VPM アセンブラ仕様書

Ver 0.0 — 2009/3/2

牧野淳一郎

August 13, 2009

Contents

1 知られている未修正バグ

1.1 2006/4/29

長語定数の il 形式は上位 36 ビットを設定しない。
回避方法: hl 形式を用いる。

2 履歴

2.1 2009/3/2

内部資料「VPM アセンブラ定義」からユーザー向け文書への変更開始。

3 TODO

- ハードウェアで実際に実行できるライブラリを作る方法を書く。

4 アセンブラ定義

4.1 基本構造

ホストプログラムの基本的な構造は、(粒子間相互作用の場合) 以下のようになる。

```
色々ホスト側処理
初期化関数を呼ぶ
色々ホスト側処理
タイムステップ毎に以下を繰り返す
    相互作用を及ぼす側の粒子データ（j 粒子データ）を送る
    以下全部の i 粒子の処理が終わるまで繰り返す
        ハードウェアが処理できる数の i 粒子を送る
        計算をスタートさせる
        計算が終わるのを待つ（この間他の仕事も可能）
        結果回収が終わり、まだ処理するべき i 粒子があればループ先頭に戻る
    相互作用計算が終わったので、時間積分とか他の処理をする
```

これらの処理のためにホストプログラムが呼ぶ関数は、アセンブラ記述の変数宣言から自動生成される。

アセンブラソースファイルは以下の部分からなる

変数宣言部
実行部

実行部は

全体初期化部
ループ初期化部
ループ本体部
ループ終了処理部

の 4 つの実行部分をもつことができる。全体初期化部は、アプリケーションプログラムがハードウェアを獲得する (foo_grape_init() 関数の呼び出しによる。foo はアセンブラ内で指定) 時に一度だけ実行される。ループ部は実行関数 (foo_grape_run ないし foo_type2_run) の呼び出しの時に順番に実行され、本体は指定した回数、初期化、終了処理はそれぞれ 1 回だけ実行される。上に述べたように、foo_grape_init 等の関数はアセンブラ記述から生成される。その詳細は 4.4 節で述べる。

この形をとることで、例えば粒子間重力計算や行列乗算、その他一般に繰り返しのある処理を表現できる。但し、ネストしたループをハードウェア側だけで実行させる機能は今のところもっていない。内部ループの回数が少なく、固定ならばアンロールすることで対応できる。アセンブラ自体はこのようなアンロールされるための繰り返しを表現する構造をもたないが、現在のアセンブラは erb によるテキスト処理を実行できるので、それによって展開されるループ構造を表現することは可能である。これについては 4.6 節で述べる。

4.2 変数 (アドレス) 記述

変数等の記述には以下のものがある

- ローカルメモリ変数
- 汎用レジスタ変数
- T レジスタ
- 乗算器フィードバック
- 放送メモリ変数
- PB, BB 番号

ローカルメモリと放送メモリには名前がついた変数をおくことができる。また、汎用レジスタには別名をつけることができる。それ以外のものが予約されたキーワードでアクセスする。

以下、順に説明する。

4.2.1 ローカルメモリ変数

変数宣言部で以下の形成で宣言した変数 name を、実行部で指定可能である。

```
var [vector] [{long,short}] name [address value]  [transfer type] [conversion type]
```

変数宣言は var で始まる。 vector があると 4 要素ベクタ、ないとスカラ変数になる。 1, 4 以外のベクトル長はサポートしない。 long だと長語 (72 ビット)、short だと単語であり、省略できたかもしれないがこれは指定して欲しい。その次のフィールドが名前である。これは英字で始まり、その後に英数字が続くものである。変数名の長さには制限はない。次の address value は省略可能で、これを書くことで絶対アドレス、また上で既に宣言された変数名を書くことでその変数と同じアドレスに宣言した変数がおかれる。address value が指定されない変数は宣言した順番におかれる。スカラー単語のあとのような、奇数アドレスが空き領域の先頭になっている状態で長語変数が宣言されると 1 短語のメモリホールができる。4 語のベクトルは、任意の語境界に置くことができる。

その次の transfer type と conversion type はホストとの通信を記述するためのものであり、後述する。なお、変数宣言は 1 行目に 1 変数とする。

なお、変数とは別に直接アドレスを

```
$[l]mn[v][n2]
```

という形で指定できることにする。ここで n は 10 進の短語アドレスである。m はローカルメモリを表す。例えば

```
$lm0v     アドレス 0 から始める長語ベクトル
$lm10      アドレス 0xa の長語
$m12v      アドレス 0xc からの短語ベクトル
```

となる。最後の n2 はローカルメモリの場合にだけ有効で、ベクトルストライドを表す。省略された場合には 1 である。なお、このことからわかるように、ストライドアクセスをするには現在のところ直接アドレス指定をする必要がある。

また、 T レジスタ間接アクセスは

```
$[sl]t
```

の形で表記する。T レジスタ間接アクセスの時、アドレスはアクセスが短語か長語かに関わらず常に短語アドレスとして解釈される。(これはハードウェアの仕様である)。

4.2.1.1 例

```
var vector long pos
```

で、posi は 4 要素の 1 次元配列ということになる。

アクセスがベクトルかスカラーかは、上の例では pos[i] ならスカラーとして (ベクトル命令の中でインデックスが固定) アクセス、 pos ならベクトルとしてアクセスする。

```
var vector long r2
```

ならベクトルで、

```
var long eps2
```

ならスカラーである。スカラーの場合には eps2[i] とかの記法はエラーになる。インデックスは、整数定数だけが書けることにする。

4.2.2 汎用レジスタファイルアクセス

これは

```
$[l]rn[v]
```

の形式とする。意味はローカルメモリの場合と同じ。ストライドアクセスや間接アクセスはないので、そのような表現はサポートしない。

4.2.2.1 レジスタ別名

レジスタ番号だけによるアクセスでは可読性が低いコードになるので、名前をつけることができるようにする。

```
alias name register_number
```

の形式で、 name はアルファベットで始まるスペースを含まない任意の文字列、register_number は $[l]rn[v] の形式の番号指定である。

4.2.3 T レジスタ

```
$t
```

とする。これは長語のみ、必ずベクトルデータである。直前の命令の結果が必要な時には $ti と書く。

4.2.4 乗算器フィードバック

```
$fb
```

とする。これは長語のみ。これは浮動小数点加算器 (または整数 ALU) の A ポート入力 (第一オペランド) でのみ使用でき、直前の乗算命令の結果を与える。

4.2.5 BM (放送メモリ)

var の代わりに

```
bvar [vector] [{long,short}] name  [transfer type] [conversion type]
```

として放送メモリ上に変数を定義する。アドレス表記は

```
$[l]bn[v]
```

とする。

4.2.6 即値

SING 命令セットではあまり役に立つ即値は指定できないが、アセンブラを書く時に即値が使えないのは不便である。従って、即値を書くとそれがローカルメモリに格納され、即値へのアクセスと書いたものががそのアドレスへのアクセスになるようになっている。

具体的には、オペランドに

```
[ifh][l]"sssss"
```

というものがあると、これらは

- ローカルメモリ上に領域が確保され
- IDP パケットと、BM からコピーする命令列が自動的に生成される。
- この命令列は全体初期化時に実行される。

上で ifh の一つは必須、 l はなければ短語になる。i は 10 進整数、f は浮動小数点数、h は 16 進数の形式で書くことを示す。

ローカルメモリ上の変数名は const0 から順番に付ける。同じ値なら再利用する。引用符の中は Ruby の表現が書ける。定数用の IDP パケットとコピー命令は、実際のコードの前に実行されることを前提にしたコード生成をする。具体的には、BM についてはコード内で使っているアドレスでもかまわずに上書きする。

4.2.7 PB,BB 番号

FADDSUB/IALU には PE, BB 番号を供給できる。これは特殊レジスタということで、 $pe, $bb と書く。

4.3 命令形式

基本的な形式は

[fmul op1 op2 result] [; {ialu_op,f{add,sub}} op1 op2 result] [; bm op1 op2 [peid]]

というものである。 fmul は乗算命令であり、その後の op1, op2 は入力オペランド (前節での変数やレジスタ指定)、 result は出力オペランド ((前節での変数やレジスタ指定) である。整数 ALU 命令と浮動小数点加減算はそのうち 1 つだけを指定可能である。 bm は放送メモリと PE の間のデータ転送命令である。これらは、1 行に記述したものは同時実行される。 bm 命令は peid を指定することができ、その場合には bm と指定した pe の間だけでデータ転送が行われ、他の pe は影響をうけない。

例えば浮動小数点乗算と加算を同時に行うには

```
fmul a b c ; fadd c d e
```

というようなコードを書く。1 つの命令に対して result は複数指定できる。つまり、例えば乗算器の結果をローカルメモリと T レジスタの両方に格納することが可能である。

BM の入出力は直接メモリユニットに行き、入力は fadd/ialu と重なっている。出力は間違って同じもの (ユニット) が指定されたら警告がでる。入力は faddsub/ialu と bm で矛盾している (違うものが指定されている) とエラーになる。

4.3.1 疑似命令

命令は 1 行目が SING 1 命令に対応するが、ベクトル長、マスクレジスタ指定等の命令毎に設定する必要性が薄いものは「疑似命令」とし、1 行使って書くがその効果は次の行以降に適用される。疑似命令はアセンブラに対する指示であるので、実行に時間はかからない。

4.3.2 NOP

なにもしない命令。形式は

```
nop
```

である。これは、これだけで 1 行でなければならない。ベクトル長以外の全ての命令フィールドを 0 にする。空白行は無視されるので、実際に 4 サイクルアイドルにするためにのは nop を書く必要がある。

4.3.3 FMUL

単純に fmul と書くだけでは多様な演算モードを指定できないので、これらを指定する方法を与える。

fmul のフラグは以下のように非常に沢山あるが、

SHIFT50A	ポート A の仮数を 50 ビット上にシフト
ROUND50A	ポート A の仮数を (シフト後)50 ビットに丸める
ROUNDA	ポート A の仮数を (シフト後)25 ビットに丸める
NORMALA	ポート A の入力を正規化数とみなす
SHIFT25B	ポート B の仮数を 25 ビット上にシフト
SHIFT50B	ポート B の仮数を 50 ビット上にシフト
ROUNDB	ポート B の仮数を (シフト後)25 ビットに丸める
NORMALB	ポート B の入力を正規化数とみなす
ROUND	出力を丸める
NORMALO	出力を正規化する

通常使う可能性があるのは以下の組合せである。

```
      s r r n s s r n r n
      h o o o h h o o o o
      i u u r i i u r u r
      f n n m f f n m n m
      t d d a t t d a d a
      5 5 a l 2 5 b l : l
      0 0 : a 5 0 : b : o
      a a : : b b : : : :
      0 0 1 1 0 0 1 1 1 1          単精度乗算、結果を丸める
      0 0 1 1 0 0 1 1 0 1          単精度乗算、結果を丸めない
(倍精度で格納)
      0 1 0 1 0 0 0 1 0 0          倍精度乗算の 1 ステップ目
      0 1 0 1 1 0 1 0 0 0          倍精度乗算の 2 ステップ目
```

このうち単精度乗算については、結果を丸めるべきかどうかは結果の型で暗黙に指定されるので、これは fmul、後は dmul0,1 とする。つまり

```
      フラグ                  命令語

      s r r n s s r n r n
```

```
      h o o o h h o o o o
      i u u r i i u r u r
      f n n m f f n m n m
      t d d a t t d a d a
      5 5 a l 2 5 b l : l
      0 0 : a 5 0 : b : o
      a a : : b b : : : :
      0 0 1 1 0 0 1 1 1 1  fmul          単精度乗算、結果を丸める
      0 0 1 1 0 0 1 1 0 1  fmul          単精度乗算、結果を丸めない
(倍精度で格納)
      0 1 0 1 0 0 0 1 0 0  dmul0         倍精度乗算の 1 ステップ目
      0 1 0 1 1 0 1 0 0 0  dmul1         倍精度乗算の 2 ステップ目
```

それ以外のややこしいことをしたい時にはフルに指定する。

```
u  SHIFT50A   ポート A の仮数を 50 ビット上にシフト
v  ROUND50A   ポート A の仮数を (シフト後)50 ビットに丸める
w  ROUNDA     ポート A の仮数を (シフト後)25 ビットに丸める
A  NORMALA    ポート A の入力を非正規化数とみなす
x  SHIFT25B   ポート B の仮数を 25 ビット上にシフト
y  SHIFT50B   ポート B の仮数を 50 ビット上にシフト
z  ROUNDB     ポート B の仮数を (シフト後)25 ビットに丸める
B  NORMALB    ポート B の入力を非正規化数とみなす
R  ROUND      出力を仮数 25 ビットに丸める
O  NORMALO    出力を正規化しない
```

として、fmul の後にこれらのフラグがあったらそういう処理をすることにする。これらの指定がある時には、オペランドの型からの丸めモード等の推定は行われない。また、オプションが大文字のものはデフォルトが on で指定すると off になるが、小文字のものはデフォルトが off で指定すると on になる。

4.3.4 FADD/SUB

fadd についてはフラグが 5 個しかない。

```
A   ポート A の入力を非正規化数とみなす
B   ポート B の入力を非正規化数とみなす
S   B の符号を反転する (減算)
O   出力を正規化しない
```

として、 fadd[A][B][S][O] という表記をする。 ROUND ビットは結果変数の型から暗黙に指定されるので、命令として指定する必要はない。このようにすることで、通常の加算は fadd と書くことができる。

また、 fadd の代わりに fsub と書くことによっても減算は行える。この場合にもフラグは認識する。

4.3.5 整数演算器

整数演算器については、 i または u (unsigned) が prefix について、

```
add     A+B
sub     A-B
inc     A+1
dec     A-1
not     not A
and     A and B
or      A or B
xor     A xor B
max     max (A,B)
min     min (A,B)
passa   A
passb   B
lsl     A lshiftl B
lsr     A lshiftr B
bsl     A bshiftl B
bsr     A bshiftr B
lnot    logical not A
imm     immediate
```

とする。

即値である immediate は他とは文法が異なる。

```
(u/i)imm source_value store_location [store_location2 ...]
```

の形で、 source_value は 36 ビットまでの 16 進値である。また、この命令は浮動小数点乗算とは並列実行できない。 BM 転送とは並列処理できるところもあるが、面倒なのでこれもサポートしない。

4.3.6 BM との転送

BM との転送は、

- オペランドが 2 つである
- PE から BM への転送では PE 番号を指定する必要がある

という点で他の命令とは違う形式を取る

具体的には

```
bm op1 op2 [peid]
```

という形になり、op1 と op2 のどちらかは BM を指定し、もう一方は PE の記憶域を指定する必要がある。 また、 op1 が PE の記憶域で op2 が BM の場合には、必ず peid を指定する必要がある。この番号で指定された PE のデータが BM に格納される。

PE から BM への転送の際指定できるオペランドは GRF にあるものだけであり、LM や T レジスタを指定することはできない。

4.3.7 マスク

マスクの扱いは疑似命令とする。

```
mi      x:      入力マスクレジスタ番号を指定
mo[f/i] x:      出力マスクレジスタ番号を指定
```

入力マスクはそれ以降の全ての命令に適用される。マスクなし命令にするには、

```
mi 0
```

とすればよい。

出力は次の命令に適用され、次の命令のフラグ出力が指定したマスクレジスタにストアされる。 mof で浮動小数点加減算器、 moi で整数 ALU のフラグ出力が選択される。

4.3.8 ベクトル長

ベクトル長も疑似命令である。

```
vlen    x:      ベクトル長
```

4.3.9 IDP パケット

実際の計算では IDP パケット、 RRN 命令は変数宣言から暗黙に生成されるが、ホストプログラムなしでアセンブラだけでもデバッグのための実行を可能にするため IDP パケットを直接書ことをサポートする。

IDP からのパケットでは、 BM の開始アドレス、語数、 BB 番号、マスク、 SEQ フラグの順番に指定して、開始アドレスには変数名、語数はデフォルトでは 1、BB 番号、マスクはデフォルトでは 0 として、省略可能にする。つまり、以下の記法を許す。

```
IDP varname
FLT 1.0
```

IDP パケットヘッダの本来の形式は

```
IDP len addr bbn bbnmask [seq]
```

というものなので、IDP のすぐあとが数字で始まっていなければこれを変数名と解釈する。長さは、自然なのはその変数の長さ自体である。とりあえずはそれがあれば十分と思われる。従って、

```
IDP varname [bbn [bbnmask]] [SEQ]
```

という形式であるとする。ここで、 SEQ はキーワードで、指定されていると SEQ ビットがたつ。

これは、シミュレータでの実行でのみ評価される。

4.3.10 RRN 命令

IDP パケットと同様に RRN 命令もシミュレータ上で直接実行できるようにする。

```
RRN fadd/ialuop/number memloc length/round/odpoe/sregen
```

で全フィールドが指定する。最後の length/odpoe/sregen は

```
[n][r][o][s]
```

という表記を許す。但し、 n は数字で、 o,s は指定すると 0 になる (デフォルトは 1) ということにする。

4.3.11 デバッグ出力

シミュレータ自体に DBG 命令があって任意のアドレスの中身をダンプできるが、変数名とかを使ってデバッグ出来ないと能率が悪いのでアセンブラレベルでシミュレータに出力を指示する。これは以下の形式を取る

```
print bbid peid format varname
```

パラメータの意味は以下の通り

```
bbid:    BB 番号
peid:    PE 番号
format:  出力形式。 f, i, h が浮動小数点、10 進、16 進となる
varname: 変数名。 $形式も指定できる。
```

これはシミュレータの DBGP 命令に変換される。実機での実行では (今のところ) 無視される。

4.4 制御プロセッサ用タイプ I 拡張

以下、 GRAPE 的制御プロセッサのためのインターフェースライブラリを生成するためのアセンブラ拡張仕様について述べる。より柔軟な制御ができるタイプ II 拡張については次節で述べる。

4.4.1 アセンブラ記述の拡張

ここでは、GRAPE 的な、粒子間相互作用計算に対応した単純な制御プロセッサのための拡張を扱う。

GRAPE モード制御プロセッサの動作では、以下のことが記述できる必要がある

1. 全体初期化ルーチン (定数設定等)
2. ホストから EM への粒子データ転送
3. ホストから LM への i 粒子転送
4. 1 粒子への力の計算前の初期化ルーチン
5. 計算中の EM から LM への j 粒子転送
6. 相互作用計算ルーチン
7. LM からホストへの結果転送

これらのためにアプリケーションプログラマがいちいちなにかコードを書くのは面倒なので、単純なアセンブラに追加したディレクティブのようなものから必要なコード (最終的にはホスト側でのライブラリ関数。当初はシミュレータへのインターフェース) を自動的に生成したい。

以下で、これらをどのように記述するかをまとめる。

このうち、全体初期化は定数転送だけなので、自動的に生成されて別に何か必要なわけではないが、一応念のため指定可能にするために

`initialization`	全体初期化部
`loop initialization`	ループ初期化部
`loop body`	ループ本体部
`loop finalization`	ループ終了処理部

という 4 つの疑似命令を追加する。このどれかの次の命令から次のどれかの直前までが指定したタイプのルーチンということになる。

loop finalization は指定した回数ループを実行した後に一度だけ実行される。

データ転送については、 EM またはホストからの転送が起こることの指定は

```
bvar  {long,short} name  [address value] elt[i] conv_type
```

または

```
var vector {long,short} name  [address value] hlt[i] conv_type
```

となる。 elt は EM から LM、 hlt はホストメモリから LM への転送である。最後のワードは変換タイプで、以下のように定義される。

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

このように記述しておくことで、

1. ホストから EM への粒子データ転送
2. ホストから LM への i 粒子転送
3. 計算中の EM から LM への j 粒子転送

のために必要なコードが自動的に生成される。

elt/hlt の後には数字 (省略時は 0 と解釈) をつけることができる (elt のほうは実装してないかもしれない)。これは、 0 (または数字なし) のものが通常の j または i 粒子データであり、それ以外のものは定数テーブル等である。以下、0(または数字なし) のものを粒子データ、それ以外のものをテーブルデータと呼ぶ。

粒子データについては、メモリへの転送等が自動的に行われる。これに対して、テーブルデータについては転送用の関数は準備されるが、明示的に転送を起動する必要があり、また、どの PE に送るかといった指定も必要になる。

結果回収のほうは

```
var vector {long,short} name [address_value] rrn conv_type  opcode option
```

とする。変換タイプ conv_type は以下の通り。

```
0 flt72to64   72bit 浮動小数点 ->  64bit 浮動小数点 つめて丸める
1 flt36to64   36bit 浮動小数点 ->  64bit 浮動小数点
```

```
 2 fix72to64l  72bit 固定小数点 ->  64bit 固定小数点 下をカット
 3 fix72to64r  72bit 固定小数点 ->  64bit 固定小数点 上をカット
 4 fix72to64w  72bit 固定小数点 ->  36bit 固定小数点 2 語。36 ビットを下
に
```

opcode, option は RRN 命令と同じだが length 指定はない。つまり、

```
 fadd/ialuop/number  round/odpoe/sregen
```

ということになる。なお、オプションに O 指定、つまり、出力が unnormal で、さらに変換タイプが 0 の時には内部的に正規化してからアプリケーションに結果を返すものとする。これは、非正規化数のままアプリケーションに返すと有効数字が若干落ちるからである。

この指定から、結果回収のためのコードが生成される。

この動作のため、 elt, hlt, rrn は「予約語」扱いで変数名には使えない。現状の動作では、変数名に使ってもいいがアドレスフィールドとして利用するとアセンブルエラーになる。

さらに、 i 粒子側のデータ、すなわち hlt, rrn を指定されるデータは全部ベクトルでなければならない。ホスト側のデータ構造体はベクトルではないものを使う。つまり、4 個の粒子データを組み立て直して LM に送るのはライブラリが面倒を見る。 j 粒子側は指定されたデータ構造がそのまま反映される。

4.4.2 制御プロセッサ用拡張の出力形式

4.4.2.1 ファイル名、関数名プレフィックス

もとの vsb ソースの名前が foo.vsb なら foo.c と foo.h という名前でライブラリソースとヘッダファイルが出力されるとする。内部で作られる関数名はアセンブラソース内で指定されたプレフィックスがつく。指定の方法は以下の型の議事命令である。

```
 prefix foo
```

という行があると、 foo がプレフィックスとなる。プレフィックス疑似命令がない場合のデフォルトのプレフィックスは SING である。以下、名称は SING_xxx として生成される関数を解説する。

4.4.2.2 初期化関数

全体初期化の関数である。。これは、ハードウェア獲得の他、定数設定、コードメモリへのデータ格納等を行う。

API は

```
void SING_grape_init()
```

である。

4.4.2.3 EM への粒子データ転送

API は

```
int SING_send_j_particle(struct grape_j_particle_struct *jp,
                         int index_in_EM)
```

である。ここで、 grape_j_particle_struct はヘッダファイル内で定義された構造体の名前であり、その形式は、アセンブラ拡張記述 ”elt” を与えられた変数が、変換前の型で並べられたものである。例えば

```
bvar long xj         elt  flt64to72
bvar long yj         elt  flt64to72
bvar long zj         elt  flt64to72
bvar short mj        elt  flt64to36
```

という記述からは

```
struct grape_j_particle_struct{
  double xj;
  double yj;
  double zj;
  double mj;
};
```

という構造体が生成される。 index_in_EM は EM 内での番号であり、0 からつける。複数の SING チップで並列計算とかする場合には、ここで指定する番号は実際のメモリロケーションとは違うことがありえる。また、チップに複数の放送ブロックがある場合 (普通ある) には、それだけの数の粒子からの力が並列に計算される。このため、送られる粒子の数は放送ブロックの数の整数倍である必要がある。

なお、 elt をつけられるのは、放送メモリに置く変数のみである。つまり、明示的に bvar である必要がある。

4.4.2.4 LM へのデータ転送

LM への転送 API は EM のものとほとんど同じであり、

```
int SING_send_i_particle(struct grape_i_particle_struct *ip,
                         int n)
```

となる。grape_i_particle_struct は ”hlt” 拡張記述から生成される構造体定義であり、

```
var vector long xi   hlt  flt64to72
var vector long yi   hlt  flt64to72
var vector long zi   hlt  flt64to72
```

という記述から

```
struct grape_i_particle_struct{
  double xi;
  double yi;
  double zi;
};
```

という構造体定義が生成される。ここで注意して欲しいのは、 EM 転送の場合と違って、

- hlt を指定する変数はベクトルでなければならない
- しかし、対応する構造体はベクトルではない

ということである。つまり、この例では、 xi, yi, zi は力を受ける粒子の座標であるが、これらはベクトルであり 4 粒子分が宣言される。しかし、アプリケーションプログラム側では 4 個であることはわからなくて、構造体一つに 1 粒子のデータをしまう。データのパッキングはライブラリの中で自動的に行われる。

なお、 n は粒子の数だが、これは放送ブロックにある物理プロセッサの数 × 4 (ベクトル長) 以下でなければならない。これ以上のものを指定した時の結果は不定である。

4.4.2.5 EM/LM へのテーブルデータ転送

LM へのテーブルデータ転送は、 i 粒子転送と以下の点で異なる

- 少なくともデフォルトでは全 PE に同一データを送る。
- ベクトルではないものも送ることができる。

EM へのテーブルデータ転送も LM と同様である。

API は、 LM 転送の場合

```
int SING_send_hlt_data1(struct SING_hlt_struct1 *ip,
                        int peindex);
```

EM 転送の場合

```
int SING_send_elt_data1(struct SING_hlt_struct1 *ip)
```

となる。関数名および変数名の最後の ”1” は任意の正の整数だが、あまり大きくするとアセンブルに時間がかかるようになる。

SING_hlt_struct[k] は ”hlt[k]” 拡張記述から生成される構造体定義であり、

```
var vector long test    hlt1  flt64to72
var vector long testa   hlt1  flt64to72
```

という記述から

struct SING_hlt_struct1{

```
double test[4];
double testa[4];
```

};

という構造体定義が生成される。 EM 転送の場合には hlt ではなく elt である。

なお、 peindex は現在のところ 0 でなければならない。この時に全 PE に同一データが放送される。 0 以外は拡張のために用意されている。

なお、この関数を呼んだ後はデータ変換テーブルが破壊されるので、その後に i 粒子転送を行う時は、その前に変換テーブル再設定の関数

```
SING_register_i_particle_conversion()
```

を呼ぶ必要がある。これは引数は不要であり、 hlt 記述から生成された i 粒子用変換テーブルをロードする。 elt の場合にはこの必要はない。但し、現在のシミュレータ実装では 結果回収 (SING_get_result) 実行の後に EM に書かれたデータが破壊される。これはそのうちに修正する。

4.4.2.6 計算実行

計算をスタートさせる関数の API は

```
int SING_grape_run(int n)
```

である。引数は EM にしまった粒子データの数であるが、放送ブロック数でわった数を指定する必要がある。

この関数が呼ばれると、

1. 計算前の初期化ルーチンを実行する
2. EM から 0 番目の粒子データを SING チップに送る
3. ループ本体を実行する
4. 本体の実行中、次の粒子データを送ってよいところまできたら転送を始める
5. 本体の実行が終わり、指定した繰り返し数が終わっていれば終了。そうでなくて次の粒子データの転送も終わっていたら 3 に戻る

という動作をする。

4.4.2.7 結果回収

API は

```
int SING_get_result(struct grape_result_struct *rp)
```

である。構造体定義は rrn 拡張記述から、 hlt, elt の場合と同様に生成される。帰ってくるデータの数は SING_send_i_particle で指定したものである。また、 SING_send_i_particle と同様に rrn 変数はベクトル型である必要があり、それがアプリケーション側では見えない。つまり、例えば

```
var vector long accx rrn  flt72to64 fadd
var vector long accy rrn  flt72to64 fadd
var vector long accz rrn  flt72to64 fadd
```

という記述から

```
struct grape_result_struct{
  double accx;
  double accy;
  double accz;
};
```

という構造体が生成される。

4.5 制御プロセッサ用拡張 (2)

ここでは、 IDP, RRN を制御プロセッサからの命令実行と並列に起動できる、GRAPE 版よりもより柔軟な拡張について述べる。以下、これをタイプ II 拡張と呼ぶ。タイプ I 拡張は GRAPE 的インターフェース専用である。なお、アセンブラは両方の拡張をサポートするべきものとする。

制御プロセッサモデルはタイプ II では拡張されるが、その機能はタイプ I 拡張を表現するのに十分なので、タイプ II 対応制御プロセッサでタイプ I 拡張のためのコードを生成できるものとする。タイプ I、II のどちらであるかはアセンブラにオプションを与える。

```
-g: type1 拡張
-t: type2 拡張
```

とりあえず、 type1 拡張の時には上の GRAPE の時と同じ動作である。

4.5.1 概要

タイプ II 拡張においては、以下を可能にする

- ISP からの命令実行と IDP, RRN での転送の同時実行。これを、アセンブラレベルで明示的に書く。
- IDP 転送を明示的に指定するため、EM 上の領域指定を可能にする。

以下、まず EM 上の領域指定について、次に IDP, RRN の表記についてまとめる。

4.5.1.1 EM 上領域指定

EM 領域は以下のように宣言する

```
evar name eltnumber size
```

ここで evar はキーワードで、この行が EM 領域であることを指定する。name は領域名で、転送指定の時に用いられる。 eltnumber は 0 または正の整数で、 bvar につけた elt id (指定しない場合は 0) に対応する。最後は確保する領域の大きさであり、 実際の大きさは BM 上の領域のサイズに size をかけたものになる。例えば

```
bvar long xj         elt  flt64to72
bvar long yj         elt  flt64to72
bvar long zj         elt  flt64to72
bvar short mj        elt  flt64to36
evar jparticle 0 16384
```

とすれば、 EM 上の jparticle という名前の領域に、上の xj, yj, zj, mj の 4 変数を単位とする領域が 16384 個とられることになる。

```
bvar short density   elt1  flt64to36
bvar short pressure  elt1  flt64to36
evar jsph 1 16384
```

とすれば、 density, pressure を単位とする領域がやはり 16384 個とられる。

以下では、 BM 上のデータ領域、ここでは xj, yj, vj, mj の組や density, pressure の組を、 BM データ構造体と呼ぶ。

4.5.1.2 IDP 転送指定

idp 転送指定は以下の形式を取る。

idp eltnumber/name [bcastmode] [bbid]

idp はキーワードである。 eltnumber/name は EM/BM 領域名を指定する。動作は、 isp 命令列が一度実行される度に、EM 領域から nbb (放送ブロックの数) だけの BM データ構造体を読み出して BB の BM に 1 つづつ書く。

bcastmode と bbid はオプショナルなパラメータで、省略可能である。これらは

```
bcastmode=0  通常動作
          1  放送/singlewrite、ループ毎にインクリメントする
          2  放送/singlewrite、ループ毎にインクリメントしない
bbid >=0  放送/singlewrite モードの時にその bb にだけ書く
     <0   本当に放送
```

となる。通常動作では、 idp 命令が実行されるたびに放送ブロックの数だけ領域が読み込まれ、各放送ブロックには違うものが送られる。つまり、 EM 上の領域を e0, e1, ... とし、放送ブロックが 0 ... p-1 の p 個あるとすると、 idp を最初に呼んだ時には 放送ブロック 0 ... p-1 に e0 ... e[p-1], 次に呼んだ時には e[p] ... e[2p-1], という具合に、先に進んでいく。これに対して bcastmode=1 では idp 実行毎に先に進むのは同じだが、 p 個ではなく 1 個しか領域を読まない。また、bcastmode=2 では毎回同じものを 1 つだけ EM 領域から BM に転送する。省略時の値は bcastmode=0 であり、これがデフォルト動作である。

bbid は、 bcastmode=0 の時には無視され、 1 または 2 の時にだけ意味をもつ。値が正または 0 の時には、指定した放送ブロックだけにデータが書かれる。値が負、または省略された時には、放送となって全 BB に同じものがかかれる。

idp 転送指令はアセンブラの実行命令と同じ行に書くと、並行動作する。idp 命令は PE の命令とは独立のポートから投入されるため、 PE 命令サイクルを消費しない。但し、転送指令投入から実際の実行終了までは IDP ポートを占有する。従って、 IDP 転送指令が投入されてから終了までは、次の IDP 指令も RRN 指令も投入することができない。

アセンブラは現在のところ転送指令にかかるサイクルを計算しないので、idp 命令が実際には実行不可能でもエラーをださない。

4.5.1.3 RRN 転送指定

RRN 命令のほうは、シミュレータへの RRN 命令と同様な表記になる。具体的には

```
rrn fadd/ialuop/number memloc   length/round/odpoe/sregen convtype [name]
```

となる。 length/odpoe/sregen は

```
[n][r][o][s]
```

という表記を許すことにする。但し、 n は数字で、 o,s は指定すると 0 になる (デフォルトは 1) ということにする。また、 memloc は BM 領域指定でなければならない。 length は指定しなければ変数のタイプによって決まる。

name は指定されていれば C 言語ライブラリのほうで生成される構造体のメンバ名となる。

4.5.2 タイプ II 拡張の出力形式

4.5.2.1 ファイル名、関数名プレフィックス

GRAPE 拡張と同じ。デフォルトプレフィックスは SING である。

4.5.2.2 初期化関数

```
void SING_grape_init()
```

である。

4.5.2.3 EM への J 粒子データ転送

API は

```
int SING_elt_data[i](struct SING_elt_struct[i] *jp,
                     int index_in_EM)
```

である。ここで、 SING_elt_struct[i] はヘッダファイル内で定義された構造体の名前であり、その形式は、アセンブラ拡張記述 ”elt” を与えられた変数が、変換前の型で並べられたものである。[i] は 0, 1 等の整数で、アセンブラ拡張記述で与えた elt 番号である。例えば

```
bvar long xj          elt  flt64to72
bvar long yj          elt  flt64to72
bvar long zj          elt  flt64to72
bvar short mj         elt  flt64to36
```

という記述からは

```
struct SING_elt_struct0{
  double xj;
  double yj;
  double zj;
  double mj;
};
```

という構造体が生成される。 index_in_EM は EM 内での番号であり、0 からつける。複数の SING チップで並列計算をする場合には、ここで指定する番号は実際のメモリロケーションとは違うことがありえる。また、チップに複数の放送ブロックがある場合 (普通ある) には、それだけの数の粒子からの力が並列に計算される。このため、送られる粒子の数は放送ブロックの数の整数倍である必要がある。

なお、 elt をつけられるのは、放送メモリに置く変数のみである。つまり、明示的に bvar である必要がある。

以前のバージョンとの互換性のために、生成されるヘッダファイル内で

```
#define grape_j_particle_struct  SING_elt_struct0
```

という定義がされており、構造体には grape_j_particle_struct という別名がついている。但し、複数の関数を使う場合にはこのマクロ定義は上書きされるので、grape_j_particle_struct は使うべきではない。もしもアプリケーションプログラム中で使うと、最後に読み込んだヘッダファイルの中での程度が使われる。

4.5.2.4 LM へのデータ転送

LM への転送 API は EM のものとほとんど同じであり、

```
int SING_send_i_particle(struct SING_hlt_struct[i] *ip,
                         int n)
```

となる。SING_hlt_struct[i] は ”hlt” 拡張記述から生成される構造体定義であり、

```
var vector long xi   hlt  flt64to72
var vector long yi   hlt  flt64to72
var vector long zi   hlt  flt64to72
```

という記述から

```
struct SING_hlt_struct0{
  double xi;
  double yi;
  double zi;
};
```

という構造体定義が生成される。ここで注意して欲しいのは、 EM 転送の場合と違って、

- hlt を指定する変数はベクトルでなければならない
- しかし、対応する構造体はベクトルではない

ということである。つまり、この例では、 xi, yi, zi は力を受ける粒子の座標であるが、これらはベクトルであり 4 粒子分が宣言される。しかし、アプリケーションプログラム側では 4 個であることはわからなくて、構造体一つに 1 粒子のデータをしまう。データのパッキングはライブラリの中で自動的に行われる。

なお、 n は粒子の数だが、これは放送ブロックにある物理プロセッサの数 $\times$ 4 (ベクトル長) 以下でなければならない。これ以上のものを指定した時の結果は不定である。

EM 転送の場合と同様に、以前のバージョンとの互換性のために、生成されるヘッダファイル内で

```
#define grape_i_particle_struct  SING_hlt_struct0
```

というマクロ定義がなされている。

なお、現行のバージョンは複数の hlt 領域に対応していない。今後対応の予定である。

4.5.2.5 LM へのテーブルデータ転送

LM へのテーブルデータ転送は、 i 粒子転送と以下の点で異なる

- 少なくともデフォルトでは全 PE に同一データを送る。
- ベクトルではないものも送ることができる。

API は、

```
int SING_send_hlt_data1(struct SING_hlt_struct1 *ip,
                        int peindex);
```

となる。関数名および変数名の最後の ”1” は任意の正の整数だが、あまり大きくするとアセンブルに時間がかかるようになる。

SING_hlt_struct[k] は ”hlt[k]” 拡張記述から生成される構造体定義であり、

```
var vector long test    hlt1  flt64to72
var vector long testa   hlt1  flt64to72
```

という記述から

```
struct SING_hlt_struct1{
  double test[4];
  double testa[4];
};
```

という構造体定義が生成される。

なお、 peindex は現在のところ 0 でなければならない。この時に全 PE に同一データが放送される。 0 以外は拡張のために用意されている。

なお、この関数を呼んだ後はデータ変換テーブルが破壊されるので、その後に i 粒子転送を行う時は、その前に変換テーブル再設定の関数

```
SING_register_i_particle_conversion()
```

を呼ぶ必要がある。これは引数は不要であり、 hlt 記述から生成された i 粒子用変換テーブルをロードする。また、現在のシミュレータ実装では 結果回収 (SING_get_result) 実行の後に EM に書かれたデータが破壊される。これはそのうちに修正する。

4.5.2.6 計算実行

計算をスタートさせる関数の API は

```
int SING_type2_run(int n, void * outp)
```

である。第一引数は EM にしまった粒子データの数であるが、放送ブロック数でわった数を指定する必要がある。第二引数は結果をしまう領域への先頭ポインタである。

この関数が呼ばれると、

1. 計算前の初期化ルーチンを実行する
2. EM から 0 番目の粒子データを SING チップに送る
3. ループ本体を実行する

4. 本体の実行中、次の粒子データを送ってよいところまできたら転送を始める
5. 本体の実行が終わり、指定した繰り返し数が終わっていれば終了。そうでなくて次の粒子データの転送も終わっていたら 3 に戻る

という動作をする。

この間に、 rrn 命令が実行された出力結果は outp が指す領域に格納される。領域のオーバーラン等のチェックはしないので注意。

4.6 マクロ定義

アセンブラ内でマクロやループ展開ができると便利である。そのために言語仕様を決めるのは煩雑なので、 vsm.rb の中で *erb*[1] を使うことにする。erb 自体の仕様はここでは詳しくは述べないが、例えば

```
upassa $lm0v<%=stride%> $lr0v $lr0v
```

と書くことで、ストライドをアセンブル時に決まる変数とすることができる。書式としては、 ¡%= xxx %¿ の中に書かれた xxx がアセンブル時に Ruby によって評価される。 従って、上の例で意味があるアセンブルができるためには stride というここでは普通の変数に値を与える必要がある。ソースプログラムの中では

```
<% stride=2 %>
```

と stride を評価する行より上に書いておけばいい。アセンブル実行時に値を与えるには vsm.rb の -e オプションを使って

```
ruby vsm.rb -e stride=2 -i foo.rb ...
```

とする。 -e オプションで与えたものは最初に評価されるので、その後で上書きすることができる。 指定したらその値がはいって、指定しなければデフォルトが使われるようにしたければ

```
<%
unless defined? stride
  stride=1
end
%>
```

というように定義されていなければ定義する、というコードを書いておけばよい。以下に、多少高度な例を示す。

---

[1] http://www.ruby-lang.org/ja/man/index.cgi?cmd=view;name=ERB

```
## sreadtest.vsm
##
## Time-stamp: <2006/02/18 17:04:21 makino>
##
## test program for stride read
prefix readtest
bvar vector long index elt  fix64to72rs
bvar vector long outdata

<%
unless defined? stride
  stride=1
end
unless defined? first
  first=0
end
a=""
64.times{|i| a += "var vector long a#{i} hlt flt64to72\n"}
%>
<%=a%>

evar jdata  0 8192
vlen 4
loop body
idp 1 1 -1
nop 2
upassa $lm<%=first%>v<%=stride%> $lr0v $lr0v
<%
p first
p stride
x=first+stride*8
%>
upassa $lm<%=x%>v<%=stride%> $lr8v $lr8v
nop
bm $lr0v outdata 0
nop
rrn umin outdata 4 flt72to64
nop 3
bm $lr8v outdata 0
nop
rrn umin outdata 4 flt72to64
nop 3
```

ここでは、 Ruby のイテレータを使って 64 個の変数を生成している。

```
var vector long a0 hlt flt64to72
...
var vector long a63 hlt flt64to72
```

が

```
<%
a=""
64.times{|i| a += "var vector long a#{i} hlt flt64to72\n"}
%>
<%=a%>
```

から生成されている。なお、 stride はついでにデフォルトを書いただけである。

4.7 サンプルコード

ここでは、拡張記述を使わない単純な例を考える。拡張記述を含む全体は次の節で扱う。

ある数の -1.5 乗を計算することを考える。入力は bm からコピーするとする。計算方法は線形補間とニュートン法の組合せ。つまり、初期値は以下の方法で計算する。

- 仮数が 1 なら初期値はもちろん 1
- 仮数が 2 の時に初期値は 0.35 くらい

これで最大誤差は 20% くらいなので、10% くらい小さくしておく。つまり、1 の時に 0.9, 2 の時に 0.33 くらいになるように、 0.9-(x-1)*0.57 を初期値にする。整理すると、1.47-x*0.57

指数は、オフセットが 0x3FF なので、

1. 3FF を引く
2. 引いた答を 1.5 倍して 3FF から引く。結果はここでは切り捨てだとする。
3. 答が 3FF より小さい時、元の指数が偶数なら上の初期値に $1/\sqrt{2}$ を掛ける
4. 答が 3FF より大きい時、元の指数が偶数なら上の初期値に $\sqrt{2}$ を掛ける

ということになる。例えばオフセットが 9 だったとして、

```
指数 表現   1.5 倍 13 から引く      ずれ    22 から引く
-3    6      9        4        -0.5       13
-2    7     10        3         0         12
```

```
-1    8    12       1        -0.5      10
 0    9    13       0         0         9
 1   10    15      -2        -0.5       7
 2   11    16      -3         0         6
 3   12    18      -5        -0.5       4
```

となるので、 3FF を 2.5 倍したものから 1.5 倍を引いて、もとの指数が奇数なら初期値を $\sqrt{2}$ 倍にする。

ニュートン法の近似式は、 $a = r^2$ として、

$$x_1 = \frac{x(3 - a^3x^2)}{2} \tag{1}$$

である。一応、定数マクロを使わないで BM に値を設定するところからアセンブラで書くとすると、

```
 bvar vector long br2
 bvar vector long result
 var vector short x

IDP br2
FLT 1
FLT 2
FLT 10000
FLT 1e50
print 0 0 f br2
DBG BM 0 0 3
vlen 4
bm br2 x
 upassa x $lr0v $lr0v
 ulsr $lr0v      il"60"  $t $lr0v
DBG BB 0 0 TREG
 ulsr $ti     il"1"   $lr8v
 uadd $ti     $lr8v  $t
 usub hl"9fd" $t     $t       # $lr8v は指数の 1.5 倍
 ulsl $ti     il"60"  $lr8v
DBG BB 0 0 GRF 4 7
 moi 1
 uand il"1" $lr0v
DBG BB 0 0 MREG 1
 moi 0
 upassa x $lr0v $lr0v
 uand $lr0v h"000ffffff" $t
 uor  $ti h"3ff000000" $t
 fmul $ti f"0.57" $t
```

```
 fsub f"1.57" $t $t
 mi 1
 fmul f"1.414" $t $t
 mi 0
 fmul $t $lr8v $t $r0v # Here the result is the initial guess
print 0 0 f $r0v
 ipassa x $lr0v $t
 fmul $t $t $r4v
 fmul $t $r4v $ti
 fmul $t f"0.5" $r4v # r4v is a**3/2
 fmul $r0v $r0v $t
 fmul $ti $r4v $t
 fsub f"1.5" $ti $t
 fmul $r0v $ti $t $r0v
 print 0 0 f $t
 fmul $ti $ti $t
 fmul $ti $r4v $t
 fsub f"1.5" $ti $t
 fmul $r0v $ti $t $r0v
 print 0 0 f $t
 fmul $ti $ti $t
 fmul $ti $r4v $t
 fsub f"1.5" $ti $t
 fmul $r0v $ti $t $r0v
 print 0 0 f $t
 fmul $ti $ti $t
 fmul $ti $r4v $t
 fsub f"1.5" $ti $t
 fmul $r0v $ti $t $r0v
 print 0 0 f $t
 fmul $ti $ti $t
 fmul $ti $r4v $t
 fsub f"1.5" $ti $t
 fmul $r0v $ti $t $r0v
 print 0 0 f $t
 bm $t result 0
print 0 0 f result
RRN fadd result
```

これは、ループ部分の定義がないのでこのままでアセンブル・インターフェース関数の生成ができるわけではない。シミュレータ専用のコードである。

4.8 GRAPE 実現サンプル

この節では、単純な GRAPE の実現を例にとってどのようなコードを書き、それをどのように使うかを解説する。

アセンブラコードは以下のようなものになる

```
 var vector long xi   hlt  flt64to72
 var vector long yi   hlt  flt64to72
 var vector long zi   hlt  flt64to72
 bvar long xj         elt  flt64to72
 bvar long yj         elt  flt64to72
 bvar long zj         elt  flt64to72
 bvar long vxj xj
 bvar short mj        elt  flt64to36
 bvar short eps2      elt  flt64to36
 var short  lmj
 var short  leps2
 var vector long accx rrn  flt72to64 fadd
 var vector long accy rrn  flt72to64 fadd
 var vector long accz rrn  flt72to64 fadd
 var vector long pot  rrn  flt72to64 fadd
 loop initialization
 vlen 4
 uxor $t $t $t
 upassa $ti $ti $lr40v
 upassa $t $t   $lr48v
 upassa $t $t   $lr56v
 upassa $t $t   pot
 loop body
 vlen 3
 bm vxj $lr0v
 vlen 1
 bm mj lmj
 bm eps2 leps2

# print 0 0 f xj
# print 0 0 f yj
# print 0 0 f zj
# print 0 0 f $lr0v
# print 0 0 f lmj
 vlen 4
 nop 2
 fsub $lr0 xi $r6v  $t
 fsub $lr2 yi $r10v ;  fmul $ti $ti $t
 fsub $lr4 zi $r14v
```

```
 fmul $r10v $r10v $r18v ; fadd $t leps2 $t
 fmul $r14v $r14v $r22v
 fadd $t $r18v $t

#print 0 0 h $lr0
#print 0 0 h xi
#print 0 0 h $r6v
#print 0 0 f $lr4
#print 0 0 f zi
#print 0 0 f $r6v
#print 0 0 f $r10v
#print 0 0 f $r14v
#print 0 0 f $r18v
 fadd $ti $r22v $r18v $t # rsq is now in r18 t, dx, dy,dz are in  6,10,14
# print 0 0 f $r18v
 ulsr $ti     il"60"  $t $lr22v
#print 0 0 h $t
 ulsr $ti     il"1"   $t
 uadd $lr22v     $ti  $t
 usub hl"9fd" $ti     $t       # $lr8v は指数の 1.5 倍
 ulsl $ti     il"60"  $lr30v
 moi 1
 uand il"1" $lr22v
 moi 0
 uand $r18v h"000ffffff" $t
 uor  $ti h"3ff000000" $t
 fmul $ti f"0.57" $t
#print 0 0 f $t
 fsub f"1.57" $ti $t
 mi 1
 fmul f"1.414" $ti $t
 mi 0
 nop
 fmul $t $lr30v $t $r22v # Here the result is the initial guess
#print 0 0 f $r22v
 fmul $r18v $r18v $r26v $t
 fmul $r18v $ti   $r26v $t
 fmul $ti f"0.5" $r26v # r26v is a**3/2
 fmul $r22v $r22v $t
 fmul $ti $r26v $t
 fsub f"1.5" $ti $t
 fmul $r22v $ti $t $r22v
 fmul $ti $ti $t
 fmul $ti $r26v $t
 fsub f"1.5" $ti $t
 fmul $r22v $ti $t $r22v
```

```
 fmul $ti $ti $t
 fmul $ti $r26v $t
 fsub f"1.5" $ti $t
 fmul $r22v $ti $t $r22v
 fmul $ti $ti $t
 fmul $ti $r26v $t
 fsub f"1.5" $ti $t
 fmul $r22v $ti $t $r22v
 fmul $ti $ti $t
 fmul $ti $r26v $t
 fsub f"1.5" $ti $t
 fmul $r22v $ti $t  # $t is the resulted r2**(-3/2)
# print 0 0 f $t
 fmul lmj $ti $t $r22v
 fmul $r6v $ti $t ; upassa pot pot $lr0v
 fadd $lr40v $ti $lr40v accx
 fmul $r10v $r22v $t
 fadd $lr48v $ti $lr48v accy
 fmul $r14v $r22v $t
 fadd $lr56v $ti $lr56v accz
 fmul $r18v $r22v $t
 fadd $lr0v $ti pot
# print 0 1 f accx
# print 0 1 f accy
# print 0 1 f accz
#  print 0 0 f pot
#  print 1 0 f pot
```

これは、粒子 xj から粒子 xi へのニュートン重力を計算する、基本的な GRAPE を実現するアセンブラコードである。

順番に説明していこう。

```
var vector long xi   hlt  flt64to72
var vector long yi   hlt  flt64to72
var vector long zi   hlt  flt64to72
```

これは重力を受ける粒子、いわゆる i 粒子のデータの宣言となる。ここではベクトル形浮動小数点長語で xi, yi, zi を宣言している。

```
bvar long xj         elt  flt64to72
bvar long yj         elt  flt64to72
bvar long zj         elt  flt64to72
bvar long vxj xj
bvar short mj        elt  flt64to36
```

```
var short  lmj
var short leps2
```

これは j 粒子、つまり力を及ぼす側の粒子データの宣言である。これらは全て放送メモリ上におかれる。ここで

```
bvar long vxj xj
```

は、 xj に対して vxj という別名を与えている。これは、 vxj の 0, 1, 2 番目の要素が xj, yj, zj に対応することになる。アセンブラの変数に対するアドレス割り当ては、宣言した順番に行われる。

```
var vector long accx rrn  flt72to64 fadd
var vector long accy rrn  flt72to64 fadd
var vector long accz rrn  flt72to64 fadd
```

これは回収される変数の宣言である。

```
 loop initialization
```

これは、この部分はループ初期化であり、 SING_grape_run を呼んだ時に最初に一度実行されることを示す。

```
vlen 4
```

これはベクトル命令の命令長を指定する疑似命令である。

```
uxor $t $t $t
```

t レジスタをクリアする

```
upassa $ti $ti $lr40v
upassa $t $t   $lr48v
upassa $t $t   $lr56v
upassa $t $t   pot
```

クリアした結果を汎用レジスタに格納する。これらは計算結果の加速度をしまうアキュムレータとなる。

```
loop body
```

ここからがループ本体という指定である。

```
vlen 3
```

ベクトル長を 3 とする。

```
bm vxj $lr0v
```

vxj を汎用レジスタ 0-2 番地にコピー

```
vlen 1
bm mj lmj
bm eps2 leps2
```

mj, eps2 をローカルメモリにコピー

```
vlen 4
nop
```

ローカルメモリに書き込んだ直後の命令ではローカルメモリから読み出せないので、ここで nop を入れる。

```
fsub $lr0 xi $r6v  $t
fsub $lr2 yi $r10v ;  fmul $ti $ti $t
fsub $lr4 zi $r14v
```

座標の差を計算する。で、結果の成分の 2 乗も計算できるところから計算始める。

```
fmul $r10v $r10v $r18v ; fadd $t leps2 $t
fmul $r14v $r14v $r22v
fadd $t $r18v $t
fadd $ti $r22v $r18v $t # rsq is now in r18 t, dx, dy,dz are in  6,10,14
```

多分ここまでで距離の 2 乗が計算できている

```
ulsr $ti     il"60"  $t $lr22v
ulsr $ti     il"1"   $t
uadd $lr22v     $ti  $t
usub hl"9fd" $ti     $t       # $lr8v は指数の 1.5 倍
ulsl $ti     il"60"  $lr30v
moi 1
uand il"1" $lr22v
moi 0
uand $r18v h"000ffffff" $t
```

```
 uor  $ti h"3ff000000" $t
 fmul $ti f"0.57" $t
 fsub f"1.57" $ti $t
 mi 1
 fmul f"1.414" $ti $t
 mi 0
 nop
 fmul $t $lr30v $t $r22v # Here the result is the initial guess
```

指数、仮数にややこしい操作をして -3/2 乗の初期推定を求めている。

```
 fmul $r18v $r18v $r26v $t
 fmul $r18v $ti   $r26v $t
 fmul $ti f"0.5" $r26v # r26v is a**3/2
 fmul $r22v $r22v $t
 fmul $ti $r26v $t
 fsub f"1.5" $ti $t
 fmul $r22v $ti $t $r22v
 fmul $ti $ti $t
 fmul $ti $r26v $t
 fsub f"1.5" $ti $t
 fmul $r22v $ti $t $r22v
 fmul $ti $ti $t
 fmul $ti $r26v $t
 fsub f"1.5" $ti $t
 fmul $r22v $ti $t $r22v
 fmul $ti $ti $t
 fmul $ti $r26v $t
 fsub f"1.5" $ti $t
 fmul $r22v $ti $t $r22v
 fmul $ti $ti $t
 fmul $ti $r26v $t
 fsub f"1.5" $ti $t
 fmul $r22v $ti $t  # $t is the resulted r2**(-3/2)
```

ニュートン反復して収束させている。

```
 fmul lmj $ti $t $r22v
```

質量を掛ける

```
 fmul $r6v $ti $t ; upassa pot pot $lr0v
```

xj-xi を掛ける。並行してポテンシャルをレジスタにコピーしておく。

```
fadd $lr40v $ti $lr40v accx
fmul $r10v $r22v $t
```

積算する。で、ローカル変数にも同時に値を格納する。以下は y, z, pot に同様な処理。

```
fadd $lr48v $ti $lr48v accy
fmul $r14v $r22v $t
fadd $lr56v $ti $lr56v accz
fmul $r18v $r22v $t
fadd $lr0v $ti pot
```

では、これを使うコードはというと、テスト用コードを以下に示す。

```
//
// testsimplegravity.c
// test driver for simplegravity.vsm-generated interface library
//
// J. Makino
//    Time-stamp: <05/03/15 21:41:35 makino>

#include <stdio.h>
#include "simplegravity.h"

#define N 2

struct grape_j_particle_struct xj[N];
struct grape_i_particle_struct xi[N*4];
struct grape_result_struct result[N*4];

void init()
{
    int i;
    xj[0].xj=0;
    xj[0].yj=0;
    xj[0].zj=0;
    xj[0].mj=0;
    xj[0].eps2=0;
    xj[1]=xj[0];
    xj[1].xj=0;
    xj[1].mj=1;
    for(i=0;i<N;i++)SING_send_j_particle(xj+i, i);

    xi[0].xi=0;
    xi[0].yi=0;
```

```
    xi[0].zi=0;
    for(i=1;i<(N*4);i++)xi[i]=xi[0];
    for(i=0;i<(N*4);i++)xi[i].xi=0;
    for(i=0;i<(N*4);i++)xi[i].yi=1.0/(i+1.0);
    for(i=0;i<(N*4);i++)xi[i].zi=0;
    for(i=0;i<(N*4);i++){
        fprintf(stderr,"test init i=%d, x, y, z= %e %e %e\n",
                i,xi[i].xi,xi[i].yi,xi[i].zi);
    }
}

print_result(struct grape_result_struct * result)
{
    struct grape_result_struct *p;
    int i;
    p=result;
    for(i=0;i<(N*4);i++){
        fprintf(stderr,"test result i=%d, x, y, z, pot= %e %e %e %e\n",
                i,p->accx,p->accy,p->accz,p->pot);
        p++;
    }
}

int main()
{
    int i, niparallel, ni, nj;
    ni=N*4;
    nj=N;
    SING_grape_init();
    init();
    niparallel = SING_number_of_pe()*4;
    for (i=0; i< ni;i+=niparallel){
        int njreal = nj/SING_number_of_bb();
        SING_send_i_particle(xi+i, niparallel);
        printf("i, njreal=%d %d\n", i, njreal);
        SING_grape_run(njreal);
        SING_get_result(result+i);
    }
    print_result(result);

}
```

本体は

```
 int main()
```

```
{
    SING_grape_init();
    init();
    SING_send_i_particle(xi, N*4);
    SING_grape_run(1);
    SING_get_result(result);
}
```

である。 init() の中では粒子データを設定して SING_send_j_particle を呼んでいる。これは一度計算するだけだが、粒子を入れ替えて何度も呼ぶなら

```
    SING_grape_init();
    ...
    for (...){ // 時間積分のループ
       for(... )SING_send_j_particle(xj+i, i); // 全粒子を送る
       for(...){ //NPE*4 粒子づつ力を計算
           SING_send_i_particle(xi, N*4);
           SING_grape_run(n/NBB);
           SING_get_result(result);
       }
// 軌道積分とか
    }
```

という感じになる。

5 アセンブル方法

実行は、 ruby プログラム vsm.rb による。以下にヘルプ出力を示す。

```
|gravity> ruby vsm.rb --help

    Assembler program for VSM --- VPM assembly language
    (c) 2004- Jun Makino

    This program assembles the input file written in VSM, and generates
    output machine code files. Maximum help is provided by the command
    "ruby vsm.rb --help".

-g  --grape_output: Output file name for grape type output [default: ]

    This option specifies the file name for GRAPE-type output mode. If
    not specified, GRAPE-type output is not generated.
```

```
-t  --type2_output: Output is type 2

    This option specifies that the created code is type 2, which means
    default generation of idp etc are suppressed.

-k  --koike_runtime_support: Output is for runtime support by Koike

    This option specifies that the created code is not for simulator
    but for the runtime code by Koike.

-i  --input: Input VSM file name [default: ]

    This option specifies the file name for input VSM file. If
    not specified, STDIN is used instead.

-p  --number_of_pes: Number of PEs [default: 2]

    This option specifies the number of PEs per BB, which is used in automatic
    generation of the finalization code.

-e  --erb_text: Text evaluated by erb [default: ]

    This option specifies the ruby script which is evaluated in erb at the top
    of the source file.

-h  --help: Help facility

    When providing the command line option -h, followed by one or more
    options, a one-line summary of each of the options will be printed.

    These options can be specified in either short or long form, i.e
    typing "some_command -h -x" or "some_command -h --extra" will produce
    the same output, if "-x" and "--extra" invoke the same option.

    When providing the command line option -h, with nothing else following,
    a one-line summary of all options will be printed.

    When providing the command line option --help, instead of -h, the same
    actions occur, the only difference being that instead of a one-liner,
    a longer description will be printed.

    Anything that appears on the command line between the name of the
    program and "-h" or "--help" will be ignored.  For example, typing
    "some_command gobbledygook -h -x" will produce the same output as
    typing "some_command -h -x"
```

```
---help: Program description (the header part of --help)

    This option prints only the header part of what would be printed with
    the option --help, without printing all the information about specific
    option.  In other words, it provides only the general information about
    the program itself.
```

これはオプションの完全なリストである。概要を述べると、入力 (拡張子は.vsm) は -i オプションの後に指定する。出力タイプは -g/-t で指定する。 -g を指定した時には -p によって PE 数を指定する必要がある。チップができたらデフォルトを 32 にするのでおそらく普段はデフォルトで良い。

なお、 -e オプションは、プログラムの先頭で評価される erb プログラムテキストを指定する。

実際にハードウェアで実行できるライブラリを作る方法はこれから書く。

6 用語集

- i 粒子: 相互作用を受ける粒子。この粒子への相互作用 (例えば重力) を積算する、というのが SING チップの基本的動作モードである。放送ブロック内の PE はそれぞれ違う i 粒子への力を計算するが、全ての放送ブロックは同じ i 粒子の集合への、違う j 粒子の集合からの相互作用を計算し、出力時に加算する。
- j 粒子: 相互作用を及ぼす粒子。外部メモリに格納され、計算時に計算と並行して放送メモリに転送される。
- EM: 外部メモリ。SING チップ内ではなくボード上にある DRAM

~