 ビットくらいにしたい。

変更箇所:ハードウェアと、シミュレータと、結果の変換のところだけ？

これは変更した。

ステータスの下位 4 ビットも実装した。

ステータス、他にみておきたいものはないか？

ループバックモードをつける。

これは専用ピンで制御。

testmode : 1 なら、入力をそのまま戻す。上位ビットはパリティではなくて

```
        we     : in  std_logic;
        waen   : in  std_logic;
```

```
raen   : in  std_logic;
rreq   : in  std_logic;
```

をこの順番で。

色々動いたので、パイプライン数を増やしてみる。 shift register replace を on、 physical synthesis effort を fast に。

これで EP3C120 でパイプライン 16 本体いれたらコンパイルに 2 時間、動作速度 120MHz。最適化色々いれたら 130MHz に向上。

p=8, b=2 で i=16m j=32 でテストしたら何故かボロボロに間違えた、、、

粒子 0 から間違えるが、4x2 では間違えないので「おそらく」設計の問題ではない。timing simulation にしたのが悪かった?

pg_delay の使用量

r2f 18x12 mrinv3 18x6 x,y,z: 18x3x26

安田さんからきた critical path

u105/u2 s1reg2-¿e0reg3 (4ns) 加算器-エンコーダ-加算器

polynomial.vhd

現行の速度 (Quartus Classinc TA で) 228.6 MHz (cutoff.vhd 単体)

e0reg3 のステージで指数の補正をしていたのを、次のステージに回す。

u105/u2 memout_reg-¿sum1reg (4 くらい) 乗算器+加算器

本来論理合成で融合して欲しいけど
できてない？

u127/m2_reg-¿u127/dst_reg 3.75 float_to_int で整数 64 ビット

の符号調整。次のステージに
符号 (キャリーイン) を渡すべき

u28/dst_reg-¿u31/f1_reg 3.74 32 ビットの符号調整。1 足すのに

`lpm_add` 使ったのが悪い？`adder` で
16 ステージ。

u104/memout_reg -¿ u104/u2/s1reg 3.67 sum1reg と同じ

u005/nz -¿ u007/nzx1 両者とも add_nonneg_float

`nz` がソースになっているのは、出力のセレクタがあるから。
`lpm_compare` の中が遅い。安直には、ここは指数部分だけをみればいいけど、、、

```
        -> u007/e1
        -> u007/mx1
        -> u007/my1
```

u004/nz -¿ u007/de1

```
-> u007/e1
```

u100/rreg-¿ u100/newman2 3.22 u007/nz -¿ u151/rmin ublock_0/ipcount-¿memreadaddress 3.08 u007/nz -¿ u151/id ublock_0/focount-¿focount 2.9

revision 611: e0reg3 だけいじったもの。

u127/m2_reg-¿u127/dst_reg

conv_float_to_int

方針としては、conv_float_to_int は符号と絶対値を出すようにして、pg_inc_int のほうで符号いれる。

元の実装に戻すだけ？つまり、 MSB に符号いれておいて、なんかすればいい。

これはやってみた。

```
svn diff -r 611 pg_module.vhd
Index: pg_module.vhd
===================================================================
--- pg_module.vhd       (リビジョン 605)
+++ pg_module.vhd       (作業コピー)
@@ -1129,17 +1129,29 @@
     end if;
   end process;

+--  process(clk)
+--  begin
+--    if (clk'event and clk='1') then
+--      if (sign(2) = '0') then
+--        dst(63 downto 0) <= m2;
+--      else
+--        dst(63 downto 0) <= (not m2) + conv_std_logic_vector(1, 63);
+--      end if;
+--    end if;
+--  end process;
+-- new code (2010/8/20)
+-- dst is 1's complement, not 2's
   process(clk)
   begin
-    if (clk'event and clk='1') then
-      if (sign(2) = '0') then
-        dst(63 downto 0) <= m2;
-      else
-        dst(63 downto 0) <= (not m2) + conv_std_logic_vector(1, 63);
+      if (clk'event and clk='1') then
+         dst(63) <= sign(2);
+         if (sign(2) = '0') then
```

```
+              dst(62 downto 0) <= m2(62 downto 0);
+          else
+              dst(62 downto 0) <= not m2(62 downto 0);
+          end if;
       end if;
-    end if;
   end process;
-
 end rtl;

 library ieee;
@@ -3042,8 +3054,9 @@
   srclow0    <= src0(31 downto 0);
   srchigh0   <= src0(63 downto 32);

---  carry(0)  <=     src(63);
-  carry(0)  <= '0';                     -- should always be 0.
+--  carry(0)  <= '0';                       -- should always be 0.
+-- assume that input is 1's complement
+  carry(0)  <=     src(63);
   rsig(0)   <= run;

   process(clk)
```

こんな感じ。

u28/dst_reg-¿u31/f1_reg 3.74 32 ビットの符号調整。1 足すのに

lpm_add 使ったのが悪い？adder で 16 ステージ。

conv_int_to_float はて？

```
 library ieee;

@@ -2646,12 +2658,13 @@
   s0      <=      src(31);
   fplus0  <=      src(30 downto 0);
   fminus0 <=  not src(30 downto 0);
-  u1: lpm_add_sub
-    generic map (LPM_WIDTH=>31,LPM_DIRECTION=>"ADD")
-    port map(result => fminus0a,
-             dataa  => fminus0,
-             datab  => conv_std_logic_vector(1, 31));
+--  u1: lpm_add_sub
+--    generic map (LPM_WIDTH=>31,LPM_DIRECTION=>"ADD")
+--    port map(result => fminus0a,
+--             dataa  => fminus0,
+--             datab  => conv_std_logic_vector(1, 31));

+  fminus0a <= unsigned(fminus0) + 1;
   s0a <= '0' when src(30 downto 0) = conv_std_logic_vector(0, 31) else '1';
```

```
    s0b <= s0a & s0;
```

単純に、lpm の代わりに +1 にしてみた。
(rev 612)

```
u005/mx3 -> u005/eadd4   3.34
                add_nonneg_float

10 ビット加算で mxy3
この加算で cout がキャリーアウト
キャリーアウトを加算器に入れるのではなく、
1 インクリメントを作っておいて、セレクトすればいい。

svn diff -r 612 pg_module.vhd
Index: pg_module.vhd
===================================================================
--- pg_module.vhd       (リビジョン 612)
+++ pg_module.vhd       (作業コピー)
@@ -1959,7 +1959,7 @@

   signal ex0, ey0 : std_logic_vector(7 downto 0);  -- ex) e bit, ex) 6 downto 0
   signal e0, esmall0, e1, e2, e3 : std_logic_vector(7 downto 0);
-  signal eadd3, eadd4 : std_logic_vector(7 downto 0);
+  signal eadd3, eadd3inc, eadd4 : std_logic_vector(7 downto 0);
   signal esub3, esub4 : std_logic_vector(7 downto 0);
   signal coute3    : std_logic;

@@ -2214,13 +2214,15 @@
   --

   -- exponent would be increased by 1 at max.
-  u7add: lpm_add_sub
-    generic map (LPM_WIDTH     => 8)
-    port map(result  => eadd3,
-             add_sub => '1',
-             dataa   => e3,
-             datab   => conv_std_logic_vector(0, 7) & coutm3);
+--  u7add: lpm_add_sub
+--    generic map (LPM_WIDTH     => 8)
+--    port map(result  => eadd3,
+--             add_sub => '1',
+--             dataa   => e3,
+--             datab   => conv_std_logic_vector(0, 7) & coutm3);

+  eadd3inc <= unsigned(e3) + 1;
+  eadd3 <=   eadd3inc when   coutm3 = '1' else  e3;
   -- shift mantissa 1-bit to the right, if necessary.
   process (clk) begin
     if (clk'event and clk='1') then
```

特に問題ない？ (rev 613)
u005/nz -¿ u007/nzx1 両者とも add_nonneg_float
さて、、、まず、これをオペレータにしてみる。それで答に問題なければ、比較のビット数を減らす。
u100/rreg-¿ u100/newman2 3.22

r3inv である。
加算を次のステージにすれば OK そう。

r3inv previos version: rev562
rev 616: 上の修正をした。
u007/nz -¿ u151/rmin u007/nz -¿ u151/id
nnb.... 比較が遅いと思われる。さて?
ublock_0/ipcount-¿memreadaddress 3.08 ublock_0/focount-¿focount 2.9
cutoff のパイプラインを修正する。
1 段増やした。

```
            遅延

node41      2
dx_0        5   (conv int to float = 3)
eps2sum     8
dzeps2      12  (+4)
r2f         16
rinv3       19
mrinv3      22  (rinv3+3)
cfarg       19  (r2f+3)
--------------------  ここから 1 ステージ増える
f/pscale    28  (cfarg+9)
psr2        31  (pscale + 3 = r2f+12 +3)
pf          34  (psr2+2)
pfshift     35  (pf+1)
pi          38  (pfshift+3)
ffact       31  (fscale+3)
f_0         34  (ffact+3)
node325_0   35  (f_0+1)
node326_0   38  (node325_0 + 3)

r2fD:     11->12   pscale: 27 r2f: 16           r2fD
mrinv3Da 5 -> 6     mrinv3: 22 fscale:28         mrinv3Da
mrinv3Db 3 -> 3     mrinv3Da: 28 psr2: 31        mrinv3Db
dxD_0    25->26    dx_0: 5   ffact: 31          dxD_0
dxD_1    25->26                                 dxD_1
dxD_2    25->26                                 dxD_2
jidD      16 調整必要なし
h2jD      12  調整必要なし
```

から、さらに遅延が増える

```
            遅延

node41      2
dx_0        5   (conv int to float = 3)
eps2sum     8
dzeps2      12  (+4)
r2f         16
rinv3       19
mrinv3      22  (rinv3+3)
cfarg       19  (r2f+3)
--------------------  ここから 1 ステージ増える
f/pscale    29  (cfarg+10)
psr2        32  (pscale + 3 = r2f+12 +3)
pf          35  (psr2+2)
pfshift     36  (pf+1)
pi          39  (pfshift+3)
ffact       32  (fscale+3)
f_0         35  (ffact+3)
node325_0   36  (f_0+1)
node326_0   39  (node325_0 + 3)

r2fD:     11->12->13   pscale: 27 r2f: 16           r2fD
mrinv3Da 5 -> 6->7     mrinv3: 22 fscale:28         mrinv3Da
mrinv3Db 3 -> 3-     mrinv3Da: 28 psr2: 31        mrinv3Db
dxD_0    25->26->27     dx_0: 5   ffact: 31          dxD_0
dxD_1    25->26->27                                  dxD_1
dxD_2    25->26->27                                  dxD_2
jidD      16 調整必要なし
h2jD      12  調整必要なし
```

run 信号 g8pipe.vhd u1x8 (inc_int) 36-¿37 他は変更不要なはず

で、rev 663 何か答えがあわない。 r660 で再チェック。

svn up -r 660 U cutoff.vhd U g8pipe.vhd U polynomial.vhd リビジョン 660 に更新しました。

sim:00000000 emu:00110002 vector 150 differs sim:00000000 emu:00120002 vector 151 differs sim:00000000 emu:00130002 vector 152 differs sim:00000000 emu:00150002 vector 153 differs sim:00000000 emu:00160002 vector 154 differs sim:00000000 emu:00180002 vector 155 differs sim:00000000 emu:001a0002 vector 156 differs sim:00000000 emu:001b0002 vector 157 differs sim:00000000 emu:001c0002 vector 158 differs sim:00000000 emu:001d0002 vector 159 differs sim:00000000 emu:001e0002 vector 160 differs sim:abcd005e emu:001f0002 vector 161 differs sim:00000002 emu:00220002 vector 162 differs sim:00010001 emu:00230002 vector 163 differs sim:00030003 emu:00240002 vector 164 differs sim:00040003 emu:00280002 vector 165 differs sim:00050003 emu:002b0002 vector 166 differs sim:00060003 emu:002d0002 vector 167 differs sim:00070003 emu:002e0002 vector 168 differs sim:00090002 emu:002f0002 vector 169 differs sim:000a0003 emu:00300002 vector 170 differs sim:000b0002 emu:00340002 vector 171 differs sim:000c0003 emu:00350002 vector 172 differs sim:000d0003 emu:00360002 vector 173 differs sim:000e0002 emu:00370002 vector 174 differs sim:000f0003 emu:003a0002 vector 175 differs sim:00100003 emu:003c0002 vector 176 differs sim:00110003 emu:003e0002 vector 177 differs sim:00120003 emu:003f0002 vector 178 differs sim:00130003 emu:00400002 vector 179 differs sim:00140002 emu:00440002 vector 180 differs sim:00150003 emu:00450002 vector 181 differs sim:00160002 emu:00460002 vector 182 dif-

fers sim:00180003 emu:00470002 vector 183 differs sim:001a0002 emu:00490002 vector 184 differs sim:001b0003 emu:004a0002 vector 185 differs sim:001c0003 emu:004d0002 vector 186 differs sim:001d0002 emu:004e0002 vector 187 differs sim:001e0002 emu:004f0002 vector 188 differs sim:001f0003 emu:00500002 vector 189 differs sim:00220003 emu:00530002 vector 190 differs sim:00230002 emu:00550002 vector 191 differs sim:00240003 emu:00560002 vector 192 differs sim:00260001 emu:00570002 vector 193 differs sim:00280003 emu:00580002 vector 194 differs sim:002b0003 emu:00590002 vector 195 differs sim:002c0001 emu:005b0002 vector 196 differs sim:002d0002 emu:005c0002 vector 197 differs sim:002e0003 emu:005d0002 vector 198 differs sim:002f0002 emu:005f0002 vector 199 differs sim:00300003 emu:00600002 vector 200 differs sim:00320002 emu:00610002 vector 201 differs sim:00330001 emu:00620002 vector 202 differs sim:00340003 emu:00630002 vector 203 differs sim:00350003 emu:00650002 vector 204 differs sim:00370002 emu:00690002 vector 205 differs sim:003a0002 emu:006a0002 vector 206 differs sim:003c0003 emu:006c0002 vector 207 differs sim:003e0003 emu:006d0002 vector 208 differs sim:003f0003 emu:006f0002 vector 209 differs sim:00400003 emu:00700002 vector 210 differs sim:00440003 emu:00710002 vector 211 differs sim:00450003 emu:00730002 vector 212 differs sim:00460003 emu:00740002 vector 213 differs sim:00470003 emu:00750002 vector 214 differs sim:00490003 emu:00760002 vector 215 differs sim:004a0002 emu:00770002 vector 216 differs sim:004b0001 emu:00780002 vector 217 differs sim:004d0003 emu:00790002 vector 218 differs sim:004e0002 emu:007a0002 vector 219 differs sim:004f0002 emu:007b0002 vector 220 differs sim:00500003 emu:007c0002 vector 221 differs sim:00520002 emu:007d0002

力はあってるっぽい。

あ、入力がおかしい。 -i 16 でやってた。これでは駄目。

`r663` に戻してもう一度。

functional netlist を出す。

net: g8top.qsf g8top.vhd

```
quartus_map --read_settings_files=on --write_settings_files=off \
         g8top -c g8top --generate_functional_sim_netlist
```

こんな感じでできる。

8.6 2010/9/12

パイプラインちょっといじる。

ポテンシャル、力でテーブル同じにする。これはシミュレーションでもこれでもいいはず。まあ、P3M だとややこしいけど、これは諦める。ツリーなら問題なし。

現在の critical path

```
  Startpoint: u_core/ublock_0/upipe_0/u105/u2/s1reg2_reg
              (rising edge-triggered flip-flop clocked by clk)
  Endpoint: u_core/ublock_0/upipe_0/u105/u2/exp0reg3_reg[2]
            (rising edge-triggered flip-flop clocked by clk)
```

u2: polynomial.vhd

```
sum0a <= unsigned('0' & m0reg2) + unsigned("000" & p1reg2) when s1reg2 = '0'
        else unsigned('0' & m0reg2) - unsigned("000" & p1reg2);
sum0 <= sum0a(9 downto 0)  when sum0a(10)='0'
        else "0000000000";
u2: penc_10
    port map (a => sum0, c=> exp0);
```

なので、penc には sum0a 89 downto 0) をいれて、sum0a(10) で結構をセレクトすればちょっと速くなるかもしれない。 penc は入力が all 0 なら 0x0f を返すはずなので、そういうふうに。

penc を次のステージに移動、というのも考えられるけど、ちょっと厳しい気がする、、、

いじる前のクロック: 207.8 いじった後: 217.4 本当？

一応本当らしい。答はあうのかしら？

答もだいじょぶっぽい。

392LE, 217 MHz p1reg2-¿exp0

もうちょっといじるとすると、加減算の答を両方作って、penc にいれてから選択。 答を 0 にする必要があるのは s1reg2 が 0 で sum0ap(10) = 1 の時。

なぜかこっちのほうが LE 数が減って遅くなる ?? しかも答を間違えている。

rev 663 はもちろん間違えない。rev 712 は

```
Startpoint: u_core/ublock_0/upipe_0/u032/e2_reg[2]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: u_core/ublock_0/upipe_0/u032/dst_reg[0]
          (rising edge-triggered flip-flop clocked by clk)
```

これは pg_conv_int_to_float_32_18

明らかに、 n2 の計算を前のステージでするので OK なはず。あ、そうでもない。 11111 からの減算を単なる否定で処理しているので、これを前にもっていってもあまり効果ない。シフタは 5 段で多分最適。 round_ubf1_22 は単に 0 判定なので、もうちょっとステージ減らせるはず。なので、最適化さぼってるだけと思われる。

一応、 n2 を前のステージに回してみる。

```
Startpoint: u_core/ublock_0/upipe_0/u030/dst_reg[20]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: u_core/ublock_0/upipe_0/u033/f1_reg[20]
```

u30 は sub_int_32_32_32 で、 u33 は pg_conv_int_to_float_32_18

これは、論理合成がさぼってるだけの可能性が高いのでパス。

```
Startpoint: u_core/ublock_0/upipe_0/u004/my3_reg[2]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: u_core/ublock_0/upipe_0/u004/eadd4_reg[5]
          (rising edge-triggered flip-flop clocked by clk)
```

add_nonneg_float_18_18_18

eadd3

10 ビット加算器の合成が×な感じ

```
  Startpoint: u_core/ublock_0/upipe_0/u007/nz_reg[4]
              (rising edge-triggered flip-flop clocked by clk)
  Endpoint: u_core/ublock_0/upipe_0/u100/newexp1_reg[7]
            (rising edge-triggered flip-flop clocked by clk)
```

8.7 2010/9/17

現在のクリティカルパス

```
  Startpoint: u_core/ublock_0/upipe_0/u033/n2_reg[2]  2.64
  Endpoint: u_core/ublock_0/upipe_0/u033/dst_reg[0]
```

conv_int_to_float

```
  Startpoint: u_core/ublock_0/upipe_0/u004/my3_reg[2] 2.62
  Endpoint: u_core/ublock_0/upipe_0/u004/eadd4_reg[5]
```

u_core/ublock_0/upipe_0/u004/u6/cnv/ のステージ多い。

```
  Startpoint: u_core/ublock_0/upipe_0/u006/mx3_reg[5] 2.61
  Endpoint: u_core/ublock_0/upipe_0/u006/eadd4_reg[6]

  Startpoint: u_core/ublock_0/upipe_0/u030/dst_reg[3] 2.60
  Endpoint: u_core/ublock_0/upipe_0/u033/f1_reg[29]
```

u_core/ublock_0/upipe_0/u031 の中のステージ多い。

```
  Startpoint: u_core/ublock_0/upipe_0/u104/u2/s1reg2_reg     2.58
  Endpoint: u_core/ublock_0/upipe_0/u104/u2/exp0reg3_reg[3]
```

2.74 から 2.58 に減った。

8.8 2010/10/29

eASIC (10/27 まで) やっておくはずになっていたこと

- FIFO ぬいたものを作る: こちらでやる
- signed を使っているところ: lpm で signed を使っているところ: 安田さんから資料がくるので確認して、なんかする
- メモリラッパ:コンテンションの確認があるので、全体の動作を

  するテストデータを。

- テストパターン

FIFO ぬいたもの:

信号のロジック:

fifo: 書き込み:fifowe で書く。

これは clkin で作られる

読み出し:信号は fifovalid
これは fiforead の 1 クロック遅れ
fiforead は fifoempty の反転

それぞれレベルは:

fifowe: 正論理
fifovalid やはり正論理

なので、単純に抜くだけで大丈夫なはず。

readandcompareg8toppattern.rb にオプション追加して、 HEX でシミュレータ出力がでるようにした。あと必要なのは、 g8chipsimtest_convert_pattern.rb に初めからパリティもつけるオプションをつけること。

と思ったけどこれは面倒なので、 mqv のほうをいじる。

生成は

```
5304  1:21    ruby ../emu/readandcompareg8toppattern.rb -F ../emu/xxx-0-0 -d | ruby ../emu/add_par
5330  1:32    ruby ../emu/add_parity.rb -f 2 < testin-0-0.mqv > testin
```

でできる。

8.9 2010/11/7

GRAPE-8 作業

1. 初期値の追加：g8top.vhd: 実装はした
2. rst: 修正、あと、テストパターンにリセットをいれる

   VHDL 修正はした。これは、makef8chiptestin.rb を修正すること。あ、違う、g8chipsimtest_convert_pattern.rb だ。これの変更もした。変更は以下の通り

   svn diff g8chipsimtest_convert_pattern.rb Index: g8chipsimtest_convert_pattern.rb ==========
   — g8chipsimtest_convert_pattern.rb (リビジョン 874) +++ g8chipsimtest_convert_pattern.rb (作業コピー) @@ -56,6 +56,18 @@

   + + Short name: -r + Long name: –add_reset + Value type: bool + Default value: true + Variable name: add_reset + Description: add reset cycle at the beginning + Long description: + add reset cycle at the beginning + + +

   END

@@ -83,6 +95,12 @@

```
      n.times{print "0 0        0 0 0 0 0 #{$dbgadr}\n"}

    end
```

+def add_reset(n) + print_dummy_data(1) + n.times{print ”1 0 0 0 0 0 0 #{$dbgadr}\n”} + print_dummy_data(1) +end +

```
    def print_cutoff_vector(s, table_id)
      a = s.split
      print "0 0 ", a[2], " 1 0 0 0  #{$dbgadr}\n"
```

@@ -217,6 +235,8 @@

```
    print mqv_header
```

+add_reset(1) if $clop.add_reset +

```
    cutoff_table.each_index{|id|
      addr = 0x40 + id * 0x20
      print "0 0       #{addr.to_s(16)}   0 1 0 0\n"
```

3. nbf の未使用ビットの固定 VHDL 修正はした:

svn diff -r 612 g8block.vhd Index: g8block.vhd ==========
— g8block.vhd (リビジョン 612) +++ g8block.vhd (作業コピー) @@ -1,5 +1,5 @@

```
    --  g8block.vhd
```

— Time-stamp: ¡10/08/20 22:48:09 makino> +– Time-stamp: ¡10/11/04 17:54:39 makino>

```
    -- JM 2010/6/15
    -- created
    --
```

@@ -260,6 +260,10 @@

```
                        debug_in     => x"00");
      end generate for_pipe;
```

+ for_nbf: for i in NPIPES to 15 generate + nbf(i) ¡= ’0’; + end generate for_nbf; +

```
      u_nbl :  nbl
          port map(flags  => nbf,
                   run    => runshift(16),  -- need to fix delay here
```

4. メモリの出力 (jdata) からパイプラインに配るところ 1 段入れる。
   実装はした

```
svn diff -r 612 jdata_mem.vhd
Index: jdata_mem.vhd
===================================================================
--- jdata_mem.vhd       (リビジョン 612)
+++ jdata_mem.vhd       (作業コピー)
@@ -1,5 +1,5 @@
 --  jdata_mem.vhd
---  Time-stamp: <10/08/15 18:59:27 makino>
+--  Time-stamp: <10/11/04 17:12:28 makino>
 -- JM 2010/6/17
 -- created
 --
@@ -47,6 +47,7 @@

      signal readaddress  : std_logic_vector(MEM_DEPTH downto 0);
      signal mem_q        : std_logic_vector(191 downto 0);
+     signal jdata0        : std_logic_vector(191 downto 0);
      signal we0,we1,we2  : std_logic;
      signal run1         : std_logic;
      signal waddr_reg    : std_logic_vector(14 downto 0);
@@ -110,7 +111,8 @@
              elsif (run = '1') then
                  readaddress <= unsigned(readaddress) + '1';
              end if;
-             jdata <= mem_q;
+             jdata0 <= mem_q;
+             jdata <= jdata0;
          end if;
      end process;
```

```
\begin{enumerate}
\item
   signed を使っているところをなるべく減らす。

   これは先方で対応済

\end{enumerate}
```

mqv ファイルを送ること: パリティつけて出した形に変換。

済.

あ、出力にもつけること: これもソース変更はすんだ。

週末にやる作業

タイミング制約
形式:
クロックサイクル
デューティの最大・最小
クロック遷移からの最大・最小遅延

くらいを指定する。これは Aria GX のデータシートをみて考えること。
内部回路では本当はクロックの立ち下がりは使わないが、タイミング制約の形式として必要

Arria の制約:

入力 (SSTL-18 Class I とする) table 4-49

relative to GCLK (not GCLK-PLL)

SSTL 18 Class I t_su = 1.113 t_H = -1.008
-6 speed grade 2.479 -2.202

GCLK-PLL

SSTL 18 Class I t_su = 2.555 t_H = -2.450
-6 speed grade 5.585 -5.308

出力 table 4-50

relative to PLL GCLK

SSTL 18 Class I drive strength=4mA 1.195 - 2.484

問題は内部的なクロックスキューとすると、 Table 4-46 の 100ps の値を使って

最小遅延 -100ps
最大遅延 +100ps

だけど、ちょっとおおきめにして

最小遅延 -100ps
最大遅延 +1000ps

としておく。

クロック
DCD (Arria GX Device Handbook, table 4-108-113 での最大値
112,113,110 111 109 108

```
        1.8V SSTL Class I 65          60  115  85

1.5V HSTL Class I 70          70  115  115
```

8.10 I/O 定義

以下 * がついているのは未確認のもの

ロジック関係

rst	I	リセット
testmode	I	テストモード
clkin	I	クロック
wdata[0:35]	I	データ入力
we	I	データ入力イネーブル
waen	I	書き込みアドレス入力イネーブル
raen	I	読み出しアドレス入力イネーブル
rreq	I	読出しリクエスト
rdata[0:35]	O	データ出力
rvalid	O	データ出力ストローブ
status[0:7]	O	ステータス出力

PLL 関係

pll_clk	I	pll モジュール用クロック
pll_we	I	pll モジュールライトイネーブル
pll_a[0:3]	I	pll モジュールアドレス入力
pll_d[0:11]	I	pll モジュールデータ入力
pll_req	I	
pll_rstb	I	
clk_outn	O	
clk_outp	O	
pll_ack	O	
pll_lock	O	

クロック制御関係

dclk	I	デバッグ用クロック
debugmode	I	デバッグ用クロック切換え
clken*	I	クロックイネーブル
clkrst*	I	クロックモジュールリセット

JTAG

jtag_tclk	I	JTAG 用クロック
jtag_tdi	I	TDI
jtag_tms	I	TMS
jtag_trst_n	I	TRST
jtag_tdo	O	TDO

8.11 ピン配置

方針:あまり深く考えないで、左下側を使う。左が入力、下がメインな出力、それ以外は上。で、外側 2 列に収める

```
        rst             I   C1      リセット
        testmode        I   C2      テストモード
        clkin           I   T2      クロック
        wdata[0]        I   E1
        wdata[1]        I   E2
        wdata[2]        I   F2
        wdata[3]        I   G1
        wdata[4]        I   G2
        wdata[5]        I   H1
        wdata[6]        I   H2
        wdata[7]        I   J2
        wdata[8]        I   K1
        wdata[9]        I   K2
        wdata[10]       I   L1
        wdata[11]       I   L2
        wdata[12]       I   M2
        wdata[13]       I   N1
        wdata[14]       I   N2
        wdata[15]       I   P1
        wdata[16]       I   P2
        wdata[17]       I   R1
        wdata[18]       I   R2
        wdata[19]       I   U1
        wdata[20]       I   V1
        wdata[21]       I   V2
        wdata[22]       I   W1
        wdata[23]       I   W2
        wdata[24]       I   Y1
        wdata[25]       I   Y2
        wdata[26]       I   AA1
        wdata[27]       I   AA2
        wdata[28]       I   AB2
        wdata[29]       I   AB3
        wdata[30]       I   AC1
        wdata[31]       I   AC2
        wdata[32]       I   AD1
        wdata[33]       I   AD2
        wdata[34]       I   AE2
        wdata[35]       I   AE3

        we              I   AH2     データ入力イネーブル
        waen            I   AG1     書き込みアドレス入力イネーブル
        raen            I   AG2     読み出しアドレス入力イネーブル
        rreq            I   AF1     読出しリクエスト
        rvalid          O   AK2     データ出力ストローブ
clk_outn        O   AF3
clk_outp        O   AF4
```

rdata[0]	O	AJ3
rdata[1]	O	AK3
rdata[2]	O	AJ4
rdata[3]	O	AJ5
rdata[4]	O	AK5
rdata[5]	O	AJ6
rdata[6]	O	AK6
rdata[7]	O	AK7
rdata[8]	O	AJ8
rdata[9]	O	AJ9
rdata[10]	O	AK9
rdata[11]	O	AJ10
rdata[12]	O	AK10
rdata[13]	O	AJ11
rdata[14]	O	AJ12
rdata[15]	O	AK12
rdata[16]	O	AJ13
rdata[17]	O	AK13
rdata[18]	O	AJ14
rdata[19]	O	AK14
rdata[20]	O	AJ15
rdata[21]	O	AJ16
rdata[22]	O	AJ17
rdata[23]	O	AK17
rdata[24]	O	AJ18
rdata[25]	O	AK18
rdata[26]	O	AJ19
rdata[27]	O	AK19
rdata[28]	O	AJ21
rdata[29]	O	AK21
rdata[30]	O	AJ22
rdata[31]	O	AK22
rdata[32]	O	AJ23
rdata[33]	O	AJ24
rdata[34]	O	AK24
rdata[35]	O	AK25
status[0]	O	AH30
status[1]	O	AG30
status[2]	O	AF30
status[3]	O	AD30
status[4]	O	AC30
status[5]	O	AA30
status[6]	O	Y30
status[7]	O	V30

PLL 関係

pll_clk	I	B4	pll モジュール用クロック
pll_we	I	B6	pll モジュールライトイネーブル
pll_a[0]	I	A6	pll モジュールアドレス入力
pll_a[1]	I	C7	pll モジュールアドレス入力
pll_a[2]	I	D7	pll モジュールアドレス入力

```
pll_a[3]        I   A10    pll モジュールアドレス入力
pll_d[0]        I   B10    pll モジュールデータ入力
pll_d[1]        I   F12
pll_d[2]        I   C11
pll_d[3]        I   D12
pll_d[4]        I   E12
pll_d[5]        I   A13
pll_d[6]        I   B13
pll_d[7]        I   A14
pll_d[8]        I   B14
pll_d[9]        I   C14
pll_d[10]       I   E16
pll_d[11]       I   F16
        pll_req         I   C17
pll_rstb        I   D17
pll_ack         O   A19
pll_lock        O   B19
```

クロック制御関係

```
        dclk            I  T3       デバッグ用クロック
debugmode       I  U3       デバッグ用クロック切換え
*clken          I  V3       クロックイネーブル
*clkrst         I  V4       クロックモジュールリセット
```

JTAG

```
        jtag_tclk       I   D5     JTAG 用クロック
        jtag_tdi        I   H10    TDI
        jtag_tms        I   E5     TMS
        jtag_trst_n     I   K12    TRST
        jtag_tdo        O   J10    TDO
```

公式のデータとこれを一致させるのは残念なが手でやる必要あり。

ファイルは

```
DKR_20101013/N2X_DSS10_45031_NAOJ_GRAPE_8_20101119.xls
```

を当面使う。

ピン配置に関する色々:

- N2X740_FC896_Pin_0003.pdf (papers/grape8 にあり) に色々あり
- Bank1,2,5,6 には、出荷前のテストの兼用ピンが多数ある。これらはクロックやリセット、クロックイネーブル等は避けたほうが無難
- clk 入力については資料の p.11 Table12. に一覧がある推奨ピンを使うこと
- テスト用のクロックも clk 入力の近くの推奨ピンにおくこと
- 余ったクロックの推奨ピンは、他に使うのは問題ない

- pll 関係の入力はまだ情報なし
- JTAG は資料の p.11 Table11. に一覧があり。これは他に使わないこと
- JTAG の I/O 電圧: Arria は 2.5 だが、これにこだわらないことにする。

Bank 3, 4:
現在の割り当て: 5,6,7,8 を使う。問題ありそうなピンを使ってないかどうか確認。
クロック推奨ピンは

```
C15
B15
G16
H16
R28
R29
T25
T24
AH16
AJ16
AD15
AD15
AC15
T3
T2
R6
R7
```

現在の配置なら clk T3, dclk T2

we	I	AH2	データ入力イネーブル	OK
waen	I	AG1	書き込みアドレス入力イネーブル	OK
raen	I	AG2	読み出しアドレス入力イネーブル	OK
rreq	I	AF1	読出しリクエスト	OK

これで一応割り当てた?

8.12 2010/12/2

reset でクリアしたほうがよいレジスタ
g8top の run_delays と g8block の runshift, ipcount, focount

8.13 2010/12/8

eASIC 打ち合わせ
こちらの AI
pg_delay について:

```
    begin
      if(clk'event and clk='1') then
        each_delay: for j in 0 to DELAY-1 loop
          s((DELAY + 1) * i + j + 1) <= s((DELAY + 1) * i + j);
        end loop each_delay;
      end if;
    end process;
```

loop を外に出して generate にする

メモリアドレスコンテンション

nbl メモリででる。

計算が終わったあとに write enable がおりない。

これは修正必須

FPGA のほう。jitter の特性をだして欲しいとのこと。

pll_clk: 50MHz

8.14 2010/12/13

メモリアドレスコンテンション nbl メモリででる。

```
    wren_sig <= '1' when (flags1 /= x"0000")
                and (adrcount(15 downto 10) = "000000")
                else '0';
```

で、 run1 をみていないのがおかしい。若干適当だが、以下の変更をした。

```
svn diff -r 929 nbl.vhd
Index: nbl.vhd
===================================================================
--- nbl.vhd     (リビジョン 929)
+++ nbl.vhd     (作業コピー)
@@ -1,5 +1,5 @@
 --  nbl.vhd
---  Time-stamp: <10/08/16 16:00:14 makino>
+--  Time-stamp: <10/12/13 13:07:04 makino>
 -- JM 2010/6/10
 -- created
 --
@@ -73,6 +73,7 @@
     data_sig <= jcount & flags1;
     wren_sig <= '1' when (flags1 /= x"0000")
                 and (adrcount(15 downto 10) = "000000")
+                and (run = '1' or run1 = '1')
                 else '0';
     wraddress_sig <= adrcount(9 downto 0);
```

これでいいはず？

8.15 2010/12/30

回路変更についてのメモ

”¿” 付きは牧野、なしは安田さんから。 12/22 辺りのやりとり

```
> 1. 1x1 のもの
>
> foreadaddress
>      パイプラインレジスタをいれるのは可能ですが、問題はこれが遅いのが配
>      置のせいか、それとも論理合成に問題があって段数が増えているのかだと
>      思います。単純なセレクタで、パイプライン 1 本の時に負荷が大きいような
>      論理でもないので。詳細情報を見る必要があると思います。

 配置に依存していると考えられます。
 ワーストのパスを確認しましたが、LUT が 4 個に対し、Inverter が 14 個、パス中に
 存在します。32bit 分ありますので、それぞれに引っ張られていると思います。
 クローン化も試しておりますが、レジスタを追加していただく方がよいかと
 思っております。
```

foreadaddress は g8block.vhd の中。これをとりあえず 1 クロックずらす。で、g8top の rvalid も 1 クロックずらした。これだけでいいはずだが一応動作チェック。

チェックしたら間違ってきた。明らかにおかしいのはネイバーリストかな？いじってないし。

あ、そうではなくて、これはそもそもメモリにコピーするだけなので関係なかった。メモリへの書き込みタイミングを修正。

```
> iid->rmin
>      jid と iid の比較を数クロック前にずらすことでレジスタをいれることが
>      できます。
> eps2
>      配置の問題であれば、レジスタ入れるのは問題ありません

 ご変更いただければ、と思います。
```

iid はどうせそのままでてるだけ。 jid を変更する。 で、 jnei という信号を作った。 eps2 は単純にレジスタいれた。

```
> rst_sig
>      レジスタ入れるのは問題ありません

 入力側で 1 段追加し、少し分割してみましたが、まだ改善の必要があるようです。
 最新のレポートで、弊社でも検討します。
```

こちらは対応しないことで。

```
> astart
>      ここは論理はなくてレジスタとレジスタが線でつながっているだけなので、
>      クローニング で対応していただけないかと。

 すでに追加したもので、まだ間に合わなかった箇所ですので、もう少し分割してみます。
```

これもこちらでは今回対応しない。

とりあえずここまで。(2010/12/30)

```
> 6x8
>
> ipbuf_q_d ->  iid_reg
> wdata_d1
> waddr_d1
> waddr ->  foreadaddress
>     レジスタ入れるのは問題ありません
```

クローン化も試しておりましたが、レジスタを入れていただく方が効果的だと思います。
また、クローン化を行うにあたっても、レジスタが入っている方が行い易いと思われます。

```
p_we    => pipewe_d(i)
p_adri  => ipwriteaddress_d(2 downto 0)
p_datai => ipbuf_q_d,
```

の 3 つをそれぞれ 1 クロック遅らせる必要あり。これはやってみた。

wdata_d1: g8block への書き込みデータ。なので、 waddr, wdata, we, cutoff_addr, cutoff_data, cutoff_wen を全て g8block の中のほうで遅らせる。これもいれた。

waddr: rtrig, rtaddr, rtlen

これも、 g8block の中でおくらせる。

```
> rst_sig
>     レジスタ入れるのは問題ありません
```

1x1 と同様で、ファンアウトが大きくなっていますので、最新結果をもとに検討したいと
思います。

```
> pg_delay
>     これは最終段ないしは 2 段程度を eCell に、となると思いますが、、、
```

はい。そのように対応する予定です。

```
> foreadaddress_d ->  fobuf bram の何か
>     ？
```

レジスタの配置を踏まえて検討しております。

これらは対応しない。

8.16 2011/1/16

jmem 関係パイプラインレジスタ追加。

既存 既存 既存 1 2 3 4 5 6 既存

```
 ================================================================================================
 mem_q - jdata0 - jdata - blk01 - blk0 - pipe0123 - pipe0_0 - pipe0_1 - pipe0_2 -> u_block0/u_pipe0
                        |       |      |          + pipe1_0 - pipe1_1 - pipe1_2 -> u_block0/u_pipe1
                        |       |      |          + pipe2_0 - pipe2_1 - pipe2_2 -> u_block0/u_pipe2
                        |       |      |          + pipe3_0 - pipe3_1 - pipe3_2 -> u_block0/u_pipe3
                        |       |      + pipe4567 - pipe4_0 - pipe4_1 - pipe4_2 -> u_block0/u_pipe4
                        |       |                 + pipe5_0 - pipe5_1 - pipe5_2 -> u_block0/u_pipe5
                        |       |                 + pipe6_0 - pipe6_1 - pipe6_2 -> u_block0/u_pipe6
                        |       |                 + pipe7_0 - pipe7_1 - pipe7_2 -> u_block0/u_pipe7
                        |       + blk1 - pipe0123 - pipe0_0 - pipe0_1 - pipe0_2 -> u_block1/u_pipe0
                        |       |      |          + pipe1_0 - pipe1_1 - pipe1_2 -> u_block1/u_pipe1
                        |       |      |          + pipe2_0 - pipe2_1 - pipe2_2 -> u_block1/u_pipe2
                        |       |      |          + pipe3_0 - pipe3_1 - pipe3_2 -> u_block1/u_pipe3
                        |       |      + pipe4567 - pipe4_0 - pipe4_1 - pipe4_2 -> u_block1/u_pipe4
                        |       |                 + pipe5_0 - pipe5_1 - pipe5_2 -> u_block1/u_pipe5
                        |       |                 + pipe6_0 - pipe6_1 - pipe6_2 -> u_block1/u_pipe6
                        |       |                 + pipe7_0 - pipe7_1 - pipe7_2 -> u_block1/u_pipe7
                        + blk23 - blk2 - pipe0123      (以下、同上)
                        |       |      + pipe4567
                        |       + blk3 - pipe0123
                        |              + pipe4567
                        + blk45 - blk4 - pipe0123
                                |      + pipe4567
                                + blk5 - pipe0123
                                       + pipe4567
```

いくらなんでも多すぎると思うが、面倒なので基本的にはこのまま。

1. g8top.vhd : 1 段追加、さらに、3 つに分割。結果的に 2 段
2. g8block.vhd: 1 段追加、さらに、2 つに分割
3. g8pipe.vhd: ここで 3 段追加。なんか美しくないな、、、

とはいえ、まあしょうがない。あまりに、美しくないので、 g8block.vhd の中で処理。
jdata に関するパイプラインレジスタ追加で必要となる他の作業:

- g8top_vhd の run_block をいじる

他は？
念のため、テストパターンのダミーサイクル数を増やしておく。

g8chipsimtest_convert_pattern.rb:

```
Index: g8chipsimtest_convert_pattern.rb
===================================================================
--- g8chipsimtest_convert_pattern.rb    (リビジョン 960)
+++ g8chipsimtest_convert_pattern.rb    (作業コピー)
@@ -177,7 +177,7 @@
   print "#RUN control patternss --- start calculation\n"
   print "0 0        0 0 1 0 0 #{$dbgadr}\n"
```

```
   printf("0 0 %08x 1 0 0 0 #{$dbgadr}\n",nj)
-  print_dummy_data(50+nj)
+  print_dummy_data(60+nj)
 end

 def print_foreadpattern
```

おそらくこれは関係なくて、 g8top.vhd の関係するところが全然間違っていたため。

```
--- g8top.evhd  (リビジョン 966)
+++ g8top.evhd  (作業コピー)
@@ -1,5 +1,5 @@
 --  g8top.vhd
---  Time-stamp: <11/01/16 17:40:34 makino>
+--  Time-stamp: <11/01/16 18:03:59 makino>
 -- JM 2010/6/15
 -- created
 -- 2010/8/13 Major change in top level interface
@@ -308,7 +308,7 @@
                 run_delays (0) <= run;
                 run_delays(60 downto 1) <= run_delays(59 downto 0);
             end if;
-            run_block <= run_delays(3);
+            run_block <= run_delays(6);
         end if;
     end process;

@@ -326,15 +326,15 @@
     begin
         if (clk'event and clk='1') then
             jdata0 <= jdata;
-<%a=""; (nblocks/2).times{|i| a+= "            jdata1_#{i} <= jdata;\n"}%><%=a%>
-<%a=""; (nblocks).times{|i|   a+= "            jdata2_#{i} <= jdata2_#{i/2};\n"}%><%=a%>
+<%a=""; (nblocks/2).times{|i| a+= "            jdata1_#{i} <= jdata0;\n"}%><%=a%>
+<%a=""; (nblocks).times{|i|   a+= "            jdata2_#{i} <= jdata1_#{i/2};\n"}%><%=a%>
         end if;
     end process;
 <%a=""; (nblocks).times{|i|
 a+= <<-END
         ublock_#{i}: g8block
             generic map(NPIPES => NPIPES)
-            port map(p_jdata  => jdata(JDATA_WIDTH-1 downto 0),
+            port map(p_jdata  => jdata2_#{i}(JDATA_WIDTH-1 downto 0),
                      pclk     => clk,
                      rst      => rst_sig,
                      run      => run_block,
```

8.17 2011/1/29

2/2 に修正 RTL

```
block からの出力のセレクタ: パイプライン化する
セレクタ出力の後にももう 1 段パイプラインレジスタ
g8block.vhd fobuf_data の後にレジスタ入れる
```

テストパラメータの修正が必要。6 ブロックから取るようにする。

この辺はやった。

6x1 だと答があわない？

xxx-1-0

この数がおかしい。

```
/tmp/emuout

1 7ff4a576e
1 ffffffff
1 8fd3ade60
1 ffffffff
1 afdbed572
1 ffffffff
1 a01e1f755
1 0
1 71
1 70001585d
1 7fffd9d43
1 ffffffff
1 8fe28a31e
1 ffffffff
1 8020c6f9c
1 0
1 d0483173b
1 0
1 10000005d
1 4000155b1
1 0
```

問題なかった。 g8block.evhd をまず修正。

修正したら間違えた?

```
G8VECTOR_FO 0 0 0 fffffffff4a576e fffffffffd3ade60 fffffffffdbed572 1e1f755 1585d00000071
G8VECTOR_FO 0 1 0 fffffffffffd9d43 fffffffffe28a31e 20c6f9c 483173b 155b10000005d
G8VECTOR_FO 0 2 0 0 0 0 38b000 161ab0000004a
G8VECTOR_FO 0 3 0 f08d46 36404f8 9a9830 47f0280 1563d00000004
G8VECTOR_FO 0 4 0 fffffffff88efad 29379ef 2a0fa4a 47641ae 1555d00000035
G8VECTOR_FO 0 5 0 fffffffffdb9feb 2c21eb4 fffffffffca66846 1d9c6c6 158db0000004d
```

物理的パイプライン配置:

```
0 1
2 3
4 5
```

論理的な順番

```
2  4  3  5  0  1
```

なので、以下のマッピングにする:

元々の番号	エンコードした後
0	4
1	5
2	0
3	2
4	1
5	3
6	6
7	7

で、答はあわない、、、

```
G8VECTOR_FO 0 0 0 fffffffffde9e95b fffffffffaaf4040 fffffffffab40cbc 4b9a020 1585d00000071
G8VECTOR_FO 0 1 0 337077 fffffffffd41e9ba 2e5a5de 69374a0 155b10000005d
G8VECTOR_FO 0 2 0 fffffffffff567723 199158 140c03e 1ea3100 161ab0000004a
G8VECTOR_FO 0 3 0 fe2ea2 442511e fc8044 68764a0 1563d00000004
G8VECTOR_FO 0 4 0 fffffffffff716b74 319ca44 3c2d763 67e0fe0 1555d00000035
G8VECTOR_FO 0 5 0 1ce6b3 5361210 fffffffff9ea4e94 4d07f20 158db0000004d

vector 0 differs
sim:fde9e95b
emu:00fe2ea2
vector 1 differs
sim:ffffffff
emu:00000000
vector 2 differs
sim:faaf4040
emu:0442511e
```

なので、 0 を読むべきところで 3 がでている。

8.18 テストパターンの作成方法

```
5027  21:27   ruby testrun0.rb -p 8 -b 6 -I 48
5029  21:30   csh -f makepatterns.csh
```

で 6x8 構成ができるはず。

8.19 イノテックからの vcd ファイル

```
 5222  0:33    wget ftp://ftp.innotech.co.jp/pub/eASIC_NAOJ/DATA_20101111.tgz
 5223  0:37    dir
 5224  0:37    tar tvzf DATA_20101111.tgz
 5225  0:38    tar xvzf DATA_20101111.tgz DATA_20101111/sim/test_top_0_0.vcd
 5226  0:38    dir
 5227  0:38    cd DA*1
 5228  0:38    dir
 5229  0:38    cd sim
 5230  0:38    dir
 5231  0:38    less *.vcd
 5232  0:42    less *.vcd
 5233  0:45    dir
 5234  0:45    grep rdata *vcd
 5235  0:46    grep ft3 *vcd
 5236  0:46    cd ../..
 5237  0:46    hh | grep tar
 5238  0:47    ar xvzf DATA_20101111.tgz DATA_20101111/sim/testout-0-0.dat
 5239  0:47    tar xvzf DATA_20101111.tgz DATA_20101111/sim/testout-0-0.dat
 5240  0:47    dri
 5241  0:47    dir
 5242  0:47    cd *1
 5243  0:47    cd sim
 5244  0:47    dir
 5245  0:47    cat testout-0-0.dat
 5246  0:48    grep '/-$/' *vcd
 5247  0:48    awk '/-$/{print}' *vcd
 5248  0:49    awk '/ -$/{print}' *vcd
 5249  0:55    ruby ../../../emu/printcheckpatternfromvcd.rb -s 20 -f *vcd -v rvalid,rdata
 5250  0:57    ruby ../../../emu/printcheckpatternfromvcd.rb -s 20 -f ../../../vhdl/g8top-0-0.vcd -
 5251  0:58    ruby ../../../emu/printcheckpatternfromvcd.rb -s 0 -f *vcd -v rvalid,rdata
 5252  1:00    ruby ../../../emu/printcheckpatternfromvcd.rb -s 0 -f *vcd -v rvalid,rdata -d
 5253  1:01    ruby ../../../emu/printcheckpatternfromvcd.rb -s 20 -f ../../../vhdl/g8top-0-0.vcd -
 5254  1:01    ruby ../../../emu/printcheckpatternfromvcd.rb -s 20 -f ../../../vhdl/g8top-0-0.vcd -
```

ここまでが作業ログ。問題点: rdata が色々ある。

```
$var reg      36 -    exp_rdata [35:0] $end
$var wire      1 r    rdata [35] $end
$var wire      1 s    rdata [34] $end
$var wire      1 t    rdata [33] $end
$var wire      1 u    rdata [32] $end
$var wire      1 v    rdata [31] $end
$var wire      1 w    rdata [30] $end
$var wire      1 x    rdata [29] $end
$var wire      1 y    rdata [28] $end
$var wire      1 z    rdata [27] $end
$var wire      1 {    rdata [26] $end
```

rvalid も

```
$var wire      1 8!   rvalid  $end
$var wire      1 8!   rvalid  $end
$var wire      1 C"   po_rvalid  $end
$var wire      1 W#   ro_rvalid  $end
$scope module u_rvalid $end
$var wire      1 {u3  rvalid  $end


>awk '/^[0-9]8\!$/{print}' *vcd
08!
18!
08!
18!
08!
18!
08!
18!
08!
18!
08!
18!


awk '/\{u3$/{print}' *vcd
X{u3
0{u3
1{u3
0{u3
1{u3
0{u3
1{u3
0{u3
1{u3
0{u3
1{u3
0{u3
1{u3
```

なので、 rvalid はどれをみても同じ？どれかが出力だと思うけど。

exp_rdata を拾うので OK

awk '/ -$/{print}' *vcd bx - b11111111110111101001111010010101101 1 - b11111111111111111111111111111111
- b11111110101010111101000000001000000 - b11111111111111111111111111111111 - b11111101010110100000011001011110
- b11111111111111111111111111111111 - b11010000010010111001101000000000100000 - b0 - b1110001
- b11100000000000000001010110000010111011 -

4C@ はなんだろう？

02 b1) b11111111110111101001111010010101101 1 -

02 b10) b11111111111111111111111111111111 - 02 b11) b11111110101010111101000000001000000
-

サイクルタイム 2856ps?

NEW 0 0 1fab40cbc 1 NEW 0 1 10000005d 1 d0140c03e 1 NEW 0 1 100000004 1 dabcd005e 0 60003 1 5000e0002 1 400150003 1 1001e0002 1 2b0003 1 100330001 1

ruby ../../../emu/printcheckpatternfromvcd.rb -f *vcd -v exp_rdata,rvalid -d -c 2.856

NEW 0 1 ffde9e95b 1 fffffff 1 fffffff 1 3faaf4040 1 fffffff 1 1fab40cbc 1 1fab40cbc 1 fffffff 1 d04b9a020 1 d04b9a020 1 NEW 0 1

NEW 0 0 ffde9e95b 0 ffde9e95b 1 fffffff 1 3faaf4040 1 fffffff 1 1fab40cbc 1 fffffff 1 d04b9a020 1

サイクルタイムがなんかおかしい？

最後のほう、eASIC:

500700002 1 100710002 1 100710002 1 500730002 1 100740002 1 100740002 1 500750002 1 500760002 1 500760002 1 100770002 1 100780002 1 500790002 1 500790002 1 5007a0002 1 1007b0002 1 1007b0002 1 5007c0002 1 1007d0002 1

こっち:

1006a0001 1 1006c0001 1 5006d0001 1 1006f0001 1 500700001 1 100710001 1 500730001 1 100740001 1 500750001 1 500760001 1 100770001 1 100780001 1 1007b0001 1 5007c0001 1 1007d0001 1

あ、この辺はパイプライン構成が違うせいだな。

8.20 2011/2/18

新しいデータ。

15914000 15914500

clock cycle:

1% 0% 1% 8000 ps

4ns クロックのように見える

```
 5245  23:36   cd src
  5246  23:36   cd grape8
  5247  23:36   cd g8
  5252  23:36   cd vhdl
  5261  23:38   ruby testrun0.rb -p 8 -b 6 -I 48
  5263  23:41   csh -f makepatterns.csh
```

g8work で

```
  6121  23:39   ruby ~/src/grape8/g8/emu/printcheckpatternfromvcd.rb -f DATA_20110208/SIM_20110130/v

   /home/makino/src/grape8/g8/vhdl/testout-0-0.pat
```

と

/home/makino/src/grape8/g8/g8work/xxx_0.0

を比較。

yuu

ffde9e95b 0 fffffff 1 3faaf4040 1 fffffff 1 1fab40cbc 1 fffffff 1 d04b9a020 1 0 1 71 1 70001585d 1 200337077 1 0 1 bfd41e9ba 1 fffffff 1 c02e5a5de 1 0 1 69374a0 1 0 1 10000005d 1 4000155b1 1 1ff567723 1

ffde9e95b 0ffffffff 3faaf4040 0ffffffff 1fab40cbc 0ffffffff d04b9a020 000000000 000000071 70001585d 200337077 000000000 bfd41e9ba 0ffffffff c02e5a5de 000000000 0069374a0 000000000 10000005d 4000155b1 1ff567723 0ffffffff 700199158 000000000 d0140c03e 000000000 e01ea3100

最初のほうは OK

最後のほうは

400610021 1 5006200e9 1 1006300e9 1 500640002 1 1006500e9 1 500670001 1 1006900e9 1 1006a0001 1 5006b0031 1 1006c00e9 1 5006d00cb 1 4006e0021 1 1006f00e9 1 4007000a9 1 100710051 1 720030 1 5007300e9 1 1007400e9 1 4007500a9 1 40076004b 1 1007700cb 1 780071 1 5007900e9 1 5007a00a1 1 1007b00e9 1 4007c00e1 1 7d00e1 1 1007e0008 1

5006200e9 1006300e9 500640002 1006500e9 500670001 1006900e9 1006a0001 5006b0031 1006c00e9 5006d00cb 4006e0021 1006f00e9 4007000a9 100710051 000720030 5007300e9 1007400e9 4007500a9 40076004b 1007700cb 000780071 5007900e9 5007a00a1 1007b00e9 4007c00e1 0007d00e1 1007e0008

割合大丈夫っぽい。

但し、使っている信号ではちょっとおかしい。

検証用のプログラムを一応書いた。

```
emu/checkpatterns2.csh
```

8.21 2011/2/21

メモリの読み書き並列動作用のテストパターン作成する必要あり。

まず、アドレスを 0 でないところから書くとか。

j 粒子メモリのアドレスは 20000h から始まる。

書き込みは、開始アドレス、バースト長を指定している

run レジスタは、上のほうの記述を更新する必要あり

ビット位置	説明
0-15	j 粒子数
16-31	開始アドレス (粒子番号)

アドレスマップも

アドレス	名称	説明
00h	RUN	ここへの書き込みが計算開始となる
01h	ICSTART	i 粒子バッファからパイプラインブロックへコピー開始
02h	FCSTART	パイプラインブロックから fo 結果バッファへコピー開始

ICSTART 及び FCSTART フィールドマップ

0-9	アドレス
16-25	ワード数 (粒子数ではないみたい)

なので、まずするべきテスト:

チップを 2x2 構成に戻す: g8top.evhd と g8block.evhd を編集、と思ったら、現在の g8top.evhd は 6 ブロック構成でないと動作しないんだった、、、とりあえず 6x1 構成でテスト。

j 粒子のスタートアドレス等を指定可能にする。これをつけてるのは g8chipsimtest ではないので、emu/g8chipsimtest_convert_pattern.rb と思われる。指定できるようにする必要があるもの:

- j 粒子スタートアドレス。これは書き込みと run データ生成の両方に使われる
- ip, fo バッファメモリの開始アドレス。一応それぞれ別々に指定。書き込み、転送命令の両方で使用。

```
ruby testrun0.rb -p 1 -b 6 -I 6 -t 0..0 -i 0..0
```

あれ、エラーになってら。

```
ruby testrun0.rb -p 1 -b 6 -I 6 -t 0..0 -i 0..0
```

これはテストパターン生成は成功した。

まず j 粒子。

```
5211  20:54   ruby testrun0.rb -p 1 -b 6 -I 6 -t 0..0 -i 0..0 -h
5212  20:54   ruby testrun0.rb -p 1 -b 6 -I 6 -t 0..0 -i 0..0 -J 1024


5219  21:06   ruby testrun0.rb -p 1 -b 6 -I 6 -t 0..1 -i 0..0 -J 1024
 acslab:
 12  21:04   csh -f testrun1.csh 1
5221  21:23   csh -f testrun2.csh 1
```

これはパスした。

これはどうか？

```
5223  21:24   ruby testrun0.rb -p 1 -b 6 -I 6 -t 0..1 -i 0..0 -J 1024 -f 512
```

オーバーラップしたデータにするには:

最初: jp, ip 送って計算始める。同時に動作チェックのためには、 ip のバッファからパイプラインへの転送を始めてすぐに次の ip のホストからバッファへの転送を始める。 で、計算始めたら jp の転送を始める。

あれ、fo の初期設定どこでしてるんだっけ？これは block では rtrig 信号、、、あれれ？その上では start_pr だけど、、、そもそも、fo の初期設定をするコマンドが定義も実装もされてないね。

fo についてはこれはあまり問題ではない？

何故これ複数ブロックで動いているんだろう？

あれ、そうではなくて、fo バッファへのコピーと読み出しは全く独立なはず？で、ブロック毎にバーストで読み出ししているみたい。で、 foreadaddress はパイプラインのアドレスなので 0 から始まるので OK。fo メモリへの読み出しと書き込みのどっちかしか今は設定されてない気が。

結局これ、 emu/g8chipsimtest_convert_pattern.rb のほうの問題だった。なので、ちゃんと全部設定できる。従って、上の論理で正しい。もう一度整理。

最初: jp, ip 送って計算始める。

同時動作のチェックのためには、 ip のバッファからパイプラインへの転送を始めてすぐに次の ip のホストからバッファへの転送を始める。 で、計算始めたら jp の転送を始める。fo についても、2 回目の転送をスタートさせてから 1 回目の read をする。

```
JP write     000000      111111   222222    333333
IP write           000111      222       333
IP transfer           0        1         2        3
calculation            00000000  11111111 22222222 33333333
FO transfer                     0         1        2        3
FO read                                   000      111      222333
```

制約: IP 転送と計算は並列に動作しない。これは性能の制約には (それほど) ならないはず。ブロック内の転送なので十分速い。

とりあえず、手で mqv ファイルを書いてみる？

上のをやるなら、ネイバーリストは読まないようにする必要あり。

まず、単純に 2 つを cat したファイルでテストしてみる。

```
5313  8:29    copy testin-0-0.mqv testin-0-0.mqv.non0addrs
5319  8:40    cat testin-0-1.mqv testin-0-0.mqv.non0addrs > mergetest0.mqv
```

この後手でちょっと編集。ヘッダ部をとる

```
5321  8:42    ruby ~/src/rubyutils/mqv/make_quartus_vwf.rb -i mergetest0.mqv -t g8top.vwf
```

acslab:

```
  44  8:46    copy mergetest0.vwf testin.vwf
  46  8:46    quartus_sim --read_settings_files=on --write_settings_files=off g8top -c g8top &


5328  9:05    getfilemm.csh g8top.vcd
5330  9:05    mv g8top.vcd mergetest0.vcd
5336  9:07    ruby ../emu/printcheckpatternfromvcd.rb -s 20 -f mergetest0.vcd -v rvalid,rdata | aw
5338  9:08    ruby ../emu/printcheckpatternfromvcd.rb -s 20 -f g8top-0-1.vcd -v rvalid,rdata | awk
5339  9:08    ruby ../emu/printcheckpatternfromvcd.rb -s 20 -f g8top-0-0.vcd -v rvalid,rdata | awk
5341  9:08    cat /tmp/emuout1 /tmp/emuout0 > /tmp/emuout10a
5343  9:09    diff -C0 /tmp/emuout10 /tmp/emuout10a
```

merge してないテストと比較。これはパスした。

2 個目のテーブルロードを消したもの。もちろん問題なかった。

結局、jp, ip, fo についてシミュレーションでは concurrent read/write もちろん問題ないので、そういうテストパターンを作る必要あり。

テストパターンの生成方法を変更して、バラバラにできるようなコメントをいれた。

パターンをだす順番

```
header 1
cutoff 1
```

```
ip     1
jp     1
ipmove 1 (dummy removed)
ip     2
run    1(dummy removed)
jp     2
fomove 1
ipmove 2
run    2
fomove 2
fo     1
fo     2
```

生成方法は結局

```
5568  20:36   ruby testrun0.rb -p 1 -b 6 -I 6 -t 0..0 -i 0..1 -J 1024 -f 512
5571  20:37   mv testin-0-1.mqv testin-0-1.mqv.non0adrs
5572  20:37   ruby testrun0.rb -p 1 -b 6 -I 6 -t 0..0 -i 0..1
5582  20:49   ruby mergepatterns.rb -F testin-0-0.mqv -f testin-0-1.mqv.non0adrs > mergetest1.mqv
5594  20:55   ruby ~/src/rubyutils/mqv/make_quartus_vwf.rb -i mergetest1.mqv -t g8top.vwf ; putfil


  98  20:54   copy mergetest1.vwf testin.vwf
  99  20:54   quartus_sim --read_settings_files=on --write_settings_files=off g8top -c g8top &
```

あれ、何かが不足している感じ？良くわからないので作り直してみると:

```
5606  21:25   ruby testrun0.rb -p 1 -b 6 -I 6 -t 0..0 -i 0..1 -J 1024 -f 512
5607  21:26   mv testin-0-0.mqv testin-0-0.mqv.non0adrs
5608  21:26   ruby testrun0.rb -p 1 -b 6 -I 6 -t 0..0 -i 0..1
5609  21:26   ruby mergepatterns.rb -f testin-0-1.mqv -F testin-0-0.mqv.non0adrs > mergetest1.mqv
5610  21:27   diff -C0 mergetest0.mqv mergetest1.mqv
5611  21:28   ruby ~/src/rubyutils/mqv/make_quartus_vwf.rb -i mergetest1.mqv -t g8top.vwf ; putfil
```

これは mergetest0 と dummy cycle の数以外では変わらない。
でも、それだけのために結果が違っている。

8.22 2011/2/25

FOMOVE をかけてから RUN が入るまでのサイクル数の問題だった。

```
#FOMOVE start
0 0        2 0 1 0 0 00
0 0 000a0000 1 0 0 0 00
#DUMMY start
0 0        0 0 0 0 0 00
0 0        0 0 0 0 0 00
```

```
0 0         0 0 0 0 0 00
0 0         0 0 0 0 0 00
0 0         0 0 0 0 0 00
0 0         0 0 0 0 0 00
0 0         0 0 0 0 0 00
0 0         0 0 0 0 0 00
#DUMMY end
#FOMOVE end
#IPMOVE start
0 0         1 0 1 0 0 00
0 0 00060200 1 0 0 0 00
#DUMMY start
0 0         0 0 0 0 0 00
0 0         0 0 0 0 0 00
0 0         0 0 0 0 0 00
0 0         0 0 0 0 0 00
0 0         0 0 0 0 0 00
0 0         0 0 0 0 0 00
0 0         0 0 0 0 0 00
0 0         0 0 0 0 0 00
#DUMMY end
#IPMOVE end
#RUN start
0 0         0 0 1 0 0 00
0 0 04000080 1 0 0 0 00
```

上のは動いたもの。 19 サイクルあり。元々のは 14 サイクルだった。テストのため、 IPMOVE と FOMOVE の同時動作も試してみる。

これが動いたら、自動生成の方法のほうを変更する。

まだなんか間違ってる？

r1022 で本当に答えがあっているか？

r1022 では確かに答えがあう。

RUN のあとのダミーサイクルは必要？んなことはないはずだが、、、

でも、これいれたら答えがあった (r1025)

なので、現在は自動生成でも答があう。

さて、どういうテストをいれるべき？

真面目に考えると

パイプライン

1. 座標の加減算器のテスト
2. 距離 2 乗の計算、加算のテスト
3. 距離の -1.5 乗のとこ、そのあとの乗算のテスト
4. 質量との乗算のテスト
5. 積算のテスト

全ビットテストが望ましいが無理。上のほうから回してみる必要はあり。

ip, fo バッファ: 48 パイプラインでテストすれば OK

カットオフ:返ってきた力のグラフを書いてみる必要あり。

加減算器のテスト:

オフセットは 24 ビットになっている。大きな数から大きな数を引くのでどうか？

引き算の答は同じになるとして。粒子数は物理パイプライン数まででまあいいや。上は 32 ビットなので 64 から始める。

テストパターンに入れるべきもの:

testin.pat と testout.pat

で、作ってみたら答えがあわない、、、えええええ？

vector 10 differs sim:a73d3a00 emu:a73f1bb0 vector 12 differs sim:58bd3f80 emu:58bf1c70 vector 14 differs sim:58c2c600 emu:58c0e450 vector 16 differs sim:16f0a000 emu:16b4a800 vector 18 differs sim:00000002 emu:00000001

(1,0,0) - (0,0,0) でも rnnb が変？

力とポテンシャルはあう。何故？

ちょっとよくわからないので、ランダムテストをもう一度。 6 粒子で。あれ、これも通らない。なんかもっと違うところが間違ってる？

testin-0-0 を流してみると、、、これは問題ない。

パターン生成のスクリプトが間違っていた。修正してはみた。

なにが間違ってたのかもうひとつわからないが、全く問題なく動くようになった。

次は距離 2 乗のテスト。

テストの観点は？まず、 i 粒子の数がどうなってるか確認する必要あり。

g8chipsimtest は変更した。複数の計算ブロックがあってもデータはでる。

g8chipsimtest_convert_pattern.rb も変更の必要あり。

これは、今の構造は向いてない、、、全部読んでからなんかする構造になっていて修正ができなくはないがあんまり良くない。

仕様変更:

g8chipsimtest の出力に

```
G8PIPE PRINT_CUTOFF
G8PIPE CLEAR_IP
G8PIPE PRINT_IP
G8PIPE CLEAR_JP
G8PIPE PRINT_JP
G8PIPE RUN
G8PIPE FOREAD
G8PIPE CLEAR_NBL
G8PIPE PRINT_NBL
```

を追加して、これらに従って動作するようにする。

この変更はできたように見える。

で、そうすると、 ip を沢山、というのが容易にできるのでそっちを色々テスト。

距離の自乗の加減算用のテスト:

既に対数表現なので、x 固定して y と z をそれぞれ別々に与えてみればよいかな。

粒子生成コードは書いた。

あ、ソフトニングのテストもしないと。これも導入した。

-1.5 乗のユニットと関数テーブルのテスト。

これは、 -1.5 乗については適当にに広いレンジで r を変化させれば OK。関数テーブルは、、、 xi, xj を別々に変化させる。

-1.5 乗の変化レンジは -10^4 から 100 としておく。6 桁なので、128 粒子として、、、まあ、計算して。

次は質量を変化させるテスト。

8.23 2011/3/3

パイプラインのテストは一応一通りいれた。

mergetest0 r1036 は通らない気がするので 1014 でテスト。

---

r1018 — makino — 2011-02-25 07:27:47 +0900 (金, 25 2 月 2011) — 2 lines

vhdl/mergetest0.mqv fo also concurrent read/write

---

r1017 — makino — 2011-02-25 06:44:44 +0900 (金, 25 2 月 2011) — 2 lines

vhdl/mergetest0.mqv neighbour list read removed

---

r1016 — makino — 2011-02-25 06:16:38 +0900 (金, 25 2 月 2011) — 2 lines

vhdl/mergetest0.mqv dummy cycles removed

---

r1015 — makino — 2011-02-25 05:41:47 +0900 (金, 25 2 月 2011) — 2 lines

vhdl/mergetest0.mqv jp send simultaneously

---

r1014 — makino — 2011-02-25 04:42:44 +0900 (金, 25 2 月 2011) — 2 lines

vhdl/mergetest0.mqv slightly modified version. Still passed

---

r1013 — makino — 2011-02-24 09:09:58 +0900 (木, 24 2 月 2011) — 2 lines

vhdl/mergetest0.mqv added

```
5033  3:04    svn up -r1014 mergetest0.mqv
5035  3:05    ruby ~/src/rubyutils/mqv/make_quartus_vwf.rb -i mergetest0.mqv -t g8top.vwf
5038  3:05    putfilemm.csh mergetest0.vwf
```

r1014 は以下のような感じ。これは /tmp/emuout10 と一致していて確かに正しい。

1 a019cf712 1 0 1 b01064f10 1 0 1 603bf9db1 1 0 1 60394e0c0 1 0 1 100000034 1 500015931 1 603a1d548 1 0 1 9fe69881a 1 fffffff 1 cfea2749a 1 fffffff 1 703544640 1 0 1 100000057 1 500015985 1 904b4f98c 1 0 1 d0298246e 1 0

削っていいかどうかの確認

user(replaced)(G)89709DFF/u_core/ublock_1/upipe_3/u103/u1/lpm_mult_component/cnv/stage1_01_reg[0]ublock_0/upip
と同じ user(replaced)(G)89708DFF/u_core/ublock_1/upipe_3/u103/u1/lpm_mult_component/cnv/stage1_01_reg[1]ublock
と同じ user(replaced)(G)89649DFF/u_core/ublock_1/upipe_3/u103/u1/lpm_mult_component/cnv/stage2_05_reg[0]ublock
と同じ user(replaced)(G)89648DFF/u_core/ublock_1/upipe_3/u103/u1/lpm_mult_component/cnv/stage2_05_reg[1]ublock
と同じ user(replaced)(G)89647DFF/u_core/ublock_1/upipe_3/u103/u1/lpm_mult_component/cnv/stage2_05_reg[2]ublock
と同じ user(replaced)(G)89646DFF/u_core/ublock_1/upipe_3/u103/u1/lpm_mult_component/cnv/stage2_05_reg[3]ublock
と同じ user(replaced)(G)89645DFF/u_core/ublock_1/upipe_3/u103/u1/lpm_mult_component/cnv/stage2_05_reg[4]ublock
と同じ user(replaced)(G)89644DFF/u_core/ublock_1/upipe_3/u103/u1/lpm_mult_component/cnv/stage2_05_reg[5]ublock
と同じ

- /u_core/ublock_X/upipe_Y/ の形式のものについては全パイプラインで同じかどうかチェック
  - /u_core/ublock_X/ でそれ以外のものについては全ブロックで同じかどう

  かチェック
- それ以外はプリントアウトしてみる

```
ruby check-unr*rb rtl2synthesis_unreachable_nowrapper.csv
```

でブロック、パイプライン間で同じであることのチェックはした。