CEDEC 2007

ゲーム開発にもきっと役立つ Cell/B.E.プログラミングの基礎

CEDEC 2007

株式会社フィックスターズ
安田 絹子 (Kinuko Yasuda)

Fixstars Corporation
Nisshin Building 3F, 1-8-27, kounan, Minato-ku, Tokyo 108-0075, Japan
Tel:+81-3-5781-5001 Fax:+81-3-5781-5002

概要

- Cell/B.E.のアーキテクチャ、プログラミングの概要やモデル、テクニックについて基礎的なところをお話します
  - ゲーム開発にはものすごくは役に立たないかも...

- Cell/B.E. (Cell Broadband Engine™)とは...
  - PLAYSTATION®3のプロセッサ
  - 2種類・7個 (1 + 6個) のプロセッサコアを利用可能
  - 単精度浮動小数点で200 GFlops以上のプロセッサパワー
  - (余談) Linuxでもプログラミングを楽しむことができます

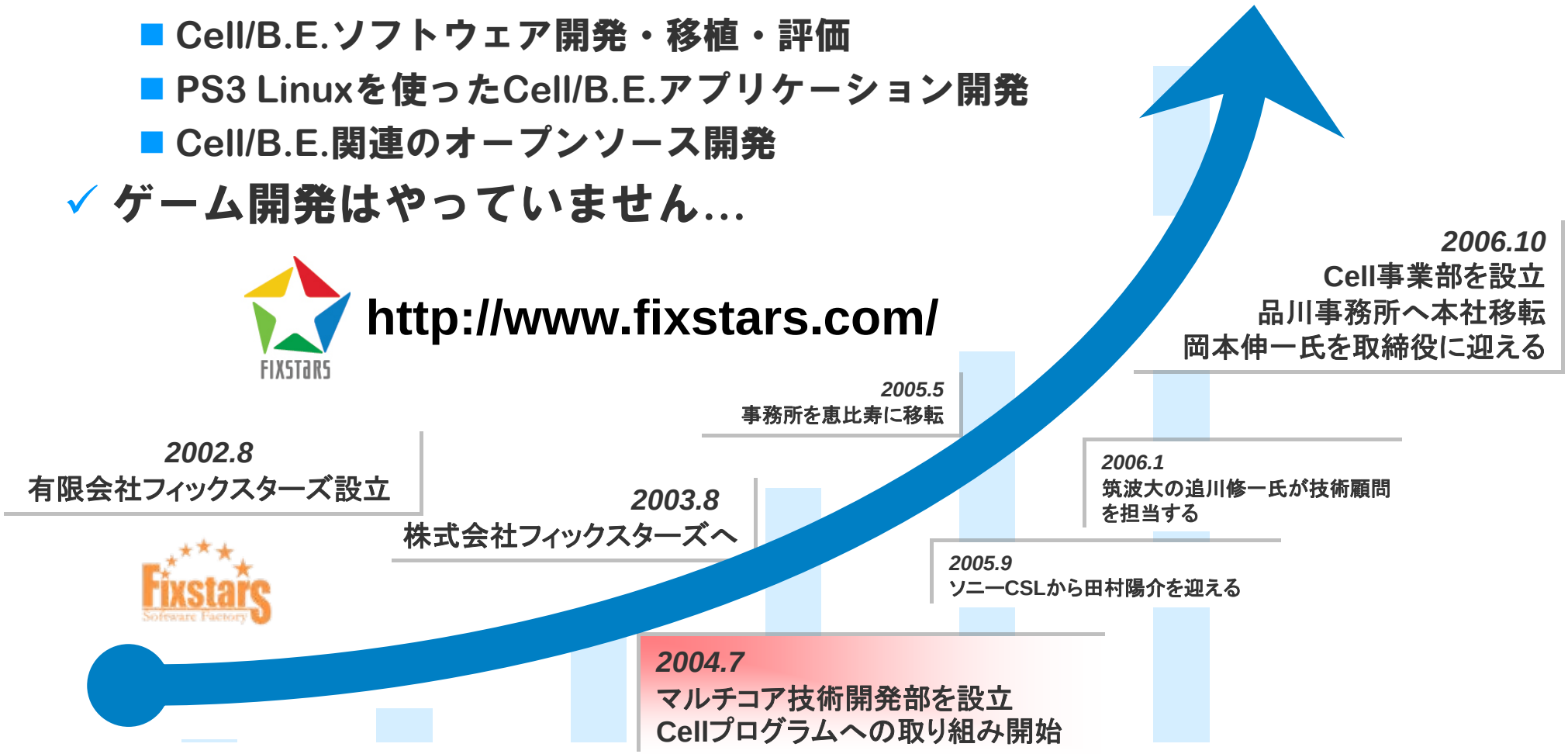
Who are We

➔株式会社フィックスターズ

- ✓ 2002年創業。2004年7月からCellへの取り組みをスタートし、Cell事業を中心に先進的なソフトウェア開発を積極的に行っています
  - Cell/B.E.ソフトウェア開発・移植・評価
  - PS3 Linuxを使ったCell/B.E.アプリケーション開発
  - Cell/B.E.関連のオープンソース開発
- ✓ ゲーム開発はやっていません…

FIXSTARS

http://www.fixstars.com/



For More Information

➔Webページもご覧ください

- ✓ (ゲームの話は載ってません)

内容の概要

- Cell/B.E.アーキテクチャとその上のプログラミング
- Cell/B.E.プログラミングの組立て
- Cell/B.E.プログラムを速くするには

CEDEC 2007

Cell/B.E.アーキテクチャと その上のプログラミング

Fixstars Corporation
Nisshin Building 3F, 1-8-27, kounan, Minato-ku, Tokyo 108-0075, Japan
Tel:+81-3-5781-5001 Fax:+81-3-5781-5002

Cell/B.E.アーキテクチャとは

- Sony, TOSHIBA, IBMの3社 (STI Alliance) によって開発された (比較的) 新しいマルチコアプロセッサ
  - ✓ PLAYSTATION®3 (PS3®) のメインプロセッサとしてもっとも有名

- 厳しい性能要求への1つの解
  - ✓ リッチコンテンツの増大、リアルタイム処理・インタラクティブ処理への要求、よりインテリジェントな・「人間らしい」コンピューティングへの要求

- “*Supercomputer-on-a-Chip*”
  「日常生活にスパコンレベルの処理能力を」
  - ✓ デジタルホームから高性能サーバまでスケールするアーキテクチャを

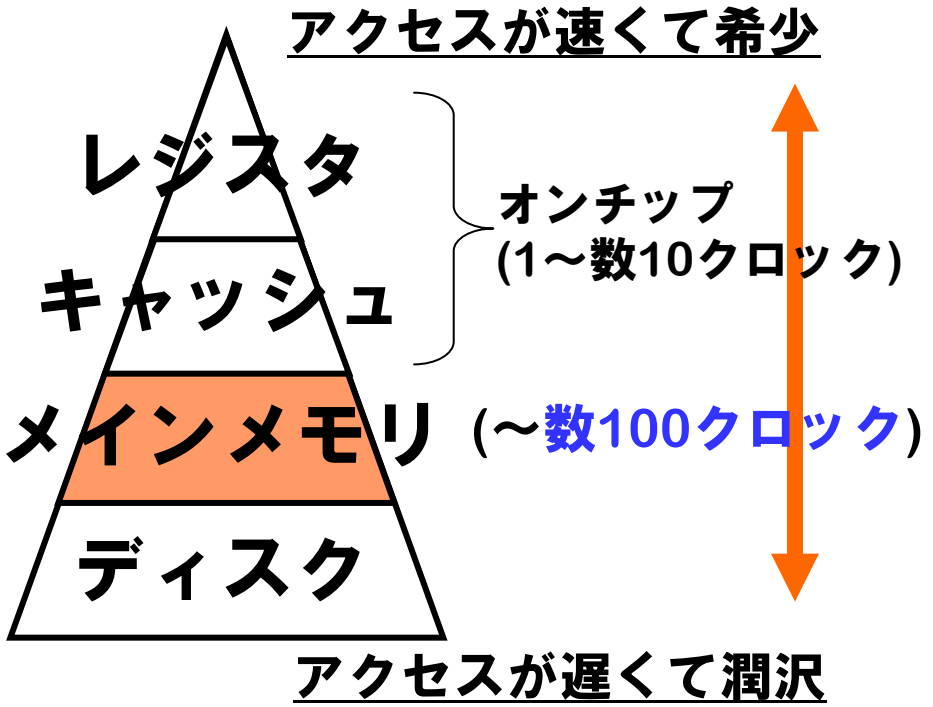
プロセッサ・アーキテクチャにおける性能問題

➔メモリの壁

- ✓ メインメモリを読み書きする速度が性能ボトルネックになってしまう
- ✓ キャッシュを使いこなすのも難しい
- ✓ SMPだとなお事態悪化?

➔周波数の壁

- ✓ プロセッサ動作周波数限界説
- ✓ 消費電力と発熱との戦い
- ✓ シングルコアではもう性能をあげられない?



➔消費電力の壁

- ✓ 周波数の3乗の割合に比例 (単純化したモデルにおいて)
- ✓ リーク電流も増えてきて大変...

Cell/B.E.アーキテクチャの解

➔マルチコア (複数コア)

- ✓ シングルコアで速くするのはなかなかつらい…
- ✓ マルチコア化で消費電力を下げながら性能をかせごう！
- ✓ スパコンレベルの性能が欲しい… → 9個くらい?

(＊ 当時にしてはかなり多い)

➔非対称

- ✓ メインメモリがボトルネック…各コアで分散して計算させよう！
- ✓ メインメモリとオンチップメモリの間の転送は
  ソフトウェアの好きなときに非同期で一括転送できるように…
  → 自前でDMAプログラムしてね

➔1 + N

- ✓ 汎用コアを増やしてもマルチスレッド化しないと恩恵はない
- ✓ でも莫大なデータの高速演算への要求は高まるばかり…
  → 1個の汎用コア + N個のデータ演算用コアで補助
- ✓ これからは専用プロセッサじゃなくてソフトウェアの時代だよね
  → ぜんぶ汎用・ただし得意分野をしぼる

Cell/B.E.のマルチコア・アーキテクチャ

➔「ヘテロジニアス」

- ✓ 対称的ではない、「異なる性質をもつ違う種類の」コアが複数搭載されたプロセッサ

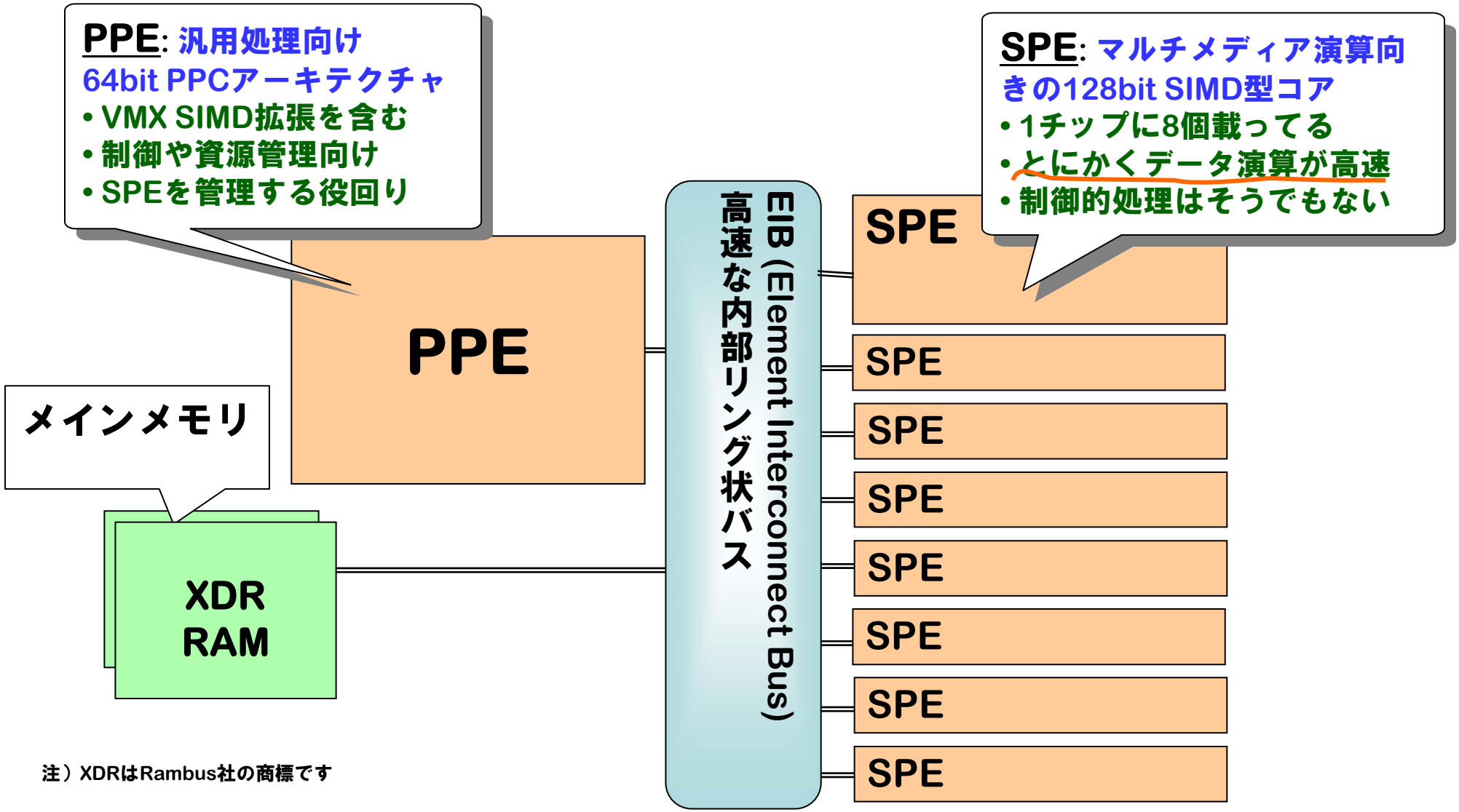
Intel系のプロセッサなどでいうマルチコア・プロセッサではホモジニアスな対称マルチコアが主流

➔Cell/B.E.の構成：1 + N

- ✓ 1個の制御用プロセッサ：**PPE** (PowerPC Processor Element)
  - ■ OSなどを動作させる資源管理・制御用の汎用プロセッサ
- ✓ N個の演算用プロセッサ：**SPE** (Synergistic Processor Element)
  - ■ メディア処理に最適化された汎用プロセッサ
  - ■ シンプルな128ビットSIMD演算器 (ベクタプロセッサ)
  - ■ 現モデルでは N = 8
    (PS3でプログラムから使えるのは６個)

SIMDとは同時に複数のデータに対して演算を行うこと

Cell/B.E.のハードウェア基本構成

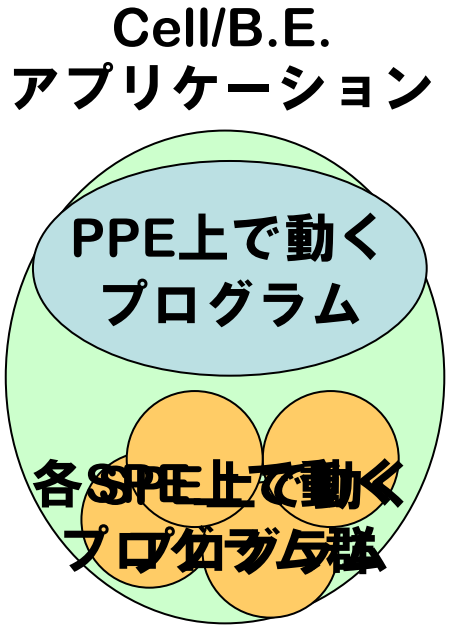


注）XDRはRambus社の商標です

Cell/B.E.上のプログラムの基礎の基礎

- PPEの上ではPPE向けにコンパイルされたプログラムが動く
- SPEの上ではSPE向けにコンパイルされたプログラムが動く
- たくさん(1＋8個の)プロセッサコアがあり、各コアの上で1つずつ違うプログラムが動く
- Cell/B.E.アプリケーションとは…

  複数のプログラムが協調して1つのアプリケーションを構成



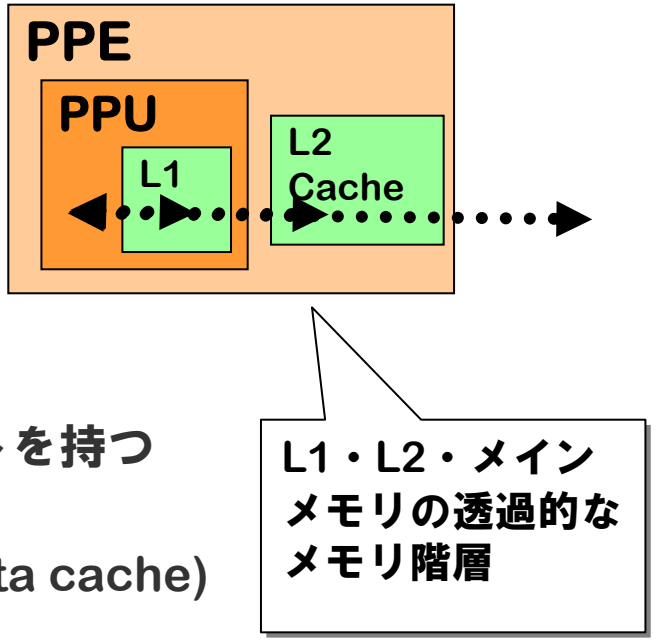
PPE: PowerPC Processor Element

汎用用途の64bitプロセッサエレメント

- ✓ PowerPC Architecture Version 2.02準拠
- ✓ オペレーティングシステム(OS)などのシステムソフトウェアはこの上で動作
- ✓ SPEやSPEプログラムを制御する役割を負う
- ✓ ネットワークやディスク資源へのアクセスも基本的にPPEが行う

PPU: PowerPC Processor Unit

- ✓ PPEの中のプロセッサユニットのことをPPUと呼ぶ
- ✓ PowerPC 970 (G5)などと比べるとややシンプル
  - 2ウェイマルチスレッディング
  - インオーダー型
- ✓ 基本命令セットのほかにマルチメディア拡張命令セットを持つ
  - VMX: Vector/Multimedia SIMD Extension
- ✓ 各 32KB L1 キャッシュ (32KB inst. cache、32KB data cache)
- ✓ 512KB L2 キャッシュ (I+D)
  - ソフトウェアで制御可能

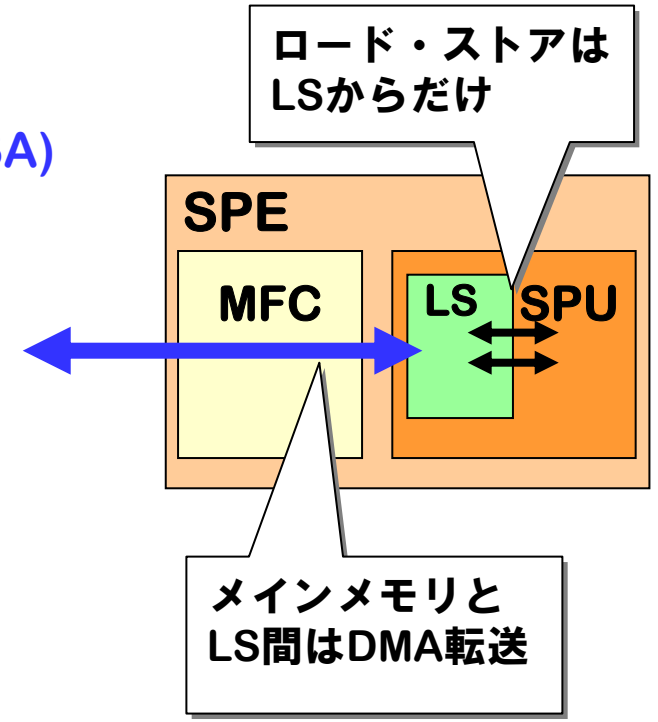


SPE: Synergistic Processor Element

- マルチメディア演算処理向きの(でも汎用な)ベクタプロセッサ
  - ✓ SPE用にコンパイルされたバイナリが動作する
  - ✓ PPE上で動作するOSやユーザプログラムによって制御される

SPU: Synergistic Processor Unit

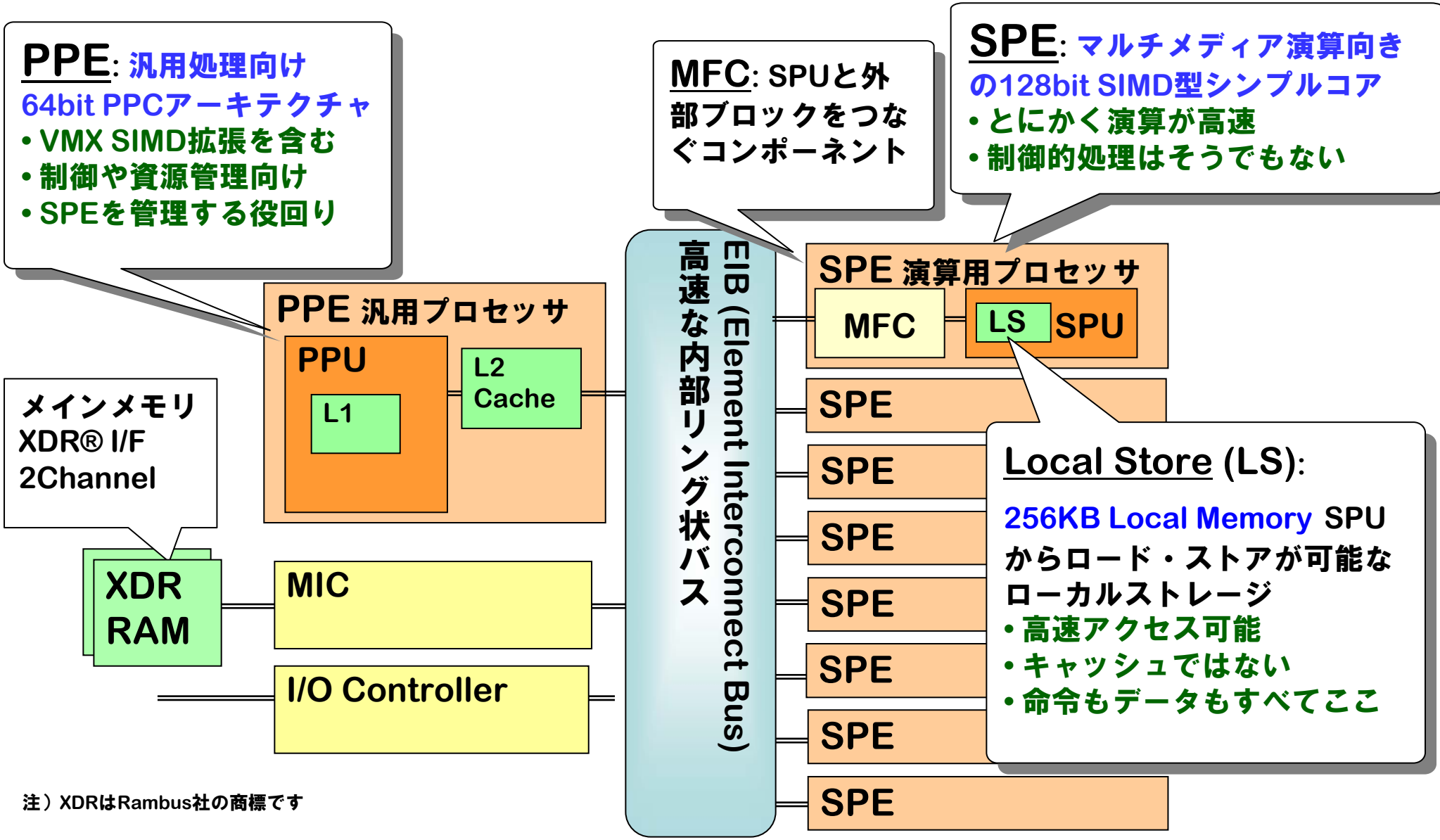
- ✓ 新しい汎用ベクタプロセッサアーキテクチャ (SPU ISA)
  - シンプルな128ビットSIMD RISC
  - インオーダー型のパイプライン
  - 潤沢な(128本の)128ビット長の汎用レジスタ
- ✓ オンチップのローカルストア (LS) を持つ
  - SPUプログラムはLSからのみロード・ストア可能
  - 命令・データあわせて256KB
  - キャッシュではない・メインメモリなどのメインストレージにはDMA転送でアクセス



MFC: Memory Flow Controller

- ✓ SPUと外部のブロック(PPEや他のSPE, I/Oデバイスなど)の間の通信を制御
- ✓ DMAコントローラ(DMAC)によりLSとメインストレージ間のバルク転送が可能
- ✓ SPUとは独立して非同期に動作
- ✓ アドレス変換ユニットを持つ → 実効アドレスでメインストレージにアクセス可

Cell/B.E.のハードウェア基本構成 (詳細)



注）XDRはRambus社の商標です

PPUプログラムとSPUプログラムの基礎

➔PPUプログラム (PPEプログラム)

- ✓ 64bit PowerPC互換CPU上で動作する普通のプログラム
- ✓ メインメモリ上のデータを(キャッシュ越しに)読み書きしながら動作
- ✓ OSやライブラリの機能を使ってSPE (SPU)やその他ハードウェア資源を制御・管理する

➔SPUプログラム (SPEプログラム)

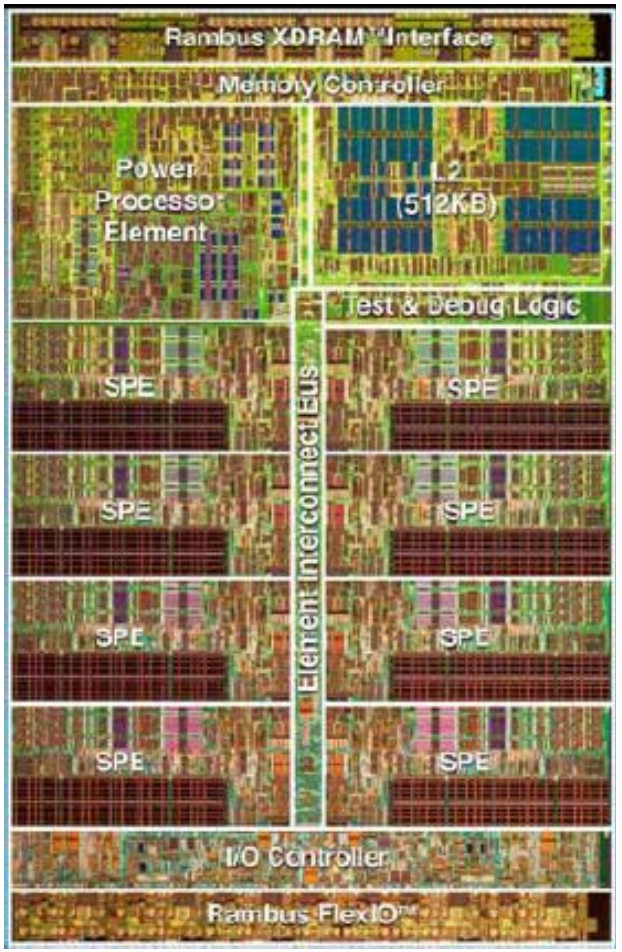
- ✓ 128bit SIMD型の新しいコアSPU上で動作する小さめのプログラム
- ✓ データ演算に特化されているが汎用アーキテクチャ (**各基本型をサポート**)
- ✓ LS (ローカルストア)上のデータを読み書きしながら動作
- ✓ メインメモリやその他のSPEのLS上のデータにアクセスするにはDMA転送が必要

原則的に、別々に書き、別々のコンパイラでコンパイル

どちらも標準的なC/C++でプログラミングが可能

参考: 90nm Cell B.E.プロセッサ (3.2GHz) の性能概要

- ➔2億4100万トランジスタ
- ➔9コア (1 PPE + 8 SPE)
- ➔ピーク性能
  - ✓ 単精度 ～200 GFlops
    - 4 (32-bit SIMD並列) * 2 (FMA) Flop * 8 SPE * 3.2GHz = 204.8 GFlops per socket
  - ✓ 倍精度 ～20 GFlops
  - ✓ メモリ帯域 25.6 GB/s
  - ✓ I/O 帯域 35 GB/s (out) + 25 GB/s (in)
  - ✓ EIBバス帯域 ～200 GB/s
    - 16byteの4本のリングバス
    - 12本まで同時転送可能
    - ピーク時 1cycleあたり96byte



CEDEC 2007

Cell/B.E.プログラムの組立ての基礎

トピック: SPEの制御とDMA転送の基礎

Fixstars Corporation
Nisshin Building 3F, 1-8-27, kounan, Minato-ku, Tokyo 108-0075, Japan
Tel:+81-3-5781-5001 Fax:+81-3-5781-5002

Cell/B.E.上のマルチコア・プログラミング

➔マルチコア・プロセッサ上のプログラミングの原則

- ✓ アプリケーション内で並列に動かせるところを
  うまく複数のコア上で同時に動かすことで高い性能を出す
- ✓ 並列化は、自分でがんばってやる…

➔Cell/B.E.上のプログラムの場合、さらに…

- ✓ 各プロセッサコアの異なる性質を活かしてうまく役割分担
- ✓ PPEは資源を調停したり管理をしたりし、SPEには演算をさせる
- ✓ 性能のためには、できる限りSPEにたくさんやらせる
  - ■ SPEの方が演算はとにかく高速
- ✓ データアクセスが支配的な場合はDMA転送設計がすべてを決める
  - ■ プリフェッチ可能か(処理前に転送したいアドレスが計算できるか)が大きな課題
- ✓ LSの256KBの壁も全体のプログラム構成に影響
  - ■ 入りきらない場合は必要に応じてプログラムやデータを入れ替える

Cell/B.E.上のプログラム組立ての基礎

➔演算が重いところをSPEにやらせる (オフロード)

PPEプログラム

```
int main() {
  …
  いろいろ前処理

  SPEの準備
  大変な計算！SPEにお願い
  (自分じゃやらない)

  やらせてる間に何かやっとく

  終わったかな?
  SPEの破棄

  いろいろ後処理
}
```

問題を切り出してSPEプログラムにオフロードする(分担してもらう)

SPEプログラム

```
int main() {
  計算データをDMA転送で取得

  がんばって計算

  計算結果をDMA転送して返す
}
```

- 最も基本となる組み立て
- SPEは1個でも(チューンすると) 速い

Cell/B.E.プログラムにおける最も基本的なポイント

➔基本1: PPEからSPEプログラムを制御する

- ✓ OSやライブラリが提供しているAPIを使ってプログラミング

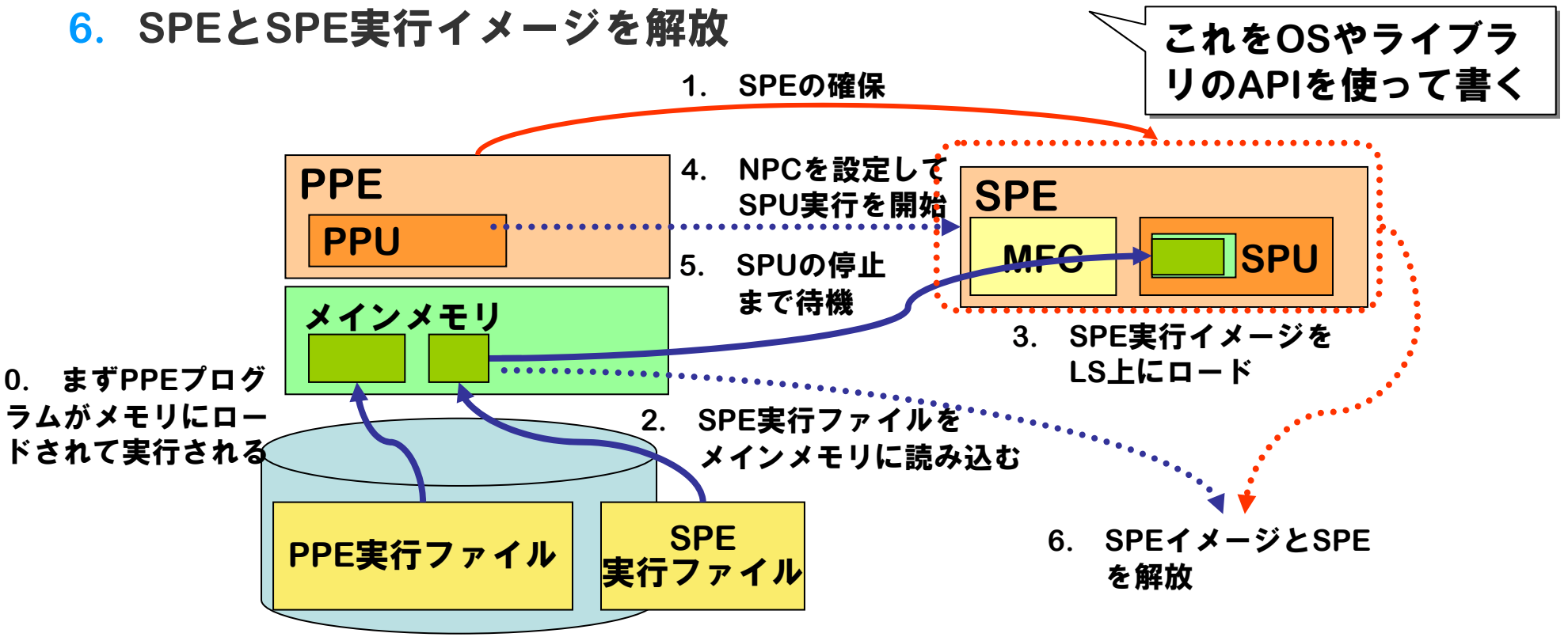
➔基本2: 計算データなどをSPEに渡す(結果をもらう)

- ✓ 各SPEに載っているMFC の機能を使って**DMA転送**する

基本1: PPEからSPEプログラムを制御する

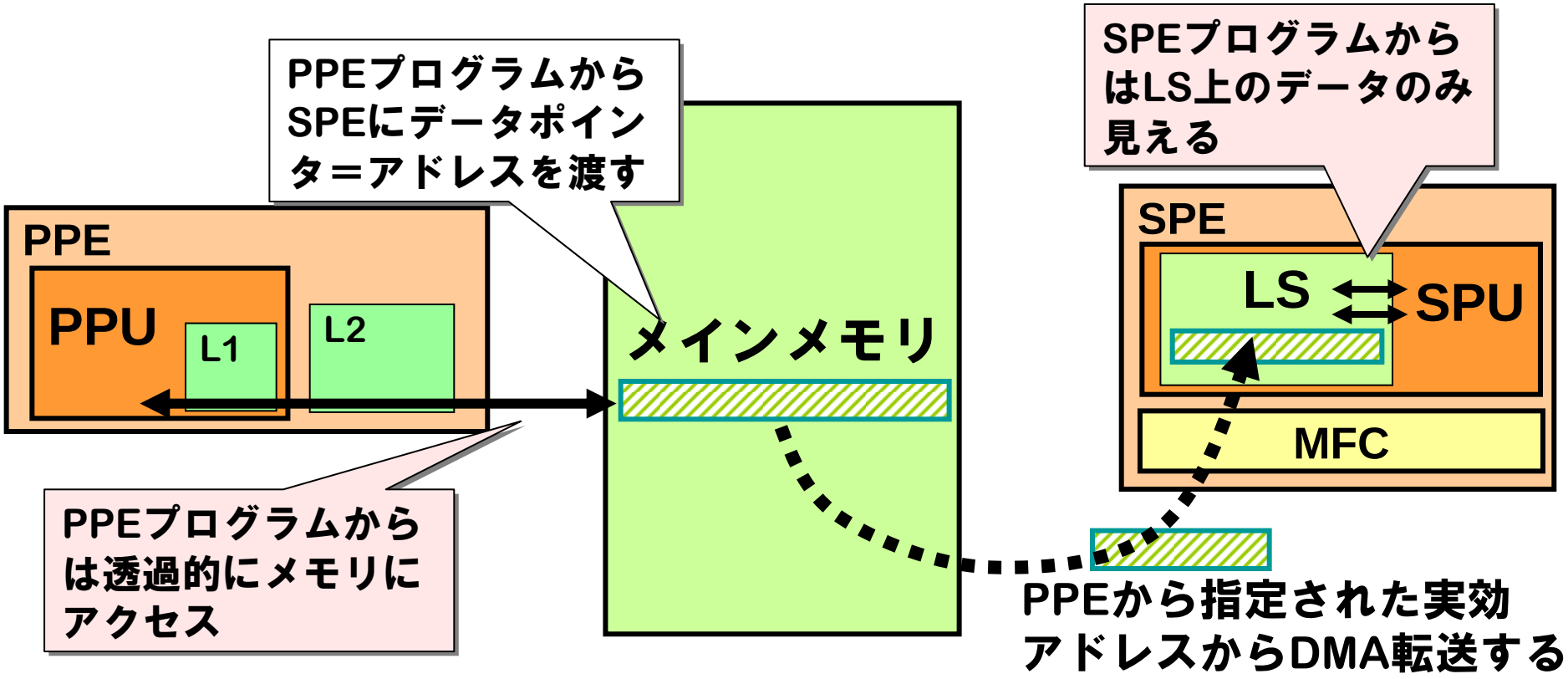
➔ PPEからSPEプログラムを実行するまでの基本シーケンス：

1. PPEプログラムのためにSPEを確保
2. SPEプログラムの実行ファイルをメモリに読み込む
3. SPEプログラムの実行イメージをターゲットのSPEのLS上に読み込む
4. SPUのNPCや実行パラメータを設定してSPEの処理を開始
5. SPUの状態を見て開始したプログラムの実行が終了するのを待つ
6. SPEとSPE実行イメージを解放



基本2: DMA転送でSPEに計算データを渡す

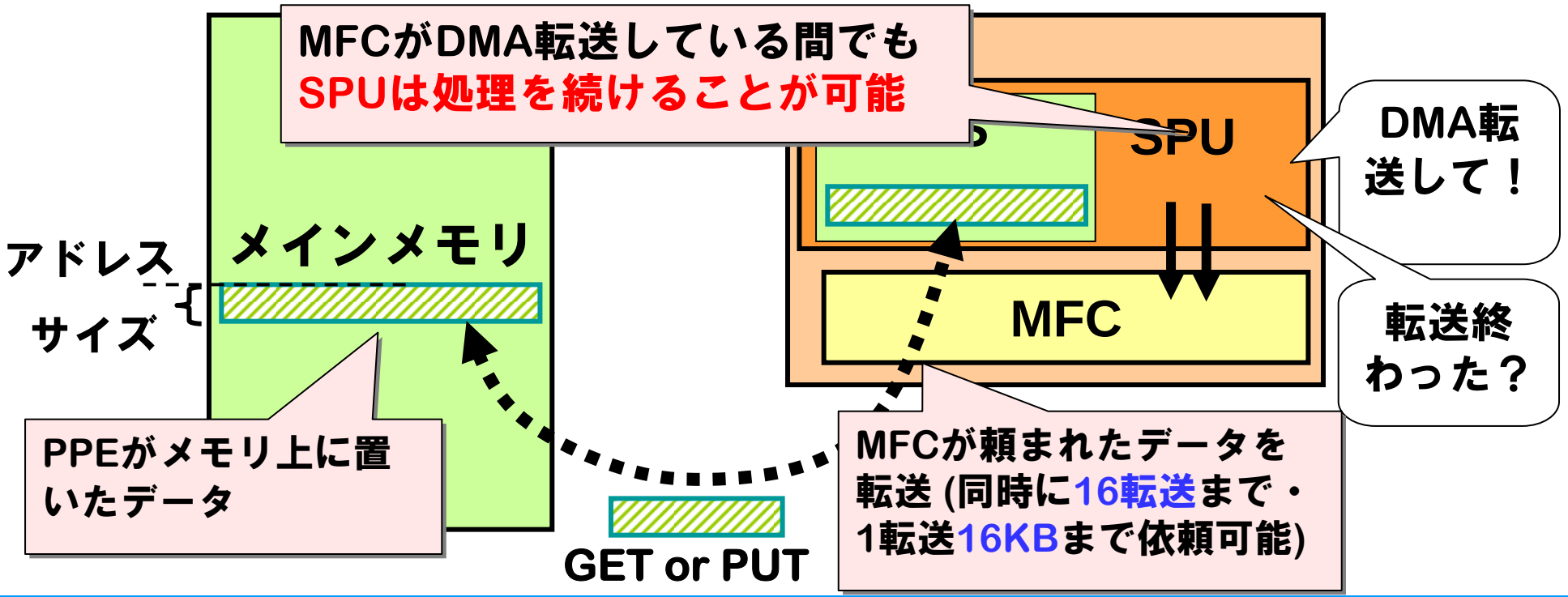
- SPEにデータを渡す典型的な方法は...
  - ✓ PPEでメインメモリ上にデータを用意
  - ✓ SPEの実行を開始するときにデータの開始**実効アドレス**(＝ポインタ値)をSPEにパラメータとして渡す
  - ✓ SPEプログラムからそのデータをDMA転送してLS上に取得



MFCによるDMA転送のイメージ

SPEからDMA転送を発行する基本的な手順は…

1. 「MFCにお願いする」
   - SPUからMFCにDMA転送をリクエストする
   - 転送したいメインメモリ領域のアドレスとサイズを指定する
2. 「前にお願いした転送が終わるまで待つ」
   - DMA転送の完了を待つためにMFCに状態の更新をリクエストする
   - 複数のDMA転送をお願いしておいて、そのうちの特定の転送の完了を待てる

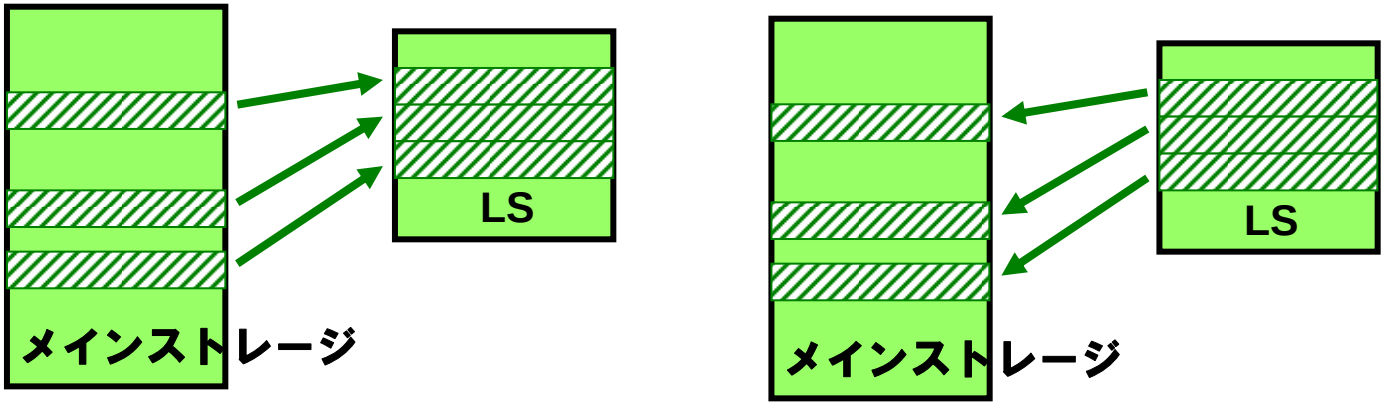


MFCによるDMA転送：DMA転送の例 (Pseudo Code)

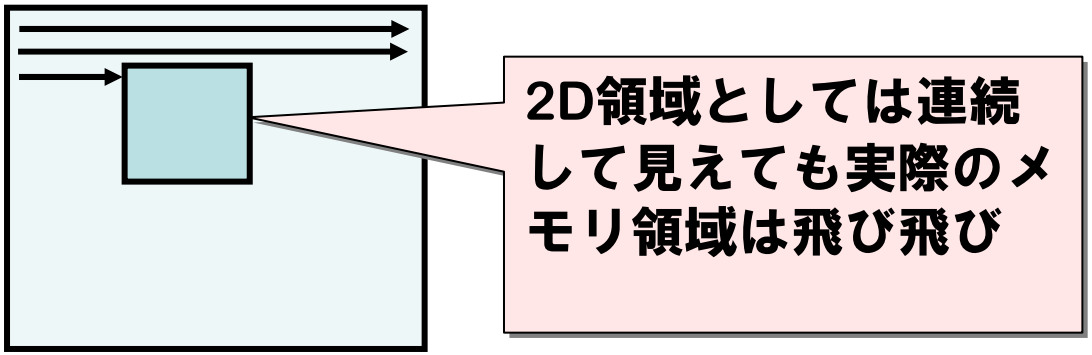
```
// メインメモリからのデータ取得を2つ依頼
DMA_GET(LSアドレス1, メインメモリアドレス1, サイズ, タグID0);
DMA_GET(LSアドレス2, メインメモリアドレス2, サイズ, タグID1);

// 何か別の処理...

DMA_WAIT(タグID0);                  // タグID0の転送終了を待つ
process_data(LSアドレス1);          // 転送したデータを処理
DMA_PUT(LSアドレス1, メインメモリアドレス1, サイズ, タグID2);

DMA_WAIT(タグID1);                  // タグID1の転送終了を待つ
process_data(LSアドレス2);          // 転送したデータを処理
DMA_PUT(LSアドレス2, メインメモリアドレス2, サイズ, タグID2);

// 何か別の処理...

// データをメインメモリに転送し終えるのを待つ
DMA_WAIT(タグID2);
```

より進んだDMA転送: DMAリストによる転送

- MFCはメインストレージ上のバラバラな領域とLS間でのデータ転送を1リクエストで実行するDMAリスト転送をサポートする
  - ✓ メインストレージ上の複数の非連続領域を一度に転送
  - ✓ 16KBまでのサイズの領域を合計2048個まで一度にリクエスト可能



- 1次元配列として確保された2Dや3Dの領域の一部を転送するときなどによく使われる



CEDEC 2007

複数SPEを使った
並列Cell/B.E.プログラムの組立ての基礎

トピック: 並列化アプローチ・コア間の通信と協調

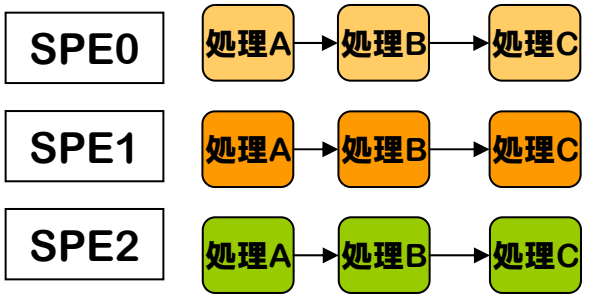
Fixstars Corporation
Nisshin Building 3F, 1-8-27, kounan, Minato-ku, Tokyo 108-0075, Japan
Tel:+81-3-5781-5001 Fax:+81-3-5781-5002

複数SPEを使った並列Cell/B.E.プログラムへの拡張

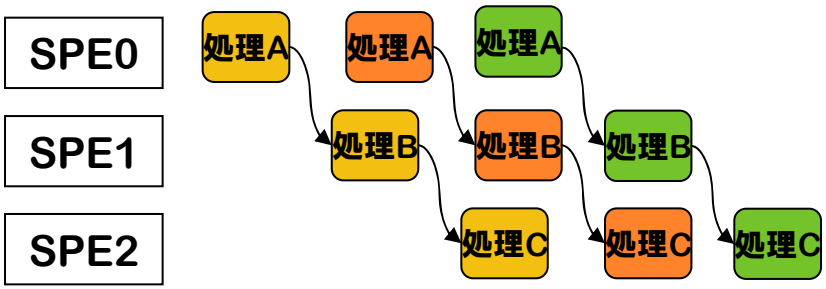
➔プログラムをどうにか並列化して複数SPEへ...

➔並列化アプローチの例

- ✓異なる特性を活かして機能分担する
  - 制御的・管理的役割... PPE
  - マルチメディア演算・浮動小数演算... SPE
  - 空いているコアをとにかく活用するのも重要
- ✓一般的な並列化モデルの適用
  - データ並列
  - パイプライン並列



データ並列による並列化



パイプライン並列による並列化

Cell/B.E.上の並列プログラム組立ての基礎 (1)

➔入力データを分割して複数SPEにやらせる (データ並列)

PPEプログラム

```
int main() {
  …
  入力データの分割など
  for (i = 0; i < NSPE; i++) {
    複数SPEスレッドの準備
    計算の一部をお願い
  }

  やらせてる間に何かやっとく

  for (i = 0; i < NSPE; i++) {
    終わったかな?
    複数SPEの破棄
  }
  結果の収集など
}
```

入力データを分割して複数SPEに
並行してオフロードする

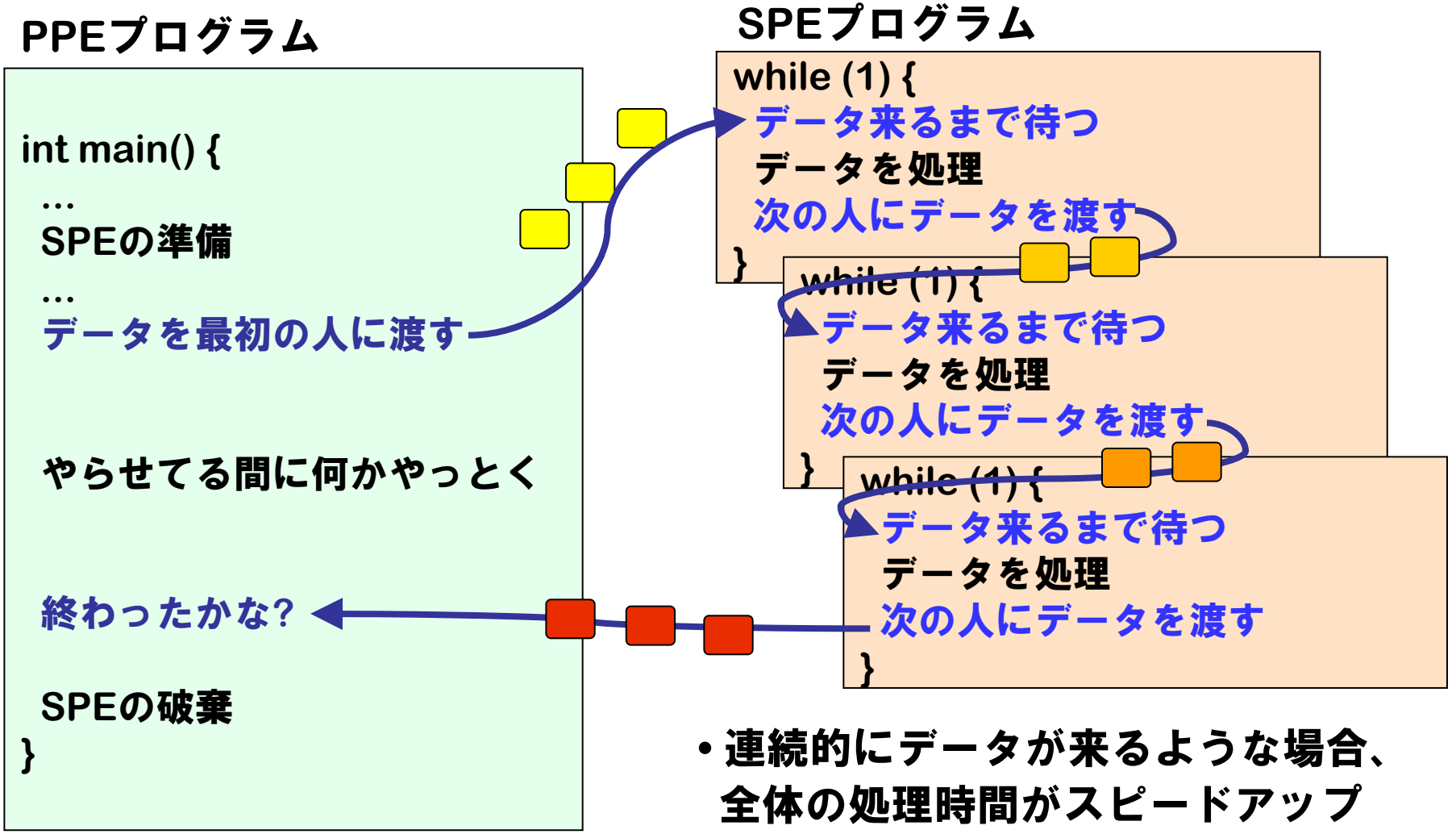
SPEプログラム

```
int main() {
  計算データをDMA転送で取得
  がんばって計算
  計算結果をDMA転送して返す
}
```

- 並列化の最も基本的な組み立て
- 処理があるデータ単位で分割できてかつ
  各データの処理に依存がなければ最適性能
- データ転送はメインメモリとLS間が主

Cell/B.E.上の並列プログラム組立ての基礎 (2)

➔複数SPEに流れ処理させる (パイプライン並列)



- 連続的にデータが来るような場合、全体の処理時間がスピードアップ
- データ転送は各SPEのLS間が主

並列Cell/B.E.プログラムにおける基本的なポイント

➔基本3: PPEを介した複数SPEの協調

- ✓ MFCによるメッセージング機構を使ったPPE-SPE間のメッセージングと複数SPEの協調

➔基本4: SPE-SPE間のデータ転送と協調

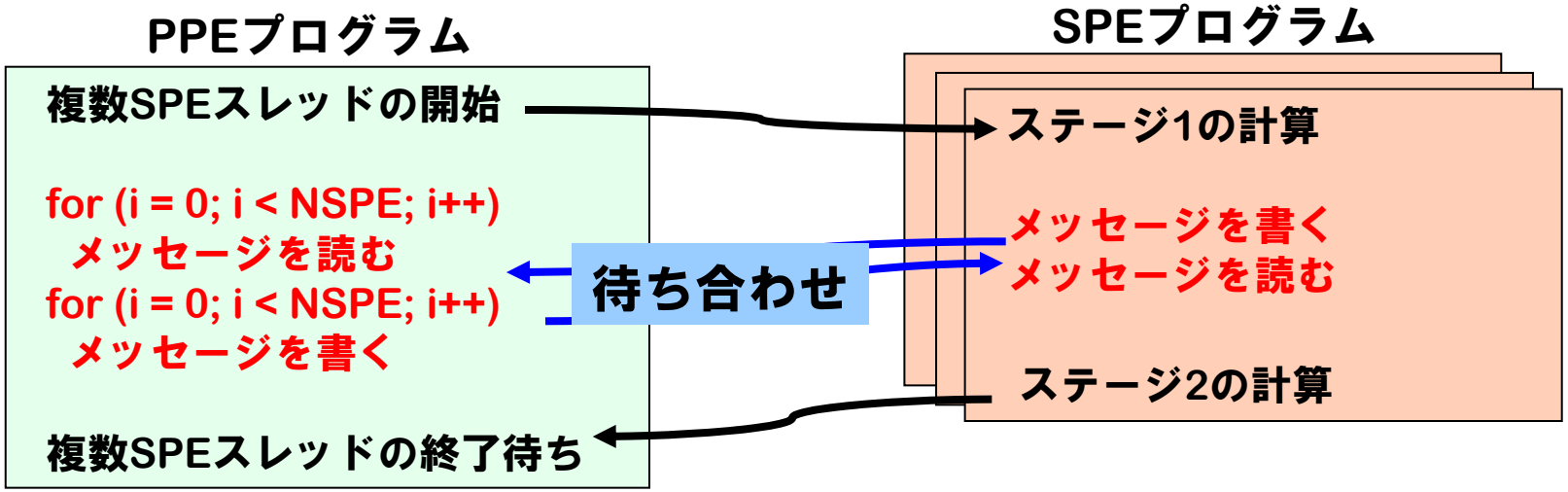
- ✓ MFCによるSPE-SPE間のDMA転送とメッセージング

基本3: PPEを介した複数SPEの協調

➔MFCによるコア間のメッセージング機構

- ✓ **メールボックス**...SPEへ、あるいはSPEからの小さい(32bitの)メッセージを送受するための機構。メッセージキュー的に使える
- ✓ **シグナル通知レジスタ**...SPEに小さい(32bitの)値を通知するためのレジスタ。各SPE(MFC)に2つずつあり、ORモードでの値の書き込みが可能

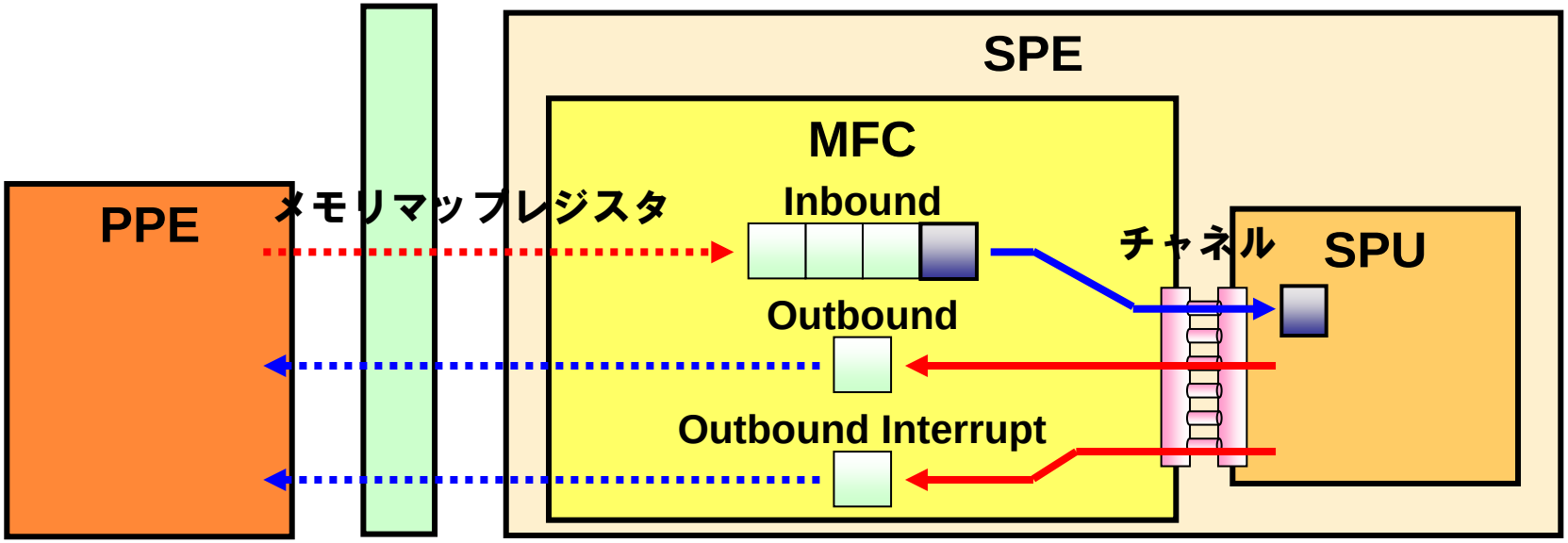
➔ DMA転送があるコアからの「ストレージ間のデータ転送」機構を提供するのに対し、これらの機構は「**片方が書く**」「**片方が読む**」という対照的なコア間通信APIを提供する



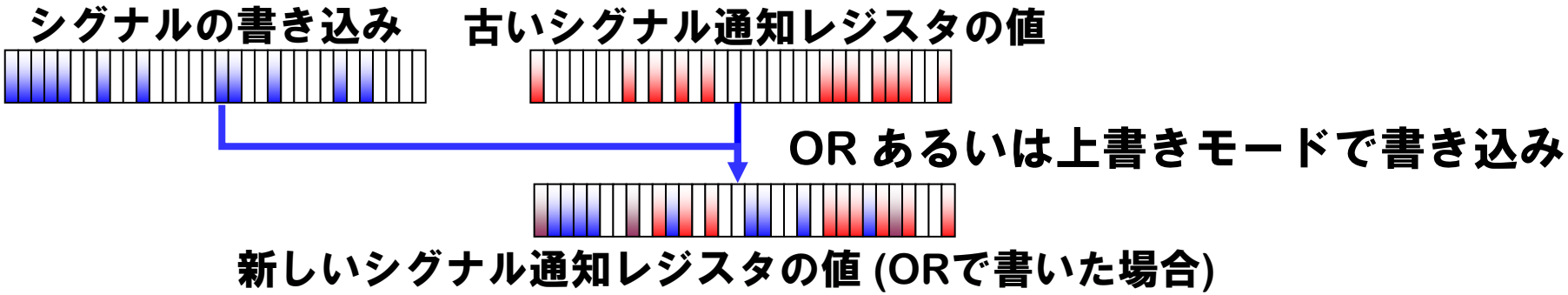
PPE-SPE間のメッセージングを使った簡単なバリア同期の例

メールボックスとシグナル通知レジスタの違い

メールボックスは：基本的にメッセージキュー



シグナル通知レジスタは：基本的にビットフィールド



Cell/B.E.上のより一般的なプログラム組立て

➔メッセージを使ってSPEに必要な処理を依頼

- ✓1つのSPEプログラムに同時に良く使われる複数の機能を入れる

PPEプログラム

SPEの準備
while (メインループ) {
 そのとき必要な処理を
  適当な数のSPEに
  お願いする

 自分もいろいろやる…

 終わったSPEがいたら
  必要ならまた別の処理を
  お願いする

 状態の更新など…
}
SPEの破棄

SPEの初期化と破棄はなるべく最初と最後だけにして、メッセージングを中心に状態遷移していく

SPEプログラム

while (1) {
 頼まれるまで待つ
 switch (頼まれた仕事は) {
  case 物理計算: …; break;
  case 3D計算: … ; break;
  case 夏休みの宿題: …; break;
 }
 終わったことを通知
}

Cell/B.E.プログラムにおけるSPEの使い方の原則

➔一般的に、OSが提供するSPE実行体の初期化やプログラムのロードはコストが高い

- ✓ SPEスレッドやSPEコンテキストなど
- ✓ OSへの依頼コストの他に、SPE資源とのバインドやSPE状態のクリアなど、資源確保や保護のための様々な初期化コストがかかる

➔SPEの初期化は最初にやっておき、メイン処理部ではメッセージなどだけで遷移させるのが低コスト

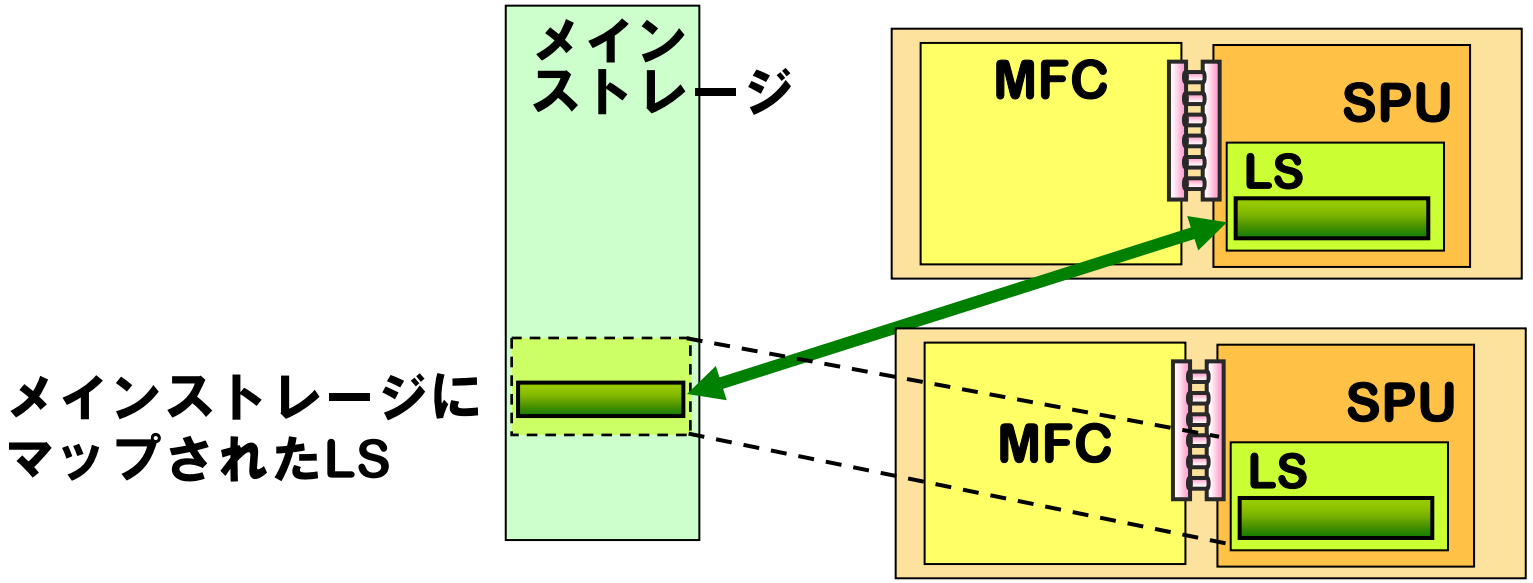
➔あるいはOSの提供するSPE実行体より低コストなアプリケーションレベルの実行体を使う方法も

- ✓ SPURSタスクなど (自前で作る方法ももちろんある)
- ✓ 資源保護などがないため、軽量だが使う側にも多少注意が必要

基本4: SPE-SPE間のデータ転送と協調

➔ メインストレージにマップされた他のSPEのレジスタやLSをMFCからアクセスすればSPE間通信が可能

- ✓ 例えば…転送先のSPEのLSがマップされてるアドレスと自分のLS間でDMA転送すれば2つのSPEのLS間でDMA転送できる
- ✓ シグナル通知やメールボックスも同様の方法でSPE間で直接(PPEを介さずに)メッセージングが可能



CEDEC 2007

SPEを中心とする Cell/B.E.プログラムの組立ての基礎

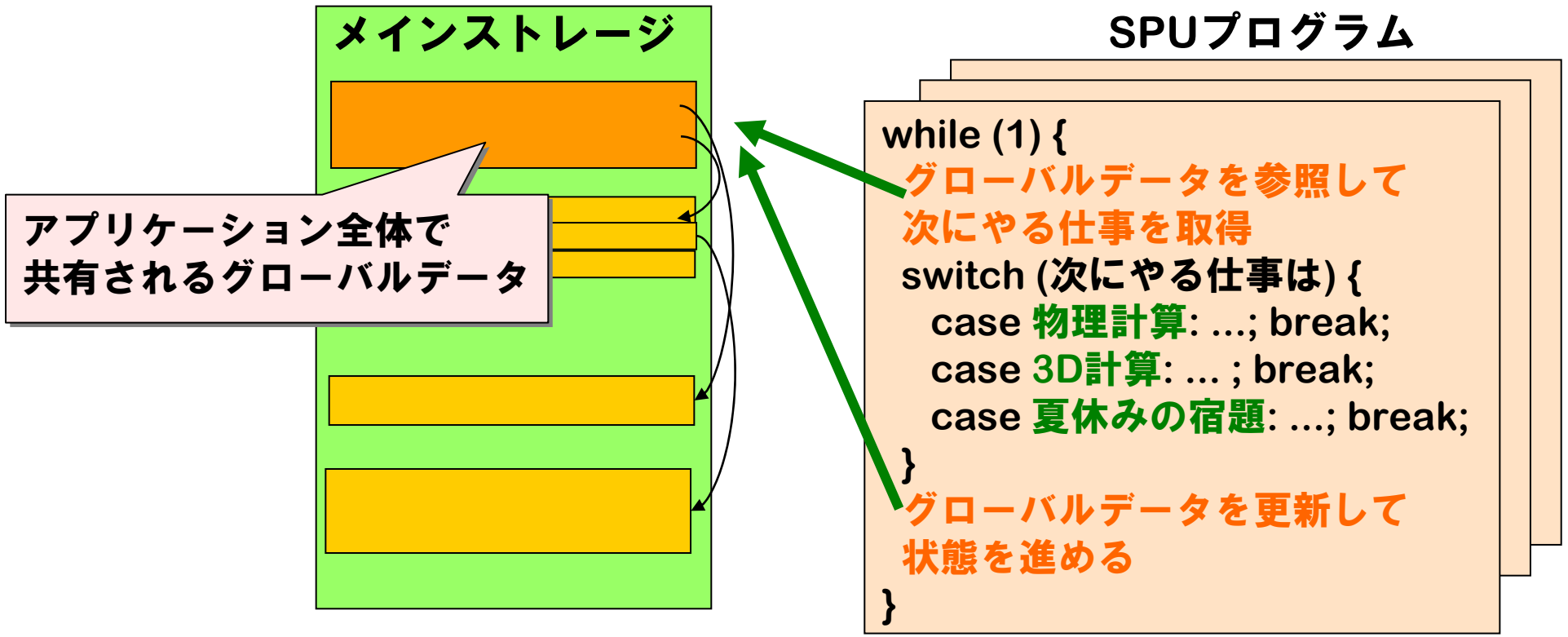
トピック: アトミック更新・オーバレイ

Fixstars Corporation
Nisshin Building 3F, 1-8-27, kounan, Minato-ku, Tokyo 108-0075, Japan
Tel:+81-3-5781-5001 Fax:+81-3-5781-5002

SPEを中心とするCell/B.E.プログラム組立ての基礎

➔複数SPEに自立的に処理させる (SPE-Centricモデル)

- ✓ 共通の状態やグローバルなデータを共有領域に保持しておき、複数SPEから更新しながら処理を進めていく
  - PPEの調停を減らしてSPEの自立性を高めることでSPEの演算性能を最大限活用
  - やる仕事が256KBのLSに載らなかったら自分でコードを入れ換える(オーバレイ)



SPEを中心とするCell/B.E.プログラムの基本ポイント

➔基本5: 共有データのアトミックな(安全な) 更新

- ✓ アトミックDMA転送によるアトミックなメモリ更新

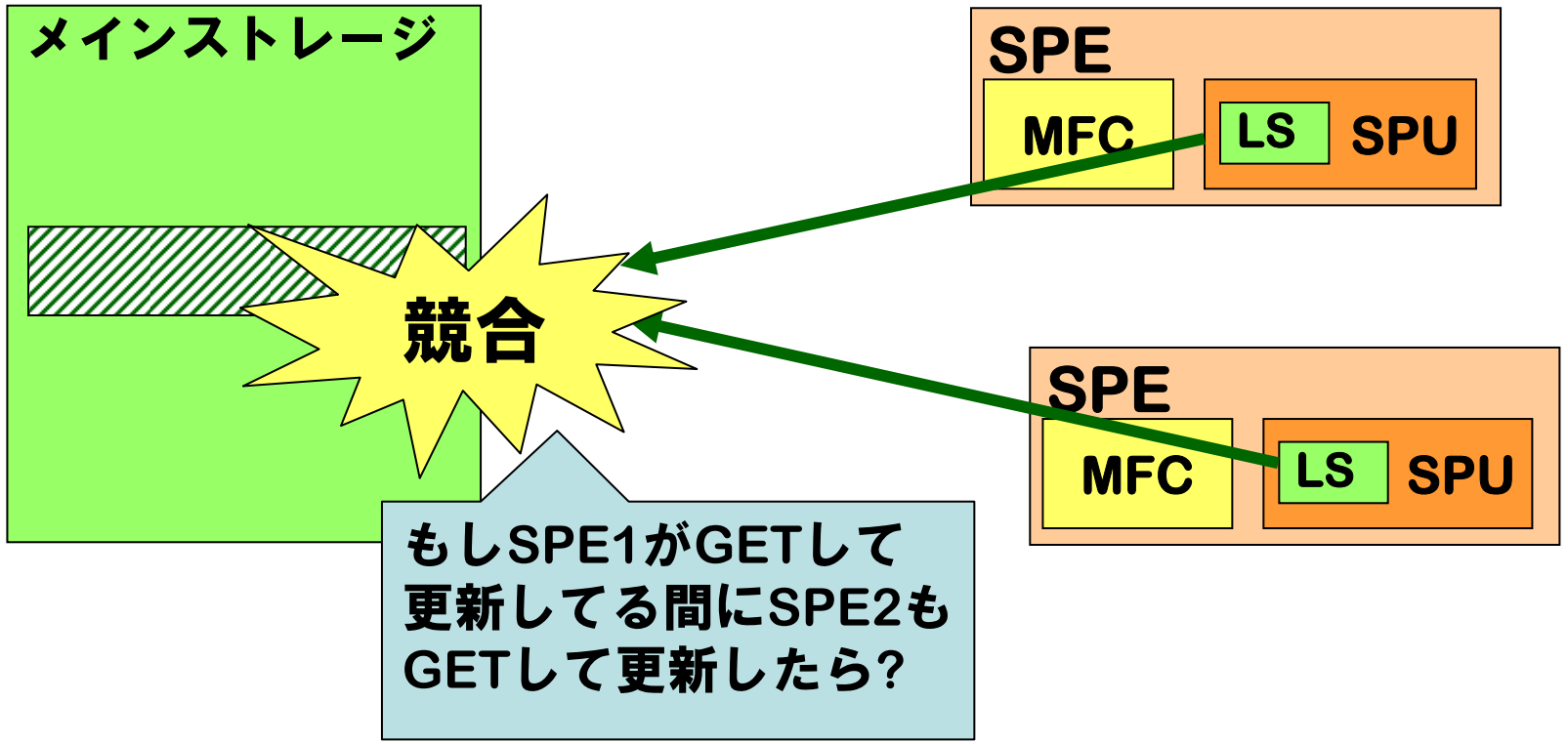
➔基本6: オーバレイによる自己コード書き換え

- ✓ DMA転送により自分のLS上のプログラムコードを書き換える

基本5: 共有データのアトミックな更新 (1/2)

➔複数のSPEから1つの共有メモリ領域を更新するには…
アトミック(不可分)に更新しないと危険

- ✓ 読むだけ・書くだけなら良いが、「読んで(GET)して更新して(modify)書き戻す(PUT)」ような処理を複数SPEが行うと…



基本5: 共有データのアトミックな更新 (2/2)

➔Cell/B.E.の仕組みを使って複数コアから共有データの値を**アトミックに更新**するには？

- ✓ Cell/B.E.では、リザベーション(予約)付きでロックラインを取得してロード/ストアあるいはDMA転送を行うことでメモリのアトミック更新を実現できる

  - リザベーション付きでロックラインを取得してメモリ内容を取得 (GET)
  - メモリ内容を使って新しい値を計算 (modify)
  - リザベーション条件つきでメモリに新しい内容をストア (PUT)

- ✓ Mutexロック(排他ロック)などの同期プリミティブも基本的にはこの機構で実現される
  - Mutexロックは通常アトミックに0か1に更新されるメインメモリ上の領域
  - アトミック更新の仕組みが分かっていれば、Mutexロックを使わなくてもそこを直接アトミックに更新することが可能 (ただし領域が128バイト以内の場合)

リザベーションロックライン機構の詳細

➔Cell/B.E.のリザベーションロックライン機構とは...

- ✓ あるメモリ番地に鍵をかけて誰もさわれなくする機構ではない
- ✓ その代わり、リザベーションつきでロックラインを取得してメモリ内容をロードすると、そのメモリ内容が他からアクセスされないか見張ってくれる
- ✓ 誰かが同じロックラインを持つアドレスからロードするとリザベーションは失われる
- ✓ リザベーション条件付きでメモリ内容をストアしようとすると、リザベーションが失われていた場合にはストアが失敗する
- ✓ ストアが成功するまでリザベーション付きのロード・ストアを繰り返すことでアトミックなメモリ更新が達成できる

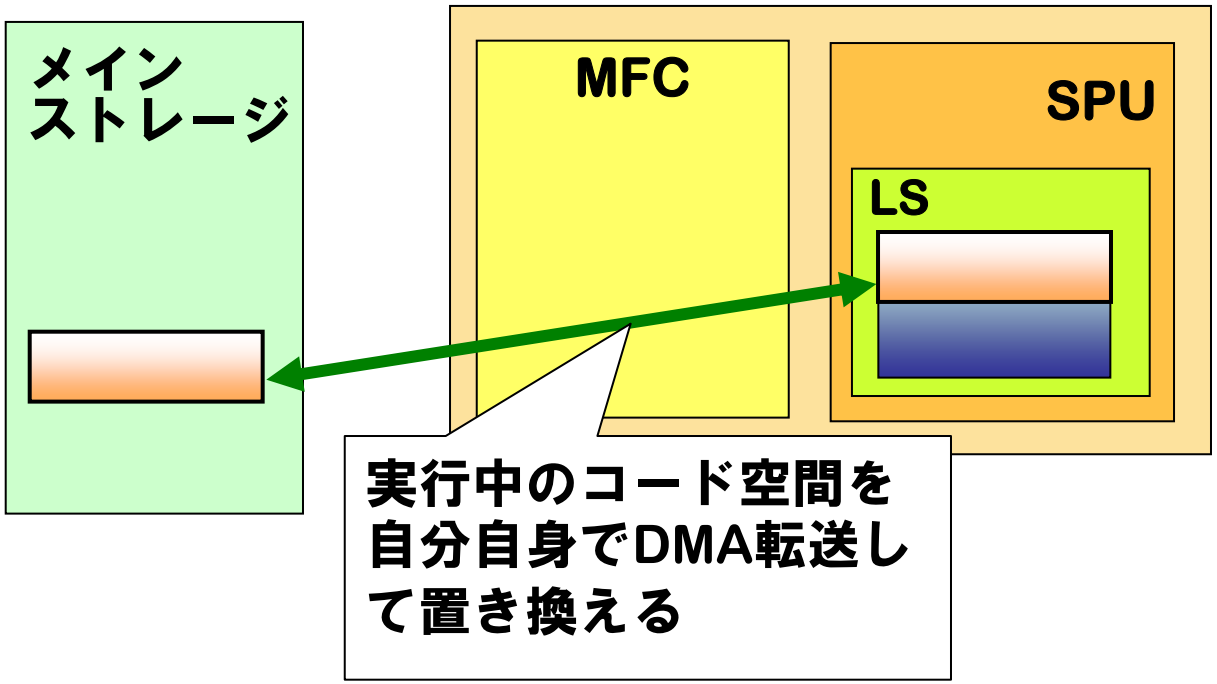
➔リザベーションロックラインに関する補足

- ✓ PPU/SPUどちらからも使えるので、汎用的なコア間同期に使うことができる
- ✓ PPUからのリザベーションつきロード/ストアが32/64ビットに限られるのに対し、SPUからはDMA転送によって128バイトのライン全体をアトミック更新できる
- ✓ 条件付のストアが失敗した後に次のロードをretryする際に、他コアがリザベーションを失うMFCイベントを監視することで効率的に待つことが可能

基本6:オーバレイによる自己コード書き換え

➔SPEのLS上の空間は基本的にフラット

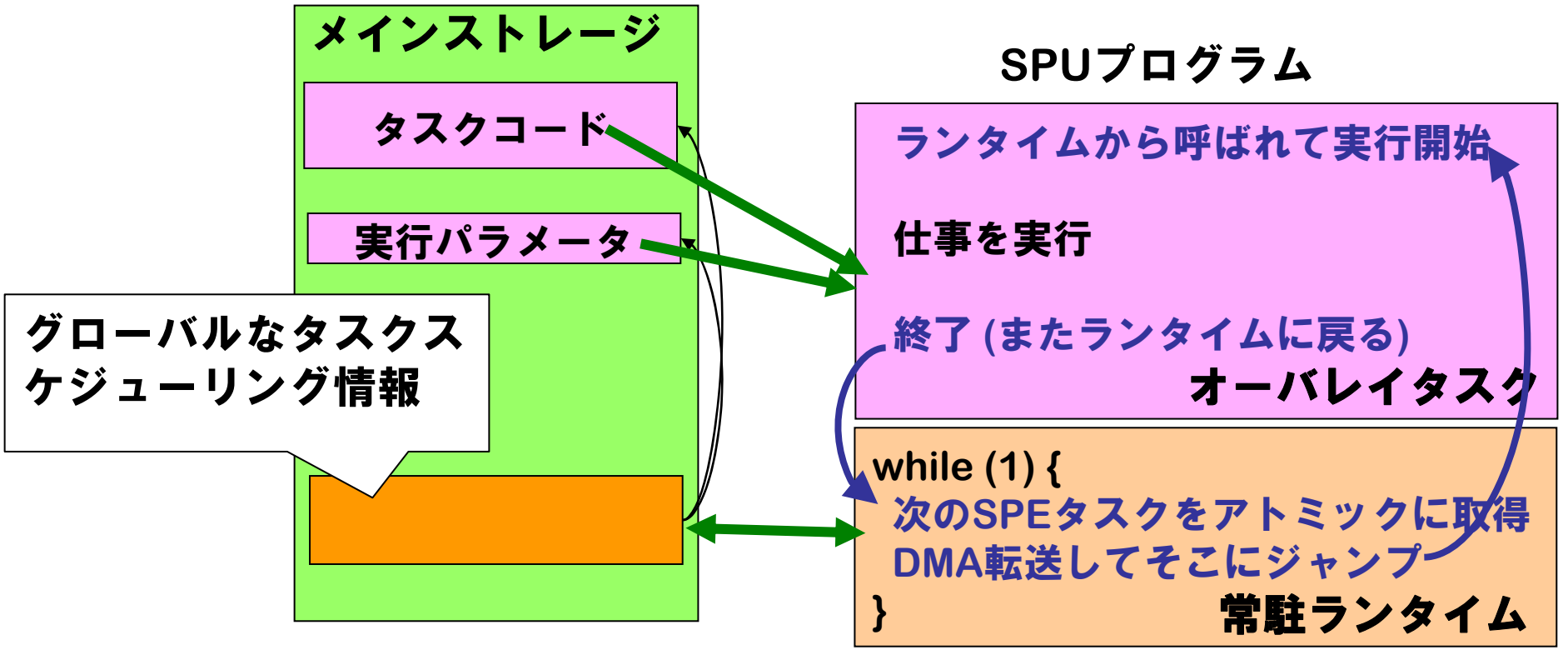
- ✓ コードもデータも保護なしでフラットな空間に置かれる
- ✓ LS上の**プログラムの一部を自分でDMA転送して置き換える**ことでオーバレイ可能
- ✓ 位置非依存のオブジェクトコードであれば任意の場所に、リンク済の実行コードであれば(Elfヘッダなどを元に)規定アドレスにロード



より一般化されたSPE-Centricプログラムの組立て

➔一般化されたランタイムとオーバレイタスクの実現

- ✓ SPEコードをメモリから自立的に取得して自己書き換えするような**常駐ランタイム**を作ることで軽量なオーバレイタスクを実現可能
- ✓ あらかじめメインメモリ上にLSとレジスタを退避する空間を確保すれば実行中のコンテキストスイッチも可能



CEDEC 2007

Cell/B.E.プログラムを速くする(1): SIMDプログラミング

Fixstars Corporation
Nisshin Building 3F, 1-8-27, kounan, Minato-ku, Tokyo 108-0075, Japan
Tel:+81-3-5781-5001 Fax:+81-3-5781-5002

Cell/B.E.のプロセッサパワーを引き出すには？

➔Cell/B.E.の基本的な性能特性

- ✓ 潜在的には非常にパワフルなプロセッサ
- ✓ 性能が出るようにプログラムしないと性能が出ないプロセッサでもある

➔高い性能を達成するためのヒント

- ✓ SPEを使う・SPEになるべく自立的に処理させる
- ✓ SIMD演算器を活かしてSIMDプログラミングをする (特にSPUで)
- ✓ メモリアクセス階層を意識してメモリアクセスレイテンシを隠蔽する (レジスタ ← LS ← メインストレージ)
- ✓ SPE/SPUのアーキテクチャ的特徴を活かす
  - ■ MFCによるバルクDMA転送とMFCの独立性を活用する– 転送と演算の多重化
  - ■ 128個の128-bitレジスタを活用する – 128-bitのベクタデータをできるだけ使い、ループアンローリングなどでレジスタを贅沢に使う
  - ■ SPUのパイプライン特性を活かして二命令同時発行(デュアルイシュー)を増やし分岐ミスを削減する

Cell/B.E.プログラムの主な最適化手法

➔主な最適化手法と速度向上の目安 (あくまで目安)

- ✓ DMAダブルバッファ、ループアンローリング、一般的な最適化手法
  - ■ 数パーセント～数倍
- ✓ SPE化、PPUコードのSIMD化 (VMX利用)
  - ■ 数倍 (SPE化だけでは速くならないことも多い)
- ✓ 複数SPEによる並列化
  - ■ 数倍
- ✓ SPUコードのSIMD化
  - ■ 数倍～数十倍

SPE (SPU)ではSIMD化が非常に大きな高速化効果を持つことが多い（その効果を引き出すために他の最適化手法との組み合わせも重要）

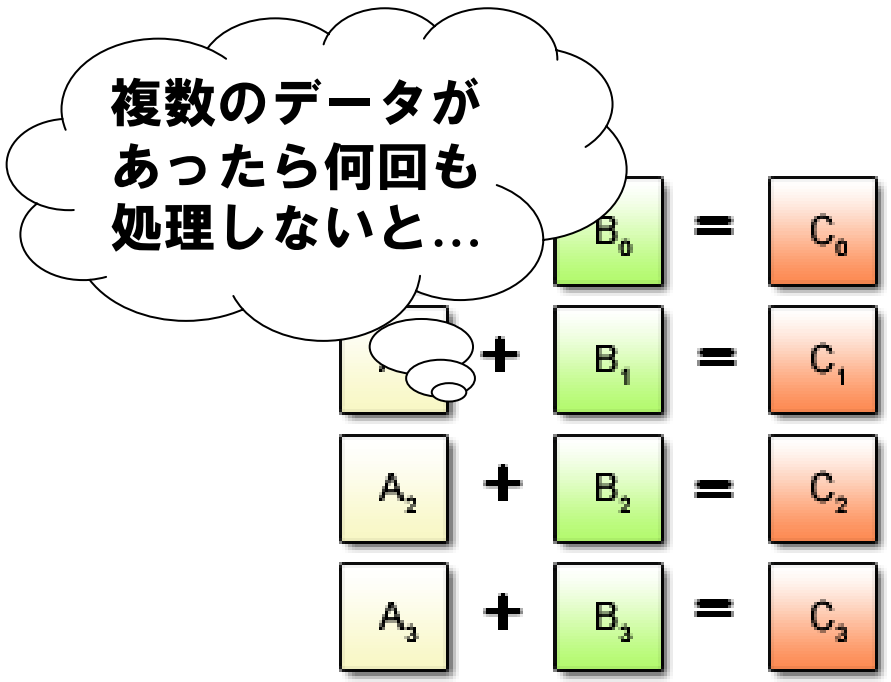
➔SPE化、並列化、SIMD化を組み合わせて100倍以上の速度向上を達成することも(問題によっては)可能

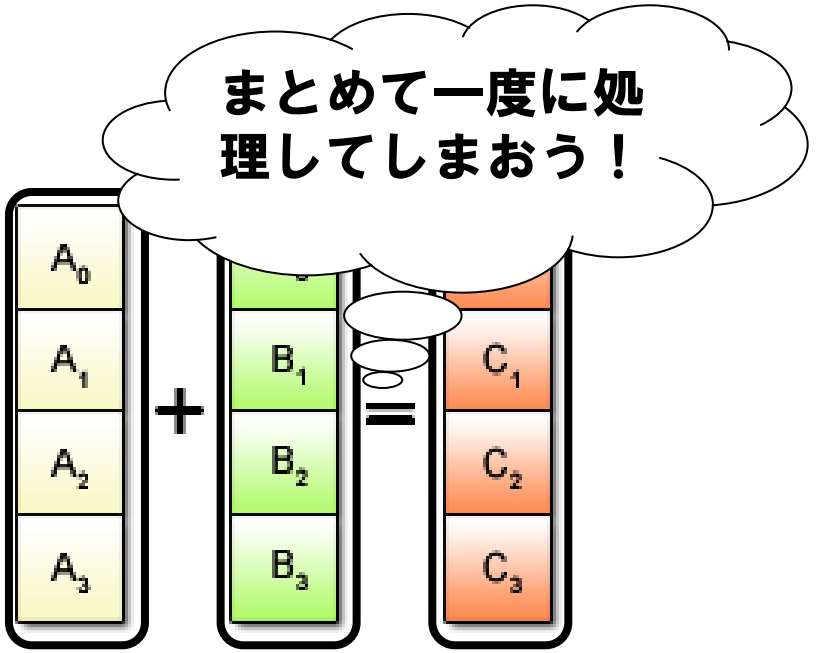
- ✓ 最適化してないPPEプログラムだけの状態はそれくらいCell/B.E.の性能を無駄遣いしている

SIMDとは？

➔SIMD: Single Instruction / Multiple Data

- ✓ 複数データを一度に並列処理する演算や命令のこと
- ✓ 大量のデータに同じような演算を行うマルチメディア処理や行列演算で特に効力を発揮する
- ✓ Cell/B.E.ではIntrinsicsと呼ばれる組み込み関数を使って関数的にSIMD演算 (SIMD命令による複数データの並列演算)を記述できる





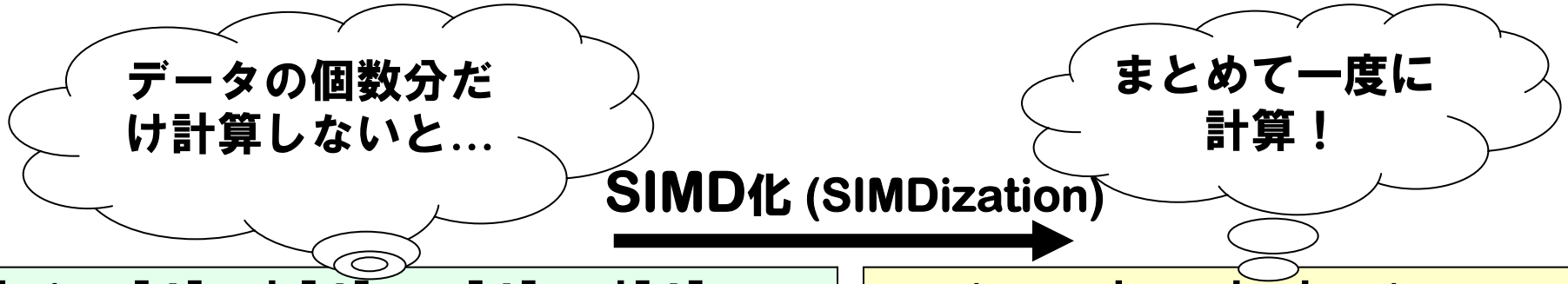
SIMD演算のための新しいデータ型：ベクタデータ型

ベクタデータ型 (vector data type) とは?

- ✓ **SIMD演算の処理**などのために「同じ型を持つ複数のデータ」を1つにまとめて扱えるようにしたデータ型 (Cell/B.E.のための拡張)
  - 配列に似たイメージ (キャストにより相互変換可能)
  - たとえば「unsigned int」型の値を要素に持つベクタデータは「vector unsigned int」と言う型名になる
- ✓ Cell/B.E.ではすべてのベクタ型は**128bit長**
  - 各ベクタ型のデータは、128bitを要素の型のサイズで割った個数だけの要素を持つ

SIMD演算をするプログラムコードの例

➔積算の結果を加算する積和の例




```
int a[4], b[4], c[4], d[4];

for (i = 0; i < 4; i++) {
   d[i] = a[i] * b[i] + c[i];
}
```

SIMD化前のコード

```
vector signed short va, vb;
vector signed int vc, vd;

vd = spu_madd(va, vb, vc);
```

SPU SIMD化後のコード

➔上のコード例では：

- ✓「vector signed short」や「vector signed int」がベクタデータ型
- ✓「spu_madd」が積和SIMD演算のためのIntrinsics (組み込み関数)

SPU SIMDとSPU SIMD Intrinsicsの概要

SPU SIMDの特徴

- SPUでは**全レジスタが128bitレジスタ**で、命令も**ほとんどSIMD命令**
- PPUのVMX SIMDに比べるとよりコンパクトな命令セットとなっており、演算対象のデータ型も制限されている (float型がもっとも演算が豊富)

主なSPU SIMD命令とIntrinsics

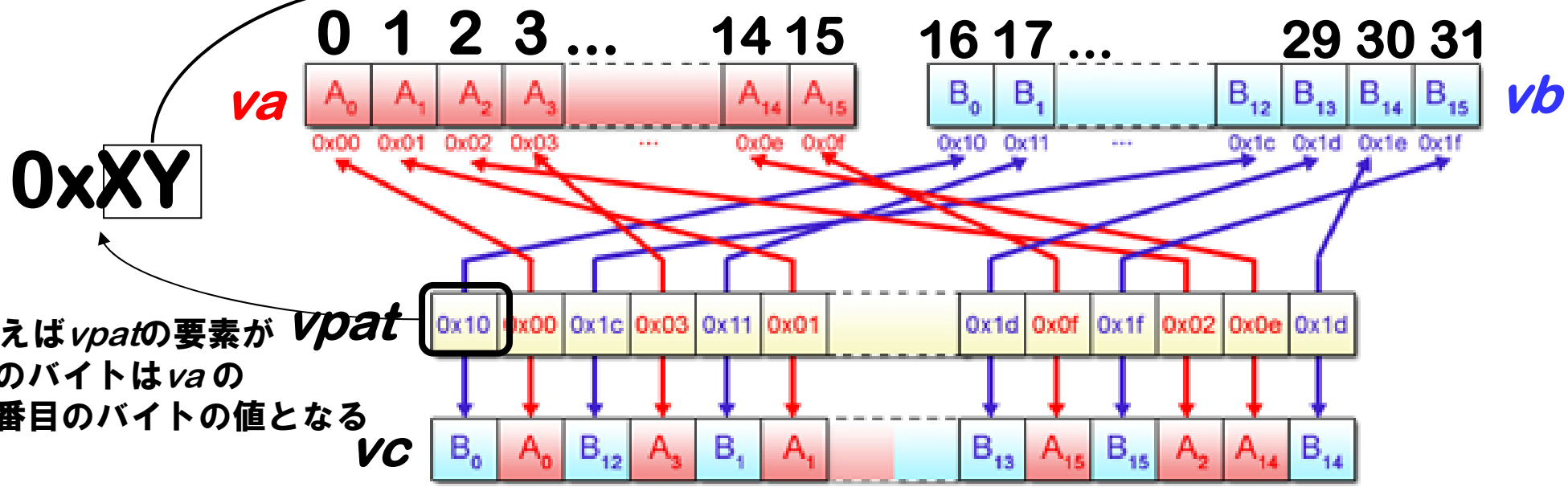
- 四則演算、逆数、平方根などの**算術演算** (spu_add, spu_sub, spu_madd, spu_nmsub, spu_re, ...)
- AND, OR, NORなどの**論理演算** (spu_and, spu_or, ...)
- ベクタデータの要素の並べ替え、ビットパターンによる値選択などの**シャッフル演算・ビット演算** (spu_shuffle, spu_sel)
- シフト、ローテートなど**シフト・ローテート演算** (spu_rl, spu_sl, ...)
- 比較のための**比較演算** (spu_cmpeq, spu_cmpgt)
- スカラデータとベクタデータを相互に扱うための**スカラデータ演算**など (spu_extract, spu_insert, spu_promote)

便利なSPU SIMD命令: バイトシャッフル (spu_shuffle)

➔バイト列の任意のシャッフル(並び替え)

- ✓ *vc* = spu_shuffle (*va*, *vb*, *vpat*)
  - ベクタ値*va*と*vb*の各バイトを並び替えて合成し、新しいベクタ*vc*を生成する
  - *vpat*はvector unsigned charであり、 *vpat*の*n*番目の要素が結果のベクタ*vc*の*n*番目のバイトを決定する

*vpat*の各バイトの下位5bit が *va, vb*の32バイトのうち何バイト目を使うかを決める (0～15なら*va*のバイト、16～31なら*vb*のバイトとなる)

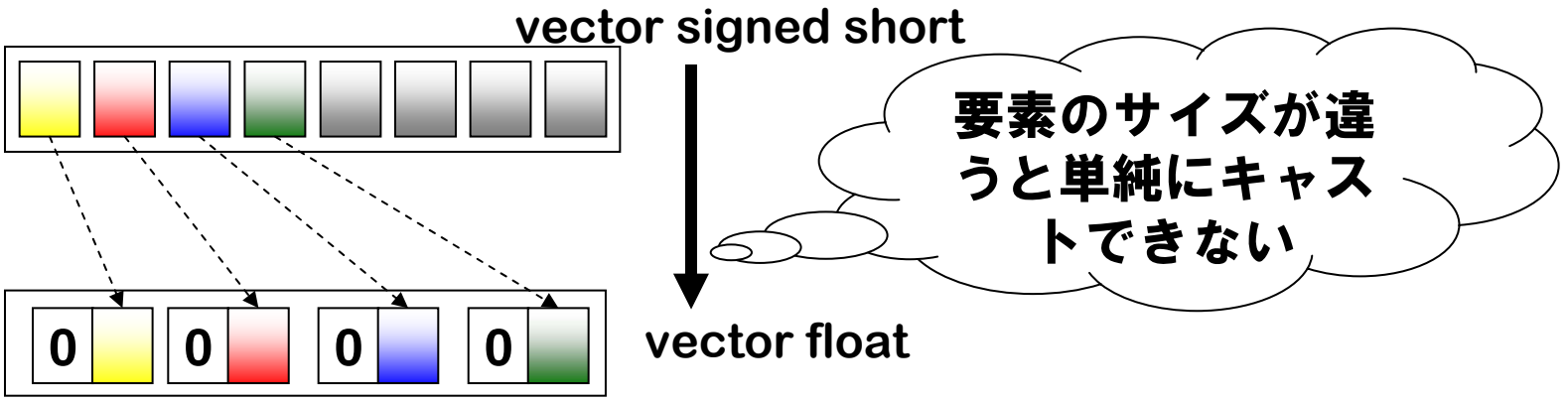


例えば*vpat*の要素が
3 のバイトは*va*の
4 番目のバイトの値となる

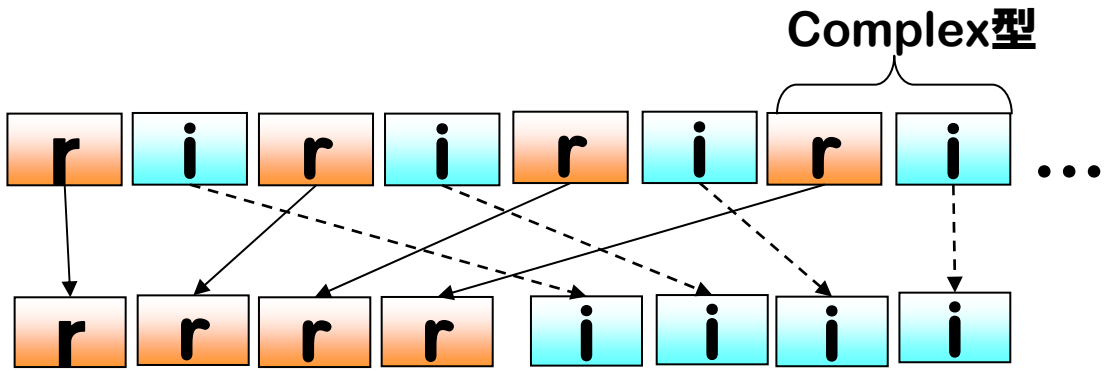
shuffleによるバイトデータの並び替え例 (1)

- SIMD演算化の前処理・後処理でバイト並び替えが必要なことが多い → バイトシャッフル命令が有効

例1: vector short (2バイトx8要素)のデータをvector float (4バイトx4要素)に変換して処理したい

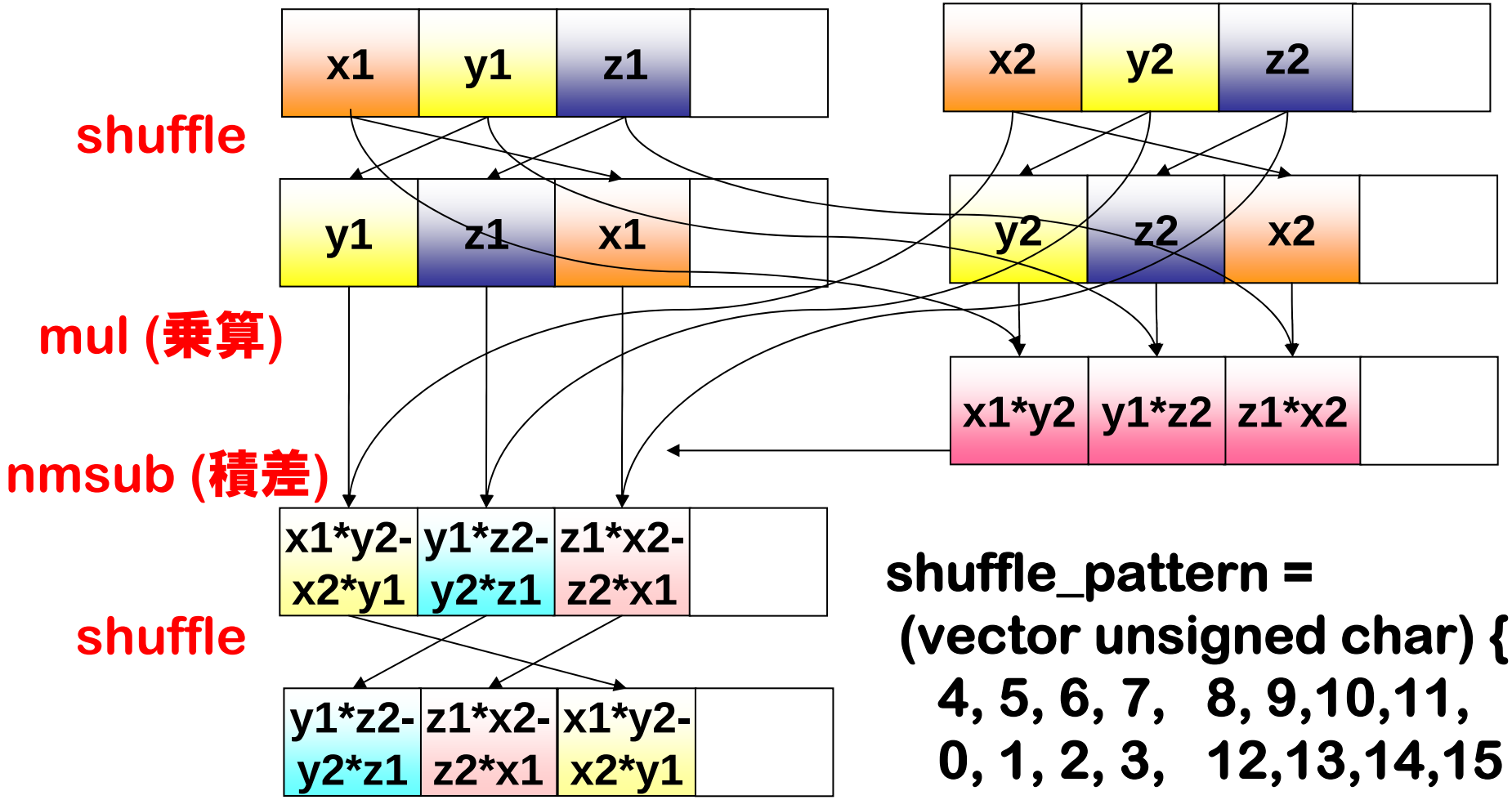


例2: 複素型の配列から実部と虚部を分けて取り出して処理したい



shuffleによるバイトデータの並び替え例 (2)

➔SIMD演算によるベクタの外積計算の例



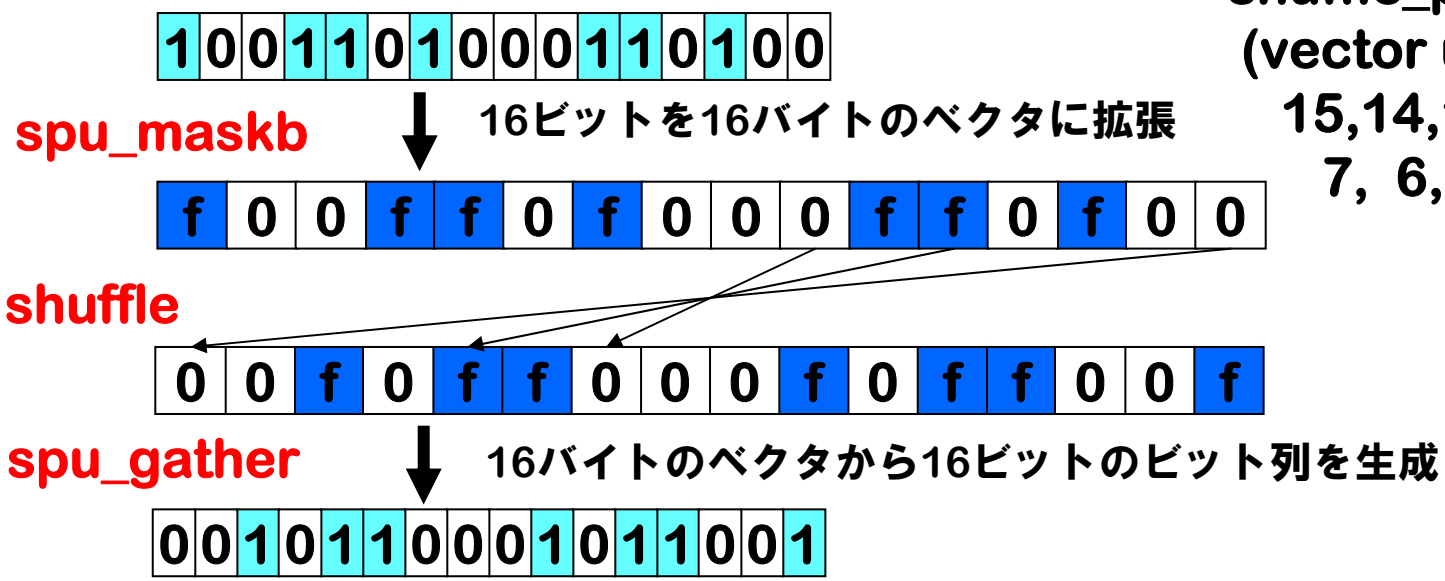

```
shuffle_pattern =
 (vector unsigned char) {
    4, 5, 6, 7,   8, 9,10,11,
    0, 1, 2, 3,   12,13,14,15 };
```

shuffleのビット並び替えへの応用

バイト列とビット列の相互変換に使えるSIMD命令

- ✓ *vmask* = spu_maskb (*a*)
  - スカラ値*a*の下位16ビットについて、各ビットを8回ずつ複製することで128ビットのバイトマスク*vmask*を生成する
- ✓ *vb* = spu_gather (*va*)
  - ベクタ*va*の各要素値のLSBを集めて連結し、*vb*の0番目の要素の下位ビットへ格納して返す

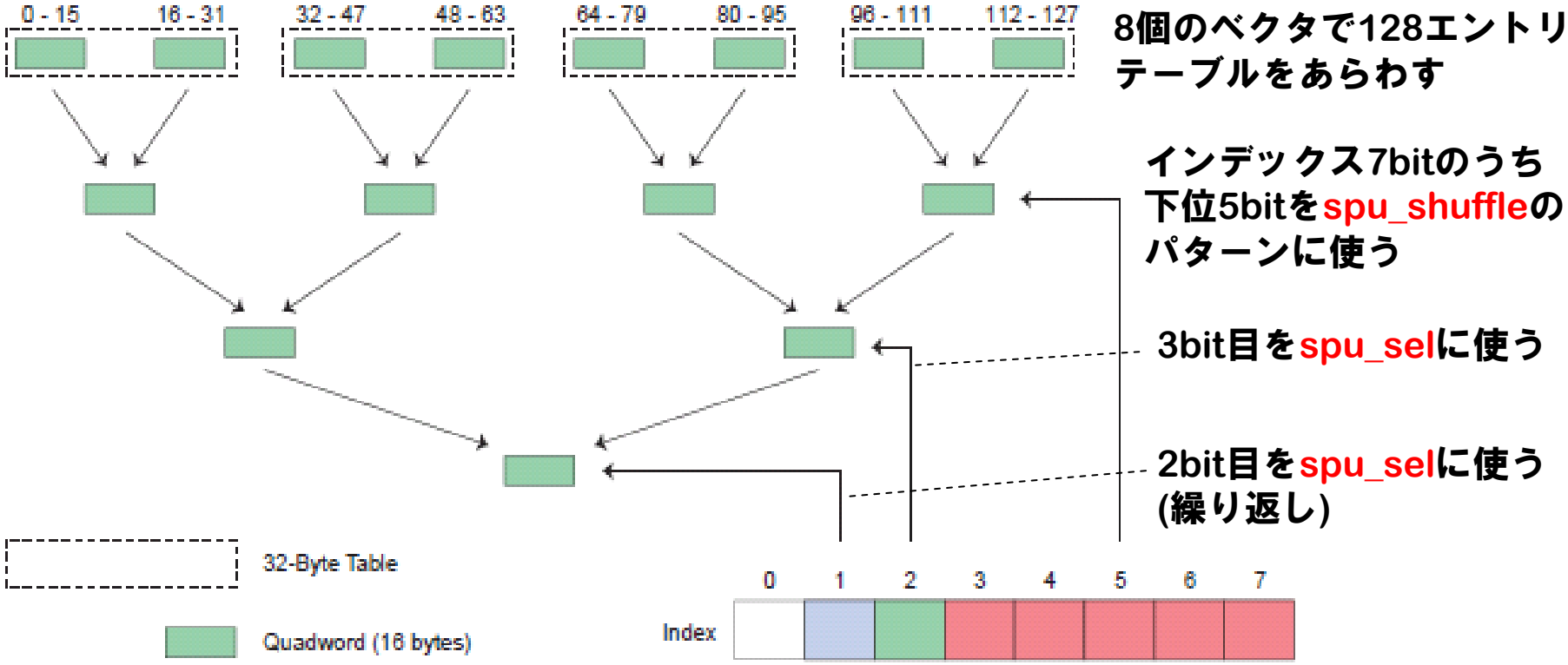
shuffleを使ったビット反転の例




```
shuffle_pattern =
 (vector unsigned char) {
   15,14,13,12,11,10, 9, 8,
    7,  6,  5,  4,  3,  2,  1, 0 };
```

shuffleのテーブルルックアップへの応用

- shuffle命令を用いたテーブルルックアップ
  - ✓ テーブルエントリ数が32以下ならshuffle命令を使って16並列にルックアップできる
    - テーブルを16バイトx2のベクタとし、インデックス値のベクタをパターンに使う
  - ✓ 32エントリよりもエントリ数が多い場合、2分木的なルックアップ手法が考えられる
    - spu_shuffleとspu_sel (bit select命令, パターンのビットが0ならベクタ1から、1ならベクタ2からビットを選択して新しいベクタを作る) を使う



128エントリの16並列ルックアップ例 (from IBM CBE Programming Handbook)

その他の便利なSPU SIMD命令: バイト操作命令

➔バイト操作命令群

- ✓ 要素がバイト(unsigned char)であるベクタにしか使えないが、便利な計算を一度にできる命令がいくつか用意されている

- ✓ *vd* = spu_absd (*va*, *vb*)
  - *va*の各要素値から*vb*の各要素値を引き、その結果の絶対値を返す (element-wise absolute difference)
- ✓ *vd* = spu_avg (*va*, *vb*)
  - *va*の各要素値と*vb*の各要素値を足してさらに1を足し、その結果を1ビット右シフトした(=2で割った)結果を返す (average of two vectors)
- ✓ *vd* = spu_sumb (*va*, *vb*)
  - *va*, *vb*は vector unsigned char型、 *vd* はvector unsigned short型
  - *va*の隣り合う4要素を足した結果が*vd*の対応する奇数要素に、*vb*の隣り合う4要素を足した結果が*vd*の対応する偶数要素に格納される

- ✓ spu_absd, spu_sumbを組み合わせるとSAD (Sum of the Absolute Differences) の計算などに便利

CEDEC 2007

Cell/B.E.プログラムを速くする(2): プログラム最適化のヒント

Fixstars Corporation
Nisshin Building 3F, 1-8-27, kounan, Minato-ku, Tokyo 108-0075, Japan
Tel:+81-3-5781-5001 Fax:+81-3-5781-5002

最適化のヒント: SPE化とSIMDプログラミング

➔SPE化のポイント

- ✓ 高速化のためにSPE化する場合、基本的にSIMD化も視野にいれる
  - SIMD化しない場合、単体SPEではPPEとあまり変わらないことも
  - 並列化できる処理なら複数SPEを使えば(SIMD化なしでも)高速化はもちろん可能
- ✓ SPEはSIMD化・最適化すれば1個でもかなり高速
- ✓ 複数SPEを使ってSIMD化もするとPPEがボトルネックになりやすい
  - DMA転送やメッセージ通信をうまく使ってできるだけSPEだけで自立的に処理させる部分を長くする

➔SIMD化とベクタデータの活用

- ✓ SPUアーキテクチャは128bit SIMD型アーキテクチャ
  - レジスタも全部128bit長
- ✓ 128bitのベクタデータを使ったSIMD処理が最も効率が良い
  - 128bitより小さい基本型を使った演算では、余分なコストが発生してることも！
  - 連続してないデータに対する処理はSIMD化しにくいが、前処理などによってSIMD化できるか検討した方が良い

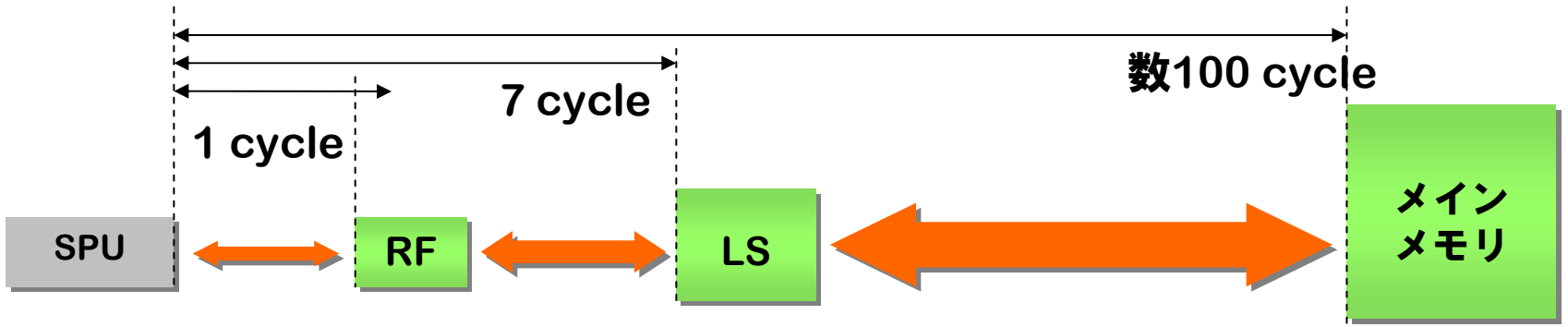
最適化のヒント: メモリアクセス階層とレイテンシの隠蔽

➔ SPEにおけるメモリアクセスの階層構造

- ✓ SPUからレジスタ(Register Files, RF)へのアクセス：1 cycle (16B)
- ✓ SPUからLSへのアクセス：1 + 6 cycles (16B)
- ✓ SPUからメインメモリへのアクセス (DMA転送)：数100cycles (128B)

➔ メモリアクセスレイテンシの隠蔽が重要

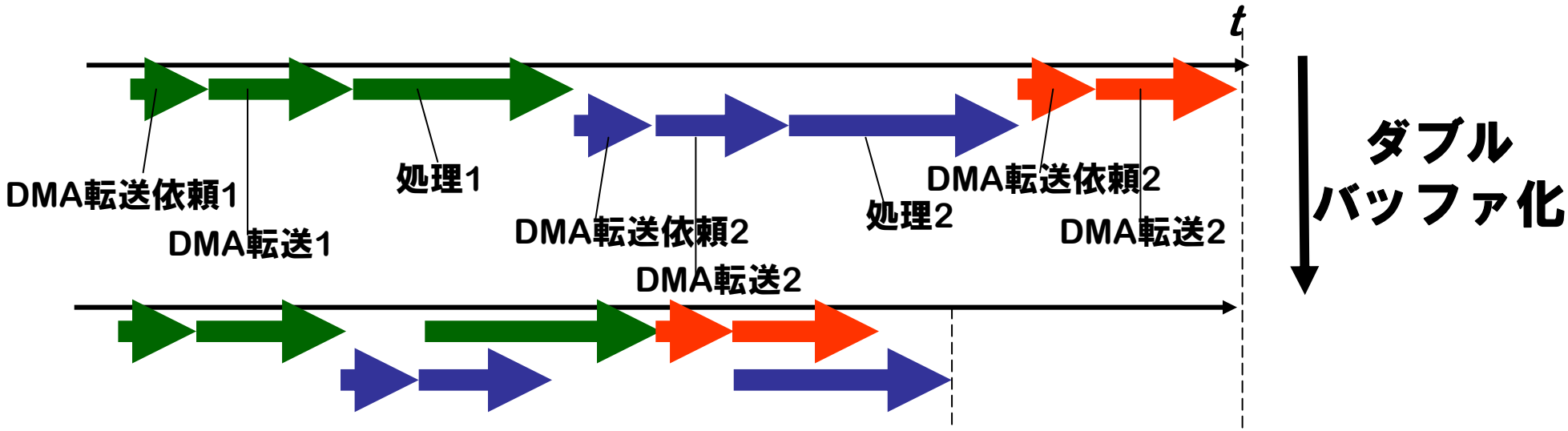
- ✓ LSへのアクセスレイテンシの隠蔽... ループアンローリングやIntrinsicsでの記述による128本の豊富なレジスタの積極的な活用
- ✓ メインメモリへのアクセスレイテンシの隠蔽... DMAダブルバッファによる演算と転送の多重化



最適化のヒント: DMAダブルバッファリング

➔DMAダブルバッファリング：

- ✓ 転送用バッファを2つあるいは複数用意し、片方を転送中にもう片方の転送済みデータを使って処理を行う
  - ■ 転送タグIDの切替とバッファの切替は連動させる
  - ■ 例えばタグIDの切替には xor (^) を使うと便利
- ✓ DMA転送時間と演算時間を多重化させることが可能になる



最適化のヒント: ループアンローリング

ループアンローリングとは

- ループがあるとき、その繰り返しをいくつか手で展開する
  - ある程度はコンパイラがやってくれることも
  - SPEはレジスタ数が多いこともあり、**手動でのアンロールにかなり効果がある**
- ループを手で展開してレジスタを多用することで以下を達成する
  - コンパイラが最適化のために使えるコード範囲を広げる
  - レジスタ競合による依存関係が減り、ストールが隠蔽される
  - ループ内のロード/ストア総数が減り、LSへのアクセスレイテンシが隠蔽できる
  - 分岐数が減少するので最適化が期待できる
- 一方で、コードが長くなって見にくくなる弊害もある

```
for (i = 0; i < N; i++) {
  a[i] = b[i] * c[i];
}
```

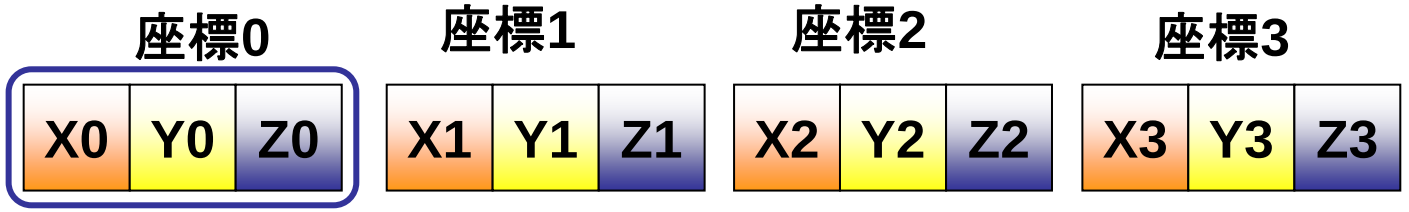
```
for (i = 0; i < N;) {
  a[i] = b[i] * c[i]; i++;
  a[i] = b[i] * c[i]; i++;
  a[i] = b[i] * c[i]; i++;
  a[i] = b[i] * c[i]; i++;
}
```

左のコードより右のコードの方が速くなりやすい

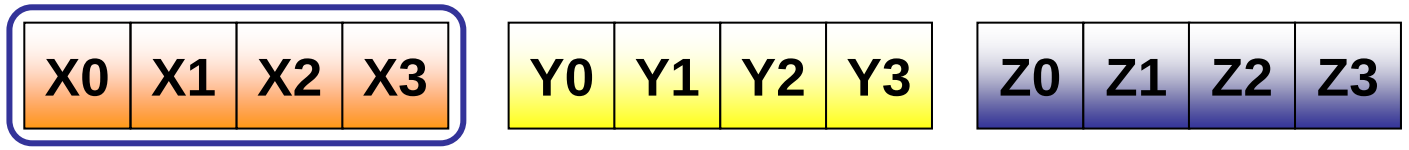
最適化のヒント: データ配置の考慮

➔SIMD演算しやすいデータ配置の検討: AOSとSOA

- ✓ 例えば3D画像の座標データを使って計算する場合
- ✓ 各点の情報はx, y, zを一まとまりとする構造体を使って次のように配置されているのが普通 (Array of Structure)



- ✓ しかし、複数座標のx, y, zそれぞれが並んで配置されている方が効率よくSIMD演算できるかも知れない (Structure of Array)
  - ■ SPEでは汎用的なSIMD演算が可能なので、shuffle命令の活用やデータ配置の工夫を行うことでより効率よく処理できる可能性がある



最適化のヒント: SPUパイプラインの考慮

➔二命令同時発行 (デュアル・イシュー)

- ✓ SPUは Even / Odd の非対称なパイプライン構造を持つ
  - ■ Even パイプライン (パイプライン0) は演算系命令を実行
    - – 算術演算、シフト・ロテート、論理演算など
  - ■ Odd パイプライン (パイプライン1) はロード・ストアと制御系命令を実行
    - – ロード・ストア、分岐と分岐ヒント、シャッフル(shuffle)、LNOPなど
- ✓ もし命令が”0”, “1”, “0”, “1”, …のように綺麗に整列可能で、かつ隣り合う”0”と”1”の命令に依存性や競合がなければ同時発行が可能

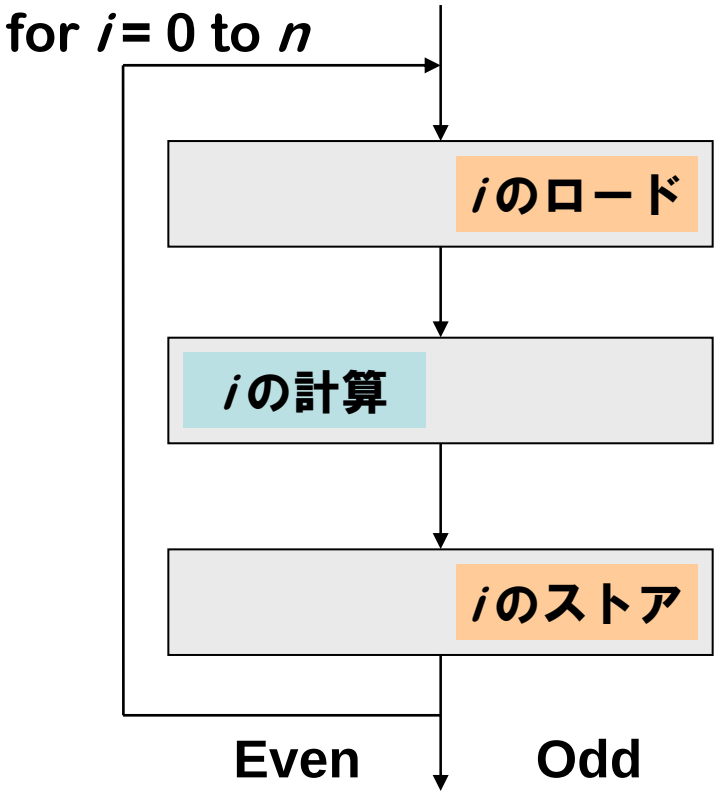
➔分岐ミスペナルティの回避

- ✓ SPUはハードウェア的な分岐予測機構を持たず、かつパイプラインがかなり深い → 分岐の削減や分岐ヒントの挿入が性能に大きく影響
- ✓ 分岐ヒントは…__builtin_expect(*条件文*, *0か1*)のように記述できる
- ✓ 分岐の削減とは…分岐先の演算を両方計算しておき、条件式を比較とビットマスク演算に置き換えることで必要な回答だけを最後に得る

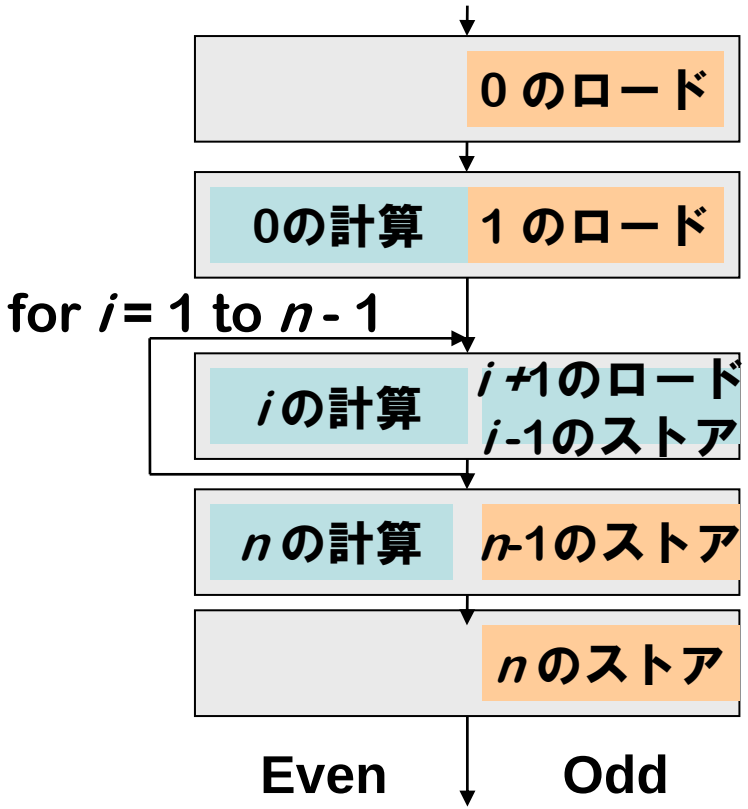
二命令同時発行を増やす: ソフトウェアパイプライニング

➔ソフトウェアパイプライニング

- ✓ ループをソフトウェア的にパイプライン化してやることで、ループ内で二命令同時発行が起こりやすいように組み替える



一般的なループ



パイプライン化

分岐ミスペナルティの回避 (1)：分岐ヒントの活用

➔ 分岐先に偏りがあるような場合、分岐ヒント命令を使ってメジャーなケースでの分岐ミスをできるだけ減らす

```
if (__builtin_expect(flag, 1)) {
   /* 多くの場合こちらを通る */
} else {
   /* めったに通らない */
}
```

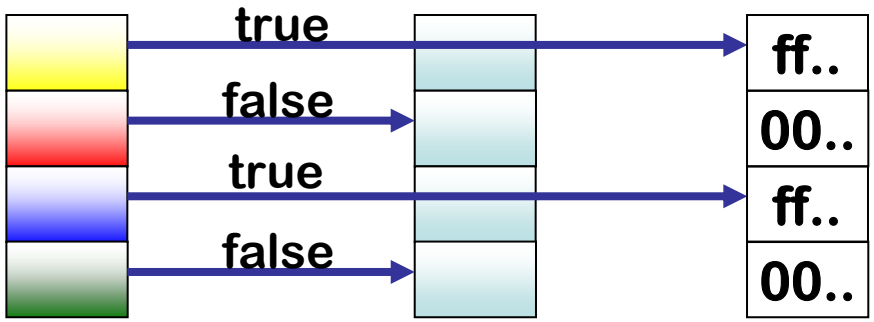
➔ ソフトウェア的な動的予測を使う例

```
while (...) {
  ...
  cond = ...
  if (__builtin_expect(cond, expect)) {
     /* expect=1ならこちらを通るとノーペナルティ*/
  }
  expect = cond;  /* 次の分岐は前回と同じと予測 */
}
```

分岐ミスペナルティの回避 (2)：条件分岐の演算化

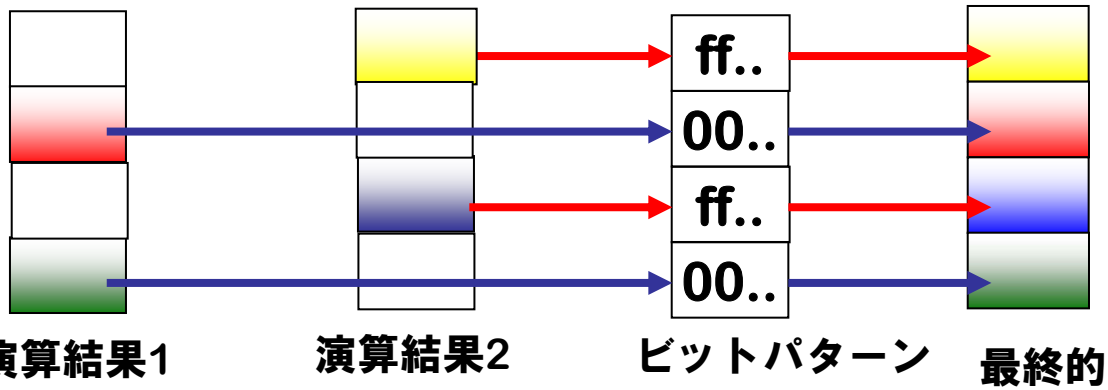
➔ 比較とビット選択命令を使って分岐の演算化する

1. 条件分岐の先の各ケースの演算をあらかじめ計算しておく
2. 比較命令で条件を判定し、結果のビットパターンを作る



spu_cmpeqやspu_cmpgt
で各要素を比較すると、
trueの要素には全ビット1
falseの要素には全ビット0
が設定される

3. 1. の結果のベクタ値に対し、2. で作ったビットパターンを使って
ビット選択命令spu_selにより条件に合う方の結果だけを取り出す



spu_selはパターンのビットが
0なら1つめのベクタから
1なら2つめのベクタから
該当するビットを選ぶ

最適化によるSPUコードの速度向上例

➔単純な演算ループを教科書的に最適化した例

- ✓2つの単精度浮動小数点の配列から値を順に取ってきて乗算するだけのループ
- ✓SPE化 → PPE上での実行から1.5倍
- ✓SIMD化 (4並列計算) → さらに6倍
- ✓ループアンローリング (4ループ結合) → さらに2.9倍
- ✓デュアルイシュー率の向上 → さらに1.2倍

➔全体で37倍以上の最適化

- ✓単一SPE上の最適化のみ

Cell/B.E.プログラミングの基礎：まとめ

- Cell/B.E.は1+8個の非対称なコアを持つプロセッサ
  - ✓ PPUを実行ユニットに持つ1個のPPE
  - ✓ SPUを実行ユニットに持つ8個(6個)のSPE
- Cell/B.E.上のソフトウェアはPPUプログラムと(複数の)SPUプログラム群から構成される並列プログラム
  - ✓ 各SPE上のMFCによって提供されるDMA通信、メッセージング、アトミック更新などによってPPEや複数SPE間での協調・通信が可能
- Cell/B.E.プログラムで性能のカギを握るのはSPE化とSIMD化
  - ✓ 複数のSPEが無駄なく自立的に動作し続けられるのが理想
  - ✓ 細粒度のデータ並列処理を提供するSIMD化が性能に大きく影響

Thank you for listening!

➔質疑応答

➔Cellプログラマを会社で募集しています。