

# GRAPE-6 システム機能記述書

Jun Makino  
The University of Tokyo

Version 0.0 — Document Created: June 11, 1997

Version 0.1 — August 7, 1997

Version 0.2 — October 23, 1997

Version 0.3 — October 24, 1997

Version 0.4 — December 5, 1997

Version 0.5 — December 15, 1997

Version 0.6 — February 17, 1998

Version 0.7 — April 21, 1998

Version 0.8 — August 7, 1998

1998 年 11 月 18 日





# 目 次

<b>1</b>	<b>変更記録</b>	<b>1</b>
1.1	Version 0.0 . . . . .	1
1.2	Version 0.1 . . . . .	1
1.3	Version 0.2 . . . . .	1
1.4	Version 0.2 . . . . .	1
1.5	Version 0.4 . . . . .	1
1.6	Version 0.5 . . . . .	2
1.7	Version 0.6 . . . . .	2
1.8	Version 0.7 . . . . .	3
1.9	Version 0.8 . . . . .	4
<b>2</b>	<b>概要</b>	<b>7</b>
2.1	概要 . . . . .	7
2.2	TODO リスト . . . . .	7
<b>3</b>	<b>アーキテクチャに関する検討</b>	<b>9</b>
3.1	GRAPE-6 の概要 . . . . .	9
3.2	基本機能 . . . . .	9
3.3	GRAPE-6 プロセッサチップの要求性能 . . . . .	10
3.3.1	システム I/O 性能 . . . . .	10
3.3.2	メモリ別チップの場合 . . . . .	12
3.3.3	I/O ポートの構成 . . . . .	12
3.4	プロセッサ間およびプロセッサ-ホスト間結合ネットワーク . . . . .	15
3.4.1	並列化とネットワーク構成に対する基本方針 . . . . .	15
3.4.2	$j$ 粒子書き込みネットワーク . . . . .	16
3.4.3	バタフライネットワークを使う構成 . . . . .	17
3.4.4	代案 1 . . . . .	18
3.4.5	代案 2 . . . . .	18
3.4.6	NREAD の実装 . . . . .	19
3.4.7	ネットワーク構成に関するまとめ . . . . .	21
<b>4</b>	<b>エラーチェック/コレクションについて</b>	<b>23</b>
4.1	エラー検出 . . . . .	23
4.2	エラーからの回復 . . . . .	24
4.3	耐障害性 . . . . .	25
4.4	故障診断 . . . . .	27

<b>5</b>	<b>アーキテクチャに関するまとめ</b>	<b>29</b>
5.1	チップの構成	29
5.2	ボードの構成	29
5.2.1	IP ポート	30
5.2.2	JP ポート	30
5.2.3	FO ポート	30
5.2.4	ボード間接続	30
5.3	クラスタの構成	30
5.3.1	IP ポート	30
5.3.2	JP ポート	31
5.3.3	FO ポート	31
5.4	全体の構成	31
<b>6</b>	<b>G6 チップ機能の検討</b>	<b>33</b>
6.1	予測子パイプライン	33
6.1.1	a) $DT = T - TJ$	35
6.1.2	多項式パイプラインの精度とブロック固定小数点表現に関する一般論	36
6.1.3	乗算器	38
6.1.4	定数乗算器	39
6.1.5	加算器	39
6.1.6	b) $K4 = DT * A2BY18$	39
6.1.7	c) $K4A = K4 * 0.75$	39
6.1.8	d) $K3 = K4A + A1BY6$	39
6.1.9	e) $K3A = K3 * DT$	40
6.1.10	f) $K2 = K3A + ABY2$	40
6.1.11	g) $K2A = K2 * DT$	40
6.1.12	h) $K1 = K2A + V$	40
6.1.13	i) $K1A = K1 * DT$	40
6.1.14	j) $XP = X + K1A$	40
6.1.15	k) $K3C = K4 + A1BY6$	41
6.1.16	l) $K3D = K3C * 1.5$	41
6.1.17	l') $K3E = K3D * DT$	41
6.1.18	m) $K2B = K3E + ABY2$	41
6.1.19	n) $K2C = K2B * 2$	41
6.1.20	o) $K2D = K2C * DT$	41
6.1.21	p) $VP = V + K2D$	41
6.1.22	ブロック浮動小数点における指数の決定	42
6.1.23	テストモード	45
6.1.24	メモリとのインターフェース	46
6.1.25	チップ内他ユニットとのインターフェース	47
6.2	相互作用パイプライン	48
6.2.1	$dx(K) = X(K,J) - X(K,I)$	50
6.2.2	$dx(K)$ の乗算	52
6.2.3	$dr2 = dx(1)**2 + dx(2)**2 + dx(3)**2 + eps2$	52
6.2.4	$1/r^3$ と $1/r$	52
6.2.5	カットオフユニット	54
6.2.6	質量とのかけ算	60
6.2.7	ポテンシャルの積算	61

6.2.8	$dx(k)$ との乗算 . . . . .	61
6.2.9	加速度の積算 . . . . .	61
6.2.10	粒子一つからの力の誤差について . . . . .	61
6.2.11	$dv(K) = XDOT(K,J) - XDOT(K,I)$ . . . . .	63
6.2.12	$dx(k)*dv(k)$ . . . . .	63
6.2.13	$drdv = drdv + \dots$ . . . . .	63
6.2.14	$j1 = dx(k)*mdr5i$ . . . . .	63
6.2.15	$j2 = dv(K) *dr3i$ . . . . .	63
6.2.16	$j1 + j2$ . . . . .	63
6.2.17	$FD(k) +=$ . . . . .	63
6.2.18	ネイバーリスト . . . . .	64
6.2.19	ニアレストネイバー . . . . .	64
6.2.20	実際のパイプライン構成 . . . . .	65
6.2.21	固定小数点表示にする時の指数の扱いについて . . . . .	67
6.2.22	テスト用制御信号 . . . . .	72
6.2.23	チップ内入出力インターフェース . . . . .	76
6.3	制御回路 . . . . .	80
6.3.1	クロック . . . . .	80
6.3.2	内部シーケンサ . . . . .	83
6.3.3	JP メモリライト . . . . .	83
6.3.4	IP レジスタライト . . . . .	88
6.3.5	結果読み出し . . . . .	93
6.3.6	計算開始／終了の制御とメモリデータのパイプラインへの供給 . . . . .	99
6.3.7	メモリデータ ECC 生成方法について . . . . .	101
6.4	チップ全体 . . . . .	104
6.4.1	チップ全体のテストについて . . . . .	106
7	<b>G6 チップの詳細仕様</b>	<b>107</b>
7.1	チップ全体 . . . . .	107
7.2	予測子パイプライン . . . . .	107
7.2.1	予測子パイプライン全体 . . . . .	107
7.2.2	a) $DT = T - TJ$ . . . . .	110
7.2.3	定数乗算器 . . . . .	110
7.2.4	加算器 . . . . .	110
7.2.5	b) $K4 = DT * A2BY18$ . . . . .	110
7.2.6	c) $K4A = K4 * 0.75$ . . . . .	111
7.2.7	d) $K3 = K4A + A1BY6$ . . . . .	111
7.2.8	e) $K3A = K3 * DT$ . . . . .	111
7.2.9	f) $K2 = K3A + ABY2$ . . . . .	112
7.2.10	g) $K2A = K2 * DT$ . . . . .	112
7.2.11	h) $K1 = K2A + V$ . . . . .	112
7.2.12	i) $K1A = K1 * DT$ . . . . .	112
7.2.13	j) $XP = X + K1A$ . . . . .	112
7.2.14	k) $K3C = K4 + A1BY6$ . . . . .	113
7.2.15	l) $K3D = K3C * 1.5$ . . . . .	113
7.2.16	l') $K3E = K3D * DT$ . . . . .	113
7.2.17	m) $K2B = K3E + ABY2$ . . . . .	113
7.2.18	n) $K2C = K2B * 2$ . . . . .	113

7.2.19 o) K2D = K2C * DT . . . . .	113
7.2.20 p) VP = V + K2D . . . . .	113
7.2.21 チップ内他ユニットとのインターフェース . . . . .	114
<b>7.3 相互作用パイプライン . . . . .</b>	<b>115</b>
7.3.1 a)dx(K) = X(K,J) - X(K,I) . . . . .	117
7.3.2 b)dx2(K) = dx(K)*dx(K) . . . . .	118
7.3.3 c0)dr2xy = dx2(1) + dx2(2) . . . . .	118
7.3.4 c1)dr2ze = dx2(3) + eps2 . . . . .	119
7.3.5 c2)dr2 = dr2xy + dr2ze . . . . .	119
7.3.6 d) r5inv = dr2**2.5 . . . . .	119
7.3.7 e) mr5inv = r5inv*m . . . . .	120
7.3.8 f) mr3inv = mr5inv*dr2 . . . . .	120
7.3.9 g) mrinv = mr3inv*dr2 . . . . .	120
7.3.10 h) r = sqrt(dr2) . . . . .	120
7.3.11 i0) pcut = pf(r) . . . . .	120
7.3.12 j0) mrinv2 = mrinv*pcut . . . . .	121
7.3.13 j1) mr3inv2 = mr3inv*fcut . . . . .	121
7.3.14 k) phi = phi + mrinv2 . . . . .	122
7.3.15 l) f(k) = mr3inv2*dx(k) . . . . .	122
7.3.16 m) acc(k) = acc(k) + f(k) . . . . .	122
7.3.17 n)dv(K) = XDOT(K,J) - XDOT(K,I) . . . . .	123
7.3.18 o) xv(k) = dx(k)*dv(k) . . . . .	123
7.3.19 p)drdvxy = xv(1) + xv(2) . . . . .	123
7.3.20 q)drdv = drdvxy + xv(3) . . . . .	123
7.3.21 r)drdv3 = drdvxy*3 . . . . .	123
7.3.22 s)mdrdv3byr5 = drdv3 * mr5inv . . . . .	123
7.3.23 t) t1(k) = dv(k)*mdr3inv . . . . .	123
7.3.24 u) t2(k) = dx(k)*mdrdv3byr5 . . . . .	123
7.3.25 v) j(k) = t1(k) + t2(k) . . . . .	123
7.3.26 w)jerk(k) = jerk(k) + j . . . . .	123
7.3.27 x) ネイバーリスト . . . . .	124
7.3.28 y) ニアレストネイバー . . . . .	124
7.3.29 チップ内入出力インターフェース . . . . .	126
<b>7.4 制御回路 . . . . .</b>	<b>130</b>
7.4.1 クロック . . . . .	130
7.4.2 内部シーケンサ . . . . .	130
7.4.3 JPW . . . . .	130
7.4.4 IPW . . . . .	134
7.4.5 FO . . . . .	138
7.4.6 計算開始／終了の制御とメモリデータのパイプラインへの供給 . . . . .	143
<b>8 チップ入出力詳細 . . . . .</b>	<b>147</b>
8.1 入出力プロトコル . . . . .	147
8.2 入力ポート一般 . . . . .	147
8.3 JP ポート . . . . .	148
8.4 IP ポート . . . . .	148
8.5 出力インターフェース . . . . .	149
8.6 メモリインターフェース . . . . .	151

# 図 目 次

3.1 計算ステップの時間依存関係 . . . . .	13
3.2 GRAPE-6 プロセッサモジュールの構成単位 . . . . .	14
3.3 16 プロセッサの場合のネットワーク構成例 . . . . .	16
3.4 部分的にバタフライを使ったネットワーク構成例 . . . . .	18
3.5 最小限の放送機能をサポートするネットワーク . . . . .	19
3.6 現時点での全体構成案 . . . . .	22
 6.1 予測子パイプライン . . . . .	34
6.2 $\Delta t$ 計算回路 . . . . .	37
6.3 予測子での乗算器の基本構成 . . . . .	38
6.4 予測子での加算器の基本構成 . . . . .	40
6.5 予測子パイプライン内部インターフェース . . . . .	47
6.6 予測子入力タイムチャート . . . . .	48
6.7 予測子 TI 書き込みタイムチャート . . . . .	48
6.8 予測子出力タイムチャート . . . . .	49
6.9 相互作用パイプライン . . . . .	50
6.10 乗算器のシステムテックバイアス (左) と rms 誤差 (右) . . . . .	53
6.11 乗算器のシステムテックバイアス (左) と rms 誤差 (右) のサンプル数への依存性 . . . . .	54
6.12 符号なし加算器のシステムテックバイアス (左) と rms 誤差 (右) . . . . .	55
6.13 符号なし加算器のシステムテックバイアス (左) と rms 誤差 (右) のサンプル数への依存性 . . . . .	56
6.14 補間器による幂乗計算の自乗平均相対誤差。左上は $x^{-1/2}$ , 右上は $x^{-3/2}$ , 下は $x^{-5/2}$ である。それぞれ、横軸は 0 次の項の仮数部ビット数であり、縦軸は r.m.s. 誤差である。三角、四角、五角および丸はそれぞれテーブルエントリ数が $2^7, 2^8, 2^9, 2^{10}$ の場合である。 . . . . .	57
6.15 ポテンシャル (左) と力 (右) の、ガウシアンカットオフの場合の補正項とその 3 次までの高次導関数。いずれも、極値、ゼロ点の多いものほど高次である。 . . . . .	58
6.16 ポтенシャル (左) と力 (右) の、ガウシアンカットオフの場合の誤差一定にするための補間区間の相対長を 2 次補間の場合に示す。いずれも、極値、ゼロ点の多いものほど高次である。 . . . . .	59
6.17 テーブルエントリ 64、語長 12 ビットの場合のポテンシャル (左) と力 (右) のカットオフ関数の絶対誤差 . . . . .	60
6.18 ペアワイズの力の RMS 誤差を距離の関数として示す。サンプル数は各点あたり 1000 個であり、距離は 1.1 倍間隔である。左は入力座標のオフセットが 24 ビット、右は 40 ビットである。それぞれ、最終段の加算器のバイアスをいくつか変えてプロットした。 . . . . .	62
6.19 ネイバーリスト . . . . .	64
6.20 最近接粒子 . . . . .	65

6.21	相互作用パイプライン内部インターフェース . . . . .	76
6.22	予測子からの入力のタイムチャート . . . . .	77
6.23	相互作用パイプライン入力データタイムチャート . . . . .	80
6.24	入力ポートタイムチャート . . . . .	84
6.25	入力ポート詳細タイムチャート . . . . .	84
6.26	JP データ受けとり回路 . . . . .	85
6.27	JPW 制御部インターフェース . . . . .	87
6.28	JPW メモリポート詳細タイムチャート . . . . .	88
6.29	IP ポート制御回路とパイプラインのタイミング . . . . .	89
6.30	IPW 制御部インターフェース . . . . .	91
6.31	IPW 制御部概念図 . . . . .	93
6.32	FO ポート制御回路とパイプラインのタイミング . . . . .	94
6.33	出力ポートタイムチャート（ウェイトなし） . . . . .	96
6.34	出力ポートタイムチャート（ウェイトあり） . . . . .	96
6.35	出力ポートタイムチャート（最初に WD アサート） . . . . .	96
6.36	FO 制御部インターフェース . . . . .	97
6.37	CALC 制御部インターフェース . . . . .	101
6.38	CALC 部動作 . . . . .	102
6.39	G6 チップの入出力ピン定義 . . . . .	105
6.40	G6 チップのトップレベルブロック図 . . . . .	105
7.1	G6 チップの入出力ピン定義 . . . . .	108
7.2	G6 チップのトップレベルブロック図 . . . . .	108
7.3	予測子パイプライン . . . . .	109
7.4	$\Delta t$ 計算回路 . . . . .	110
7.5	予測子での乗算器の基本構成 . . . . .	111
7.6	予測子での加算器の基本構成 . . . . .	112
7.7	予測子パイプライン内部インターフェース . . . . .	114
7.8	予測子入力タイムチャート . . . . .	115
7.9	予測子 TI 書き込みタイムチャート . . . . .	115
7.10	予測子出力タイムチャート . . . . .	117
7.11	相互作用パイプライン . . . . .	117
7.12	ネイバーリスト . . . . .	124
7.13	最近接粒子 . . . . .	125
7.14	相互作用パイプライン内部インターフェース . . . . .	125
7.15	予測子からの入力のタイムチャート . . . . .	126
7.16	相互作用パイプライン入力データタイムチャート . . . . .	130
7.17	入力ポートタイムチャート . . . . .	131
7.18	入力ポート詳細タイムチャート . . . . .	132
7.19	JP データ受けとり回路 . . . . .	132
7.20	JPW 制御部インターフェース . . . . .	133
7.21	IP ポート制御回路とパイプラインのタイミング . . . . .	135
7.22	IPW 制御部インターフェース . . . . .	137
7.23	IPW 制御部概念図 . . . . .	137
7.24	FO ポート制御回路とパイプラインのタイミング . . . . .	140
7.25	出力ポートタイムチャート（ウェイトなし） . . . . .	141
7.26	出力ポートタイムチャート（ウェイトあり） . . . . .	141
7.27	出力ポートタイムチャート（最初に WD アサート） . . . . .	141

7.28 FO 制御部インターフェース . . . . .	142
7.29 CALC 制御部インターフェース . . . . .	144
7.30 CALC 部動作 . . . . .	146
8.1 入力ポートタイムチャート . . . . .	148
8.2 出力ポートタイムチャート（ウェイトなし） . . . . .	149
8.3 出力ポートタイムチャート（ウェイトあり） . . . . .	150
8.4 出力ポートタイムチャート（最初に WD アサート） . . . . .	150

# 第1章 変更記録

## 1.1 Version 0.0

1997/6/11 作成

## 1.2 Version 0.1

1997/8/7 タイトルを機能記述書と変更。 book style にする。予測子と演算パイプラインの概要を記述。

1997/8/27 基本方針は SSRAM 外付けとする。具体的には 4Mbit (128kx36) のものを 2つというものが基本。動作速度はパイプラインチップの内部クロックと同一とする。実装は C4 (モトローラ等) または 119pin BGA (7 × 17, 14 × 22mm) を使うものとする。

コアが 2.5V で動作するものはまだあまりないような気もするので、2.5V と 3.3V の 2 電源になるかもしれない。

1997/10/13 ネットワーク回りをこの日から全面修正。

## 1.3 Version 0.2

1997/10/23 チップ I/O ポートに関する検討を詳細化。

## 1.4 Version 0.2

1997/10/24 タイトルを GRAPE-6 システム機能記述書と変更。それに応じて全体の構成を変更。

## 1.5 Version 0.4

1997/12/5 前のバージョンからは無数の変更を加えた... 詳細は略。

## 1.6 Version 0.5

1997/12/5

予測子と、相互作用の計算のうち加速度の分についての検討部分が一応完成した。特に、相互作用の計算についてはペアワイズの力の相対誤差評価を済ませた。時間導関数の部分はまだである。また、予測子と、相互作用の計算についても、実際のシミュレーションプログラムに組み込んでの評価はまだすんでいない。

1997/12/21

最終段でのオーバーフロー、アンダーフロー時の動作を決める必要がある。

1997/12/21

SUN 版 jlatex から、割と新しい platex2e を使うように変更した。

1997/12/27

ボード間接続として、LVDS ベースの Channel Link を coaxial FC などのシリアル転送に替えて使うことにした。これは決定というわけではないが、ほぼこの線でいくことにしたい。

1998/2/5

予測子パイプに関するいくつかのバグを修正した。

- 速度の最終段での正規化についての記述を追加した
- 速度の途中で乗算器が一つ落ちていたのを追加した

1998/2/7 加速度等のホストでのデータ変換についての記述を加えた

1998/2/12

单一チップ+メモリのフルシミュレータの構築が終了したので、その実際の  $N$  体積分に使った結果の評価のセクションを追加した。

## 1.7 Version 0.6

1998/2/17

各パイプラインのインターフェース記述を明確化した。さらに、制御ユニットの仕様を加えた。

1998/2/25

相互作用パイプラインの記述で、演算器がすべて示されていたわけではなかったものをすべてしめす形に変更した。

1998/3/1

予測子パイプラインの演算順序が

- n)  $K2C = K2B * DT$
- o)  $K2D = K2C * 2$

となっていたのを

- n)  $K2C = K2B * 2$
- o)  $K2D = K2C * DT$

とした。これは、どちらでも精度に違いができるわけではないが、シミュレータに合わせた。

1998/3/2 3章にクラスタ間並列化に関する記述(3.4.1節)を追加した。

1998/3/5 相互作用パイプラインのレジスタライトポートの幅を32ビットから36ビットとした。これは、内部レジスタに36ビットのものがあるからである。これに応じて制御回路の動作変更(データ形式変換)が必要になるが、まだここは表現されていない。

また、予測子のmjポートについても同様な変更を行なった。

## 1.8 Version 0.7

1998/4/21

1. 7.2.20節に、VPの形式についての記述を追加した。
2. 6.2節の最初に、VPの形式についての記述を追加した。
3. 6.2.5節において、テーブルの出力ビット長を変更した(0,1次がそれぞれ13,12ビット)さらに、1次の項だけに符号をつけることにした。
4. 6.2.18節で、インデックスが一致するものはスキップするという記述を追加した
5. 制御部-相互作用インターフェース信号定義(表6.9)で、ATを8ビットに変更した。
6. 相互作用パイプライトレジスタマップ(表6.2.23)で、R0,CTABのアドレスエントリを1ビットずつ増やした。これは、テーブルが2個あるからである(但しR0は共通)。さらに、それに関する説明をそのあとに追加した。
7. 表6.12の下にネイバーリストのサイズについての記述を加えた。
8. 7.3節の最初に、VPの形式についての記述を追加した。
9. 7.3.3節に、浮動小数点の例外処理についての記述を加えた。
10. 7.3.7節に、質量にも符号が必要な旨明記した。
11. 7.3.11節において、テーブルの出力ビット長を変更した(0,1次がそれぞれ13,12ビット)さらに、1次の項だけに符号をつけることにした。
12. 7.3.13節に、指数のオフセット処理とインデックスが一致したらスキップする処理についての説明を加えた。
13. 7.3.22節で、mr5invの丸め処理に関する説明を追加した。
14. 7.3.27節に、インデックスが一致したらスキップする処理についての説明を加えた。
15. 7.3.28節に、インデックスが一致したらスキップする処理についての説明を加えた。
16. 相互作用パイプライトレジスタマップ(表7.3.29)で、R0,CTABのアドレスエントリを1ビットずつ増やした。これは、テーブルが2個あるからである(但しR0は共通)。さらに、それに関する説明をそのあとに追加した。
17. 同表に、SCALESのエントリを追加し、その説明を加えた

## 1.9 Version 0.8

1998/4/28—

1. 表7.16を修正し、ディレイ段数を増やした。また、ODLYの定義を変更した。
2. 7.2.21節において、座標データがシーケンシャルにでるように変更した。

3. 予測子の2次導関数のビット数を12ビットから10に変更した。また、これが符号を含むものかどうか明確でなかったが、符号を含まないものであるとした。
4. あちこちのテーブルで SCALES の記述が落ちていたので、なるべく直した。
5. ネイバーリストのアドレス割り当てと、ワード内データ割り当てについての記述を、6, 7両章に加えた。チップからのデータ出力はもうちょっと詳しく書く必要がある。
6. JPW から EN 入力信号をなくした。これは使わない。
7. CALC から VDOUT に関する記述をなくし、RUN は VD と同期することにした。
8. FO の NBREAD の時の記述を詳細化した。
9. FO の内部レジスタに UN を追加し、ホストポート線も非使用を示す信号をつけた。
10. IPW の出力に WETL, WETH を追加した。
11. ランダム読みだしでは WD を見ないことにする。これはまだ仕様書に反映されていない。
12. (98/8/4) 32ビット IEEE形式から36ビット G6内部表現への変換形式を変更した。6.1.24にあるものが正しい。これと一致していない表記があればそれは間違いなので注意。
13. (98/8/4) 6.3.3節で、メモリに書く時のタイミングシーケンスの詳細を与えた。
14. (1998/8/4) FO ポートの STS 信号線の極性を low active に統一した（はず）。
15. (1998/8/5) ネイバーリストのデータを返すところの記述のうち、仕様書に明記されていなかったものを 6.2.18 節に追加した。また、FO ポートから返るネイバーリストの第一ワード（カウンタ出力）の最上位ビットを1固定にすることようにした。
16. (1998/8/10) LFORCE についての説明が CALC シーケンサの説明になかったので追加した。
17. (1998/8/10) FO の WD, ND についての説明を追加した。
18. (1998/8/21) FO INACTIVE のデータ形式を変更。IPW からのレジスタ書き込みによっていたが、これでは全チップに書き込まれてしまって話にならないので、VCID, MASK をデータのなかに埋め込むように変更。
19. (1998/9/15) FO INACTIVE の属性が明確ではなかったので、表を訂正した。=0 のときにアクティブである。

## 第2章 概要

### 2.1 概要

このドキュメントは、GRAPE-6 システムの機能記述書である。なお、動作、機能に関する検討、評価の部分と、実際に決定した仕様とは、このドキュメントのなかでは必ずしも分離されていない。最終的な仕様記述書はこのドキュメントとは別につくられることになるであろう。

3 章においては、GRAPE-6 システム全体の構成に関する検討がなされる。4 章では、エラー検出、障害からの回復などに関する検討がなされる。6 章では、G6 チップのより詳細な仕様、実現についての検討がなされる。

### 2.2 TODO リスト

- シミュレータの各ユニットに対して、テストデータ作成プログラムと出力インターフェースを与えないといけないかな。これは予測子については完了(1998/3/1)
- FO のところの動作が明確でない。いつ動作が始まるか(計算が終ったら勝手に)データを返す順番(アドレス昇順)などが書いてない(1999/3/29)
- カットオフテーブル書き込みのアドレスが明確でない。テーブルの数、どこからアドレスができるか、その他。(1999/3/29)
- カットオフテーブルの 0,1 次の次数が間違っている。(名村)  
CALC の REGOUT が何をするか良くわからない。  
パリティエラーなどはどうやって返されるか良くわからない  
FO も変換テーブルをつけてもいいような気がする。(福重)  
(1998/3/30)
- パイプと制御部のインターフェース、で 36 ビットが 32 と書いてあるところがある。(名村)  
(1998/4/13)
- ネイバーリストのアクセスシーケンスを追加する必要あり。  
(1998/8/22)
- FO inactive bit の書き込み方を変えたのが、6 章は変わっているが 7 章は変わっていない。
- JP port からの書き込みの時、アドレスとして与えるのが本当に物理的なメモリアドレスであって粒子番号ではないことを明記する必要あり。



## 第3章 アーキテクチャに関する検討

### 3.1 GRAPE-6 の概要

GRAPE-6 システムは、重力多体シミュレーション、特に独立時間刻みとエルミート積分法を用いた高精度計算のための専用計算機であり、シミュレーションのうち相互作用の計算の部分だけを担当する。それ以外の、粒子の軌道積分等は、汎用のホスト計算機で行なう。

GRAPE-6 システムでは、相互作用の計算をカスタム LSI を使って行なう。この章では、以下、カスタム LSI の機能からシステム全体の構成までを、特に技術的制約の観点から検討する。

### 3.2 基本機能

GRAPE-6 パイプラインチップ（以下、G6 chip）は、GRAPE-6 システムのうち粒子間重力の計算を担当する。具体的には、以下の式で与えられる粒子間重力とその一階時間導関数をハードワイヤー化したパイプラインを複数格納し、各サイクル毎に複数の粒子への重力を計算する。

$$\mathbf{a}_i = \sum_j G m_j \frac{\mathbf{r}_{ij}}{(r_{ij}^2 + \epsilon^2)^{3/2}} \quad (3.1)$$

$$\dot{\mathbf{a}}_i = \sum_j G m_j \left[ \frac{\mathbf{v}_{ij}}{(r_{ij}^2 + \epsilon^2)^{3/2}} - \frac{3(\mathbf{v}_{ij} \cdot \mathbf{r}_{ij})\mathbf{r}_{ij}}{(r_{ij}^2 + \epsilon^2)^{5/2}} \right], \quad (3.2)$$

ここで

$$\mathbf{r}_{ij} = \mathbf{x}_j - \mathbf{x}_i, \quad (3.3)$$

$$\mathbf{v}_{ij} = \mathbf{v}_j - \mathbf{v}_i. \quad (3.4)$$

であり、 $\mathbf{x}_i$ 、 $\mathbf{v}_i$ 、 $\mathbf{a}_i$ 、 $\dot{\mathbf{a}}_i$  はそれぞれ粒子  $i$  の位置、速度、加速度とその一階時間導関数、 $G$  は重力定数、 $m_j$  は粒子  $j$  の質量である。また、 $\epsilon$  はソフトニングパラメータと呼ばれる定数である。

なお、独立時間刻み法を用いるために、粒子  $j$  の位置、速度は以下のような補間多項式によって計算される。

$$\Delta t = t - t_j \quad (3.5)$$

$$\mathbf{x}_p = \frac{\Delta t^4}{24} \mathbf{a}_0^{(2)} + \frac{\Delta t^3}{6} \dot{\mathbf{a}}_0 + \frac{\Delta t^2}{2} \mathbf{a}_0 + \Delta t \mathbf{v}_0 + \mathbf{x}_0 \quad (3.6)$$

$$\mathbf{v}_p = \frac{\Delta t^3}{6} \mathbf{a}_0^{(2)} + \frac{\Delta t^2}{2} \dot{\mathbf{a}}_0 + \Delta t \mathbf{a}_0 + \mathbf{v}_0, \quad (3.7)$$

ここで、 $\mathbf{x}_p$  と  $\mathbf{v}_p$  は粒子  $j$  の時刻  $t$  での予測された位置、速度であり、 $\mathbf{x}_0$ ,  $\mathbf{v}_0$ ,  $\mathbf{a}_0$ ,  $\dot{\mathbf{a}}_0$   $\mathbf{a}_0^{(2)}$  は粒子  $j$  の時刻  $t$  での予測された位置、速度、加速度とその一階、および二階導関数である。

すなわち、チップ全体としては、式 (3.5) および (3.7) をホーナーの公式を用いたパイプライン ( $x, y, z$  の 3 成分のために 3 本) を 1 セット持ち、残りのシリコンに式 (3.1) の重力と式 (3.1) のその導関数を評価するパイプラインに入るだけ詰め込んだものとなる。以下、前者を予測子パイプライン、後者を相互作用パイプラインと呼ぶことにする。なお、 $\mathbf{x}_i$  などの、そこでの重力を計算される粒子のデータは各相互作用パイプラインが持つレジスタに保持される。これに対し、 $\mathbf{x}_j$  などの、重力を及ぼす粒子のデータは、順序アクセスされるメモリに格納される。メモリを G6 Chip に集積するか、あるいは外づけのメモリを持たせるかは後で論じる。

この機能記述書では、まず GRAPE-6 システムの想定する性能と、その性能から決まる G6 Chip の仕様への制限について簡単にまとめる。その後、G6 Chip の 2 つのパイプラインの実現の概要を述べる。

### 3.3 GRAPE-6 プロセッサチップの要求性能

ここでは、まずシステム全体の性能を規定し、そこから GRAPE-6 プロセッサチップの要求性能、特に入出力およびメモリアクセス性能がどのように規定されるかについて述べる。

今回の未来開拓学術研究事業「計算科学」次世代並列計算機開発プロジェクトで開発する GRAPE-6 システムの目標性能は、ピーク 200 Tflops とし、その性能の 50% 程度を粒子数  $5 \times 10^5$  体の球状星団シミュレーションで達成できるものとする。また、ソフトウェア側からみたパイプラインの並列度、すなわち同時に加速度を計算される粒子の数は 500 を越えないものとする。これは、粒子数が 100 万程度の場合、独立時間刻みで十分な性能を達成できる最大の数である。

#### 3.3.1 システム I/O 性能

まず、ホスト計算機とのデータ転送速度について述べる。ピーク性能 200 Tflops は、1 秒に  $3.3 \times 10^{12}$  個の粒子ペアについて相互作用を計算できることを意味する（相互作用一つはおよそ 60 演算である）従って、系の粒子数が  $5 \times 10^5$  とすれば、1 秒に  $6.67 \times 10^6$  個の粒子への重力が計算できる。

GRAPE-4 の場合、1 粒子ステップ当たりのホストと GRAPE 間の転送データ量は約 60 ワード、すなわち 240 バイトである。1 秒に  $6.67 \times 10^6$  個の粒子を扱うために必要な転送速度は、従って、1.6 GB/s となる。

さて、 $0.25\mu\text{m}$  テクノロジーを使って、2.5 M ゲート程度のチップを作った場合、入る相互作用パイプラインの本数は 6-8 本程度と想定される。このパイプラインを 133 MHz 程度のクロックで動作させることはそれほど困難ではないものと思われる。仮にパイプラインの本数を 6 本とすれば、チップ当たりの計算性能は約 50 Gflops となり、200 Tflops の性能を実現するのに必要な G6 Chip の個数は 4000 個となる。なお、仮に  $0.18\mu\text{m}$  の技術をつかって 10 M ゲート程度を集積でき、150 MHz で動作したとすれば、チップ単体の性能は 300 Gflops まで向上する。以下、とりあえず  $0.25\mu\text{m}$  テクノロジーを使った場合について、I/O 性能とシステム構成の関係を考える。

実装の容易さという観点からは、G6 Chip の I/O バンド幅をなるべく低く、例えば 100 MB/s 程度に押えることが望ましいであろう。この場合、ボード実装技術は GRAPE-4 の場合と同程度のもので済むことになる。

ここで問題となるのは、G6 Chip の I/O バンド幅と計算中のメモリからチップへのデータ転送のバンド幅の関係である。この 2 つは、同時に力を計算される粒子数を通して関係する。

一つのパイプラインが要求するデータ転送量は、クロック当たりおよそ 80 バイトである。仮に 133MHz のクロックを想定すれば、メモリからのバンド幅は 8 GB/s となり、パイプライン 6 本にそれぞれ別な粒子のデータを要求するとすれば、チップ一つのトータルのデータ要求は 64 GB/s となる。

この数字は、仮にメモリとパイプラインを同一チップに集積したとしても、非現実的なものであろう。クロックサイクル 133MHz で同期アクセスをしたとしても、データバスの幅が 480 バイト、すなわち 3840 ビットとなる。しかし、この場合、500 ビット程度は不可能ではないとすれば、8 GB/s 程度、すなわちパイプライン 1 本分程度の速度は得られることになる。

この場合、6 本のパイプラインを有効に使うには、それぞれのパイプラインが別の粒子への重力を計算すればよいことになる。つまり、一つのパイプラインチップが、6 個の異なる粒子への力を同時に計算する。

システム全体は 4000 チップからなるわけだが、同時に重力を計算する粒子の数（これを以下  $i$  並列度と呼ぶ）は 384 とすれば、これを 64 個のチップから構成された 64 のグループに分割できる。一つのグループの中では合計 384 本のパイプラインが 384 個の粒子への力を計算し、各グループからデータを読みだして、384 個の粒子それぞれについて 64 のグループで計算された合計を求め、これをホストに送り返す。つまり、一つの粒子への力は、64 本のパイプラインで並列に計算されていることになる。これを  $j$  並列度が 64 であるということにする。

この場合、G6 Chip 一個の必要バンド幅は、先の 1.6 GB/s を 64 で割ったもの、すなわち 25 MB/s でよいことになる。

ただし、この場合、オンチップのメモリは、64 個のコピーが存在することになる（グループの中のチップは同じデータをもつ）。これには、この構成では扱える粒子数が小さくなりすぎるという問題がある。つまり、64 個のコピーがあるということは、メモリ量の 1/64 の粒子数しか扱えないことになる。これは、もちろん、割り振りをかえることで対応できるが、チップの仕様を変えないままでは通信量が増える。仮にチップあたり 10,000 粒子程度まで格納できるとすれば（5Mbit 程度）、全体で 640k 粒子しか扱うことになる。実際には直接計算で少なくとも  $10^7$  体程度までは扱いたいので、そのためにはコピーの数を 4-8 程度まで減らす必要がある。この分だけ I/O の速度は増大するので、200-400MB/s 程度が要求される。

ただし、コピーの数を粒子数に応じて変化させることができれば、コピーの数が小さい時には粒子数が多く、相対的に必要な通信速度はちいさいことになろう。

上では  $i$  粒子の書き込み／読み出しについてだけ考えたが、 $j$  粒子、すなわちメモリへの書き込みについてはどうであろうか？ここで注意するべきことは、チップをグループ化した時に、各グループが同じデータを持たなければならないということである。すなわち、 $i$  粒子の書き込み／力の読みだしについては、たとえば 64 グループに分ければそれぞれ 25 MB/s の独立なデータ通信路をもてば良いのに対し、 $j$  粒子の書き込みについてはグループの数に無関係に 1.6 GB/s のデータがすべてのグループに放送されなければならない。これについては後でより詳しく検討する。

### 3.3.2 メモリ別チップの場合

さて、メモリを外づけにした場合はどうであろうか？G6 Chip とメモリとの転送速度は、SSRAM, SDRAM あるいは RAMBUS DRAM といった高速技術をつかっても、メモリチップ 1つあたり 0.5 GB/s 程度であろう。メモリサイズは 64 Mbit は使えると思われる。この場合、コピーの数を増やしても扱える粒子数に問題はないが、そのかわり同時に力を計算される粒子数が大きくなり過ぎるという問題が生じる。

プロセッサチップ 1つあたりメモリチップを 2 個とすれば、バンド幅は 1GB/s 程度が可能であろう。この場合、VMP (Virtual Multiple Pipeline) を使ってみかけ上プロセッサチップのなかに遅いパイプラインがたくさんあるように見せかけることで、バンド幅のミスマッチを解消する。Virtual/real 比が上の場合 8 となるので、みかけ上チップ一つにパイプラインが 48 本ということになる。

このため、トータルで 500 本という制限から、コピー多重率は 10、G6 Chip の I/O 性能は 200 MB/s ということになる。メモリ速度 1 GB/s に比べて I/O の 200 MB/s は小さいよう見えるが、メモリは chip-to-chip のローカルな通信であるのに対し、I/O は最終的にはケーブルを伝わってホストまで帰っていくものなので、ある程度の速度差があることが実装上必要であると考えられる。

### 3.3.3 I/O ポートの構成

GRAPE-4 チップでは、汎用の I/O ポート一つを通して（計算中のメモリデータ入力以外の）すべての入出力を行なった。しかし、この方法が適切なものかどうかは検討の余地がある。なぜなら

- 一般に、双方向入出力があるものは設計の手間が増える
- 双方向にすることによるメリットはピン数の節約だが、バンド幅を固定して考える場合、同時に双方向の転送が出来れば必要なピン数は変わらない。電気的には、双方向のもののほうが高速化が困難などの問題もある。
- 一般に、大規模 LSI を作る場合に問題になるのは、入出力の数であるが、安定動作の観点からすれば入力ピンの本数はあまり問題ではない。問題なのは同時に動作する出力ピンの数である。
- GRAPE-6 の場合、メモリ書き込みとレジスタ I/O は別のルートで、並列にデータがやりとりされうる。従って、別ポートにするほうが高性能化がはかれる。

以下、具体的に発生する I/O とそのタイミングシーケンスについて考えてみる。詳しい説明は省くが、オペレーションは表 3.1 の 4 種類になる

これらと、実際の力の計算との時間的な依存関係は図 3.1 のようになる。なお、一応、こでは、複雑なバッファリングをチップ側では行なわないものとする。すなわち、粒子座標や計算された力に対して、複数のバッファを用意して、転送と計算を並列に行なうことまでは考えない。また、独立時間刻みの場合だけを考える。

仮に、ホストでの計算時間が完全に無視できれば、転送と計算を並列に行なうことで最大 2 倍の性能が得られるが、これまでの経験では、ホストでの計算時間のほうが転送時間よりも大きな問題であり、こちらと計算を並列化するだけで十分な性能向上が見込める。さらに、実際の I/O においてボトルネックとなるのは GRAPE-6 の中ではなくホストと GRAPE-6

表 3.1: チップへの入出力操作

ID	Symbol	説明
1	JPSEND	メモリに $j$ 粒子（等）を書く
2	IPSEND	レジスタに $i$ 粒子（等）を書く
3	FREAD	レジスタから 加速度（等）を読む
4	NREAD	レジスタから 近接粒子リストを読む

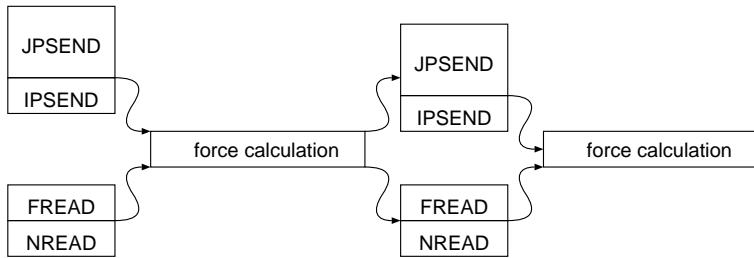


図 3.1: 計算ステップの時間依存関係

のインターフェースであろうから、そのすぐ後ろ、すなわちホスト・インターフェース・ボードに双方向のバッファをつけることで、実際には十分な計算と転送の並列化が行なえる。

さて、この図からわかることは、「すべてのデータ転送は、効率を最大化するなら並列に行なえる」ということである。したがって、共通の I/O ポートを使って順番に読み書きしても、それとも例えば 4 つの操作にすべて別のポートを使っても、実効転送性能を同じにするならチップの必要なピン数は同じである。

以下の議論では NREAD オペレーションは無視する。これは通常 FREAD に比べて小さい（毎ステップするとは限らない）からである。

現在のところ、IPSEND と NREAD でのデータ量は粒子当たり 300 ないし 360 ビット程度である。これに対し、JPSEND では、500 ビット程度であるが、これを 2 回送る必要がある。

2 回送る必要があるのは、自分との相互作用を避けるためである。これは GRAPE-4 でも同様にした。並列処理をしているために、条件つき実行によって自分との相互作用をさけるのはそれほど簡単ではない。したがって、メモリ内のデータとレジスタ内のデータを全ビット一致させることで、力、ポテンシャルが厳密に 0 になるようにしている。このために JPSEND が 2 回必要になるわけである。

なお、もう一つ別の考え方として、粒子にユニークな ID を与え、これを粒子メモリとレジスタの両方に入れておくというのも考えられる。理論的にはこちらが優れているようにも思われる。以下、この、ユニークな ID を持たせるという前提で話を進める。

GRAPE-6 の中の通信パターンとしては、IPSEND と F(N)READ は、方向は違うがトポロジー的には同じパターンをとる。これに対して JPSEND は全く違ったルートから送られることになる。従って、この 3 つすべてが別ポートを持ち、並列に転送されるとするには必ずしも非現実的というわけではないであろう。

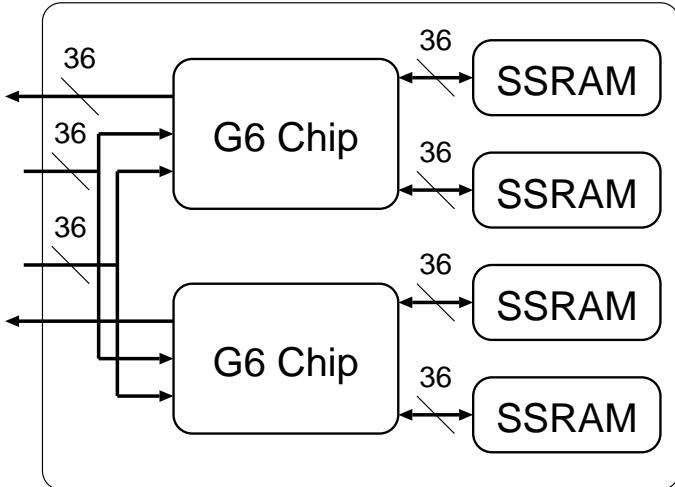


図 3.2: GRAPE-6 プロセッサモジュールの構成単位

粒子 ID により相互作用を避けるなら、JPSEND, IPSEND と FREAD の転送量に大差はない。従って、それぞれ同じ幅、具体的には 32 ビット幅を与えることにする。16 bit ではかなりクロックをあげないと (33MHz 以上) 性能的に不足である。これにたいし、32 ビット幅であればその半分で良いのでかなり楽といえる。I/O ピンは 96 (入力専用 64、出力専用 32) となるが、まあそれほど極端に多いということはないであろう。

メモリと G6 チップは密に結合している必要があり、ベースになにを使うかはともかくとして MCM にすることが望ましい。これによりボード不良率を大きく下げができると考えられる。

MCM の場合、考慮すべきことは、MCM に複数の G6 チップを載せることは現実的かどうかである。結論を述べると、多数の G6 チップが I/O をシェアすることはできない。これは、I/O をシェアすると、結局チップのパイプライン本数が増えたのと同じ結果になり、必要な I/O 速度が増加するからである。しかし、実は入力はシェアできる。したがって、2-4 チップ程度までならば大きくバス幅が増加するわけではない。図 3.2 に、2 チップの場合の MCM の構成を示す。

以上をまとめると、プロセッサの構成として、以下のような案が考えられる。

- ・メモリは外づけとし、RDRAM、SDRAM、またはSSRAM を G6 Chip あたり 2 個搭載する。
- ・相互作用パイプラインは 6 本集積し、100ないし 150MHz クロックで動作させる。V/R 比は 8 とする。これで必要なメモリ転送速度を 1 GB/s 程度まで落す。
- ・G6 Chip の入出力ポートは 32 ビット幅のものを 3 本とし、動作クロックは 25MHz とする。
- ・何らかの高密度実装技術を用いて、2-8 チップ（実装技術による）を 1 モジュールに集積する。メモリチップが G6 Chip に 2 個ずつつくるので、計 6-24 チップ程度を集積することになる。発熱は 20 ないし 100 W 程度となる。

- ・集積度に余裕があれば、メモリとプロセッサチップを1チップに集積する。
- ・モジュールを交換することで、ある程度までの性能向上が可能であるような考慮をする。

最後の2点は、システムの寿命をある程度長くするという面からみて重要である。製造コストの大半はプロセッサチップとモジュールにいくが、設計、調整の手間の相当部分はそれ以外のところが占める。したがって、仮にモジュール交換だけで性能向上が可能であれば、開発の人的なコストを大きく軽減でき、迅速な性能向上が図れる。

最後の点については、インターフェースを変えないとすれば、仮想（あるいは物理）パイプラインの数が変わっても対応できるようにネットワークが出来ていればいいということになる。もっとも簡単には、チップ（MCM モジュール）やホストがパイプラインの本数を指定し、ネットワークインターフェースは特定の本数を仮定しなければよいわけである。

## 3.4 プロセッサ間およびプロセッサ-ホスト間結合ネットワーク

### 3.4.1 並列化とネットワーク構成に対する基本方針

上に述べたように、ホストの I/O 速度を 1.6 GB/s としてコピー多重率が 10 程度にしたい。例えば多重率を 8 とすれば、全体が 8 個の独立なリダクション・ネットワークになっている必要がある。相互作用パイプラインへのデータ書き込み ( $i$  粒子書き込み) では、このリダクションネットワークに逆向きにデータを流し、ブロードキャストバスとして使う。

これに対し、 $J$  粒子書き込み（外づけメモリへの書き込み）は若干複雑となる。 $J$  粒子書き込みでは、各リダクション・ネットワークの中では同じデータは一ヶ所にのみ書かれる。しかし、各リダクション・ネットワークのなかでの書き込み速度が 1.6 GB/s である必要がある。これを、小規模な例で考えてみよう。チップ数が 16、チップ内パイプラインが 1、 $i$  並列度  $P_i = 4$ 、 $j$  並列度  $P_j = 4$  の場合を考える。

図 3.3 に、構成例を示す。縦に並んだ 4 個がリダクション／ブロードキャストツリーを構成する。これは左側に示されている。これにたいし、横に並んだ 4 個が  $j$  粒子書き込みバスにつながる。これは右側に示される。

4096 チップ、16 クラスタの場合でもトポロジーは同じである。

この時、並列化については、クラスタ内ではいわゆる  $j$  並列とする（ $j$  でいいんだっけ？）。すなわち、クラスタ内では各チップが異なる  $j$  粒子、同じ  $i$  粒子を持つ。 $j$  粒子ポートは共通で、全チップに放送され、そのなかから各チップが自分の担当分をとる。 $i$  粒子データは放送され、全チップが同じデータを使う。また計算結果はクラスタ内で組まれたリダクションネットワークで合計してから回収される。

複数クラスタの時の並列化は、粒子数によって変わる。粒子数がもっとも少ない、従って転送速度がもっとも必要になる時には、クラスタ間では  $i$  粒子を分担する。これによって  $i$  粒子書き込みと結果回収のバンド幅を上げる。ただし、 $i$  粒子の全体的な並列度がちょっと大きくなり過ぎるが、これには目をつぶる。

粒子数が大きくて転送速度がそれほど必要でなければ、 $j$  並列と  $i$  並列を混在させて使うのが望ましい。例えば、 $8 \times 2$  に分けて 8 では  $i$  粒子を分担し、2 では  $j$  粒子を分担するのである。

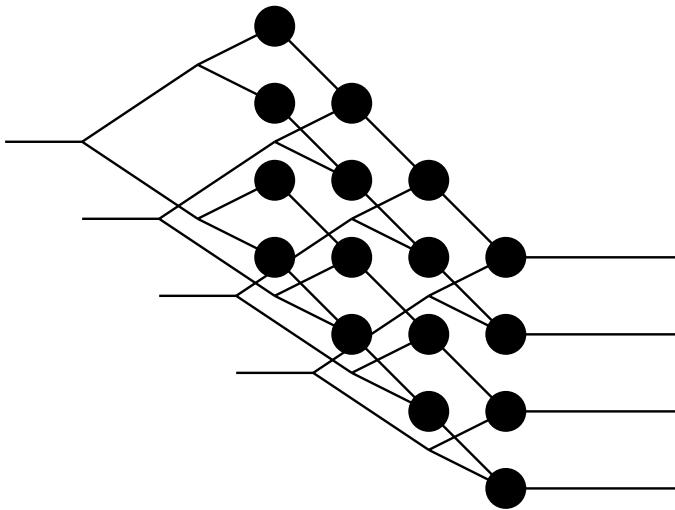


図 3.3: 16 プロセッサの場合のネットワーク構成例

### 3.4.2 $j$ 粒子書き込みネットワーク

上の方針で問題になるのは、書き込み側の実装である。16 クラスタを  $i$  並列で使ったとすると、全クラスタに同じ  $j$  粒子のデータが放送される必要がある。

例えばホストからそれぞれに線を出すことになると、合計 256 本のリンクが出てしまう。現実的な方法は、デイジーチェーン的につなぐことであろう。例えば Gigabit Ethernet のトランシーバを使って、送られてきたデータを受けとると同時に横に流すようなものを使えばいい。物理的にクラスタを横並びに配置できれば、線の長さが短く、また目立たないで済むことになる。なお、Ethernet は線が非常に少なくて済み、長い距離を伝送できるという利点はあるものの、インターフェースチップが比較的高価（特に Gbit の場合、GaAs が必要になるため）という問題もあるので、今回は LVDS (RS-644) を使った比較的高速のパラレル転送、具体的には National DS90CR215/DS90CR216 を使った転送を想定する。これは、ツイストペア線 4 組で、最大 400MHz 以上の速度で同期転送が行なえる。チップ自体に PLL とシリアル/パラレル変換を内蔵するので、外からは 21 ビットで最大 66MHz の同期転送に見える。これをさらに 1/2 に落して、25–33 MHz 程度で動作するようにする。

大規模なシステムを構築するにも、このような構成をとって  $i$  粒子のバスと  $j$  粒子のバスを独立に準備するのがもっとも簡単である。しかし、この構成の大きな問題点は、システムを分割して利用できない、さらに並列化の方法も変えられないということである。

200 Tflops のシステムを、単一のジョブが占有するのは現実的ではない。が、TSS のようなタスク切替えを実現するのは面倒である。このため、物理的にシステムを分割して使える必要がある。

分割出来ない理由は、 $j$  粒子書き込みバスにある。すべてのポートを使わないと書き込みが出来ないため、例えばホストの口を半分にしたのではデータの書き込みが出来なくなってしまうわけである。

分割の問題を考える前に、クラスタとホストの間の配線について考えたい。この構成では、各クラスタに書き込み専用のリンクが 16 本、双方向のリンクが 1 本となる。ここで問題なの

は、これは配線本数という観点から見て最適化されていないのではないかということである。すなわち、双方向のリンクを多少増やすことで、書き込み専用の線を減らすことは出来ないであろうか？

これは実際には難しくはない。例えば、書き込み線を4本、双方向線を4本とした場合、クラスタ4個で $j$ 粒子を分割して持つ。これで、上の場合に比べると接続本数は $1/2$ 以下に減るわけである。これは、論理的には4クラスタを束ねて1クラスタとみなすことに相当している。そうみなしたため、見かけ上双方向のリンクの速度が4倍になっているわけである。

しかし、それでも線の数は半分にしかならない。しかも、双方向の線が増えているので、見かけほど良いことがあるわけではない。従って、とりあえずこの方向では考えないことにしたい。

クラスタ内の配線としては、ボードの中には32個からなるツリーを1つ収納する。ただし書き込みリンクは2つだす。ボード8枚から16本書き込みリンクが出るので、これがそのままホストに向かう。双方向の方は、8枚のボードからの出力を合計するボードを介して一本だけがホストに向かうことになる。

なお、チップのポートはJPSEND, IPSEND, F(N)READに対応して3個あり、そのそれに対して物理的な線は別になっている。従って、ボード内、あるいはクラスタ制御ボード内でも、この3つのネットワークは物理的に独立なものであり、線の共有などは起きない。

### 3.4.3 バタフライネットワークを使う構成

分割可能なシステムを单一のネットワークで構成する一つの方法は、バタフライネットワークを使うことである。図3.4にその例を示す。4つのグループごとにバタフライネットワークが組まれ、その上に4個の2進木がかぶさる。この形のネットワークでは、システムを半分に分ける、あるいは4つに分ける、2:1:1の3個に分けるといった分割が容易に行なえる。

4096チップ構成となるGRAPE-6では、最上位に16個の2進木（それぞれ最下層が16ノード）があり、その下に $16 \times 16$ のバタフライネットワークがつながる。さらにその下に4階層の2進木を介して16個のG6 Chipがつながるという三層構造のネットワークにすることで、必要なバンド幅を実現し、同時にフレキシビリティのあるシステムを構築することができる。

ボード上のプロセッサ数とボード間結合の詳細は、モジュールの大きさ、実装法に依存するが、ここでは、ボード上に64チップ実装できたとしよう。また、一つのラックには、ボード8枚、したがって512プロセッサを搭載するものとする。ボード1枚からはリンクが4本、したがって、仮に25MHzでの同期転送を行なうとすれば幅は256ビットとなる。これは多いが、実現はそれほど困難でもないであろう。

8枚のボードの上に $4 \times 4$ のバタフライを8個重ねて、必要な $16 \times 16$ のバタフライが2つ構成される。この上に1段のツリーをおき、本数を半分にしたもののが、ラック一つからなることになる。ラック一つから出るリンクの数は16となる。

このリンクの数は、非常に多い。もっとも、線の本数は最初の案の単純な構成でも本質的には違わない。8ラックから構成する場合に、現在想定しているシステム分割では、各ラックがシステムのコピーを持つことになるので、それぞれが1.6GB/sの bandwidthを持つ必要が生じるからである。

ただし、システムの分割運用の程度を制限すれば、ラックからなるケーブル本数を $1/2$ 程度にすることはできる。

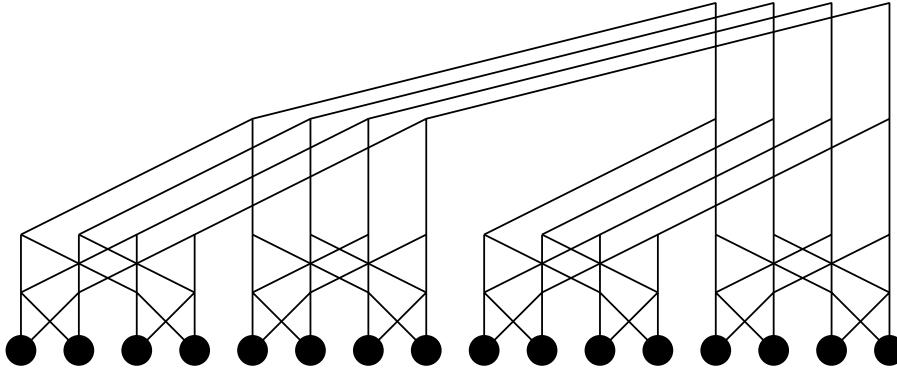


図 3.4: 部分的にバタフライを使ったネットワーク構成例

安価なフラットケーブルを使って、25 MHz 程度でのデータ転送を行なうのは、長さが短ければそれほど難しくない。が、実用的な範囲を超えてケーブルの本数が多くなっているというのも否定し難い。受信側に、送信側からのクロックで動作する FIFO をつけ、クロックスキーの問題を押さえた上で、高速のクロックで送るなどの技術を使うことが望ましいであろう。

#### 3.4.4 代案 1

上のバタフライネットワークを使う構成は、美しいものではあるが設計、実装ともに比較的複雑である。特に実装においては、バタフライを実装するボードで配線長が長く、またパターン交差がおきやすい。このため、密度をあげることはそれほど簡単ではない。

例えば、1 クラスタだけを独立に使う場合のことを考えると、必要な機能は単にクラスタ内での  $j$  粒子の放送である。これだけならば、単にそのための放送網を用意しておけば良く、それほど手間ではない。

任意に分割可能にしようとすると話が面倒になる。しかし、例えば分割を 2 のべきに制限し、さらに分割した時にホストのどのポートから信号が来るかを固定して考えてみよう。この場合は、1, 2, 4, 8, 16 のそれぞれの数の場合に、1 通りの方法で放送（1 の時は一つに、2 の時にはクラスタの中を 2 グループに、、、、以下同様）できればいい。この場合、図 3.5 に示すもので済む。これは、トポロジーを書き換えれば、単に 16 本葉がある二進木のすべての「右側」ノード（葉を含む）から上に線がでているということに対応する。どちらからのデータをとるべきかは分割によって変わるが、各レベル毎に同じである。これはスイッチの動作モードが分割の種類に対応した 5 通りしかないことによる。

この場合、スイッチボードは 15 個のセレクタとその間の配線だけで構成されることになる。

クラスターとホストの間の配線については、この場合、分割を考えない場合とまったく同等でよい。

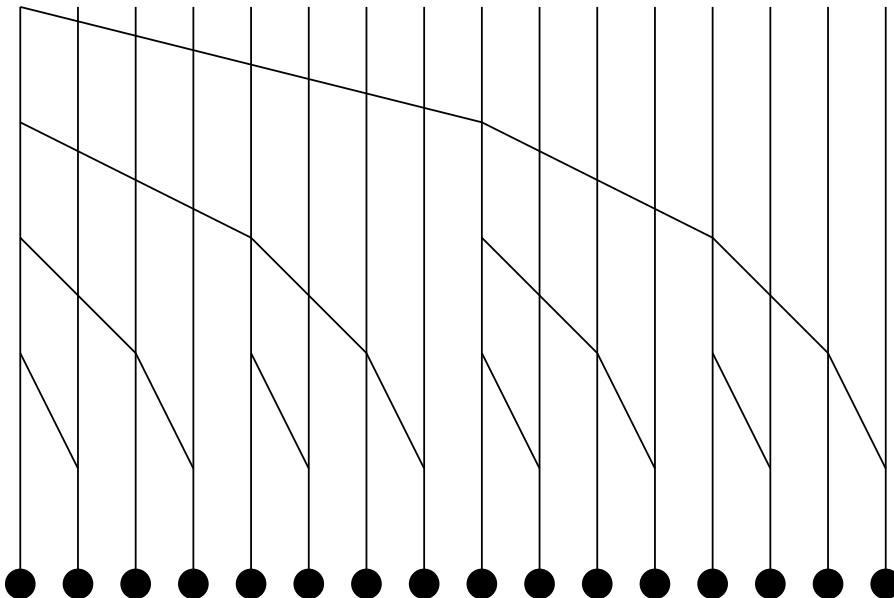


図 3.5: 最小限の放送機能をサポートするネットワーク

### 3.4.5 代案 2

代案 1 はミニマリスト的 ideals に近いものと思われるが、さらに簡単にすることの可能性を考えてみる。これは、最初に述べた書き込み線を減らす方策により実現可能である。この場合の問題は、しかし、仮に 16 クラスタを 8 グループに分けた場合、グループ内で結果を合計する仕掛けがさらに必要になることである。これは、ホストとクラスタを直結出来ず、もう一つレベルが必要になるということを意味する。これはあまり嬉しいので、この代案はとらないものとする。

### 3.4.6 NREAD の実装

データ転送性能に関する評価においては、NREAD はそれほど重要ではないが、実装上は FREAD とは違った動作をすることに注意する必要がある。というのは、FREAD ではネットワークがリダクションツリーとして振舞い、全体としてパイプライン的にデータが流れるのに対し、NREAD ではデータの縮約は起きないからである。

問題になるのは、バッファリングが必要になるということである。すなわち、FREAD の場合は入ってきたワードを足し合わせてそのまま吐き出すので、ネットワークの各点でデータ転送速度が同じでいいのに対し、NREAD では下にいくほど読み出しが遅くなっているかないと、途中でデータが溢れてしまう。

溢れるのを防ぐためには、以下のような方法が考えられる

- 下位のノードとのハンドシェイクを行ない、データを止める。
- 下位のノードからのデータを溜めるだけの大きさのバッファを各ノードが持つ。

- 下位のノードが、上で溢れないように初めからゆっくり出す。

ハンドシェイクする方法がもっともスマートであるのはいうまでもないが、面倒臭いのも確かである。したがって、まず、バッファリングする方法について検討しよう。

ハンドシェイクを一切行なわない、すなわち、各プロセッサチップがデータを垂れ流しにする場合というものを想定してみる。この場合、まず、ホストインターフェースにおいては、全チップから出てくるデータをすべて保持できる必要がある。これは、ホストにデータを送るのは PCI バスのタイミング依存であり、一定の転送速度で送れる保証はないからである。

それより下のノードでは、一定の転送速度が保証されているといつていい。これは、「上の都合を考えないで勝手に送っていい」ことにするからである。しかし、これはつまり下から送られてくるデータレートのほうが必ず速いということでもあるので実際には上に送れるということにあまり意味があるわけではなく、下から上がってきたものをほとんど完全にバッファリングできる必要がある。少なくとも半分なり  $3/4$  の容量がいることになる。

それでは、返ってくるデータの量はそもそもどの程度であろうか？同時に力を計算される粒子が（クラスター内、あるいは单一チップ内では）せいぜい 64 個なので、このそれぞれに平均 30 個のネイバーがあったとして 2k words、 $i$  粒子テーブル（後述）の分でデータ量が 2 倍になったとして 4k words である。したがって、オーバーフローなどを完全に防止しようとすれば、例えば 8kwords = 256kbits 程度の容量を持てば良い。これはもちろん、下にいくほど小さくなる。

ホストインターフェースのところでは、8k words というのはそれほど大きくはない。いずれにしても力をいれるだけでも 1-2kwords は必要だからである。しかし、各ノードにその程度（下ほど小さいとはいえ）を持たせるのはそれなりに面倒である。現在の FPGA チップでは、チップ内のメモリ容量はまだ非常に小さいものが多いからである。もちろん、ノード間すべてに FIFO チップを入れればいいが、8k words というのは深い部類に属するし、チップ数、ピン数がほぼ倍近くに増えるので出来ればしないで済ませたい。

オンチップで済ませる場合のことを考える。この場合、当面使えるメモリ量は例えせいでせいぜい 16kbits 程度とすれば、バッファのワード数は 512 語程度、仮に 4 入力が実現出来るとするならポート当たり 128 語ということになる。

プロセッサボードのレベルでも、この語数で十分であるとはちょっといい難い。以下、残る 2 つの可能性について検討する。

まず、ハンドシェイクである。これはもちろんすればいいというだけの話だが、ボード間接続については現在想定していない上から下への制御線が余計に必要になるという問題はある。もちろん、ボード間接続は数が少ないし、どうせ FIFO が必要になるところなので、深い FIFO をつけてここはハンドシェイクしないという解も可能であろう。これに対し、ボード内では線が 1 本増えるだけなのでそれほど大きな問題ではない。

ハンドシェイクするとすれば、ネイバーリスト転送モードに入ったところで、各ノードは自分の受け持ちのノードに順番にリクエストを出し、一つ読み終ったら次という具合にデータを見ていく。ここでちょっと厄介なのは、適当に作ると読みだし先が切り替わる時にデータが途切れるので、実質的に転送レートが落ちるということである。もちろん、あらかじめ先読みしておけば問題にならないが、この場合にはバッファのための FIFO がワード数は少ないととはいえやはり必要になる。バッファリングもしてハンドシェイクもするというのはどうも面倒である。少なくとも、なるべく設計するものを減らすという趣旨からは好ましくないであろう。

さて、それでは、「データレートを減らす」という解はどうであろうか？これは、要するに下にいくにしたがって実際にデータを出すレートを落してしまおうというものである。これ

の実装はそれほど困難ではない。というのは、一番下だけでレートを落してやれば上は自動的に落ちるからである。例えば 256 チップから読み出すなら、256 クロックに一度しかデータを出さないようにすればいいわけである。

この方法の実現は、実際には厄介である。もっとも大きな問題といえるのは、実効的なデータ転送効率の低下であろう。すなわち、チップによって出来たネイバーリストの大きさのバラツキが大きい時に、もっとも長いチップでトータルの転送時間が決まり、ホストへの口が遊んでしまうのである。

この遊びについては、もちろんいくつかの考え方があり得る。一つの考え方とは、遊んでいても問題ではないのではないかというものであろう。すなわち、統計的に平均をとれば無駄はそれほど大きくならないであろうから、まあいいんじゃないのという考え方である。実際、おそらく無駄は 2 倍程度であろう。

もう一つの問題は、実はバッファリングが不要になるわけではないということである。つまり、チップ毎に出すタイミングをずらさなければ、語数が減るだけでバッファがいることには変わりがない。

いずれにしてもバッファリングがいるのなら、ハンドシェイクもバッファリングもするという前提で、チップ数をなるべく減らす方向で考えてみる。具体的には、一番下の、プロセッサチップとつながるところにはバッファは必要ない。ここはつけないことにして、その上へのリンクについて下のノード内にバッファをつける。これがいっぱいになってきたところで読むのを止めるようとする。ツリーの上の階層についても話は同じである。この場合、バッファは数ワード（理論的にはツリーの深さ × レイテンシ）程度あれば十分ということになる。なお、ボード間接続については、受け側、すなわちコントロールボード側に十分大きなバッファを用意することで、ボードを超えたハンドシェイクは不要になるようになる。

### 3.4.7 ネットワーク構成に関するまとめ

以上で見たように、バタフライは理想的であるが、必要以上の柔軟性を実現するためにハードが非常に複雑化する。このため、代案 1 に述べた方法によってクラスターを構成し、 $j$  粒子プロードキャストは一次元 open ended ネットワーク、双方向の  $i$  粒子に関するコミュニケーションはスター型のトポロジーとする。図 3.6 に完成イメージを示す。

これで、クラスタ間並列化は  $i$  並列だけ、 $j$  並列だけ、その組合せを粒子数によって選べ、またシステムを分割してつかうことも可能であるという（本当かなあ）なかなか結構な構成ということになる。

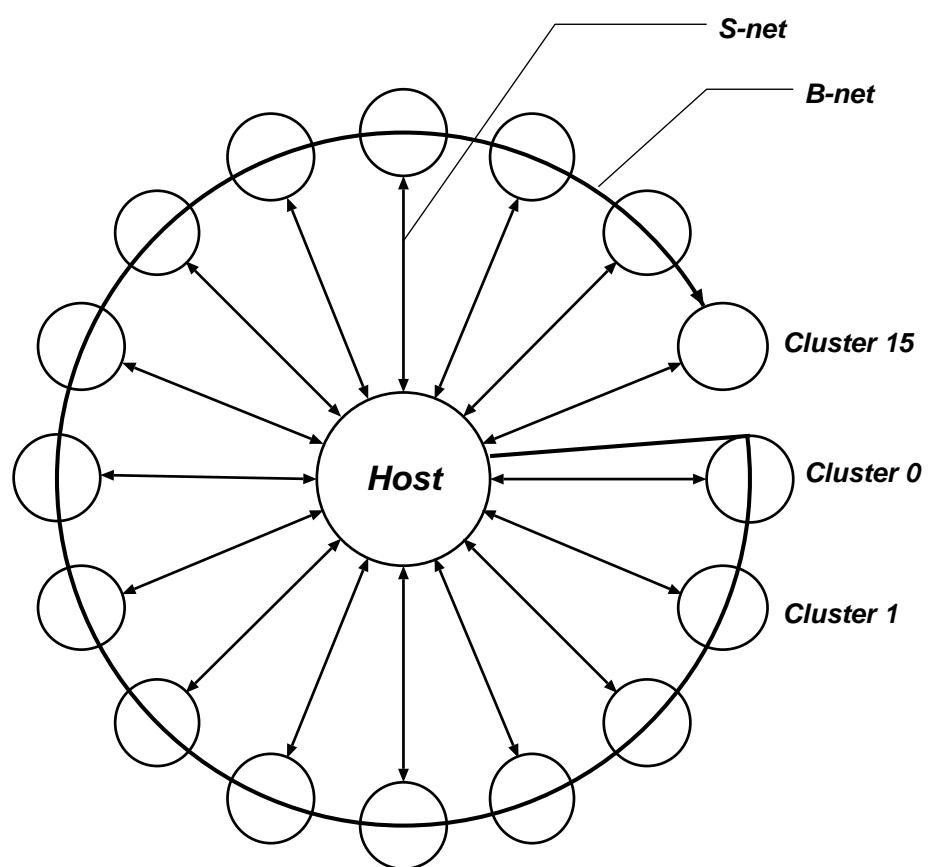


図 3.6: 現時点での全体構成案

## 第4章 エラーチェック／コレクションについて

GRAPE-4 では、マシンのどこにおいてもどんなエラーチェックも行なっていない。これで一応大きな問題なく動作しているが、GRAPE-6 では高速シリアル通信を使うことを前提とするため、エラーに対するなんらかの配慮がどうしても必要になる。以下、どうすればいいかを少し考えてみる。

なお、非常に抜本的な対策として、Gigabit Ether や FibreChannel のような面倒なものを使わないで、LVDS の数本の束で済ますという考え方もある。この場合には、例えば NS DS90CR211/212/281/282 のようなものを使うのである。この場合、クロック信号が別に出るのでビットずれを考慮する必要はなくなり、通常のパリティチェックだけで済むことになる。また、パーツも安価でありコネクタ等もより簡単なもので済むことになる。以下、こちらを使うことを想定して話を進める。

### 4.1 エラー検出

まず、LVDS による転送と、ボード上のパラレル転送でどのようなエラーが発生する可能性があるかについて考えてみる。これは、大雑把には 3 つに分けられる

- シリアル転送におけるクロックずれ
- シリアル転送におけるビット化け
- パラレル転送におけるビット化け

まず、クロックずれについて考えてみる。LVDS による Channel Link の、FDDI, FC などのシリアル転送に比べた利点は、クロックずれが永久的なエラーではないということである。すなわち、FC 等の場合には、一度クロックずれが起きると次に同期キャラクタが送られるまでの全てのデータが化ける。これに対し、Channel Link ではクロックジッタなどですれ起こっても、それは（伝送線路の特性による永久的なものでなければ）ほっておいても回復すると考えていい。言い替えれば、これはクロックずれを他のエラーと同等に考えていいということである。もちろん、実はまだ違いがある。それは、エラーが 7 ビット（シリアルパラレル変換の単位）全体について起きる可能性が高いということ、言い替えれば、ワード毎に ECC を掛けるのは困難であるということである。言い替えれば、ここで起きるエラーを訂正するためには、例えりード・ソロモンのような面倒な符号化が必要になる。エラーレートは、しかし、それほど悪くはないと想像される（これは別途実験が必要である）ので、パリティをつけておけばつけないよりはましであろう。

複数ビットのエラーの場合パリティでは検出できない可能性もある。これを検出または修正するためには、結局リード・ソロモンか何かを使うしかないでの、とりあえずこの問題は考えない。予備的な転送テストを行なってみて、複数ビットのエラー確率が無視できないほど大きければ検討することにする。

以下、单一ビットエラーに対する対応について考える。

ビット化けを検出する基本的な方法はパリティチェックである。もちろん ECC を行なうことも考えられるが、ホストとの通信のところではおそらくパリティしか使えないで、通信に関してはパリティチェックだけにしたい。メモリとプロセッサの間の転送については、転送量がはるかに大きくなるので ECC にすることが必要である。

従って、ホストとのデータ転送については全て通常のバイトパリティをつける。これは PCI インターフェースまでそのまま伝わることになる。

G6 チップが受信するデータについては、途中で加工されることはない。従って、チップ内にエラー状態を保存しておく。一旦エラーが起きたら、これはエラーをクリアするコマンドが送られない限り残るということにしておく。

FREAD ではちょっと話が面倒になる。(加算を行なうため) しかし、單に入ってきたデータについてパリティをチェックし、出ていくデータのためにはパリティを生成するだけである。もちろん、注意するべきことは、入ってきたデータにパリティエラーがあれば、それを伝える必要があるということである。

なお、上に述べたように、メモリについては ECC を行なう。すなわち、64 ビットワード単位で SECDED 符号化することにする。このため、書き込み、読み出し共に少し回路が複雑になる。

## 4.2 エラーからの回復

データにエラーがあった場合にすることは単にデータを送り直して再計算することである。ハードウェア層は回復のための特別な機能を持たず、回復はすべてソフトウェアの責任で行なうものとする。これは、ハードを単純化するためである。

もっとも簡単には、アプリケーションにハードエラーが起ったことが伝わる。そこでアプリケーションは何らかの初期化処理をすることになる。GRAPE-4 では、プログラムによってはそういう処理をしているものもある。

これは、しかし、アプリケーションプログラマ、つまり実際に使う人が GRAPE のハードエラーのことまで気にしないといけなくなってしまってあまり好ましくない。インターフェースライブラリレベルに隠蔽することが本来は望ましいであろう。すなわち、GRAPE-6 が持っているべきデータの完全なコピーを、アプリケーションがではなく（これは持っているとは限らない）インターフェースライブラリが持ち、エラーが起きたら単に自分が持っているデータを送り直すのである。

この実現の詳細についてはまた別に述べることにする。

さて、今まで GRAPE-6 が動作はしていて、データが化けた場合というのを考えてきた。しかし、実際には、エラーに関しては単なるデータ化けではなく、コマンドにエラーがあつたためにデッドロックになった場合の処理を考える必要があろう。これがホスト側からみてどう見えるかを考えてみると、要するにデータ、コマンドを送って、計算を始めさせているつもりなのにデータが返ってこないとか、必要な語数のデータが戻ってこないとか、そういうことになる。

原理的には、データ、コマンドを送ってからホストがどれだけまたないといけないかはわかっているわけなので、ライブラリのなかで適当な方法でどれぐらい待ったかを推定し、あまりに長く待っているようであればエラーと判断して全体をリセットし、新しくハードを初期化すればよいであろう。

### 4.3 耐障害性

GRAPE-4 では、特に MCM の信頼性に問題があることが当初から予測されていたので、故障したものを使わないようにする仕掛けをつけている。しかし、これは、ボードを組み上げる段階でどこが壊れているかをあらかじめ知っている必要があり、さらにそれによってハードウェア（具体的にはデコーダーチップの中身）を書き換える必要があるという野蛮なものであった。

GRAPE-6 ではどういう観点からこれを扱うべきであるかということを以下に考える。

まず、考るべきことはどのレベルで耐障害性を持たせるべきかということであろう。つまり、並列性がチップ内、ボード内、クラスタ内、そしてクラスタ間のどのレベルにもあるので、原理的にはそれぞれのレベルで冗長性を持たせる、あるいは故障した部分を切り離すことができるはずである。

例えばクラスタ間での冗長性は、ホスト計算機のソフトウェアだけで対処できる問題である。それでは、逆の端、すなわちチップ内はどうであろうか。まず問題なのは、「なにが」「いつ」故障する場面を想定するかということである。すなわち、GRAPE-4 の場合、想定したのは基本的には単に MCM の初期不良、それもほぼあるタイプのボンディングエラーだけである。これは、「全ての MCM の全てのチップが動作する」という確率はかなり低いのではないかと想定したためであった。実際、パッケージメーカーから納入されたものは、かなりの率で 1 または 2 個動作しないものが含まれていた。

しかし、今回は MCM を使う場合でも、おそらく GRAPE-4 の MCM ほど不良率は高くならないであろう。これには 2 つの理由がある。一つは、MCM を使う目的がことなり、そのため集積度（チップ数）も異なることであり、もう一つは使う技術の違いである。

GRAPE-4 で MCM を使った理由は、集積度を上げるためにあった。通常ならば 5 センチ角のセラミックパッケージに 1 チップとなるところが、およそ 8 センチ × 12 センチのモジュールに 8 個収めることができた。ボード上の配線に必要な面積なども考えれば、およそ 1/3 に面積を下げることができていると思われる。なお、付加的な利点として、プロセッサボード上の配線が大幅に減少し、生産コスト、不良率を下げることができたこと、また、MCM パッケージ自体が実は通常のパッケージよりも安価（チップあたりでは）ということもあった。<sup>1</sup>

しかし、GRAPE-6 では集積度を上げる要求は GRAPE-4 に比べてずっと小さい。これは、特に BGA に代表される小サイズで安価なパッケージが利用可能になったことと、チップ自体の発熱が増加しているため冷却を考慮すると GRAPE-4 よりも実装密度を下げる必要があることによる。GRAPE-6 においても MCM を使いたい理由は、最も実装での不良率が高いであろうと思われるプロセッサチップおよびメモリチップを基板に直付けしたくないというものである。これは、言い替えれば、別モジュールになっていれば必ずしも MCM、例えば MCM-D や MCM-C などの高価でリスクの大きな方法を使う必要はないということである。BGA のパッケージを小さな多層基板に実装してモジュールを作り、これを親基板に差し込むのでも目標は達成される。

もちろん、この場合多数のコネクタ（ソケット）を使うことになるので、接点不良がおきる可能性が付け加わり、不良率自体は必ずしも下がるわけではないかも知れない。しかし、プロセッサボード自体の接点、配線数は激減する。従って、主な不良モードはモジュール上の配線、実装に現れることになろう。このテストは比較的容易と考えられる。

モジュールの不良率はかならずしも低いとは限らない。また、最初はつながっていてもあとで剥がれるというような故障もあり得なくはない。これはソケットについてもいえる。従つ

<sup>1</sup> 値段は、パッケージメーカーの見積りが甘かったのではないかという気もしなくはないが

て、ボード上のモジュールについて故障があっても動くようにできればそれに越したことはない。

パイプラインチップに予備を持たせるのはそれほど簡単ではない。というのは、ツリー構成になっているため、どこかが壊れた時にその代わりになるためには、全てのところにつながっていないといけないことになるからである。しかし、例えばモジュール不良については、別の考え方も可能である。

今、単一クラスターで使っている状況を考える。この場合には、実はモジュール（パイプラインチップ）不良にはソフトウェア的に対処できる。というのは、全てのチップが同じ  $i$  粒子への力を計算しているので、不良チップに  $j$  粒子がいいかのようにして、結果も不良チップのは無視するようにすれば、それだけで特に問題なく使えるからである。つまり、実質的な耐故障性が、故障検出さえできれば実現できることになる。もちろん、読み出し用の FPGA が、不良チップから出てくるデータを無視できるようになっている必要はある。このためには、FPGA の動作をホストからのコマンドで変更できる必要がある。したがって、例えば IP のツリーを使ってコマンドを送るなら、それを受けとるためのコマンドバスが必要ということになる。なお、配線を増やさないでますために、IP ツリーを通って G6 チップに到達したコマンドがそのまま帰ってくるような仕掛けのほうが望ましいかもしれない。また、不良チップでもインターフェースは生きているとすれば、答として 0 を出すようにするという方法もある。こちらのほうが簡単であろう。

これに対し、クラスタが複数になると話が厄介である。というのは、別のクラスタで違う位置のチップが壊れている場合、あまりうまくいかないからである。

もちろん、これは、エラーレイトがどの程度である場合を想定するかによってかなり話が違ってくる。例えば平均してクラスタあたり一箇所程度と思って良ければ、1 つのクラスタで壊れている箇所を全部のクラスタで潰してしまってもたいした問題ではない。最大でも 16 個しか潰す箇所はないからである。これに対し、例えばボード 1 枚に一つ位という状況を考えると、非常に厄介な話になる。というのは、最悪の場合全て潰れてしまうからである。

仮にボード 1 枚に 1 つ程度のエラーに対応したいとすれば、論理的にエラー箇所をボード間で合わせてやればよい。これは GRAPE-4 では行なっている。GRAPE-4 においては、しかし、あくまでも MCM の初期不良に対応することだけが目的であったので、不良がある MCM を挿すソケットの位置は同じとし、その MCM へ出るデコード信号を変えることでボード間で不良の論理的な位置を揃えた。

GRAPE-6 では、もしも上のようなことをしたいとするとちょっと複雑になる。というのは、 $j$  粒子書き込みの時の各チップの動作を変更する必要があるからである。

$j$  粒子書き込みの時は、プロードキャストネットワークを通じてボード上のリンクを共有するチップ全てに同じデータが流れる。各チップは、そのなかで実際に自分が担当するところを、インデックスデータの上位ビットを見て判断することになる。判断するためには自分が誰かを知らないといけない。これは、もっとも簡単にはチップ外から入力する。例えば 10 ビット程度の情報を、固定して与えておけばいい。現在の仕様では最大でも 8 ビットでいいが、余裕をみて 10 ビットとしておく。

さて、上のように外から ID を与えるとすれば、チップ動作を変更するためには単にこの外からくる ID を書き換えればよい。そのためには、ボード上にレジスタをおいて適当なコマンドで書き換えればよいであろう。

もう一つの考え方は、チップ自体に物理 ID と論理 ID の変換を行なわせることである。つまり、通常のレジスタ書き込みのほかに特別なコマンドを用意し、それによってチップの物理 ID に応じて論理 ID の書き換えを行なえばよい。これは、特に付加チップなしに行なえるし、チップ内での実現も容易である。

最後に、ボードレベルの不良、例えば書き込みポートが死んだとかいう場合にはそのボードをチップやクラスタの場合と同様に切り離すことができよう。各クラスタでそれぞれどこかが死んでいるとかいう場合は、仮に起こったとすればコントロールボードとプロセッサボード間の配線の変更によって対応する。これに物理的な配線の変更なしに対応するためには、コントロールボード上の  $j$  粒子書き込み用ネットワークをよりフレキシブルな（従って線やスイッチの数の多い）ものに変更する必要があるからである。

## 4.4 故障診断

さて、ソフトウェア的なエラー回復ができるとしても、実際にその機能を使うためには故障診断も自動的に出来る必要がある。ここでは、その方法について検討する。システム全体がツリー状の構造をとっているので、基本的には各末端ノードが動いているように見えれば動いているわけである。

基本的には、まず適当な計算をしてみて、答が合えば全体が動いていると思っていい。合わない場合については、本来は「どこが具合が悪いか」が特定されていてほしい。特に、マシンが完成してからの、再現性のない（確率の低い）動作不良については、実用計算に使いながらエラー統計をとって故障箇所を特定するといったことができるにこしたことはない。従って、単にエラーを検出できるだけでなく、どのチップで起きたか特定できることが望ましい。

書き込み時の転送エラーについては、チップ自体がステータスを知っているのでこれをホストに返す仕掛けがあればいい。では、計算間違い、あるいは結果回収時のエラーはどうであろうか？

まず注意して欲しいのは、計算間違いはチェックされないとということである。つまり、データ転送等についてはパリティなどで保護されているが、実際の演算については何の保護もなされていない。これは問題といえば問題であるが、あまり有効な対策はない。2回同じ計算をしてみるくらいであろう。あるいは、求まった答が物理的にあり得るかどうか（予測された力から許容範囲内か）というようなチェックも有効と考えられる。いずれにしても、計算間違いをしたチップを計算中に特定するのはそれほど簡単ではない。

何らかの理由によって、（すべての粒子ではなく）特定の粒子だけが具合が悪いと判断できるのなら、例えば全パイプラインでその粒子への力を計算すれば、答が違っているところが怪しいということは判断出来ることになる。これでは、しかし、チップのなかのどのパイプラインが悪いかということはわからても、どのチップがおかしいかということはわからない。この時は、例えば6本のパイプラインに対して計算結果のコンペアをチップ内で行ない、おかしければそれを憶えておくというようなことをすればチェックは可能である。しかし、実際にそんなことまでするようなしかけを付けたいかどうかという問題はある。

エラーが再現性を持つものであれば、同じデータを送っておいてチップ1つずつチェックすれば特定できる。が、そうでない場合には結局のところ原理的な困難があるといわざるを得ない。本当にランタイムで再現性のないエラーを見つけながら計算しようとするなら、本当に冗長な計算を行なうというのは一つの方法である。つまり、2つまたは3つのクラスタで完全に同一の計算を行なう。この場合、どこで違ったかを事後的に決めるのは可能とは限らない。

結果回収時については、エラーを検出することはできる。その場所の特定については、例えばツリーノードがエラーステータスを覚えていて、それを返すような仕掛けがあればいいことになる。



## 第5章 アーキテクチャに関するまとめ

この章では、全体の構成に関するまとめを行なう。チップの方から順番に上にあげていくことにする。

### 5.1 チップの構成

ここでは、演算精度などについての細かいことはおいて、特に I/O の仕様について述べる。

表 5.1 に制御線などを除いたチップの入出力ポートをまとめる。データは基本的にパケットの形で送られるものとする。その中にコマンドなどが埋め込まれる。チップは SSRAM チップ (データ幅 36 ビット)2 個と直結可能なものとし、この 3 チップで GRAPE-6 のすべての機能を実現する。その上につくのは並列化のためのネットワークに過ぎない。

このため、最小構成としては、この 3 チップの先にホストインターフェースがあればそれで使えるということになる。初期のテスト用システムとしては、現行の GRAPE-4 PCI インターフェースボードの先にこのモジュールをつけたものを使う。

なお、全ての I/O ポートにパリティをつける。これはバイトパリティとし、ECC 等は考えない。エラー検出は、JP, IP およびメモリポートについて、フラグレジスタを持たせておく。これを適当なタイミングで読み出す。

### 5.2 ボードの構成

以下、ボード 1 枚に 32 チップとする。ボードは IP、FO ポートを各 1、JP ポートを 2 持つ。以下、各ポートのボード内接続について簡単に述べる

表 5.1: チップの入出力ポート

ポート	I/O	幅	速度	説明
JP	I	36	25MHz	<i>j</i> 粒子入力
IP	I	36	25MHz	<i>i</i> 粒子入力
FO	O	36	25MHz	計算結果出力
MEMD	I/O	36 × 2	125MHz(?)	メモリデータ
MEMA	O	21(×2)	125MHz(?)	メモリアドレス

### 5.2.1 IP ポート

ボードへの入力が全チップに放送されるだけである。適当な数のバッファを使ってファンアウトが大きくなり過ぎないようにする。負荷は8程度とし、抵抗終端（あるいは、アクティブターミネーションが出来ればそちら）をすることでややこしい問題が起きることを防ぐ。

### 5.2.2 JP ポート

チップを16個ずつの2グループにわけ、それぞれについてボードへの入力がグループの全チップに放送される。電気的な配慮はIPポートと同様である。

### 5.2.3 FO ポート

ボード内の全チップが同一の粒子からのデータを計算するので、それぞれバラバラに計算した力を合計する必要がある。さらに、ネイバーリストなどの場合には単に素通りでデータを抜けさせる必要もある。すべてのチップから同時に結果を読みだしながら送る必要があるので、適当なツリー構造のネットワークでリダクションを行なう必要がある。なお、この形をとるので、返ってくるデータは固定小数点であることが、必須ではないものの非常に望ましいといえる。これは、合計をとるためのハードウェアが非常に簡単なものですからである。

具体的には、例えば4入力、1出力程度のものを280ピン程度のFPGAで実装するのはそれほど困難ではなく、またチップ数も少なくて済むであろう(11チップ)。これを2入力とすれば30チップ必要となる。なお、NREADのためのバッファはFPGAのオンボードメモリで済ますように考える。

### 5.2.4 ボード間接続

ボード間の接続にはLVDSを使ったChannel Linkを使う。具体的には、DS90CR215/216またはSN75LVDS81/82/84/85当たりを使うことにする。これにさらにバス幅変換を行なうFIFOをつける。詳しくは次節に述べる。

## 5.3 クラスタの構成

以下、クラスタ1つに8ボードとする。各ポートのクラスタ内接続について述べる。プロセッサボードの他に、制御用のボードが必要になる。これをコントロールボードと呼ぶことにする。

### 5.3.1 IP ポート

クラスタへの入力が全ボードに放送されるだけである。適当な数のバッファを使ってファンアウトが大きくなり過ぎないようにする。スキューレの増大を防ぐため、一旦パラレルに戻してから広げることにする。

### 5.3.2 JP ポート

ここは、概念的には少しややこしいところになる。まず、図3.6に示したようにリング状につなぐためには、入力をすべてそのまま出力する必要がある。その上で、各プロセッサボードには3.4.4節の図3.5で示したようなツリーと只の線が重なったものをつける必要がある。これは単なるマルチプレクサの固まりであり、ピン数の多いFPGAを使って容易に実現出来よう。

なお、マルチプレクサの動作モードの切替えは、クラスタ毎に別に行なえる必要がある。したがって、IPポートからコマンドを送ってモード切替えを行なう。

### 5.3.3 FO ポート

ボード内と同様、合計を計算するチップが必要になる。これはボード内の接続と同じものになる。但し、入力ポート（ボード側）に適当なサイズの FIFO によるバッファをつけることで、NREAD の実現上の面倒を減らす。これはポートあたり 2 kwords 程度つけることにしたい。

## 5.4 全体の構成

以下、16 クラスタで全体システムを構築するものとする。IP、FO の各ポートについては、単にホストに直結するだけである。これに対して、JP ポートの配線は複雑であるが、基本的にはホスト—クラスタ 0—クラスタ 1—と順につないでいくだけである。ただし、どこどこをつなぐべきかというのはかなり面倒な話になる。



## 第6章 G6 チップ機能の検討

本章では、G6 チップの詳細な動作を検討する。まず予測子パイプライン、次に相互作用パイプラインについて述べ、最後に制御部の演算時動作について述べる。

### 6.1 予測子パイプライン

予測子パイプラインの構成を図 6.1 に示す。

まず、入出力のデータ形式について述べる。位置の入力および出力は 64 ビット固定小数点形式とする。これは、同じビット数の浮動小数点形式に比べ実効的に精度があげられること、回路が若干単純化されること、誤差解析が単純化されることなどの理由による。なお、附加的な利点として、周期境界も扱える。不利な点としては、ホストでの処理が若干繁雑になることが主なものであるが、64 ビット CPU が一般化し、変換が従来の 32 ビット CPU に比べ高速化されているので大きな問題ではないであろう。実際、例えば 533MHz 21164 では 100MOPS 以上の速度で 64 ビット整数と 64 ビット浮動小数点数の間の変換が可能である。速度と、それ以上の高階導関数については、ブロック浮動小数点形式とする。速度の予測子の最終的な出力は、正規化した浮動小数点数を出す。これは、速度については固定小数点であつかうのは若干面倒があるからである。

演算精度は、基本的には位置の多項式の最終段の加算器は 64 ビット、速度は 32 ビットであり、その前の乗算器はそれぞれ 24 ビット、20 ビットとする。さらに前にいくほど 4 ビット程度ずつビット数を減らしていく。

ここでブロック浮動小数点を使いたい理由はいくつかある。

- 1) 必要なメモリバンド幅を減らすため
- 2) パイプラインの設計を単純化するため
- 3) パイプラインのゲート数を減らすため
- 4) パイプラインの動作速度を上げるため
- 5) パイプラインのテストを簡単にするため

以下、それぞれについて簡単に述べる。ブロック浮動小数点形式を使うことのもっとも大きな利点は、冗長な指数データを送らなくて済むのでデータ転送量が減らせるということである。具体的には、例えば位置に 64 ビット、速度等に 32 ビットとすれば、192 ビット必要になる。これに対し、速度以降を普通に 4 ビットずつ減らしたとすれば、1 座標成分のデータは  $168 (= 64 + 32 + 28 + 24 + 20)$  ビットと 24 ビット減らすことができる。これに対し、指数を速度だけに与えて、それ以降は共通のものとしたとすれば、さらに 24 ビット、すなわち最初の 192 ビットから比べれば 25% 近くデータを減らせることになる。粒子一つのデータ量は、ホストとのデータ転送ばかりでなく、仮想パイプラインの多重度にも影響するので、全体システムの性能を上げるために可能な限りデータ量を減らすべきである。

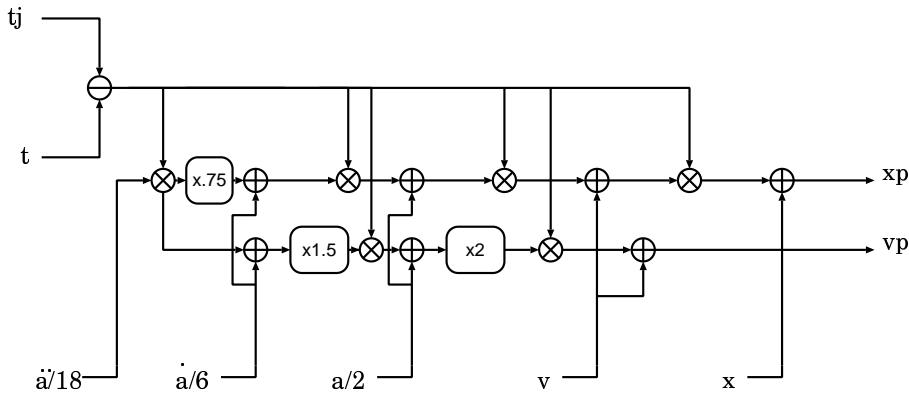


図 6.1: 予測子パイプライン

設計が簡単になるかどうかはそれほど自明ではないかも知れない。というのは、回路規模が小さく、また構成要素数が小さくなるのは確かだが、各演算で真の小数点位置やオーバーフローを考慮する必要があるために設計の詳細はかえって繁雑になるところもある。通常の浮動小数点を使った場合、演算器のなみをブラックボックスとしてよいのに対し、ブロック浮動の場合は実質的に固定小数点なので、話が面倒になるわけである。

ゲート数や速度については、ある程度の向上が見られるのは確実である。しかしながら、これはそれほど重要ではないであろう。というのは、もともと大した大きさではないからである（せいぜい 50k ゲート）。速度も、ここがクリティカルパスになることはあり得ない。

最後に、テストについてであるが、これについても、各演算器が簡単になっている分テスト設計は簡単であろう。現在のところスキャンバス設計をすることになっているが、オンボードで動作している状態でもある程度のテストを可能にするために、いろいろテスト用のアドホックな回路（といっても、基本的には各演算器の出力がそのあと素通りで最終段に現れるような仕掛け）をつけることにする。

演算は式(3.5)に示した通りだが、以下詳細を示す。まず、等価な Fortran プログラムを示し、演算ごとに書き下してそれぞれの演算フォーマットについて説明する。

```

DO J = 1,N
  S = TIME - T0(J)
  S0 = 0.75*S
  S1 = 1.5*S
  S2 = 2.0*S
  DO K = 1,3
    $      X(K,J) = (((F2DOT(K,J)*S3+FDOT(K,J))*S + F(K,J))*S +
    $                  VO(K,J))*S+ X0(K,J)
    V(k,j) = ((f2dot(k,j)*s+s+fdot(k,j))*s1 + f(k,j))*s2
    $                  + VO(k,j)
  enddo
enddo
  
```

これは、NBODY4 のコードを一部改変したものである。これを、以下のような演算順序で実現することにする。以下 C ライクなシンタックスで書く

- a) DT = T - TJ
- b) K4 = DT\*A2BY18
- c) K4A = K4 \* 0.75
- d) K3 = K4A + A1BY6
- e) K3A = K3 \* DT
- f) K2 = K3A + ABY2
- g) K2A = K2 \* DT
- h) K1 = K2A + V
- i) K1A = K1 \* DT
- j) XP = X + K1A
- k) K3C = K4 + A1BY6
- l) K3D = K3C \* 1.5
- l') K3E = K3D \* DT
- m) K2B = K3E + ABY2
- n) K2C = K2B \* 2
- o) K2D = K2C \* DT
- p) VP = V + K2D

次に時刻であるが、現在時刻のレジスタには 64 ビット固定小数点で時刻を格納することにする。

以下、各演算について述べる。

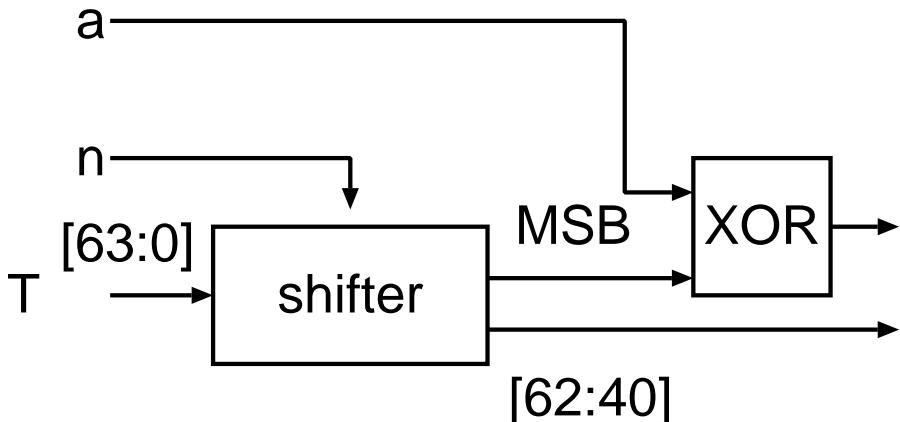
### 6.1.1 a) DT = T - TJ

もっともナイーブな実装としては、T, TJ の双方を 64 ビット固定小数点で与え、減算結果を正規化（浮動小数点表示に変換）すればよい。但し、実際に起きる計算について考えると、これはいかにも無駄である。ゲート数についてはせいぜい 5000 ゲートとチップの 1% 以下であり、たいした問題ではないが、メモリー量については 10% 以上を占め、ここで減らせることは重要である。

もう少し具体的に書いてみよう。増分  $\Delta t$  は現在時刻  $t$  から粒子の元の時刻  $t_i$  を引いたもの  $t - t_i$  であるが、この減算ではかならず大きな桁落ちがおきる。しかし、丸めは必ずしも起きない。これは、 $t_0$  の下位ビットは hierarchical timestep を使っている場合にはかならず 0 であり、 $\Delta t$  はその範囲にのみ現れるからである。

これは、言い替えれば実は  $t - t_i$  の「減算」は必要がないということである。具体例で考えてみよう。例えば粒子の時間刻み  $\Delta t_i$  が 0.25 であれば、 $t_i$  はその整数倍である。従って、 $t_i$  の  $\Delta t_i$  の値よりも下のビットはすべてゼロである。また、 $\Delta t$  は  $\Delta t_i$  を越えないで、 $\Delta t$  の最上位ビットよりも上のビットはゼロである。つまり、 $\Delta t$  の MSB 以外は、 $t$  の上位ビットをマスクしたもの、MSB は  $t$  の  $\Delta t$  の MSB 位置とその一つ上から、 $\Delta t$  の MSB を引いたものということになる。

引き算した結果の  $\Delta t$  は、 $\Delta t_i$  に比べた相対的な大きさとなっているようにすると話が簡単である。予測子の係数を、あらかじめ  $\Delta t_i$  でスケールしておけるからである。これにより、多項式による  $\Delta x$  等の計算を固定小数点形式（ブロック浮動）で行なうことができる。

図 6.2:  $\Delta t$  計算回路

まあ、減算器一つをけちったところでさしてシリコン面積を稼げるわけでもないのだが、テスト容易化という観点からは無駄な回路を省くのは重要であろう。

もっとも簡単な回路は、以下のようなものになる。

入力：

$T$  現在時刻  $t$

$a$  粒子  $i$  の時刻  $t_i$  を  $\Delta t_i$  で割った答え（整数）の LSB

$n$   $\Delta t_i$  の MSB の位置

出力は以下のように書ける

$$\text{output } < 23 : 0 > = T < n : n - 23 > \text{exor}(a << 23). \quad (6.1)$$

これでは丸めにバイアスが発生することがあり得るが、これは実際には非常に稀である（カレントブロックのタイムステップが  $2^{-24}\Delta t_i$  より小さい必要があるため）従って、丸め処理は行なわない。

実際の回路は図 6.2 のようになる。これは、回路が実質上 64 ビットのシフタだけの大きさ、遅延で済むので、設計が容易になるほか、時刻のデータが  $a$  に 1 ビット、 $n$  に 6 ビットの合計 7 ビットですむという大きな利点がある。これにより、必要メモリバンド幅、容量を 10% 以上減らすことができる。

出力は 24 bit unsigned integer で、通常のブロックタイムステップスキームの場合は値の範囲が 0x000000 から 0x800000 までとなる。0x800000 よりも大きな値は通常発生しないが、非ブロック化並列スキームなど、異常な使い方では発生しうる。しかし、標準的な動作モードでは、0x800000 よりも大きな場合は考慮しない。こうすることで実効的な精度を 1 ビット上げられるからである。なお、DT が負になる場合は考慮しない。このようなことが発生しうるスキームで使うためには、TJ をシフトして係数を計算し直せばよいからである。

### 6.1.2 多項式パイプラインの精度とブロック固定小数点表現に関する一般論

予測子に必要な演算精度は、もちろん達成したい精度の関数である。ここでは、2粒子間の重力の相対精度が24ビットとGRAPE-4と同程度ということにする。この精度が予測子の相対精度としても実現される必要がある。この時、時間刻み $\Delta t_i$ は以下の条件を満たしているとして良い:

$$\frac{a^{(3)}}{5!} \Delta t_i^5 < 2^{-24} v \Delta t_i \quad (6.2)$$

従って、いまテイラー展開の高次の項が等比級数的に小さくなるとすれば、 $a^{(2)}$ までの各項に必要なビット数は以下のようになる

項	$a^{(2)}$	$\dot{a}$	$a$	$v$
ビット数	6	12	18	24

理論上はこれで十分なはずだが、なんとなく心配なので実際に作るものは以下のビット数(符号含まず)を与えることとする。

項	$a^{(2)}$	$\dot{a}$	$a$	$v$
ビット数	10	16	20	24

さて、上の条件式6.2が満たされていない場合はちゃんと計算できているのであろうか? 級数の打ち切り誤差項が例えば $2^{-16}$ 程度の場合を考えてみる。この時、その上の項の大きさは $2^{-16}(3/4)$ 程度とすれば、一応問題はない。一般に、誤差項の底が2の対数が $-p$ 程度の時、その前の項は $-3p/4$ 程度であり、それに由来する丸め誤差は $-3p/4 - q$ (ここで $q$ は最高次の項のビット長)となる。従って、 $q = 6$ で十分な精度がすべての場合に得られるということがわかるであろう。より低次の項についても状況は同じである。

ここでも、 $\Delta t$ の計算と同じく、ビット長を減らすことによって得られる主な利点はメモリ必要量およびバンド幅の減少である。ゲート数の減少はたかが知れている。これは、そもそも予測子パイプラインはチップあたり1本ないしは1/3本であり、全ゲート数の10%以下であるからである。但し、固定小数点で演算を行なうことで、丸め誤差などの振舞いが単純になるという付加的な利点もある。

具体的な回路構成としては、ホーナーのアルゴリズムの1段毎に、高次の項は4ビットずつ右シフトされて次の項に足されるというふうにする(LSBは切捨て)。もともと入力のビット数が4ずつ違うので、これは実際には丸めたあとのLSBの位置を合わせて足すということである。さらに、最終的に求まった $\Delta x$ と $\Delta v$ を、必要なビット数だけシフトして $x_0$ などに足し込む。なお、 $\Delta v$ を $v$ に足す時にはシフトは必要ない。これは、そのようにあらかじめ小数点位置が選ばれているはずであるからである。すなわち、シフトが発生するのは $x_0$ に足されるところだけである。つまり、それ以外の演算はすべて固定小数点で行なわれていることになる。

なお、ここでの固定小数点形式は、2の補数でも絶対値+符号形式でもなんでもいいが、2の補数表示の場合は単純に切捨てを行なうとマイナス無限大にむかって丸めることになることに注意する必要がある。絶対値のバイアスはあまり問題にならないが、符号つきバイアスは保存量(重心位置)に直接誤差をもたらすので修正が厄介である。

丸めは乗算結果が桁落ちするときに生じるので、一つの方法は乗算は絶対値で行なうということであろう。で、符号を残してもう一度変換する。符号付乗算を行なうのに比較して回路規模が大きくはならないはずである。このためには、データ自体は符号+絶対値表示のほうが簡単である。

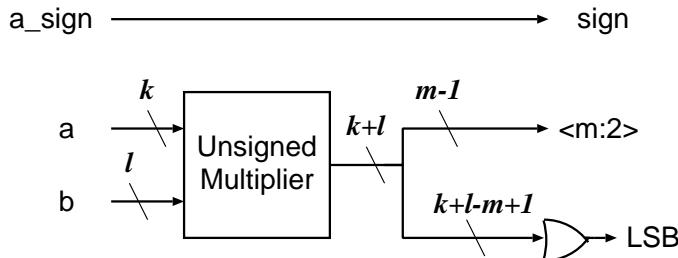


図 6.3: 予測子での乗算器の基本構成

もう一つの方法は、乗算結果を正負の場合両方とも正しく丸めるか、あるいはゼロに向かって丸めることである。これは負の時に全ビットを見て、キャリーを生成すればよい。次段の加算器がキャリー入力を持っていれば、遅延の増大はない。こちらのほうが加算器の実装は単純である。その代わり乗算器は2の補数表示が必要になる。なお、1.5倍および0.75倍回路についても、同様な丸めを行なう必要がある。

さて、設計の手間、ソフトウェアの手間という観点から見れば、固定小数点（ブロック浮動小数点）表示を本当に使うべきかどうかということも検討する必要がある。汎用の浮動小数点演算器1つで全体を構成するほうが設計は簡単だからである。しかし、これにはテスターの低下という問題もあり、また指数を8ビット程度しかとっていないと、高次の項がオーバーフローするといった問題もある。高次の項のオーバーフローを防ぐためにすべての項に指数を供給するのは、かなり大きな無駄（メモリバンド幅の15%程度）であることを考えると、ブロック浮動小数点を使うことのメリットはかなり大きいと判断できよう。

2の補数での乗算器は存在しない可能性があるので、以下符号つき絶対値表示の場合について考える。まず、乗算器の実現について述べ、次に加算器の実現について述べる。

### 6.1.3 乗算器

乗算器については、特に述べることはない。基本的には符号なしの乗算器で、符号はDTではない方のデータのものがそのまま伝わるだけである（図6.3）。ブロック浮動小数点表示であり、あらかじめオーバーフローが起きないように指数が選ばれているはずであるとする。指数の選択については後で述べる。

なお、まるめは corrected force-1 rounding、つまり、打ち切られるものがすべて0である場合を除いて、LSBを強制的に1にするという方法をとることにする。これは、例えばIEEEで使われている正しい丸めに比べて平均誤差が大きいが、回路が単純になるという利点がある。クリティカル・パスも短く、動作速度は上げやすい。特に、固定小数点の場合、單に結果の LSB とその下の全ビットを見て、すべて0でなければ結果の LSB を1にするだけである。

なお、単純 force-1 にしても大した違いはないような気がするかもしれないが、これはかならずしもそうではない。なぜなら、シフトアウトされるものが大きいか小さいかによってバイアスに差があるので、その分は補正不可能であるからである。例えば1ビットだけシフトアウトされる場合を考えると、修正 force-1 では 1/4 の確率で LSB が 0 になるのに対し、

単純 force-1 では常に 1 である。従って、0.125 LSB ものバイアスがあることになる。ところが、シフトアウトされるビット数が大きいと、指数関数的にバイアスが小さくなる。

なお、force-1 丸めのちょっと普通でない利点は、2 の補数表示をとった場合でも丸めのロジックは同じであることである。つまり、2 の補数表示で負数の場合でも、切捨てられるものがすべて 0 ならばそのまま、一つでも 1 があれば LSB を 1 にするというロジックは同じでいい。したがって、2 の補数表示での高速な乗算器が使えれば、そちらをつかった方が加算器の構成を大幅に単純化できることになる。

#### 6.1.4 定数乗算器

これらは、加算器と配線で実現する。ここでもオーバーフローは起きないように指数が選ばれているものとする。とする。0.75, 1.5 倍のところでは force-1 丸めが必要になる。仮に 2 の補数表示であるとすると、符号を保ってシフトする必要がある。

#### 6.1.5 加算器

加算器の構成を図 6.4 に示す。絶対値表現なので、加減算器を 2 つ使い、2 入力  $a, b$  の符号が違う場合には  $a - b$  と  $b - a$  の双方を計算する。結果が正になったものを選べばよい。この時、符号はどちらの演算結果が利用されたかで決まるので、同様に選ぶことができる。符号が同じ時には冗長な演算を行なっていることになる。ここではシフトは起きないので、丸めについて考える必要はない。乗算についてのみ丸め誤差を考えればよいのは、固定小数点形式の大きな利点であるといえる。

#### 6.1.6 b) $K4 = DT^* A2BY18$

これはビット長 10 ビットの乗算器。なお、 LSB 10 ビットは force-1 rounding を行なう。DT については入力前に丸めを行なう必要があることに注意。これは他の演算器でも同様である。

#### 6.1.7 c) $K4A = K4 * 0.75$

ここでは、1 ビット右シフトしたものと加算しさらに 1 ビットシフトする (LSB を force-1 丸め) ことで 0.75 倍する。符号はそのまま。

#### 6.1.8 d) $K3 = K4A + A1BY6$

ここでは、10 ビットの K4A と 16 ビットの A1BY6 を足し合わせる。符号つきなので、K4A 側の入力の上位ビットは 0 固定ということになる。

#### 6.1.9 e) $K3A = K3 * DT$

これはビット長 16 ビットの乗算器。丸めは force-1

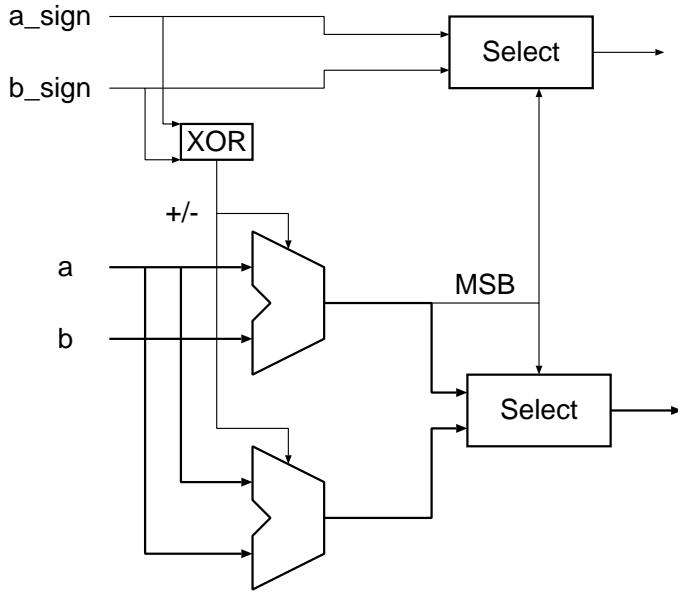


図 6.4: 予測子での加算器の基本構成

6.1.10 f)  $K2 = K3A + ABY2$ 

ここでは、16 ビットの  $K3A$  と 20 ビットの  $ABY2$  を足し合わせる。

6.1.11 g)  $K2A = K2 * DT$ 

これはビット長 20 ビットの乗算器。

6.1.12 h)  $K1 = K2A + V$ 

ここでは、20 ビットの  $K2A$  と 24 ビットの  $V$  を足し合わせる。

6.1.13 i)  $K1A = K1 * DT$ 

これはビット長 24 ビットの乗算器。

6.1.14 j)  $XP = X + K1A$ 

ここでは 64 ビット 2 の補数表示の  $X$  と、符号付非正規化浮動小数点数である  $K1A$  を足し合わせる。そのために、指数分だけシフトして  $K1A$  を固定小数点に直したあと、加減算器に入れる。64 ビットの右論理シフタを使って、MSB を合わせて入力する。シフト量は 64 から（指数-オフセット）を引いた値である。

**6.1.15 k)  $K3C = K4 + A1BY6$** 

ここでは、10ビットのK4と16ビットのA1BY6を足し合わせる。符号つきなので、K4側の入力の上位ビットは0固定ということになる。

**6.1.16 l)  $K3D = K3C * 1.5$** 

ここでは、1ビット右シフトしたものと加算することで1.5倍する。シフト時に LSB は修正 force-1 丸めを行なう。符号はそのまま。

**6.1.17 l')  $K3E = K3D * DT$** 

これはビット長16ビットの乗算器。丸めは force-1

**6.1.18 m)  $K2B = K3E + ABY2$** 

ここでは、16ビットのK3Eと20ビットのABY2を足し合わせる。

**6.1.19 n)  $K2C = K2B * 2$** 

これは単に配線で1ビットシフトするだけである。

**6.1.20 o)  $K2D = K2C * DT$** 

これはビット長20ビットの乗算器。

**6.1.21 p)  $VP = V + K2D$** 

ここでは、20ビットのK2Dと24ビットのVを足し合わせる。ここは演算(h)と同一である。さらに、結果を正規化し、ゼロフラグを設定する。正規化の際には、leading zeros の分入力の指数から引く。ここで指数を10ビットに拡張し、オフセット512を加える。また、さらに指数からDTJMLSBを引いておく。これは、もともと速度の指数はタイムステップを繰り込んだ形で入力されているからである。

なお、結果として浮動小数点表現は36ビットになるが、これはMSBから以下の順でしまわれる

1. 指数 10 ビット
2. 符号 1 ビット 1なら負
3. ゼロフラグ 1 ビット 1ならゼロ
4. 仮数 24 ビット 非ケチ表現、絶対値

### 6.1.22 ブロック浮動小数点における指数の決定

ここでの指数の決定は、基本的には

- 最終結果がオーバーフローしないこと
- 各演算でそれぞれオーバーフローが起きないこと

という条件を満たす必要がある。それだけでなく、区間の端に近いところ ( $\Delta t$  が  $\Delta t_i$  に近い) で、演算精度が保たれる、すなわち桁落ちが小さい必要もある。

まず、位置の多項式について考える。予測子の  $x_0$  からの変位を

$$\Delta x = a^{(2)} \Delta t^4 / 24 + a' \Delta t^3 / 6 + a \Delta t^2 / 2 + v_0 \Delta t. \quad (6.3)$$

と書く。今、 $s = \Delta t / \Delta t_i$  とすれば、 $0 \leq s \leq 1$  であり、上の式を以下のように書き直せる

$$\Delta x = As^4 + Bs^3 + Cs^2 + Ds. \quad (6.4)$$

但し、

$$A = a^{(2)} \Delta t_i^4 / 24, \quad (6.5)$$

$$B = a' \Delta t_i^3 / 6, \quad (6.6)$$

$$C = a \Delta t_i^2 / 4, \quad (6.7)$$

$$D = v_0 \Delta t_i, \quad (6.8)$$

$|\Delta x|$  が  $s$  の定義域の全域で 1 を越えないための十分条件は

$$|A| + |B| + |C| + |D| \leq 1. \quad (6.9)$$

であるので、これを満たす最小の値に指数を決めればよい。但し、これだけではまだ中間結果が溢れる可能性がある。ややこしいので具体例で考える。例えば

$$A = 1, \quad B = C = D = 0, \quad (6.10)$$

の場合を考えてみる。この場合、最終結果は 1 を越えないで別に問題はないように見える。しかし A は実はパイプライン中で実効的に 14 ビット右シフトされるので、A が可能な最大値であったとしても  $\Delta x$  の値は  $2^{-14}$  にしかならない。従って、実際には A は初めから絶対値が  $2^{-14}$  以下である必要がある。 $|B|$  と  $|C|$  についても、同様な配慮が必要であり、さらに速度の予測子の方でもオーバーフローが起きないようになっている必要がある。

具体的な手続きとしては、それぞれを基準に指数を決定し、最大のものをとればいいわけである。

もうちょっとちゃんと書いてみる。速度については、入力値の絶対値が 23 ビットに収まっているればいい。指数の解釈としては、24 ビット目のところの小数点があるものと思えば、元の指数から 2 を引いたもの（バイアスは同じとして）になる。

加速度については、 $\Delta t$  を掛けたものが 19 ビットに収まっていればいい。

今、位置の予測子の固定小数点化する前の出力が

$$s \cdot 2^e \cdot m \quad (6.11)$$

表 6.1: $C_k$ の値		
k	位置	速度
1	1	1
2	1	2
3	1	3
4	3/4	3

という形になっているとする。ここで  $s$  は符号で  $\pm 1$  であり、 $e$  は整数の指数、 $m$  は仮数部で 1 を越えないものとする。

今、 $e$  は与えられている。また、タイムステップの最大値は  $2^{-l}$  であるとしよう。一般に  $x^{(k)} = a^{(k-2)}$  (に  $1/2, 1/6, 1/18$  の係数を掛けたもの) についてどのような演算が起きるかを考えてみる。これらがもともとは

$$p_k = s_{k,0} \cdot 2^{e_{k,0}} \cdot m_{k,0} \quad (6.12)$$

という形で表現されていたとする。予測子の出力として出てくるものは、2 倍とか 1.5 倍の定数との乗算を別にすれば、時間刻が最大値の場合で  $-lk$  だけ指数部が小さくなつたものがでてくる。つまり、いま、 $k$  次の項の予測子への寄与の最大値を単に  $A_k$  と書くと、

$$A_k = C_k s_{k,0} \cdot 2^{e_{k,0}-lk} \cdot m_{k,0} \quad (6.13)$$

ということになる。ここで  $C_k$  は定数であり、表 6.1 に与えられる。

パイプラインのブロック浮動小数点計算でなにが起きるかを考えてみる。各係数が

$$p_{k,b} = s_{k,0} \cdot 2^{e_{0,b}} \cdot m_{k,b} \quad (6.14)$$

という形で表現されていたとすると、(指標  $e_{0,b}$  は係数の次数  $k$  によらないことに注意)  $\Delta t$  が最大値の時にはこれらに  $C_k$  が掛かっただけでで出るので、結局

$$2^{e_{k,0}-lk} \cdot m_{k,0} = 2^{e_{0,b}} \cdot m_{k,b} \quad (6.15)$$

という関係がなり立つ必要があることになる。この関係式から、 $e_{0,b}$  を与えれば、

$$M_{k,0} - M_{k,b} = e_{0,b} - e_{k,0} + lk \quad (6.16)$$

の形で、 $p_{k,b}$  の MSB がどこに出るかが定まる。但し、 $M$  は仮数部の値を、MSB の前に小数点があると解釈して底が 2 の対数をとった値、すなわち仮数部の MSB にならんでいる 0 の数にマイナス記号をつけたものである。

さて、演算中にオーバーフローが起きてはいけないということを考慮すると、逆に各  $p_{k,b}$  について MSB の上限位置が決まっていることになる。これは、速度の MSB からの相対位置では表 6.2 のようになる。

ここで、速度 ( $k = 1$ ) 以外ですべて 2 ビットずつ余計に下げているのは、最大で 3 倍にされることがあるためである。速度で 1 ビット下げているのは桁上がりの可能性を考慮したた

表 6.2: 予測子係数の MSB の位置

k	$M_{k,\min}$ (MSB 位置の上限)
1	1
2	$2 + 4 = 6$
3	$2 + 8 = 10$
4	$2 + 14 = 16$

めである。 $M_{k,0} - M_{k,b}$  が 1 から 4 までのすべての  $k$  に対して  $M_{k,\min}$  よりも大きくないといけないので、結局  $e_{0,b}$  の値は

$$M_{k,\min} \leq e_{0,b} - e_{k,0} + lk \quad (6.17)$$

をすべての  $k$  について満たす最小の値ということになる。

なお、実際には位置が固定小数点形式なので、ブロック指数  $e_{0,b}$  の値はさらにその小数点位置の分だけずらす必要がある。相互作用パイプラインにはこの値をそのまま入れる。

さて、今、位置の補正項のシフトするビット数がメモリから入ってくるものとすると、これは予測子の速度の指数とは一致していない。というのは、位置の補正項は  $\Delta t$  が掛かった形で出てくるべきであるのに対し、速度の方はそれが掛かっていてはいけないからである。いいかえれば、速度の指数は  $\Delta t_i$  の指数だけずれている必要があることになる。これは単に実際にその分ずらせばいいことになる。

### 予測子パイプラインのためのデータ変換の詳細

以下、シミュレータプログラムにおけるデータ変換の実現について詳細に述べる。変換は、以下の 4 つのパラメータによって規定される：

- **xunit** 空間分解能を与える。固定小数点形式での 1 が元の浮動小数点形式での  $2^{-xunit}$  を表すということにする。いいかえれば、小数点の位置が LSB から **xunit** 目の下にあるということになる。
- **tunit** 時間分解能を与える。解釈は **xunit** と同様である。
- **tj** 粒子の時刻を通常の浮動小数点形式で与える
- **dtj** 粒子のタイムステップを通常の浮動小数点形式で与える。ただし、これは実際には 2 のべきに (1 より小さい) になっている必要がある。

まず、時刻の変換について述べる。時刻については、必要なものは  $a$  と  $n$  であった。 $n$  は次のように定義される。

$$n = \log_2(DTJ/DTMIN) = \log_2(DTJ) + tunit \quad (6.18)$$

次に、 $a$  は TJ を DTJ で割った答えの LSB なので、C のシンタックスなら

```
a = ((ULONG) (tj/dtj)) & 1
```

とするだけである。ここで `ULONG` は `unsigned long` であり、`long` が 64 ビットの計算機を想定している。64 ビット整数のデータ型がない計算機では使えない。それが `long` ではない計算機では適宜 `typedef` を変更する必要がある。

次に、位置 `x` であるが、固定小数点化したものは単に

```
ix = rint(scalb(x,xunit));
```

で与えられる。`scalb` は `xunit` だけ `x` の指数を増やし、`rint` は IEEE-754 の正しい丸めをして浮動小数点数を整数に変換する。

ここまでまあなんということはない。問題は `v` 等である。これらを、今、配列 `x[5]` と `ix[5]` に入っているものとする（0番目の要素は位置である）。まず、式 (6.16) を満たすように共通の指數  $e_{0,b}$  を決める。これは、具体的には例えば `frexp` を使って指數を決めればいい。なお、実際にはここで指數は 3 成分の指數の最大のものをとることにする。

共通の指數  $e_{0,b}$  を決めたら、各係数の MSB の位置は

$$MSB_k = 24 + e_{k,0} - e_{0,b} - lk \quad (6.19)$$

から求まる。式 (6.16) と符号が反転しているのは下から数えているためであり、24 は仮数部の語長である。

さて、最後に、ブロック指數の値を決めないといけない。これは、既に求まっている  $e_{0,b}$  に、まず `xunit` を足してやる必要がある。

仮に `xunit` を加えてもまだ指數が負であったとすると、これは、この指數の範囲内では速度が表現できない（アンダーフローしている）ということである。この時は、とりあえず仮数部、指數部とともに 0 にしておく。なお、この時は高次の項も強制的に 0 にしておく必要があることに注意してほしい。

### 6.1.23 テストモード

- a)  $DT = T - TJ$   $V=1$ , それ以外 0 とすることで直接観測
- b)  $K4 = DT * A2BY18$  位置予測子で A2BY18 以外を 0 とし、後の乗算器をバイパスモードにすることで観測
- c)  $K4A = K4 * 0.75$   $DT=1$ , A2BY18 以外を 0 として観測
- d)  $K3 = K4A + A1BY6$   $DT=1$ , ABY2 より低次の項を 0 に
- e)  $K3A = K3 * DT$  A1BY6 以外 0, 後の乗算器をバイパスモードに
- f)  $K2 = K3A + ABY2$   $DT=1$ , A1BY6, ABY2 以外の項を 0 に
- g)  $K2A = K2 * DT$  ABY2 以外 0, 後の乗算器をバイパスモードに
- h)  $K1 = K2A + V$   $DT=1$ , ABY2, V 以外の項を 0 に
- i)  $K1A = K1 * DT$  V 以外 0
- j)  $XP = X + K1A$   $DT=1$ , V, X 以外の項を 0 に
- k)  $K3C = K4 + A1BY6$   $DT=1$ , ABY2 より低次の項を 0 に
- l)  $K3D = K3C * 1.5$   $DT=1$ , A1BY6 以外を 0 として観測

表 6.3: 予測子パイプラインへの演算中入力データ

変数	ビット数	個数
位置	64	3
速度	32	3
加速度	21	3
第一導関数	17	3
第二導関数	11	3
時刻	8	1
質量	36	1
番号	32	1
合計	—	511

l')  $K3E = K3D * DT$  A1BY6 以外 0, 後の乗算器をバイパスモードに

m)  $K2B = K3E + ABY2$  DT=1, A1BY6, ABY2 以外の項を 0 に

n)  $K2C = K2B * 2$  ABY2 以外 0, DT=1

o)  $K2D = K2C * DT$  ABY2 以外 0, 前の 2x をバイパスに

p)  $VP = V + K2D$  DT=1, ABY2, V 以外の項を 0 に

#### 6.1.24 メモリとのインターフェース

メモリとの接続は、現在の想定では 64 ビット幅同期インターフェースである。このクロックは一応相互作用パイプラインのクロックと等しいものとする。粒子 1 個分のデータは表 6.3 のようになる。

メモリバス幅を 64 ビットとすれば、これは 8 ワードに収めることができる。なお、質量については、ホストは IEE-754 単精度で書き込む。メモリから読み出してから GRAPE-6 相互作用パイプラインの内部形式に変換して予測子パイプラインに送るということにする。これはちょっと面倒で、以下のような作業が必要になる。

1. ゼロかどうか判定する。これは、IEEE-754 なので本来全ビットが 0 なら 0 であるので、この時にだけ 0 フラグを立てる。
2. 非数、無限大については、判定しない。これは、変換後のデータに対応する表現がないからである。したがって、これらも普通の数に変換されてしまう。質量が非数だったり無限大だったりすることは本来ないので、これは問題ではない。
3. それ以外の普通の数については、仮数は削っていた 1 を付け加え、指数はオフセットが違うのでその分を加算するだけである。オフセットは 386 (384 でないことに注意) である。なお、この処理は 0 であった時も行なうことに注意してほしい。

メモリへの書き込みは、入力インターフェースから渡ってきたデータをそのまま書くだけである。読み出しほは、連続アドレスで読みだし、8 ワードごとにラッチして予測子パイプラインに渡す。

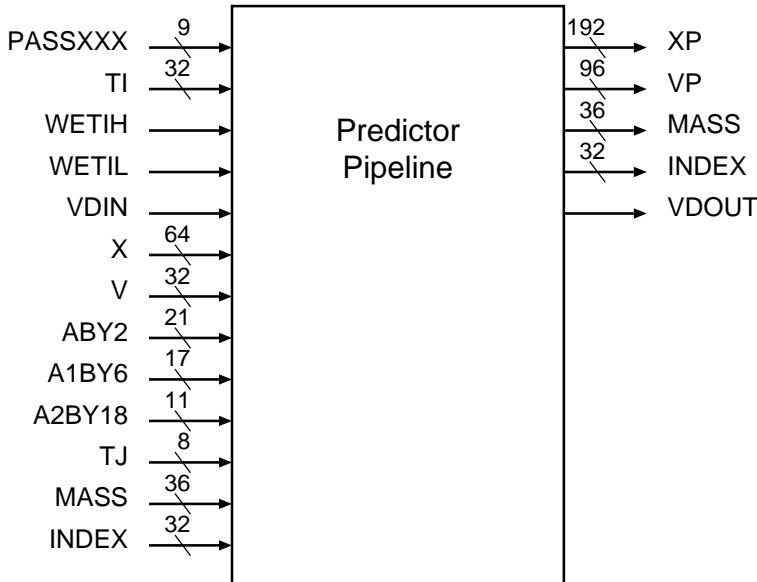


図 6.5: 予測子パイプライン内部インターフェース

ここで注意して欲しいことは、予測子パイプラインは実は1本ある必要は全くないことがある。つまり、8クロックの間にx, y, zの3成分を計算出来ればいいのである。これをを利用して回路規模を小さく、また設計（特にテスト設計）を簡単にすることはできないだろうか。

一つの方法は、位置と速度のパイプラインを兼用することである。この2つはほとんど同じ回路であり、違いは定数倍をしたりしなかったりするだけである。したがって、サイクル毎に定数倍とバイパスを切替えることで、両方を兼用させることはそれほど難しいわけではない。

### 6.1.25 チップ内他ユニットとのインターフェース

前節ではメモリとのインターフェースについて書いたが、設計上はそこは制御部が面倒をみることであり予測子ユニットの問題ではない。ここでは、チップ内の他ユニットとのインターフェースを示す。図 6.5 に信号定義を与える。

予測子ユニットのデータ入力は表 6.3 の通りであるが、さらに  $t_i$  の入力が必要である。これに加えてさらに制御入力がある。これらを表 6.4 にまとめる。ここで、位置から第二導関数までは、 $x, y, z$  の3成分を持つことに注意して欲しい。これらは3サイクルで順に渡ってくるものとする。それ以外のデータは最初のサイクルで渡される。

なお、VD はデータが有効であることを示す信号（アクティブ・ハイ）であり、一度出たらそのあと7サイクルはでることがないものとする。PASS なんとかの一連の信号は、パイプライン動作にあわせてラッピングする必要はない。

図 6.6 に予測子の入力のタイムチャートを示す。

時刻  $t_i$  の入力については、専用のポートから WETI(H/L) 信号（アクティブ・ハイ）と同期して書き込むものとする。タイムチャートは図 6.7 のようになる。H, L の二本でそれぞれ上位、下位の32ビットに書き込む。

表 6.4: 予測子パイプラインへの入力信号

入力名	機能
TI	現在時刻 (32 ビット)
WETIH/WETIL	現在時刻書き込み信号 (アクティブハイ)
VD(Valid data)	演算用データが揃っていることを示す
PASSK4	K4 に A2BY18 をコピーする
PASSK4A	K4A に K4 をコピーする
PASSK3A	K3A に K3 をコピーする
PASSK2A	K2A に K2 をコピーする
PASSK1A	K1A に K1 をコピーする
PASSK3D	K3D に K3C をコピーする
PASSK3E	K3E に K3D をコピーする
PASSK2C	K2C に K2B をコピーする
PASSK2D	K2D に K2C をコピーする

```

clk      _____/=====\_____
VD       /=====\
DATA    <datax0><datay0><dataz0>-----<datax1><datay1><dataz1>

```

図 6.6: 予測子入力タイムチャート

予測子パイプラインの出力は、位置、速度、質量、インデックスである。質量、インデックスは、単にパイプラインレジスタで待ち合わせするだけである。ただし、さらにこれに加えて、VD に相当する「データが揃った」という信号を出す。つまり、出力は表 6.5 のようになる。

図 6.8 に予測子の出力のタイムチャートを示す。

## 6.2 相互作用パイプライン

相互作用パイプラインの構成を図 6.9 に示す。計算精度は、位置の初段が 64 ビット、最終段（加速度）が 64 ビットの固定小数点とする。速度は入力が浮動小数点 32 ビット、最終段（加速度の第一微分）は 32 ビットの固定小数点とする。また、ポテンシャルについても 64 ビットの固定小数点で積算するものとする。なお、それぞれ独立に指数バイアスを設定できるものとする。つまり、実効的には積算に関してはブロック浮動小数点ということになる。

```

clk      _____/=====\_____
WETIx   /=====\
TIDATA  <=data==>-----

```

図 6.7: 予測子 TI 書き込みタイムチャート

表 6.5: 予測子パイプライン出力

変数	ビット数	個数
位置	64	3
速度	36	3
質量	36	1
番号	32	1
VD	1	1

```

clk      =====
VD       /=====\
DATA    <=data0>-----<=data1>-----

```

図 6.8: 予測子出力タイムチャート

なお、いうまでもないが VMP の場合にはこの指数レジスタを独立にもつ必要がある。

中間の演算については、加速度は仮数 24 ビット、指数 10 ビットで行ない、その微分は仮数 20 ビット、指数 10 ビットで行なうことにする。内部表現で指数を多めにとってもそれほどゲート数には影響しないので、ややこしい問題を防ぐために指数はここでは多めにとっておく。テストビリティに影響がでたら減らすこととも考える。

浮動小数点表現はゼロフラグ、符号ビットを合わせて 36 ビットになる（仮数 24 ビットの場合）が、これは MSB から以下の順でしまわれる

1. 指数 10 ビット
2. 符号 1 ビット 1 なら負
3. ゼロフラグ 1 ビット 1 ならゼロ
4. 仮数 24 ビット 非ケチ表現、絶対値

なお、今回は対数表示などはつかわない。

また、周期境界条件のもとでの計算を Ewald 法により高速に行なうため、12 ビット程度の絶対精度でカットオフ関数を計算する仕掛けをつける。これはポテンシャルと加速度のみにつけ、導関数にはつけない。これは、周期境界で非常に高精度の計算をするということはとりあえず考えないためである。（これは後で変更する可能性がある）

以下に相互作用パイプラインでの計算の Fortran ソースを示す。これは NBODY1H からとったものである。

```

DO 15 J = 1,N
  dr2 = EPS2
  drdv = 0.
  DO 11 K = 1,3
    dx(K) = X(K,J) - X(K,I)
    dv(K) = XDOT(K,J) - XDOT(K,I)
    dr2 = dr2 + dx(k)**2
  11 CONTINUE
  drdv = dr2
  dr2 = dr2 + drdv
15 CONTINUE

```

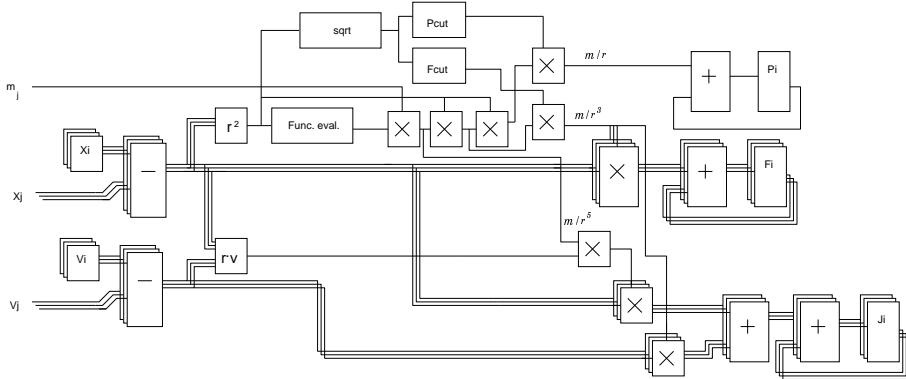


図 6.9: 相互作用パイプライン

```

        drdv = drdv + dx(k)*dv(k)
11      CONTINUE
*
dr2i = 1.0/dr2
dr3i = BODY(J)*dr2i*SQRT(dr2i)
drdv = 3.*drdv*dr2i
*
DO 12 K = 1,3
    Firr(K) = Firr(K) + dx(K)*dr3i
    FD(K) = FD(K) + (dv(K) - dx(K)*drdv)*dr3i
12      CONTINUE
15 CONTINUE
    
```

### 6.2.1 $dx(K) = X(K,J) - X(K,I)$

入力は 64 ビット固定小数点、出力は仮数 24 ビットの浮動小数点である。出力はゼロフラグ、符号、指数 6 ビットを合わせた合計 32 ビットとなる。ただし、めんどうなのでここで指数は 10 ビットに拡張しておく。なお、仮数 24 ビットへの丸めは修正 force-1 とする。また、指数はバイアスフォーマットとする。これは大小比較を簡単にするためである。

なお、修正 force-1 とは、切り捨てられる仮数部がゼロでなければ、結果の仮数部の LSB を 1 にするという方法である。この方法は、IEEE-754 で採用されている「正しい」丸めと同様、バイアスのない回路を実現できる。また、回路構成が単純である。そのかわり、最大誤差は 1 LSB と、IEEE-754 の 0.5 LSB に比べて大きい。

このことは、しかし、実際上問題ではないだろう。というのは、仮数部を 1 ビット長くすれば済む話だからである。正しい四捨五入の場合はさまざまな付加回路が必要になる。特に、結果をシフトした後で加算する必要が生じる。このための付加回路の規模は、仮数部 1 ビット分増やすのとほぼ同程度である。

force-1 や正しい丸めの代わりに、いわゆるチョッピングを行なったほうが一見誤差が小さくなるような気もする。これは、チョッピングでは誤差は最大 1 LSB であるが、「真の値」

が 0.5 LSB ずれていると思えばそこからの誤差は最大 0.5 LSB であるような気がするからである。但し、実は LSB の相対的な大きさは、仮数の値自体によって最大 2 倍変化するので、このような補正をしてもバイアスが距離の絶対値によって変わることになり、正確には望ましくない。

さて、厳密にいうと問題なのは、「引き算した答が正確に  $-2^{63}$  であった場合の処理」である。もちろん、自由境界の計算をする場合には、そもそも入力座標の絶対値が  $2^{62} - 1$  を超えないようになっている必要があるので、引き算した答が  $-2^{63}$  になることはありえない。しかし、周期境界を使うとこの値が発生し得ることになる。

このようなことが起きるのは、2 の補数表示が正負対称ではないためである。しかし、実際にこれは問題なのであろうか？整数表現の周期境界を使って計算できるということは、力のカットオフがボックスサイズの半分よりも小さいということである。というのは、そうでなければミニマムイメージ以外からの力も計算する必要がおきているはずだからである。したがって、引き算した答が  $-2^{63}$  になった場合、いずれにしても力の絶対値は 0 であるべきであり符号は関係ないものと考えられる。

しかし、厳密にいうとまだ問題であるのは、そもそも、周期境界を表現するのに  $2^k$  をすべて使いたいかどうかということである。つまり、正負で壁の位置が微妙に違うことになっているのである。これはしかし周期境界の原理からいって大きな問題ではないのかもしれない。もうすこし理論的な検討が必要である。

まず、変換前の数が（有限精度ではなく）真に連続分布する実数である場合について考えてみる。周期境界を例えば  $[0, 1)$  という形で与えたとすると、常識的な変換は、符号つき整数を unsigned integer と思って、 $[0, 1)$  を 0 から  $2^k - 1$  （ここで  $k$  は整数の語長）に対応させるものであろう。この場合、演算結果については、ちょうど  $2^{k-1}$  だけ離れている場合を除いては対称であり、周期境界についても正しく表現できている。但し、一般に整数  $i$  にマッピングされる区間が  $[i/2^k, (i+1)/2^k)$  になっていないと、そもそも  $[0, 1)$  がマップされていることにならないことに注意して欲しい。すなわち、四捨五入ではなく切捨てによって変換しないと、奇妙な現象がおきていることになる。

仮に四捨五入したとすると、整数  $i$  にマッピングされる区間が  $[(i - 0.5)/2^k, (i + 0.5)/2^k)$  になる。ここで問題は  $[(2^k - 0.5)/2^k, 1)$  の区間はどこにいってしまうかということであり、これを 0 にマップさせれば周期境界という意味では正しい。かりにこれが  $2^k - 1$  に押しつけられると、0 と  $2^k - 1$  の間で非対称性が生じていることになる。

以上の考察から、変換前の数が真に連続分布であれば、2 の補数表示に変換して特に問題は生じないということがわかる。しかし、現実には変換前の数はすでに浮動小数点か固定小数点かのいずれかであり、通常は浮動小数点である。もとが固定小数点ならば、もちろん問題は生じようがない。もとの表現のビット長の方が長い場合には切捨て、短い場合にはゼロで埋めることで、周期性は保存されるので力のバイアスが生じることはない。

しかし、浮動小数点の場合にはどうなるのか良くわからない。例えば浮動小数点で  $[0, 1)$  の範囲を使った場合を考えてみる。この時は、そもそも実数での表現自体が一様ではない。指数がある値よりも大きい時は、固定小数点に変換されるときに丸めは発生しない。これに対して、それ以外の場合は丸めが発生することになる。この丸めの効果はちょとはっきりしない。

もちろん、実効的に固定小数点表示になるような座標系をとることで、上の問題を完全に回避することも理論的には可能である。これは、具体的には、空間座標を  $[0, 1)$  ではなく  $[1, 2)$  の範囲にとることで実現される。この場合には、指数の値が常に等しいので浮動小数点から固定小数点への変換が、(hidden bit を除いた) 仮数部をとることで直接に実現される。いいかえれば、もともと固定小数点表現を使っているのと同じことなので、位置によって分解能がことなるといった問題は生じないことになるわけである。

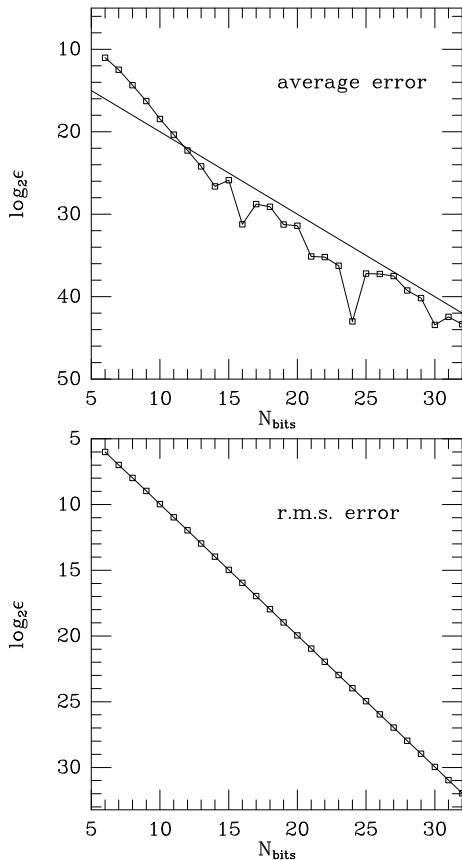


図 6.10: 乗算器のシステムテックバイアス（左）と rms 誤差（右）

### 6.2.2 $dx(K)$ の乗算

これは通常の 24 ビット仮数の浮動小数点乗算器である。ここから指数を 10 ビットに拡張する。なお、仮数の正規化は一応やることにする。ゲート数がきつくなったら別途考える。ここでも force-1 による丸めを行なうものとする。

例えば、ここでは正規化は省き、次のステップでまとめて正規化して、最終結果がでてから一度だけ正規化し直せば、必要なシフタの数を半分程度にできる。

図 6.10 と図 6.11 にシミュレータによる誤差の測定結果を示す。図 6.10 は  $2^{23}$  回乱数を出して平均をとった一次と二次の誤差である。二次の誤差についてはビット長に見あった精度が得られている。一次については、誤差がビット長に見あってはいるが、かならずしも小さくはない。これは、試行数が十分には大きくないためである。この状況は図 6.11 を見れば理解されよう。左は一次の項の試行数への依存（別の曲線はビット長が違う）を示している。非常にビット長が短い場合は、試行を増やした時に一次の誤差が一定値に漸近する。これは、もともと GRAPE による表現が有限個の値しかとらないので、試行数を無限にしても自由度が無限にならないためである。

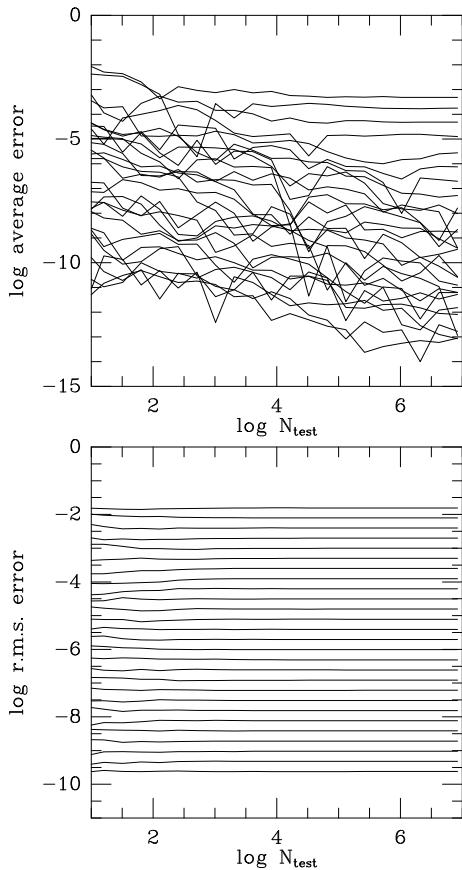


図 6.11: 乗算器のシステムティックバイアス（左）と rms 誤差（右）のサンプル数への依存性

### 6.2.3 $dr2 = dx(1)^{**2} + dx(2)^{**2} + dx(3)^{**2} + eps2$

とりあえず force-1 丸めによる加算を行なう。浮動小数点加算器を 3 個用いる。但し以下のようにしてゲート数を減らすことを場合によっては検討する必要がある。

- 正の数同士なので、ポストシフトは 1 ビット右シフトしかあり得ない。
- 正の数 4 個を足すので、まとめて 1 回だけシフトすればよい。

以下、加算器（符号なし）のシミュレーション結果を示す。

図 6.12 と図 6.13 にシミュレータによる誤差の測定結果を示す。図 6.12 は  $2^{23}$  回乱数を出して平均をとった一次と二次の誤差である。振舞いについては乗算器の場合と基本的には同じであることが理解されよう。

### 6.2.4 $1/r^3$ と $1/r$

これらは（ほぼ）同一の 2 次補間器から生成する。 $1/r^5$  についても同様である。基本的には、GRAPE-2 での補間器と同等の仕様となる。すなわち、大雑把にいうと、

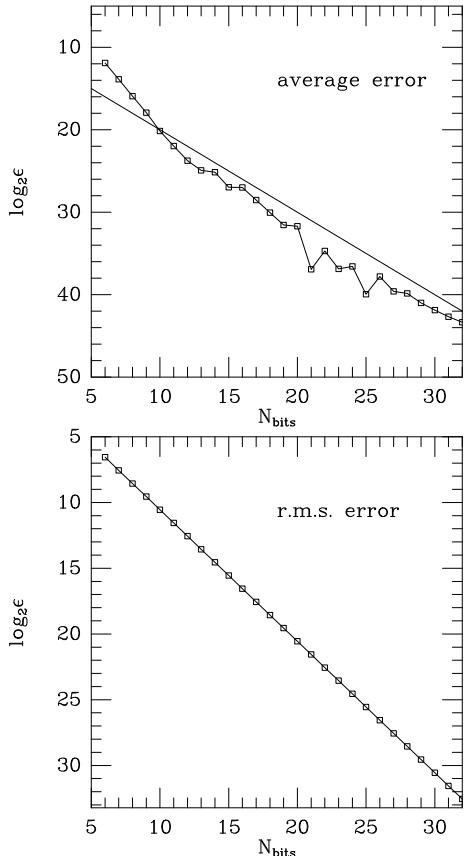


図 6.12: 符号なし加算器のシステムバイアス（左）と rms 誤差（右）

- 指数は演算してベースの指数を求める。
- 近似多項式のテーブルは、0次の項については指数を出す。この指数は前項のベース指数に加えられる
- 最大 1 ビット多項式の計算結果はシフトされる。この時さらに指数は加算される。これはキャリーに入れれば余計なインクリメンタはいらない。

但し、補間器 1 つの大きさが、乗算器 1 つよりも大きければ、 $1/r^5$ だけを計算してあとはそれに  $r^2$  を掛けて求めるという方法もありえる。これは、少なくとも設計が簡単という利点はある。また、 $r^2$  を掛ける前に質量を掛けておけば、乗算器を 2 個節約できる。言い換えれば、乗算器の数を変えないで補間器を 1 つで済ませられることになる。したがって、計算精度のことを考えなければ、このやり方が望ましい。 $1/r^5$  はベキが大きいので、結果として得られる  $1/r$ ,  $1/r^3$  の誤差をちゃんと評価する必要がある。が、例えば  $1/r^5$  を 1-2 ビット高い精度で求めておけば問題はないであろう。

なお、負ベキの計算なので係数の正負は項毎に変わる。また、高次の項が小さいことは保証されている。特に、補間の結果符号が変わることはない。したがって、それらの性質を利用した回路ということになる。

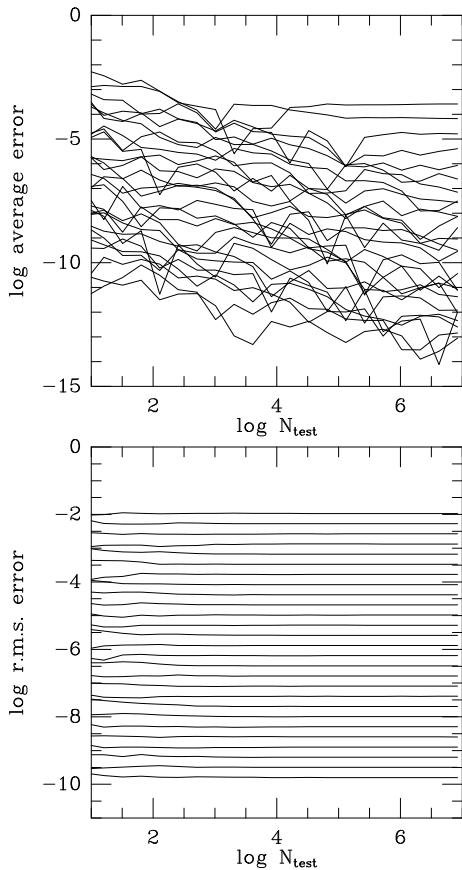


図 6.13: 符号なし加算器のシステムバイアス（左）と rms 誤差（右）のサンプル数への依存性

図 6.14 にシミュレータによる誤差の測定結果を示す。図 6.12 は  $10^6$  回乱数を出して平均をとった二次の誤差である。

図 6.14 から、 $x^{-3/2}$  と  $x^{-5/2}$  で誤差の違いはそれほど大きくななく、実用的な範囲のなかでほとんど 2 倍程度になっていることがわかる。

なお、ここで実用的とは、要求精度に対して入力ビット数とテーブルサイズを考慮して、トータルの必要シリコン面積が最小になるような組合せという意味である。これを一般的に実現するには、結構ややこしい考察が必要になるが、今回の場合、例えば要求精度を  $10^{-7}$  とすれば、エントリーが 7 ビットでは達成出来ないし、8 ビットで達成できる。これに対して、それ以上エントリーを増やしても入力ビットを減らすこととは出来ない。したがって、例えばエントリー 8 ビットテーブル 24 ビット、あるいは余裕をみてエントリー 9 ビットテーブル 25 ビットで十分であることがわかる。

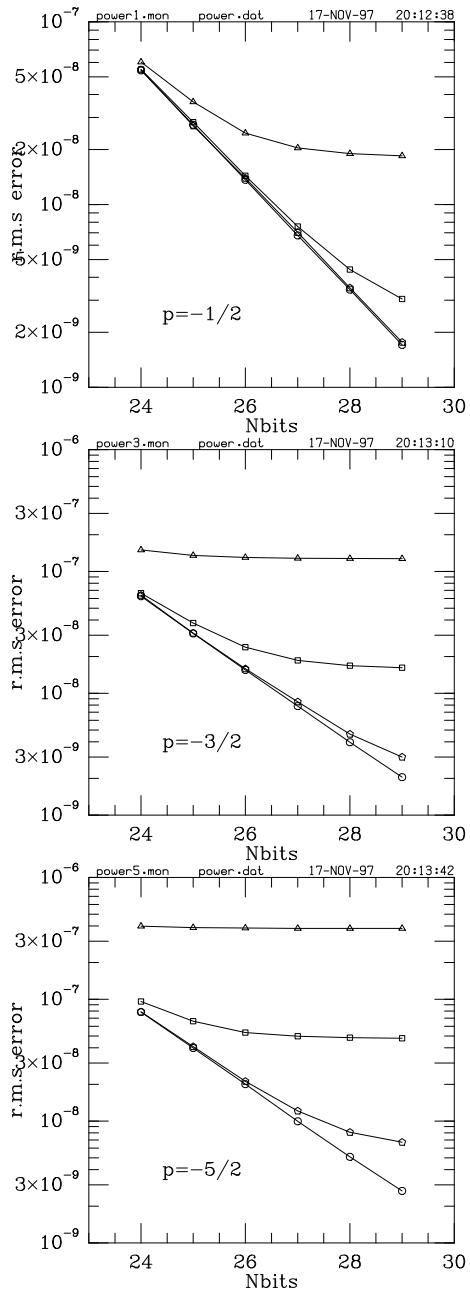


図 6.14: 補間器による冪乗計算の自乗平均相対誤差。左上は  $x^{-1/2}$ , 右上は  $x^{-3/2}$ , 下は  $x^{-5/2}$  である。それぞれ、横軸は 0 次の項の仮数部ビット数であり、縦軸は r.m.s. 誤差である。三角、四角、五角および丸はそれぞれテーブルエントリー数が  $2^7, 2^8, 2^9, 2^{10}$  の場合である。

### 6.2.5 カットオフユニット

これは、Ewald 法のためのカットオフ関数を評価する。 $r^2$ に適当な係数を掛けたあと、固定小数点に変換しエントリー 5 ビット程度の RAM テーブルと二次補間器によって構成する。なお、このあとで  $1/r$  や  $1/r^3$  とかけ算する必要があるが、これは 12 ビット程度の精度があればよいので、force-1 丸めを行なった後 12 ビットの乗算器に入れる。結果はどうせ 24 ビットなので、そのまま使う。

通常のガウシアン風のカットオフを行なう場合、力の相対精度を 12 ビット程度とするなら、ガウシアンのスケール長 ( $\exp[(r/r_0)^2]$  の形に書いた時の  $r_0$ ) に対して 3 倍程度よりも  $r$  が大きければ力は無視してよいことになる。従って、テーブルは  $4r_0$ まで、実際には  $r^2$  が入力になるのでスケールした後で 16 まで作成する必要がある。

距離が小さい側については、テーブルはもちろん 0 までなにかしら値を出す必要はある。しかし、必要精度は場所によって異なるということに注意する必要がある。具体的には、通常のガウシアンによる力では、力への補正項は以下の形をしている。

$$\alpha_f = \frac{2}{\sqrt{\pi}} r e^{-r^2} 1 - \operatorname{erfc}(r). \quad (6.20)$$

また、ポテンシャルへの補正是  $\alpha_p = \operatorname{erfc} = 1 - \operatorname{erf}$  である。従って、それぞれの高次導関数は以下の形になる。

$$\begin{aligned} \alpha_f^{(1)} &= -2Cr^2 e^{-r^2} \\ \alpha_f^{(2)} &= -4C(r - r^3)e^{-r^2} \\ \alpha_f^{(2)} &= -4C(1 - 5r^2 + 2r^4)e^{-r^2} \end{aligned} \quad (6.21)$$

$$\begin{aligned} \alpha_p^{(1)} &= -Ce^{-r^2} \\ \alpha_p^{(2)} &= 2Cre^{-r^2} \\ \alpha_p^{(3)} &= 2C(1 - 2r^2)e^{-r^2} \end{aligned} \quad (6.22)$$

これらについて、区間  $[0, 4]$  での関数自身と導関数のグラフを図 6.15 に示す。どちらも、 $r = 4$  で  $10^{-7}$  程度と実際に無視できるところまで小さくなる。これらの図からわかるることは、高次補間をした場合は誤差項が  $r$  の大きいところまで 0 にならない傾向がでるということである。例えばポテンシャルの場合、単なるテーブルルックアップであれば誤差項、すなわち 1 次の項はガウシアンであり、1 を超えれば急速に落ちる。しかし、高次になるに従い、複雑な多項式がつくため  $r$  の大きいところまで誤差項が小さくならなくなる。

このことをより明確に示したのが図 6.16 である。ここでは、区分多項式による補間を行なった時の、区間内の誤差が一定になるような相対的なステップサイズ（すなわち、 $k$  次導関数について  $k$  乗根の逆数）をプロットしてみた。次数を上げるに従い、区間幅の  $r$  への依存性はより複雑なものになるが、 $r$  が大きいところでそれほど区間幅を大きく出来ないことがわかる。

RAM テーブルのサイズと補間次数の関係はどうなるであろうか？いま、要求する絶対精度が 12 ビット程度であるとする。カットオフについては相対精度を要求することは物理的に

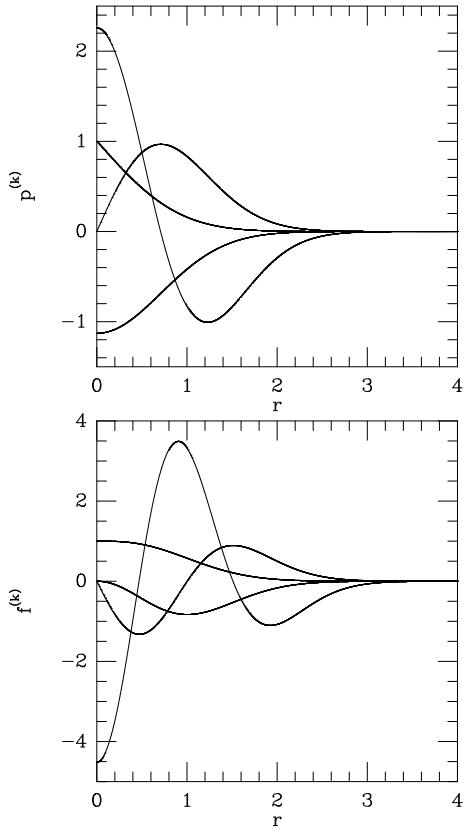


図 6.15: ポテンシャル（左）と力（右）の、ガウシアンカットオフの場合の補正項とその3次までの高次導関数。いずれも、極値、ゼロ点の多いものほど高次である。

意味がないので、問題になるのは絶対精度である。0, 1, 2 次補間について、区間内最大誤差はおおよそ以下の程度である

$$\begin{aligned} e_0 &= \frac{h}{2} f' \\ e_1 &= \frac{h^2}{16} f^{(2)} \\ e_2 &= \frac{\sqrt{3}h^3}{216} f^{(3)} \end{aligned} \tag{6.23}$$

本当は  $e_2$  はもう少し小さく出来るが、大勢に影響はない。力の場合にこれから最大誤差が11ビットになるように区間幅を決めたとすれば、0 次の場合に  $1/1024$ 、1 次で  $1/16$ 、2 次で  $1/8$  となる。すなわち、0 次は現実的とはいい難い。1 次補間で 64 エントリ、2 次補間では 32 エントリとなる。2 次補間にするとエントリ数は半分になるが、テーブルの幅は増えるし、また補間器の構成も大幅に複雑になる。したがって、ここは1次補間で済ますことにする。この場合、例えば [2, 3] で区間幅を 2 倍、[3, 4] で 8 倍程度にしても誤差に影響はない。従って、

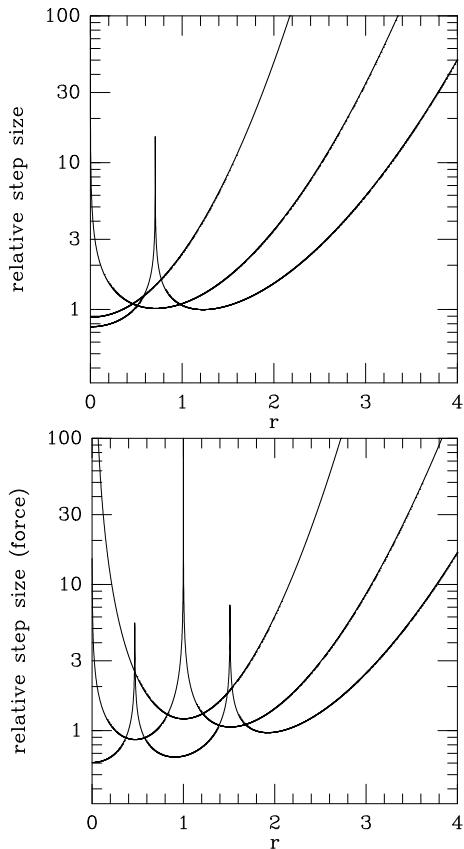


図 6.16: ポテンシャル（左）と力（右）の、ガウシアンカットオフの場合の誤差一定にするための補間区間の相対長を 2 次補間の場合に示す。いずれも、極値、ゼロ点の多いものほど高次である。

エントリ一数を 50 程度まで減らすこともできないわけではない。もちろん、これが現実的であるかどうかは、どのような RAM ブロックライブラリが使えるかによる。

なお、ここまででは補間のための独立変数として  $r$  が使えると仮定したが、現実には  $r^2$  の他には  $1/r$  等しか求まっていない。もっとも単純な方法は、ここでは、単に  $r^2$  に  $1/r$  を掛けて  $r$  を作ることであるが、計算精度がたいしていらないので、テーブルと補間にによって  $r^2$  から  $r$  を生成するほうが、ゲート数、パイプライン段数の観点からは有利であろう。また、そもそも  $m/r$  しか求まらないという構成を考えているので、これも問題ではある。したがって、一応今回は  $r^2$  から作るということにする。すなわち、実際の変換は以下の手順をとることになる。

- (1) まず、 $r^2$  から  $r$  を作る。これは仮数 12 ビットに詰めてから行なう。
- (2) 次にその結果にスケールファクター  $1/r_0$  を掛ける。これは全粒子、パイプラインで共通とする。
- (3) 次に、これを固定小数点化する。要するに 0 から 1 までの値にするだけで、それよりも大きい場合は補間して出来る最大値の値をとるものとする。

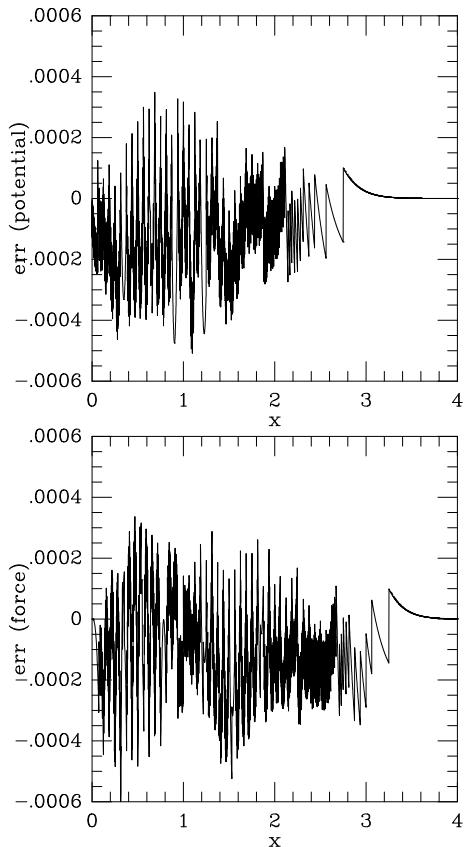


図 6.17: テーブルエントリ 64、語長 12 ビットの場合のポテンシャル（左）と力（右）のカットオフ関数の絶対誤差

- (4) 固定小数点化した内の上位 6 ビットをテーブルに入れて、0 次と 1 次の項を出す。0 次は固定小数点 13 ビット、1 次は 12 ビットで出す。1 次の項は、距離の下位 6 ビットと掛け、上 12 ビットをとって 0 次の項に加える。なお、1 次の項には符号をつける。
- (5) 出た答を浮動小数点に直し、 $1/r$  や  $1/r^3$  と掛け合わせて必要なものをにする。ここは、結果の精度は 24 ビットを維持する必要があることに注意してほしい。

図 6.17 に、カットオフ関数評価器での絶対誤差（但し、カットオフ関数の最大値は 1）を、ガウス型カットオフの場合について示す。誤差が 1LSB を超えるところはほとんどなく、実用上十分以上の精度が得られていることがわかる。このために必要な回路の内、最大なもののは 64 エントリ 20 ビットの RAM であり、まあ無視してよい大きさである。

### 6.2.6 質量とのかけ算

浮動小数点乗算器である。丸めは force-1。

表 6.6: 積算器のステータスフラグ

名称	ビット位置	説明
SOVFL	0	シフタでのオーバーフロー
AOVFL	1	積算器でのオーバーフロー
SUNFL	2	シフタでのアンダーフロー

### 6.2.7 ポテンシャルの積算

前述したように、ここは 64 ビット固定小数点で積算する。指数のオフセット値はレジスタに記憶され、それと入力の指数の差だけシフトしてから加算する。このシフトの際にも、force-1 丸めを行なうものとする。

なお、ステータスピットをつけてオーバーフローを検出できるようにする。これは、表 6.6 に示す 3 ビットのフラグとなる。

これらのフラグは以下の時にセットされるものとする。

- SOVFL: シフトした結果の絶対値が 63 ビット（ポテンシャル、加速度の場合）または 31 ビット（jerk の場合）に収まらない時。この時、シフトした結果は強制的に 0 にされる。
- AOVFL: 加算器がオーバーフローした、すなわち、キャリが符号まで伝わった時にセットされる。
- SUNFL: シフトした結果の絶対値が 0 になる時。

これらのフラグと、積算レジスタ自体の値は、計算開始 (RUN 信号の立ち上がり) でクリアされ、立ち下がったあとはホールドされる。

### 6.2.8 $dx(k)$ との乗算

浮動小数点乗算器である。丸めは force-1。

### 6.2.9 加速度の積算

ポテンシャルの積算と同じ。

### 6.2.10 粒子一つからの力の誤差について

図 6.18 に 2 粒子間の力の相対誤差を距離の関数としてプロットしたものを示す。まず、左、すなわち入力座標のオフセットが 24 ビットのものに注意してほしい。この場合、まず、 $r \sim 0.1$  のより小さいところで誤差が距離に反比例している。これは、入力座標が桁落ちするためである。

さらに、 $r$  が大きい方では、4 本の曲線がわかっているが、どれも  $r^2$  に比例して増えていることがわかる。これは、最終段で固定小数点に変換するときに桁落ちするためである。曲線の間の違いは最終段加算器でのシフト量であり、実線、短破線、長破線、一点鎖線の順に

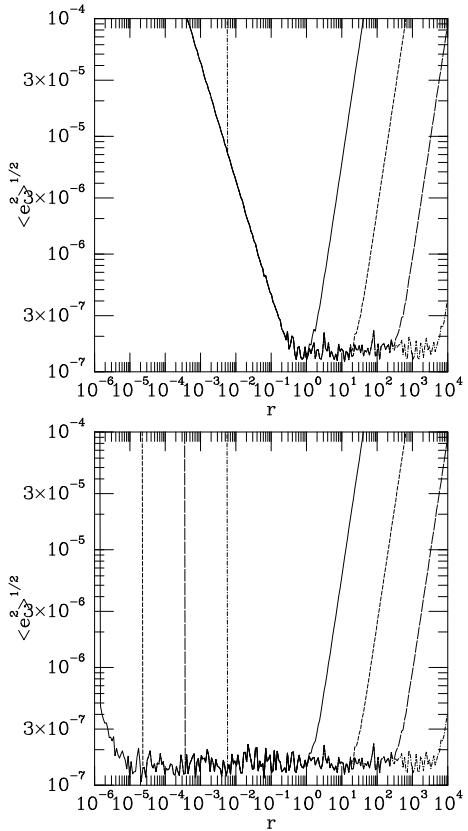


図 6.18: ペアワイズの力の RMS 誤差を距離の関数として示す。サンプル数は各点あたり 1000 個であり、距離は 1.1 倍間隔である。左は入力座標のオフセットが 24 ビット、右は 40 ビットである。それぞれ、最終段の加算器のバイアスをいくつか変えてプロットした。

左シフト量を 8 ビットずつ大きくしている。従って、曲線 1 本ごとに 16 倍分左にずれている（それだけ小さい力まで桁落ちしないで扱える）ことがわかる。

ここで注目するべきなのは、一点鎖線は  $r$  が小さいところで突然誤差が大きくなっていることである。これは、 $r$  が大きいところとは逆に最終段でオーバーフローがおきて、答が完全に破綻しているためである。

この、オーバーフローと桁落ち（アンダーフロー）の兼ね合いは、入力座標のオフセットが 40 ビットの右図でよりはっきりとわかる。こちらでは、シフト量が非常に小さいもの（実線）を除いて、曲線の形は同じであり左右にシフトしているだけである。すなわち、入力オフセットと出力シフトを固定したとき、 $r$  でほぼ  $10^6$  の範囲で相対誤差が一定にたもたれることになる。

なお、実際に使う時には、入力オフセットは全粒子に共通であるが出力シフトは粒子ごとに変えることができることに注意してほしい。これにより、実際に大きな加速度を受けている粒子でもそうでないものでも、桁落ちを起こさずに計算できることになる。

**6.2.11  $dv(K) = XDOT(K,J) - XDOT(K,I)$** 

仮数 20 ビットでの減算。指数は 9 ビットに拡張。

**6.2.12  $dx(k)*dv(k)$** 

仮数 20 ビットでの乗算。 $dx$  はあらかじめ force-1 で丸める。

**6.2.13  $drdv = drdv + \dots$** 

仮数 20 ビットでの加算。

**6.2.14  $j1 = dx(k)*mdr5i$** 

仮数 20 ビットでの乗算。 $dx$  はあらかじめ force-1 で丸める。

**6.2.15  $j2 = dv(K) *dr3i$** 

仮数 20 ビットでの乗算。 $dr3i$  はあらかじめ force-1 で丸める。

**6.2.16  $j1 + j2$** 

仮数 20 ビットでの加算。

**6.2.17  $FD(k) +=$** 

固定小数点 32 ビットに拡張してから加算。

なお、ここもステータスビットをつけてオーバーフローを検出できるようにする。

この語長が何ビットいるかはちょっと難しい問題である。力の相対精度が 24 ビット、ジャークが 20 ビットとしている。例えば、タイムステップが最近接粒子で決まっているとすれば、ジャーク自体ほぼその程度の大きさである。一般に、 $1/r^3$  で落ちるので近傍で決まるることは確かである。最近接粒子の  $k$  倍の距離にある粒子からの寄与は  $1/k^3$  程度であるが、そういう粒子は（距離を対数で切れば） $k^3$  個あるのでジャークへの寄与は  $1/k^{1.5}$  でしか落ちないことになる。とすれば、例えば  $10^6$  個の粒子の場合、系のサイズ程度の距離にある粒子の寄与もまだ  $1/10^3$  程度に効くので無視できない。一般に、20 ビットの精度を得ようと思えば、 $k = 1000$  当たりまですべて計算する必要があることになる。

$k = 1000$  まで計算するのに必要なアキュムレータの語長は 32 ビットである。したがって、32 ビットあれば最低限必要な精度は得られるはずである。

なお、上の考察は若干奇妙なところがある。というのは、実際には  $k$  が大きいところの精度については、相対的なステップサイズから議論する必要があるからである。すなわち、力については、丸め誤差は  $f \cdot \Delta t$  に対して相対的に入ってくるのに対し、ジャークには  $j \cdot \Delta t^2$

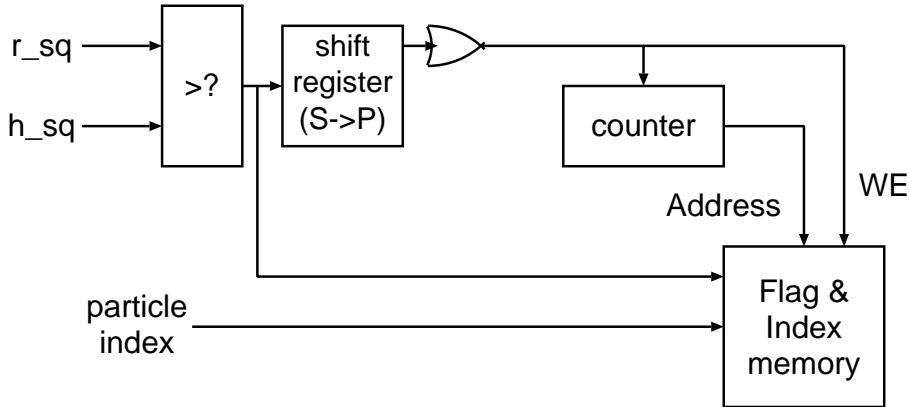


図 6.19: ネイバーリスト

に乗る形になる。2体の相対距離に対する $\Delta t$ は $k$ に反比例し、 $j/f$ も同様なので、結局 $k^2$ に反比例してジャークの相対誤差は大きくなっている。

この考察からも、20ビットの精度を最近接粒子に対して得るとすれば、 $k = 1000$ まで計算出来ればよく、語長は32ビットでいいということがわかる。

### 6.2.18 ネイバーリスト

これは、GRAPE-4風のネイバーリストをチップ内に集積する。VMP毎に判定距離を変える。これもオーバーフローフラグをつける。実パイプライン一本あたり256ワード程度で十分であろう。GRAPE-4と同様、仮想パイプ16本程度で1つのリストをシェアする。従って、ネイバーリスト1ユニット当たりのメモリ量は $512 \times 48 = 24\text{kbits}$ となる。これはまあ何とかなる範囲であろう。入り切らなければ半分程度にすることも考慮する。

GRAPE-4では同じリストをシェアする仮想パイプの数は32本であったが、GRAPE-6ではまだパイプライン本数が確定していないので少し少なめにする。16本の場合、2本の物理パイプがリストをシェアし、8サイクルの間シフトレジスタでデータをためたあと必要ならばメモリに書き込むという回路になる。なお、粒子インデックスが入力としてメモリからくるので、これをそのままネイバーリストメモリにしまう。これにより、GRAPE-4のソフトウェア開発上で大きな問題になった、「ネイバーリストの解釈が非常に複雑である」という問題を解消する。さらに、自分自身であれば、距離は小さいに決まっているが、これは無駄な情報なので省く。これは、メモリから来る粒子インデックスと自分の値を比較することで実現出来る。

図6.19に概念図を示す。ここでは物理パイプライン1本分の回路になっているが、実際には2個の比較器とシフトレジスタの出力のORがとられることになる。また、 $h^2$ は仮想パイプライン毎に違う値をとるので、ここはリングレジスタまたはレジスタファイルで実現される必要がある。

メモリを読む時には、偶数アドレスのときオーバーフローフラグ + 16bit のネイバーフラグの合計17ビット、奇数アドレスのときに32ビットの粒子インデックスが見えるようになる。

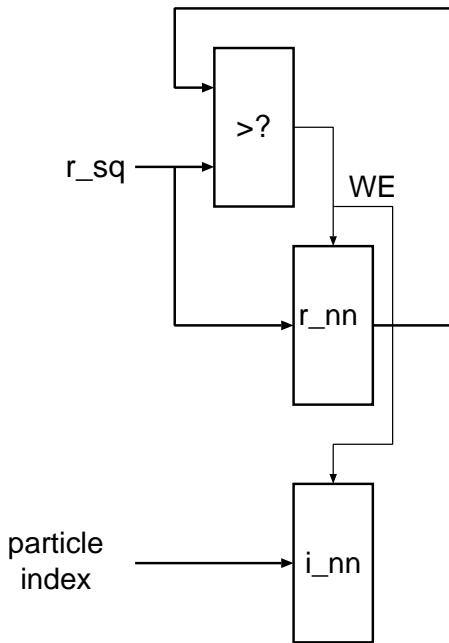


図 6.20: 最近接粒子

### 6.2.19 ニアレストネイバー

GRAPE-4 になかった新しい試みとして、ニアレストネイバーのレジスタをつける。これは、計算中にもっとも近くにあった粒子とその番号を記憶する。結果の回収時にも、reduction tree で最小値が選ばれるような仕掛けをつける。

これは、比較的簡単な回路で実現でき、また多くの計算で有用であることがわかっている。実際、ネイバーリストを使った多くの計算において、実際に必要なのはニアレストネイバーだけである。

図 6.20 に概念図を示す。なお、ここでも  $r_{nn}$  よび  $i_{nn}$  は実際には仮想パイプライン分のレジスタファイルまたはリングレジスタである必要があることに注意する必要がある。

なお、例えば微惑星系やリングの計算では、質点として求めた（あるいはソフトニングが入った）距離よりも、それから 2 つの粒子の半径を引いたもののほうがより重要である場合もある。従って、そのような、粒子の半径を表すパラメータをつけるということも考えられなくもないが、面倒なのでつけない。これが必要になるのは、粒子の半径にかなり幅があり、さらに平均粒子間距離が非常に小さい場合である。そのような場合にはいずれにしても複雑な扱いが必要になるので、ニアレストネイバーだけでは済まない。

ニアレストネイバーだけで済むのは、粒子間距離が小さく、3 個以上の同時衝突が十分に稀な場合だけである。

### 6.2.20 実際のパイプライン構成

前節までの考察で、演算順序等について曖昧さがあったところを一応確定したので、どのような演算ユニットがあるかはきまつたことになる。それを以下にしめす。なお、図6.9 はすでにこの構成に対応したものになっていることに注意してほしい。

- a)  $dx(K) = X(K,J) - X(K,I)$  入力 64 ビット固定、出力仮数 24 ビット浮動
- b)  $dx2(K) = dx(K)*dx(K)$  入出力仮数 24 ビット浮動
- c0)  $dr2xy = dx2(1) + dx2(2)$  入出力仮数 24 ビット浮動
- c1)  $dr2ze = dx2(3) + \text{eps}2$  入出力仮数 24 ビット浮動
- c2)  $dr2 = dr2xy + dr2ze$  入出力仮数 24 ビット浮動
- d)  $r5inv = dr2^{**}2.5$  入出力仮数 24 ビット浮動
- e)  $mr5inv = r5inv*m$  入出力仮数 24 ビット浮動
- f)  $mr3inv = mr5inv*dr2$  入出力仮数 24 ビット浮動
- g)  $mrinv = mr3inv*dr2$  入出力仮数 24 ビット浮動
- h)  $r = \sqrt{dr2}$  入力仮数 24 ビット浮動、出力仮数 12 ビット浮動
- i0)  $pcut = pf(r)$  入出力仮数 12 ビット浮動
- i1)  $fcut = ff(r)$  入出力仮数 12 ビット浮動
- j0)  $mrinv2 = mrinv*pcut$  入出力仮数 24 ビット浮動
- j1)  $mr3inv2 = mr3inv*fcut$  入出力仮数 24 ビット浮動
- k)  $\phi = \phi + mrinv2$  累算器 32 ビット固定
- l)  $f(k) = mr3inv*dx(k)$  入出力仮数 24 ビット浮動
- m)  $acc(k) = acc(k) + f(k)$  累算器 64 ビット固定
- n)  $dv(K) = XDOT(K,J) - XDOT(K,I)$  入力仮数 24 ビット浮動、出力仮数 20 ビット浮動
- o)  $xv(k) = dx(k)*dv(k)$  入出力仮数 20 ビット浮動
- p)  $drdvxy = xv(1) + xv(2)$  入出力仮数 20 ビット浮動
- q)  $drdv = drdvxy + xv(3)$  入出力仮数 20 ビット浮動
- r)  $drdv3 = drdvxy*3$  入出力仮数 20 ビット浮動
- s)  $mdrdv3byr5 = drdv3 * mr5inv$  入出力仮数 20 ビット浮動
- t)  $t1(k) = dv(k)*mdr3inv$  入出力仮数 20 ビット浮動
- u)  $t2(k) = dx(k)*mdrdv3byr5$  入出力仮数 20 ビット浮動
- v)  $j(k) = t1(k) + t2(k)$  入出力仮数 20 ビット浮動
- w)  $jerk(k) = jerk(k) + j$  累算器 32 ビット固定
- x) ネイバーリスト
- y) ニアレストネイバー

### 6.2.21 固定小数点表示にする時の指数の扱いについて

GRAPE-6 相互作用パイプラインでは、位置は固定小数点、速度は浮動小数点で与えられる。また質量は浮動小数点で与えられる。結果はすべて固定小数点であり、最終段での変換をうまくやらないと破綻が起きるようになっている。以下に、どのようなアルゴリズムで変換が行なわれるかについてまとめた後、最終段でのバイアスの設定について述べる。

#### 加速度とポテンシャル

まず、位置は固定小数点で入ってくる。この分解能を

$$\epsilon_x = P_2(-k_x) \quad (6.24)$$

と表すこととする。ここで、 $P_2(x) = 2^x$ である。引き算の結果は浮動小数点に変換される。この変換は、指数が引き算の結果の有効桁数に 512 を足したものになるという形になる。

浮動小数点形式での乗算では、指数バイアスは 513 である。つまり、指数が  $513 = 0x201$  で仮数が “10000...0” の数が 1 であり、これの 2 乗は値が変わらない。

ここで、入力座標として 1 であった数がどう表現されることになるかを考えてみると、まず、固定小数点に変換された後は 1 の後に 0 が  $k_x$  個並ぶことになる。従って、浮動小数点に変換された後の指数は、 $512 + k_x + 1$  であり、内部表現はちょうど  $P_2(k_x)$  だけ大きく評価するということになる。従って、ポテンシャルについては  $P_2(k_x)$ 、加速度については  $P_2(2k_x)$  だけ小さく評価しているということになるわけである。

さて、ポテンシャルと加速度はそれぞれ  $k_p, k_f$  なる指数オフセットを加算したあとで固定小数点化される。加算した後の指数の値を  $e$  とすると、これが変換後の絶対値の桁数になる。 $e$  の計算は 10 ビットで行なうが、結果は下位 6 ビットだけを見ることになる。

仮に、質量が 1 であり距離も 1 の場合を考える。求まる結果は加速度もポテンシャルも 1 である。これを、分解能

$$\epsilon_f = P_2(-k_o) \quad (6.25)$$

で表現したいとする。すると、固定小数点に変換後の数が、1 の後にゼロが  $k_o$  個並ぶものということになる。従って、入力の力／ポテンシャルの指数をそれぞれ  $a = 513 - 2k_x, b = 513 - k_x$  とすると、 $k_f, k_p$  と  $k_o$  の関係は次式で与えられる。

$$k_o = a + k_f \quad (6.26)$$

$$k_o = b + k_p \quad (6.27)$$

これらから  $k_f, k_p$  を定めると

$$k_f = k_o - 513 + 2k_x \quad (6.28)$$

$$k_p = k_o - 513 + k_x \quad (6.29)$$

ということになる。

さて、一般の場合、すなわち力が 1 の程度ではない場合にどうするべきかということであるが、この時は最終的に積算されて求まった力がオーバーフローを起こさない範囲でなるべく精度よく求まっている必要がある。実際の力が  $2^p$  の程度であったとする。これを  $q$  ビット使って求めるとすれば、先ほどの  $k_o$  に対して  $k_o = p - q$  となるので、これを使って  $k_f, k_p$

を定めればいいことになる。逆に、 $k_f, k_p$  が与えられていれば、上式から結果の固定小数点表現をホスト計算機での浮動小数点表現に変換するやりかたが定まることになる。

さて、ちょっと気になるのは、例えば  $10^7$  粒子といった計算でもちゃんと桁落ちしないで力がもとまっているかどうかである。典型的な粒子であれば、システムサイズ位の距離の粒子からの力は全加速度の  $10^{-7}$  ということになり、桁落ちは 23 ビット足らずである。従って、それほど問題にはならない。しかし、例えば singular isothermal な粒子分布を考えると、半径に反比例して加速度が大きくなるので、例えば  $10^{-5}$  といったところを考えるとさらに 16 ビット桁落ちし、half-massあたりからの力より先は精度がおちることになる。もっとも、最終的な力の相対精度という観点からは、この最終段での桁落ちによるものは  $2^{-q}\sqrt{N}$  の程度にしかなり得ないので、原理的に他の項に比べてはるかに小さくなっている。

### 加速度の時間導関数

加速度の時間導関数 jerk についても、基本的な考え方は同じである。まず、入力についてであるが、これは予測子の出力である。予測子で速度に与える指数は、 $v\Delta t$  を固定小数点形式に変換した後で  $x$  に足せるように選ばれている。これを具体的に書くと以下のようになる。

まず、実際に位置に足し込むところで C のプログラムでしていることは、まず仮数部を 64 ビット符号なし整数に左詰めで入れ、それから速度の指数（7 ビット）をみてそれから 63 を引いたものが変換後の桁数になるという操作である。つまり、出てきた指数を  $k_v$  とすれば、左詰めで入ってきた数に対して  $127 - k_v$ だけ右論理シフト（正しく丸めつつ）をすることに対応する。今、速度が 1 でありタイムステップが  $\Delta t = 2^{-l}$  であるとすれば、 $v\Delta t = \Delta t$  となるので、前節で定義した  $k_x$  を使えば

$$k_v - 63 = k_x - l \quad (6.30)$$

となることがわかる。位置に加算する時にはこの  $k_v - 63$  をシフタの入力にすることになる。速度の出力の方は、まず  $\Delta t$  が掛かっていてはいけないということに注意してほしい。これは、 $\Delta t$  の値は粒子毎に違うので、このままのかたちでは引き算できないからである。従って、パイプラインの出力を  $\Delta t$  で割ってやる、言い替えれば指数に  $l$  を加えてやる必要がある。

実際には、 $l$  そのものはチップに入力されていない。これは、チップに入力されているものは、時間分解能を

$$\epsilon_t = P_2(-k_t) \quad (6.31)$$

と書いた時に

$$s = k_t - l \quad (6.32)$$

だからである。とりあえずこの値を引いておくと、 $v$  に掛かる係数が全粒子で共通にはなる。

さらに、 $s$  を引いた答に 512 を足して、予測子の出力ということにする。結局、速度が 1 の時の予測子の出力の指数は

$$u = k_v - s + 512 = k_x - k_t + 575 \quad (6.33)$$

ということになる。先に述べたようにバイアスが 513 なので、結局

$$u' = k_x - k_t + 62 \quad (6.34)$$

だけ指数がずれていることになる。

さて、jerk は相対速度の 1 乗、相対位置の -3 乗なので、浮動小数点形式で求まった jerk は、結局

$$c = u' - 3k_x = -2k_x - k_t + 62 \quad (6.35)$$

だけ指数がずれていることになることがわかるであろう。さて、今 jerk の値が 1 である時にこれを  $k'_o$  ビット、すなわち

$$\epsilon_f = P_2(-k'_o) \quad (6.36)$$

の相対精度で表現したいとしよう。以下の議論は加速度の場合とまったく同様であり、

$$k_j = k'_o - 513 - c = k'_o + 2k_x + k_t - 575 \quad (6.37)$$

が指数に加算するべき値ということになる。一般の場合には、とにかく固定小数点で 1 以下に使いたいビット数として  $k'_o$  を使えばいいというのも加速度の場合と同様である。

なお、ここでの指数に関する議論はすべてホスト計算機の上での扱いに関するものであり、チップの設計とは特に直接関係するものではない。

### アプリケーションプログラムでの扱い

固定小数点にする上でちょっといやなのは、アプリケーションプログラム側でそれを意識する必要があることである。G6 Chip の場合、具体的には以下のようないものを設定する必要がある：

1. `xunit`
2. `tunit`
3. `fyscale`
4. `jyscale`
5. `pyscale`

これらに対する対応であるが、まず最初の 2 つはよほど変な使い方をしない限りグローバルである。また、実際上変更の必要は少ないのである。つまり、システムの大きさ、時間スケールを適当に想定しておいて、最初に設定すればいい。

問題は、後の 3 個である。これについては、実際 2 つの非常に異なる使い方をすると考えられる。一つは、ツリーコードなどと比較的に大きなソフトニングを併用する場合である。あるいは、ツリーコードでなくてもとにかくソフトニングを使う場合は大部分当てはまるかもしれない。この時は、力とその導関数の最大値は、最近接粒子からのものに上限があることではほぼ定まる。システムサイズが 1 程度の単位系で考える時に、ソフトニング  $\epsilon$  の現実的な値は  $10^{-5}$  程度であろう。この時、典型的な力に対して、最大発生し得る力は一般に  $1/(\epsilon^2 N)$  の程度 ( $G = M = 1$  として) であり、 $1/\epsilon < N$  とすれば (これは現実的な仮定といって良い) 力は  $10^5$  をこえないとみなせる。したがって、1 のオーダーの力が 48 ビット程度で表現されていればほとんど常に桁落ち、桁溢れともに起きることはない。

もちろん、桁溢れ、桁落ちが起きたら対応する必要はある。が、プログラムの中で自動処理が必要ということもないと思われる。

さて、これに対し、ソフトニングを使わない場合、あるいは使っても小さい場合は粒子毎にスケールを変える必要が生じる。したがって、この 2 つの場合で若干異なる API を持つのが望ましいといえよう。

以下に、まず、粒子毎に与える場合の API の概要をまとめる。

### ソフトウェアエミュレータの API

とりあえずまだ現物のチップやボードはないので、エミュレータの方で考える。

まず、グローバルな定数を設定する関数がいろいろ出てくる：

```
void set_tunit(int newtunit);
    予測子の時刻スケールの設定
void set_xunit(int newxunit);
    位置のスケールの設定
void set_ti(double ti);
    予測子の時刻の設定
void set_njp(int n);
    j粒子の数の設定
```

次に、*j*粒子の設定：

```
ULONG set_j_particle_on_emulator(int address,
int index,
double ti, /* present time */
double tj, /* particle time */
double dtj, /* particle time */
double mass,
double a2by18[3], /* a2dot divided by 18 */
double a1by6[3], /* a1dot divided by 6 */
double aby2[3], /* a divided by 2 */
double v[3], /* velocity */
double x[3] /* position */);
```

ここで、*address* はメモリアドレスであり、*index* はメモリに書かれる（パイプラインに渡される）インデックスである。通常の使い方では同じで、*j*粒子のインデックスになりそれがボードメモリ上での位置にもなる。それ以外は旧来の HARP-2/3 のインターフェースと大差なかろう (A2DOT がついていることに注意)。

次に、*i*粒子の設定、これが問題である。独立時間刻みに対応し、粒子毎にスケールを持たせるとして、例えば以下のようないインターフェースを持たせたとしよう。

```
ULONG set_i_particle_xv_on_emulator(int address,
double x[3], /* position */
double v[3] /* velocity */);

ULONG set_i_particle_parms_on_emulator(int address,
ULONG eps2,
ULONG h2,
ULONG rscale,
LONG index);
```

```

ULONG set_i_particle_scales_on_emulator(int address,
    LONG fscale,
    LONG jscale,
    LONG phiscale);

void get_i_particle_parms_on_emulator(int address,
    LONG * fscale,
    LONG * jscale,
    LONG * phiscale);

```

これらは、ライブラリ内のメモリに*i*粒子をコピーするだけで、実際にチップメモリ（のエミュレーション部）にデータを渡すわけではない。このようになっていた時、仮にオーバーフロー等が起きた時にはアプリケーションの方で適当にスケールを設定しなおすことになる。

この方法にはいくつか問題がある。まず、上に述べたように、`fscale`などは、実際にチップに設定される値は `xunit` などに依存した複雑な値となる。これは面倒なので、ここで設定する値は、「加速度等の指數の大きさ」にしたい。しかし、実は、パラメータがもう一つあるのでちょっとややこしい。つまり、「結果の数を何ビットくらいで表現したいか」という自由度があるからである。

しょうがないので、インターフェースルーチンには、「大きさが 1 の程度であった時に何ビットで表現するか」を渡すことにする。しかし、これもまだアプリケーションが面倒を見るにはかったるので、さらに簡単にするために、加速度、ジャーク、ポテンシャルなどを直接渡せばそれからスケールを設定する以下のようなインターフェースを与える。これにより、すでに一度力が計算された粒子についてはオーバーフロウ、アンダーフロウは起きなくなると考えられる。

```

set_i_particle_scales_on_emulator_from_real_value(int address,
    double acc[3],
    double jerk[3],
    double phi);

```

これをチップレジスタに書くときの変換はライブラリが面倒を見る。さらに、`eps2`、`h2`、`rscale` などは GRAPE-6 の内部フォーマットではなく実数を与える。これについても変換はライブラリが面倒を見る。つまり、アプリケーションは `xunit` などを意識しない。

さて、独立時間刻みの場合、ライブラリに送った*i*粒子のなかから粒子を選んでチップ（ボーデ？）に送り、それから力を計算させるということになる。これを行なうのが次の関数である。これらは `gchip` 構造体を引数にとるので、スタティックにそれらを確保したものを Fortran インターフェースには準備する。

```

int move_i_particles_to_chip(int nparticles,
    int * index,
    struct gchip *chip);
void run_chip(struct gchip *chip );

```

結果回収は以下のようになる:

表 6.7: 相互作用パイプラインのテストモード

名称	演算	モード内容	備考
T01	$dx(k) * mr3inv$	$mr3inv$ to 1	
T02	$dr2 * mr5inv$	$mr5inv$ to 1	
T03	$dr2 * mr5inv$	$dr2$ to 1	
T04	$mr3inv * dx(k)$	$dx(k)$ to 1	
T05	$dx(k) * dx(k)$	pass through	
T06	$mrinv * pcut$	$mrinv$ to 1	
T07	$mr3inv * fcut$	$mr3inv$ to 1	
T08	$r5inv * m$	$r5inv$ to 1	
T11	$mr3inv * dr2$	$dr2$ to 1	
T12	$drdv3 * mr5inv$	$mr5inv$ to 1	
T13	$t1 + t2$	$t1$ to 0	
T14	$drdv * 3$	pass through	
T15	$drdv3 * mr5inv$	$drdv3$ to 1	

```
void get_force(int pipeid,
    struct gchip *chip,
    double acc[3],
    double jerk[3],
    double *phi,
    int *aflag,
    int *jflag,
    int *pflag);
```

これでオーバーフローフラグなどは戻って来るので、具合が悪ければ設定し直すし、そうでなければこれで済ませられる。

### 6.2.22 テスト用制御信号

システムを組んでからアドホックなテストができるようにするため、各演算器の出力がおむねそのまま見えるような仕掛けをつけることにする。最終段で固定小数点にするので指数がそのまま見えるというわけにはいかないが、仮数は見えるし、指数オフセットをとってから見るから間接的には指数もみえるわけなのでこれは大きな問題ではない。

実装するテストモードを表 6.7 に示す。

$$a) dx(K) = X(K,J) - X(K,I)$$

入力は完全に任意に設定できるので、出力をみることが出来ればよい。出力をそのままみることは、その次のシフタで切捨てるために出来ないが、24ビットまではみえるので、まああまり問題はないであろう。

この減算器の出力が直接アキュムレータに現れるためには、 $dx(k)$  と  $mr3inv$  の乗算で、 $mr3inv$  のほうを強制的に 1 にできればいい。T01

$$b) dx2(K) = dx(K)*dx(K)$$

ここは 2 乗なので、入力が完全に自由に制御できるわけではないが、それは実際に使う時もそうなのであまり問題ではない。本来はこの性質を利用して回路規模を小さくできるはずであるが。2 乗器としてみる限り、入力は完全に自由に制御できる。

出力がアキュムレータに現れるためには、たとえば  $x$  を見る時に  $y, z$  は入力データ自体を 0 にするとすれば、 $r^2$  と  $m/r^5$  を掛けるところで  $m/r^5$  を 1 にし、さらに  $m/r^3$  と  $dx$  を掛けるところで  $dx$  を 1 にすればいい。T02, T04

$$c0) dr2xy = dx2(1) + dx2(2)$$

ここは加算器である。入力を制御するには、まず  $dx(k)**2$  のところで素通しにする回路がいる。あとは上の  $dx(k)$  の乗算と同じ制御で結果を見れる。T02, T04, T05

$$c1) dr2ze = dx2(3) + eps2$$

(c0) と同様。

$$c2) dr2 = dr2xy + dr2ze$$

(c1) と同様。

$$d) r5inv = dr2**2.5$$

ここもは上の加算器と同様に、その前の乗算器を素通しにできれば入力は制御できる。出力については、 $r^2$  を掛けるところ、その後に  $dx(k)$  を掛けるところを制御する必要がある。T05, T03, T04

$$e) mr5inv = r5inv*m$$

質量は自由に制御できる。 $1/r^5$  は任意入力が作れるわけではない。これについてはパイプラインレジスタを介して  $r^2$  をそのまま出すようにする必要がある。これはちょっと大がかりになりすぎるような気もするので、検討が必要である。出力については、(d) と同様でよい。T03, T04, T05

**f)  $\text{mr3inv} = \text{mr5inv} * \text{dr2}$** 

$\text{mr5inv}$  については、前の  $m$  と  $r5inv$  の乗算のところから  $m$  がでてくれれば、あとは  $\text{dr2}$  は  $\text{eps2}$  にすればよいので入力は制御できる。出力には、 $\text{dx}(k)$  のほうを 1 にするようすればいいであろう。T08, T04

**g)  $\text{mrinv} = \text{mr3inv} * \text{dr2}$** 

これはちょっとややこしい。 $\text{mr3inv}$  に  $m$  を通すとすれば、 $\text{dr2}$  の方に例えば  $\text{eps2}$  ができる。このときに、ステージ (f) では  $\text{dr2}$  が 1 にしておく必要があることになる。T08, T03

**h)  $\text{r} = \text{sqrt}(\text{dr2})$** 

入力は  $\text{eps2}$  から任意にできる。出力は、まずテーブルの方で素通しにした上で、 $\text{mrinv}$  とのかけ算で他方を 1 にすればポテンシャルに出ることになる。T06

**i0)  $\text{pcut} = \text{pf}(\text{r})$** 

これはビット落ちがあるので、容易に全入力を制御できる。結果を見るためには、 $\text{mrinv}$  とのかけ算器で 1 を入れる必要がある。T06

**i1)  $\text{fcut} = \text{ff}(\text{r})$** 

これはビット落ちがあるので、容易に全入力を制御できる。結果を見るためには、 $\text{mr3inv}$  とのかけ算器で 1 を入れ、さらに  $\text{dx}(k)$  を掛けるところで 1 をいれる必要がある。T07, T04

**j0)  $\text{mrinv2} = \text{mrinv} * \text{pcut}$** 

$\text{mrinv}$  は  $m$  を出せばいいのでステージ g でも  $\text{dr2}$  を 1 にできればいい。 $\text{pcut}$  のほうは面倒だが、テーブルの中身を変えれば任意の値がだせることになる。出力はそのままポテンシャルの累算器に出る。T08, T03, T11

**j1)  $\text{mr3inv2} = \text{mr3inv} * \text{fcut}$** 

$\text{mr3inv}$  は  $m$  を出せる。 $\text{fcut}$  は  $\text{pcut}$  と同様に制御可能である。結果を見るためには、 $\text{dx}(k)$  を掛けるところで 1 をいれる必要がある。T08, T03, T04

**k)  $\text{phi} = \text{phi} + \text{mrinv2}$** 

$\text{mrinv2}$  がほぼ自由に制御できるので、特に問題はない。

1)  $f(k) = mr3inv2 * dx(k)$

ここまで m を素通しにできるので、問題ない。T08, T03

m)  $acc(k) = acc(k) + f(k)$

ここも特に考える必要はない。

n)  $dv(K) = XDOT(K,J) - XDOT(K,I)$

XDOT は制御できる。出力を見るには、mr3inv が 1 に出来ればよい。これは mr5inv を 1 にすると同時に dr2 も 1 にすることで実現できる。T02, T03

o)  $xv(k) = dx(k) * dv(k)$

$dx$ ,  $dv$  は制御可能である。結果を見るには、例えば  $dv$  の y, z 成分を 0 にして、 $dx$  の y 成分を 1 に、さらにステージ (s) で mr5inv を 1 に、(v) で t1 を 0 にすればいいことになる。さらに (r) を素通しにする。T12, T13, T14

p)  $drdvxy = xv(1) + xv(2)$

上と同様に、 $dvz$  を 0 に、 $dz$  を 1 に、さらに mr5inv を 1 にすればいい。T12, T13, T14

q)  $drdrv = drdrvxy + xv(3)$

x, y のどちらかについて、(p) と同様な操作をすればいい。T12, T13, T14

r)  $drdrv3 = drdrvxy * 3$

$drdrv$  は制御可能であり、(o) での操作に準じる。T12, T13

s)  $mdrv3byr5 = drdrv3 * mr5inv$

$drdrv3$  は制御可能である。mr5inv は m を出せるので制御可能である。ステージ (u) で  $dx(k)$  に 1 を出すのも、 $drdrv3$  とは独立に可能である。後はステージ (v) で t1 を 0 にすればいいであろう。T14, T03, T08, T13

t)  $t1(k) = dv(k) * mdr3inv$

$mdr3inv$  を m に出来るはずなので、制御可能。あとは t2 を 0 にすればいいが、これは  $dx$  を 0 にすればいい。T03, T08

**u)  $t2(k) = dx(k) * mdrdv3byr5$**

ステージ (s) で drdv3 を 1 にする。これで mdrdv3byr5 を m にできる。あと、t1 を 0 にしたいので、dv が 0 になるように入力をつくる。T03, T08, T15

**v)  $j(k) = t1(k) + t2(k)$**

それぞれ、dv と dx が見えればいい。T02, T03, T15, T08

**w)  $jerk(k) = jerk(k) + j$**

ここは dx がみえればいいので、(v) と同じでよい。

**x) ネイバーリスト**

dr2 が制御できればいいので、座標等は 0 にして eps だけを与えればよい。

**y) ニアレストネイバー**

ネイバーリストと同様。

### 6.2.23 チップ内入出力インターフェース

図 6.21 に信号定義を与える。

相互作用パイプラインの入出力は、以下の 4 種に大別できる。

1. 予測子からの入力
2. レジスタ書き込み
3. レジスタ読みだし
4. 動作制御入力

予測子からの入力については、すでに述べた予測子の出力とつじつまが合えばいい。

レジスタへの書き込みについては、どういう風に実装するかが問題だが、その面倒は相互作用パイプラインのほうで面倒をみるとすることにして、制御部のほうからはアドレス、データ、WE 信号ができるということにする。

レジスタへからの読みだしについても、制御部のほうからはアドレスがでて、相互作用パイプラインからは固定したディレイの後にデータが出るということにする。

以下、それぞれについてまとめる。

#### 予測子からの入力

予測子パイプラインからの入力は、位置、速度、質量、インデックスである。ただし、さらにこれに加えて、VD に相当する「データが揃った」という信号を出す。つまり、入力は表 6.8 のようになる。これは表 6.5 と同一であるはずである。

図 6.22 に予測子からの入力のタイムチャートを示す。これは図 6.8 と同じはずである。

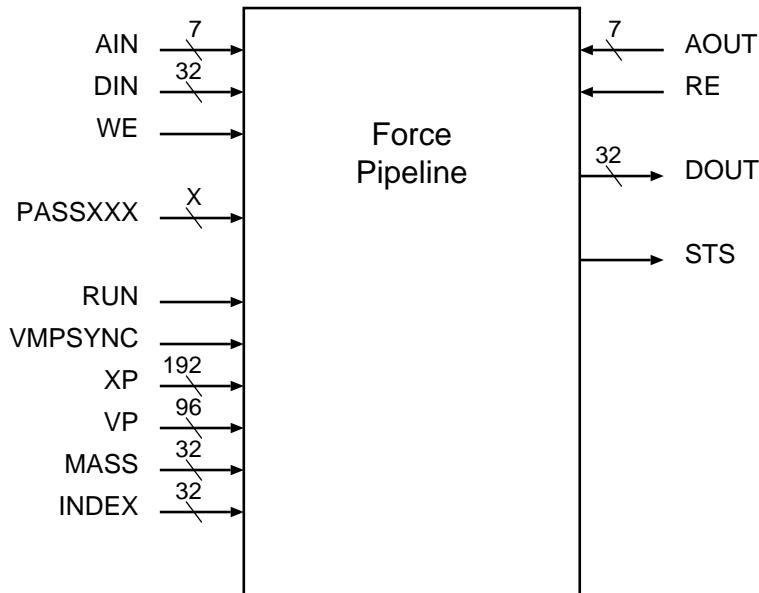


図 6.21: 相互作用パイプライン内部インターフェース

### レジスタ書き込み

すでに述べたように、制御部と相互作用パイプラインとのインターフェースでは、パイプライン内のレジスタについては相互作用パイプライン側でデコードすることにする。従って、インターフェース信号は表 6.9 のようになる。

タイミングは同期型であり、すべての信号をクロック立ち上がりでサンプルする。WE がハイであるときには指定したアドレスへの書き込みが起きる。そうでなければ何も起こらない。なお、WE<sub>i</sub> は *i* 粒子データの、WET<sub>i</sub> はカットオフテーブルの書き込みに使う。これらで WE を別にするのは、論理的に別ものであるからである。つまり、*i* 粒子データは物理パイプライン、仮想パイプラインでそれぞれべつものであるのに対し、テーブルデータは物理的には物理パイプライン毎に違うが、仮想パイプライン間では共有される。また、使い方として物理パイプ毎に変えるというのは考えないので、WE を共有して一度に書いてしまいたい。アドレスも同様に 2 つ準備する。

表 6.8: 予測子パイプラインからの入力

変数	ビット数	個数
位置	64	3
速度	36	3
質量	36	1
番号	32	1
VMPSYNC	1	1

```

clk      _____/====\_____
VMPSYNC  /=====\
DATA    <=data0>-----<=data1>-----

```

図 6.22: 予測子からの入力のタイムチャート

表 6.9: 制御部-相互作用パイプ書き込みインターフェース信号定義

信号	幅	説明
DIN	36	書き込むデータ
AI	7	アドレス
AT	8	アドレス
WEI	1	書き込み信号 (アクティブハイ)
WET	1	書き込み信号 (アクティブハイ)

表 6.2.23 に相互作用パイプライン一つのなかのレジスタマップを示す。上位 3 ビットは仮想パイプラインの ID であり、下位 4 ビットが物理レジスタの区別になる。実装としては、違う仮想パイプの同じデータは物理的には一つのレジスタファイルなりメモリユニットに存在することには注意して欲しい。

ポートのデータ幅は 36 ビットである。64 ビットのデータは 2 語に分けて書かれる。また、LSB 側 32 ビットにデータを埋める。連続してデータがくるとかそういう保証はない。

なお、すこしややこしいが、cutoff テーブルは仮想パイプで共通でいい。さらに、物理パイプ間ではもちろん違うものをもつが、書き込むポートは共通であり、同時に書かれるということにする。このため、cutoff テーブルのアドレスは物理的に別であるということにする。テーブルはポテンシャルと加速度のために 2 つあるので、表の Y でこれを区別する。ただし、R0 は共通であることに注意すること。

CTAB は、1 ワードにデータ (0 次、1 次、1 次の符号) をパックする。ビット位置はそれぞれ (25,13), (12,1), (0) ( LSB を 0 として ) となる。

### レジスタ読みだし

書き込みの場合と同様、相互作用パイプライン側でデコードを行なう。従って、インターフェース信号は表 6.11 のようになる。

タイミングは同期型であり、アドレスをクロック立ち上がりでサンプルする。チップ内なのでトライステートバスとか使うのも嫌なので、とにかくアドレスを入れたらデータが出るものとする。ただし、計算（積算）中には出てくるデータは嘘でも構わない。表 6.12 に相互作用パイプライン一つのなかのレジスタマップを示す。BBB で示されたビットは仮想パイプラインの ID であり、下位 4 ビットが物理レジスタの区別になる。なお、ネイバーリストについては 2 つの物理パイプにつき一つしかないので、NNB, NBL はそれらだけのための独立なポートを持つことになる。

ポートのデータ幅は 32 ビットである。RNB は 36 ビット形式なので溢れるが、これは下 4 ビットを切捨てて返すものとする。

表 6.10: 相互作用パイプライトレジスタマップ

データ	アドレス (2進)	説明
XH	BBB0000	I 粒子 x 座標
XL	BBB0001	I 粒子 x 座標
YH	BBB0010	I 粒子 y 座標
YL	BBB0011	I 粒子 y 座標
ZH	BBB0100	I 粒子 z 座標
ZL	BBB0101	I 粒子 z 座標
VX	BBB0110	I 粒子速度 x 成分
VY	BBB0111	I 粒子速度 y 成分
VZ	BBB1000	I 粒子速度 z 成分
EPS2	BBB1001	ソフトニング
H2	BBB1010	ネイバー半径
I	BBB1011	粒子番号
SCALES	BBB1100	PHISCALE, FSCALE, JSCALE
R0	10000000	カットオフスケールファクタ
CTAB	0YXXXXXX	カットオフテーブル

表 6.11: 制御部-相互作用パイプ読み出しインターフェース信号定義

信号	幅	説明
DOUT	32	データ
AOUT	7	アドレス
DNB	32	ネイバーデータ
ANB	10	ネイバーアドレス

表 6.13 にステータスレジスタの中身をしめす。この表で、相互作用パイプのために実際に必要なのは 21 ビットなので、まだ 11 ビット余っている。これを、データ転送エラーを示すのに使うことにする。なお、これは、一応ここに書くが、相互作用パイプラインが面倒を見るわけではなく、FO ポート制御部で適当に処理する。したがって、相互作用パイプラインはここに関しては 0 を返すだけである。

ネイバーリストは 256 粒子分用意するが、1 ワードが 32 ビットを超えるのでアドレスは 512 語分必要となる。偶数ワードの下の方にフラグを入れる。語数として返るのは 256 までであるとする。

### 動作制御入力

動作制御入力には表 6.14 がある。テスト入力は別に詳細を述べる。ここでは、計算開始／停止信号と VMP 同期信号の関係について述べる。

表 6.12: 制御部-相互作用パイプリードレジスタマップ

データ	アドレス(2進)	説明
FXH	000BBB0000	I 粒子加速度 x 座標(上位)
FXL	000BBB0001	I 粒子加速度 x 座標(下位)
FYH	000BBB0010	I 粒子加速度 y 座標
FYL	000BBB0011	I 粒子加速度 y 座標
FZH	000BBB0100	I 粒子加速度 z 座標
FZL	000BBB0101	I 粒子加速度 z 座標
POTH	000BBB0110	ポテンシャル
POTL	000BBB0111	ポテンシャル
JX	000BBB1000	I 粒子 jerk x 成分
JY	000BBB1001	I 粒子 jerk y 成分
JZ	000BBB1010	I 粒子 jerk z 成分
RNB	000BBB1011	最近接粒子距離
INB	000BBB1100	最近接粒子番号
STS	000BBB1101	パイプライン演算ステータス
NNB	1000000000	ネイバーリスト長さ
NBL	0XXXXXXXXX	ネイバーリストデータ

図 6.23: 相互作用パイプライン入力データタイムチャート

VMPSYNC は予測子パイプから供給され、 $j$  粒子の有効なデータがあることを示す。実際に計算が行なわれるのは、RUN 信号がハイでかつ VMPSYNC がハイであった時ということにする。すなわち、図 6.23において、RUN は VMPSYNC とは非同期に上がるかもしれない。

相互作用パイプラインは、RUN の立ち上がり（立ち上がりの次の VMPSYNC）で積算レジスタ、ステータスフラグをクリアする。積算レジスタについては、実際には、最初のデータによる計算結果がアキュムレータに来た時に、加算器のもう片方の入力を強制的に 0 にする。ステータスフラグについても、同様に最初に 0 にすることになる。ネイバーリスト、最近接粒子についても同様な処理が必要である。

RUN が下がったら、VMPSYNC で値が来ても、演算結果を積算しないでアキュムレータの値を保持する。図 6.23 の場合では、サイクル 25 に次のデータがくるが、これによる演算結果は積算されないことになる。

表 6.13: 相互作用パイプステータスレジスタの内容

データ	ビット位置	説明
STSFX	0:2	加速度 X 成分
STSFY	3:5	加速度 Y 成分
STSFZ	6:8	加速度 Z 成分
STSJX	9:11	ジャーク X 成分
STSJY	12:14	ジャーク Y 成分
STSJZ	15:17	ジャーク Z 成分
STSP	18:20	ポテンシャル成分
IPWP	28	IP 入力パリティエラー
JPMF	29	JP メモリ入力 ECC エラー
JPMC	30	JP メモリ入力 ECC 修正フラグ
JPWP	31	JP 入力パリティエラー

表 6.14: 相互作用パイプへの制御信号

入力名	機能
RUN	積算開始／停止信号
VMPSYNC	VMP 同期入力
T01-T15	テスト用制御

## 6.3 制御回路

### 6.3.1 クロック

まず、G6 チップ、メモリ、インターフェース回路のクロック系について考える。一応動作速度として 100 ないし 150 MHz を想定する。この場合、外部回路の速度としては、20 ないし 25 MHz 程度に抑えたい。これに対し、メモリの動作速度は G6 チップと常に同一とする。

100 MHz を超えるクロック系をボード全体にばらまくのは不可能ではないが、あまりしたいことではない。もちろん、異なるチップ同士で直接データをやりとりすることはないので、クロックスキューの許容範囲はかなり大きいということに注意してほしい。しかし、それでもやはり大変であることに変わりはない。

従って、一応、チップに供給するクロックは遅いものとし、それを倍速して使うものとする。チップ内の PLL で  $4\times$  から  $6\times$  のものが生成できることが望ましいが、それが出来ない場合にはチップ内の PLL は  $4\times$  固定とする。なお、もしもチップ内では PLL が全く利用出来ないというようなことがあれば、モジュール上にクロックマルティプライア、例えば Cypress CY2308 といったもの置く方法もある。これは出力が LVTTL である。2.5V 入出力の SSRAM でもそれほど問題なく駆動出来るはずである。

以下、SSRAM にどういうものを使うかをちょっと考えてみる。現在アベイラブルな SSRAM はほとんどすべて I/O が 2.5V、コアが 3.3V である。しかし、I/O が 2.5V といってもそれ

表 6.15: 98/2 現在で利用可能な SSRAM の信号レベル規定

メーカー、型番	入力	出力	クロック	備考
Samsung KM736V789	TTL (0.8-2.0)	TTL (0.4-2.4)	TTL	No LW, No JTAG
Moto MCM69R738A	0.7-1.7	0.7-1.7	Diff, center=1.6	LW, JTAG
Moto MCM69R737A	LVTTL	LVTTL	Diff, center=1.6	LW, JTAG
Moto MCM69R736A	HSTL	HSTL	Diff, center=1.6	LW, JTAG
Moto MCM69P737	0.7-1.7	LVTTL	0.7-1.7	No LW, JTAG
Micron MT58LC128K36G1	0.7-1.7	0.7-1.7	0.7-1.7	No LW, No JTAG

でユニークにものごとがきまるわけではない。表 6.15 にどんなものがあるかを示すが、I/O レベルだけでも LVTTL, 2.5V CMOS, HSTL の 3 種、さらに搜せば GTL, CTTL 等いろいろあるわけである。さらにクロックレベルにシングルエンド (CMOS/LVTTL)、差動とあり、さらに差動にも CTTL と Pseudo ECL がある。さらに、動作モードに late write (遅延書き込み: 書き込みデータがアドレスの次のサイクルに入る) とそうでないものがあり、また読みだしのクロックが別に入るようなものもある。読みだしのタイミングは調べた範囲ではみんな同じである。

これらすべてに対応するのは、特に信号レベルについては困難である。したがって、信号レベルについては 2.5V で問題なく可能な LVTTL とし、クロックも低速分はそれですます。高速クロックについても、G6チップに入れるものはとりあえず LVTTL ですますことにする。

実際には、ちょっと問題になるのはクロックの信号レベルである。上に見たように、差動クロックを要求する製品が、特に高速の遅延書き込みタイプでは多い。

これに対しては、もっとも「簡単」な対応は、G6チップ側からクロックを提供することである。が、まあ、できるとも限らないので、そういうことは期待しないことにする。とすれば、PECL か何かの差動クロックを作る必要があろう。これには、例えば、ベースクロックを PECL レベルで供給しておけば、モトローラの 991/990あたりが使える。これを TTL に変換するにはやはりモトローラの 973 を使うことが考えられる。また、ベースクロック自体を供給するのにも 991 が使えるであろう。

差動クロックが必要でなければ、990のかわりに 950/952 を使えばよい。

もちろん、G6チップ側に PLLを入れ込むことができて、しかもそのクロックを外に取り出せるのならばこのあたりは大した問題ではない。G6チップに PLLを入れる場合には、一応 シングルエンドの LVTTL と差動の両方をそれぞれ 2組ずつ出すことにする。

さて、外部回路がチップの 4 分周で動作する場合は、遙倍は容易である。しかし、例えば 6 分周のクロックを使うとすれば、ちょっと面倒な話になる。もっとも簡単には、1:3 のクロックを生成できる、例えばモトローラ MPC950/952/990/992 のようなものをを使えば良いであろう。モジュール上に MPC992 のようなものを載せることの問題点は、これ自体がかなり高価なものであることである。が、ボード全体に 100MHz 以上のクロックを這い回らせるのは現実的とはいい難い。まあ、結局はモトローラ、サイプレスおよび IDTあたりの製品を適当に組み合わせれば、大抵のクロックは作れる。従って、コスト(下の表参照)を気にしなければ、まあ、G6チップとメモリのクロック仕様がどのようなものでも実際上問題はない。

MPC952FAR2	CMTA 2	2000	20.3500	-	LOW VOLT CLOCK DRIVER
MPC950FAR2	CMTA 2	2000	20.3500	-	LOW VOLT CLOCK DRIVER
MPC952FA	CMTA N	250	20.4000	-	LOWVOLT CLOCK DRIVER PLL
MPC950FA	CMTA N	250	20.4000	-	LOW VOLT CLOCK DR.PLL
MPC951FA	CMTA N	250	20.4000	-	LOW VOLTAGE CLOCK DRIVER
MPC990FA	CMTA N	160	31.1300	-	LOW VOLT CLOCK DRIVER PLL
MPC992FA	CMTA N	250	31.9500	-	LOW VOLT CLOCK DRIVER
MPC991FA	CMTA N	160	32.0800	-	LOW VOLT CLOCK DRIVER PLL

とりあえず、メモリには差動 PECL クロック、LVTTL インターフェースのもの（例えば IBM043641RLAB）を想定しておく。また、G6 チップのクロックは、PLL がなくても構わないものとしておく。

さて、外部クロックは、一応内部の 1/6 と 1/4 の 2 通りの場合を考えることにする。これはちょっと面倒な気もするが、150MHz 程度で回った時にそれをいかせないのであまりに残念であるので、1/6 で動作することも可能にする。

内部で PLL を使う場合でも使わない場合でも、外部クロックに同期した信号を外から入力し、これを使って内部のシーケンサを制御することにする。

### 6.3.2 内部シーケンサ

さて、内部シーケンサにはどんなものがあるかここでまとめてみる。

- JP からのデータをメモリに書き込む
- IP からのデータをチップ内の適当な場所に書き込む
- FO からデータを出す
- 計算開始／終了の制御
- 計算中にメモリからデータを読み込み、パイプライン側に供給する。

基本的なものは上の 5 個である。以下、これらがどんなものか順に考えていこう。

### 6.3.3 JP メモリライト

これに JPW という名前をつけることとする。まず、どういう形式でデータが入ってくるかを考える。

なお、古い版においては、ボード間接続インターフェースに ファイバーチャネルまたは Gigabit Ether の、非常に高速なシリアル通信を想定していたが、技術的に面倒なこと、また、（まだ）高価である可能性も高いことを考慮して、より周波数の低いもので済む技術である LVDS を使った 3-4 ビットパラレル転送を使うことにしたい。このメリットは、消費電力が低いこと、データ転送が例えば 3 ビットパラレルの場合、実効的に 21 ビット・30-60MHz の転送が可能であり、可能な範囲が非常に狭い FC などに比べて使い勝手がよい。また、この場合転送幅に余裕があり、ビットずれの心配もほとんどない（少なくとも、ずれっぽしない可能性はない）ので、パリティチェック、あるいは ECC などの普通の方法が使える。

JP からのメモリ書き込みでは、ホストからのネットワークはデータを素通しで放送するだけで、全く加工や制御はしないものとする。

表 6.16 に JP ポートの信号定義を示す。入力データは 32 ビット+バイトパリティの 36 ビットとする。

表 6.16: JP ポートインターフェース信号定義

信号	幅	説明
JPD	36	データ
JPWE	1	書き込み信号 (アクティブハイ)

```

clk      -----=====-----=====-----=====-----=====
WE       ======\-----/=====
data    -----<=data0==>-<=data1==>-----

```

図 6.24: 入力ポートタイムチャート

図 6.24 に write アクセスのタイミングを示す。

WE が下がっている時には有効なデータが来ている。で、これが連續して来るという保証はない。また、G6チップ側から要求できるとか、そういうことは考えない。単にホスト側の都合で送り付けられるだけである。ここで、考えないといけないことは3つある。一つは、外部クロックと内部クロックの違いであり、もう一つはG6チップが実際にメモリにデータを書くかどうかの判定条件である。最後に問題になるのは、メモリとのデータ転送のプロトコルである。以下、順に考える。

まず、外からのデータ取り込みについて考えてみる。要するに、問題は単にいつデータをサンプルするかということである。基本的には、外部クロックの後半の、安定したところで取り込めばよい。この動作速度はそれほど速くないので、次のクロックぎりぎりまで待って取り込むという必要はそれほどはない。したがって、4分周の場合なら、例えば外部クロックから3サイクル目でサンプルすればいいということになる。ここで問題になるのはクロックスキューである。いま、sysclk（外部クロック）とチップクロックがほぼ同期しているが、完全には同期していないとしよう。すると、単にsysclk（あるいはその反転）を内部クロックのクロックエッジでサンプルしたのでは、クロックスキューの具合によって、どこでデータをひろうのかわからなくなり好ましくない。そこで、チップ内でインバータをいれて反転クロックを作る。これで、この信号がチップクロックより位相が遅れることは保証される。この信号が立ち上がった最初のクロックでデータをサンプルする。もっとも、このままでは外部から来た信号をそのまま受けとることになってちょっと面倒なので、外部からの入力はあらかじめ DFF で受けておいて、それをどうするかが判断されることになる。

図 6.26 に実現例を示す。ENABLE 線で外部データをラッチすることになる。この ENABLE 線は、有効なデータをラッチしたときにだけハイになる。したがって、ここから先は、外部クロックのことはもう考えなくていい。

```

clk      -----=====-----=====-----=====-----=====
sysclk  -----=====-----=====-----=====-----=====
/sysclk -----=====-----=====-----=====-----=====
WE       ======\-----/=====
data    -----<=====data0==>-<=====data1==>-----

```

図 6.25: 入力ポート詳細タイムチャート

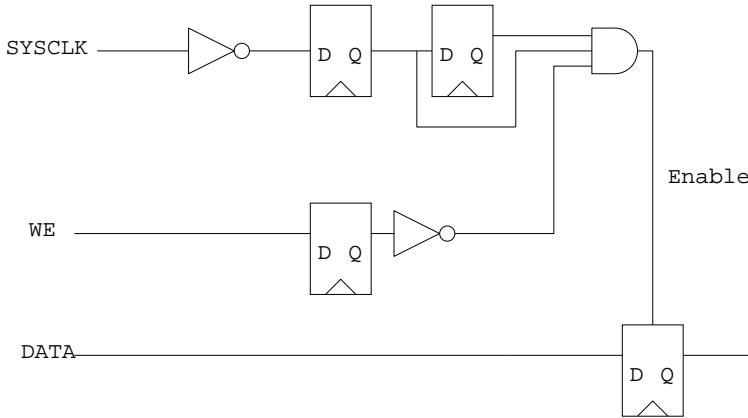


図 6.26: JP データ受けとり回路

表 6.17: JP ポートコマンドフィールド

フィールド名	先頭／第二	ワード内位置	説明
コマンドコード	先頭	[31:30]	コード
チップ番号マスク	先頭	[19:10]	
チップ番号	先頭	[9:0]	
メモリアドレス	第二	[20:0]	

ENABLE 信号を見て、メモリに書くシーケンサが回る。これは、単なるカウンタであって、値が 0 でなければダウンカウントする。0 なら、来るはずの語数にセットする。来るはずの語数は、JP データの 16 ワード（4 バイトワード）とコマンド 2 ワードを足した 18 ワードである。とりあえず、コマンドは先頭ワードの最上位 2 ビットが 0 の時に粒子データ、それ以外は予約コードとする。必要なデータは、メモリ上での先頭アドレス（粒子番号）、チップ番号、チップ番号のどこまでを見るかのフィールドの 3 種であろう。これらの必要なビット数について考えてみる。メモリのアドレスは、仮に 128Mbit くらいをつけたとして（まずありえないが）21 ビットもあればよいので、それくらいということにしておく。チップ番号は、同一クラスタのなかで認識できればよい。したがって 8 ビットで良いが、一応余裕をみて 10 ビットとする。最後に、どこを見るかのデータは、マスクを 10 ビット用意して、そこが 1 の時だけ実際のチップのデータと比べるということにする。結局、表 6.17 の形式でデータをしまう。

ホスト側のライブラリの実装としては、このコマンドワードを必要になるたびに粒子番号から生成してもいいし、またあらかじめ粒子番号とコマンドワードの対応表を作つておいて、そこから読んでもいい。

シーケンサの動作としては、制御カウンタが 0 の時、ENABLE が来たらデータを見て、最上位が 00 ならカウンタにセットする。同時に、来たデータの粒子番号と、チップ内にストアされた粒子番号レジスタを比べ、比べた結果をマスクして依然違っていれば、このチップが受けとるべきデータではないので、メモリへの write 信号を出なくする。

次のデータがきたら、メモリアドレスカウンタに値をセットする。

その次から、データが来たら、メモリアドレスカウンタの値でメモリに書く。で、アドレスをインクリメントする。最下位のアドレスは2つのチップのどちらに書くかを変える。で、必要なデータを書き終えたら、制御カウンタが0になっておしまいである。

ここで、ちょっと考えておいたほうがいいのは、いわゆる endianness をどうするかである。CPUによって、あるいはCPUと同じでもOSによって、64ビットワードを32ビットワード2語としてアドレスしたときに、小さいほうのアドレスに上位ビットがくるか下位ビットがくるかは違う。従って、どこかのレベルでこれに対応する必要がある。PCIの場合にはこれは自動的に決まるような気もするが、調査が必要である。

もうすこし落ち着いて考えると、PCIはおそらく64ビットインターフェースをとるので、word orderはPCIインターフェースカードのところで決めるほうが現実的であるかもしれない。いずれにしても、最小転送単位は4バイトではなく8バイトとしておく。

なお、SECDED (Single Error Correct Double Error Detect) のECCを使って書き込むことにするので、32ビット2語(バイトパリティ)を受けとてから、ECCを生成して上下両方のメモリに一度に書き込むことになる。ECC生成については別6.3.7節で述べる。

入力データにパリティエラーがあれば、そのことをチップ内部のレジスタに記憶しておく。

さて、異常動作からの回復とかいったことも考える必要がある。シーケンサをクリアする方法としては、シーケンサリセット信号(同期リセットでいい)を準備して、入出力ポートとは無関係に強制的にカウンタ、シーケンサをリセットすればいいであろう。これによりJPWの他のシーケンサもすべてリセットされてアイドル状態に戻るものとする。これでエラーステータスもクリアされる。

JPライトについては、すべてのデータを送る必要があるわけではないということを考慮する必要がある。つまり、例えば独立時間刻みであればA2BY18といった高次の導関数が必要になるが、ツリーコードであれば位置と質量だけでいいわけである。また、座標も32ビットで十分である。例えばこれだけにすれば、データ総量が128ビットとフルに送る場合の1/4にすることができる。特にツリーコードの場合、 $j$ 粒子のデータ転送量が究極的には計算速度をリミットすることになるので、これに対して何らかの対策をとれるに越したことはない。まあ、今のところ粒子毎に8バイトのヘッダをつけるという想定なので、1/4までは落ちないがそれでも1/3にはなる。

この問題に対応するとして、2通りの方針があり得るであろう。一つは、G6チップ側ではなにもしないということである。この場合、JPデータのリンクにはフルデータが流れることになる。例えばPCIインターフェース、あるいはJP書き込みの最初または最後で面倒を見る。

もう一つの考え方は、G6チップ側で面倒をみるとことである。この場合、モードとか考え出すと面倒なので、アドレス変換テーブルを持たせて完全にプログラマブルにするほうが簡単であろう。すなわち、一旦インターフェースのシーケンサのなかでデータを組み立てて、それからまとめて書き込むようにする。

もっとも力任せな設計では、512ビット長のレジスタを2本用意して、来たデータを一方のレジスタにシーケンサがアドレス変換テーブルを見ながら書く。で、一粒子分が終ったら、それをもう片方にコピーするとともに、メモリに書き込むほうのシーケンサを起動する。こちらは必ず8サイクル回って終了する。実際には、ヘッダ2ワードを読み込むのに必ず8サイクル以上かかるので、コピー用のレジスタは持たないで直接書いてしまうというのも構わない。

アドレス変換テーブルが昇順になっているという制限をつけるなら、512ビットのレジスタではなく64ビット分だけがあるというのでも構わぬことになる。

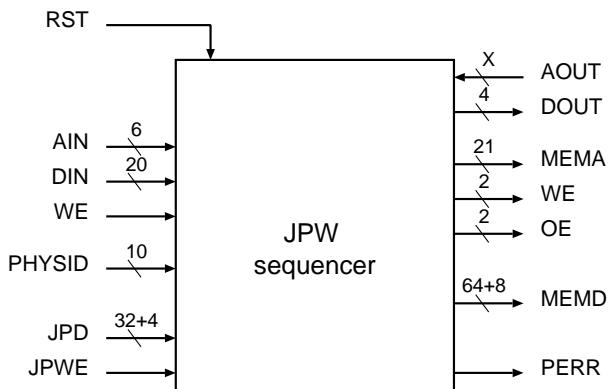


図 6.27: JPW 制御部 インターフェース

この場合、JP データ取り込みのカウンタのカウント数上限と、カウント数からレジスタアドレスを生成するテーブルを設定することになる。カウント数は最大 16 なので、4 ビットとして 4 ビット入力 4 ビット出力のテーブルを持たせることになる。これらは IP 側から書き込むことにする。

この機能はちょっと邪悪であるが、これによりエンディアンネスの問題等にもチップ側で対応できるし、また、ハードを設定可能にしておくことで結局ハードバグの可能性を減らせるであろう。

図 6.27 に信号定義を与える。

A, D, WE は上の制御入力であり、PHYSID はチップの物理 ID, JPWE, JPD は JP ポートからの入力である。MEMA, WE, OE, MEMD は外部メモリへのアドレス、データ、ライトイネーブル制御線、出力イネーブル制御線である。

表 6.18 に JPW の制御レジスタのアドレスマップをしめす。これらの制御データはすべて IP 側から書かれるということにする。ADLY, WDLY, DDLY の 3 つは、基準時刻に対してアドレス、WE、データを遅延させるパイプライン段数を示す。なお、OE のネゲートが書き込みサイクルの始まりを定義する。これらを制御可能にすることで、例えばディレイドライトに対応したり、またメモリアドレスにレジスタードバッファ、あるいは変換テーブルをつけたりすることを可能にする。さらに、ODLY が、最後のデータが出てから何クロック後に OE を下げる（アサートする）かを決める。

VCIS は物理／仮想チップ番号であり、MSB 10 ビットが物理、LSB 10 ビットが仮想とする。物理番号が外から供給されている物理 ID と等しい時にのみ、仮想番号を変更する。この仮想番号は上でのべたメモリ書き込み制御に使う。

ステータスレジスタは今のところ 4 ビットだけで、入力データにパリティエラーがあった時その場所を記憶する。これは SRST がハイでクリアされる。パリティは奇数パリティで、8 ビットごとに 1 の数を数えて奇数の時 1 である。なお、このすべての OR をとったものも出力しておく。ステータスレジスタは、現在のところこれしかないのでこれを DOUT から直接出力し、AOUT は無視する。

WE, OE, MEMA はメモリチップが 2 個あることに対応して 2 本（組）出す。

表 6.18: JPW 制御テーブル入力アドレスマップ

データ	アドレス (2進)	ビット長	説明
MAP	00XXXX	4	アドレス変換テーブル
ND	010000	4	一粒子のデータ数 (32ビットワード)
VCID	010001	20	物理チップ番号／仮想番号
ADLY	010010	2	アドレスディレイ
WDLY	010011	2	WEディレイ
ODLY	010100	2	OEディレイ
DDLY	010101	2	データディレイ
予約	1XXXXX	20	拡張用予約

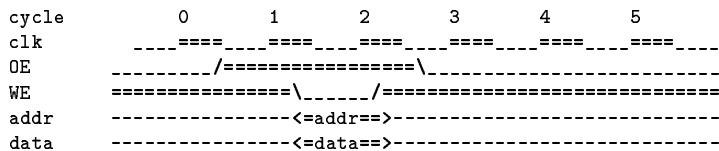


図 6.28: JPW メモリポート詳細タイムチャート

図 6.28 にメモリ側のタイミング例をしめす。これは ADLY=WDLY=DDLY=1, ODLY=2 の場合であり、遅延書き込みでない通常の SSRAM に対応している。ADLY=WDLY=0, DDLY=1, ODLY=2 とすれば遅延書き込み対応になる。

なお、明確に述べていないが、このユニットではメモリに書かれるデータが物理的に何と対応しているかとか、いないかとか、そういうことは一切考えない。来たデータを 2 ワード毎にアラインして書くだけである。

なお、メモリに書く時には、2つのメモリユニットに交互に書く（従って、全部で 16 サイクル使う）ようにする。これは、同時スイッチングする信号線の数を減らすためである。

### 6.3.4 IP レジスタライト

ポートとの物理的なインターフェースは JP の場合と全く同じで、有効なデータが WRITE ENABLE 信号で示されるということになる。

I 粒子データについては送るものは表 6.19 のようになる。これは全部で 12 語ということになる。I 粒子データについては、最初のコマンドワード（これも 8 バイトバウンダリに合わせたいので、2 語とする）で指定すべきことは粒子数だけである。その他、チップのモード等（もししあれば）を書くコマンドはすべてここから送ることにする。

なお、6.1.24 節と同様に、速度、ソフトニング、ネイバー半径については 32 ビット IEEE754 形式から 36 ビットの GRAPE-6 内部形式に変換する必要があることに注意して欲しい。

I 粒子データについては、J 粒子と違ってあまり邪悪なことは考えない。これは、まあ、少なくともツリーコードの場合にはやってもそれほどメリットが大きくならないからである。なお、

表 6.19: I粒子データ

データ	ビット数	個数
位置	64	3
速度	36	3
ソフトニング	36	1
ネイバー半径	36	1
粒子番号	32	1

注意して欲しいことは、ここではチップに入ってくるものと相互作用パイプラインに送られるものは違うということである。具体的には、速度、ソフトニング、

その他、チップのモードなどを書くコマンドはすべてここから送ることにする。シーケンサ等の設計を簡略化するため、論理的にはすべてのレジスタがユニークなアドレスを持つものとし、書き込みは指定したアドレスから指定した語数のパケットという形で来ることにする。

粒子データ以外にチップに送られるべき情報としては、

- 粒子数
- 計算開始信号
- 現在時刻
- カットオフスケールファクタ
- カットオフテーブルの中身

といった、パイプラインの通常のレジスタの他、

- 論理チップ ID 変更データ
- チップディセレクト命令
- テストモード設定レジスタ
- JP データ書き込みシーケンサ設定レジスタ
- ネイバーリスト読みだし開始命令

といったものがある。これらはそれぞれ独立なアドレスをもつものとする。なお、実際には、JP 制御部、相互作用パイプラインなどのブロック毎にアドレスマップがあるので、IP 制御部では入ってくるアドレスの上位をデコードして必要なブロックへの WE を出すということになる。

レジスタマップを表 6.20 に与えたようなものとする。ここで、AAA となっているのは物理パイプライン番号、BBB は仮想パイプライン番号である。

この方法においてちょっと嬉しいのは、パイプライン 1 本分のデータが 2 の幂乗ではないことである。このため、実際に連続アドレスで書き込みが行なえるようにするには、アドレスデコーダがかなりややこしいものになってしまう。これを、JPW の時と同様にアドレス変換テーブルを使った間接バーストにより実現する。この実現についてはもうちょっとあとで述べる。まず、相互作用パイプラインのほうから見た信号定義とタイミングを表 6.21 と図 6.29 に与える。

図 6.29 に write タイミングを示す。単なる同期書き込みである。

図 6.30 に IPW ユニット全体の信号定義を与える。AFP は相互作用パイプライン用、A はそれ以外のすべて用のアドレスである。

表 6.20: IP ポートデータライトレジスタマップ

データ	アドレス (2進)	ワード数	説明
XH	0000AAABBB0000	2	I 粒子 x 座標
XL	0000AAABBB0001	2	I 粒子 x 座標
YH	0000AAABBB0010	2	I 粒子 y 座標
YL	0000AAABBB0011	2	I 粒子 y 座標
ZH	0000AAABBB0100	2	I 粒子 z 座標
ZL	0000AAABBB0101	2	I 粒子 z 座標
VX	0000AAABBB0110	1	I 粒子速度 x 成分
VY	0000AAABBB0111	1	I 粒子速度 y 成分
VZ	0000AAABBB1000	1	I 粒子速度 z 成分
EPS2	0000AAABBB1001	1	ソフトニング
H2	0000AAABBB1010	1	ネイバー半径
I	0000AAABBB1011	1	粒子番号
SCALES	0000AAABBB1100	PHISCALE, FSCALE, JSCALE	
IPRG	0001XXXXXXXXXX	1	IP 制御部内部レジスタ
JPRG	0010XXXXXXXXXX	1	JP 制御部内部レジスタ
CTBL	0011XXXXXXXXXX	1	カットオフテーブル
FORG	0100XXXXXXXXXX	1	FO 制御部内部レジスタ
CARG	0101XXXXXXXXXX	1	計算制御部内部レジスタ
PRTI	01100000000000	1	予測パイプ TI 上位ワード
PRTL	01100000000001	1	予測パイプ TI 下位ワード

表 6.21: IP ポート制御回路とパイプラインのインターフェース

信号	説明
A0-AX	アドレス線
D0-D31	データ線
WE	ライトイネーブル

```

clk      =====---=====---=====---=====---=====
WE       ======\_____/_=====---=====
address -----<=address0>-<=address1>-----
data     -----<=data0==>-<=data1==>-----

```

図 6.29: IP ポート制御回路とパイプラインのタイミング

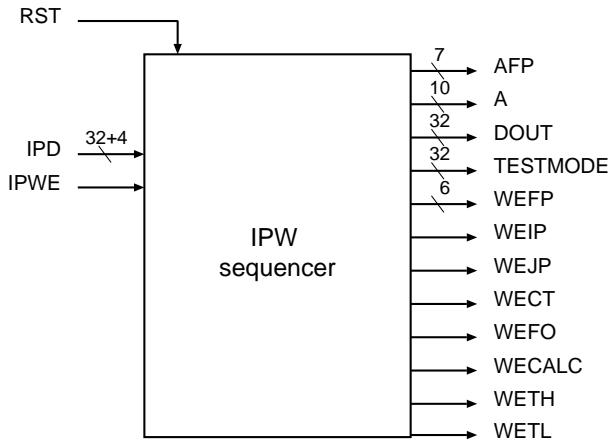


図 6.30: IPW 制御部 インターフェース

さて、こんなものをどうやって実現するかということを考えないといけないが、バグがあったりするといやなのでなるべくプログラマブルにしたい。それにより、場合によっては後で邪悪な機能を実現することも可能にしておく。このためには、ある程度汎用的なシーケンサにしておくことが望ましいであろう。

IP ポートからのパケットは 2 バイトヘッダとその後のデータという形を想定している。で、*i* 粒子データを実際に書く時は間接バーストをするが、それ以外のデータについてはすべて連續バーストということにしておく。従って、*i* 粒子領域以外に不連続アクセスするためには、複数の転送を行なう必要がある。これは、性能に影響するものではないのでいいことにする。

間接バーストの場合に面倒なのは、アドレス変換テーブルがいることと、カウンタが途中でキャリを出して戻る必要があることである。

実際に間接アクセスする必要があるのは、せいぜいアドレス下位 4 ビットなので、テーブルには 4 ビットの入力があればいいことになる。このテーブルを中心に持つておいて、これを使ってアドレスを出す。

とにかく、図 6.30 のようなものを実現しさえすればいい。WE はたくさんあるが、これは実際にはアドレスの上位ビットを適当にデコードして出てくるだけなので、例えば 4 ビットフルデコードで出しておけば何も考へる必要はない。

通常の転送時に入ってくるデータとしては、スタートアドレスと転送語数がコマンドに入っていて、その後は指定した語数だけデータが来るということになる。ホストの方の都合を考えて、データは 64 ビット単位でくるということにしておくので、最初の 2 ワードにスタートアドレスと転送語数ということにする。これらは、一応この順でくるということにしておく。

つまり、まずスタートアドレスと転送語数が来て、これで動作が始まる。後は、データが来るたびに語数カウンタを減らし、データを書き、アドレスカウンタを必要に応じて増やす。なお、実際に間接バーストに対応するのは *i* 粒子を書く時だけなので、それ以外の時の動作は確定している。つまり、データが来るたびに WE を出してアドレスカウンタを増やす。どちらをとるかはアドレスの最上位で決める。

*i* 粒子のために必要なアドレスは 10 ビットである。この上にさらに 4 ビット付け加えてここを使ってブロックごとのデコードをすることにするとすれば、全部で 14 ビットのアドレス

表 6.22: I粒子書き込みサイクル

サイクル	アドレス (2進)	説明
0	X0000	X 第1ワード
1	X0001	X 第2ワード
2	X0010	Y 第1ワード
3	X0011	Y 第2ワード
4	X0100	Z 第1ワード
5	X0101	Z 第2ワード
6	X0110	VX
7	X0111	VY
8	X1000	VZ
9	X1001	EPS2
10	X1010	H2
11	X1011	INDEX
12	X1100	SCALES
13	Y0000	X 第1ワード
14	Y0001	X 第2ワード

があればいい。*i*粒子の物理パイプのためのデコードはさらに別ユニットで行なう。*i*粒子一つを書くためのアドレスシーケンスを表 6.22 に示す。

具体的には、アドレス生成部は図 6.31 のようなものを作ればよいであろう。

図 6.31 では細かい制御線は省いてあるが、DR はデータレジスタで IPD から有効なデータが来た時にロードされる。ワードカウンタは、ワード数が来たら憶える。で、あとは有効なデータが来る度にカウントダウンし、0 になったら止まる。アドレスカウンタは、ワードカウンタのデータが 0 でなければカウントアップするが、下位 4 ビットについては、設定された値まで来たらキャリーを出して次で 0 に戻るようにしておく。

下位 4 ビットの値は、アドレス変換テーブルを通して外に出ていく。

デコーダ 1 は単なるデマルチプレクサで、アドレスの上位 4 ビットを 16 ビットのなかの 1 つだけが 0 のものに変換する。オール 0 は相互作用パイプライン用ということにするので、それ以外の 15 本がでることになる。なお、これはワードカウンタと IPWE から制御線が入って、有効なデータが来た時にだけ 1 を出す。

デコーダ 2 はも基本的には単なるデマルチプレクサで、アドレスの上位 4 ビットの次の 3 ビットを 8 ビットのなかの 1 つだけが 0 のものに変換する。ただし、上位 4 ビットがすべて 0 の時だけ 1 をだす。

これらのテーブルは、IPREG のアドレス空間の中に書き込むことによって書き込めるものとする。アドレス割つけは適当で構わない。例えば、アドレステーブルは全部で 64 ビット、WE テーブルは 16 ビットしかないので、それぞれまとめて 1 語として書くというのもいいし、適当に分けてかくのでも構わない。

IPREG のマップは一応 6.23 のように決めておく。

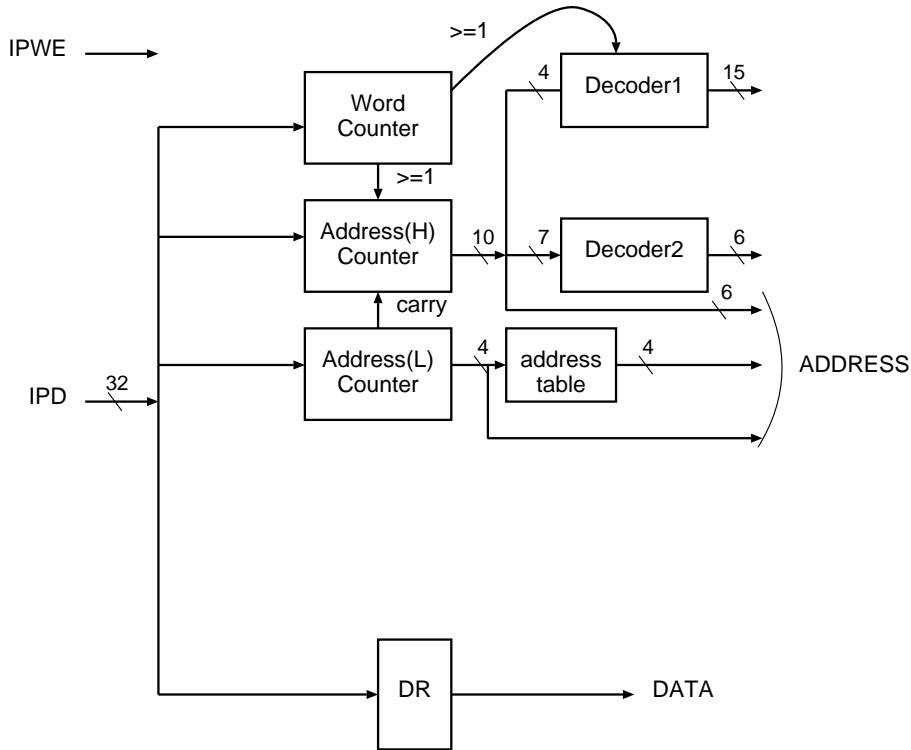


図 6.31: IPW 制御部概念図

### 6.3.5 結果読み出し

結果読みだしについても、パイプラインの側はランダムアクセスに対応できるものとする。なお、最終段の加算器で仮想パイプライン用に使うレジスタの実現法によっては、ランダムアクセスがかなり困難になることはあり得る。例えば、仮想パイプライン用レジスタの最も安直な実装は、シフトレジスタによるものである。が、この場合、計算を止めてからランダムアクセスをしようとなれば、

- 計算が終った時にデータホールドする仕掛け
- ランダムアクセスするためのマルチプレクサ

が必要になり、結構大規模な回路になる。もちろん、他のところに比べて無視できる大きさであるといえばそうかも知れないが、レジスタファイルの大きさは一般に馬鹿にならない。これに対し、ランダムアクセスを許さないとすれば、シフトレジスタからそのままデータを出すだけでいいことになる。読みだし要求自体を仮想パイプラインのベースクロックに合わせて出すことにしてしまえば、それほど面倒なことはない。

レジスタファイルをデュアルポートメモリを使って実現すれば、上のような問題はないが、ワード数が小さいので相対的に面積が大きくなり過ぎるかもしれない。

表 6.23: IPREG 制御レジスタマップ

データ	アドレス (2進)	説明
MAP	00XXXX	アドレス変換テーブル
ND	010000	一粒子のデータ数
TESTMODE	010001	予測子パイプおよび相互作用 パイプのテストモード
予約	1XXXXX	未定義

表 6.24: FO ポートデータリードレジスタマップ

データ	アドレス (2進)	説明
FPREG	0000AAABBBXXXX	相互作用パイプライン結果レジスタ
NNBREG	01AA1000000000	ネイバーリスト語数レジスタ
NBLMEM	01AAOOXXXXXXXX	ネイバーリストメモリ
チップ予約	1XXXXXXXXXXXXX	未定義

但し、一つ問題になるのは、仮想パイプラインの分周比が外部クロックの分周比よりも大きいことである。このために、仮想パイプラインの 1 サイクル毎のデータ出力では外部バスの要求バンド幅を満たせない。

以下では、とりあえず何らかの方法でランダムアクセスを実現したとする。制御回路とパイプラインチップのインターフェース信号は表 6.25 の通りであり、タイミングは図 6.32 に示す。単なる同期読みだしである。リードレイトンシは別途指定する。ここは、リードイネーブルとかではなくて、アドレスに対応したデータが常時出てくる。FO 部はパイプラインの本数だけの入力ポートを持ち、中のマルチプレクサでデータを選ぶ。

さて、相互作用パイプラインの他にデータを返すべきものがないかどうかというと、以下の 2 つがあることがわかる

1. ネイバーリストユニット
2. 諸々のステータス

ステータスは、主に転送エラー状態を記憶するものである。送り返すデータが正しいかどうかは受けとってみないとわからないので、これはインターフェースボードにステータスがあることになる。従って、ステータスがあるのはもうう側、すなわち JPW, IP, JPMEM のインターフェースということになる。FO ユニットからみたチップ内アドレスマップは表 6.24 のようになる。

FO 部用のアドレス生成回路は IP 部のものと基本的には同じである。つまり、相互作用パイプラインから読み出す時にはアドレス変換テーブルを使う必要がある。

外部インターフェースの仕様について考える。ネイバーリストのところで述べた様に、フロー制御が必要であることに注意すると、信号定義は表 6.27 以下になる。

図 6.33 から 6.35 に 出力ポートのタイミングを示す。WD はサンプルされたクロックの次のクロックで出るデータに反映されるものとする。つまり、タイムチャートとしては WD が

表 6.25: FO ポート制御回路とパイプラインのインターフェース

信号	説明
A0-AX	アドレス線
D0-D31	データ線

```

clk      -----=====
address  -----<=address0>-<=address1>-----
data     -----<=data0==>-<=data1==>-----

```

図 6.32: FO ポート制御回路とパイプラインのタイミング

表 6.26: FO ポート相互作用データリードレジスタマップ

データ	アドレス (2進)	ワード数	説明
FX	000AAABBB0000	2	I 粒子加速度 x 座標
FY	000AAABBB0010	2	I 粒子加速度 y 座標
FZ	000AAABBB0100	2	I 粒子加速度 z 座標
POT	000AAABBB0110	2	ポテンシャル
JX	000AAABBB1000	1	I 粒子 jerk x 成分
JY	000AAABBB1001	1	I 粒子 jerk y 成分
JZ	000AAABBB1010	1	I 粒子 jerk z 成分
RNB	000AAABBB1011	1	最近接粒子距離
INB	000AAABBB1100	1	最近接粒子番号
STS	000AAABBB1101	1	パイプライン演算ステータス

表 6.27: FO ポート制御回路とパイプラインのインターフェース

信号	入出力	説明
D0-D35	O	データ線 (バイトパリティ含む)
VD	O	Valid data 有意味なデータが出ていていることを示す
ND	O	New data データが前のサイクルと変わっていることを示す
STS	O	status アイドルではなくなにかしていることを示す
ACTIVE	O	0 のとき使われていることを示す。
WD	I	Wait data データを止める

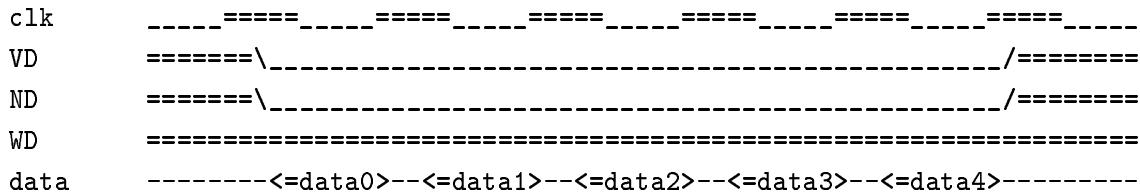


図 6.33: 出力ポートタイムチャート（ウェイトなし）

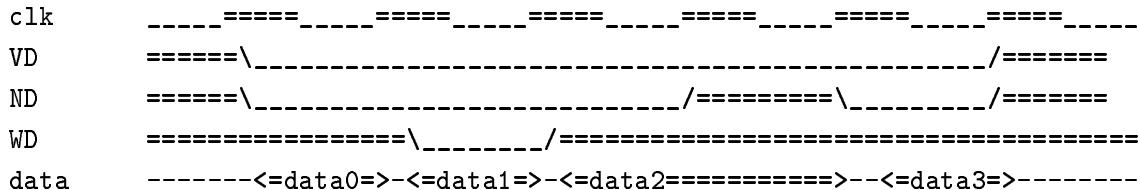


図 6.34: 出力ポートタイムチャート（ウェイトあり）

サイクル $i$ で下がっていたとすると、ND がサイクル $i+2$ で上がっているということになる。ND が上がっているサイクルではデータは更新されなければいけないことに注意してほしい。さらに、最初に WD がアサートされている時の動作は特殊になる。というのは、この場合は WD が出ていても VD, ND をアサートするからである。

FO 部からチップ内部へのアクセスはもちろんレイテンシがあるので、これらの図に示したように、WD に対してディレイなしで応答するのは一見困難にみえるかもしれない。しかし、ここで示しているのは外部クロックであるということに注意して欲しい。このため、FO のアドレス生成は 6 サイクルに一度しかインクリメントされる必要はない。チップ内の他ユニットのレイテンシが異常に大きくなる限り、FO 部は WD を見てから動作を変えればいいことになる。

外部への出力インターフェースで問題になるのは、出力が終ったことを知らせる方法である。これはネイバーリストの場合に特に問題となる。LVDS を使うことにしたので、線の本数には余裕が出来る。したがってこれに専用線を割り当てても問題はなかろう。計算結果出力中、ネイバーリスト出力中は VD を下げるということにする。これは、最後のデータに同期して上げる。

さらに、計算中は STS を下げる。

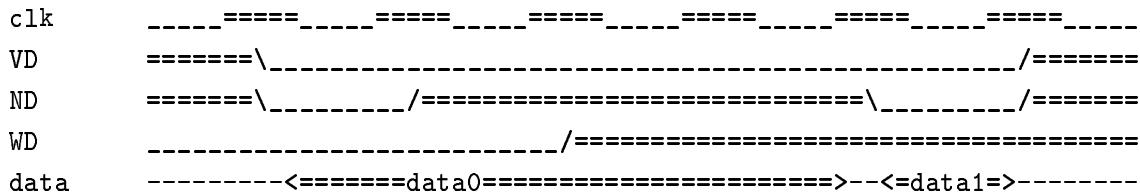


図 6.35: 出力ポートタイムチャート（最初に WD アサート）

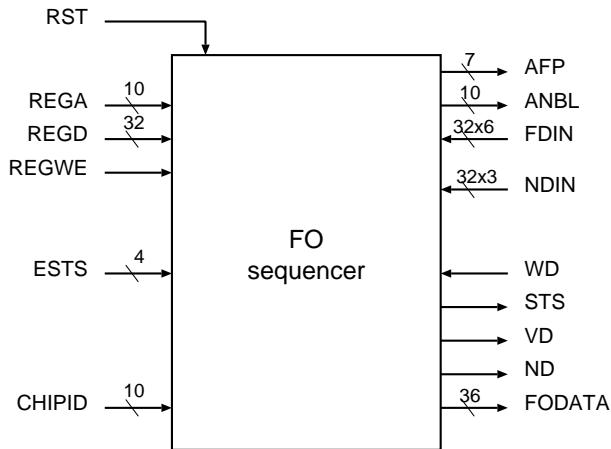


図 6.36: FO 制御部 インターフェース

さらに、ACTIVE という信号線をつけ、これがハイならホストはこのチップの存在を無視するということにする。これは、故障したチップをスキップするのに使う。物理的にこの線をカットできるように基板を作つておく必要がある。レジスタ書き込みによってソフトウェアで操作し、論理的に無視することもできるようにしておく。

ちょっと全体像が見えなくなっているので、FO 部のインターフェースを図 6.36 に示す。左上は IP から来るコマンド用の入力インターフェイスである。レジスタはたくさんあるので、適当にコマンドを作れることになる。表 6.28 は信号の簡単な説明である。

FO 部の動作は今のところ以下の 3 種類ということになろう：

1. 相互作用パイプライン読みだし
2. ネイバーリスト読みだし
3. その他ランダム読みだし

相互作用パイプライン読みだしは、それ以外の 2 つとはちょっと違うことを注意しておく。これは WD を見る必要がない。また、計算が終ったら勝手に動作を始める。送り返すべき粒子数については、IP 側からコマンドで書くということにする。

ネイバーリスト読みだしでは、WD を見る必要がある。さらに、コマンドが来たら、3 つのユニットを順に見ていく。また、それぞれについて、まず語数を読みだして、その語数だけ送るようにする。

ランダム読みだしでは、指定されたアドレスと次の 2 語だけを返す。これは WD を見ることにしておく。

ネイバーとランダム読みだしについては、IP 側の仕様から、2 ワードのコマンドが来るこことになる。そのフォーマットを表 6.29 のようにしておく。なお、アドレスマップは表 6.31 である。FOREG は、IP と同様の間接バーストを実装するためのテーブル等に使う。また、チップ番号とチップ番号マスクは、JPW と同様にアドレスとして使う。

INACTIVE は、他の制御レジスタとは違ってチップ毎に独立に制御したい。このため、チップ番号マスクとチップ番号をエンコードして送ることにする。その形式は表 6.32 に示す通りである。

表 6.28: FO ポート制御回路の全体インターフェース

信号	入出力	説明
RST	I	状態リセットしアイドルに戻す
REGA0-9	I	コマンドレジスタアドレス
REGD0-31	I	コマンドレジスタデータ
REGWE	I	コマンドレジスタライトイネーブル
ESTS0-4	I	データ転送エラーステータス
CHIPID0-9	I	仮想チップ番号
AFP0-6	O	相互作用パイプレジスタアドレス
ANBL0-9	O	ネイバーリストユニット内部アドレス
FDIN0-31	I	相互作用パイプデータ入力(6本)
NDIN0-31	I	相互作用パイプデータ入力(3本)
FODATA0-35	O	データ線(バイトパリティ含む)
VD	O	Valid data 有意味なデータが出ていていることを示す
ND	O	New data データが前のサイクルと変わっていることを示す
STS	O	status アイドルではなくなにかしていることを示す
ACTIVE	O	0 の時使われていることを示す。
WD	I	Wait data データを止める

コマンドは IPW 側から 2 語、CMD0 と CMD1 が順に来るので、CMD1 が来たら動作を始める。なお、仮に計算結果を返している最中であれば、コマンドを無視してよい。

ネイバーリストについては、まず語数を返し、それからメモリの中身をその語数(の 2 倍)だけ続けて送るということになる。最初の語数のところに、チップ番号、ユニット番号等をパックしておいて、どのユニットから読んだデータかわかるようにしておく。具体的には、表 6.33 の形式で最初の語を構成する。なお、念のためにこのワードだけ最上位を 1 にしておく。もちろん粒子インデックスは 32 ビットをフルに使えるので、これで区別が保証されるわけではないがい普通はわかる。

表 6.29: FO 部コマンドフォーマット

フィールド名	先頭／第二	ワード内位置	説明
コマンドコード	先頭	[31:30]	コード
チップ番号マスク	先頭	[19:10]	
チップ番号	先頭	[9:0]	
メモリアドレス	第二	[20:0]	

表 6.30: FO 部コマンドコード

コード	名称	説明
00	NBREAD	ネイバーリスト読みだし開始
01	RREAD	2ワードランダム読みだし

表 6.31: FO 部レジスタマップ

データ	アドレス (2進)	説明
FOREG	0XXXX	FO部内部レジスタ
CMD0	10000	コマンドレジスタ下位ワード
CMD1	10001	コマンドレジスタ上位ワード
NI	10010	$i$ 粒子数レジスタ
NW	10011	語数レジスタ
INACTIVE	10100	「非使用」レジスタ

### 6.3.6 計算開始／終了の制御とメモリデータのパイプラインへの供給

計算開始／終了の制御とメモリデータの読みだし／供給は同一のシーケンサで行なうものとする。これを CALC 部と呼ぶ。インターフェースを図 6.37 および表 6.34 に示す。

なお、質量については、ホストは IEE-754 単精度で書き込むことにする。GRAPE-6 の内部表現は別なので、どこかで変換する必要がある。今回はメモリから読み出してから GRAPE-6 相互作用パイプラインの内部形式に変換して予測子パイプラインに送るということにする。これは IP ポートからの相互作用パイプラインの内部レジスタへの書き込みと同様である。

表 6.35 に、メモリの 512 ビットのどこに何を割り当てるかを示す。メモリは 2 つあるので、偶数アドレスに対応するほうを下位ワードと解釈する。また、JJX 等は 2 階導関数である。

メモリへの書き込みについては JPW が面倒をみるのでここではふれない。バスラインが浮かないようにするために、JPW でメモリの出力制御を行なう。

なお、SSRAM のリードレイテンシについても CALC 内でプログラマブルとして、後で附加回路とかも付けられるようにする。表 6.36 にそれらを含めたレジスタマップをしめす。

基本的な動作としては、アドレスカウンタに初期値 N がロードされると、STS がアサートされる（ロウになる）と同時にアドレスが出始める。アドレスの数える数は常に粒子数の 8 倍なので、送られるのは粒子数でありカウンタの設定値は CALC が判断する。で、メモリか

表 6.32: FO INACTIVE データ形式

フィールド名	ワード内位置	説明
チップ番号マスク	[20:11]	
チップ番号	[10:1]	
INACTIVE bit	[0:0]	

表 6.33: ネイバーリスト第一ワード

フィールド名	ワード内位置	説明
1 固定	[31:31]	
仮想チップ番号	[21:12]	
ユニット番号	[11:10]	ネイバーユニットの番号
語数	[9:0]	リストに入っている粒子数

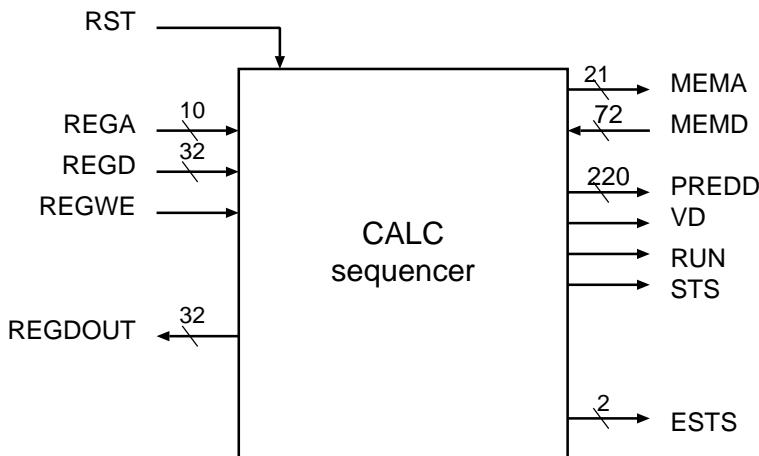


図 6.37: CALC 制御部インターフェース

表 6.34: CALC 制御回路のインターフェース

信号	入出力	説明
RST	I	状態リセットしアイドルに戻す
REGA0-9	I	コマンドレジスタアドレス
REGD0-31	I	コマンドレジスタデータ
REGDOUT0-31	O	レジスタデータ出力
REGWE	I	コマンドレジスタライトイネーブル
MEMA0-20	O	メモリアドレス
MEMD0-71	I	メモリデータ
PREDD0-219	O	予測子パイプ向けデータ出力
VD	O	予測子パイプ同期信号
RUN	O	相互作用パイプ演算信号
STS	O	計算ステータス信号

表 6.35: メモリデータのワード割り付け

ワード番号	データ	説明
0	X	
1	Y	
2	Z	
3	VX VY	(0:31) (31:63)
4	VZ AX AY	(0:31) (32:52) (53:63) 下位 11 ビット
5	AY AZ JX JY	(0:9) 上位 10 ビット (10:30) (31:47) (48:63) 下位 16 ビット
6	JY JZ JJX JJY JJZ T	(0:0) 上位 1 ビット (1:17) (18:28) (29:39) (40:50) (51:59)
7	MASS INDEX	(0:31) (31:63)

らデータが出てくるようになれば、そのあと 8 サイクル毎にデータをまとめて予測子パイプに渡す。この時 VD をアサートする。また、RUN 信号を制御する。これは、最初の VD が上がるのと同期してあげ、最後の VD と同期して下げる。また、STS は相互作用パイプで計算が終るまで下げておく。

全体のタイミングチャートは例えば図 6.38 のようになる。

なお、MEMA が出てから MEMD が来るまでの遅延と、計算が終了してから STS が出るまでの遅延はプログラマブルである必要がある。これらはすべてシフトレジスタで実現する。具体的には、LRAM の値を見て MEMA から MEMD までの遅延を決定する。また、LFORCE で計算が終ってからいつまで STS を下げているかを決める。

なお、SSRAM には WE, OE の他にも山のような制御信号がある。表 6.37 にまとめる。実際上ほとんどのものは制御する必要はない。具体的には、ADSC は low 固定、SB1-3 はアクティブ固定、ADSP, ADV, LBO, SBx, SW はハイ固定でいい。つまり、制御の必要があるのは G と SW のみである。

表 6.36: CALC 制御部のレジスタマップ

名称	アドレス	説明
LRAM	00000	SSRAM リードアクセスレイテンシ
LFORCE	00001	相互作用パイプラインレイテンシ
N	00010	粒子数

(a) 2 粒子の場合。途中で切れていることに注意。

```
clk _____  
(clk/8) -----  
MEMA -----<07><06><05><04><03><02><01><00><XX>  
MEMD -----<D7><D6><D5><D4><D3><D2><D1><DO><XX>  
VD _____ /==\  
PREDD -----<P1>  
RUN _____ /==\  
STS ==\ _____ /==
```

(b) 1 粒子の場合。

図 6.38: CALC 部動作

表 6.37: 典型的な SSRAM の制御信号

名称	説明
ADSC	address status controller 外部アドレスを取り込み、R/W動作
ADSP	外部アドレスを取り込み、READ動作
ADV	advance バーストカウンタを制御。
G	非同期出力イネーブル
K	クロック
LBO	リニアバースト順序制御
SBx	バイトライトイネーブル
SE1-3	チップセレクト
SGW	グローバルライトイネーブル
SW	ライトイネーブル

### 6.3.7 メモリデータ ECC 生成方法について

メモリデータは 64 ビット + 8 ビットなので SECDED (単一エラー修正、2 重エラー発見) が可能である。パリティビットの生成方式は無限にたくさんあるが、ここではパリティマトリックスに (127,120) ハミング符号の最初の 71 行を用いて单一エラーを訂正し、この生成された 71 ビット符号に対して奇数パリティをつけることで 2 重エラー発見を可能にするという安直な生成方式をとる。

以下、簡単にパリティ生成／チェック方式についてまとめておく。まず、 $(n, k)$  符号とは、長さが  $n$  ビットでそのなかの意味がある情報が  $k$  ビットであるものをいう。一般には、符号語のなかに元のデータがそのまま含まれているとは限らないが、ここでは  $k$  ビットは元の送りたいデータ、 $n - k$  ビットはそれから生成されたチェックビットになっているという形式の符号（いわゆる組織符号）を使うことにする。

今回は、 $n = 72$ ,  $k = 64$  である。上位 64 ビットにデータをおき、下 8 ビットにチェックビット（以下、パリティビットと呼ぶ）をおく。この、パリティを付加したデータを  $\mathbf{y} = (y_0, y_1, \dots, y_{71})$  とする。元のデータは  $\mathbf{x} = (x_0, x_1, \dots, x_{63})$  であり、両者の間には

$$y_k = x_{k-8} \quad \text{但し } k \geq 8 \quad (6.38)$$

なる関係がある。

さて、符号語のうちパリティビットの部分を  $\mathbf{p} = (y_0, y_1, \dots, y_7)$  と書くことにして、これが

$$\mathbf{p}^t = \mathbf{G} \cdot \mathbf{x}^t \quad (6.39)$$

という、 $\mathbf{x}$  と行列  $\mathbf{G}$  の乗算で生成されるということにする。ここで、乗算は普通の乗算であるが、その中の加算が modulo 2 のものであることに注意してほしい。この  $\mathbf{G}$  をパリティ生成行列という。 $\mathbf{G}$  は 8 行 64 列である。

これに対し、伝わってきた符号  $\mathbf{y}$  にエラーがあるかどうかの判定は、

$$\mathbf{s}^t = \mathbf{H} \cdot \mathbf{y}^t \quad (6.40)$$

というふうに  $\mathbf{y}$  を行列  $\mathbf{H}$  に掛け、その結果を見ることで判定できる。

結果  $\mathbf{s}$  は 8 ビットであり、 $\mathbf{H}$  は 8 行 72 列である。この  $\mathbf{s}$  をエラー・シンドロームといい、 $\mathbf{H}$  をパリティ行列といいう。

ここで使う  $\mathbf{H}$  は以下のものである。

```

10000000 11011010 10110101 01010101 01101010 10101010 10101010 10101010 11010101
01000000 10110110 01101100 11001100 11011001 10011001 10011001 10011001 10110011
00100000 01110001 11100011 11000011 11000111 10000111 10000111 10000111 10001111
00010000 00001111 11100000 00111111 11000000 01111111 10000000 01111111 10000000
00001000 00000000 00011111 11111111 11000000 00000000 01111111 11111111 10000000
00000100 00000000 00000000 00000000 00111111 11111111 11111111 11111111 10000000
00000010 00000000 00000000 00000000 00000000 00000000 00000000 01111111
11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111

```

この  $\mathbf{H}$  がどのように出来たかということであるが、まず、7 列目（一番上が 0 として）は単にすべて 1 である。これは、全体の偶数パリティをチェックすることに対応している。これは、すでに述べたように 2 重エラーを検出するためにつけていている。一番下を無視すると、まず 0 列目から 7 列は単位行列になっていることがわかるであろう。この行列の各行は符号に

対して1つの拘束条件をつけるわけだが、これが、パリティビットが陽に、しかも他と独立に計算できるという形の拘束条件になっているわけである。

一般には、 $\mathbf{H}$ が单一エラー訂正可能であるための必要十分条件は $\mathbf{H}$ の各列が異なることである。これは、以下のような理由による。

今、 $\mathbf{y}$ が誤りを持たない符号であるときは、 $\mathbf{H} \cdot \mathbf{y}^t = 0$ である。また、 $\mathbf{H} \cdot \mathbf{0} = 0$ である。今、 $\mathbf{y}' = \mathbf{y} + \mathbf{e}_k$ が入力である場合を考える。ここで $\mathbf{e}_k$ は単位ベクトル、すなわち $k$ 番目が1でそれ以外はすべて0であるようなものである。加算であるが、modulo 2なのでこれは $k$ 番目のビットを反転するという操作になる。和が modulo 2 であっても行列とベクトルのかけ算は線形演算であるので、かけ算の結果は $\mathbf{s}_k^t = \mathbf{H} \cdot \mathbf{e}_k^t$ に等しい。つまり、 $\mathbf{H}$ の $k$ 列目がそのままエラー・シンドロームになるわけである。

従って、各位置にあらわれるエラーを区別できるためには、 $\mathbf{H}$ の各列が0でなくてかつ互いに違えばいいということになる。ここでは、全体パリティ以外に7ビット使っているので、 $\mathbf{H}$ の要素として可能なものは0を除いて127個あり、120ビットまでのデータを本来は送ることができ。データが64ビットなのでそのうちの71個しか使わない。それを、以下のよいうな原理で選んだ。

- まず、1つだけ1がある7つを頭にならべた。これは、拘束条件にパリティビットが1つしか現れないようにするためである。
- そのあとは、まだ使われていないものを下から順に並べた。十進として見ると、3, 5-7, 9-15, ... という風に、 $2^k$ を飛ばしたものになっている。
- 最後に、全体パリティの1を各列の一番下につけ、それから7列目に $(0, 0, 0, 0, 0, 0, 0, 1)^t$ を入れる。これによって全体パリティのチェックがなされる。

これは唯一の方法ではないが、とりあえずSECDEDコードを生成することはできる。この方法で $\mathbf{H}$ を作ることのメリットは、 $\mathbf{G}$ が自明に計算できることである。すなわち、拘束条件が、最初の7本、つまり $0 \leq k \leq 6$ については

$$p_k = h_{k,8}y_8 + h_{k,9}y_9 + \dots + h_{k,71}y_{71} + \dots \quad (6.41)$$

という形になっているので、これを $\mathbf{x}$ を使って書き直すと

$$p_k = h_{k,8}x_0 + h_{k,9}x_2 + \dots + h_{k,71}x_{63} + \dots \quad (6.42)$$

いいかえれば、 $\mathbf{G}$ の上7行は、 $\mathbf{H}$ の上7行の8列目以降そのものである。

さて、一番下の行はどうするかであるが、もちろん $p_0$ から $p_6$ までを計算してから全体のパリティをとればいいが、和の線形性から、結局 $\mathbf{H}$ のすべての行の和を作つておけば $p_7$ を $\mathbf{x}$ から直接計算できることがわかる。つまり、 $\mathbf{G}$ は以下のようになる。

```
11011010 10110101 01010101 01101010 10101010 10101010 10101010 11010101
10110110 01101100 11001100 11011001 10011001 10011001 10011001 10110011
01110001 11100011 11000011 11000111 10000111 10000111 10000111 10001111
00001111 11100000 00111111 11000000 01111111 10000000 01111111 10000000
00000000 00011111 11111111 11000000 00000000 01111111 11111111 10000000
00000000 00000000 00111111 11111111 11111111 11111111 10000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 01111111
11101101 00111010 01100101 10110100 11001011 01001011 00110100 11101001
```

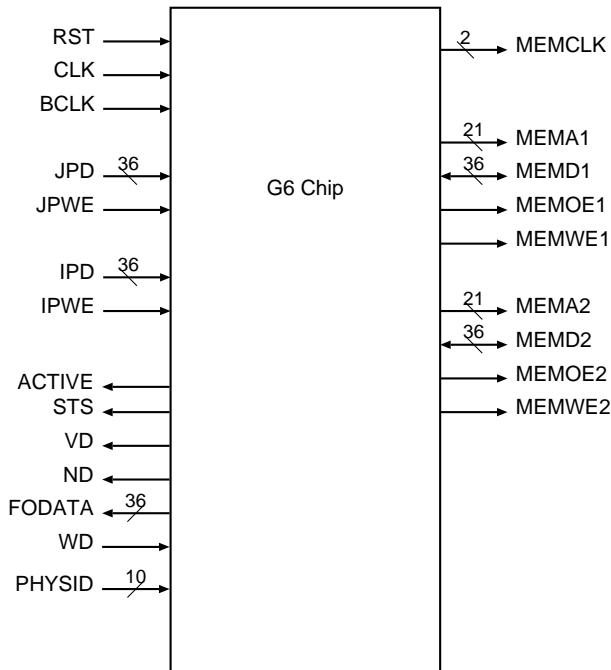


図 6.39: G6 チップの入出力ピン定義

一番下の行は、その上 7 行の偶数パリティになっていることに注意してほしい。

さて、これでパリティ生成と検出ができるわけだが、実際の修正はどうすればいいのだろうか？これには、先ほどのエラーシンドローム  $s$  を使って、 $H$  を逆引きして、一致する列のビットを反転することになる。 $s = 0$  ならエラーがない。また、 $H$  のどの列とも対応しなければ 2 つ以上のエラーがあるわけである。

ハードとしては、 $s$ を入れたら対応するエラーベクトル  $e_i$  がでてくるような表を作つておいて、それと入力符号の XOR をとることで修正出来る。また、 $s$  が 0 ならエラーがないし、 $s$  に対応する  $H$  がなければ修正不可能なエラーがあったということになるので、修正不可能なエラーがあったというのはステータスフラグにしまっておくということになる。修正があった、あるいは修正不可能なエラーがあったというのも、表にして（論理圧縮して）しまつておけるであろう。

## 6.4 チップ全体

ここまで各部の詳細が決まったので、ここでチップの全体像を与えておく。図 6.39 にピン定義、図 6.40 に前節まで述べた各ブロックの関係を示す。なお、IPW ブロックは他のすべてのブロックのレジスタライトをするので、その線がすべて出ているがここではそれは省略されている。また、クロックについては外から速いクロックが入ることを想定した絵になっているが、PLL が使えるならばそちらをつかうことはいうまでもない。

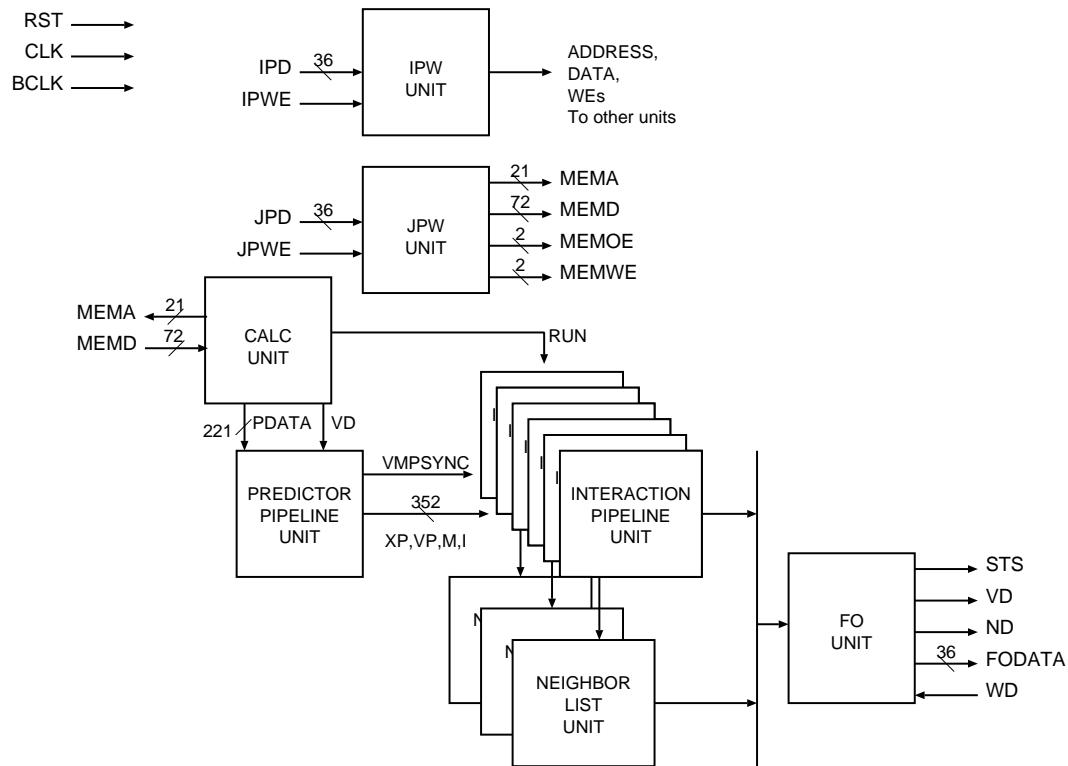


図 6.40: G6 チップのトップレベルブロック図

#### 6.4.1 チップ全体のテストについて

チップ全体について、製造前にどのようなテストを行なったかについて述べる。  
製造前テストは原理的に以下の項目について確認する必要がある

- 仕様が「正しい」かどうか
- シミュレータが仕様通りであるかどうか
- ハードウェア記述がシミュレータと同一であるかどうか

ここでは、最初の、「仕様が正しいかどうか」に関するテストについて論じる。

仕様が「正しい」とは、究極的にはここで述べられた仕様のもの、あるいはシミュレータで実現されたもの（こちらがハードウェアに実現される）で意味のある（主に精度的に）計算ができるということである。

理論的には、仕様の「正しさ」は各部のデータ形式や計算精度などからボトムアップに保証されるわけであるが、まあ、それでは信用出来ないというのは当然であろう。

というわけで、実際に  $N$  体計算コードに組み込んで実験を行なった。プラマーモデルで 5 単位時間 (Heggie unit で) 時間積分を行なった結果、粒子数が 25 から 800 の範囲で実際に計算された粒子の加速度の RMS 誤差が  $2 \times 10^{-7}$  以下（典型的には 1.5）、ジャークのそれは  $5 \times 10^{-6}$  以下（典型的には 4）であった。また、最大誤差は加速度で  $2 \times 10^{-5}$  以下、

ジャークで典型的には  $1 \times 10^{-4}$  以下であった。また、タイムステップを詰めていくことでエネルギー誤差を時間ユニットあたり  $10^{-10}$  程度までは小さく出来ており、計算精度上も特に問題はないものと考えられる。

なお、ネイバーリストに関するテストも行なうべきではあるが、まだ真面目にやってない。



## 第7章 G6 チップの詳細仕様

本章では、G6 チップの詳細な仕様を与える。まず、全体の概要を示し、次に予測子パイプライン、さらに相互作用パイプラインについて述べ、最後に制御部の演算時動作について述べる。

### 7.1 チップ全体

図 7.1 にピン定義、図 7.2 にトップレベルのブロック図を示す。なお、IPW ブロックは他のすべてのブロックのレジスタライトをするので、その線がすべて出ているがここではそれは省略されている。

### 7.2 予測子パイプライン

#### 7.2.1 予測子パイプライン全体

C でのシミュレータでは、予測子全体は 関数 predict (predictor.c) に与えられる。

予測子パイプラインの構成を図 7.3 に示す。

位置の入力および出力は 64 ビット固定小数点形式とする。

演算精度は、基本的には位置の多項式の最終段の加算器は 64 ビット、速度は仮数 24 ビットのブロック小数点形式である。すなわち、予測子パイプラインのなかでは固定小数点形式で演算を行なう。最終段では絶対値に 24 ビット、符号に 1 ビット使う符号つき形式であり、シミュレータでは絶対値は右詰めであり符号は MSB の上に置く。さらに、高次の項にいくほど 4 ビットずつビット数を減らしていく。

まず、等価な Fortran プログラムを示し、演算ごとに書き下してそれぞれの演算フォーマットについて説明する。

```

DO J = 1,N
  S = TIME - T0(J)
  S0 = 0.75*S
  S1 = 1.5*S
  S2 = 2.0*S
  DO K = 1,3
    $      X(K,J) = (((F2DOT(K,J)*S3+FDOT(K,J))*S + F(K,J))*S +
                  V0(K,J))*S+ XO(K,J)
    V(k,j) = ((f2dot(k,j)*s+fdot(k,j))*s1 + f(k,j))*s2
    $          + V0(k,j)
    enddo
  enddo

```

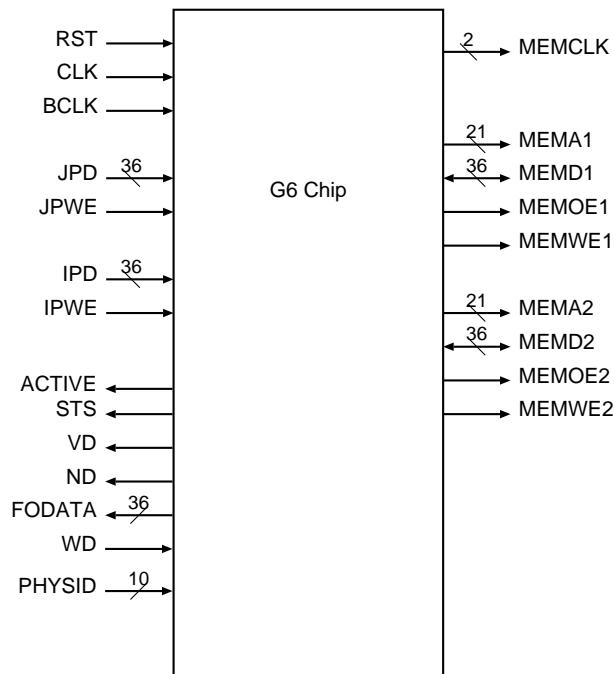


図 7.1: G6 チップの入出力ピン定義

これは、NBODY4 のコードを一部改変したものである。これを、以下のような演算順序で実現することにする。以下 C ライクなシンタックスで書く

- a) DT = T - TJ
- b) K4 = DT\*A2BY18
- c) K4A = K4 \* 0.75
- d) K3 = K4A + A1BY6
- e) K3A = K3 \* DT
- f) K2 = K3A + ABY2
- g) K2A = K2 \* DT
- h) K1 = K2A + V
- i) K1A = K1 \* DT
- j) XP = X + K1A
- k) K3C = K4 + A1BY6
- l) K3D = K3C \* 1.5
- l') K3E = K3D \* DT
- m) K2B = K3E + ABY2
- n) K2C = K2B \* 2
- o) K2D = K2C \* DT
- p) VP = V + K2D

現在時刻は 64 ビット固定小数点で与えられる。  
以下、各演算について述べる。

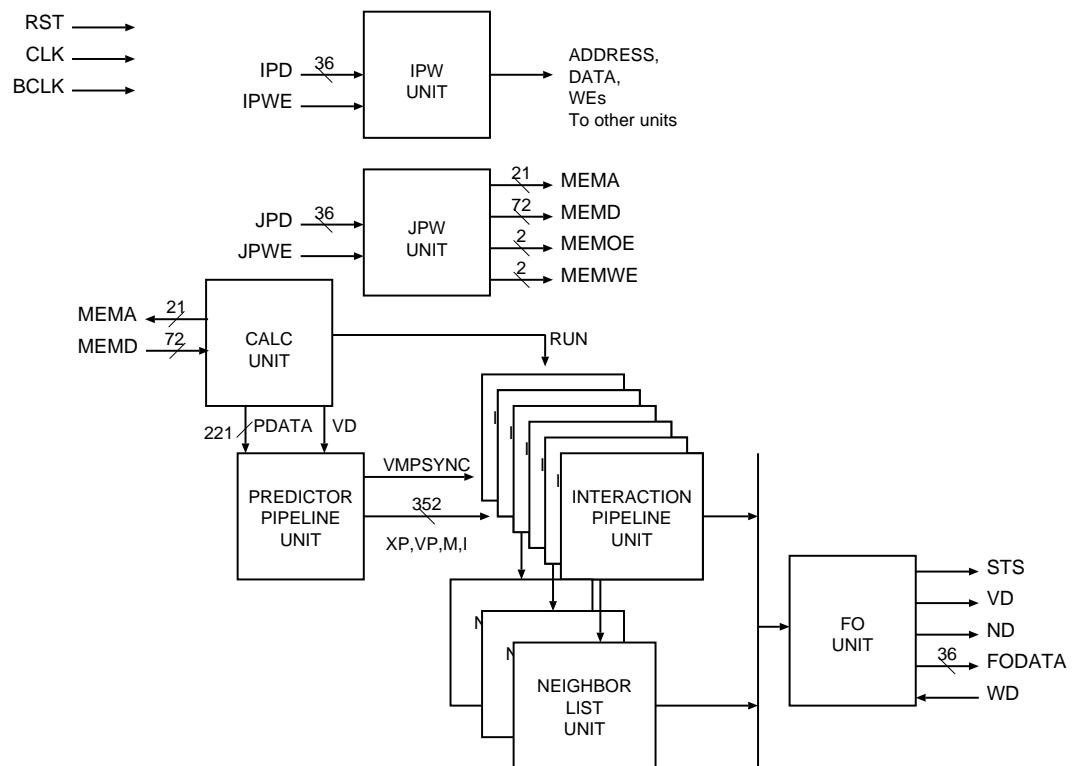


図 7.2: G6 チップのトップ レベル ブロック図

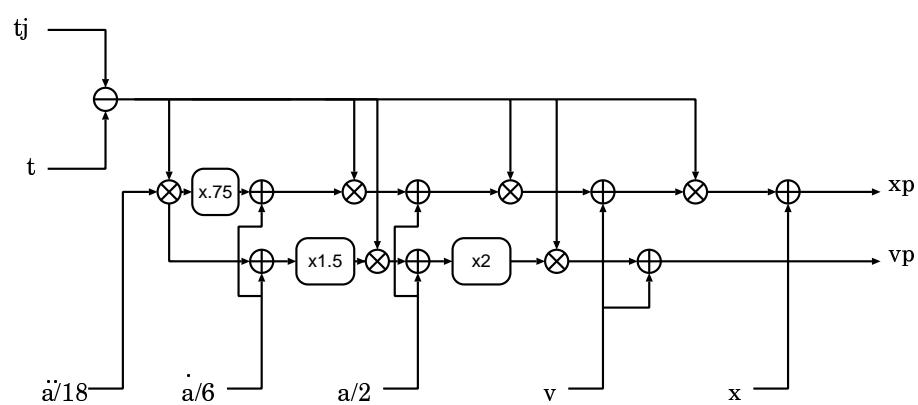
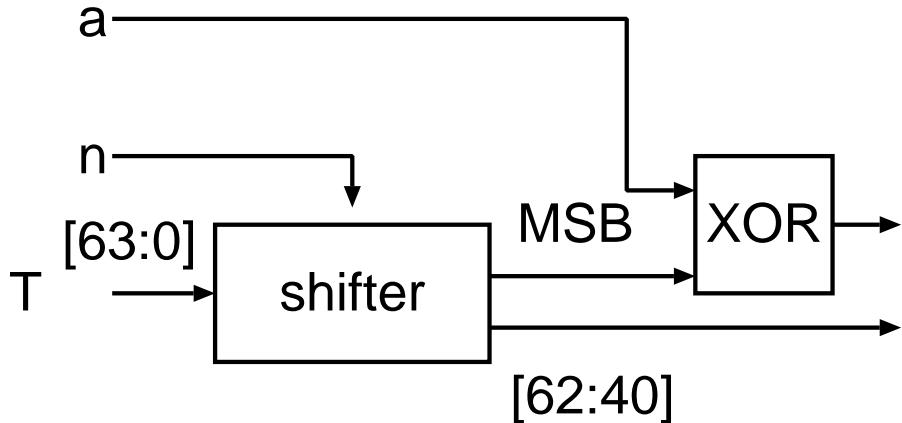


図 7.3: 予測子パイプライン

図 7.4:  $\Delta t$  計算回路

### 7.2.2 a) $DT = T - TJ$

Cでのシミュレータでは、この演算は関数 `tcalc` (`tcalc.c`) に与えられる。

すでに述べたように、実際に引き算を行なう必要はない。現在時刻 `tnow` が 64 ビット固定で与えられ、`tilsb` が粒子の時刻をそのタイムステップで割った値の LSB、`dtimsbloc` がタイムステップの大きさ、すなわち `tnow` と同じところに小数点をおいた時の、MSB の位置を与える。これは、タイムステップが時間分解能に等しい時 0 になる。

結果は 24 ビット固定小数点である。

演算は、まず `tnow` から下 `dtimsbloc+1` ビット分を切りだし、24 ビットを埋めるように左シフトする。さらに、最上位ビットは `tilsb` との排他的論理和をとる。

$T < a : b >$  が固定小数点データ  $T$  のビット範囲  $a$  から  $b$  を表すとすれば、式では以下のように書ける

$$\text{output } < 23 : 0 > = TNOW < DTMSBLOC : DTMSBLOC-23 > \text{exor}(TILSB << 23). \quad (7.1)$$

実際の回路は図 7.4 のようになる。

### 7.2.3 定数乗算器

これらは、加算器と配線で実現する。ここでもオーバーフローは起きないように指数が選ばれているものとする。とする。0.75, 1.5 倍のところでは force-1 丸めが必要になる。仮に 2 の補数表示であるとすると、符号を保ってシフトする必要がある。

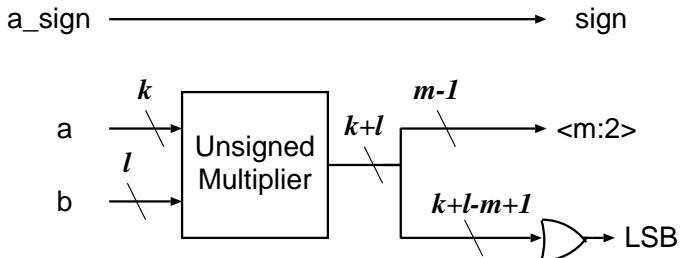


図 7.5: 予測子での乗算器の基本構成

#### 7.2.4 加算器

##### 7.2.5 b) $K4 = DT^*A2BY18$

C のソースでは、これは乗算器を特定の引数で使うことになる。乗算器は pmult\_with\_sign (pmult.c) であり、 predictor の中では

```
pmult_with_sign(&val, dt, a2by18, PRED_A2_LEN, PRED_A2_LEN);
```

の形で使われている。

pmult\_with\_sign は符号をコピーするだけである。

乗算器（実体は pmult.c 中の pmult）では、以下の順に計算を行なう。

まず、DT の桁数を落す。これは乗算器を小さくするためである。このために、DT を corrected force-1 によって丸める。すなわち、丸めた後の DT の上位 11 ビットはもとの DT の上位 11 ビットがそのままコピーされる。LSB は、元の DT の下位 13 ビットすべての OR をとったものである。シミュレータでは force\_1\_round\_and\_shift がこの演算を行なう。

次に実際にかけ算を行なう。DT、A2BY18 は共に 10 ビットなので、答えは 20 ビットになる。ここから上位 12 ビットをとるわけだが、DT は 0x200 (16 進) を超えることはないので、実は結果は必ず 0x80000 より小さい。したがって、MSB を落してその次からの 10 ビットをとる。ここでも、結果の LSB は元の対応する位置とその下すべての OR をとったものである。

##### 7.2.6 c) $K4A = K4 * 0.75$

C のソースは pmult.c の中の pmult\_by\_constant および pmult\_with\_sign\_by\_constant である。後者はサインビットをコピーするだけで、実体は前者である。

ここでは、符号以外を 1 ビット左シフトしたものと加算しさらに 2 ビット右シフトする。ただし、LSB については元の最下位 3 ビットの OR になる。

##### 7.2.7 d) $K3 = K4A + A1BY6$

C のソースは padd.c の中の padd および padd\_with\_sign である。後者はサインビットをコピーするだけで、実体は前者である。

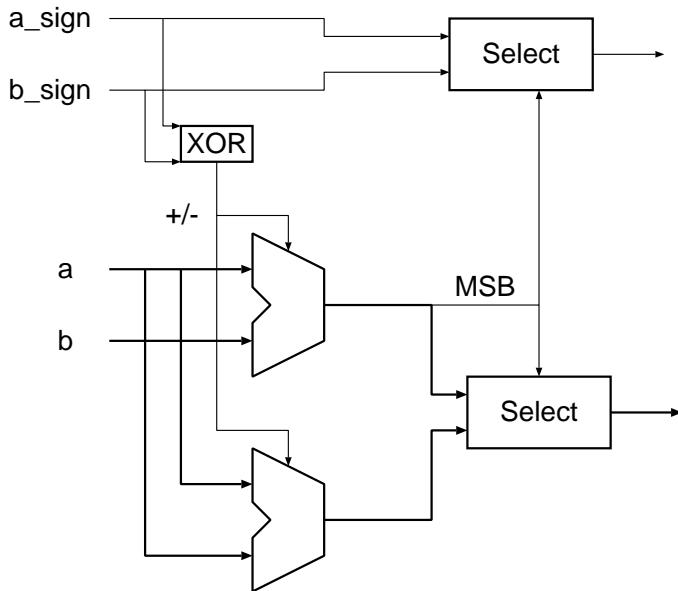


図 7.6: 予測子での加算器の基本構成

ここでは、10ビットのK4Aと16ビットのA1BY6を足し合わせる。符号つきなので、K4A側の入力の上位ビットは0で拡張する。

加算器の一般構成を図7.6に示す。絶対値表現なので、加減算器を2つ使い、2入力  $a, b$  の符号が違う場合には  $a - b$  と  $b - a$  の双方を計算する。結果が正になったものを選べばよい。この時、符号はどちらの演算結果が利用されたかで決まるので、同様に選ぶことができる。

#### 7.2.8 e) $K3A = K3 * DT$

これはビット長16ビットの乗算器である。語長が違う他は(b)の乗算器と全く同一である。

#### 7.2.9 f) $K2 = K3A + ABY2$

ここでは、16ビットのK3Aと20ビットのABY2を足し合わせる。ここも語長が違う他は演算(d)と同じである。

#### 7.2.10 g) $K2A = K2 * DT$

これはビット長20ビットの乗算器である。語長が違う他は(b)の乗算器と全く同一である。

**7.2.11 h)  $K1 = K2A + V$** 

ここでは、20ビットのK2Aと24ビットのVを足し合わせる。ここも語長が違う他は演算(d)と同じである。

**7.2.12 i)  $K1A = K1 * DT$** 

これはビット長24ビットの乗算器である。語長が違う他は(b)の乗算器と全く同一である。

**7.2.13 j)  $XP = X + K1A$** 

Cのソースはpredictor.cの中のpredictのなかである。

ここでは64ビット2の補数表示のXと、符号付非正規化浮動小数点数であるK1Aを足し合わせる。そのために、まず指数部をとりだし、指数分だけシフトしてK1Aを固定小数点に直したあと、加減算器に入れる。シフタの出力は64ビット、入力は24ビットである。入出力の下からkビット目をそれぞれ $I_k, O_k$ ( LSBは $k=0$ )と書くと、指数を符号なし整数として見た値を $p$ とする時入出力の関係は

$$O_k = \begin{cases} I_{k-p+87} & k > 0 \\ I_j & (0 \leq j \leq k-p+87 \text{ の AND}) \\ k = 0 \text{かつ } k-p+87 \geq 0 \end{cases} \quad (7.2)$$

で与えられる。ただし、 $I_k$ は $k > 23$ および $k < 0$ で0と定義する。この出力を、符号によってXに加算または減算する。符号ビットが0なら加算、1なら減算である。

**7.2.14 k)  $K3C = K4 + A1BY6$** 

ここでは、10ビットのK4と16ビットのA1BY6を足し合わせる。ここは演算(d)と同じである。

**7.2.15 l)  $K3D = K3C * 1.5$** 

Cのソースはpmult.cの中のpmult\_by\_constantおよびpmult\_with\_sign\_by\_constantである。後者はサインビットをコピーするだけで、実体は前者である。

ここでは、1ビット左シフトしたものと加算することで3倍し、右1ビットシフトする。この時、LSBはもとの LSB 2ビットのORである。LSBは切捨てる。符号はそのまま。

**7.2.16 l')  $K3E = K3D * DT$** 

これはビット長16ビットの乗算器。丸めはforce-1

**7.2.17 m)  $K2B = K3E + ABY2$** 

ここでは、16ビットのK3Eと20ビットのABY2を足し合わせる。演算(f)と同一である。

### 7.2.18 n) $K2C = K2B * 2$

これは単に配線で1ビット左シフトするだけである。

### 7.2.19 o) $K2D = K2C * DT$

これはビット長20ビットの乗算器であり、演算(g)と同一である。

### 7.2.20 p) $VP = V + K2D$

ここでは、20ビットのK2Dと24ビットのVを足し合わせる。ここは演算(h)と同一である。さらに、結果を正規化し、ゼロフラグを設定する。正規化の際には、leading zerosの分入力の指数から引く。ここで指数を10ビットに拡張し、オフセット512を加える。また、さらに指数からDTJMLSBを引いておく。これは、もともと速度の指数はタイムステップを繰り込んだ形で入力されているからである。

なお、結果として浮動小数点表現は36ビットになるが、これはMSBから以下の順でしまわれる

1. 指数 10 ビット
2. 符号 1 ビット 1なら負
3. ゼロフラグ 1 ビット 1ならゼロ
4. 仮数 24 ビット 非ケチ表現、絶対値

### 7.2.21 チップ内他ユニットとのインターフェース

ここでは、チップ内の他ユニットとのインターフェースを示す。図7.7に信号定義を与える。ここで、位置から第二導関数までは、 $x, y, z$ の3成分を持つことに注意して欲しい。これらは3サイクルで順に渡ってくるものとする。それ以外のデータは最初のサイクルで渡される。

予測子ユニットのデータ入力は表7.1の通りであるが、さらに $t_i$ の入力が必要である。これに加えてさらに制御入力がある。これらを表7.2にまとめる。

なお、VDはデータが有効であることを示す信号（アクティブ・ハイ）であり、一度出たらそのあと7サイクルはでることがないものとする。PASSなんかの一連の信号は、パイプライン動作にあわせてラッチする必要はない。

図7.8に予測子の入力のタイムチャートを示す。

時刻 $t_i$ の入力については、専用のポートからWETI(H/L)信号（アクティブ・ハイ）と同期して書き込むものとする。タイムチャートは図7.9のようになる。H, Lの二本でそれぞれ上位、下位の32ビットに書き込む。

予測子パイプラインの出力は、位置、速度、質量、インデックスである。質量、インデックスは、単にパイプラインレジスタで待ち合わせするだけである。ただし、さらにこれに加えて、VDに相当する「データが揃った」という信号を出す。つまり、出力は表7.3のようになる。

図7.10に予測子の出力のタイムチャートを示す。

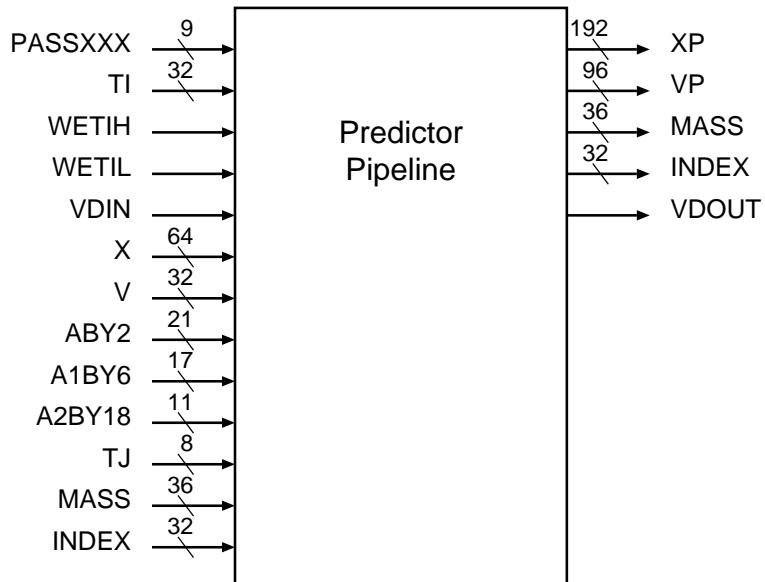


図 7.7: 予測子パイプライン内部インターフェース

表 7.1: 予測子パイプラインへの演算中入力データ

変数	ビット数	個数
位置	64	3
速度	32	3
加速度	21	3
第一導関数	17	3
第二導関数	11	3
時刻	8	1
質量	36	1
番号	32	1
合計	—	511(221)

```

clk      ======-----=====-----=====-----=====-----=====-----=====
VD      -----/=====\'-----=====-----=====-----=====\'-----=====
DATA   -----<datax0><datay0><dataz0>-----<datax1><datay1><dataz1>
  
```

図 7.8: 予測子入力タイムチャート

表 7.2: 予測子パイプラインへの入力信号

入力名	機能
TI	現在時刻 (32ビット)
WETIH/WETIL	現在時刻書き込み信号 (アクティブハイ)
VD(Valid data)	演算用データが揃っていることを示す
PASSK4	K4 に A2BY18 をコピーする
PASSK4A	K4A に K4 をコピーする
PASSK3A	K3A に K3 をコピーする
PASSK2A	K2A に K2 をコピーする
PASSK1A	K1A に K1 をコピーする
PASSK3D	K3D に K3C をコピーする
PASSK3E	K3E に K3D をコピーする
PASSK2C	K2C に K2B をコピーする
PASSK2D	K2D に K2C をコピーする

```

clk      ----====----====----====----====----=
WETIx   -----/=====\
TIDATA  -----<=data==>-----

```

図 7.9: 予測子 TI 書き込みタイムチャート

表 7.3: 予測子パイプライン出力

変数	ビット数	個数
位置	64	3
速度	36	3
質量	36	1
番号	32	1
VD	1	1

```

clk      ----====----====----====----====----====----=
VD      -----/=====\
DATA   -----<=data0>-----<=data1>-----

```

図 7.10: 予測子出力タイムチャート

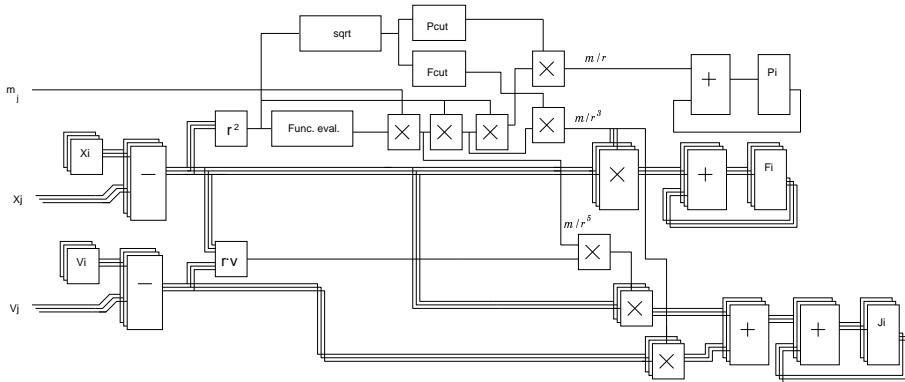


図 7.11: 相互作用パイプライン

### 7.3 相互作用パイプライン

相互作用パイプラインの構成を図 7.11 に示す。計算精度は、位置の初段が 64 ビット、最終段（加速度）が 64 ビットの固定小数点とする。速度は入力が浮動小数点 32 ビット、最終段（加速度の第一微分）は 32 ビットの固定小数点とする。また、ポテンシャルについても 64 ビットの固定小数点で積算するものとする。なお、それぞれ独立に指数バイアスを設定できるものとする。つまり、実効的には積算に関してはブロック浮動小数点ということになる。

なお、いうまでもないが VMP の場合にはこの指数レジスタを独立にもつ必要がある。

中間の演算については、加速度は仮数 24 ビット、指数 10 ビットで行ない、その微分は仮数 20 ビット、指数 10 ビットで行なうことにする。内部表現で指数を多めにとってもそれほどゲート数には影響しないので、ややこしい問題を防ぐために指数はここでは多めにとっておく。テストビリティに影響がでたら減らすこととも考える。

浮動小数点表現はゼロフラグ、符号ビットを合わせて 36 ビットになる（仮数 24 ビットの場合）が、これは MSB から以下の順でしまわれる

1. 指数 10 ビット
2. 符号 1 ビット 1 なら負
3. ゼロフラグ 1 ビット 1 ならゼロ
4. 仮数 24 ビット 非ケチ表現、絶対値

また、周期境界条件のもとでの計算を Ewald 法により高速に行なうため、12 ビット程度の絶対精度でカットオフ関数を計算する仕掛けをつける。これはポテンシャルと加速度のみにつけ、導関数にはつけない。これは、周期境界で非常に高精度の計算をするということはとりあえず考えないためである。（これは後で変更する可能性がある）

以下に相互作用パイプラインでの計算の Fortran ソースを示す。なお、これにはカットオフ演算は含まれないことに注意して欲しい。

```
DO 15 J = 1,N
    dr2 = EPS2
    drdv = 0.
```

```

DO 11 K = 1,3
    dx(K) = X(K,J) - X(K,I)
    dv(K) = XDOT(K,J) - XDOT(K,I)
    dr2 = dr2 + dx(k)**2
    drdv = drdv + dx(k)*dv(k)
11    CONTINUE
*
dr2i = 1.0/dr2
dr3i = BODY(J)*dr2i*SQRT(dr2i)
drdv = 3.*drdv*dr2i
*
DO 12 K = 1,3
    Firr(K) = Firr(K) + dx(K)*dr3i
    FD(K) = FD(K) + (dv(K) - dx(K)*drdv)*dr3i
12    CONTINUE
15 CONTINUE

```

### 7.3.1 a) $dx(K) = X(K,J) - X(K,I)$

Cでのシミュレータでは、この演算は関数 subx (subx.c)に与えられる。

入力は64ビット固定小数点、出力は仮数24ビットの浮動小数点である。出力はゼロフラグ、符号、指数6ビットを合わせた合計32ビットとなる。ただし、めんどうなのでここで指数は10ビットに拡張しておく。なお、仮数24ビットへの丸めは修正 force-1とする。また、指数はバイアスフォーマットとする。

まず入力同士の引き算をし、結果の絶対値と符号をとる。次にMSBの位置をプライオリティエンコーダで決める。これは、0でない最上位ビットの値を返すものであり、LSBだけが1なら0、一般に下からkビット目( $k < 63$ )が1でそれより上がすべて0ならkを返す。

次に、結果の正規化をする必要がある。これには、まず63ビット入力63ビット出力の左論理シフタを使う。シフト量は63から先のエンコーダの出力を引いたものである。

このシフタ出力の上位23ビットはそのまま出力の仮数の上位23ビットとなる。出力仮数のLSBはシフタ出力下位40ビットすべてのORである。

なお、引き算の結果が0であればゼロフラグをセットし、そうでなければフラグは0である。符号は正なら0である。

指数は、プライオリティエンコーダの出力(6ビット)の上に1000を付け加えて10ビットにしたものということになる。

### 7.3.2 b) $dx2(K) = dx(K)*dx(K)$

Cでのシミュレータでは、この演算は関数 mult (mult.c)に与えられる。

これは通常の24ビット仮数の浮動小数点乗算器である。ここから指数を10ビットに拡張する。

出力符号は入力符号の排他的論理和である。ゼロフラグはANDである。

指数は2つを足したものにさらに512を加え、下位10ビットをとる。

仮数はまず 24 ビットの符号なし乗算器に入れる。この出力は 48 ビットとなる。結果の MSB が 0 なら左 1 ビット論理シフトし、指数から 1 を引く。

さらに下位 25 ビットの OR をとり、これを出力仮数の LSB とする。出力仮数の上位 23 ビットはのこり 23 ビットをそのまま使う。テストモード入力 T05 により乗算しないで入力をそのまま出す。

### 7.3.3 c0)dr2xy = dx2(1) + dx2(2)

C でのシミュレータでは、この演算は 関数 add (add.c) に与えられる。

とりあえず force-1 丸めによる加算を行なう。

まず、入力 2 つのうちのどちらが大きいかを判断する。2 つの入力を in1, in2 と書くことにする。in1 の指数、符号、ゼロフラグと仮数をそれぞれ exp1, sign1, zero1, mantissa1 と書き、in2 についても同様とする。in2 が大きいとしてよいのは以下の場合である。

- zero1 が 1
- exp1 < exp2 であり zero2 が 1 でない
- exp1 = exp2 であり、zero2 が 1 でなく mantissa2 > mantissa1 である

こののどれかが成り立っていれば、2 つの入力を入れ換える。

このあと、まず、出力の符号とゼロフラグは in1 のものになる。

次に仮数の演算を行なう。まず、mantissa2 を exp1 – exp2 だけ右論理シフトする。ただし、この時も結果の LSB は切り捨てられる分すべてと LSB の OR をとる。

これを mantissa1 に加える。sign1 と sign2 が違えば減算になる。この結果は桁上がりも含めて 25 ビットの符号なし整数である。

プライオリティエンコーダで最上位の 1 の位置を決める。これが LSB なら 0、MSB なら 24 が出力になるものとする。この値を  $p$  とする。また、仮数が 0 なら出力のゼロフラグを 1 にする。

出力の指数は、 $\exp1 - 23 + p$  となる。

仮数は、まず入出力 25 ビットの左論理シフタに入れる。シフト量は  $24 - p$  である。出力の上位 23 ビットはそのままであり、下位 2 ビットは OR をとって結果の LSB とする。

なお、注意して欲しいのは、オーバーフロー、アンダーフローに対して特別な処理は行なわないということである。このため、指数が上とか下に溢れたら結果は正しくならない。が、指数が大きめにとってあるのでそのようなことは原理的に起きないのである。これは、最終段以外のすべての演算でそうなっている。

### 7.3.4 c1)dr2ze = dx2(3) + eps2

(c0) と同様。

### 7.3.5 c2)dr2 = dr2xy + dr2ze

(c1) と同様。

### 7.3.6 d) $r5inv = dr2^{**}2.5$

Cでのシミュレータでは、この演算は関数 power\_function (power\_functions.c) に与えられる。

- 指数は演算してベースの指数を求める。
- 近似多項式のテーブルは、0次の項については指数を出す。この指数は前項のベース指数に加えられる
- 最大1ビット多項式の計算結果はシフトされる。この時さらに指数は加算される。これはキャリーに入れれば余計なインクリメンタはいらない。

なお、負ベキの計算なので係数の正負は項毎に変わる。また、高次の項が小さいことは保証されている。特に、補間の結果符号が変わることはない。したがって、それらの性質を利用した回路ということになる。

指数については、以下のような演算を行なう。まず1を引き、次に1ビット右論理シフト (LSB切捨て) した後256を引く。さらに符号を反転したあと5を掛ける、すなわち左2ビット論理シフトしたものと加える。そのあととりあえず513を足しておく。

係数テーブルはエントリーが512である。テーブルへの入力アドレスは、指数の LSB と仮数の上位8ビット (MSB を除く) を合わせたものである。出力は、0, 1, 2次がそれぞれ24, 18, 12ビットである。また、指数を調節する必要があるので、そのテーブルも必要である。これは3ビットである。2次補間であるので、 $(k_2\Delta x + k_1)\Delta x + k_0$  の形の計算になる。 $\Delta x$  は入力仮数の下位15ビットである。 $k_2$  は12ビットなので、かけ算の結果は27ビットになる。この上位12ビットをとって LSB を合わせて  $k_1$  から引く。この結果にさらに  $\Delta x$  を掛け、33ビットの結果から上位18ビットをとって、これを  $k_0$  から引く。

減算まえの  $k_1$  は必ず MSB が1になっているが、減算の結果桁下がりが起きることがある。MSB が0になっていたら、仮数を左に1ビット論理シフトすると共に、指数から1を引く。指数からはさらに指数テーブルの出力を引く。

テーブルの内容は、関数 prepare\_table (power\_functions.c) によって生成される。これを、引数 (5,512,24) でコールする。

### 7.3.7 e) $mr5inv = r5inv*m$

仮数24ビットの浮動小数点乗算器である。丸めは force-1。テストモード入力 T08 によって、r5inv を1に固定する。

なお、「質量」とはいえ計算対象によっては負になることもあるので、符号を省略してはならない。

### 7.3.8 f) $mr3inv = mr5inv*dr2$

仮数24ビットの浮動小数点乗算器である。丸めは force-1。テストモード入力 T02, T03 によって、それぞれ mr5inv と dr2 を1に固定する。

### 7.3.9 g) $mrinv = mr3inv*dr2$

仮数24ビットの浮動小数点乗算器である。丸めは force-1。テストモード入力 T11 により dr2 を1に固定する。

### 7.3.10 h) $r = \sqrt{dr^2}$

C でのシミュレータでは、この演算は 関数 `grapw_low_acc_distance` (`low_acc_power.c`) に与えられる。まず仮数を 12 ビットに丸めた後、テーブル参照と線形補間で平方根を計算する。

この実現は  $1/r^5$  を計算するところと方式は同じであるが、テーブルサイズ、ビット幅、次数などが違う。以下に実現の詳細をまとめる。

指数については、以下のような演算を行なう。まず 1 を引き、次に 1 ビット右論理シフト (LSB 切捨て) した後 256 を引く。そのあととりあえず 513 を足しておく。ここで、256 をひいてから 513 を足すのは、もちろん 257 を足すのと同じである。

係数テーブルはエントリーが 64 である。テーブルへの入力アドレスは、指数の LSB と仮数の上位 5 ビット (MSB を除く) を合わせたものである。出力は、0, 1 次がそれぞれ 12, 6 ビットである。なお、平方根の場合指数を調節する必要はないので、このテーブルは不要である。1 次補間であるので、 $k_1 \Delta x + k_0$  の形の計算になる。 $\Delta x$  は入力仮数の下位 6 ビットである。 $k_1$  は 6 ビットなので、かけ算の結果は 12 ビットになる。この上位 6 ビットをとって LSB を合わせて  $k_0$  に足す。この結果桁上がりが起きることはない。

テーブルの内容は、関数 `prepare_table` (`low_acc_power.c`) によって生成される。これを、引数 (1,64,12) でコールする。

### 7.3.11 i0) $pcut = pf(r)$

C でのシミュレータでは、この演算は 関数 `cutoff_function` (`cutoff_functions.c`) に与えられる。

これは、Ewald 法のためのカットオフ関数 (ポテンシャル用) を評価する。 $r^2$  に適当な係数を掛けたあと、固定小数点に変換しエントリー 6 ビットの RAM テーブルと線形補間器によって構成する。

実際の変換は以下のよう手順をとることになる。

- (1) まず、 $r$  にスケールファクター  $1/r_0$  を掛ける。これは全粒子、パイプラインで共通とする。
- (2) 次に、これを固定小数点化する。要するに 0 から 1 までの値にするだけで、それよりも大きい場合は補間して出来る最大値の値をとるものとする。
- (3) 固定小数点化した内の上位 6 ビットをテーブルに入れて、0 次と 1 次の項を出す。0 次は固定小数点 13 ビット、1 次は 12 ビットで出す。1 次の項は、距離の下位 6 ビットと掛けて、上 12 ビットをとって 0 次の項に加える。なお、1 次の項には符号をつける。
- (4) 出た答を浮動小数点に直す。

このユニットのうちステップ 3 以降はポテンシャルと力のために独立に 2 セット準備する。ステップ (1) のスケールファクターとのかけ算は、仮数 12 ビットの浮動小数点乗算器を使う。これは他のところと同様に作る。

ステップ (2) の固定小数点化のためには、まず指数から 512 を引く。その後で指数が正ならば、固定小数点で表現できる範囲を超えてるので、結果を 0xFFFF とする。逆に指数が負で絶対値が 12 より大きければ、アンダーフローしているので結果を 0 とする。それ以外の場合は、指数の絶対値の分だけ仮数を右論理シフトする。

なお、ゼロフラグがセットされていれば、結果を強制的に 0 にする。

表 7.4: 積算器のステータスフラグ

名称	ビット位置	説明
SOVFL	0	シフタでのオーバーフロー
AOVFL	1	積算器でのオーバーフロー
SUNFL	2	シフタでのアンダーフロー

ステップ(3)のテーブルロックアップと補間は上に書いた通りである。これは cutoff\_function (cutoff\_function.c) がやっている。テーブルの中身は、関数 prepare\_cutoff\_table (cutoff\_function.c) に関数ポインタを渡すことで生成される。

ステップ(4)の結果の正規化は、util.c のなかの convert\_fixed\_to\_grape\_float が行なっている。まず入力の最上位の 1 の場所をプライオリティエンコーダで決め、そこが MSB になるように左シフトする。指数は、512 にシフト量を足し、12 を引いてもとまる。また、入力が 0 ならゼロフラグをセットする。

#### i1) $\text{fcut} = \text{ff}(\mathbf{r})$

ここは (i0) と同じである。

#### 7.3.12 j0) $\text{mrinv2} = \text{mrinv} * \text{pcut}$

ポテンシャルのカットオフ関数と  $m/r$  を掛け合わせる。ここは、結果の精度は 24 ビットを維持する必要があることに注意してほしい。このため、乗算は、入力の仮数が 12 ビットと 24 ビットで異なる形の浮動小数点乗算器となり、結果の仮数が 24 ビットとなる。これには、 $24 \times 24$  の乗算器ではなく  $12 \times 24$  の乗算器をとり、 LSB を出力の下位 13 ビットの OR とする。テストモード入力 T06 により mrinv を 1 にする。

#### 7.3.13 j1) $\text{mr3inv2} = \text{mr3inv} * \text{fcut}$

ここは (j0) と同じである。テストモード入力 T07 により mr3inv を 1 にする。

#### 7.3.14 k) $\text{phi} = \text{phi} + \text{mrinv2}$

C でのシミュレータでは、この演算は 関数 fadd (fadd.c) に与えられる。

前述したように、ここは 64 ビット固定小数点で積算する。指数のオフセット値はレジスタに記憶され、それと入力の指数の差だけシフトしてから加算する。このシフトの際にも、force-1 丸めを行なうものとする。

なお、ステータスピットをつけてオーバーフローを検出できるようにする。これは積算を始めた時にクリアされる。仮想パイプライン毎に独立なフラグが必要であることはいうまでもない。

表 7.4 に示す 3 ビットのフラグとなる。

これらのフラグは以下の時にセットされるものとする。

- SOVFL: シフトした結果の絶対値が 63 ビット（ポテンシャル、加速度の場合）または 31 ビット（jerk の場合）に収まらない時。この時、シフトした結果は強制的に 0 にされる。
- AOVFL: 加算器がオーバーフローした、すなわち、キャリが符号まで伝わった時にセットされる。
- SUNFL: シフトした結果の絶対値が 0 になる時。

これらのフラグと、積算レジスタ自体の値は、計算開始 (RUN 信号の立ち上がり) でクリアされ、立ち下がったあとはホールドされる。

なお、指数の演算とフラグの処理は以下のように行なうこととする。

指数とオフセットは 10 ビットの数として加算し、11 ビット目は見ない。で、結果の 10 ビット目が立っていれば、負になったと信じる。つまり、SUNFL をたてる。

他の積算器も同様であるが、メモリから来るインデックスが自分と一致していたら、積算をスキップする必要がある。

#### 7.3.15 l) $f(k) = mr3inv2 * dx(k)$

24 ビット仮数の浮動小数点乗算器である。丸めは force-1。テストモード入力 T01 および T04 によって、入力の  $mr3inv2$  および  $dx(k)$  を強制的に 1 にする。

#### 7.3.16 m) $acc(k) = acc(k) + f(k)$

ポテンシャルの積算と同一のものである。指数のオフセットはポテンシャルのものとは違うが、加速度の 3 成分で共通である。

#### 7.3.17 n) $dv(K) = XDOT(K,J) - XDOT(K,I)$

仮数 24 ビットでの減算である。

なお、結果は仮数 20 ビットに丸める。

#### 7.3.18 o) $xv(k) = dx(k) * dv(k)$

仮数 20 ビットでの乗算である。

#### 7.3.19 p) $drdvxy = xv(1) + xv(2)$

仮数 20 ビットでの加算である。

#### 7.3.20 q) $drdrv = drdvxy + xv(3)$

仮数 20 ビットでの加算である。

### 7.3.21 r)drdv3 = drdvxy\*3

定数倍である。仮数を1ビット左シフトして3倍する。結果の25ビット目(LSBを0として)が0であれば、さらに1ビット左シフトする。これで結果は26ビットになっているので、24ビットになるようにforce-1で丸める。指数は、2増やすが、上で1ビットシフトが起こっていれば1だけ増やす。

テストモード入力T14により入力をそのままだす。

### 7.3.22 s)mdrdv3byr5 = drdv3 \* mr5inv

仮数20ビットでの乗算である。mr5invは掛ける前に指数をforce-1で丸める。

テストモード入力T12, T15によりmr5inv, drdv3を1に固定する。

### 7.3.23 t) t1(k) = dv(k)\*mdr3inv

仮数20ビットでの乗算である。dv, mdr3invはあらかじめforce-1で仮数20ビットに丸める。

### 7.3.24 u) t2(k) = dx(k)\*mdrdv3byr5

仮数20ビットでの乗算。dxはあらかじめforce-1で仮数20ビットに丸める。

### 7.3.25 v) j(k) = t1(k) + t2(k)

仮数20ビットでの加算。テストモード入力T13によりt1を0に固定する。

### 7.3.26 w)jerk(k) = jerk(k) + j

固定小数点32ビットに拡張してから加算。実装はポテンシャルの積算器と同様である。なお、ここもステータスピットをつけてオーバーフローを検出できるようにする。

### 7.3.27 x) ネイバーリスト

実パイプライン2本毎に48ビット256ワードのメモリをつける。距離の2乗とレジスタに入っている定数 $h^2$ を比較し、 $h^2$ が大きければフラグをたてる。ただし、メモリから来る粒子インデックスと自分のインデックスを値を比較し、一致していればフラグをたてない。これは、自分自身を除くためである。

このフラグの8サイクル(仮想/実比)分をシリアル/パラレル変換して16ビットのフラグ出力を得る。これがすべて0なら何もしない。どれか一つでも1であれば、フラグと、対応する粒子番号(外付けメモリユニットから供給される)をメモリに書き込み、アドレスカウンタをインクリメントする。なお、アドレスカウンタが255になっていたら、オーバーフローフラグをセットし、そのあと書き込みは行なわない。

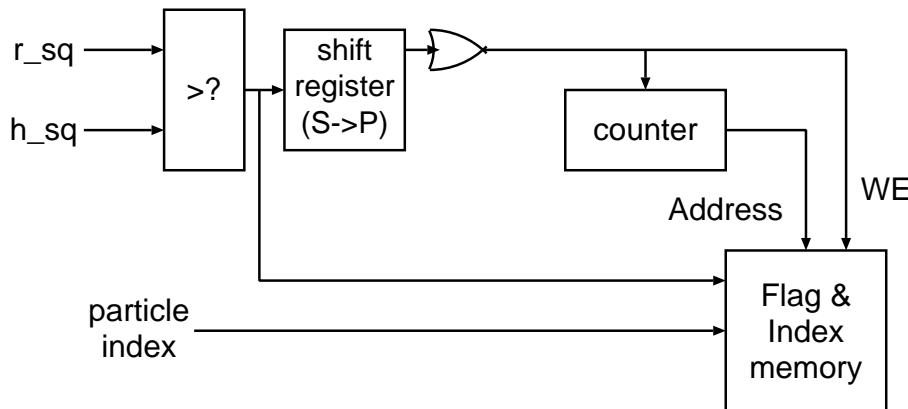


図 7.12: ネイバーリスト

図 7.12 に概念図を示す。ここでは物理パイプライン 1 本分の回路になっているが、実際には 2 個の比較器とシフトレジスタの出力の OR がとられることになる。また、 $h^2$  は仮想パイプライン毎に違う値をとるので、ここはリングレジスタまたはレジスタファイルで実現される必要がある。

メモリを読む時には、偶数アドレスのときオーバーフローフラグ + 16bit のネイバーフラグの合計 17 ビット、奇数アドレスのときに 32 ビットの粒子インデックスが見えるようになる。

### 7.3.28 y) ニアレストネイバー

図 7.13 に概念図を示す。なお、ここでも  $r_{nn}$  および  $i_{nn}$  は実際には仮想パイプライン分のレジスタファイルまたはリングレジスタである必要があることに注意する必要がある。ただし、メモリから来る粒子インデックスと自分のインデックスを値を比較し、一致していれば距離、インデックスを更新しない。これは、自分自身を除くためである。

なお、これは、相互作用パイプライン用テストデータでは RMIN の名前で与えられているものである。

### 7.3.29 チップ内入出力インターフェース

図 7.14 に信号定義を与える。

レジスタへの書き込みについては、どういう風に実装するかが問題だが、その面倒は相互作用パイプラインのほうで面倒をみるということにして、制御部のほうからはアドレス、データ、WE 信号ができるということにする。

レジスタへからの読みだしについても、制御部のほうからはアドレスがでて、相互作用パイプラインからは固定したディレイの後にデータが出るということにする。

以下、それについてまとめる。

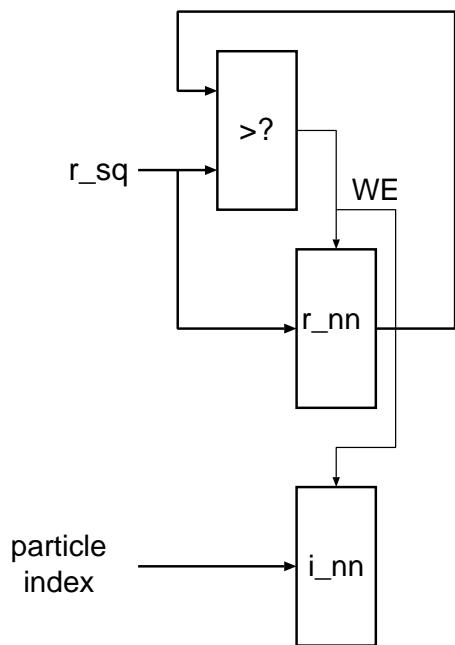


図 7.13: 最近接粒子

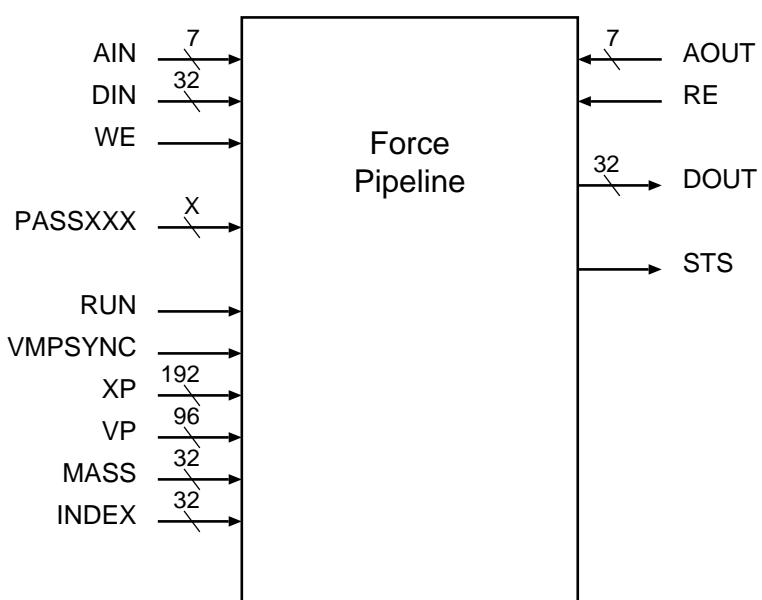


図 7.14: 相互作用パイプライン内部インターフェース

表 7.5: 予測子パイプラインからの入力

変数	ビット数	個数
位置	64	3
速度	32	3
質量	32	1
番号	32	1
VMPSYNC	1	1

図 7.15: 予測子からの入力のタイムチャート

### 予測子からの入力

予測子パイプラインからの入力を表 7.5 に示す。これは表 7.3 と同一であるはずである。図 7.15 に予測子からの入力のタイムチャートを示す。これは図 7.10 と同じはずである。

### レジスタ書き込み

すでに述べたように、制御部と相互作用パイプラインとのインターフェースでは、パイプライン内のレジスタについては相互作用パイプライン側でデコードすることにする。従って、インターフェース信号は表 7.6 のようになる。

タイミングは同期型であり、すべての信号をクロック立ち上がりでサンプルする。WE がハイであるときには指定したアドレスへの書き込みが起きる。そうでなければ何も起こらない。なお、AI, WEI は  $i$  粒子データの、AT, WET はカットオフテーブルの書き込みに使う。これらで別の信号を使うのは、制御部から別にアドレスができるからである。

表 7.3.29 に相互作用パイプライン一つのなかのレジスタマップを示す。上位 3 ビットは仮想パイプラインの ID であり、下位 4 ビットが物理レジスタの区別になる。実装としては、違

表 7.6: 制御部-相互作用パイプ書き込みインターフェース信号定義

信号	幅	説明
DIN	36	書き込むデータ
AI	7	アドレス
AT	7	アドレス
WEI	1	書き込み信号 (アクティブハイ)
WET	1	書き込み信号 (アクティブハイ)

表 7.7: 相互作用パイプライトレジスタマップ

データ	アドレス (2進)	説明
XH	BBB0000	I 粒子 x 座標
XL	BBB0001	I 粒子 x 座標
YH	BBB0010	I 粒子 y 座標
YL	BBB0011	I 粒子 y 座標
ZH	BBB0100	I 粒子 z 座標
ZL	BBB0101	I 粒子 z 座標
VX	BBB0110	I 粒子速度 x 成分
VY	BBB0111	I 粒子速度 y 成分
VZ	BBB1000	I 粒子速度 z 成分
EPS2	BBB1001	ソフトニング
H2	BBB1010	ネイバー半径
I	BBB1011	粒子番号
SCALES	BBB1100	PHISCALE, FSCALE, JSSCALE
R0	10000000	カットオフスケールファクタ
CTAB	0YXXXXXX	カットオフテーブル

う仮想パイプの同じデータは物理的には一つのレジスタファイルなりメモリユニットに存在することには注意して欲しい。

ポートのデータ幅は 36 ビットである。64 ビットのデータは 2 語に分けて書かれる。また、LSB 側 32 ビットにデータを埋める。連續してデータがくるとかそういう保証はない。

なお、すこしややこしいが、cutoff テーブルは仮想パイプで共通でいい。さらに、物理パイプ間ではもちろん違うものをもつが、書き込むポートは共通であり、同時に書かれるということにする。このため、cutoff テーブルのアドレスは物理的に別であるということにする。テーブルはポテンシャルと加速度のために 2 つあるので、表の Y でこれを区別する。ただし、R0 は共通であることに注意すること。Y が 0 のものがポテンシャル、1 のものが力である。

CTAB は、1 ワードにデータ (0 次、1 次、1 次の符号) をパックする。ビット位置はそれぞれ (25,13), (12,1), (0) ( LSB を 0 として ) となる。それより上のビットは無視される (これは名村さんの実装から書いた)。

PHISCALE, FSCALE, JSSCALE は、I の次のアドレス、すなわち BBB1100 の位置に、10 ビットずつ下から詰めて 1 語にまとめる。順番は、MSB のほうから PHISCALE, FSCALE, JSSCALE とする。

### レジスタ読みだし

書き込みの場合と同様、相互作用パイプライン側でデコードを行なう。インターフェース信号は表 7.8 のようになる。

タイミングは同期型であり、アドレスをクロック立ち上がりでサンプルする。アドレスから固定した遅れでデータができる。ただし、計算中 (STS がハイ) の時に返るデータは嘘でも

表 7.8: 制御部-相互作用パイプ読み出しインターフェース信号定義

信号	幅	説明
DOUT	32	データ
AOUT	7	アドレス
DNB	32	ネイバーデータ
ANB	10	ネイバーアドレス

図 7.16: 相互作用パイプライン入力データタイムチャート

かまわない。表 7.9 に相互作用パイプライン一つのなかのレジスタマップを示す。BBB で示されたビットは仮想パイプラインの ID であり、下位 4 ビットが物理レジスタの区別になる。なお、ネイバーリストについては 2 つの物理パイプにつき一つしかないので、NNB, NBL はそれらだけのための独立なポートを持つことになる。

ポートのデータ幅は32ビットである。RNBは36ビット形式なので溢れるが、これは下4ビットを切捨てて返すものとする。

表 7.10 にステータスレジスタの中身をしめす。

ネイバーリストは 256 粒子分用意するが、1 ワードが 32 ビットを超えるのでアドレスは 512 語分必要となる。偶数ワードの下の方にフラグを入れる。語数として返るのは 256 までであるとする。

動作制御入力

動作制御入力には表 7.11 がある。テスト入力は別に詳細を述べる。ここでは、計算開始／停止信号と VMP 同期信号の関係について述べる。

VMPSYNC は予測子パイプから供給され、 $j$  粒子の有効なデータがあることを示す。実際に計算が行なわれるのは、RUN 信号がハイでかつ VMPSYNC がハイであった時ということにする。すなわち、表 7.16において、RUN は VMPSYNC とは非同期に上がるかもしれない。

相互作用パイプラインは、RUN の立ち上がり（立ち上がりの次の VMPSYNC）で積算レジスタ、ステータスフラグをクリアする。積算レジスタについては、実際には、最初のデータによる計算結果がアキュムレータに来た時に、加算器のもう片方の入力を強制的に 0 にする。ステータスフラグについても、同様に最初に 0 にすることになる。

RUN が下がったら、VMPSYNC で値が来ても、演算結果を積算しないでアキュムレータの値を保持する。図 7.16 の場合では、サイクル 25 に次のデータがくるが、これによる演算結果は積算されないことになる。

表 7.9: 制御部-相互作用パイプリードレジスタマップ

データ	アドレス（2進）	説明
FXH	000BBB0000	I 粒子加速度 x 座標（上位）
FXL	000BBB0001	I 粒子加速度 x 座標（下位）
FYH	000BBB0010	I 粒子加速度 y 座標
FYL	000BBB0011	I 粒子加速度 y 座標
FZH	000BBB0100	I 粒子加速度 z 座標
FZL	000BBB0101	I 粒子加速度 z 座標
POTH	000BBB0110	ポテンシャル
POTL	000BBB0111	ポテンシャル
JX	000BBB1000	I 粒子 jerk x 成分
JY	000BBB1001	I 粒子 jerk y 成分
JZ	000BBB1010	I 粒子 jerk z 成分
RNB	000BBB1011	最近接粒子距離
INB	000BBB1100	最近接粒子番号
STS	000BBB1101	パイプライン演算ステータス
NNB	1000000000	ネイバーリスト長さ
NBL	0XXXXXXXXX	ネイバーリストデータ

表 7.10: 相互作用パイプステータスレジスタの内容

データ	ビット位置	説明
STSFX	0:2	加速度 X 成分
STSFY	3:5	加速度 Y 成分
STSFZ	6:8	加速度 Z 成分
STSJX	9:11	ジャーク X 成分
STSJY	12:14	ジャーク Y 成分
STSJZ	15:17	ジャーク Z 成分
STSP	18:20	ポテンシャル成分

表 7.11: 相互作用パイプへの制御信号

入力名	機能
RUN	積算開始／停止信号
VMPSYNC	VMP 同期入力
T01-T15	テスト用制御

表 7.12: 内部シーケンサー一覧

名前	機能
JPW	JP からのデータをメモリに書き込む
IPW	IP からのデータをチップ内の適当な場所に書き込む
FO	FO からデータを出す
CALC	計算開始／終了の制御をする。さらに 計算中にメモリからデータを読み込み、パイプライン側に供給する。

### 動作制御出力

一応、RUN が入っている間、および RUN が落ちてからも実際に最後のデータの積算が終るまでのあいだ、STS 出力線をハイにしておく。

## 7.4 制御回路

### 7.4.1 クロック

まず、G6 チップ、メモリ、インターフェース回路のクロック系について述べる。応動作速度として 100 ないし 150 MHz を想定する。この場合、外部回路の速度は内部の 1/4 から 1/6 とし、メモリの動作速度は G6 チップと常に同一とする。

クロック供給系としては、以下の 2 通りを想定する。

- G6 チップには速いクロックと遅いクロックの両方を供給する。これらは同期している必要があるが、それは外で面倒をみる。
- G6 チップには遅いクロックだけを供給する。速いクロックは内部で倍増して作り、メモリにも G6 チップから供給する。

とりあえず、メモリには差動 PECL クロック、LVTTL インターフェースのもの（例えば IBM043641RLAB）を想定しておく。また、G6 チップのクロックは、PLL がなくても構わないものとしておく。

さて、外部クロックは、一応内部の 1/6 と 1/4 の 2 通りの場合を考えることにする。これはちょっと面倒な気もするが、150MHz 程度で回った時にそれをいかせないのであまりに残念であるので、1/6 で動作することも可能にする。このために、内部シーケンサが分周比を仮定しないようにする。

### 7.4.2 内部シーケンサ

内部シーケンサには表 7.12 のようなものがある。以下これらについてその動作をまとめると。

### 7.4.3 JPW

表 7.13: JP ポートインターフェース信号定義

信号	幅	説明
JPD	36	データ
JPWE	1	書き込み信号 (アクティブハイ)

```

clk      -----
WE       ======\-----/=====
data    -----<=data0==>-<=data1==>-----

```

図 7.17: 入力ポートタイムチャート

JP からのメモリ書き込みでは、ホストからのネットワークはデータを素通して放送する。JP ポートでは、来たデータをコマンドパケットと解釈する。最初の 2 ワードがコマンドとそのパラメータ、その後に 16 ワードのデータ（一粒子分）がくる。なお、データは外部クロックに同期するが、連続してくるとは限らない。

表 7.13 に JP ポートの信号定義を示す。入力データは 32 ビット + バイトパリティの 36 ビットとする。

図 7.17 に write アクセスのタイミングを示す。

WE が下がっている時には有効なデータが来ている。

表 7.14 にコマンド形式をしめす。コマンドは先頭ワードの最上位 2 ビットが 0 の時に粒子データ、それ以外は予約コードとする。必要なデータは、メモリ上での先頭アドレス（粒子番号）、チップ番号、チップ番号のどこまでを見るかのフィールドの 3 種である。メモリアドレスに 21 ビット、チップ番号およびマスクにそれぞれ 10 ビットわりあてる。

JPW の基本的な機能は、18 ワードのパケットを受けとり、自分のチップ番号と一致していたら指定されたアドレスからの連続領域に書き込むことである。表 7.15 にメモリへのインターフェース信号を与える。

以下、データ取り込みと受けとったデータを書くシーケンサの実現について簡単にまとめる。

まず、外からのデータ取り込みについて考えてみる。要するに、問題は単にいつデータをサンプルするかということである。基本的には、外部クロックの後半の、安定したところで取り込めばよい。で、WE が下がっていれば、有効なデータとしてラッチすると同時に、メモリに書く方にデータが来たことを知らせる。実現例を図 7.19 にしめす。

表 7.14: JP ポートコマンドフィールド

フィールド名	先頭／第二	ワード内位置	説明
コマンドコード	先頭	[31:30]	コード
チップ番号マスク	先頭	[19:10]	
チップ番号	先頭	[9:0]	
メモリアドレス	第二	[20:0]	

表 7.15: JPW メモリインターフェース

信号	幅	説明
A	21	アドレス
D	72	データ
OE	2	出力イネーブル
WE	2	書き込みイネーブル

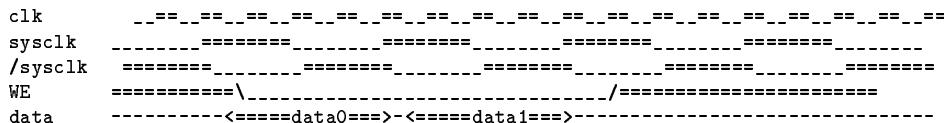


図 7.18: 入力ポート詳細タイムチャート

ENABLE 線で外部データをラッチすることになる。この ENABLE 線は、有効なデータをラッチしたときにだけハイになる。

図 7.20 に JPW シーケンサ全体の信号定義を与える。

A, D, WE は上の制御入力であり、PHYSID はチップの物理 ID, JPWE, JPD は JP ポートからの入力である。MEMA, WE, OE, MEMD は外部メモリへのアドレス、データ、ライトイネーブル制御線、出力イネーブル制御線である。EN は計算を管理するシーケンサから来るもので、WE, OE と MEM のハイインピーダンスを制御する。すなわち、EN が負（ここは正論理）であれば、仮に J 粒子書き込みデータが来ても無視して、WE, OE を出さず、データも出さない。

表 7.16 に JPW の制御レジスタのアドレスマップをしめす。これらの制御データはすべて IP 側から書かれるということにする。ADLY, WDLY, DDLY の 3 つは、基準時刻に対してアドレス、WE、データを遅延させるパイプライン段数を示す。なお、OE のネゲートが書き込みサイクルの始まりを定義する。これらを制御可能にすることで、例えばディレイドライトに対応したり、またメモリアドレスにレジスター・バッファ、あるいは変換テーブルをつけたりすることを可能にする。さらに、ODLY が、最後のデータに対して OE を下げる（アサートする）タイミングを定義する。

VCIS は物理／仮想チップ番号であり、MSB 10 ビットが物理、LSB 10 ビットが仮想とする。物理番号が外から供給されている物理 ID と等しい時にのみ、仮想番号を変更する。この仮想番号は上でのべたメモリ書き込み制御に使う。

ステータスレジスタは今のところ 4 ビットだけで、入力データにパリティエラーがあった時その場所を記憶する。これは SRST がハイでクリアされる。パリティは奇数パリティで、8 ビットごとに 1 の数を数えて奇数の時 1 である。なお、このすべての OR をとったものも出力しておく。ステータスレジスタは、現在のところこれしかないのでこれを DOUT から直接出力し、AOUT は無視する。

WE, OE, MEMA はメモリチップが 2 個あることに対応して 2 組持つ。

ENABLE 信号を見て、メモリに書くシーケンサが回る。これは、単なるカウンタであって、値が 0 でなければダウンカウントする。0 なら、来るはずの語数にセットする。

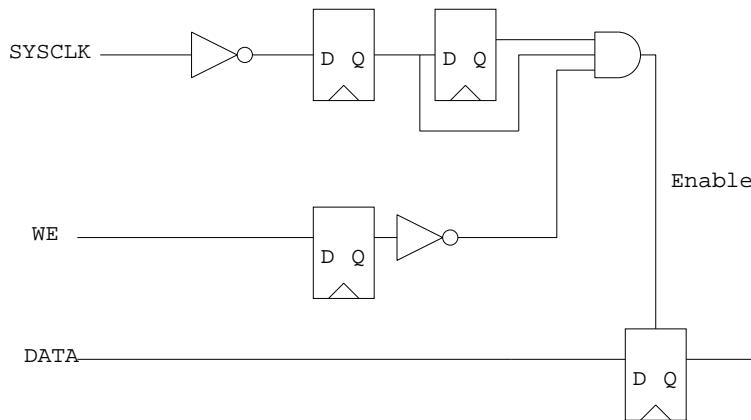


図 7.19: JP データ受けとり回路

表 7.16: JPW 制御テーブル入力アドレスマップ

データ	アドレス (2進)	ビット長	説明
MAP	00XXXX	4	アドレス変換テーブル
ND	010000	4	一粒子のデータ数 (32ビットワード)
VCID	010001	20	物理チップ番号／仮想番号
ADLY	010010	3	アドレスディレイ
WDLY	010011	3	WE ディレイ
ODLY	010100	4	OE ディレイ
DDLY	010101	3	データディレイ
予約	1XXXXX	20	拡張用予約

シーケンサの動作としては、制御カウンタが0の時、ENABLEが来たらデータを見て、最上位が00ならカウンタにセットする。同時に、来たデータの粒子番号と、チップ内にストアされた粒子番号レジスタを比べ、比べた結果をマスクして依然違っていれば、このチップが受けとるべきデータではないので、メモリへの write 信号を出なくする。

次のデータがきたら、メモリアドレスカウンタに値をセットする。

その次から、データが来たら、メモリアドレスカウンタの値でメモリに書く。

で、アドレスをインクリメントする。最下位のアドレスは2つのチップのどちらに書くかを変える。で、必要なデータを書き終えたら、制御カウンタが0になっておしまいである。

なお、SECDED (Single Error Correct Double Error Detect) のECCを使って書き込むことにするので、32ビット2語（バイトパリティ）を受けとってから、ECCを生成して上下両方のメモリに一度に書き込むことになる。ECC生成については別に述べる。

入力データにパリティエラーがあれば、あったことをステータスレジスタに記憶しておく。これはRSTでリセットされる。さらに、このRSTでシーケンサ自体もリセットされ、アイドル状態に戻るということにしておく。

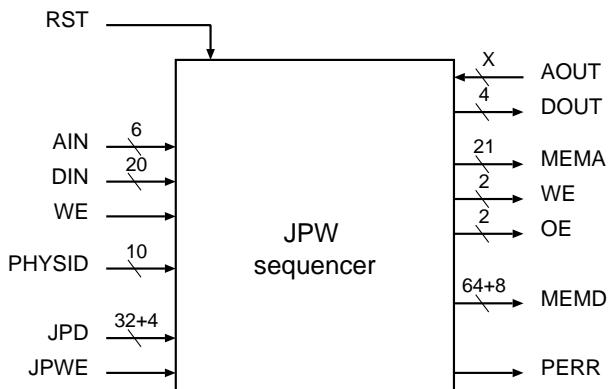


図 7.20: JPW 制御部 インターフェース

なお、すべての使い方で全データを送りたいとはかぎらないので、16 ワードのデータの一部だけを送ることもできるようにする。このために JPW にアドレス変換テーブルを持たせる。これにより、必要なところだけを送ることができるようになる。このために、ND と MAP レジスタを使う。ここでは実装の詳細は省略する。

JP データ取り込みのカウンタのカウント数上限と、カウント数からレジスタアドレスを生成するテーブルを設定することになる。カウント数は最大 16 なので、MAP は 4 ビット入力 4 ビット出力のテーブルを持たせることになる。

#### 7.4.4 IPW

ポートとの物理的なインターフェースは JP の場合と全く同じで、有効なデータが WRITE ENABLE 信号で示されることになる。

I 粒子データについては送るものは表 7.17 のようになる。これは全部で 12 語ということになる。

なお、速度、ソフトニング、ネイバー半径については 32 ビット IEEE754 形式から 36 ビットの GRAPE-6 内部形式に変換する必要があることに注意して欲しい。具体的には以下の手順で変換する。

1. ゼロかどうか判定する。これは、IEEE-754 なので本来全ビットが 0 なら 0 だが、デノーマルは 0 に落すことにして指数部が 0 なら 0 フラグを立てる。
2. 非数、無限大については、判定しない。これは、変換後のデータに対応する表現がないからである。したがって、これらも普通の数に変換されてしまう。質量が非数だったり無限大だったりすることは本来ないので、これは問題ではない。
3. それ以外の普通の数については、仮数は削っていた 1 を付け加え、指数はオフセットが違うのでその分を加算するだけである。オフセットは 385 (384 でないことに注意) である。これは設計ミスではあるが、、、

チップのモードなどを書くコマンドはすべてここから送ることにする。シーケンサ等の設計を簡略化するため、論理的にはすべてのレジスタがユニークなアドレスを持つものとし、書き込みは指定したアドレスから指定した語数のパケットという形で来ることにする。

表 7.17: I粒子データ

データ	ビット数	個数
位置	64	3
速度	36	3
ソフトニング	36	1
ネイバー半径	36	1
粒子番号	32	1

```

clk      ======-----=====-----=====-----=====
WE       ======\-----/=====-----=====
address -----<=address0>-<=address1>-----
data     -----<=data0==>-<=data1==>-----

```

図 7.21: IP ポート制御回路とパイプラインのタイミング

レジスタマップを表 7.18 に与えたようなものとする。ここで、AAA となっているのは物理パイプライン番号、BBB は仮想パイプライン番号である。

相互作用パイプラインのほうから見た信号定義とタイミングを表 7.19 と図 7.21 に与える。

図 7.21 に write タイミングを示す。単なる同期書き込みである。

図 7.22 に IPW ユニット全体の信号定義を与える。AFP は相互作用パイプライン用、A はそれ以外のすべて用のアドレスである。

これは、ある程度汎用的なシーケンサとして実現する。IP ポートからのパケットは 2 ワードヘッダとその後のデータであり、ヘッダは最初のワードがスタートアドレス、次が語数ということにする。*i* 粒子データを実際に書く時は間接バーストをするが、それ以外のデータについてはすべて連続バーストとする。

さて、実際に間接アクセスする必要があるのは、せいぜいアドレス下位 4 ビットなので、アドレス変換テーブルは 4 ビット入力 4 ビット出力である。実現の概要を

図 7.22 にしめす。WE はアドレスの最上位 4 ビットをフルデコードで出しておけばよい。

まずスタートアドレスと転送語数が来て、これで動作が始まる。後は、データが来るたびに語数カウンタを減らし、アドレスカウンタを必要に応じて増やし、書く必要があるときは書く。なお、実際に間接バーストに対応するのは *i* 粒子を書く時だけなので、それ以外の時の動作は確定している。つまり、データが来るたびに WE を出してアドレスカウンタを増やす。

*i* 粒子のために必要なアドレスは 10 ビットである。この上にさらに 4 ビット付け加えてこれを使ってブロックごとのデコードをすることにするので、全部で 14 ビットのアドレスとなる。*i* 粒子の物理パイプのためのデコードはさらに別ユニットで行なう。*i* 粒子一つを書くためのアドレスシーケンスを表 7.20 に示す。

具体的には、アドレス生成部は図 7.23 のようなものを作ればよい。

図 7.23 では細かい制御線は省いてあるが、DR はデータレジスタで IPD から有効なデータが来た時にロードされる。ワードカウンタは、ワード数が来たら憶える。で、あとは有効な

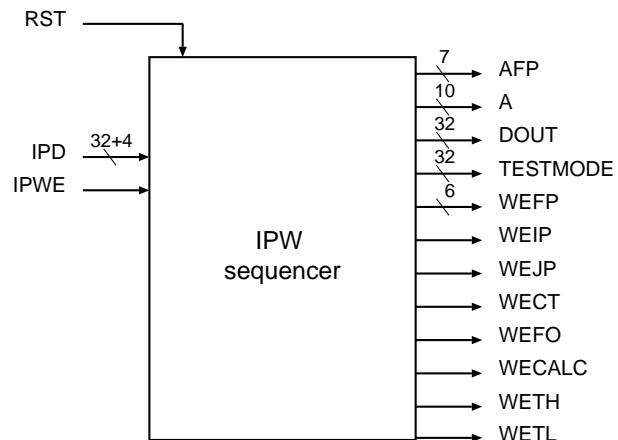


図 7.22: IPW 制御部 インターフェース

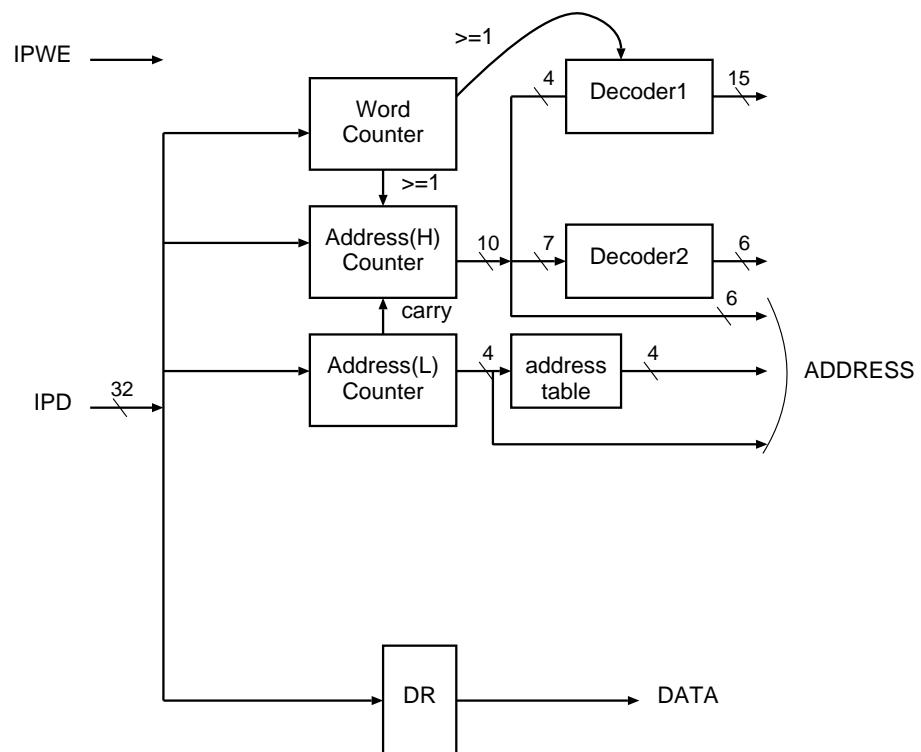


図 7.23: IPW 制御部概念図

表 7.18: IP ポートデータライトレジスタマップ

データ	アドレス (2進)	説明
XH	0000AAABBB0000	I 粒子 x 座標
XL	0000AAABBB0001	I 粒子 x 座標
YH	0000AAABBB0010	I 粒子 y 座標
YL	0000AAABBB0011	I 粒子 y 座標
ZH	0000AAABBB0100	I 粒子 z 座標
ZL	0000AAABBB0101	I 粒子 z 座標
VX	0000AAABBB0110	I 粒子速度 x 成分
VY	0000AAABBB0111	I 粒子速度 y 成分
VZ	0000AAABBB1000	I 粒子速度 z 成分
EPS2	0000AAABBB1001	ソフトニング
H2	0000AAABBB1010	ネイバー半径
I	0000AAABBB1011	粒子番号
SCALES	0000AAABBB1100	PHISCALE, FSCALE, JSCALE
IPRG	0001XXXXXXXXXX	IP 制御部内部レジスタ
JPRG	0010XXXXXXXXXX	JP 制御部内部レジスタ
CTBL	0011XXXXXXXXXX	カットオフテーブル
FORG	0100XXXXXXXXXX	FO 制御部内部レジスタ
CARG	0101XXXXXXXXXX	計算制御部内部レジスタ
PRTH	01100000000000	予測パイプ TI 上位ワード
PRTL	01100000000001	予測パイプ TI 下位ワード

データが来る度にカウントダウンし、0になったら止まる。アドレスカウンタは、ワードカウンタのデータが0でなければカウントアップするが、下位4ビットについては、*i*粒子書き込みの時には設定された値まで来たらキャリーを出して次で0に戻るようにしておく。

デコーダ1は単なるデマルチプレクサで、アドレスの上位4ビットを16ビットのなかの1つだけが0のものに変換する。オール0は相互作用パイプライン用ということにするので、それ以外の15本がされることになる。なお、これはワードカウンタとIPWEから制御線が入って、有効なデータが来た時にだけ1を出す。

デコーダ2はも基本的には単なるデマルチプレクサで、アドレスの上位4ビットの次の3ビットを8ビットのなかの1つだけが0のものに変換する。ただし、上位4ビットがすべて0の時だけ1をだす。

アドレス変換テーブルは、IPREGのアドレス空間の中に書き込むことによって書き込めるものとする。アドレス割つけは適当で構わない。例えば、アドレステーブルは全部で64ビット、WEテーブルは16ビットしかないので、それぞれまとめて1語として書くというのでもいいし、適当に分けてかくのでも構わない。

IPREGのマップは一応7.21のようとする。

表 7.19: IP ポート制御回路とパイプラインのインターフェース

信号	説明
A0-AX	アドレス線
D0-D31	データ線
WE	ライトイネーブル

表 7.20: I 粒子書き込みサイクル

サイクル	アドレス (2進)	説明
0	X0000	X 第 1 ワード
1	X0001	X 第 2 ワード
2	X0010	Y 第 1 ワード
3	X0011	Y 第 2 ワード
4	X0100	Z 第 1 ワード
5	X0101	Z 第 2 ワード
6	X0110	VX
7	X0111	VY
8	X1000	VZ
9	X1001	EPS2
10	X1010	H2
11	X1011	INDEX
12	X1100	SCALES
13	Y0000	X 第 1 ワード
14	Y0001	X 第 2 ワード

表 7.21: IPREG 制御レジスタマップ

データ	アドレス (2進)	説明
MAP	00XXXX	アドレス変換テーブル
ND	010000	一粒子のデータ数
TESTMODE	010001	予測子パイプおよび相互作用 パイプのテストモード
予約	1XXXXX	未定義

表 7.22: FO ポートデータリードレジスタマップ

データ	アドレス (2進)	説明
FPREG	0000AAABBBXXXX	相互作用パイプライン結果レジスタ
NNBREG	01AA1000000000	ネイバーリスト語数レジスタ
NBLMEM	01AA0XXXXXXXXXX	ネイバーリストメモリ
チップ予約	1XXXXXXXXXXXXXXX	未定義

表 7.23: FO ポート制御回路とパイプラインのインターフェース

信号	説明
A0-AX	アドレス線
D0-D31	データ線

#### 7.4.5 FO

結果読みだしについても、相互作用パイプラインの側はランダムアクセスに対応できるものとする。制御回路とパイプラインチップのインターフェース信号は表 7.23 の通りであり、タイミングは図 7.24 に示す。単なる同期読みだしである。リードレイテンシは別途指定する。ここは、リードイネーブルとかはなくて、アドレスに対応したデータが常時出てくる。FO 部はパイプラインの本数だけの入力ポートを持ち、中のマルチプレクサでデータを選ぶ。

相互作用パイプラインの他に返すべきデータは以下の 2 つである。

1. ネイバーリストユニット
2. 諸々のステータス

ステータスは、主に転送エラー状態を記憶するものである。送り返すデータが正しいかどうかは受けとてみないとわからないので、これはインターフェースボードにステータスがあることになる。従って、ステータスがあるのはもうう側、すなわち JPW, IP, JPMEM のインターフェースということになる。FO ユニットからみたチップ内アドレスマップは表 7.22 のようになる。

FO 部用のアドレス生成回路は IP 部のものと基本的には同じである。つまり、相互作用パイプラインから読み出す時にはアドレス変換テーブルを使う必要がある。

ACTIVE という信号線に注意してほしい。これがハイならホストはこのチップの存在を無視することにする。これは、故障したチップをスキップするのに使う。物理的にこの線をカットできるように基板を作ておく必要がある。レジスタ書き込みによってソフトウェアで操作し、論理的に無視することもできるようにしておく。

```
clk      -----=====-----=====-----=====-----=====-----=====
address -----<=address0>-<=address1>-----
data    -----<=data0==>-<=data1==>-----
```

図 7.24: FO ポート制御回路とパイプラインのタイミング

表 7.24: FO ポート相互作用データリードレジスタマップ

データ	アドレス (2進)	ワード数	説明
FX	000AAABBB0000	2	I 粒子加速度 x 座標
FY	000AAABBB0010	2	I 粒子加速度 y 座標
FZ	000AAABBB0100	2	I 粒子加速度 z 座標
POT	000AAABBB0110	2	ポテンシャル
JX	000AAABBB1000	1	I 粒子 jerk x 成分
JY	000AAABBB1001	1	I 粒子 jerk y 成分
JZ	000AAABBB1010	1	I 粒子 jerk z 成分
RNB	000AAABBB1011	1	最近接粒子距離
INB	000AAABBB1100	1	最近接粒子番号
STS	000AAABBB1101	1	パイプライン演算ステータス

表 7.25: FO ポート制御回路とパイプラインのインターフェース

信号	入出力	説明
D0-D35	O	データ線 (バイトパリティ含む)
VD	O	Valid data 有意味なデータが出ていることを示す
ND	O	New data データが前のサイクルと変わっていることを示す
STS	O	status アイドルではなくにかしていることを示す
ACTIVE	O	0 の時使われていることを示す。
WD	I	Wait data データを止める

外部インターフェースの仕様を以下に述べる。ネイバーリストのところで述べた様に、フロー制御が必要であるので、信号定義は表 7.25 以下になる。

図 7.25 から 7.27 に 出力ポートのタイミングを示す。WD はサンプルされたクロックの次のクロックで出るデータに反映されるものとする。つまり、タイムチャートとしては WD がサイクル  $i$  で下がっていたとすると、ND がサイクル  $i+2$  で上がっているということになる。ND が上がっているサイクルではデータは更新されてはいけないことに注意してほしい。さらに、最初に WD がアサートされている時の動作は特殊になる。というのは、この場合は WD が出ていても VD, ND をアサートするからである。

FO 部からチップ内部へのアクセスはもちろんレイテンシがあるので、これらの図に示したように、WD に対してディレイなしで応答するのは一見困難にみえるかもしれない。しかし、ここで示しているのは外部クロックであるということに注意して欲しい。このため、FO のアドレス生成は 4-6 サイクルに一度しかインクリメントされる必要はない。チップ内の他ユニットのレイテンシが異常に大きくない限り、FO 部は WD を見てから動作を変えればいいことになる。

FO 部のインターフェースを図 7.28 に示す。左上は IP から来るコマンド用の入力インターフェイスである。レジスタはたくさんあるので、適当にコマンドを作れることになる。表 7.26

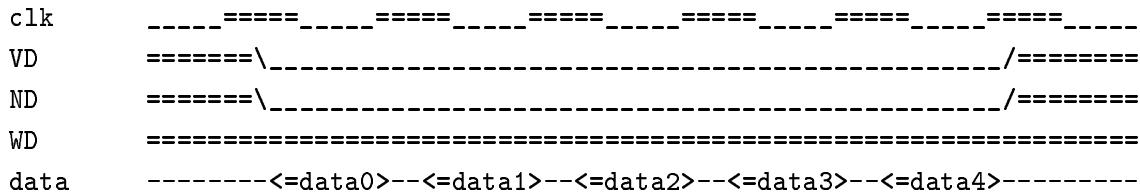


図 7.25: 出力ポートタイムチャート（ウェイトなし）

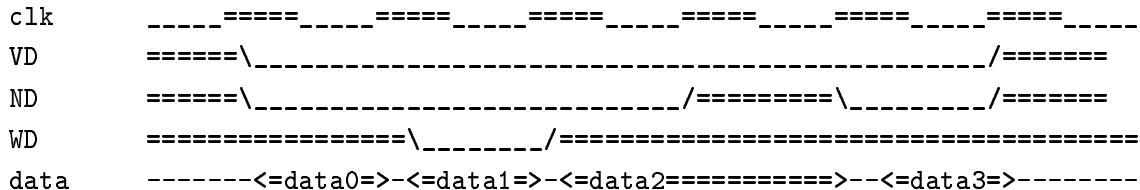


図 7.26: 出力ポートタイムチャート（ウェイトあり）

は信号の簡単な説明である。

FO部の動作は今のところ以下の3種類ということになろう:

1. 相互作用パイプライン読みだし
  2. ネイバーリスト読みだし
  3. その他ランダム読みだし

相互作用パイプライン読みだしは、それ以外の2つとはちょっと違うことを注意しておく。これはWDを見る必要がない。また、計算が終ったら勝手に動作を始める。送り返すべき粒子数については、IP側からコマンドで書く。

ネイバーリスト読みだしでは、WDを見る必要がある。さらに、コマンドが来たら、3つのユニットを順に見ていく。また、それについて、まず語数を読みだして、その語数だけ送るようにする。

ランダム読みだしでは、指定されたアドレスと次の2語だけを返す。これはWDを見ると  
いうことにしておく。

ネイバーとランダム読みだしについては、IP 側の仕様から、2 ワードのコマンドが来るところになる。そのフォーマットを表 7.27 のようにしておく。なお、アドレスマップは表 7.29 で

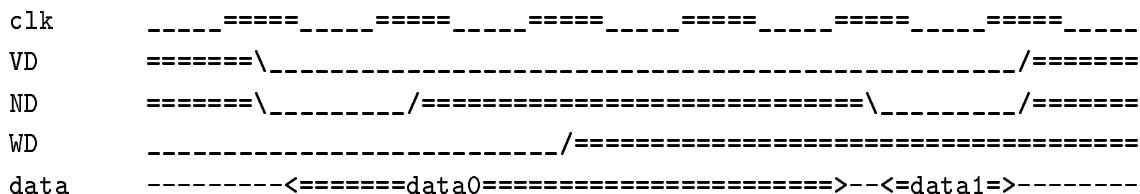


図 7.27: 出力ポートタイムチャート（最初に WD アサート）

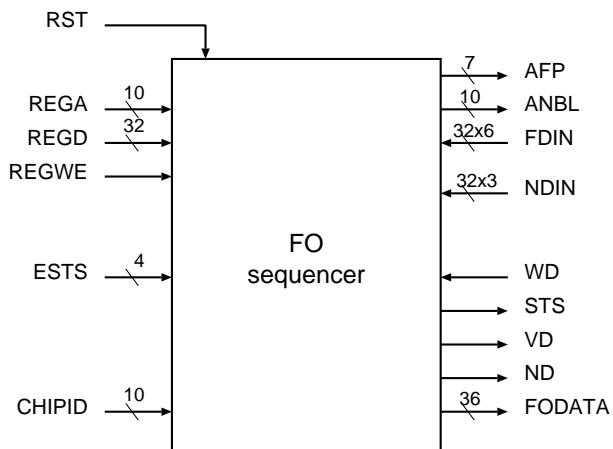


図 7.28: FO 制御部インターフェース

表 7.26: FO ポート制御回路の全体インターフェース

信号	入出力	説明
RST	I	状態リセットしアイドルに戻す
REGA0-9	I	コマンドレジスタアドレス
REGD0-31	I	コマンドレジスタデータ
REGWE	I	コマンドレジスタライトイネーブル
ESTS0-4	I	データ転送エラーステータス
CHIPID0-9	I	仮想チップ番号
AFP0-6	O	相互作用パイプラジスタアドレス
ANBL0-9	O	ネイバーリストユニット内部アドレス
FDINO-31	I	相互作用パイプデータ入力(6本)
NDINO-31	I	相互作用パイプデータ入力(3本)
FODATA0-35	O	データ線(バイトパリティ含む)
VD	O	Valid data 有意味なデータが出ていることを示す
ND	O	New data データが前のサイクルと変わっていることを示す
STS	O	status アイドルではなくなにかしていることを示す
ACTIVE	O	0 の時使われていることを示す。
WD	I	Wait data データを止める

表 7.27: FO部コマンドフォーマット

フィールド名	先頭／第二	ワード内位置	説明
コマンドコード	先頭	[31:30]	コード
チップ番号マスク	先頭	[19:10]	
チップ番号	先頭	[9:0]	
メモリアドレス	第二	[20:0]	

表 7.28: FO部コマンドコード

コード	名称	説明
00	NBREAD	ネイバーリスト読みだし開始
01	RREAD	2ワードランダム読みだし

ある。FOREGは、IPと同様の間接バーストを実装するためのテーブル等に使う。また、チップ番号とチップ番号マスクは、JPWと同様にアドレスとして使う。

コマンドはIPW側から2語、CMD0とCMD1が順に来るので、CMD1が来たら動作を始める。なお、仮に計算結果を返している最中であれば、コマンドを無視してよい。

ネイバーリストについては、まず語数を返し、それからメモリの中身をその語数（の2倍）だけ続けて送るということになる。最初の語数のところに、チップ番号、ユニット番号等をパックしておいて、どのユニットから読んだデータかわかるようにしておく。具体的には、表7.30の形式で最初の語を構成する。あとはアドレス順にそのまま返せば良い。

#### 7.4.6 計算開始／終了の制御とメモリデータのパイプラインへの供給

計算開始／終了の制御とメモリデータの読みだし／供給は同一のシーケンサで行なうものとする。これをCALC部と呼ぶ。インターフェースを図7.29および表7.31に示す。予測子パイプ向けのデータ出力のうち、はじめの144ビットは座標成分であり、 $x, y, z$ の順に出る。

表 7.29: FO部レジスタマップ

データ	アドレス(2進)	説明
FOREG	0XXXX	FO部内部レジスタ
CMD0	10000	コマンドレジスタ下位ワード
CMD1	10001	コマンドレジスタ上位ワード
NI	10010	$i$ 粒子数レジスタ
NW	10011	語数レジスタ
INACTIVE	10100	「非使用」レジスタ

表 7.30: ネイバーリスト第一ワード

フィールド名	ワード内位置	説明
仮想チップ番号	[21:12]	
ユニット番号	[11:10]	ネイバーユニットの番号
語数	[9:0]	リストに入っている粒子数

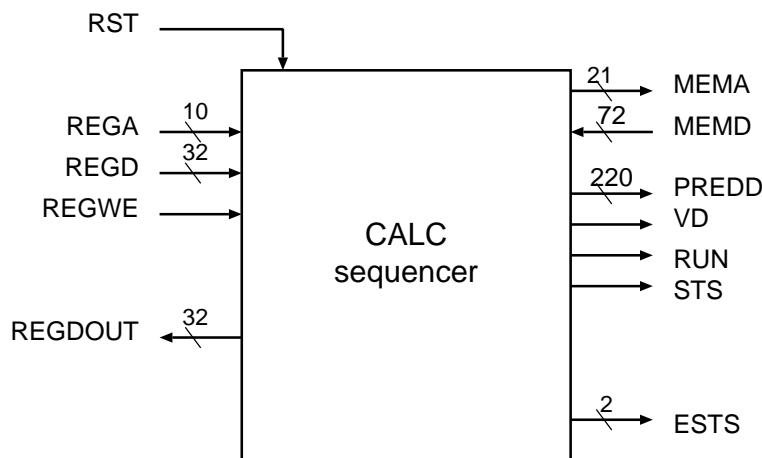


図 7.29: CALC 制御部インターフェース

表 7.31: CALC 制御回路のインターフェース

信号	入出力	説明
RST	I	状態リセットしアイドルに戻す
REGA0-9	I	コマンドレジスタアドレス
REGD0-31	I	コマンドレジスタデータ
REGDOUT0-31	O	レジスタデータ出力
REGWE	I	コマンドレジスタライトインペブル
MEMA0-20	O	メモリアドレス
MEMD0-71	I	メモリデータ
PREDD0-219	O	予測子パイプ向けデータ出力
VD	O	予測子パイプ同期信号
RUN	O	相互作用パイプ演算信号
STS	O	計算ステータス信号

表 7.32: メモリデータのワード割り付け

ワード番号	データ	説明
0	X	
1	Y	
2	Z	
3	VX VY	(0:31) (31:63)
4	VZ AX AY	(0:31) (32:52) (53:63) 下位 11 ビット
5	AY AZ JX JY	(0:9) 上位 10 ビット (10:30) (31:47) (48:63) 下位 16 ビット
6	JY JZ JJX JJY JJZ T	(0:0) 上位 1 ビット (1:17) (18:28) (29:39) (40:50) (51:59)
7	MASS INDEX	(0:31) (31:63)

なお、質量については、ホストは IEE-754 単精度で書き込む。メモリから読み出してから GRAPE-6 相互作用パイプラインの内部形式に変換して予測子パイプラインに送るということにする。これは IP ポートからの相互作用パイプラインの内部レジスタへの書き込みと同様である。

表 7.32 に、メモリの 512 ビットのどこに何を割り当てるかを示す。メモリは 2 つあるので、偶数アドレスに対応するほうを下位ワードと解釈する。また、JJX 等は 2 階導関数である。

メモリへの書き込みについては JPW が面倒をみるのでここではふれない。バスラインが浮かないようにするために、JPW でメモリの出力制御を行なう。

SSRAM のリードレイテンシについても CALC 内でプログラマブルとして、後で付加回路とかも付けられるようになる。表 7.33 にそれらを含めたレジスタマップをしめす。

基本的な動作としては、アドレスカウンタに初期値 N がロードされると、STS がアサートされる（ロウになる）と同時にアドレスが出始める。アドレスの数える数は常に粒子数の 8 倍なので、送られるのは粒子数でありカウンタの設定値は CALC が判断する。で、メモリからデータが出てくるようになれば、そのあと 8 サイクル毎にデータをまとめて予測子パイプに渡す。この時 VD をアサートする。また、RUN 信号を制御する。これは、最初の VD が上がるのと同期してあげ、最後の VD と同期して下げる。また、STS は相互作用パイプで計算が終るまで下げておく。

表 7.33: CALC 制御部のレジスタマップ

名称	アドレス	説明
LRAM	00000	SSRAM リードアクセスレイテンシ
LFORCE	00001	相互作用パイプラインレイテンシ
N	00010	アドレスカウンタ初期値

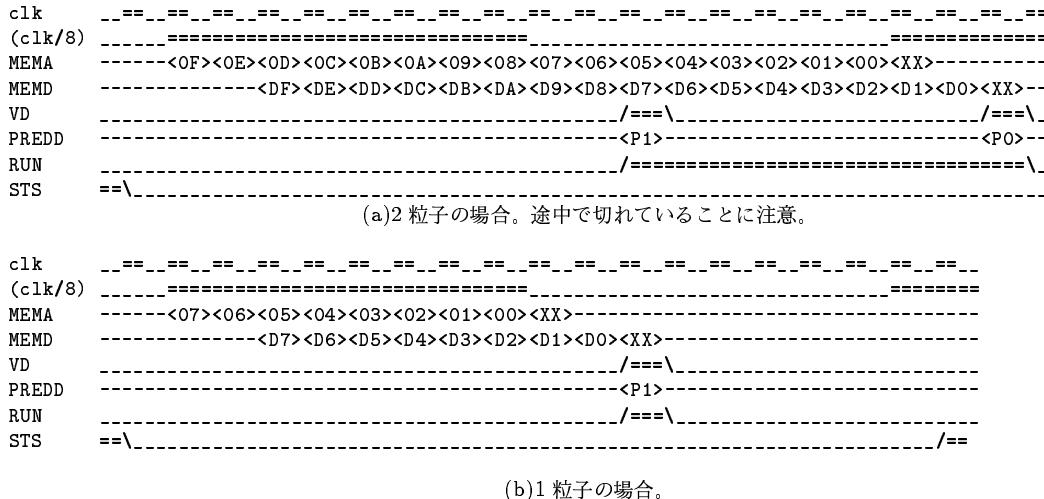


図 7.30: CALC 部動作

全体のタイミングチャートは例えば図 7.30 のようになる。

なお、MEMA が出てから MEMD が来るまでの遅延と、計算が終了してから STS が出るまでの遅延はプログラマブルである必要がある。これらはすべてシフトレジスタで実現する。具体的には、LRAM の値を見て MEMA から MEMD までの遅延を決定する。また、LFORCE で計算が終ってからいつまで STS を下げているかを決める。



## 第8章 チップ入出力詳細

本章では、入出力の仕様を述べる。なお、ここで述べるのは論理的な記述（ほぼビヘイビアレベルに対応）であり、タイミング、あるいは電気的仕様は別章になる。インターフェースは、入力2つ、出力とメモリの4個からなる。GRAPE-4とは異なり、入出力は別にする。これは、分離して並行動作を可能にすることで、実効的な転送速度を増加させるためである。

なお、この内容はまだファイナルなものではないことに注意してほしい。

### 8.1 入出力プロトコル

入出力とともに、アドレスなしの同期転送とする。基本的には可変長パケット転送とするが、詳細はポート毎に異なる。

### 8.2 入力ポート一般

書き込みは、基本的にハンドシェイクしない「書きっぱなし」とする。つまり、有効データと同時に WE (write enable) 線を下げるだけである。

図8.1にwriteアクセスのタイミングを示す。

### 8.3 JPポート

ここでは、送ったデータがメモリ上の連続アドレスに格納される。最初の2ワードに先頭アドレスと転送ワード数を格納し、その後にデータを入れる。アドレスと転送ワード数をパックすることもできるが、どうせメモリ側ポートが64bitワードアクセスであるので、ホスト

表 8.1: チップ入出力

ポート	I/O	幅	速度	説明
JP	I	36	25MHz	$j$ 粒子入力
IP	I	36	25MHz	$i$ 粒子入力
FO	O	36	25MHz	計算結果出力
FSTS	O	1	25MHz	ステータス出力
MEMD	I/O	$36 \times 2$	125MHz(?)	メモリデータ
MEMA	O	$20(?) \times 2$	125MHz(?)	メモリアドレス

```

clk      -----
WE       =====\-----/=====
data    -----<=data0==>-<=data1==>-----

```

図 8.1: 入力ポートタイムチャート

表 8.2: JP データ

データ	ビット数	個数
位置	64	3
速度	32	3
加速度	20	3
第一導関数	16	3
第二導関数	12	3
時刻	8	1
質量	32	1
番号	32	1
合計	—	504

で 8 バイトバウンダリーとぶつからないように、制御データも基本的に 64 ビットで送るものとする。メモリ 2 つには連続して送ったデータが交互に書かれるということにする。

なお、ある種のコマンドは JP ポートから送られることがある。

JP ポートからの入力を表 8.2 に示す。これは、パックした形で入ってくることを前提に、メモリイメージのまま書くことにする。なお、まだ 1 バイト余りがあるので、ここに CRC か パリティを入れる。パリティについては、転送時、メモリからのデータ入力時の両方でパリティチェックを行ない、どこでエラーが発生したかわかるようにする。

ホストの処理がちょっと増えるが、ビット操作にたいした時間は掛からないであろう。1 次 キャッシュ、あるいはレジスタ内の操作で済み、I/O ポートを通したデータ転送に比べれば 無視できるものと思って良い。

## 8.4 IP ポート

ここでは、 $i$  粒子の他、コマンド等も送る必要があるであろう。一般形式とて、最初の 2 ワードにコマンドコードと（必要であれば）その後のデータ長を書き、その後に（あれば）データを入れる。I 粒子データについては送るものは表 8.3 のようになる。

パリティエラーを検出する必要があることに注意。

表 8.4 は大嘘であるので、信じないように。

表 8.3: I 粒子データ

データ	ビット数	個数
位置	64	3
速度	32	3
ソフトニング	32	1
ネイバー半径	32	1
粒子番号	32	1

表 8.4: IP ポートコマンドフォーマット

コマンド	OP コード	パラメータ	データの有無
I 粒子書き込み	0x00XXXXXX	粒子数	無
計算開始	0x01XXXXXX	粒子数	無
チップ論理 ID 書き込み	0x02XXXYYY	物理 ID (XXX), 論理 ID (YYY)	無
現在時刻 書き込み	0x03XXXXXX	なし	有 (8 バイト)
ネイバー転送開始	0x04XXXXXX	なし	なし

## 8.5 出力インターフェース

出力も、入力同様「勝手に行なう」ものとする。すなわち、IP ポートからのコマンドに反応して、チップが勝手に出力を行なう。従って、データが出たということを伝える線が必要である。これは valid data (VD) とする。但し、NREAD の場合、ノード FPGA 側からウェイト要求がかかることがあるので、これに対応する必要がある。ウェイトが掛かっていない時には、VD をアサートするのと同時にデータを出す。ウェイト (wait data, WD) がアサートされたらその次のクロックからデータを止める。データを出すタイミングであらかじめ WD が下がっていたら、データを出すがそのまま保持する。WD がネゲートされたらその次から新しいデータを出す。

図 8.2 から 8.4 に出力ポートのタイミングを示す。

実際に送り返されるデータは、チップのステータス、加速度等の計算結果、ネイバーリストの 3 種類である。それぞれについて簡単に述べる。

まず、ステータスは、表 8.5 にあるようなものとなろう。

```
clk      =====---=====---=====---=====---=====---=====---=====
VD       ======\-----/=====
WD       ==========
data    -----<=data0>--<=data1>--<=data2>--<=data3>--<=data4>-----
```

図 8.2: 出力ポートタイムチャート（ウェイトなし）

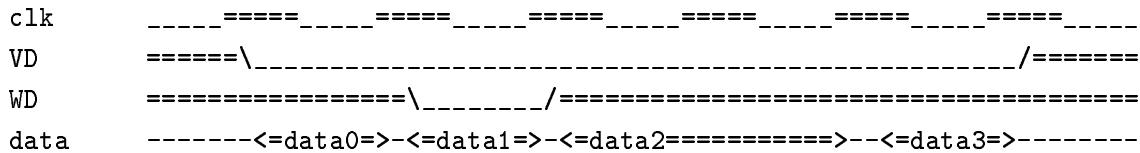


図 8.3: 出力ポートタイムチャート（ウェイトあり）

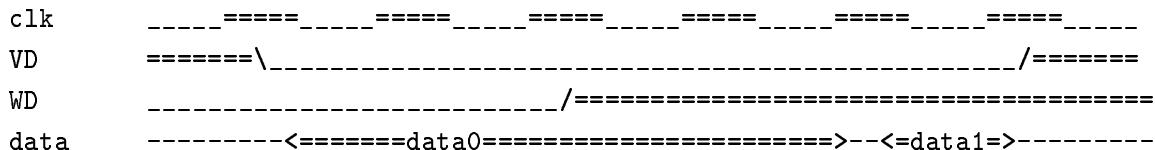


図 8.4: 出力ポートタイムチャート（最初に WD アサート）

一粒子のデータは、表 8.6 にあるようなものである。一応、これらをパケットにまとめて送り出す。データを受ける FPGA の方は、何をするべきか知っていると想定する。8 バイトバウンダリーになるようにチェックサムを最後に入れて送り出すことにする。何がオーバーフロー／アンダーフローしたかを示すために、1 バイトのステータス情報を付ける。語数を増やすために、これは RNB とまとめて書いてますことにする。

さて、NREAD の時はどのような形でデータを返すべきであろうか？チップ内の VMP の本数は 48 ないし 64 程度であるので、一つのリストをシェアする粒子の数は 16 位が適当であろうと思われる。もちろん、使い勝手からするとシェアしないもののほうがわかりやすいし、安直な使い方の場合にはメモリ効率もよい。しかし、まあ、どうせ通常の使い方では大した語数はいらないので、必要な人は頑張れば増やせるというほうがいいのではないかと思われる。従って、GRAPE-3 や 4 と同様に、リストをシェアすることにする。シェアするのは 16 粒子とし、ネイバーであったもの毎に 16 ビットのどれのネイバーであったかという情報と、パイプラインセットの番号を付ける。これでもまだフィールドが 14 ビットが余るので、パリティビット、リストが溢れたかどうかのフラグとネイバーリストメモリのなかのインデックスもつけておく。これらから、再計算するべきかどうか判定できる。なお、GRAPE-3 と同様、J 粒子のインデックスカウンタはロードした数からダウンカウントするものとし、これによりネイバーが多い時には溢れたところから再計算できるようにする。

表 8.5: チップステータスフラグ

名前	説明
JPERR	J 粒子書き込みエラー
IPERR	I 粒子書き込みエラー
MEMERR	メモリーのデータ化け
CHIPID	チップ番号

表 8.6: 出力ポートから出るデータ

	幅 (バイト)	語数	型	説明
acc	8	3	signed int	加速度
pot	8	1	signed int	ポテンシャル
jerk	4	3	signed int	加速度の導関数
sts	1	1	unsigned int	計算結果ステータス
rnb	3	1	unsigned int	最近接粒子の距離
inb	4	1	unsigned float	最近接粒子の番号
chk	4	1	unsigned int	チェックサム

## 8.6 メモリインターフェース

入力データはアドレス範囲によっては直接外づけメモリに書き込まれる。外づけメモリはSSRAMというすることにする。SDRAMまたはRAMBUS DRAMの方が安価で高い bandwidth が得られるが、いずれにしてもメモリチップの価格はプロセッサチップに比べて知れているので、あまり深く考えないでSSRAMを使うことにする。

SSRAMの動作は、同期であるということを除いて従来のSRAMと大差ないので、設計が容易であるということも重要である。また、速度も、十分に速いものが2次キャッシュ用として出荷されるようになってきていて、入手も比較的容易であることが期待できる。