第４回計算科学技術特論B(2026)
スレッド並列＋コア単体性能最適化詳細(１)

株式会社 waveZ
熊畑清

2026年5月7日（木）13:00－14:30

主催：高度情報科学技術研究機構(RIST)
　　　次世代HPC・AI研究開発支援センター(HAIRDESC)
共催：東京大学物性研究所
後援：理化学研究所計算科学研究センター、
　　　計算物質科学人材育成コンソーシアム(PCoMS)

計算科学特論B 第4回

スレッド並列＋コア単体性能最適化詳細(1)
単体性能の最適化に関係する各種の要素について

株式会社waveZ
熊畑　清

2026.5.7(木)

概要

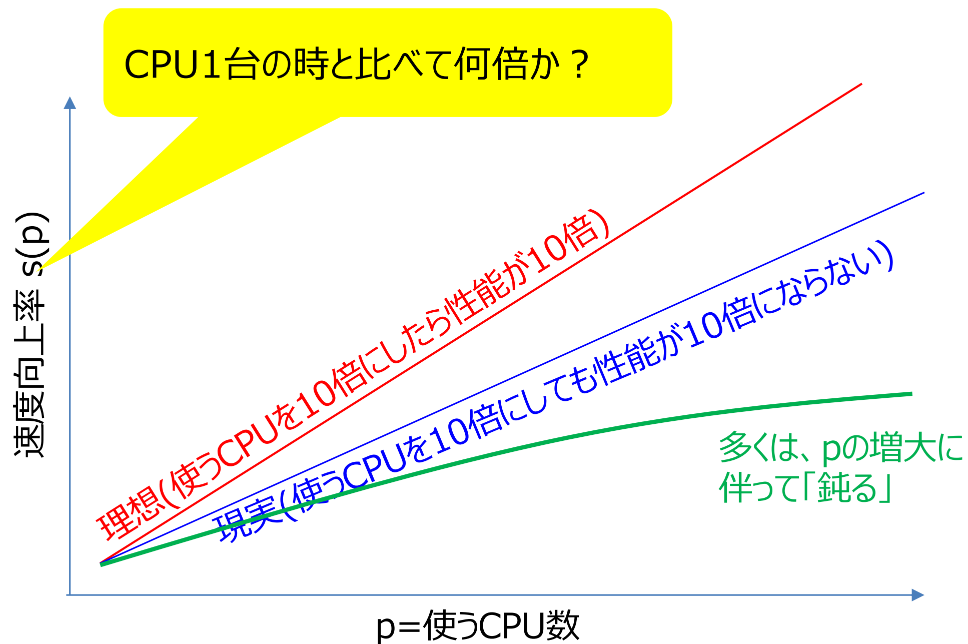
waveZ

- 4月9日 スーパーコンピュータとアプリケーションの性能
- 4月16日 アプリケーションの性能最適化1(高並列性能最適化)
- 4月23日 スレッド並列＋コア単体性能最適化概要
- 5月7日 スレッド並列＋コア単体性能最適化詳細(1)
- 5月14日 スレッド並列＋コア単体性能最適化詳細(2)

並列性能と、CPU単体性能

waveZ

並列性能



1 CPUでの実行時間はT1
p CPUでの実行時間をT(p)
S(p) = T1 / T(p)

CPU単体性能

CPU単体性能とは、読んで字のごとく、一つのCPUだけに着目した性能

例)
たくさんの荷物を、何台かの車で分担して運びたい

- なるべく大きな車を用意したい
- なるべくたくさん車を用意したい
- なるべく効率よく荷物を積みたい

こっちの観点

本日はCPU単体性能をより良くするために、考えないといけないことを紹介

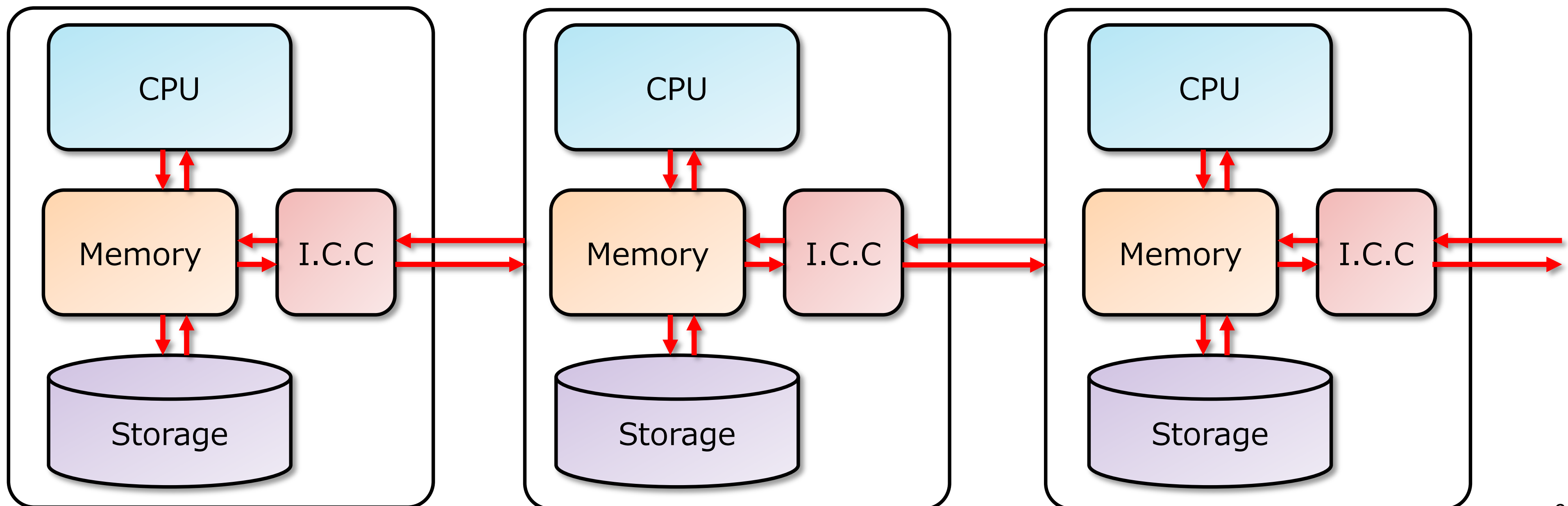
並列処理の様々なレベル

並列のレベル

waveZ

- 第1回の講義で並列処理について紹介があった
- 並列処理にはハード的にどう言うレベルで行うかに応じて種類がある

- MPI並列（分散メモリ）

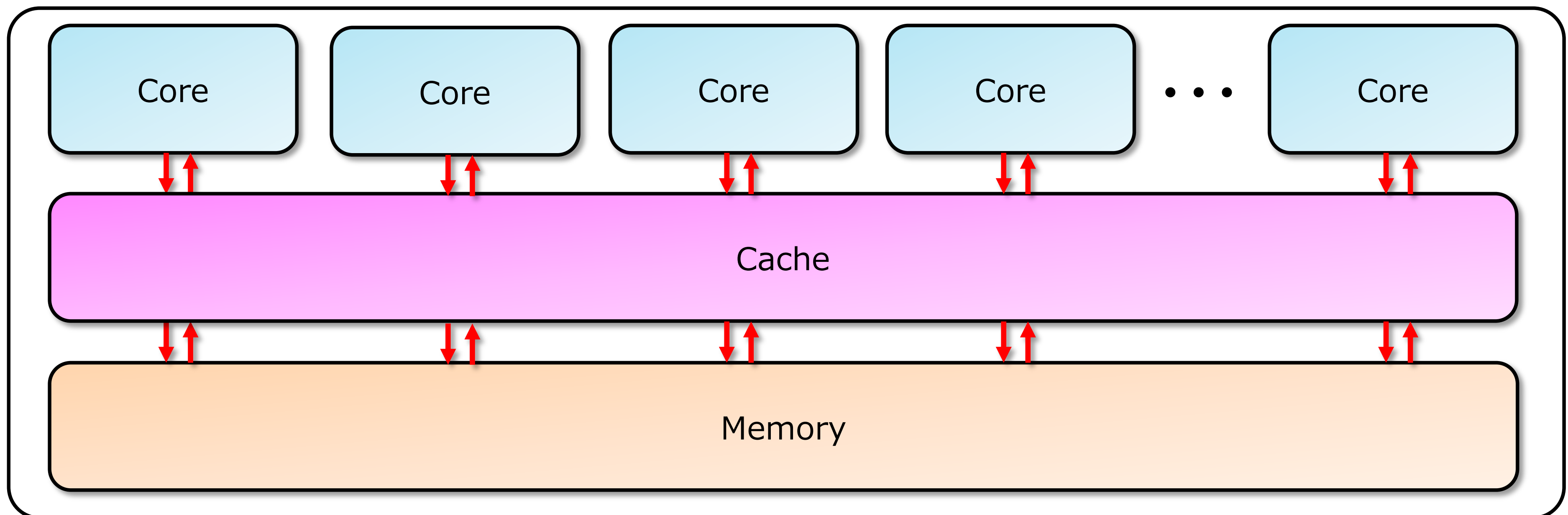


並列のレベル

waveZ

- 第1回の講義で並列処理について紹介があった
- 並列処理にはハード的にどう言うレベルで行うかに応じて種類がある

- スレッド並列（共有メモリ）

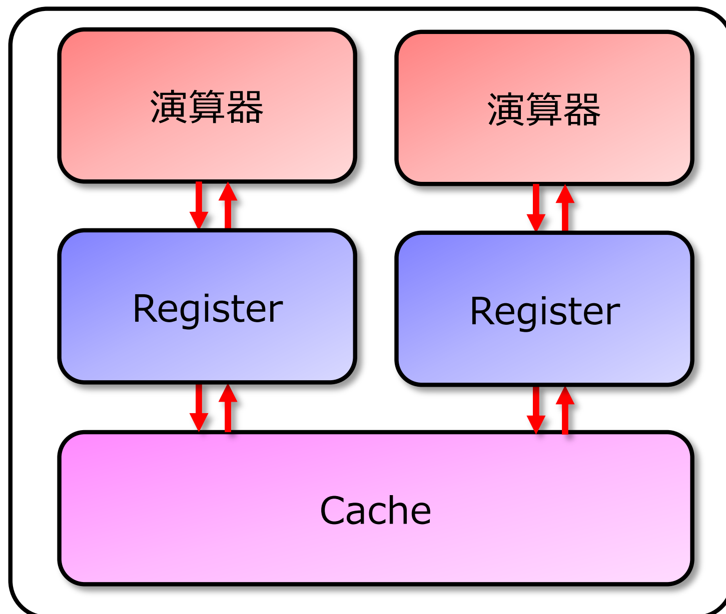


並列のレベル

waveZ

- 第1回の講義で並列処理について紹介があった
- 並列処理にはハード的にどう言うレベルで行うかに応じて種類がある

- コア内の並列（命令レベル並列）



単体性能（＝通信をしない処理）の向上には
- スレッド並列
- コア内の並列
- メモリ・キャッシュ・レジスタ間のデータ転送

が重要

本日の内容

waveZ

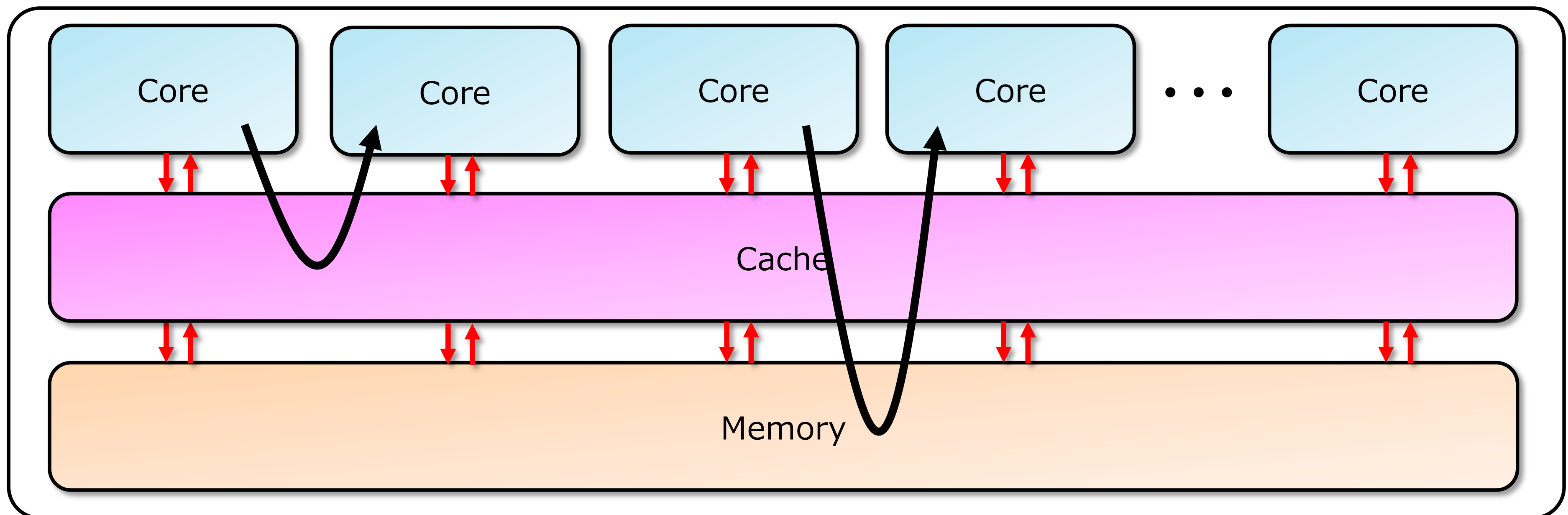
- 単体性能のおはなし
- スレッド並列
  - プロファイラ（CPU性能解析レポート）
  - リカレンスの除去とカラーリング
- コア内の並列性、演算命令とスケジュール
  - SIMD
  - ソフトウェアパイプライニング
- メモリ・キャッシュ・レジスタ間のデータ転送
  - キャッシュのアクセス効率化
  - ラインアクセス
  - キャッシュスラッシングとFalse Sharing
  - ストリームの考え方とZ-Fill
  - プリフェッチ

スレッド並列

スレッド並列

waveZ

- スレッド並列（共有メモリ）
- 一つのCPU（プロセス）内で仕事を分担して並列処理
- 通信はしないが、コア間の情報交換は必要で、前回の高並列性能最適化と同様にリカレンス（データの依存性）を気にしないといけない



リカレンス（データの依存性）

waveZ

ある有限要素法の流体解析プログラム

```
DO IE=1,NE
  IP1=NODE(1,IE)
  IP2=NODE(2,IE)
  IP3=NODE(3,IE)
  IP4=NODE(4,IE)

  SWRK=-S(IE)
  FX(IP1)=FX(IP1)+SWRK*DNX(1,IE)
  FX(IP2)=FX(IP2)+SWRK*DNX(2,IE)
  FX(IP3)=FX(IP3)+SWRK*DNX(3,IE)
  FX(IP4)=FX(IP4)+SWRK*DNX(4,IE)

  FY(IP1)=FY(IP1)+SWRK*DNY(1,IE)
  FY(IP2)=FY(IP2)+SWRK*DNY(2,IE)
  FY(IP3)=FY(IP3)+SWRK*DNY(3,IE)
  FY(IP4)=FY(IP4)+SWRK*DNY(4,IE)

  FZ(IP1)=FZ(IP1)+SWRK*DNZ(1,IE)
  FZ(IP2)=FZ(IP2)+SWRK*DNZ(2,IE)
  FZ(IP3)=FZ(IP3)+SWRK*DNZ(3,IE)
  FZ(IP4)=FZ(IP4)+SWRK*DNZ(4,IE)
ENDDO
```

4角形要素について∇pの有限要素近似を計算するループ

$$\nabla p = \frac{\partial p}{\partial x_i} \cong \sum_{j=1} \frac{\partial N_j}{\partial x_i} p_e$$

型	配列名とサイズ	内容
INTEGER*4	NODE(9,NE)	節点リスト
REAL*4	S(NE)	圧力
REAL*4	DNX(9,NE), DNY(9,NE), DNZ(9,NE)	形状関数の導関数
REAL*4	FX(NP), FY(NP), FZ(NP)	圧力勾配ベクトル

リカレンス（データの依存性）

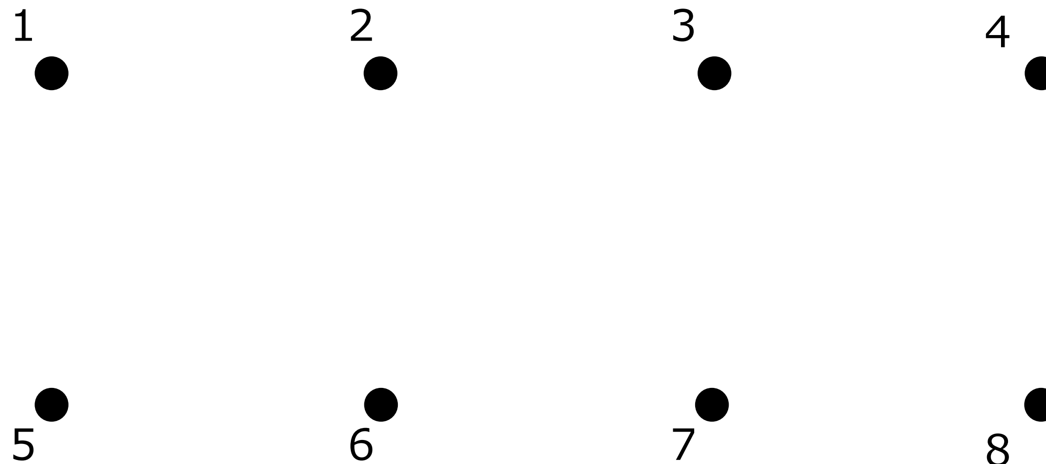
waveZ

ある有限要素法の流体解析プログラム

型	配列名とサイズ	内容
INTEGER*4	NODE(9,NE)	節点リスト
REAL*4	S(NE)	圧力
REAL*4	DNX(9,NE), DNY(9,NE), DNZ(9,NE)	形状関数の導関数
REAL*4	FX(NP), FY(NP), FZ(NP)	圧力勾配ベクトル

4角形要素について∇pの有限要素近似を計算するループ

$$\nabla p = \frac{\partial p}{\partial x_i} \cong \sum_{j=1} \frac{\partial N_j}{\partial x_i} p_e$$



リカレンス（データの依存性）

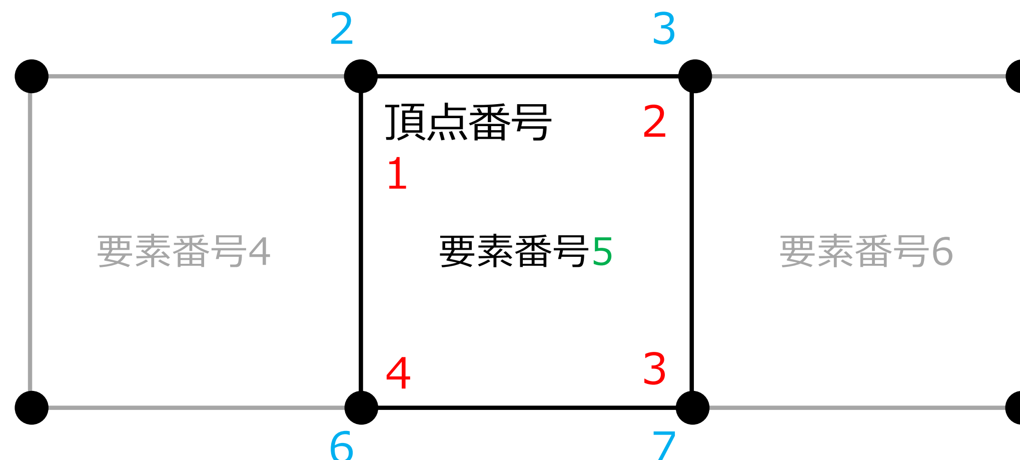
waveZ

ある有限要素法の流体解析プログラム

型	配列名とサイズ	内容
INTEGER*4	NODE(9,NE)	節点リスト
REAL*4	S(NE)	圧力
REAL*4	DNX(9,NE), DNY(9,NE), DNZ(9,NE)	形状関数の導関数
REAL*4	FX(NP), FY(NP), FZ(NP)	圧力勾配ベクトル

4角形要素について∇pの有限要素近似を計算するループ

$$\nabla p = \frac{\partial p}{\partial x_i} \cong \sum_{j=1} \frac{\partial N_j}{\partial x_i} p_e$$



2=NODE(1,5)
3=NODE(2,5)
7=NODE(3,5)
6=NODE(4,5)

リカレンス（データの依存性）

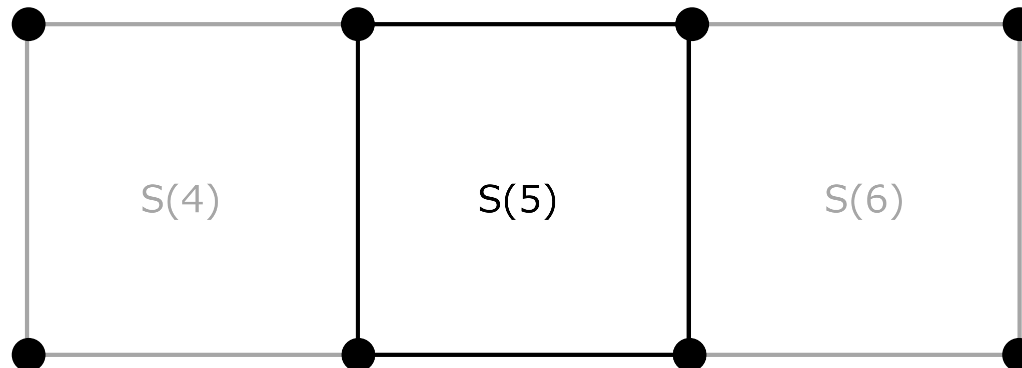
waveZ

ある有限要素法の流体解析プログラム

型	配列名とサイズ	内容
INTEGER*4	NODE(9,NE)	節点リスト
REAL*4	S(NE)	圧力
REAL*4	DNX(9,NE), DNY(9,NE), DNZ(9,NE)	形状関数の導関数
REAL*4	FX(NP), FY(NP), FZ(NP)	圧力勾配ベクトル

4角形要素について∇pの有限要素近似を計算するループ

$$\nabla p = \frac{\partial p}{\partial x_i} \cong \sum_{j=1} \frac{\partial N_j}{\partial x_i} p_e$$



リカレンス（データの依存性）

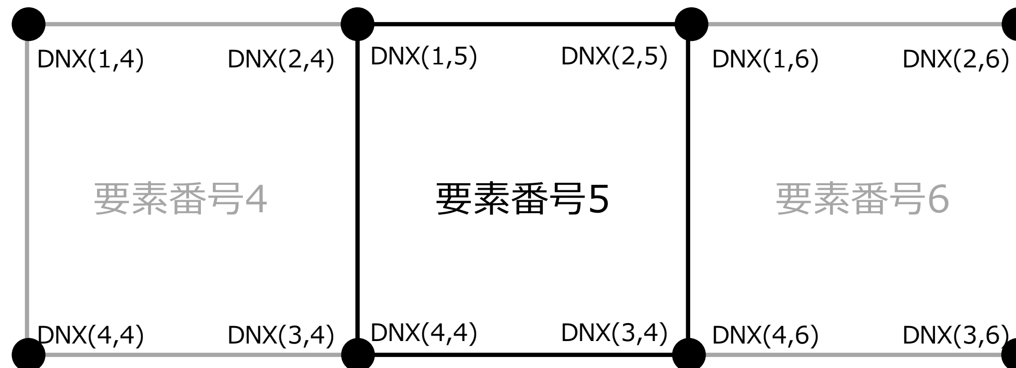
waveZ

ある有限要素法の流体解析プログラム

型	配列名とサイズ	内容
INTEGER*4	NODE(9,NE)	節点リスト
REAL*4	S(NE)	圧力
REAL*4	DNX(9,NE), DNY(9,NE), DNZ(9,NE)	形状関数の導関数
REAL*4	FX(NP), FY(NP), FZ(NP)	圧力勾配ベクトル

4角形要素について∇pの有限要素近似を計算するループ

$$\nabla p = \frac{\partial p}{\partial x_i} \cong \sum_{j=1} \frac{\partial N_j}{\partial x_i} p_e$$



リカレンス（データの依存性）

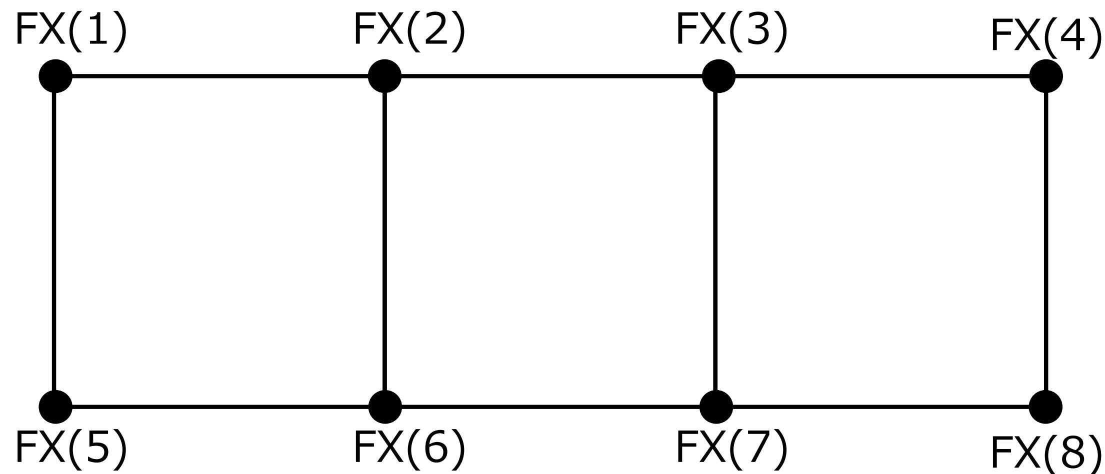
waveZ

ある有限要素法の流体解析プログラム

型	配列名とサイズ	内容
INTEGER*4	NODE(9,NE)	節点リスト
REAL*4	S(NE)	圧力
REAL*4	DNX(9,NE), DNY(9,NE), DNZ(9,NE)	形状関数の導関数
REAL*4	FX(NP), FY(NP), FZ(NP)	圧力勾配ベクトル

4角形要素について∇pの有限要素近似を計算するループ

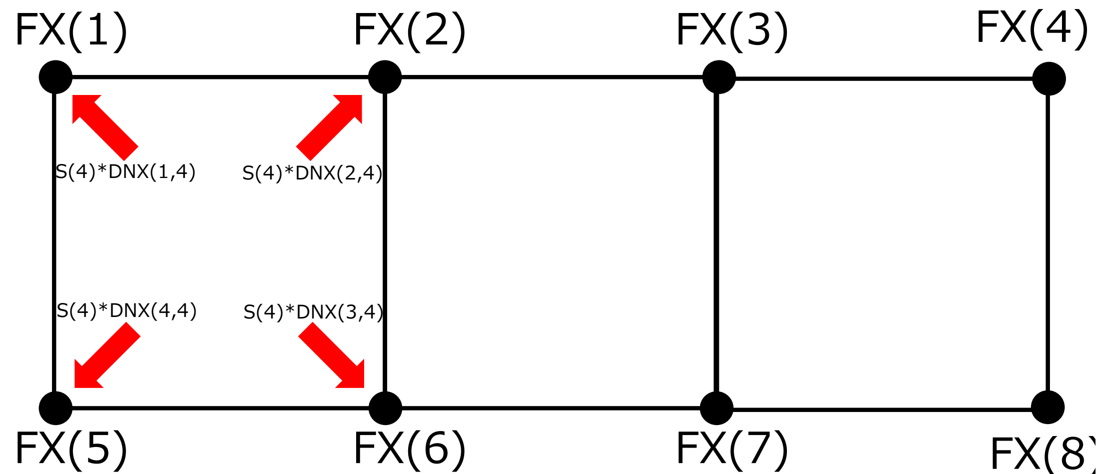
$$\nabla p = \frac{\partial p}{\partial x_i} \cong \sum_{j=1} \frac{\partial N_j}{\partial x_i} p_e$$



リカレンス（データの依存性）

このループは要素を順番に回って、要素の値と、要素の各頂点がもつ係数を掛け合わせた値を、頂点に対応する節点に足し込む

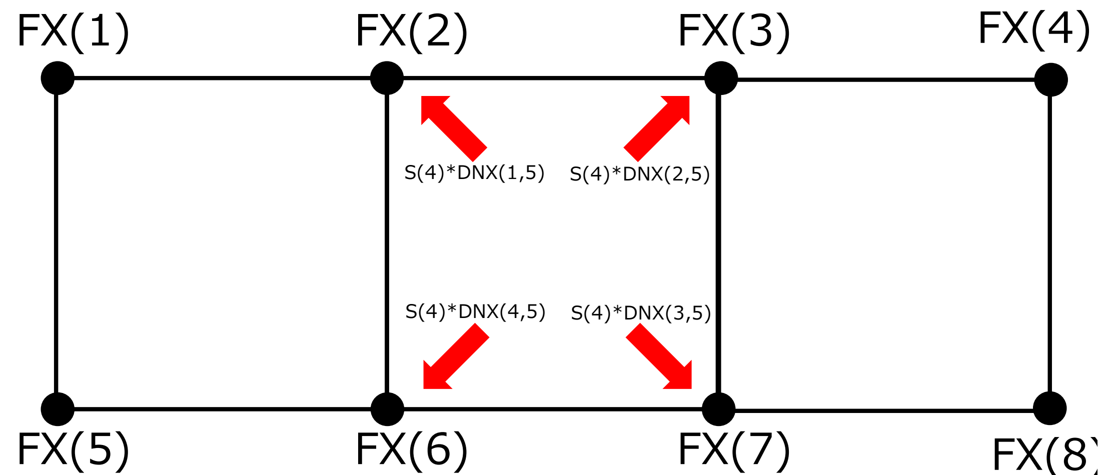
```
DO IE=1,NE
  IP1=NODE(1,IE)
  IP2=NODE(2,IE)
  IP3=NODE(3,IE)
  IP4=NODE(4,IE)

  SWRK=-S(IE)
  FX(IP1)=FX(IP1)+SWRK*DNX(1,IE)
  FX(IP2)=FX(IP2)+SWRK*DNX(2,IE)
  FX(IP3)=FX(IP3)+SWRK*DNX(3,IE)
  FX(IP4)=FX(IP4)+SWRK*DNX(4,IE)

  FY(IP1)=FY(IP1)+SWRK*DNY(1,IE)
  FY(IP2)=FY(IP2)+SWRK*DNY(2,IE)
  FY(IP3)=FY(IP3)+SWRK*DNY(3,IE)
  FY(IP4)=FY(IP4)+SWRK*DNY(4,IE)

  FZ(IP1)=FZ(IP1)+SWRK*DNZ(1,IE)
  FZ(IP2)=FZ(IP2)+SWRK*DNZ(2,IE)
  FZ(IP3)=FZ(IP3)+SWRK*DNZ(3,IE)
  FZ(IP4)=FZ(IP4)+SWRK*DNZ(4,IE)
ENDDO
```



リカレンス（データの依存性）

このループは要素を順番に回って、要素の値と、要素の各頂点がもつ係数を掛け合わせた値を、頂点に対応する節点に足し込む

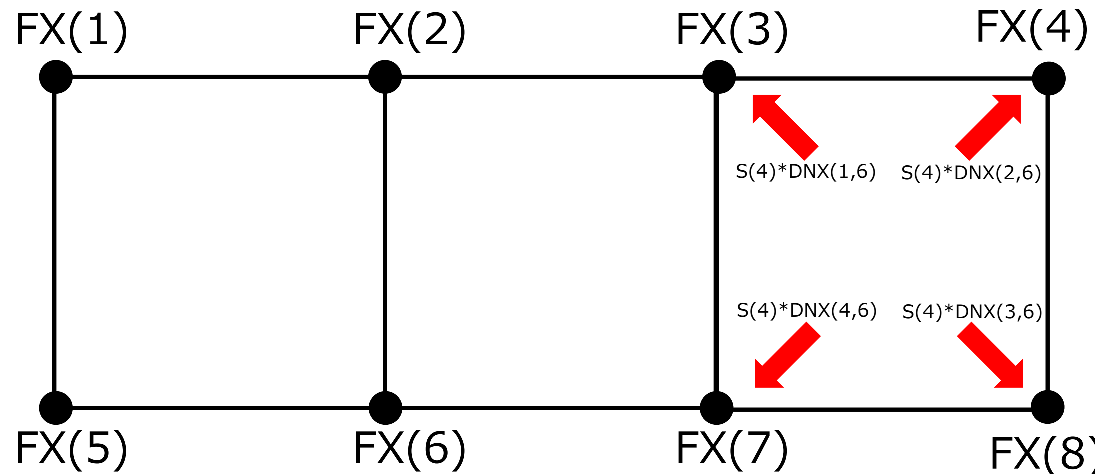
```
DO IE=1,NE
  IP1=NODE(1,IE)
  IP2=NODE(2,IE)
  IP3=NODE(3,IE)
  IP4=NODE(4,IE)

  SWRK=-S(IE)
  FX(IP1)=FX(IP1)+SWRK*DNX(1,IE)
  FX(IP2)=FX(IP2)+SWRK*DNX(2,IE)
  FX(IP3)=FX(IP3)+SWRK*DNX(3,IE)
  FX(IP4)=FX(IP4)+SWRK*DNX(4,IE)

  FY(IP1)=FY(IP1)+SWRK*DNY(1,IE)
  FY(IP2)=FY(IP2)+SWRK*DNY(2,IE)
  FY(IP3)=FY(IP3)+SWRK*DNY(3,IE)
  FY(IP4)=FY(IP4)+SWRK*DNY(4,IE)

  FZ(IP1)=FZ(IP1)+SWRK*DNZ(1,IE)
  FZ(IP2)=FZ(IP2)+SWRK*DNZ(2,IE)
  FZ(IP3)=FZ(IP3)+SWRK*DNZ(3,IE)
  FZ(IP4)=FZ(IP4)+SWRK*DNZ(4,IE)
ENDDO
```



リカレンス（データの依存性）

このループは要素を順番に回って、要素の値と、要素の各頂点がもつ係数を掛け合わせた値を、頂点に対応する節点に足し込む

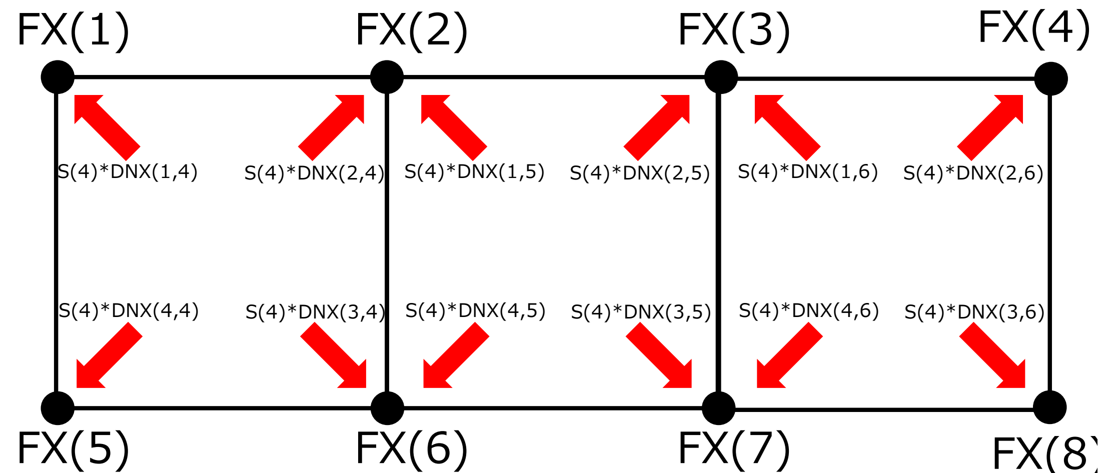
```
DO IE=1,NE
  IP1=NODE(1,IE)
  IP2=NODE(2,IE)
  IP3=NODE(3,IE)
  IP4=NODE(4,IE)

  SWRK=-S(IE)
  FX(IP1)=FX(IP1)+SWRK*DNX(1,IE)
  FX(IP2)=FX(IP2)+SWRK*DNX(2,IE)
  FX(IP3)=FX(IP3)+SWRK*DNX(3,IE)
  FX(IP4)=FX(IP4)+SWRK*DNX(4,IE)

  FY(IP1)=FY(IP1)+SWRK*DNY(1,IE)
  FY(IP2)=FY(IP2)+SWRK*DNY(2,IE)
  FY(IP3)=FY(IP3)+SWRK*DNY(3,IE)
  FY(IP4)=FY(IP4)+SWRK*DNY(4,IE)

  FZ(IP1)=FZ(IP1)+SWRK*DNZ(1,IE)
  FZ(IP2)=FZ(IP2)+SWRK*DNZ(2,IE)
  FZ(IP3)=FZ(IP3)+SWRK*DNZ(3,IE)
  FZ(IP4)=FZ(IP4)+SWRK*DNZ(4,IE)
ENDDO
```



リカレンス（データの依存性）

このループをスレッド並列すると、複数の要素を同時に処理することになるが
そうすると、リカレンスが生じるためできない

```
DO IE=1,NE                    ここをスレッド並列
  IP1=NODE(1,IE)
  IP2=NODE(2,IE)
  IP3=NODE(3,IE)
  IP4=NODE(4,IE)

  SWRK=-S(IE)
  FX(IP1)=FX(IP1)+SWRK*DNX(1,IE)
  FX(IP2)=FX(IP2)+SWRK*DNX(2,IE)
  FX(IP3)=FX(IP3)+SWRK*DNX(3,IE)
  FX(IP4)=FX(IP4)+SWRK*DNX(4,IE)

  FY(IP1)=FY(IP1)+SWRK*DNY(1,IE)
  FY(IP2)=FY(IP2)+SWRK*DNY(2,IE)
  FY(IP3)=FY(IP3)+SWRK*DNY(3,IE)
  FY(IP4)=FY(IP4)+SWRK*DNY(4,IE)

  FZ(IP1)=FZ(IP1)+SWRK*DNZ(1,IE)
  FZ(IP2)=FZ(IP2)+SWRK*DNZ(2,IE)
  FZ(IP3)=FZ(IP3)+SWRK*DNZ(3,IE)
  FZ(IP4)=FZ(IP4)+SWRK*DNZ(4,IE)
ENDDO
```

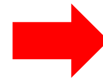


スレッド並列のためのリカレンス除去

waveZ

カラーリングを導入して、要素を互いに隣り合っていないカラー（グループ）に分ける

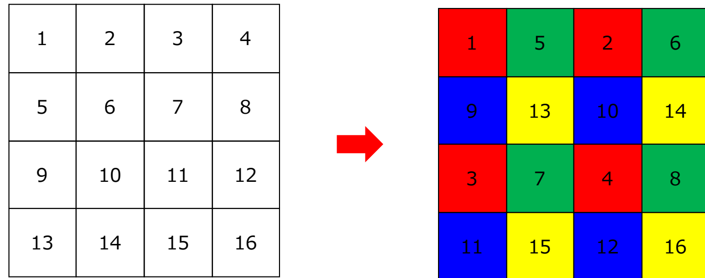
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

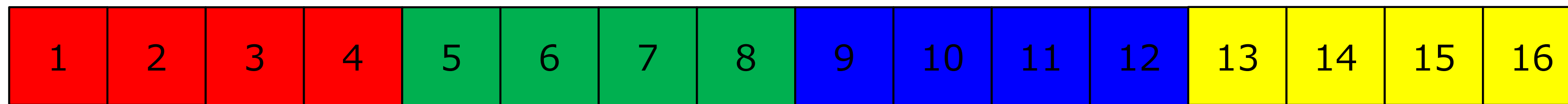


1	5	2	6
9	13	10	14
3	7	4	8
11	15	12	16

スレッド並列のためのリカレンス除去

メモリ上ではカラー毎に連続になるよう必要に応じて要素番号を付け替える

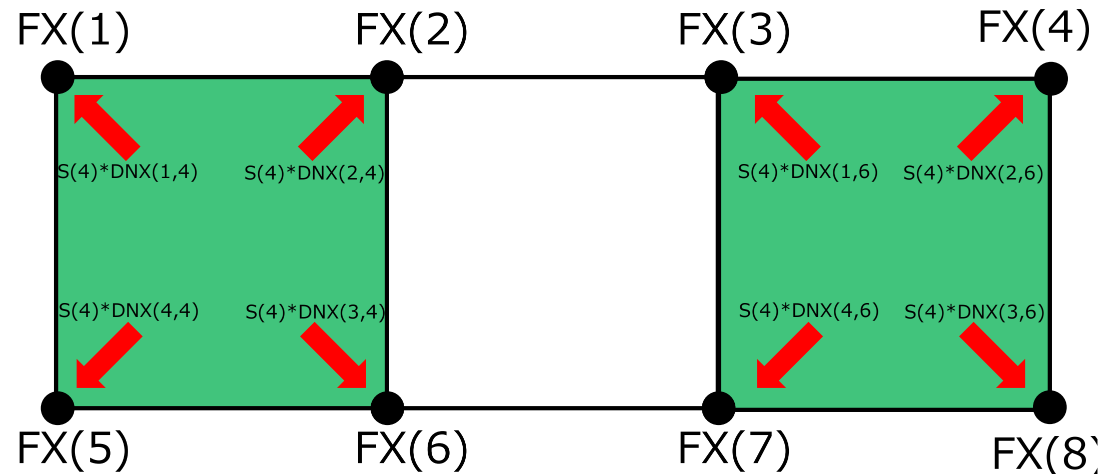




カラー1	カラー2	カラー3	カラー4
START(1)=1 END(1)=4	START(2)=5 END(2)=8	START(3)=9 END(3)=12	START(4)=13 END(4)=16

リカレンス（データの依存性）

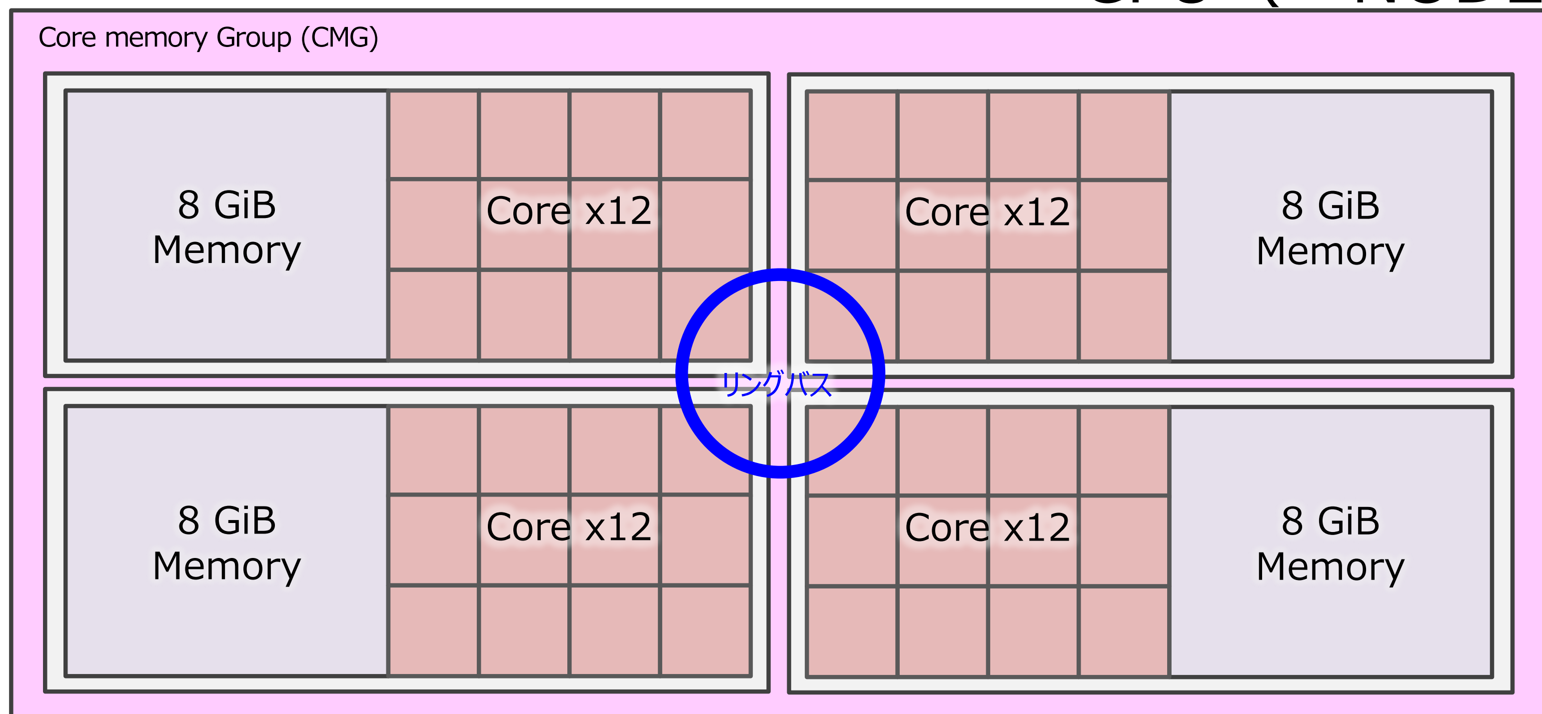
```
DO ICOLOR=1, NCOLOR
  ST=START(ICOLOR)
  ED=END(ICOLOR)
  DO IE=ST,ED                ここをスレッド並列
    IP1=NODE(1,IE)
    IP2=NODE(2,IE)
    IP3=NODE(3,IE)
    IP4=NODE(4,IE)

    SWRK=-S(IE)
    FX(IP1)=FX(IP1)+SWRK*DNX(1,IE)
    FX(IP2)=FX(IP2)+SWRK*DNX(2,IE)
    FX(IP3)=FX(IP3)+SWRK*DNX(3,IE)
    FX(IP4)=FX(IP4)+SWRK*DNX(4,IE)

    FY(IP1)=FY(IP1)+SWRK*DNY(1,IE)
    FY(IP2)=FY(IP2)+SWRK*DNY(2,IE)
    FY(IP3)=FY(IP3)+SWRK*DNY(3,IE)
    FY(IP4)=FY(IP4)+SWRK*DNY(4,IE)

    FZ(IP1)=FZ(IP1)+SWRK*DNZ(1,IE)
    FZ(IP2)=FZ(IP2)+SWRK*DNZ(2,IE)
    FZ(IP3)=FZ(IP3)+SWRK*DNZ(3,IE)
    FZ(IP4)=FZ(IP4)+SWRK*DNZ(4,IE)
  ENDDO
ENDDO
```



スレッド並列の実行形態

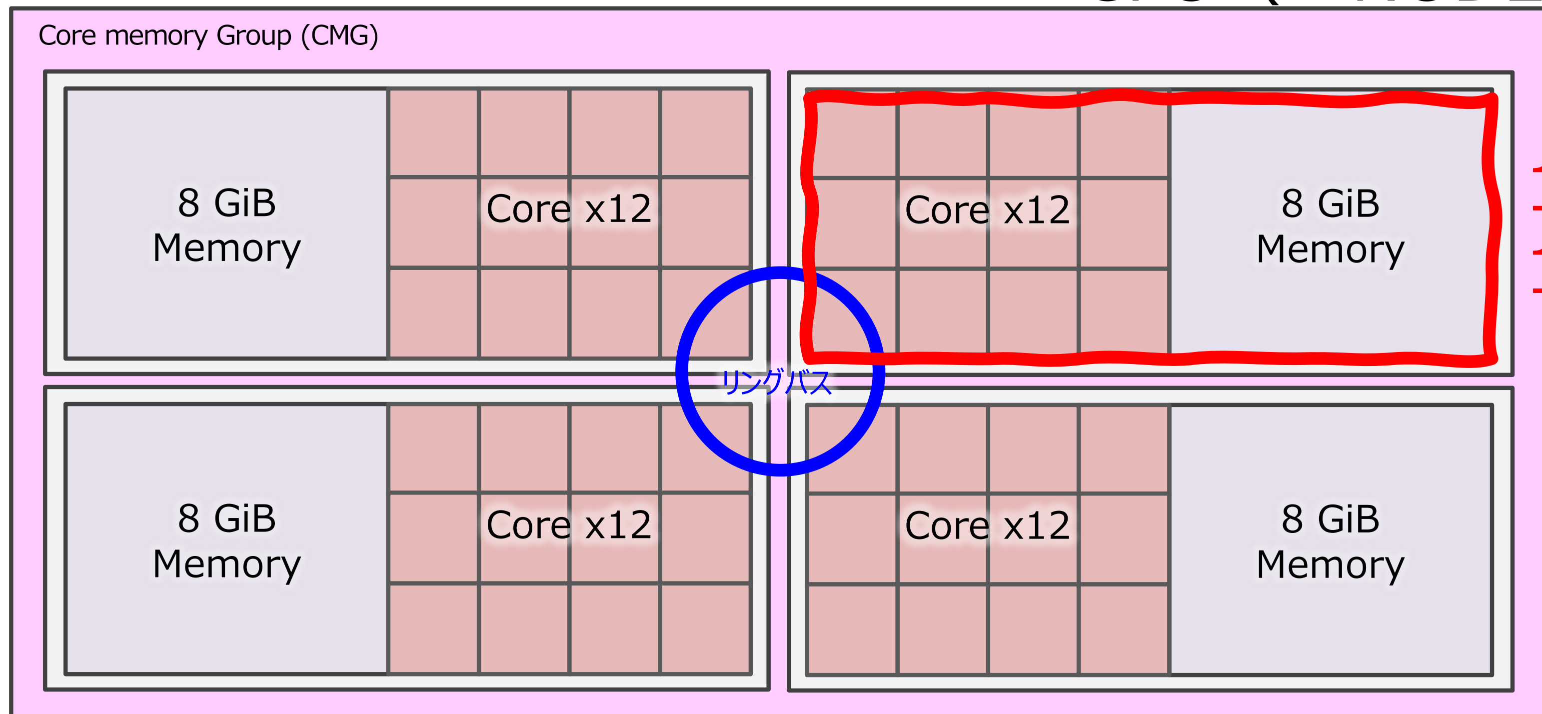
waveZ

富岳，あるいはFX1000/FX700



スレッド並列の実行形態

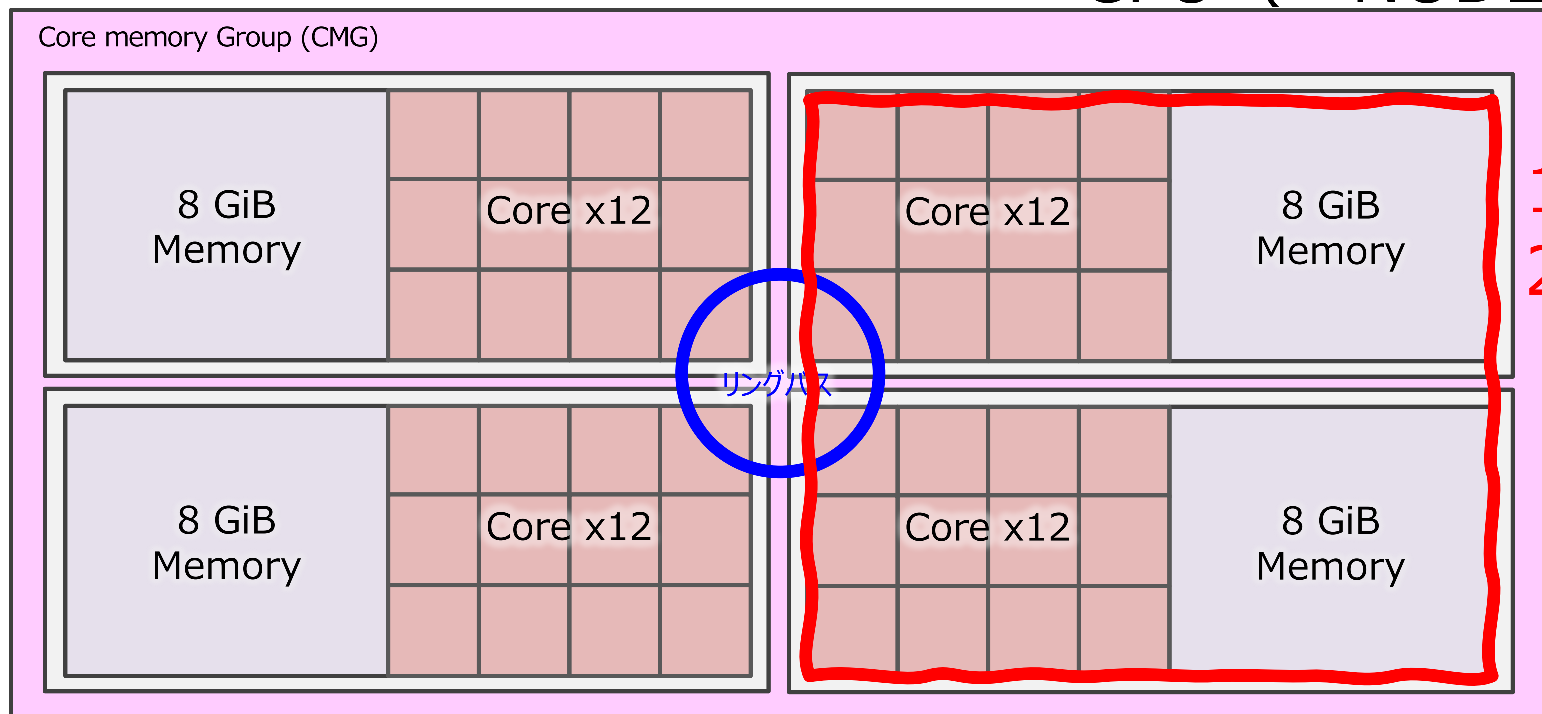
富岳，あるいはFX1000/FX700



スレッド並列の実行形態

富岳，あるいはFX1000/FX700

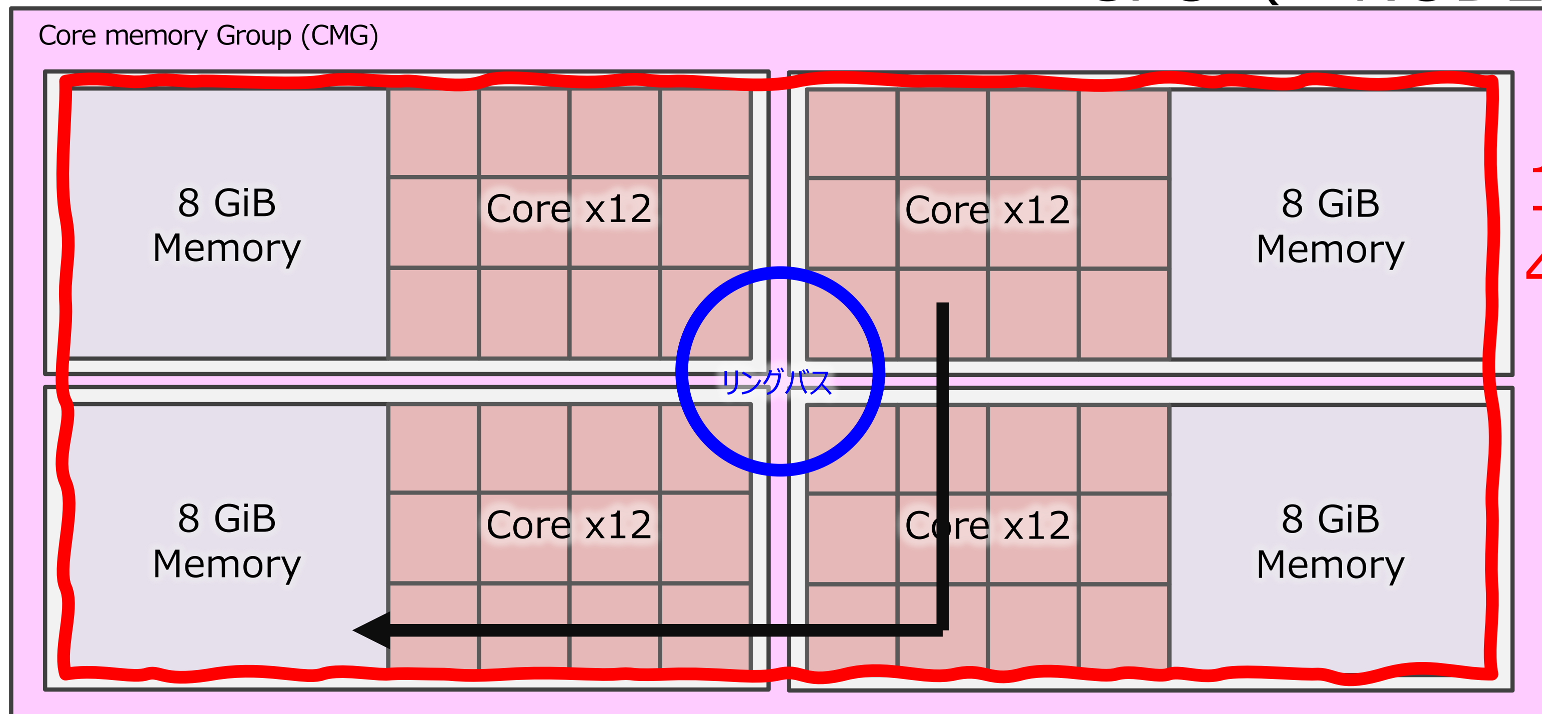
CPU（＝NODE)



スレッド並列の実行形態

富岳，あるいはFX1000/FX700

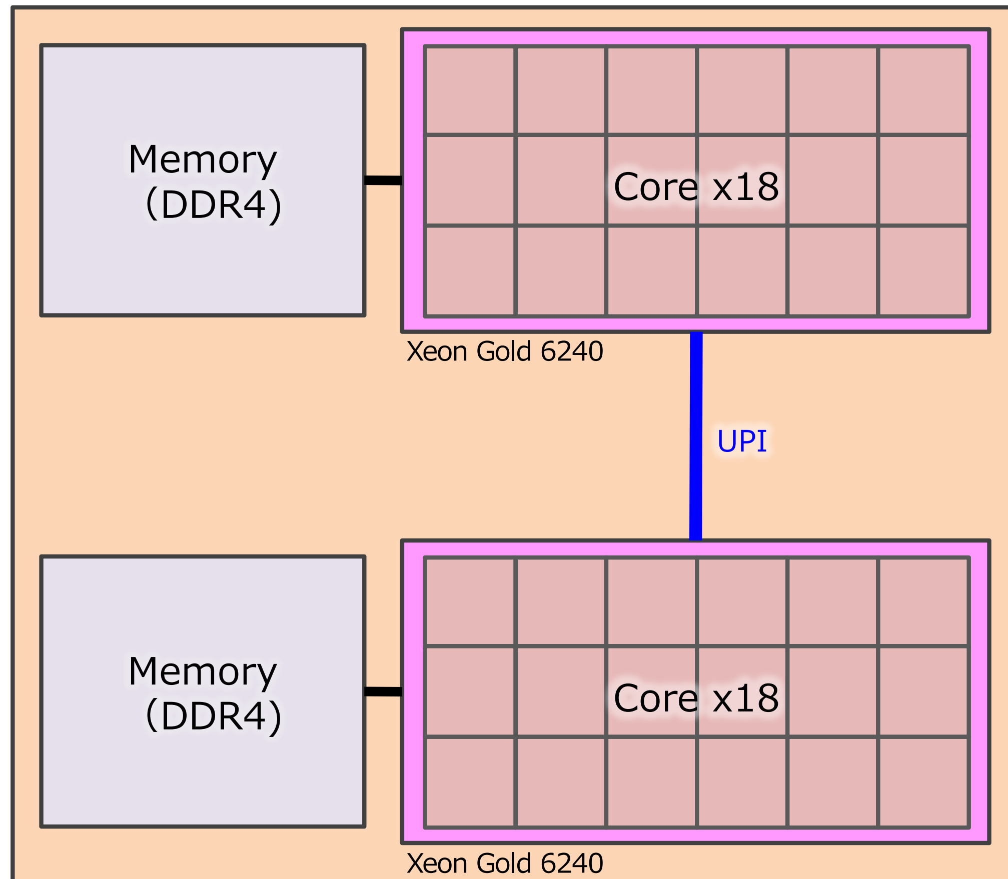
CPU（＝NODE)



遠いメモリを使ってしまうかも

スレッド並列の実行形態

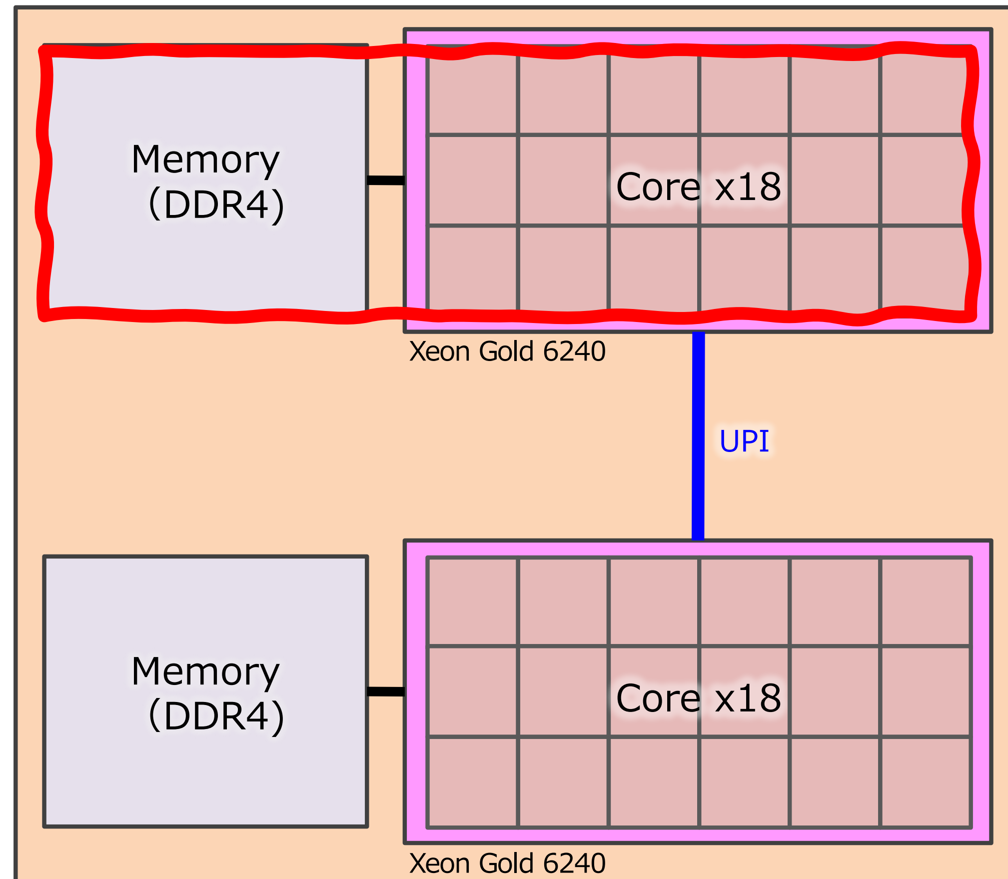
Intelの計算機



スレッド並列の実行形態

waveZ

Intelの計算機だと

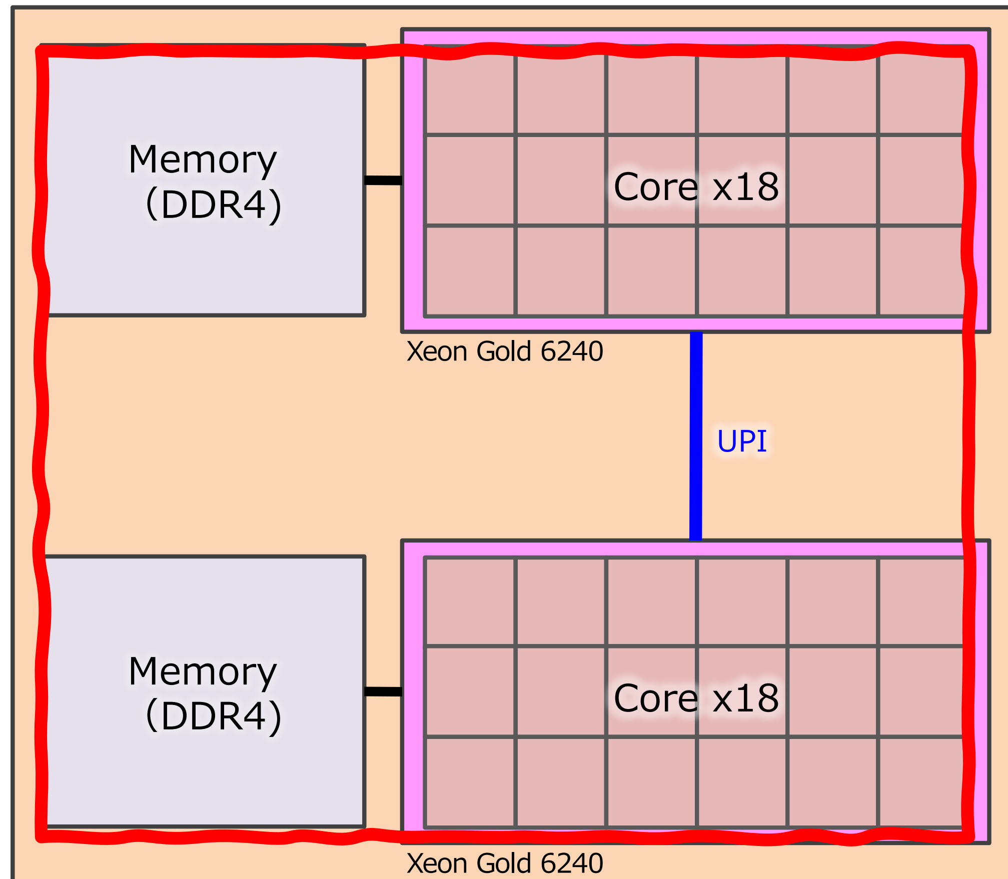


1プロセス
18スレッド

スレッド並列の実行形態

waveZ

Intelの計算機

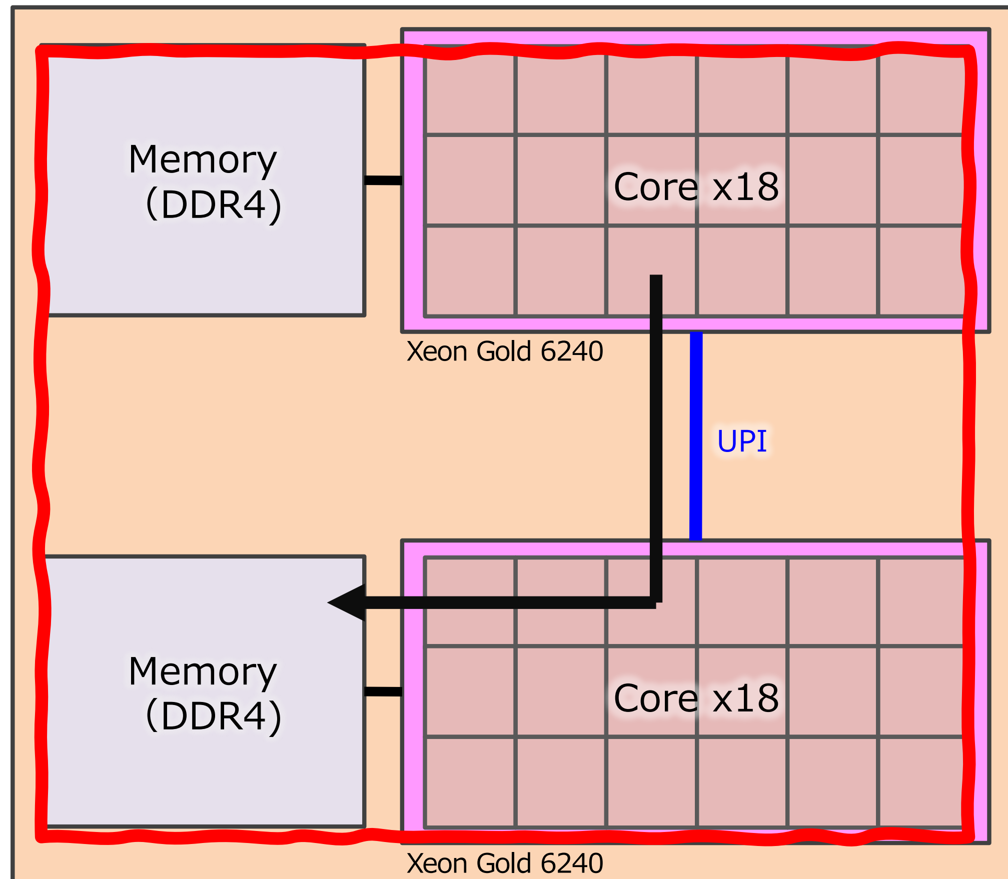


1プロセス
36スレッド

スレッド並列の実行形態

waveZ

Intelの計算機



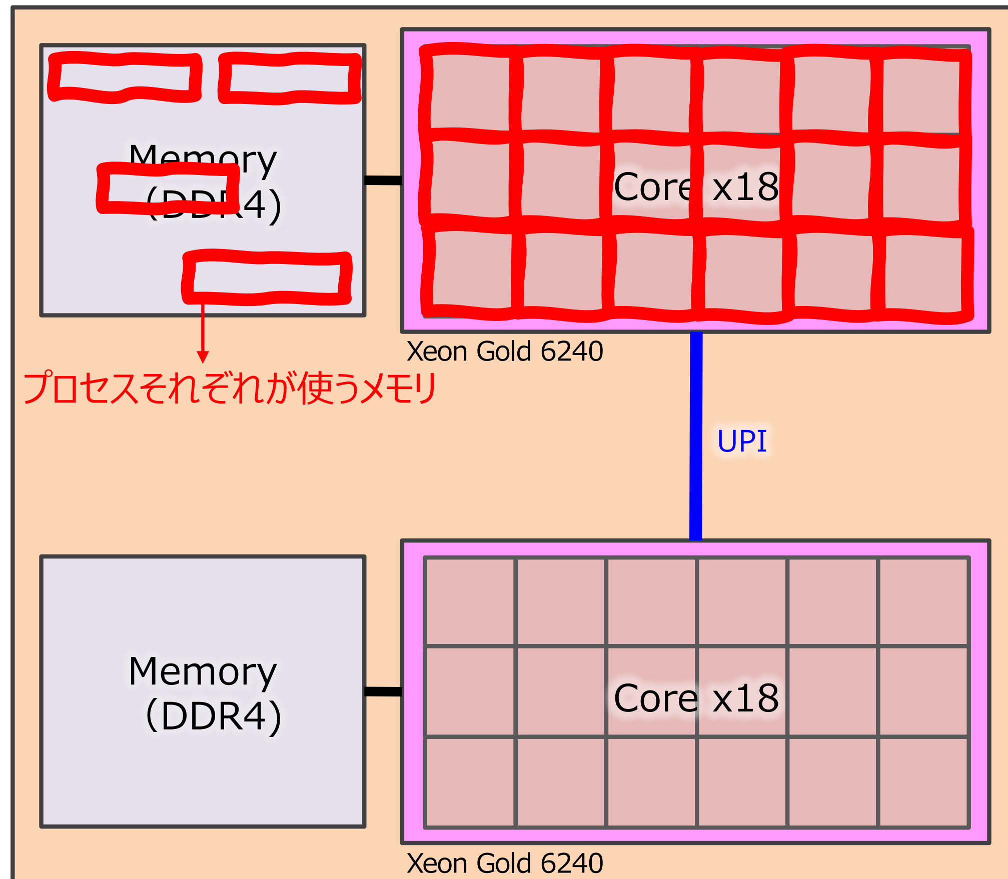
1プロセス
36スレッド

遠いメモリを使ってしまうかも

スレッド並列の実行形態

waveZ

Intelの計算機

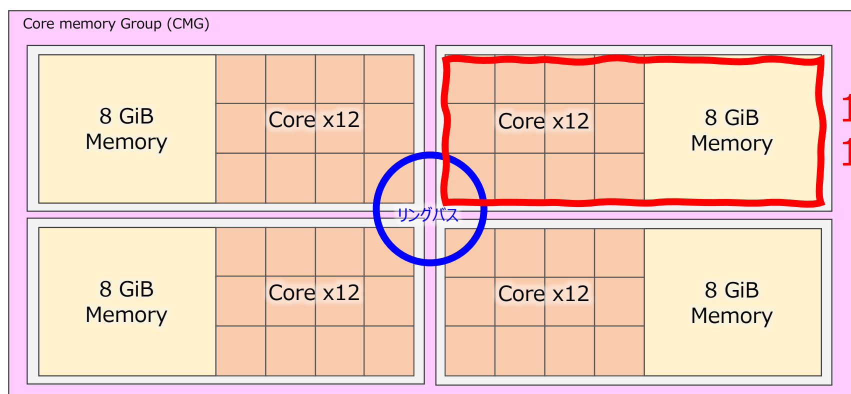


18プロセス
各1スレッド

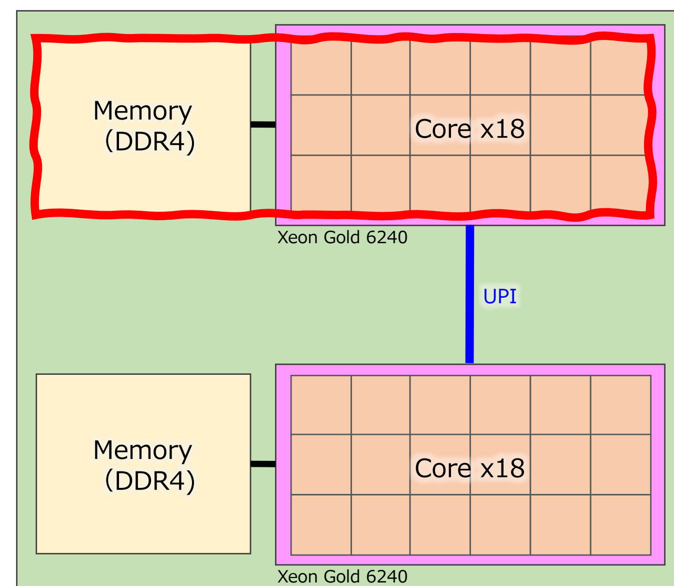
プロセスが増えると通信を管理するためのメモリが増える

推奨されるスレッド並列の実行形態

推奨の実行形態



1プロセス
12スレッド



1プロセス
18スレッド

プロファイラの使用　1/5

waveZ

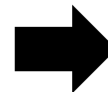
京/FX10/FX100でも使えたが、A64FX(富岳,FX1000/FX700)ではFAPPというプロファイラが使える

ソース中で、測定したい部分をサブルーチン・関数fapp_start/fapp_stopで挟み実行ファイルを、プロファイラの実行コマンドから実行．実行後できたデータを変換してエクセルで読む.

ソース例 (Fortran)

```
call fapp_start("range", 1, 0)
do i=0, N
  ...
  call fapp_start("range", 2, 1)
  do j=0, M
    ...
  enddo
  call fapp_stop("range", 2, 1)
enddo
call fapp_stop("range", 1, 0)
```

区間名　区間番号　レベル値



コンパイル例 (Fortran)

```
frtpx source.f -Nfjprof -o a.out
```

プロファイラ用のライブラリをリンクする指示

ただし指定しなくても、デフォルトで有効になっているので実際には気にしなくて良い

`-Nnofjprof`ならリンクしない

プロファイラの使用 2/5

waveZ

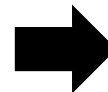
京/FX10/FX100でも使えたが、A64FX(富岳,FX1000/FX700)ではFAPPというプロファイラが使える

ソース中で、測定したい部分をサブルーチン・関数fapp_start/fapp_stopで挟み実行ファイルを、プロファイラの実行コマンドから実行． 実行後できたデータを変換してエクセルで読む.

ソース例 (C/C++)

```
fapp_start("range", 1, 0);
for(i=0; i<N+1; i++){
  …
  fapp_start("range", 2, 1);
  for(j=0; j<M+1; j++){
    …
  }
  fapp_stop("range", 2, 1);
}
fapp_stop("range", 1, 0);
```

区間名　区間番号　レベル値



コンパイル例 (C/C++)

```
fccpx source.cpp -Nfjprof -o a.out
```

プロファイラ用のライブラリをリンクする指示
ただし指定しなくても、デフォルトで有効になっているので実際には気にしなくて良い
-Nnofjprofならリンクしない

※clangモードの場合は-ffj-fjprof あるいは
-fj-no-fjprof

プロファイラの使用 3/5

waveZ

実行例(プロファイラのコマンドfappを介してa.outを呼ぶ)

```
for I in `seq 1 17` ←1, 5, 10, 17を指定
                     数が多いほど詳細な情報が取れる
do
  fapp -C —Hevent=pa${I} -d ./pa${I} ¥
       -Icpupa —L0 a.out             本当は1行なので注意
enddo
```

-C 測定時のおまじない
一つのコマンドで複数の役割があるため)

-Hevent=pa1～17
測定するデータの種類を指定

-d
結果を出力するディレクトリ指定(-Heventの指定と合わせる)

-Icpupa
CPU内で起こる様々なイベントを測定する

-L 0
測定のレベル指定
ソース中の、fapp_start/fapp_stop関数・サブルーチンで指定したレベル値が、ここでの指定値以下 (≦) のものを測定する

レベル指定 –L の例

```
call fapp_start("range", 1, 0)
do i=0, N
  ...
  call fapp_stop("range", 2, 1)
  do j=0, M
    ...
  enddo
  call fapp_start("range", 2, 1)
enddo
call fapp_stop("range", 1, 0)
```

-LNだとソース中のレベル指定の値が ≦ N である区間を測定する

-L0だと赤字の区間だけを測定
-L1だと赤字と青字の区間を測定

プロファイラの使用 4/5

waveZ

実行後は pa1~17のディレクトリができるので変換コマンドで読める形に変換する

```
for I in `seq 1 17`
do
  fapp -A -d ./pa${I} -Icpupa -tcsv -o pa${I}.csv
enddo
```

-A 測定時のおまじない
一つのコマンドで複数の役割があるため)

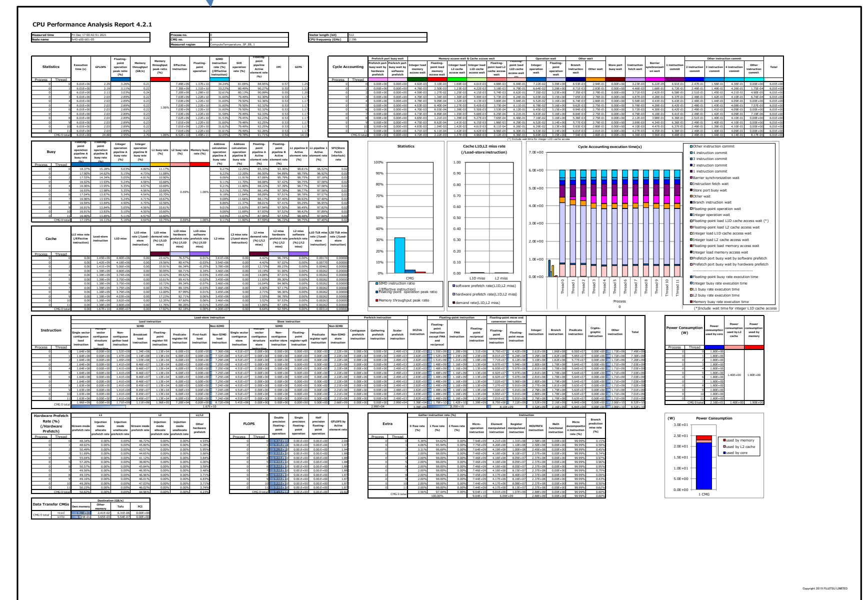
-d
結果を入力するディレクトリ指定

-Icpupa
CPU内で起こる様々なイベントの測定結果を変換

-tcsv
変換結果をcsvで出力

-o
出力するcsvファイル名

作成されたcsvと、システム上にある cpu_pa_report.xlsmをPCにコピーし、エクセルで開くと測定区間の詳細なデータが出る

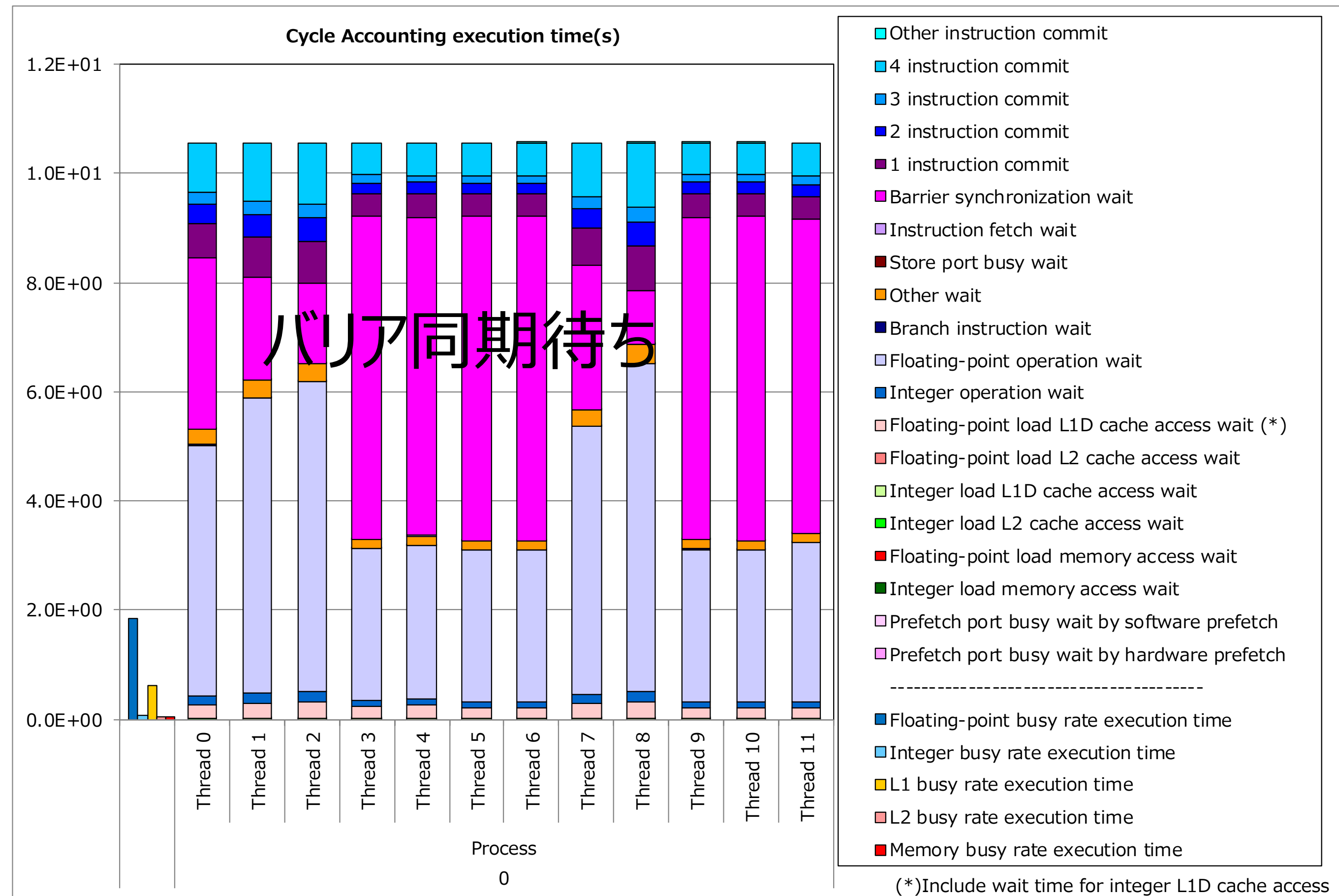


プロファイラの使用 5/5

スレッド並列化の手助けとなるプロファイラ情報

富岳・FX1000/700では
CPU性能解析レポートが
利用可能

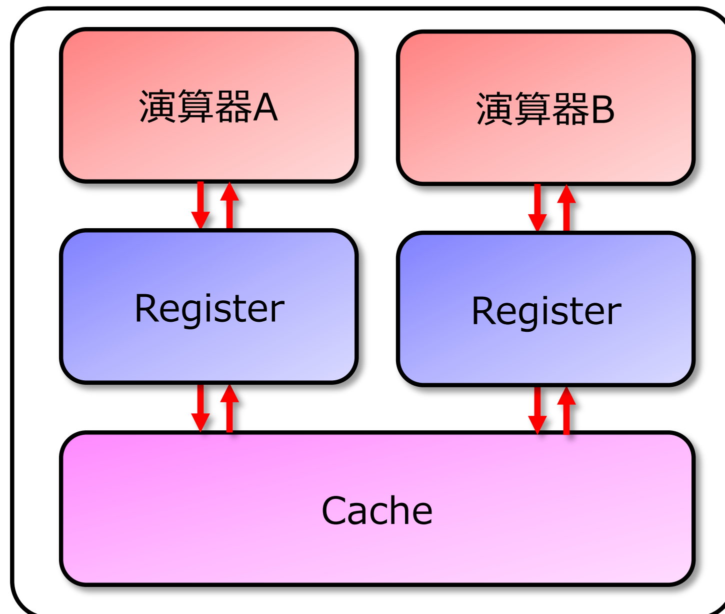
Effective instruction	Floating-point operation
1.24E+10	6.38E+10
1.44E+10	7.55E+10
1.50E+10	7.93E+10
7.89E+09	3.81E+10
7.89E+09	3.81E+10
7.89E+09	3.81E+10
7.89E+09	3.81E+10
1.31E+10	6.80E+10
1.58E+10	8.38E+10
7.86E+09	3.79E+10
7.86E+09	3.79E+10
8.19E+09	3.98E+10
1.26E+11	6.38E+11



コア内の並列性
演算命令とスケジューリング

コア内の並列性

waveZ

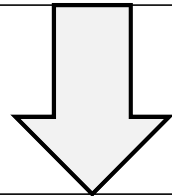


- 1スレッドが割り当てられたコアでは複数の演算器があり，プロセス並列，スレッド並列と同様に，なるべく全てが遊ぶことなく働いてほしい
- 演算・命令レベルでの並列性を考えないといけない
- ここでは以下を説明
  - SIMD
  - ソフトウェアパイプラン

演算・命令レベルの並列性

waveZ

```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```



少し分解して考える

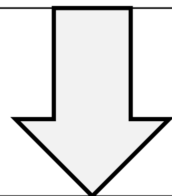
```
do i=1, n
   b(i)のロード
   c(i)のロード
   b(i)とc(i)を足す
   結果をa(i)にストアする
enddo
```

このループの動きは

演算レベルの並列性

waveZ

```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```



少し分解して考える

```
do i=1, n
   b(i)のロード
   c(i)のロード
   b(i)とc(i)を足す
   結果をa(i)にストアする
enddo
```

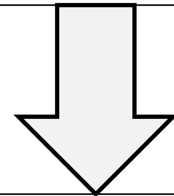
このループの動きは

b(1)のロード

演算レベルの並列性

waveZ

```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```



少し分解して考える

```
do i=1, n
   b(i)のロード
   c(i)のロード
   b(i)とc(i)を足す
   結果をa(i)にストアする
enddo
```

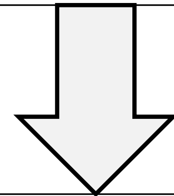
このループの動きは

b(1)のロード　c(1)のロード

演算レベルの並列性

waveZ

```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```



少し分解して考える

```
do i=1, n
   b(i)のロード
   c(i)のロード
   b(i)とc(i)を足す
   結果をa(i)にストアする
enddo
```

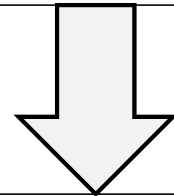
このループの動きは

b(1)のロード	c(1)のロード	加算

演算レベルの並列性

waveZ

```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```



少し分解して考える

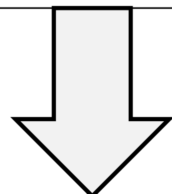
```
do i=1, n
   b(i)のロード
   c(i)のロード
   b(i)とc(i)を足す
   結果をa(i)にストアする
enddo
```

このループの動きは

b(1)のロード	c(1)のロード	加算	a(1)へのストア

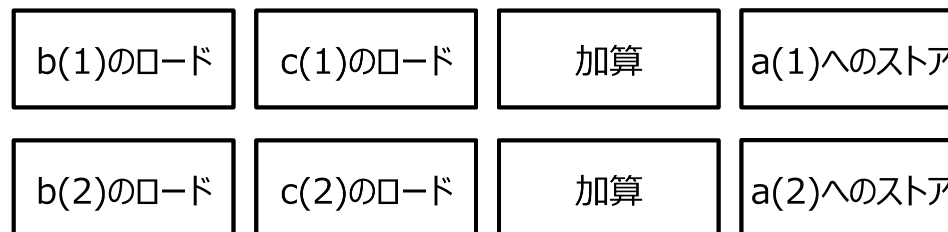
演算レベルの並列性

```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```




```
do i=1, n
   b(i)のロード
   c(i)のロード
   b(i)とc(i)を足す
   結果をa(i)にストアする
enddo
```

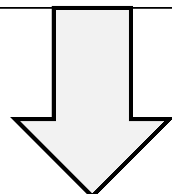
このループの動きは



演算レベルの並列性

waveZ

```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```



少し分解して考える

```
do i=1, n
   b(i)のロード
   c(i)のロード
   b(i)とc(i)を足す
   結果をa(i)にストアする
enddo
```

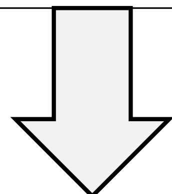
このループの動きは

b(1)のロード	c(1)のロード	加算	a(1)へのストア
b(2)のロード	c(2)のロード	加算	a(2)へのストア
b(3)のロード	c(3)のロード	加算	a(3)へのストア

演算レベルの並列性

wavez

```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```



少し分解して考える

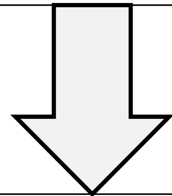
```
do i=1, n
   b(i)のロード
   c(i)のロード
   b(i)とc(i)を足す
   結果をa(i)にストアする
enddo
```

このループの動きは

b(1)のロード	c(1)のロード	加算	a(1)へのストア
b(2)のロード	c(2)のロード	加算	a(2)へのストア
b(3)のロード	c(3)のロード	加算	a(3)へのストア
b(4)のロード	c(4)のロード	加算	a(4)へのストア

演算レベルの並列性

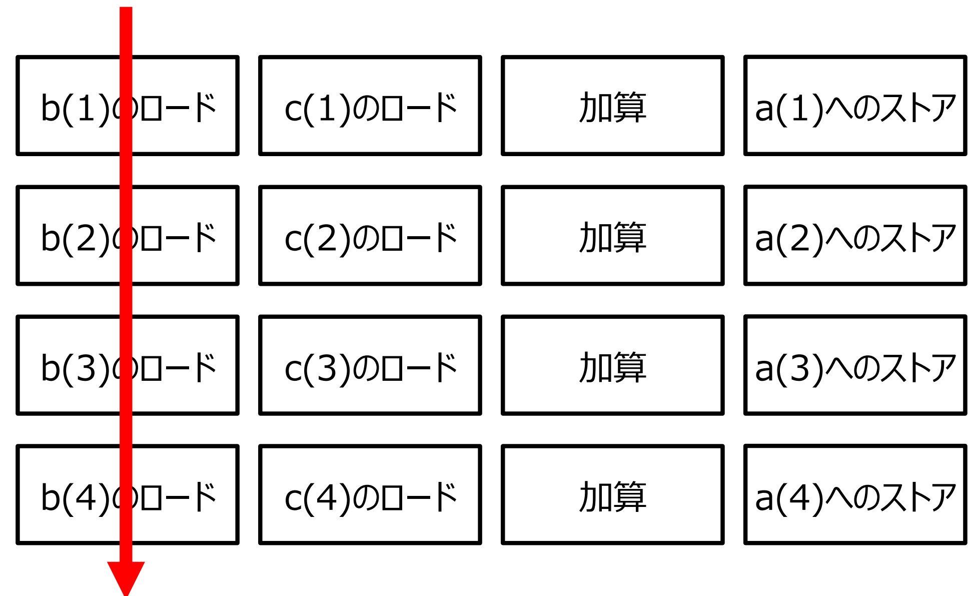
```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```



少し分解して考える

```
do i=1, n
   b(i)のロード
   c(i)のロード
   b(i)とc(i)を足す
   結果をa(i)にストアする
enddo
```

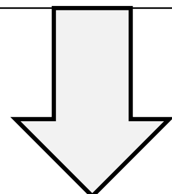
このループの動きは



演算レベルの並列性

waveZ

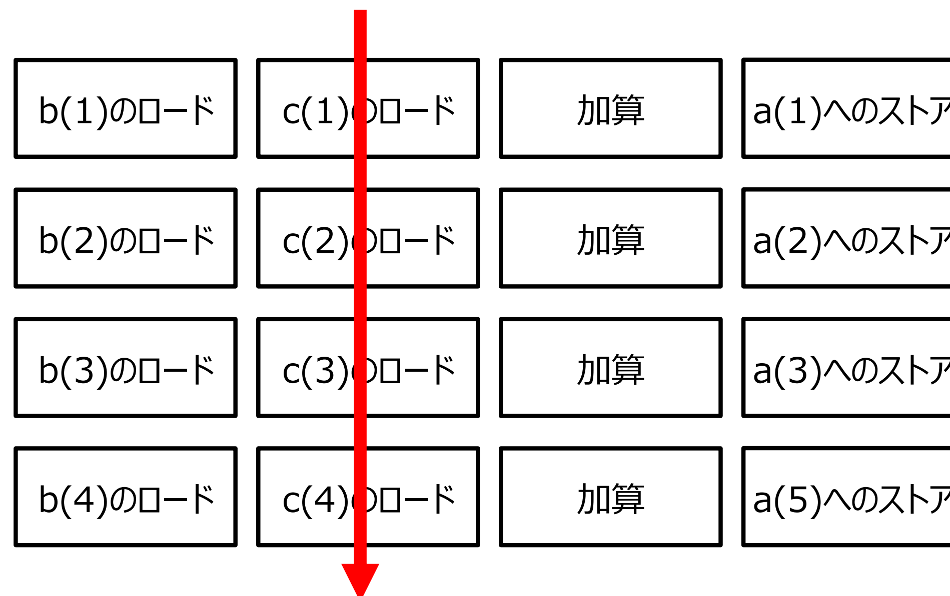
```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```



少し分解して考える

```
do i=1, n
   b(i)のロード
   c(i)のロード
   b(i)とc(i)を足す
   結果をa(i)にストアする
enddo
```

このループの動きは

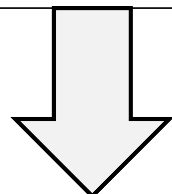


同じ処理

演算レベルの並列性

waveZ

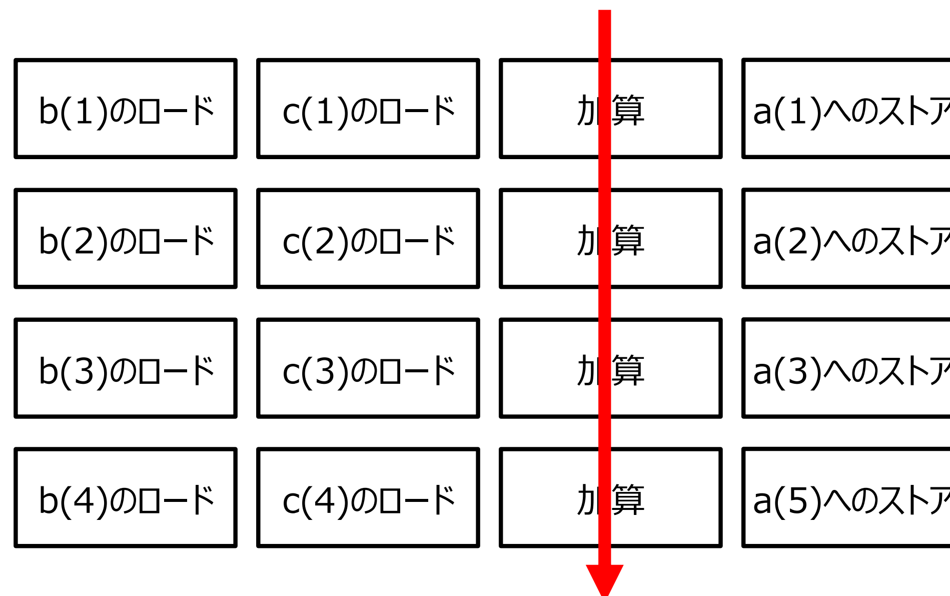
```
do i=1, n
    a(i) = b(i) + c(i)
enddo
```



少し分解して考える

```
do i=1, n
    b(i)のロード
    c(i)のロード
    b(i)とc(i)を足す
    結果をa(i)にストアする
enddo
```

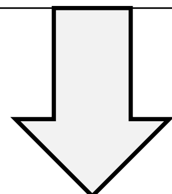
このループの動きは



演算レベルの並列性

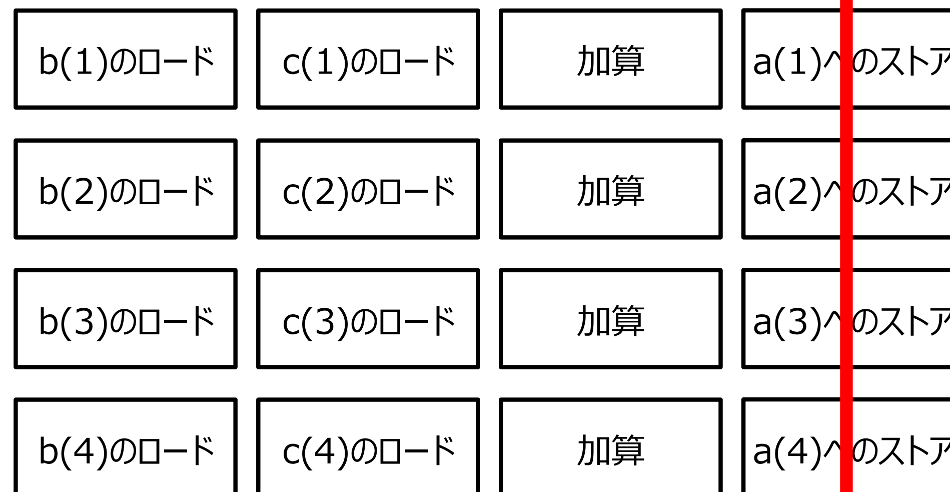
waveZ

```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```




```
do i=1, n
   b(i)のロード
   c(i)のロード
   b(i)とc(i)を足す
   結果をa(i)にストアする
enddo
```

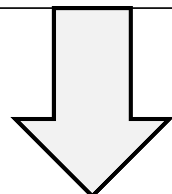
このループの動きは



演算レベルの並列性

waveZ

```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```



少し分解して考える

```
do i=1, n
   b(i)のロード
   c(i)のロード
   b(i)とc(i)を足す
   結果をa(i)にストアする
enddo
```

このループの動きは

b(1)のロード	c(1)のロード	加算	a(1)へのストア
b(2)のロード	c(2)のロード	加算	a(2)へのストア
b(3)のロード	c(3)のロード	加算	a(3)へのストア
b(4)のロード	c(4)のロード	加算	a(4)へのストア

ループの違うｉに対して、同じ処理を繰り返している

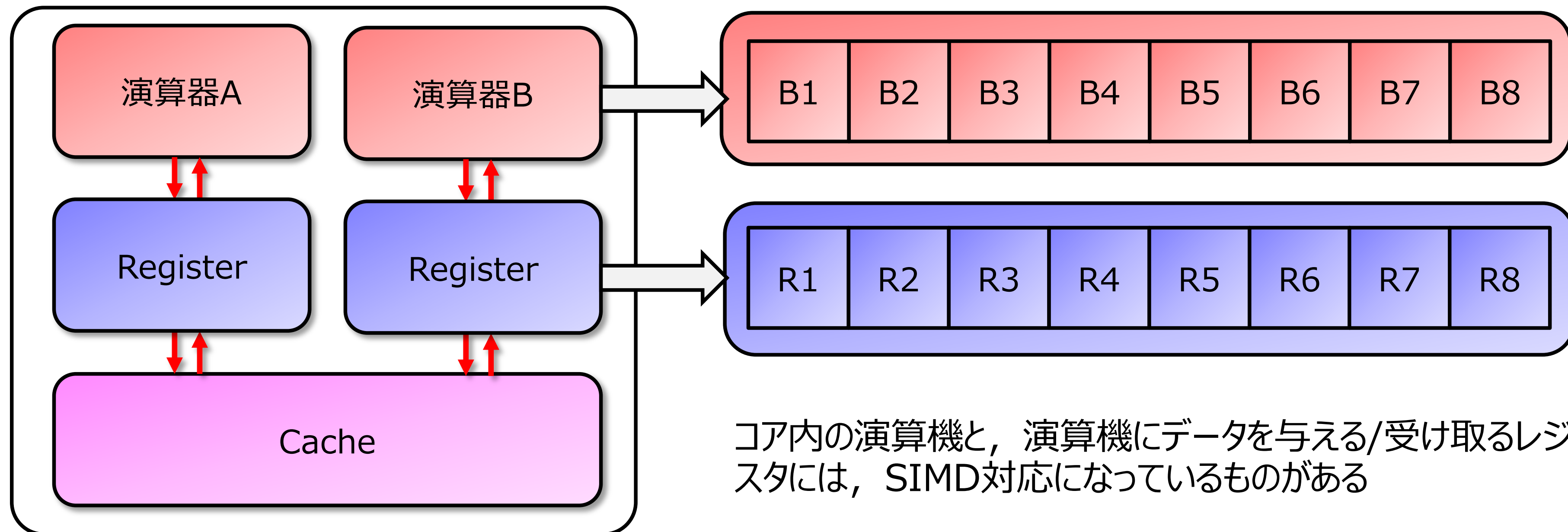
SIMD

waveZ

こういった、同じ処理を複数のデータにやることを

SIMD(single instruction multiple data)

と呼ぶ.



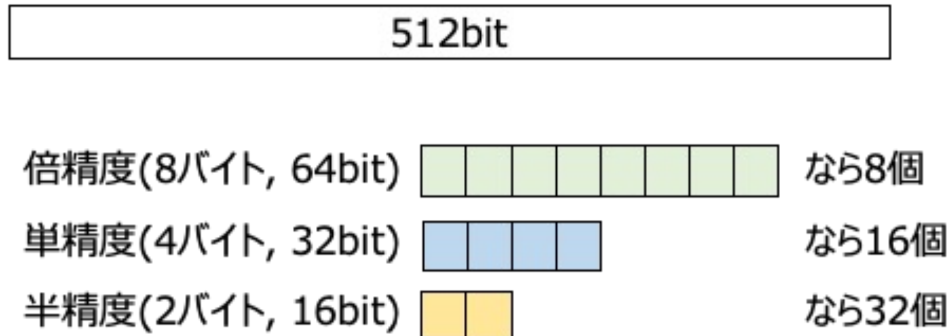
コア内の演算機と，演算機にデータを与える/受け取るレジスタには，SIMD対応になっているものがある

SIMDとは

同じ処理を同時に幾つのデータに対して行えるかという数を　SIMD幅（あるいはベクトル長）という

最近の計算機では、扱うデータによって異るSIMD幅を取れるものがある

A64FX(富岳/FX1000/FX700)の例



Intel)
SSE、AVX, AVX-512, VNNI

AMD)
SSE, AVX

Arm/富士通)
NEON, Scalable Vector Extension
現在は512bit(将来、例えば1024bitの機種ができても同じプログラムで1024bitにScalableに対応)

SIMD

waveZ

1)オリジナル

```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```

2)SIMD化ソースのイメージ

```
do i=1, n, 4
   a(i+0) = b(i+0) + c(i+0)
   a(i+1) = b(i+1) + c(i+1)
   a(i+2) = b(i+2) + c(i+2)
   a(i+3) = b(i+3) + c(i+3)
enddo
```

※コンパイラが自動でやる

3)SIMD化動作イメージ

```
do i=1, n, 4
   b(i+{0,1,2,3})のロード
   c(i+{0,1,2,3})のロード
   加算4つ
   結果をa(i+{0,1,2,3})にストア
enddo
```

SIMDとリカレンス

1)オリジナル

```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```

2)SIMD化ソースのイメージ

```
do i=1, n, 4
   a(i+0) = b(i+0) + c(i+0)
   a(i+1) = b(i+1) + c(i+1)
   a(i+2) = b(i+2) + c(i+2)
   a(i+3) = b(i+3) + c(i+3)
enddo
```

SIMDとリカレンス

waveZ

1)オリジナル

```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```

2)SIMD化ソースのイメージ

```
do i=1, n, 4
   a(i+0) = b(i+0) + c(i+0)
   a(i+1) = b(i+1) + c(i+1)
   a(i+2) = b(i+2) + c(i+2)
   a(i+3) = b(i+3) + c(i+3)
enddo
```

命令1

b(i+0), b(i+1),
b(i+2), b(i+3)
のロード

SIMDとリカレンス

waveZ

1)オリジナル

```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```

2)SIMD化ソースのイメージ

```
do i=1, n, 4
   a(i+0) = b(i+0) + c(i+0)
   a(i+1) = b(i+1) + c(i+1)
   a(i+2) = b(i+2) + c(i+2)
   a(i+3) = b(i+3) + c(i+3)
enddo
```

命令1: b(i+0), b(i+1), b(i+2), b(i+3) のロード

命令2: c(i+0), c(i+1), c(i+2), c(i+3) のロード

SIMDとリカレンス

waveZ

1)オリジナル

```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```

2)SIMD化ソースのイメージ

```
do i=1, n, 4
   a(i+0) = b(i+0) + c(i+0)
   a(i+1) = b(i+1) + c(i+1)
   a(i+2) = b(i+2) + c(i+2)
   a(i+3) = b(i+3) + c(i+3)
enddo
```

命令1 b(i+0), b(i+1), b(i+2), b(i+3) のロード

命令2 c(i+0), c(i+1), c(i+2), c(i+3) のロード

命令3 加算4つ

SIMDとリカレンス

waveZ

1)オリジナル

```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```

2)SIMD化ソースのイメージ

```
do i=1, n, 4
   a(i+0) = b(i+0) + c(i+0)
   a(i+1) = b(i+1) + c(i+1)
   a(i+2) = b(i+2) + c(i+2)
   a(i+3) = b(i+3) + c(i+3)
enddo
```

命令1	b(i+0), b(i+1), b(i+2), b(i+3) のロード
命令2	c(i+0), c(i+1), c(i+2), c(i+3) のロード
命令3	加算4つ
命令4	a(i+0), a(i+1), a(i+2), a(i+3) のストア

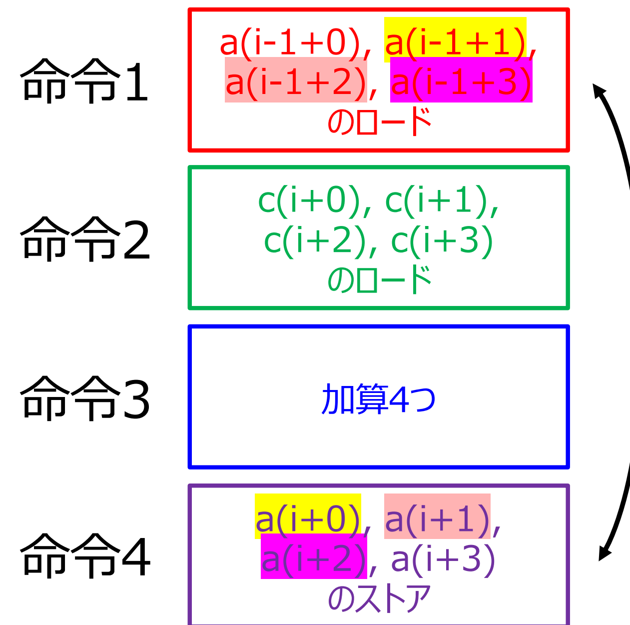
SIMDとリカレンス

1)オリジナル

```
do i=1, n
   a(i) = a(i-1) + c(i)
enddo
```

2)SIMD化ソースのイメージ

```
do i=1, n, 4
   a(i+0) = a(i-1+0) + c(i+0)
   a(i+1) = a(i-1+1) + c(i+1)
   a(i+2) = a(i-1+2) + c(i+2)
   a(i+3) = a(i-1+3) + c(i+3)
enddo
```



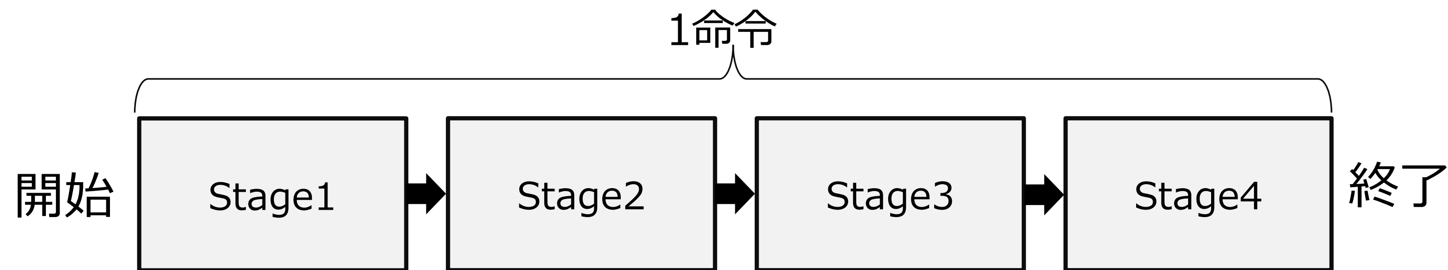
値が決まる前にロードしてしまい
結果が合わなくなるので、SIMDできない

ソフトウェアパイプライン

waveZ

現代のコンピュータは「一つの命令を1クロックサイクルで実行できる」と思われている.
これは、正しくもあるし、間違いでもある

コンピュータのCPUの中では、一つの命令はいくつかのステージに分解されて実行される



なので、実際には一つの命令が、「始まってから」に着目すると1クロックでは実行できない

FMA演算命令の平均的レイテンシ

富岳	京	FX100	Intel
9	6	6	5

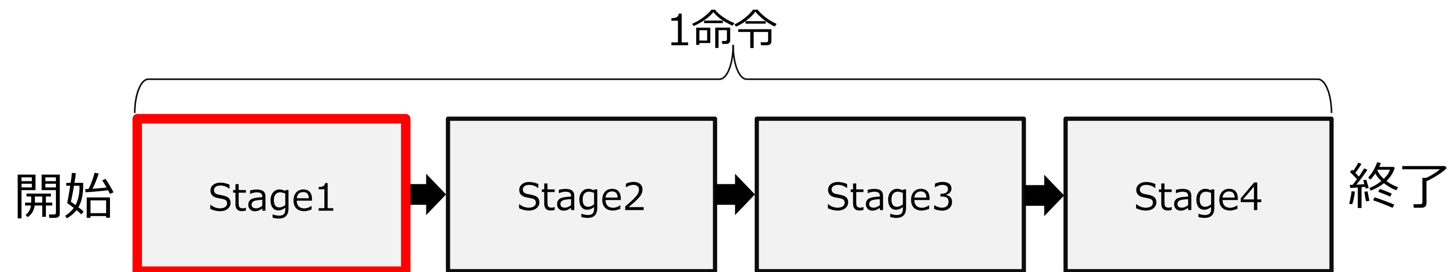
単位 クロックサイクル

ソフトウェアパイプライン

waveZ

現代のコンピュータは「一つの命令を1クロックサイクルで実行できる」と思われている.
これは、正しくもあるし、間違いでもある

コンピュータのCPUの中では、一つの命令はいくつかのステージに分解されて実行される



なので、実際には一つの命令が、「始まってから」に着目すると1クロックでは実行できない

FMA演算命令の平均的レイテンシ

富岳	京	FX100	Intel
9	6	6	5

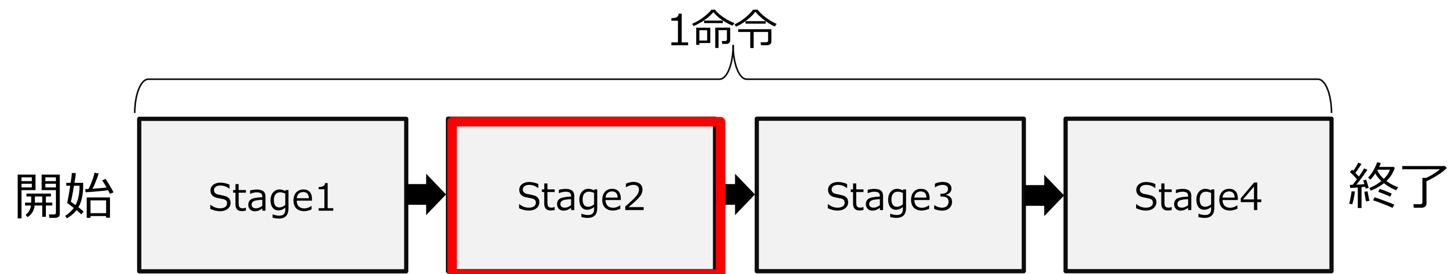
単位 クロックサイクル

ソフトウェアパイプライン

waveZ

現代のコンピュータは「一つの命令を1クロックサイクルで実行できる」と思われている.
これは、正しくもあるし、間違いでもある

コンピュータのCPUの中では、一つの命令はいくつかのステージに分解されて実行される



なので、実際には一つの命令が、「始まってから」に着目すると1クロックでは実行できない

FMA演算命令の平均的レイテンシ

富岳	京	FX100	Intel
9	6	6	5

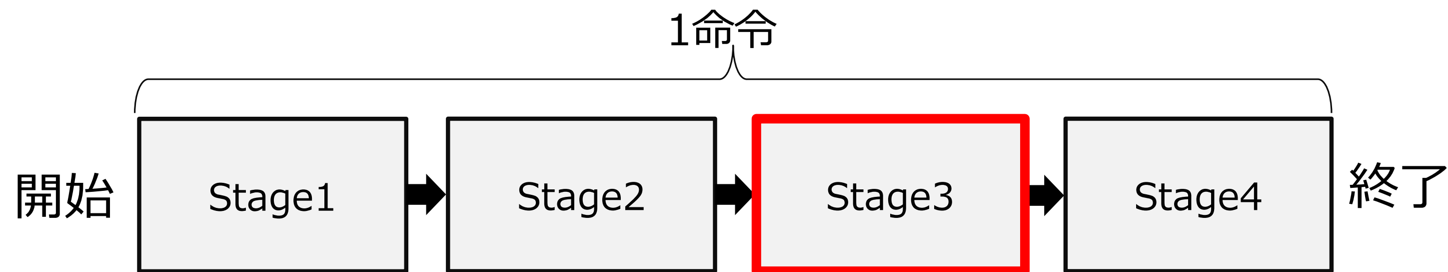
単位 クロックサイクル

ソフトウェアパイプライン

waveZ

現代のコンピュータは「一つの命令を1クロックサイクルで実行できる」と思われている.
これは、正しくもあるし、間違いでもある

コンピュータのCPUの中では、一つの命令はいくつかのステージに分解されて実行される



なので、実際には一つの命令が、「始まってから」に着目すると1クロックでは実行できない

FMA演算命令の平均的レイテンシ

富岳	京	FX100	Intel
9	6	6	5

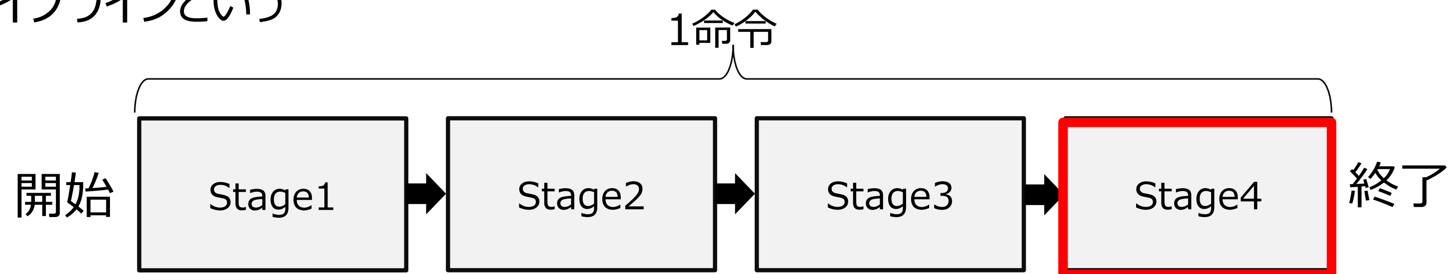
単位 クロックサイクル

ソフトウェアパイプライン

waveZ

現代のコンピュータは「一つの命令を1クロックサイクルで実行できる」と思われている.
これは、正しくもあるし、間違いでもある

コンピュータのCPUの中では、一つの命令はいくつかのステージに分解されて実行される
これをパイプラインという



なので、実際には一つの命令が、「始まってから」に着目すると1クロックでは実行できない

FMA演算命令の平均的レイテンシ

富岳	京	FX100	Intel
9	6	6	5

単位 クロックサイクル

ソフトウェアパイプライン

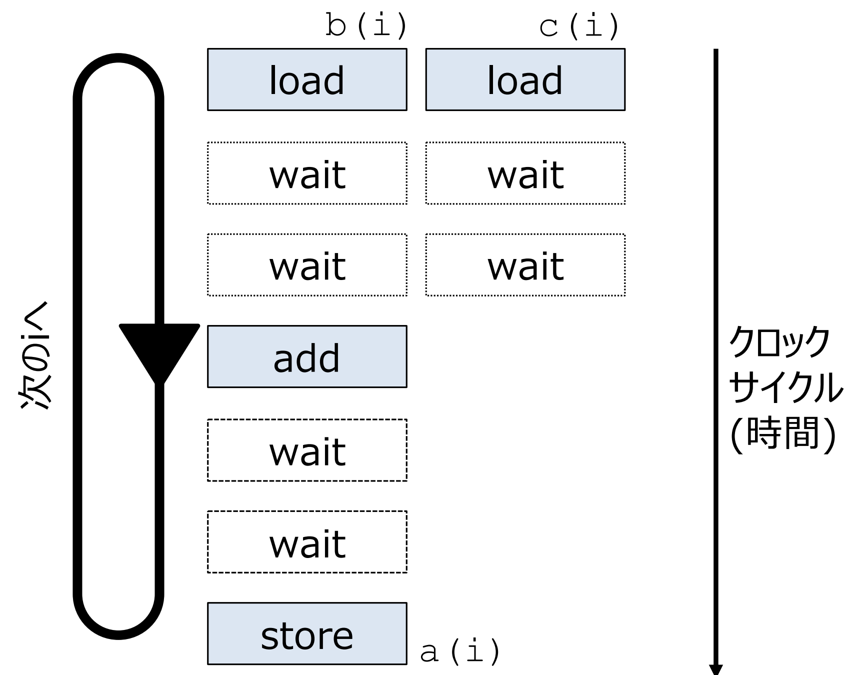
waveZ

ここから数ページは、以下の簡単なループを実行するときのことを考えてみる

```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```

このとき実行するハードウェアの想定

- Load命令は3サイクル
- Add命令は3サイクル
- Store命令は1サイクル
- ロードパイプは2本（同時に二つのロード命令を実行）
- ストアパイプは1本（同時に一つのストア命令を実行）
- 同時命令実行数は4



ソフトウェアパイプライン

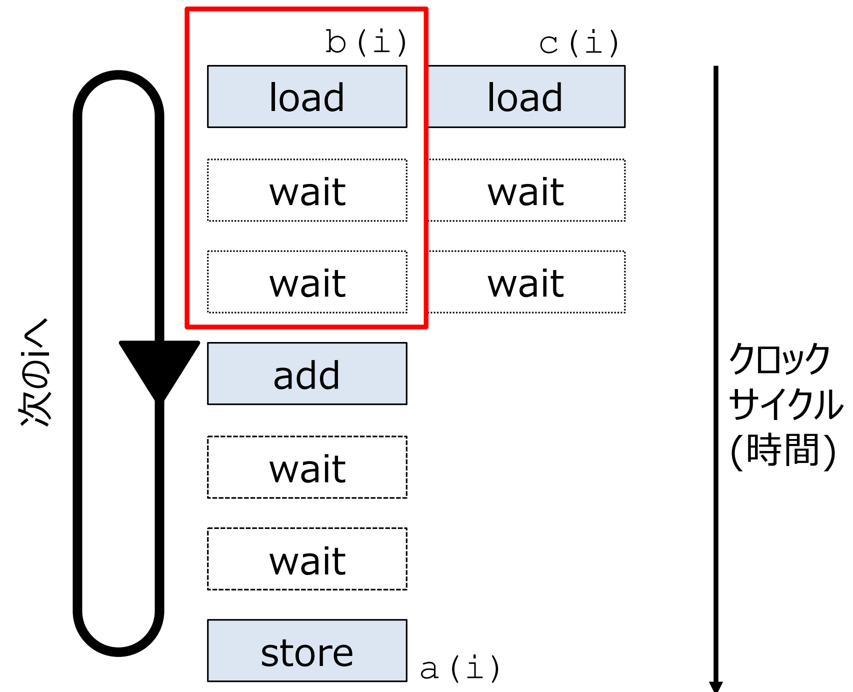
waveZ

ここから数ページは、以下の簡単なループを実行するときのことを考えてみる

```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```

このとき実行するハードウェアの想定

- Load命令は3サイクル
- Add命令は3サイクル
- Store命令は1サイクル
- ロードパイプは2本（同時に二つのロード命令を実行）
- ストアパイプは1本（同時に一つのストア命令を実行）
- 同時命令実行数は4



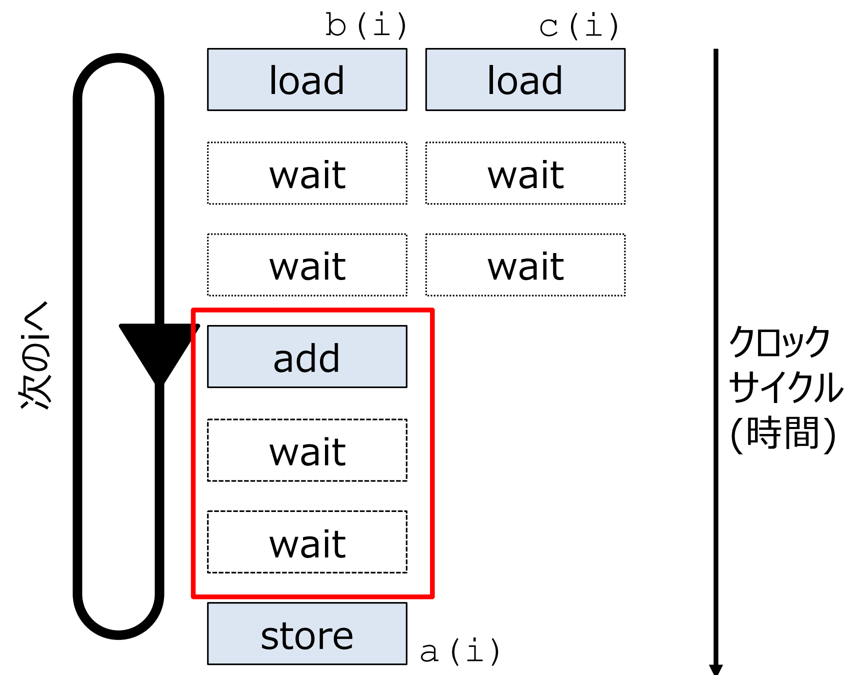
ソフトウェアパイプライン

waveZ

ここから数ページは、以下の簡単なループを実行するときのことを考えてみる

```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```

このとき実行するハードウェアの想定
- Load命令は3サイクル
- Add命令は3サイクル
- Store命令は1サイクル
- ロードパイプは2本（同時に二つのロード命令を実行）
- ストアパイプは1本（同時に一つのストア命令を実行）
- 同時命令実行数は4



ソフトウェアパイプライン

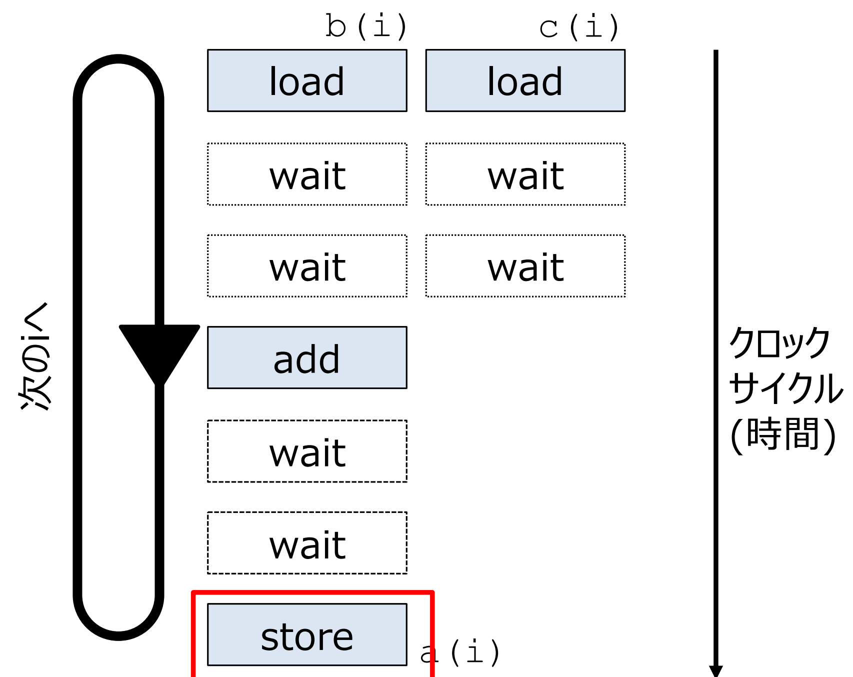
waveZ

ここから数ページは、以下の簡単なループを実行するときのことを考えてみる

```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```

このとき実行するハードウェアの想定

- Load命令は3サイクル
- Add命令は3サイクル
- Store命令は1サイクル
- ロードパイプは2本（同時に二つのロード命令を実行）
- ストアパイプは1本（同時に一つのストア命令を実行）
- 同時命令実行数は4



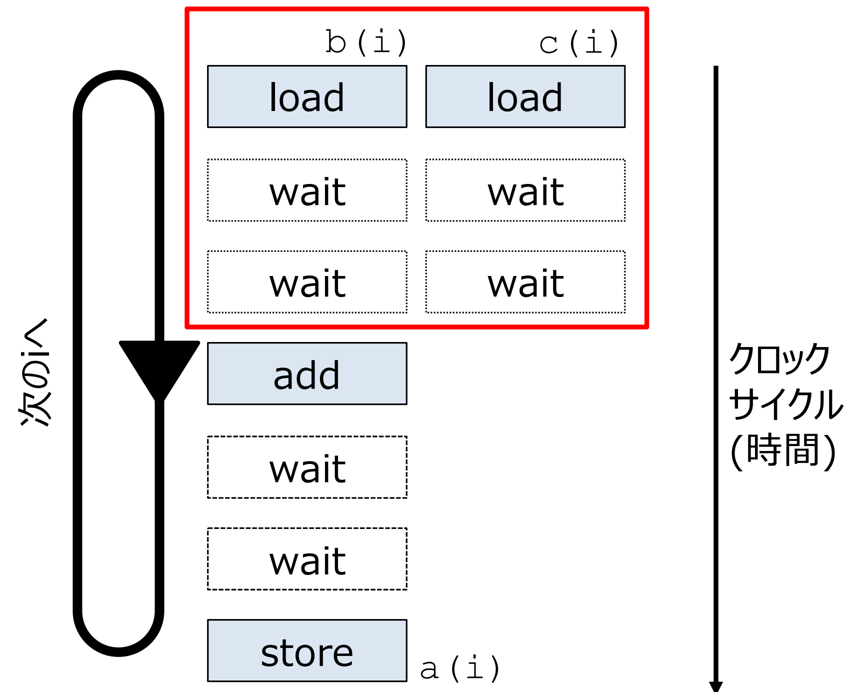
ソフトウェアパイプライン

waveZ

ここから数ページは、以下の簡単なループを実行するときのことを考えてみる

```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```

このとき実行するハードウェアの想定
- Load命令は3サイクル
- Add命令は3サイクル
- Store命令は1サイクル
- ロードパイプは2本（同時に二つのロード命令を実行）
- ストアパイプは1本（同時に一つのストア命令を実行）
- 同時命令実行数は4



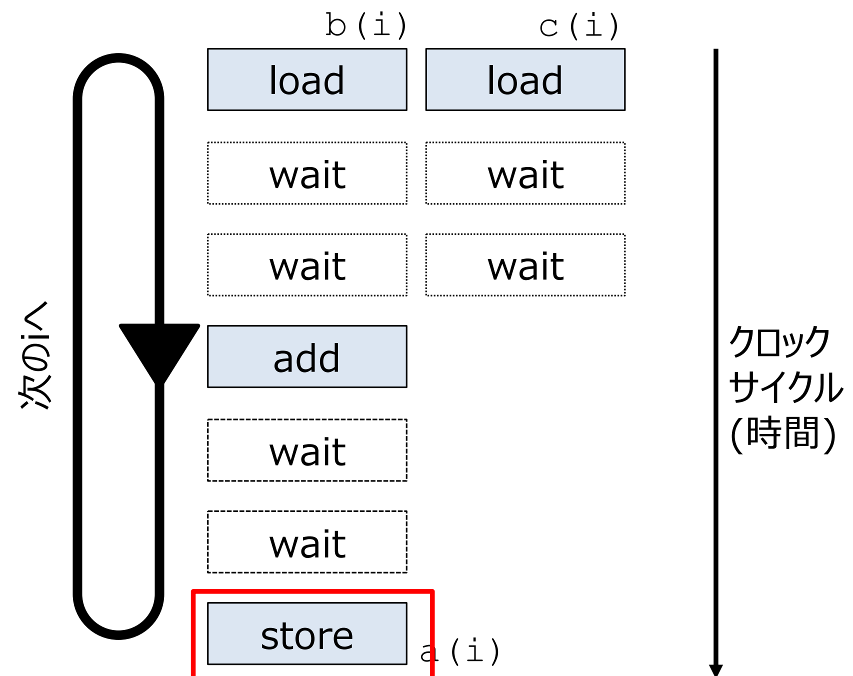
ソフトウェアパイプライン

waveZ

ここから数ページは、以下の簡単なループを実行するときのことを考えてみる

```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```

このとき実行するハードウェアの想定
- Load命令は3サイクル
- Add命令は3サイクル
- Store命令は1サイクル
- ロードパイプは2本（同時に二つのロード命令を実行）
- ストアパイプは1本（同時に一つのストア命令を実行）
- 同時命令実行数は4



ソフトウェアパイプライン

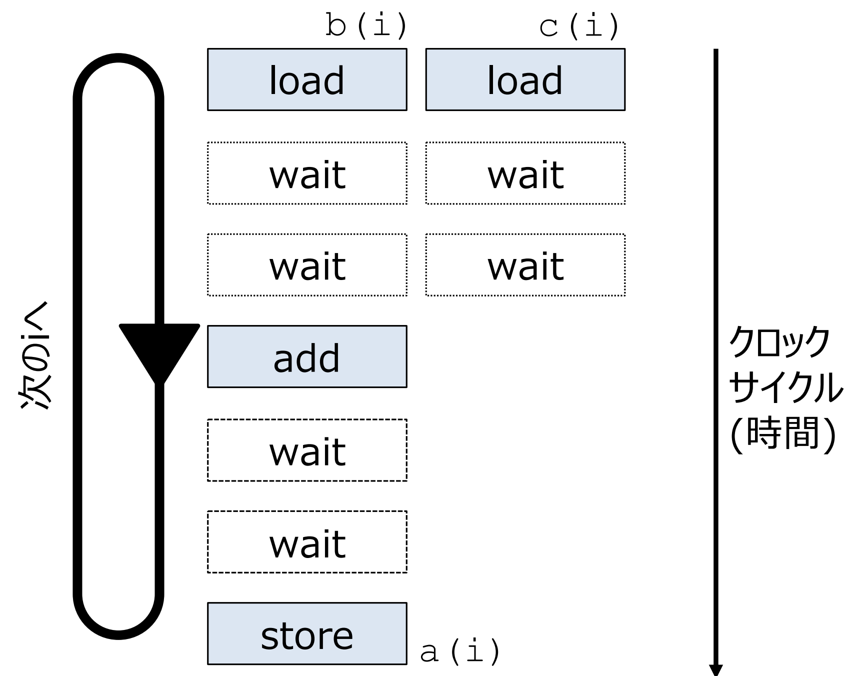
waveZ

ここから数ページは、以下の簡単なループを実行するときのことを考えてみる

```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```

このとき実行するハードウェアの想定

- Load命令は3サイクル
- Add命令は3サイクル
- Store命令は1サイクル
- ロードパイプは2本（同時に二つのロード命令を実行）
- ストアパイプは1本（同時に一つのストア命令を実行）
- 同時命令実行数は4

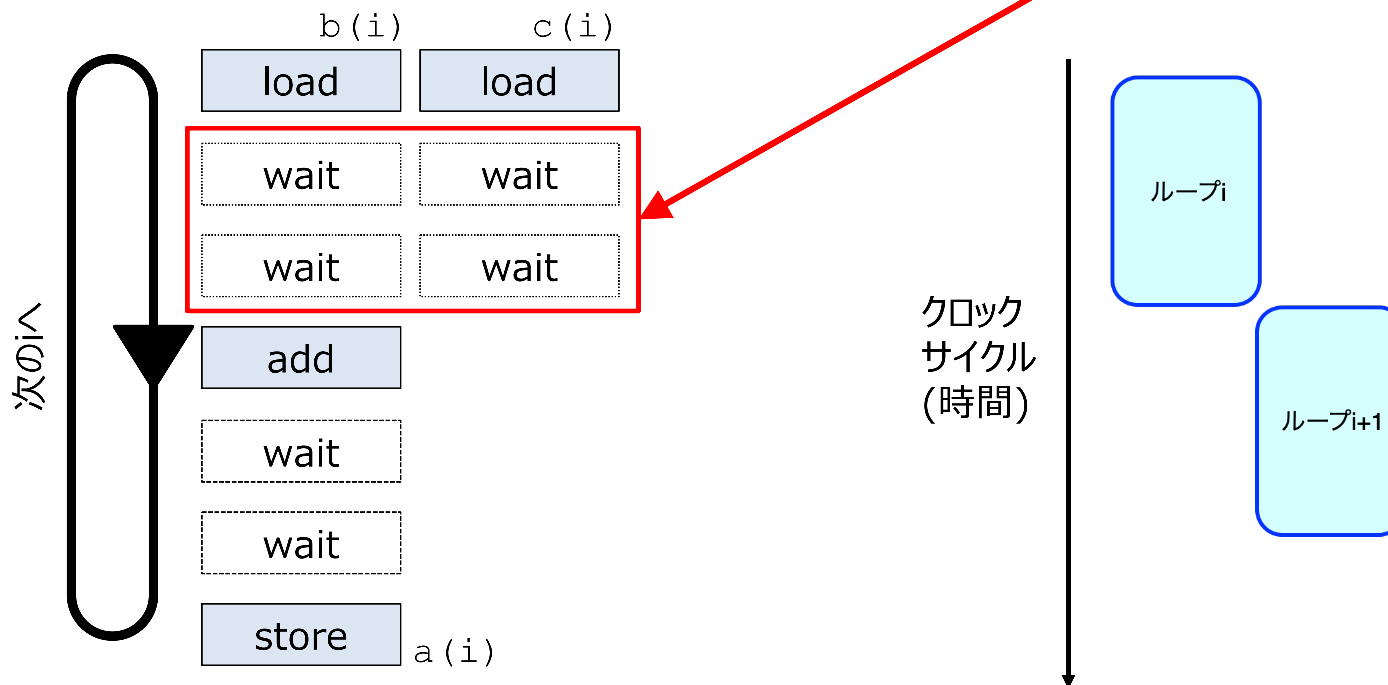


4命令で7サイクルかかる

ソフトウェアパイプライン

waveZ

```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```



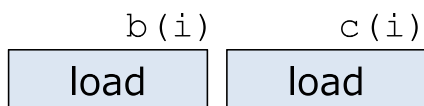
この状態だとループ i が完全に終わってから
ループ i+1 を始めているが wait の時に i+1
のloadを始めればよいのでは？

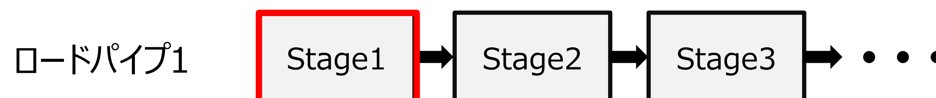
4命令で7サイクルかかる

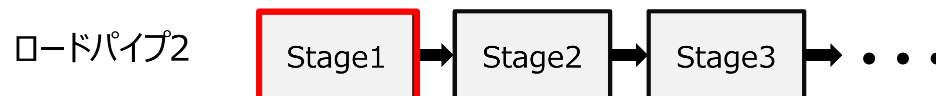
ソフトウェアパイプライン

```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```

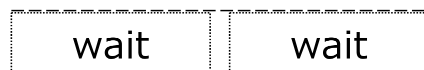
具体的にロードパイプの動きを見てみると、Stage1は空くので、次のループ回転 i+1 のロードのStage1を始めて良い

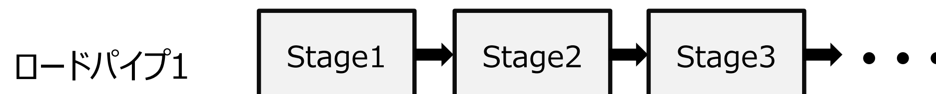


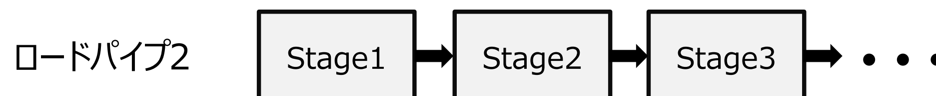


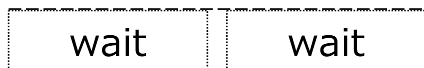


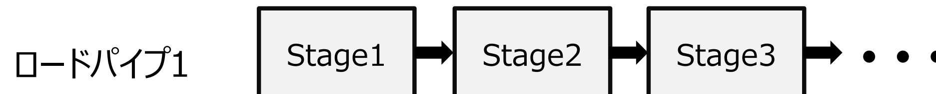
Loadの後を拡大

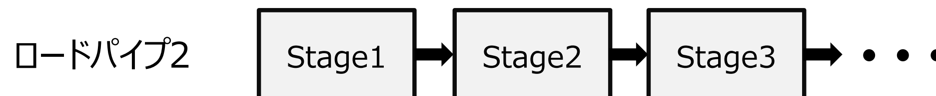


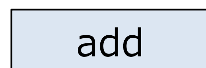










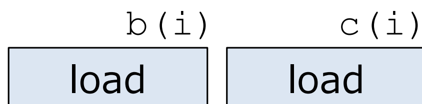


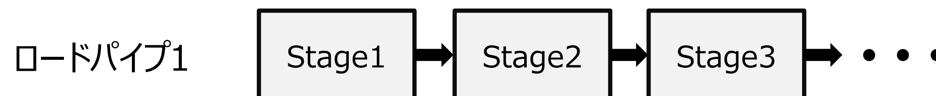
ソフトウェアパイプライン

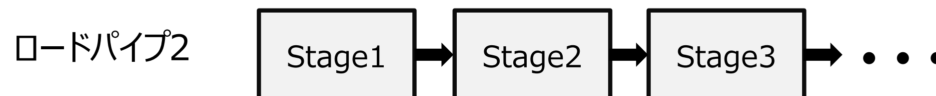
```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```

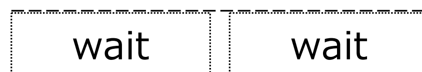
具体的にロードパイプの動きを見てみると、Stage1は空くので、次のループ回転 i+1 のロードのStage1を始めて良い

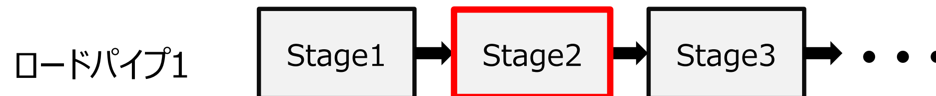
Loadの後を拡大

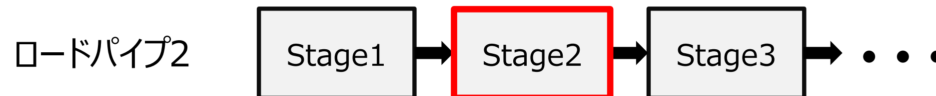


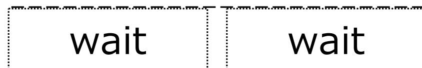


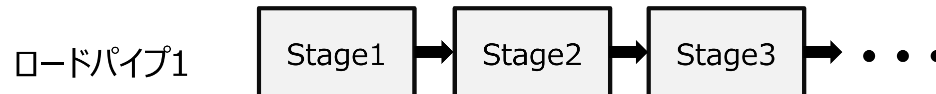


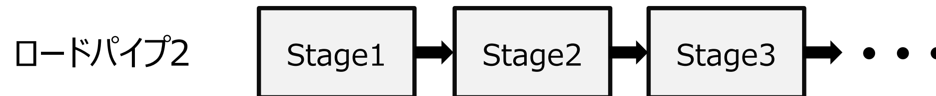


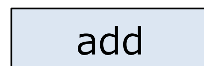








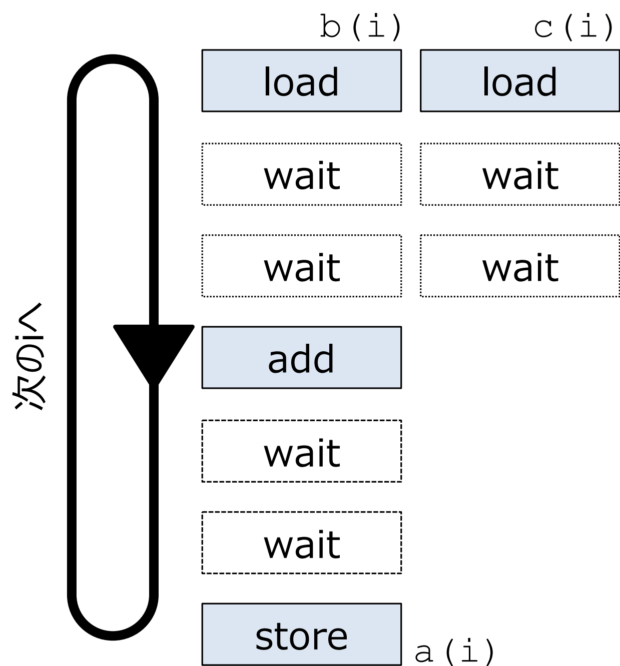


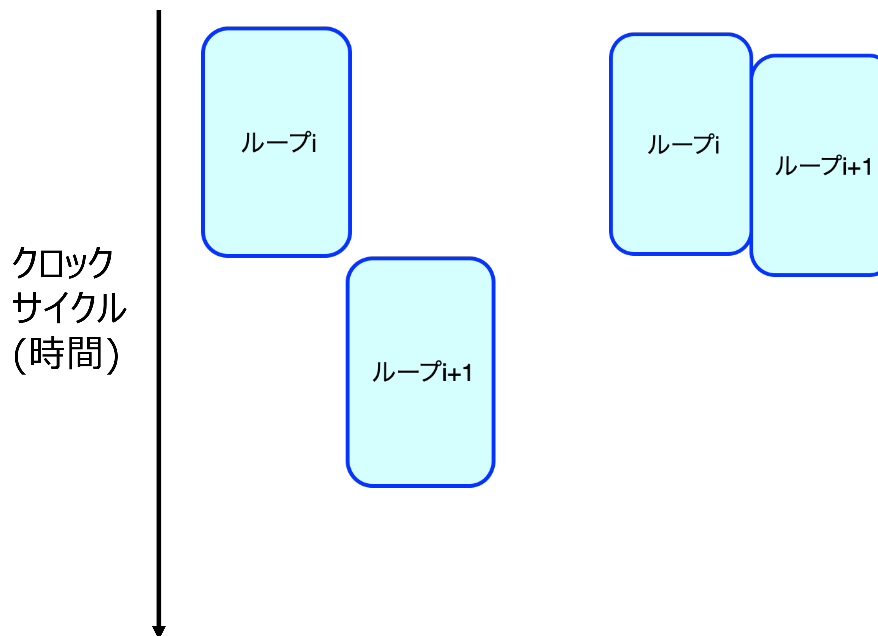


ソフトウェアパイプライン

```
do i=1, n
    a(i) = b(i) + c(i)
enddo
```

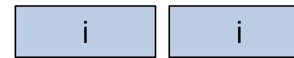
ループ i の途中からループ i+1 を始めて、二つのループを重ねて実行すれば、実行時間は短くできるはず





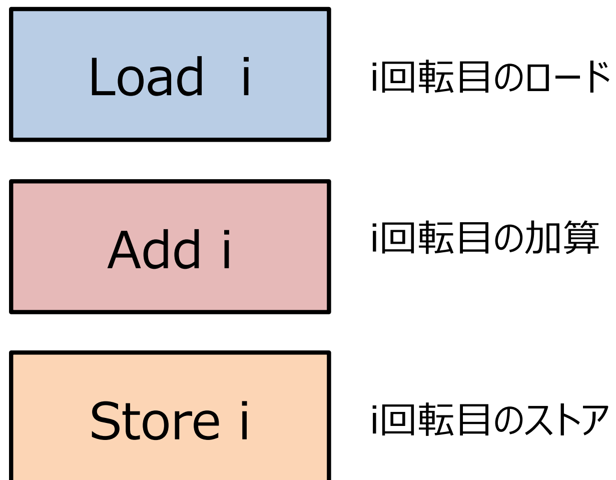
4命令で7サイクルかかる

ソフトウェアパイプライン



同時命令実行数は4

それをするのを
ソフトウェアパイプライン
という



i回転目のロード

i回転目の加算

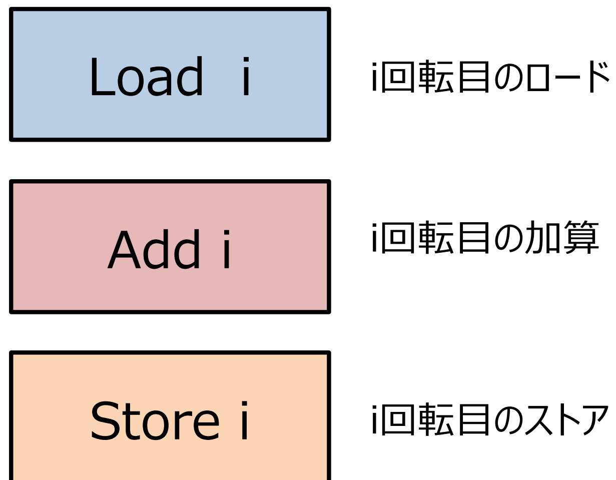
i回転目のストア

クロック
サイクル
(時間)

ソフトウェアパイプライン

同時命令実行数は4

それをするのを
ソフトウェアパイプライン
という

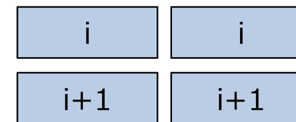


i回転目のロード

i回転目の加算

i回転目のストア

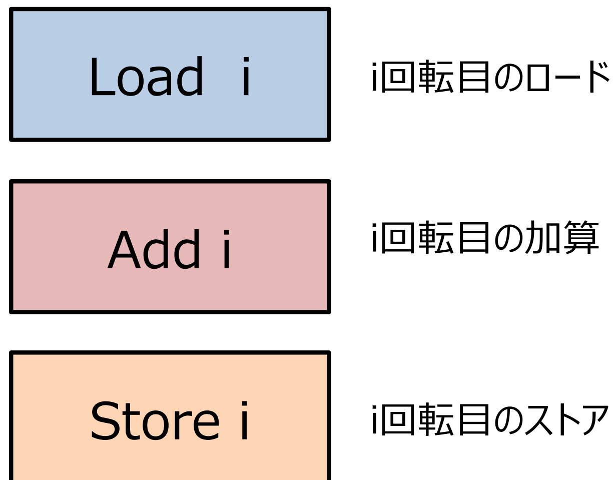
クロック
サイクル
(時間)



ソフトウェアパイプライン

waveZ

それをするのを
ソフトウェアパイプライン
という

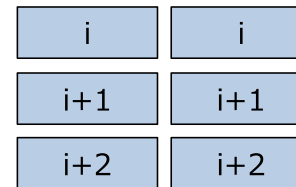


i回転目のロード

i回転目の加算

i回転目のストア

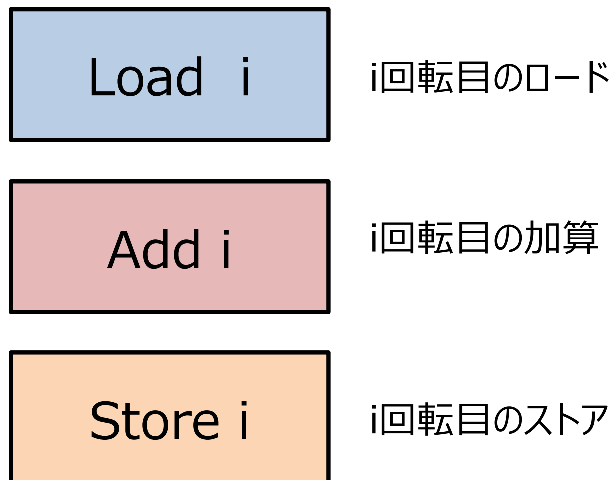
クロック
サイクル
(時間)



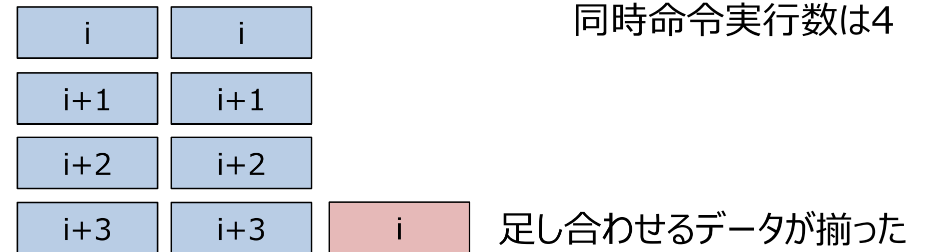
同時命令実行数は4

ソフトウェアパイプライン

それをするのを
ソフトウェアパイプライン
という



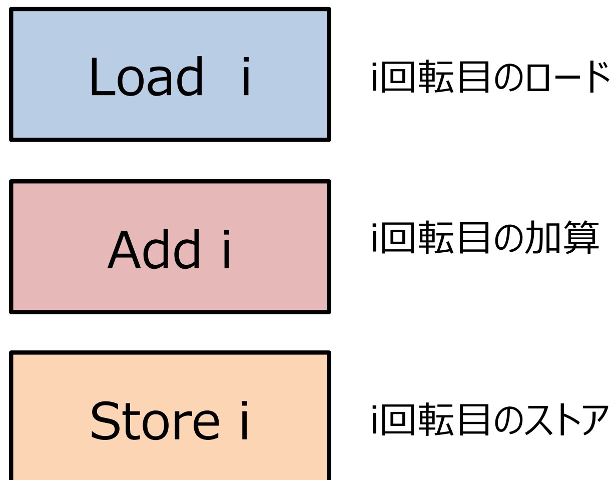
クロック
サイクル
(時間)



ソフトウェアパイプライン

同時命令実行数は4

それをするのを
ソフトウェアパイプライン
という

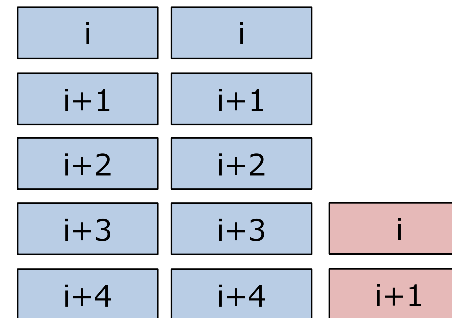


i回転目のロード

i回転目の加算

i回転目のストア

クロック
サイクル
(時間)

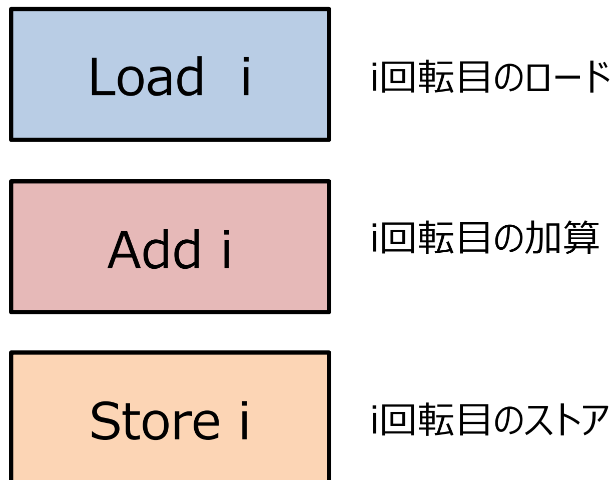


ソフトウェアパイプライン

waveZ

同時命令実行数は4

それをするのを
ソフトウェアパイプライン
という

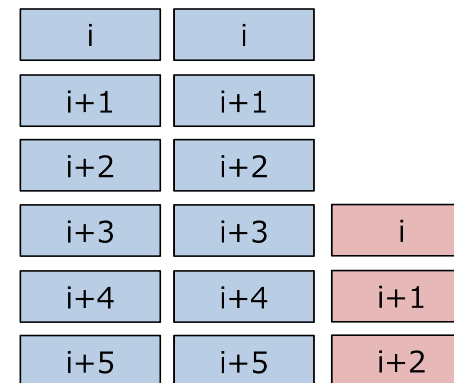


i回転目のロード

i回転目の加算

i回転目のストア

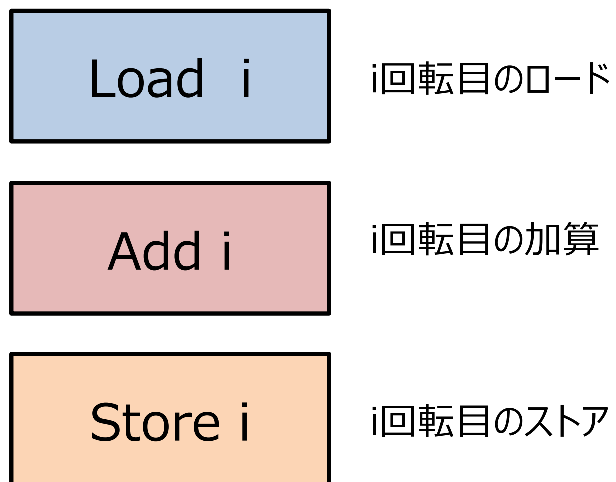
クロック
サイクル
(時間)



ソフトウェアパイプライン

waveZ

それをするのを
ソフトウェアパイプライン
という

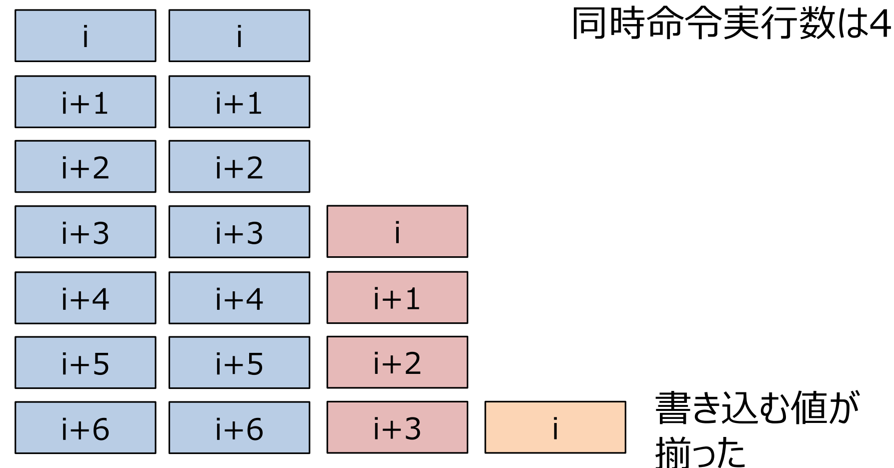


i回転目のロード

i回転目の加算

i回転目のストア

クロック
サイクル
(時間)



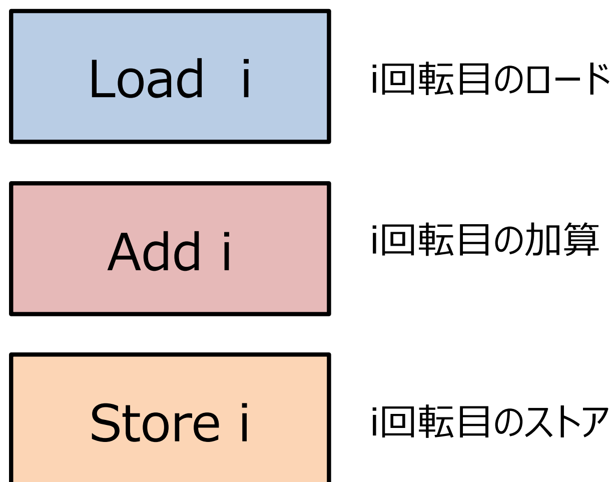
同時命令実行数は4

書き込む値が
揃った

ソフトウェアパイプライン

waveZ

それをするのを
ソフトウェアパイプライン
という

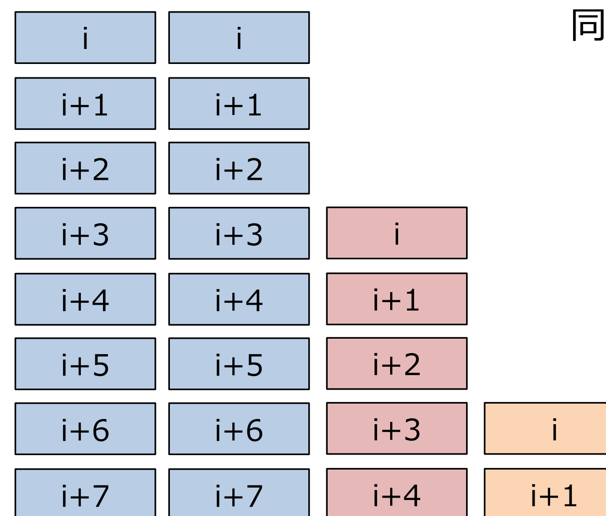


i回転目のロード

i回転目の加算

i回転目のストア

クロック
サイクル
(時間)

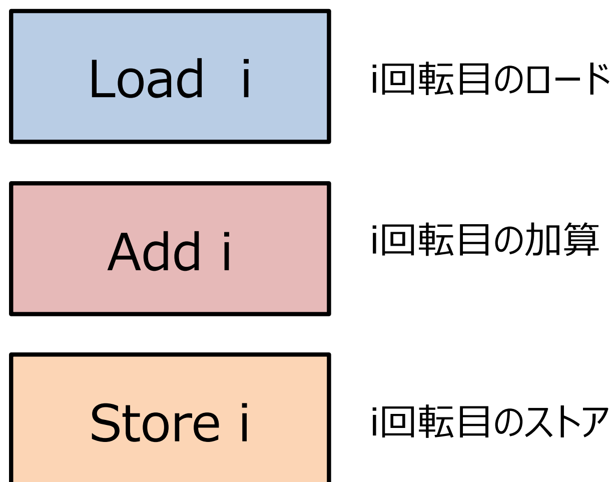


同時命令実行数は4

ソフトウェアパイプライン

waveZ

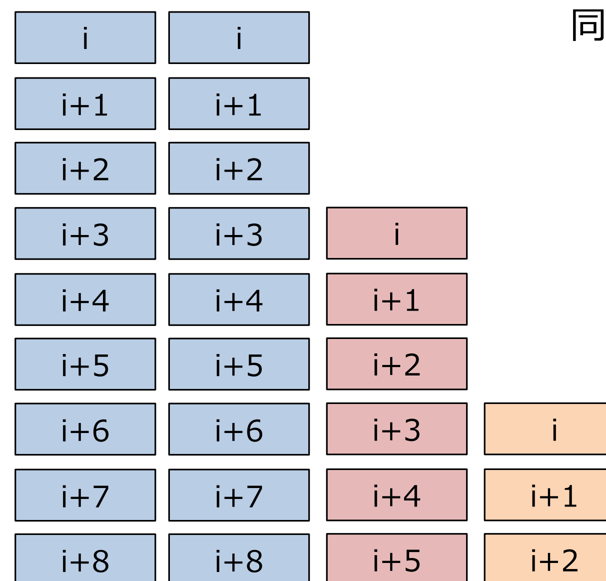
それをするのを
ソフトウェアパイプライン
という



i回転目のロード

i回転目の加算

i回転目のストア

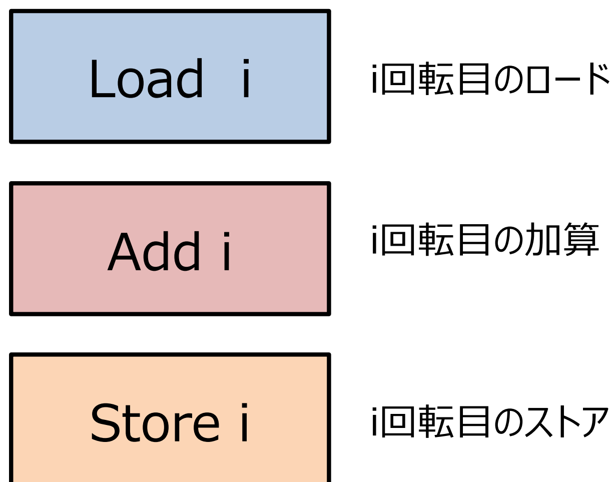


同時命令実行数は4

クロック
サイクル
(時間)

ソフトウェアパイプライン

それをするのを
ソフトウェアパイプライン
という

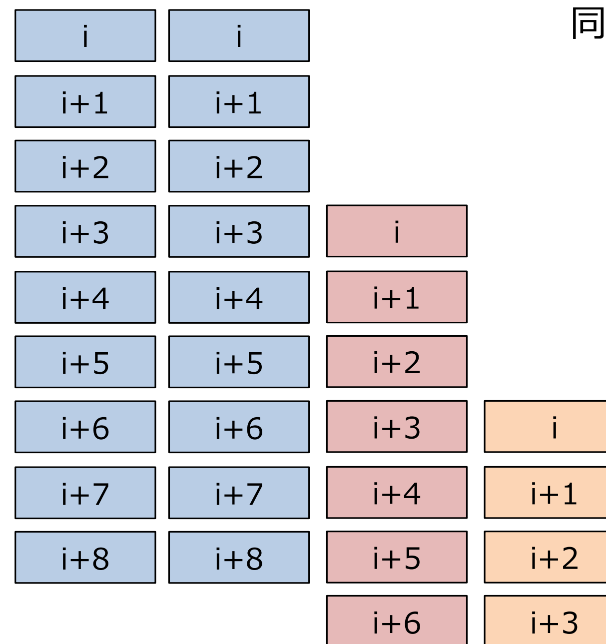


i回転目のロード

i回転目の加算

i回転目のストア

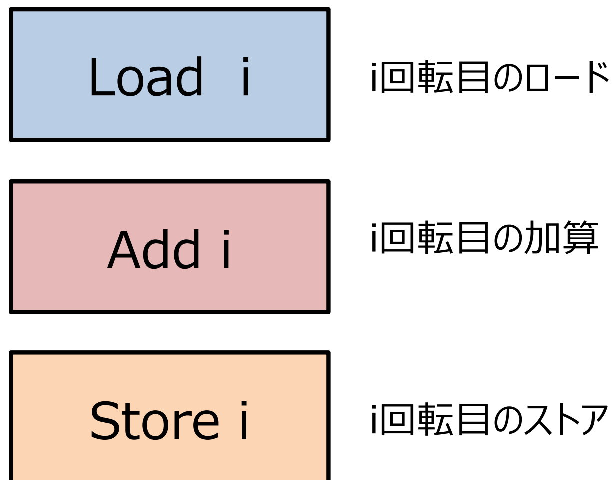
クロック
サイクル
(時間)



同時命令実行数は4

ソフトウェアパイプライン

それをするのを
ソフトウェアパイプライン
という

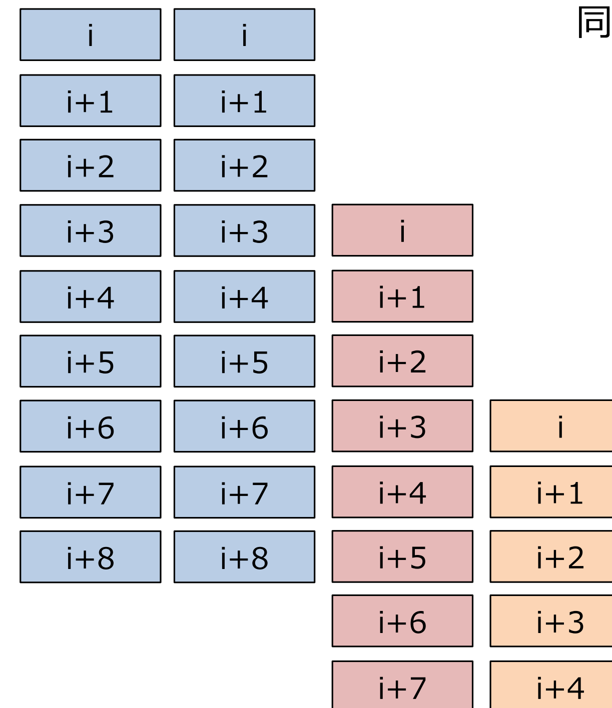


i回転目のロード

i回転目の加算

i回転目のストア

クロック
サイクル
(時間)

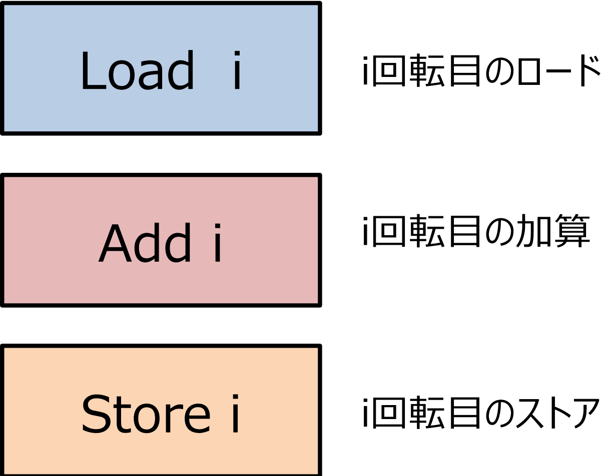


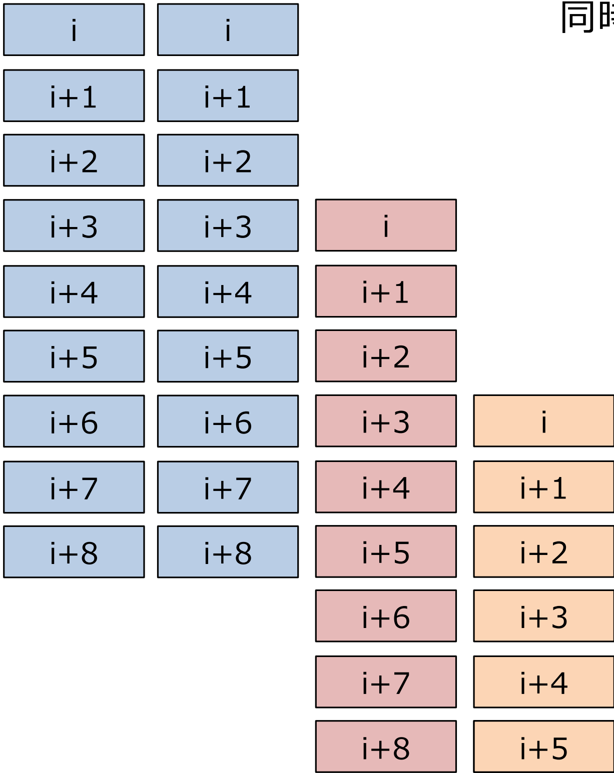
同時命令実行数は4

ソフトウェアパイプライン

同時命令実行数は4

それをするのを
ソフトウェアパイプライン
という

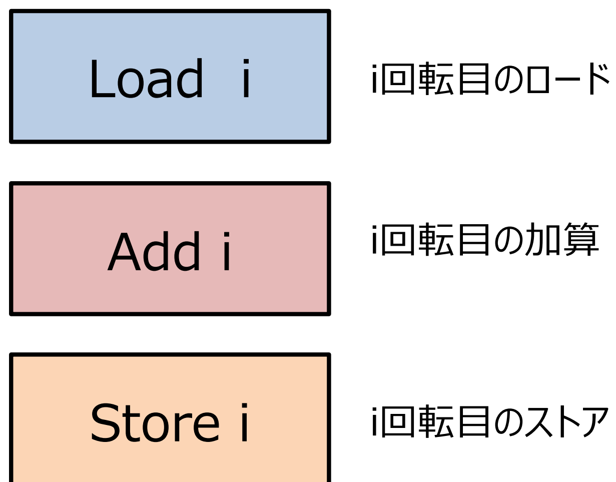


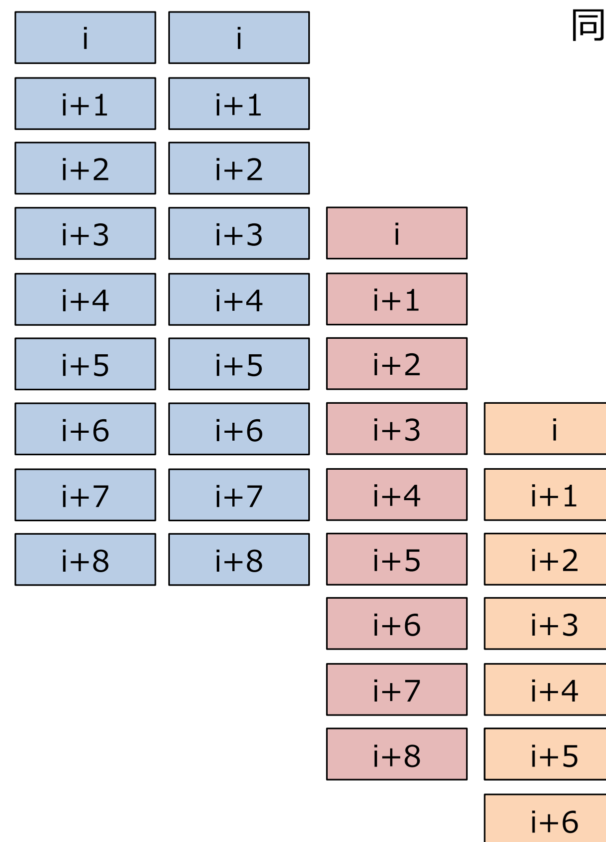


ソフトウェアパイプライン

waveZ

それをするのを
ソフトウェアパイプライン
という

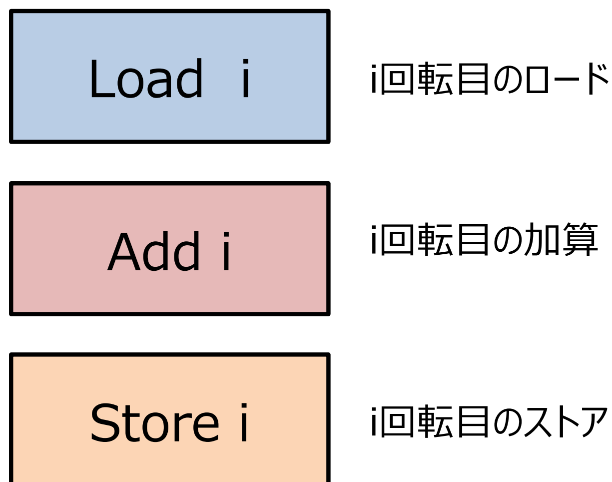




ソフトウェアパイプライン

waveZ

それをするのを
ソフトウェアパイプライン
という

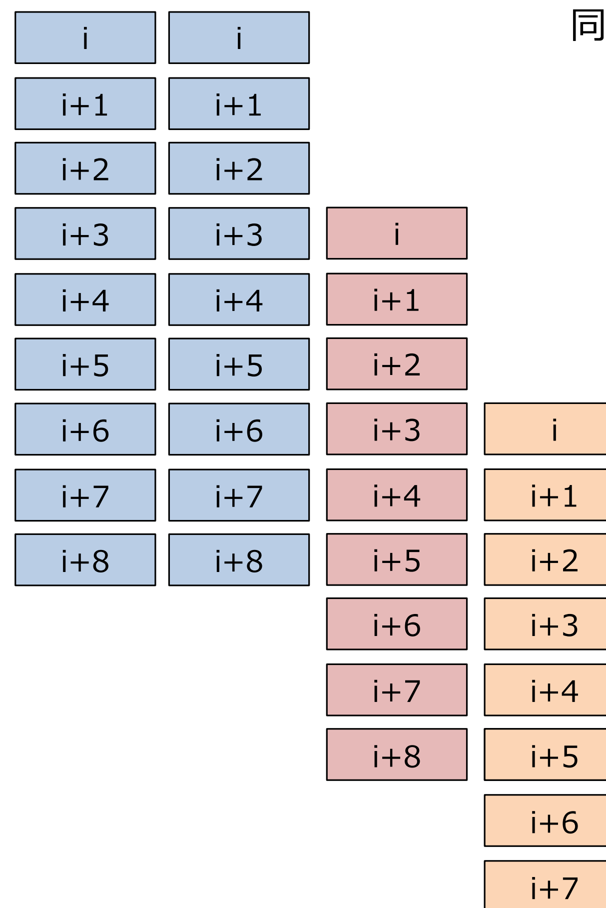


i回転目のロード

i回転目の加算

i回転目のストア

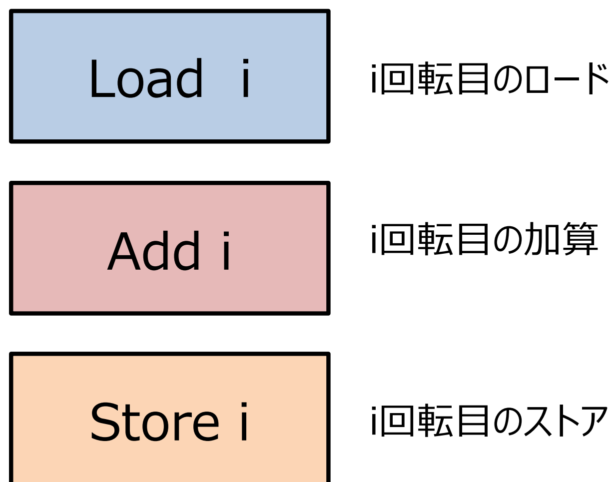
同時命令実行数は4



ソフトウェアパイプライン

waveZ

それをするのを
ソフトウェアパイプライン
という

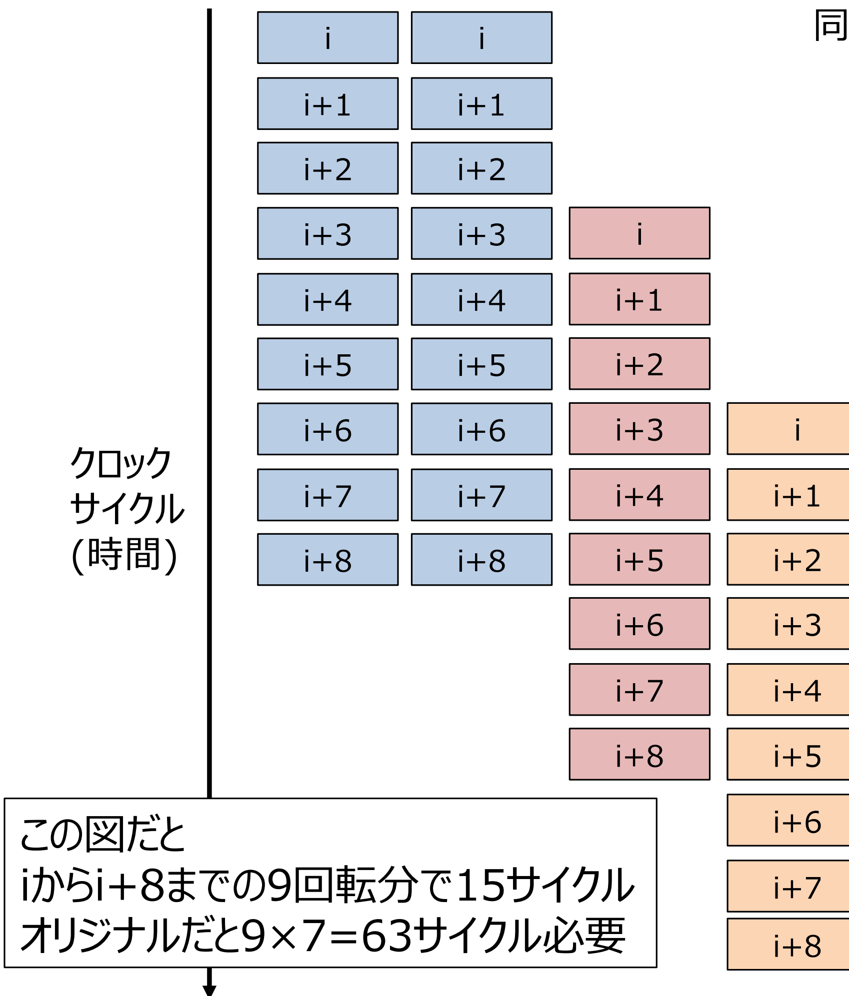


i回転目のロード

i回転目の加算

i回転目のストア

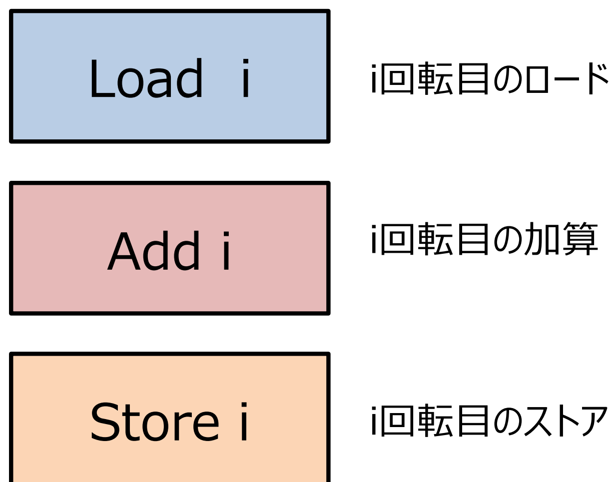
同時命令実行数は4

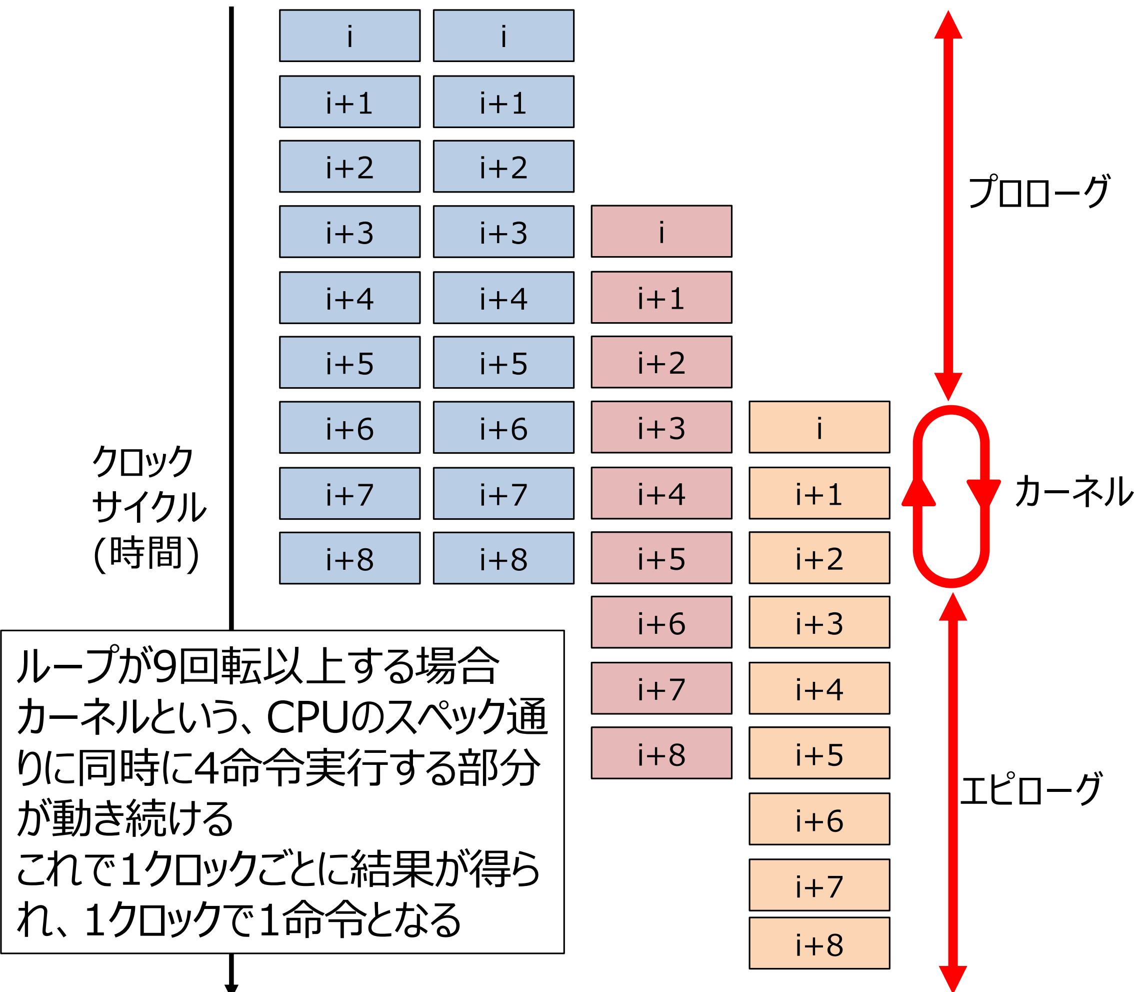


この図だと
iからi+8までの9回転分で15サイクル
オリジナルだと9×7=63サイクル必要

ソフトウェアパイプライン

それをするのを
ソフトウェアパイプライン
という

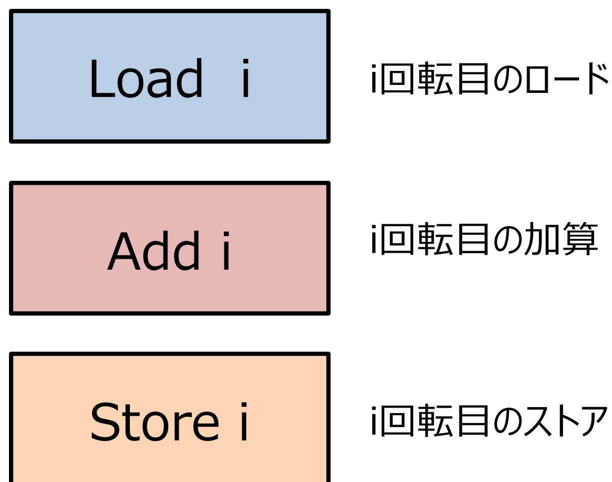




ソフトウェアパイプライン

waveZ

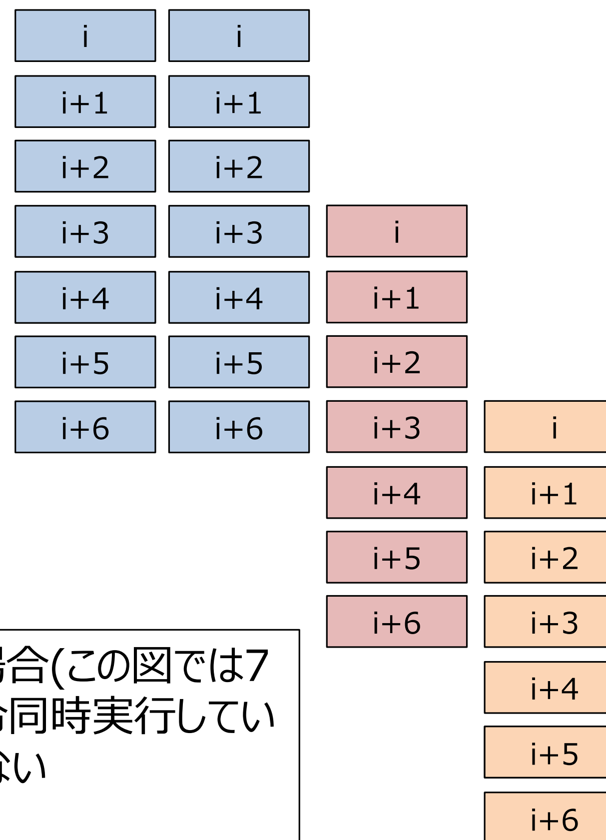
それをするのを
ソフトウェアパイプライン
という



i回転目のロード

i回転目の加算

i回転目のストア



クロック
サイクル
(時間)

ループが短い場合(この図では7回転)．4命令同時実行している時間がほぼない
↓
短いとソフトウェアパイプラインは効果がうすい

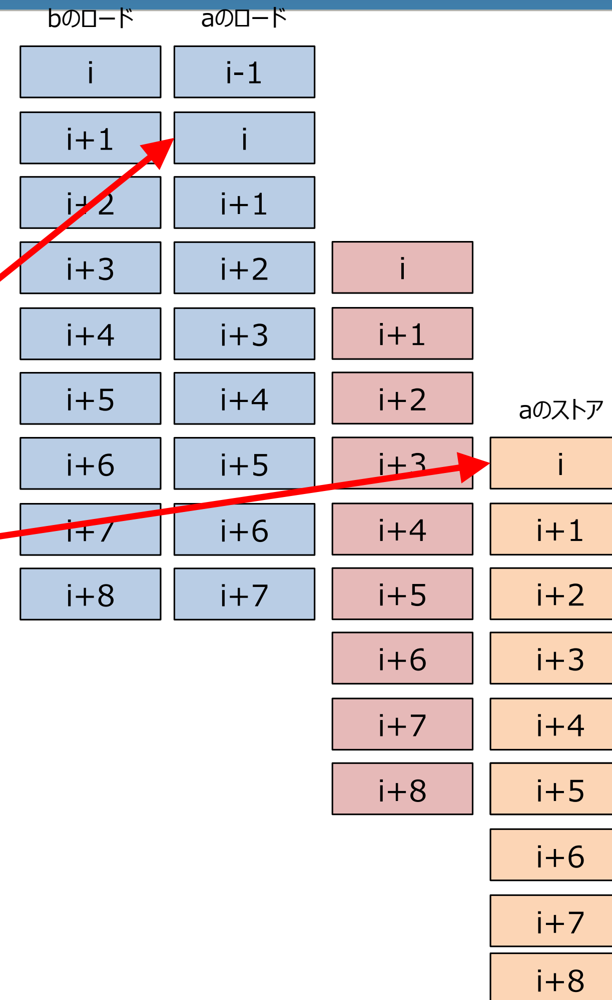
ソフトウェアパイプライン

waveZ

リカレンスがあるとソフトウェアパイプラインできない

```
do i=1, n
   a(i) = a(i-1) + b(i)
enddo
```

a(i)をストアする前に
a(i)をロードしてしまう
ので計算結果が
狂ってしまう



マシンのGFLOPSの算出

Core	Clock Freq.	2.2 GHz
	GFLOPS	70 GFLOPS
	L1D Cache	64 KiB (256GB /s)
CMG	# of Core	12
	GFLOPS	884 GFLOPS
	Memory	8 GiB (256 GB/s)
	L2 Cache	8 Mib (1024 GB/s)
CPU	# of CMG	4
	GFLOPS	3379 GFLOPS
	Memory	32 GiB
Node	# of CPU	1
System	# of Node	5760
	GFLOPS	19.4 PFLOPS
	Memory	180 TiB

1コア当たりの演算性能

クロック　2.2GHz

x

FMA　2　A=BxC+Dのような計算を1命令で実行する Fused Multiply Add

x

SIMD　8

x

PIPE　2 = 70.4GFLOPS

同時に実行できる命令数

マシンのGFLOPSの算出

Core	Clock Freq.	2.2 GHz
	GFLOPS	70 GFLOPS
	L1D Cache	64 KiB (256GB /s)
CMG	# of Core	12
	GFLOPS	884 GFLOPS
	Memory	8 GiB (256 GB/s)
	L2 Cache	8 Mib (1024 GB/s)
CPU	# of CMG	4
	GFLOPS	3379 GFLOPS
	Memory	32 GiB
Node	# of CPU	1
System	# of Node	5760
	GFLOPS	19.4 PFLOPS
	Memory	180 TiB

1コア当たりの演算性能

クロック 2.2GHz

x

FMA 2

x

SIMD 8 SIMDで実行しないと×8から×1になってしまう

x

PIPE 2 = 70.4GFLOPS

同時に実行できる命令数

Machine	京	FX100	FX1000
SIMD width	2 (128 bit)	4 (256 bit)	8 (512 bit)
# of register	256	256	32

SIMDありきの性能なのに注意

マシンのGFLOPSの算出

Core	Clock Freq.	2.2 GHz
	GFLOPS	70 GFLOPS
	L1D Cache	64 KiB (256GB /s)
CMG	# of Core	12
	GFLOPS	884 GFLOPS
	Memory	8 GiB (256 GB/s)
	L2 Cache	8 Mib (1024 GB/s)
CPU	# of CMG	4
	GFLOPS	3379 GFLOPS
	Memory	32 GiB
Node	# of CPU	1
System	# of Node	5760
	GFLOPS	19.4 PFLOPS
	Memory	180 TiB

1コア当たりの演算性能

クロック 2.2GHz ソフトウェアパイプラインができないと1クロックで1命令実行していると見なせない

x

FMA 2

x

SIMD 8

x

PIPE 2 = 70.4GFLOPS

同時に実行できる命令数

Machine	京	FX100	FX1000
SIMD width	2 (128 bit)	4 (256 bit)	8 (512 bit)
# of register	256	256	32

SIMDありきの性能なのに注意

メモリ・キャッシュ・レジスタ間のデータ転送

メモリの階層構造

メモリ(記憶装置)はコンピュータの中では階層構造をしてゆき，
上にあるものほど高速だが記憶容量が少ない

(図中色が付いているのが記憶装置)

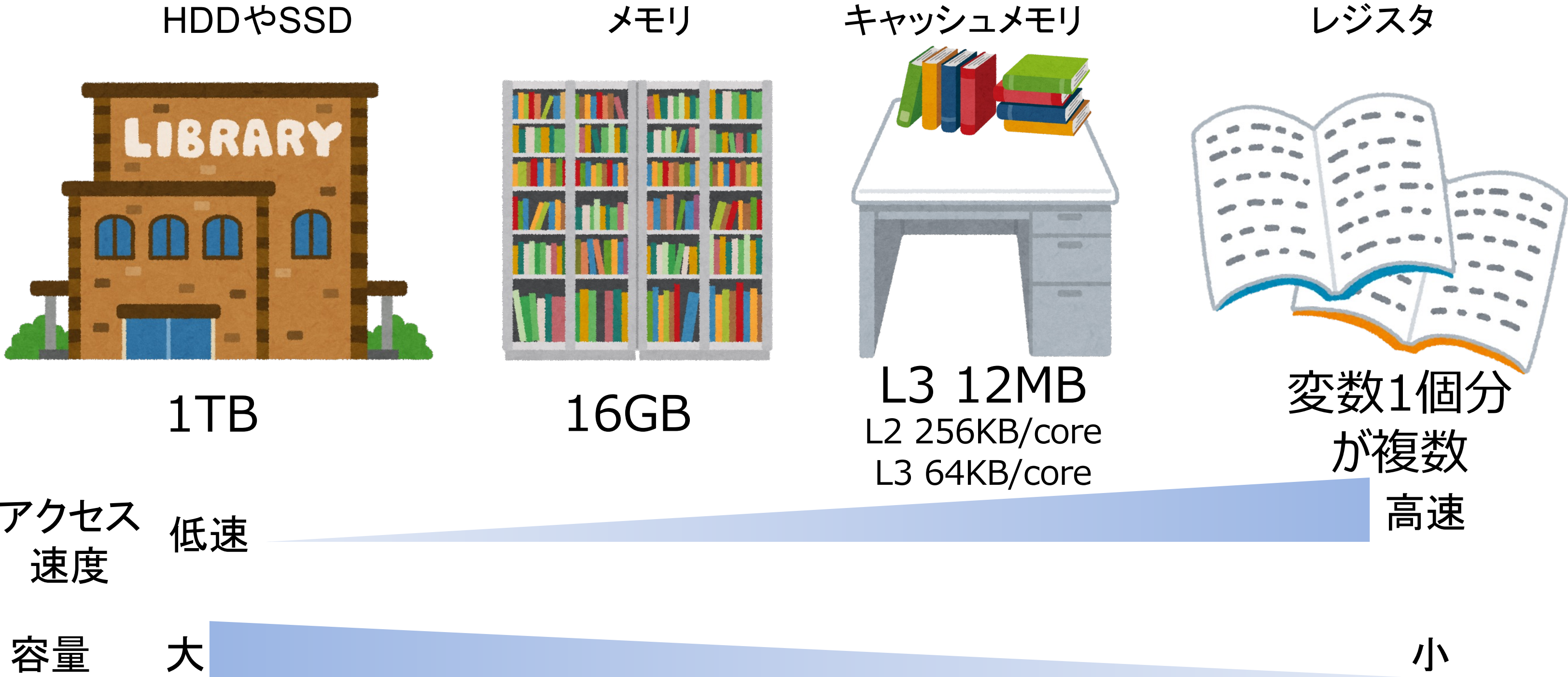
演算器
演算器
Register
Register
コア固有のCache
Core
Core
Core
Core
共有 Cache
Memory

メモリの階層構造

よくあるイメージ

メモリの階層構造

パソコンだと



メモリの階層構造

パソコンだと

HDDやSSD	メモリ	キャッシュメモリ	レジスタ
1TB	16GB	L3 12MB L2 256KB/core L3 64KB/core	変数1個分 が複数

アクセス速度　低速 → 高速

容量　大 → 小

メモリと違って小容量
独自の凝った仕組みがありと
それゆえに起きる問題をご紹介

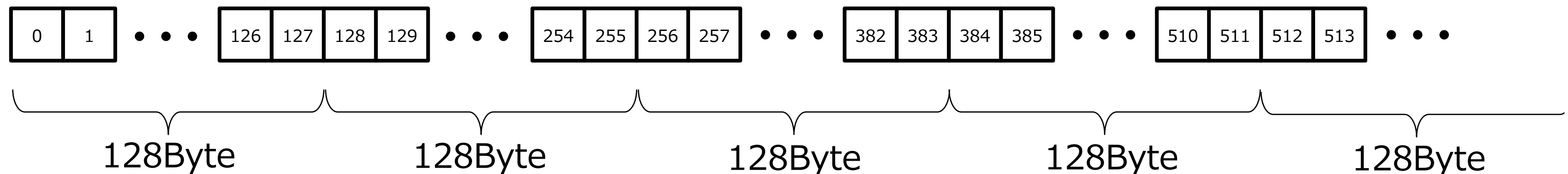
メモリとキャッシュ

waveZ

キャッシュメモリは，メモリの一部のデータを選択的に保持している
より高速な記憶領域

「選択的に」というのはどうしているのか？

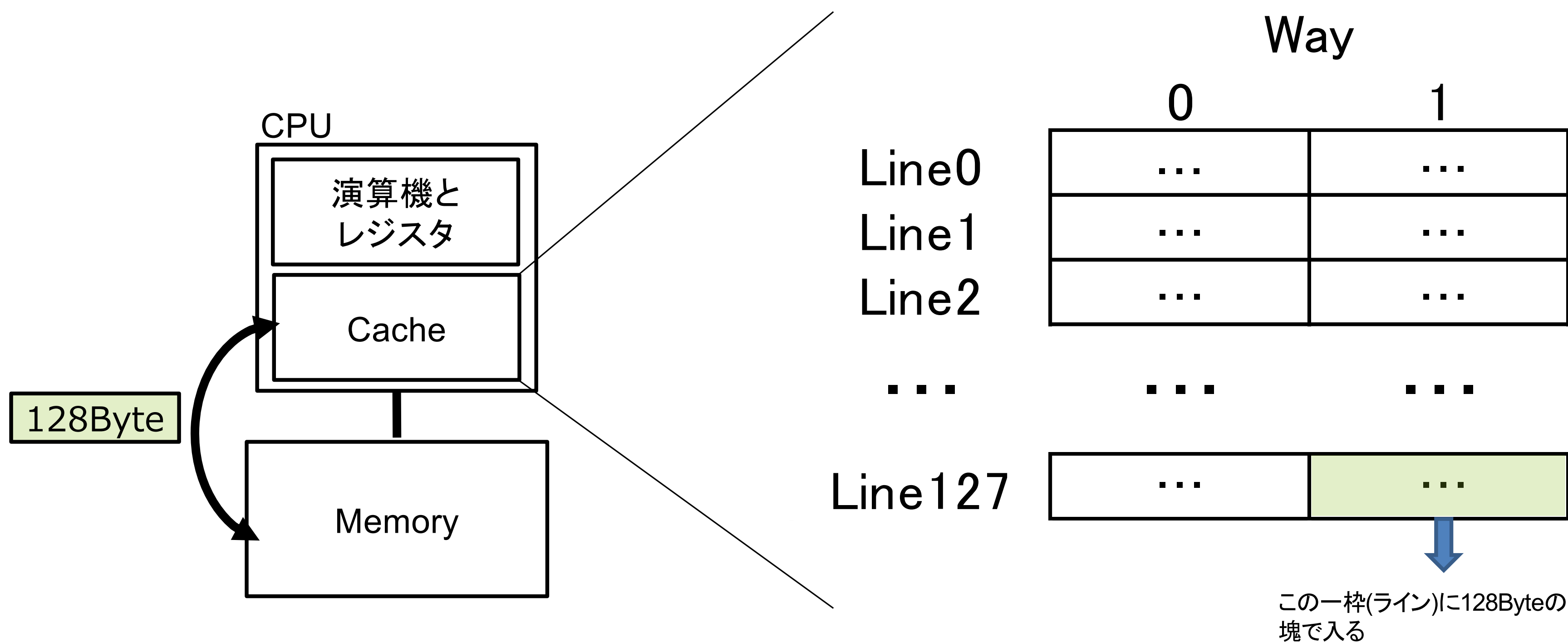
メモリは1バイトごとにアドレスがあり管理されている



キャッシュの仕組みでは，アドレスをNバイト(ここでは128バイト)ごとの単位でまとめて，それぞれをキャッシュに置くか置かないかを管理している

waveZ

どうやって管理しているのか



1ライン128Byte, 2 Way-set Associativeで
128ラインであり合計32KBのキャッシュの例

waveZ

キャッシュの動作 Hitの場合

	Way 0	Way 1
Line0	…	…
Line1	…	…
Line2	386	13186
…	…	…
Line127	…	…

1. アドレス49500の値を要求
2. 要求アドレスを128B単位のブロック番号に変換
   ADDR=49500
   ↓
   INDEX=[ADDR/128]=386
   49500の内容を含むブロックは386と判明
3. インデックスをライン番号に変換
   LINE=MOD(INDEX,128)=2
4. Line2のWay0かWay1に要求アドレスに対応する128Byteのデータが入っていたらHit

キャッシュの動作 Hitの場合

	Way 0	Way 1
Line0	…	…
Line1	…	…
Line2	386	13186
…	…	…
Line127	…	…

1. アドレス49500の値を要求
2. 要求アドレスを128B単位のブロック番号に変換
   ADDR=49500
   ↓
   INDEX=[ADDR/128]=386
   49500の内容を含むブロックは386と判明
3. インデックスをライン番号に変換
   LINE=MOD(INDEX,128)=2
4. Line2のWay0かWay1に要求アドレスに対応する128Byteのデータが入っていたらHit

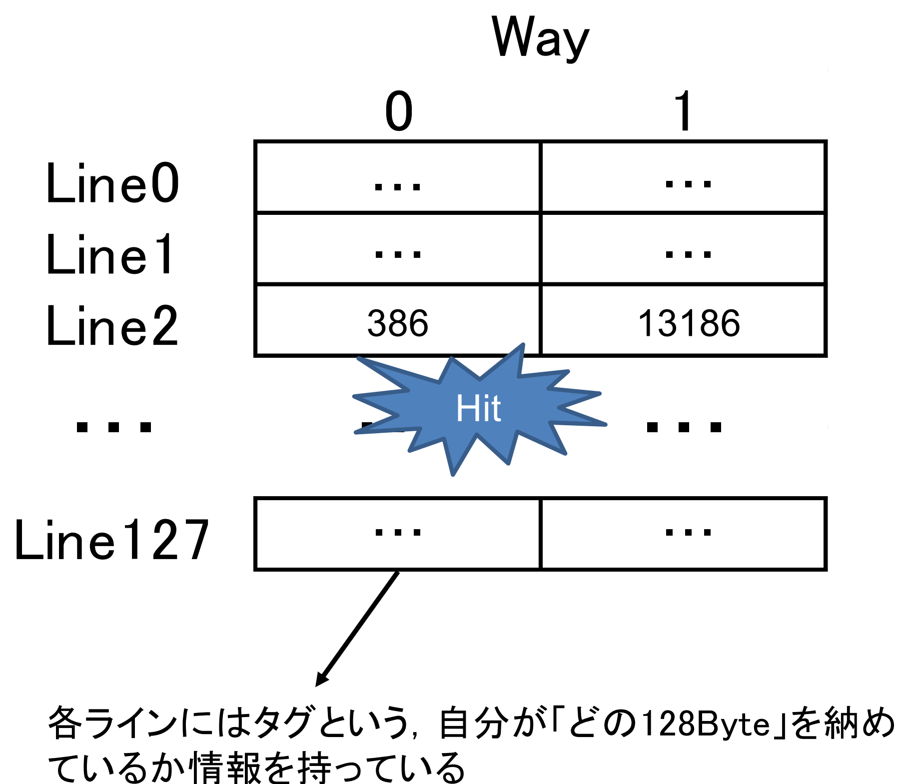
キャッシュの動作 Hitの場合

	Way 0	Way 1
Line0	…	…
Line1	…	…
Line2	386	13186
…	…	…
Line127	…	…

1. アドレス49500の値を要求
2. 要求アドレスを128B単位のブロック番号に変換
   ADDR=49500
   ↓
   INDEX=[ADDR/128]=386
   49500の内容を含むブロックは386と判明
3. インデックスをライン番号に変換
   LINE=MOD(INDEX,128)=2
4. Line2のWay0かWay1に要求アドレスに対応する128Byteのデータが入っていたらHit

waveZ

キャッシュの動作 Hitの場合



各ラインにはタグという，自分が「どの128Byte」を納めているか情報を持っている

1. アドレス49500の値を要求
2. 要求アドレスを128B単位のブロック番号に変換
   ADDR=49500
   ↓
   INDEX=[ADDR/128]=386
   49500の内容を含むブロックは386と判明
3. インデックスをライン番号に変換
   LINE=MOD(INDEX,128)=2
4. Line2のWay0かWay1に要求アドレスに対応する128Byteのデータが入っていたらHit

キャッシュの動作 Missの場合①

	Way 0	Way 1
Line0	…	…
Line1	…	…
Line2		13186
…	…	…
Line127	…	…

1. アドレス49500の値を要求
2. 要求アドレスを128B単位のブロック番号に変換
   ADDR=49500
   ↓
   INDEX=[ADDR/128]=386
   49500の内容を含むブロックは386と判明
3. インデックスをライン番号に変換
   LINE=MOD(INDEX,128)=2
4. 入っていなければ次のレベル(L2やメモリ)から
   データを持ってくる

waveZ

キャッシュの動作 Missの場合①

	Way 0	Way 1
Line0	・・・	・・・
Line1	・・・	・・・
Line2		13186
・・・	・・・	・・・
Line127	・・・	・・・

1. アドレス49500の値を要求

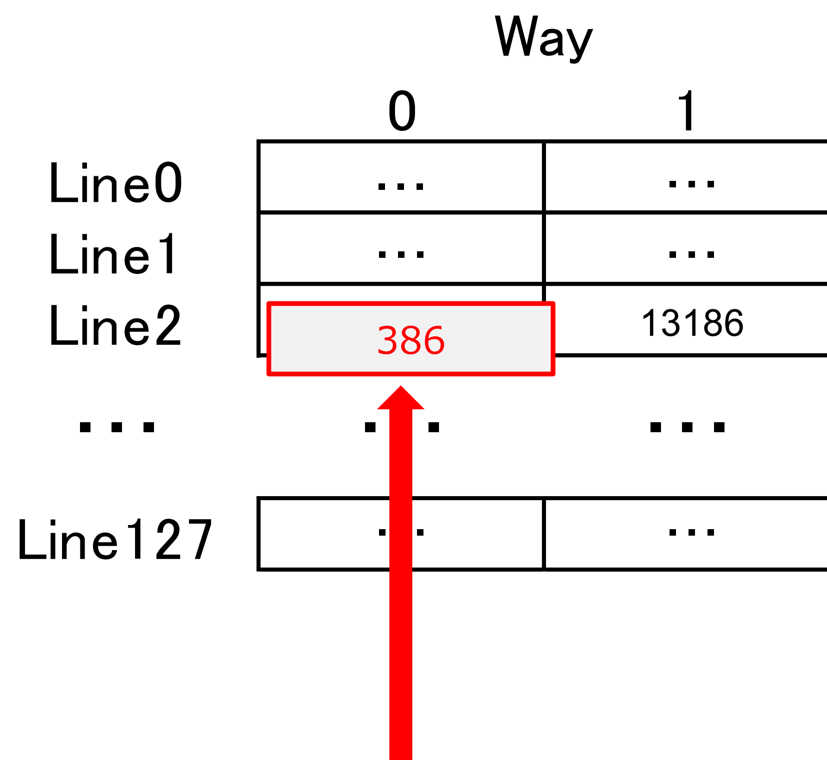
2. 要求アドレスを128B単位のブロック番号に変換
   ADDR=49500
   ↓
   INDEX=[ADDR/128]=386
   49500の内容を含むブロックは386と判明

3. インデックスをライン番号に変換
   LINE=MOD(INDEX,128)=2

4. 入っていなければ次のレベル(L2やメモリ)から
   データを持ってくる

waveZ

キャッシュの動作 Missの場合①



1. アドレス49500の値を要求

2. 要求アドレスを128B単位のブロック番号に変換
   ADDR=49500
   ↓
   INDEX=[ADDR/128]=386
   49500の内容を含むブロックは386と判明

3. インデックスをライン番号に変換
   LINE=MOD(INDEX,128)=2

4. 入っていなければ次のレベル(L2やメモリ)から
   データを持ってくる

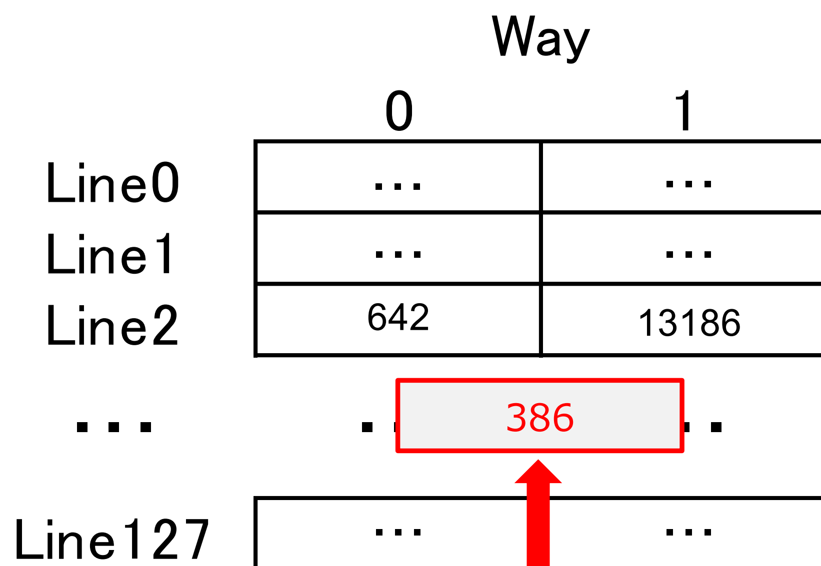
waveZ

キャッシュの動作 Missの場合②

	Way 0	Way 1
Line0	…	…
Line1	…	…
Line2	642	13186
…	…	…
Line127	…	…

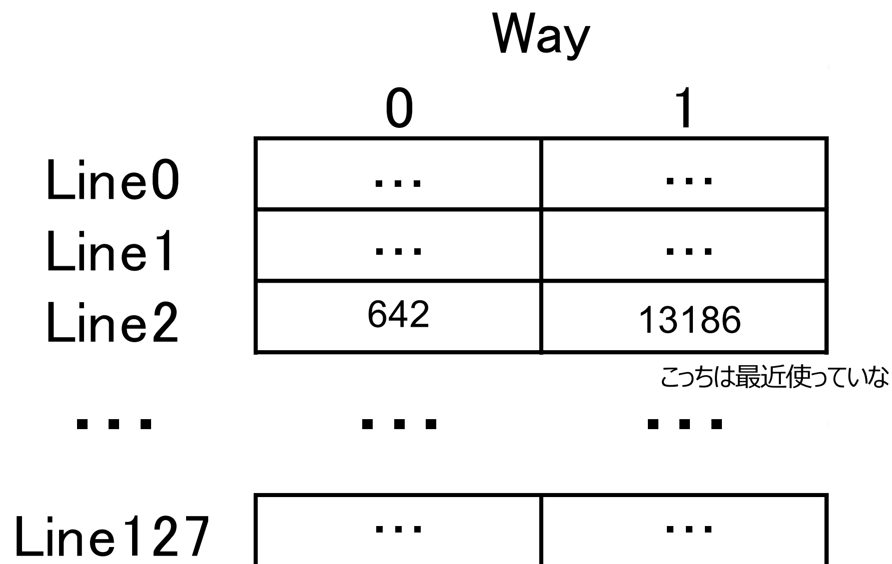
1. アドレス49500の値を要求
2. 要求アドレスを128B単位のブロック番号に変換
   ADDR=49500
   ↓
   INDEX=[ADDR/128]=386
   49500の内容を含むブロックは386と判明
3. インデックスをライン番号に変換
   LINE=MOD(INDEX,128)=2
4. 入っていなければ次のレベル(L2やメモリ)から
   データを持ってくる

キャッシュの動作 Missの場合②



1. アドレス49500の値を要求
2. 要求アドレスを128B単位のブロック番号に変換
   ADDR=49500
   ↓
   INDEX=[ADDR/128]=386
   49500の内容を含むブロックは386と判明
3. インデックスをライン番号に変換
   LINE=MOD(INDEX,128)=2
4. 入っていなければ次のレベル(L2やメモリ)から
   データを持ってくる

キャッシュの動作 Missの場合②



タグには，同じラインの中で「最近使われたか」「最近は使われていないか」を識別するための情報がある
最近使ってない方は，もういらないのでは？

1. アドレス49500の値を要求

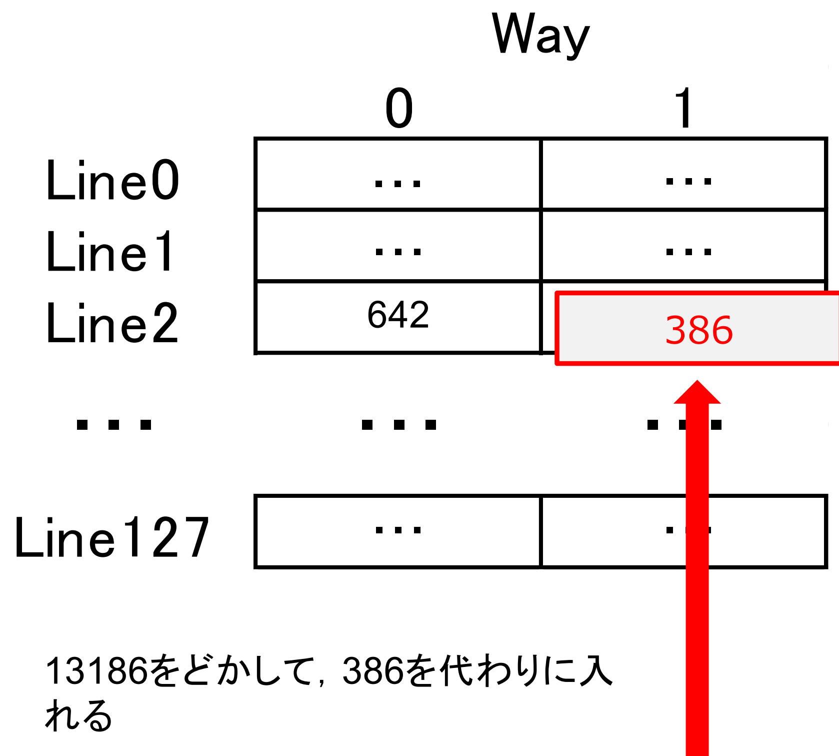
2. 要求アドレスを128B単位のブロック番号に変換
   ADDR=49500
   ↓
   INDEX=[ADDR/128]=386
   49500の内容を含むブロックは386と判明

3. インデックスをライン番号に変換
   LINE=MOD(INDEX,128)=2

4. 入っていなければ次のレベル(L2やメモリ)からデータを持ってくる

waveZ

キャッシュの動作 Missの場合②



13186をどかして，386を代わりに入れる

1. アドレス49500の値を要求
2. 要求アドレスを128B単位のブロック番号に変換
   ADDR=49500
   ↓
   INDEX=[ADDR/128]=386
   49500の内容を含むブロックは386と判明
3. インデックスをライン番号に変換
   LINE=MOD(INDEX,128)=2
4. 入っていなければ次のレベル(L2やメモリ)からデータを持ってくる

キャッシュの動作 Missの場合② どれをどかすか？

どかすやりかた(=ラインの入れ替え方式)には種類がある

- ラウンドロビン
  今着目しているラインでは，さっきはWay0の中身をどかしたので，今度はWay1の中身をどかそう
- ランダム
  呼んで字のごとく
- LRU★
  - Least Recently Used の略
  - さっき使ったばかりのデータは，また，すぐに使いそうだからとっておこう
  - その逆に，しばらく使っていないデータは，もう使わないだろう
  - 今着目しているLine中で，一番過去にアクセスされたWayの中身をどかす

キャッシュの動作 Missの場合② データ更新方式

CPUでデータを書き換えたら(ストア命令発生)，そのデータはメモリに戻さないといけないが，いつそれをやる？

- ライトスルー方式
  - ストア命令で，まずはキャッシュ中にあるデータが書き換えられる
  - このタイミングでメモリ上の対応する部分も書き換える
  - L2もライトスルー方式なら，メモリも書き換える
- ライトバック方式
  - キャッシュ中のデータを書き換えたか否かのフラグ(dirty flag)がある
  - そのデータがキャッシュからどかされるときに，dirty flagが立っていればメモリ上の対応する部分を書き換える
  - 2ページ前でどかされたブロック番号13186のdirty flagが立っていれば，メモリ上の対応する部分を書き換える

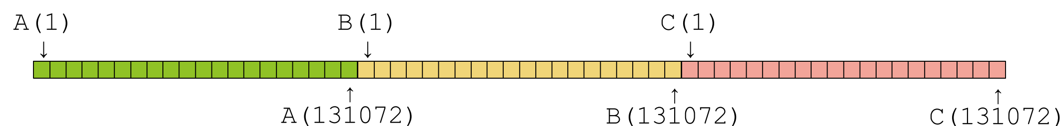
キャッシュスラッシング

キャッシュの同じラインへのデータの出し入れが頻発することをThrashingという

ソース

```
real(8) :: A(131072)
real(8) :: B(131072)
real(8) :: C(130172)

DO i=1, 131072
   tmp1=A(i)
   tmp2=B(i)
   tmp3=C(i)
   ・・・
ENDDO
```

メモリ配置



先頭アドレス

```
LOC( A ) = 1048576
LOC( B ) = 2097152
LOC( C ) = 3145728
```

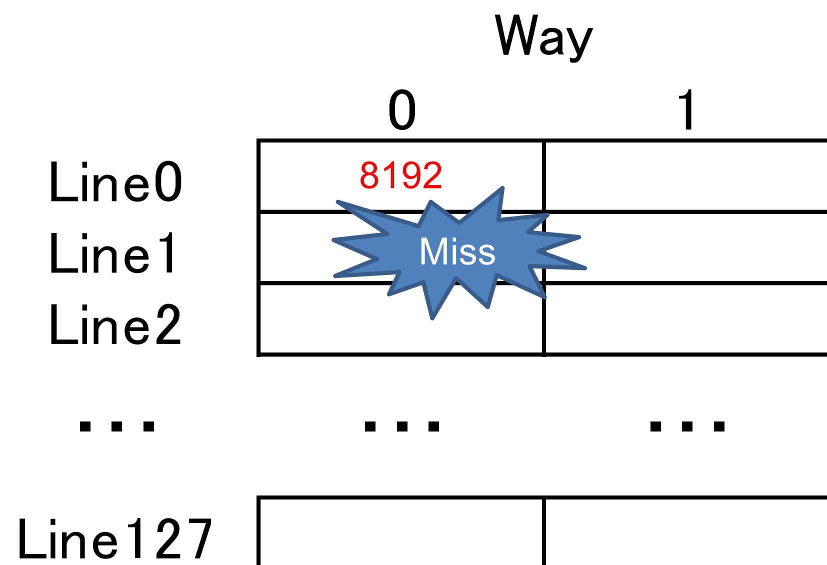
この例でループ変数iを追っかけてみよう

キャッシュの構造とラインアクセス

ループがi=1のとき

配列	アドレス	128B単位の ブロック番号	ライン番号	レジスタ
A(1)	1048576	8192	0	A(1)
B(1)	2097152	16384	0	B(1)
C(1)	3145728	24576	0	C(1)

```
DO i=1, 131072
   tmp1=A(1)
   tmp2=B(1)
   tmp3=C(1)
   ・・・
ENDDO
```

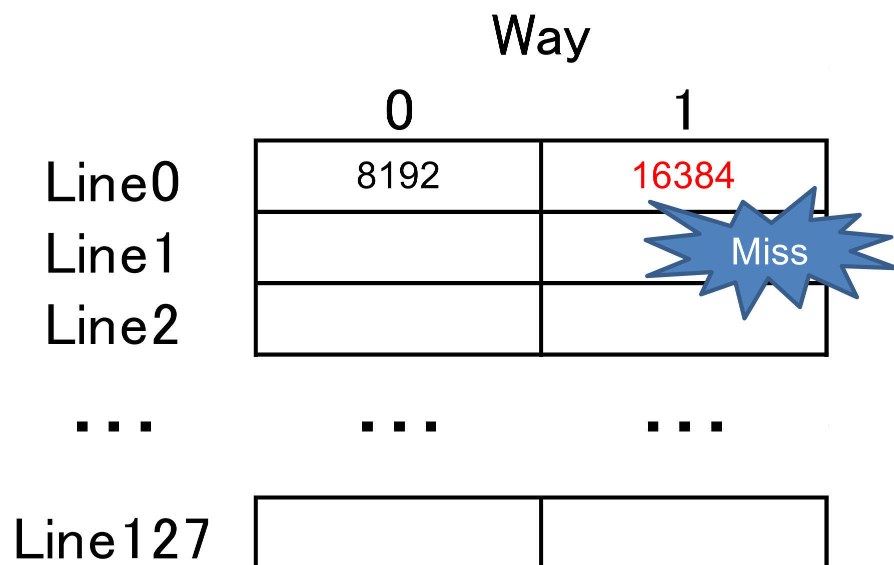


キャッシュの構造とラインアクセス

ループがi=1のとき

配列	アドレス	128B単位の ブロック番号	ライン番号	レジスタ
A(1)	1048576	8192	0	A(1)
B(1)	2097152	16384	0	B(1)
C(1)	3145728	24576	0	C(1)

```
DO i=1, 131072
   tmp1=A(1)
   tmp2=B(1)
   tmp3=C(1)
   ・・・
ENDDO
```

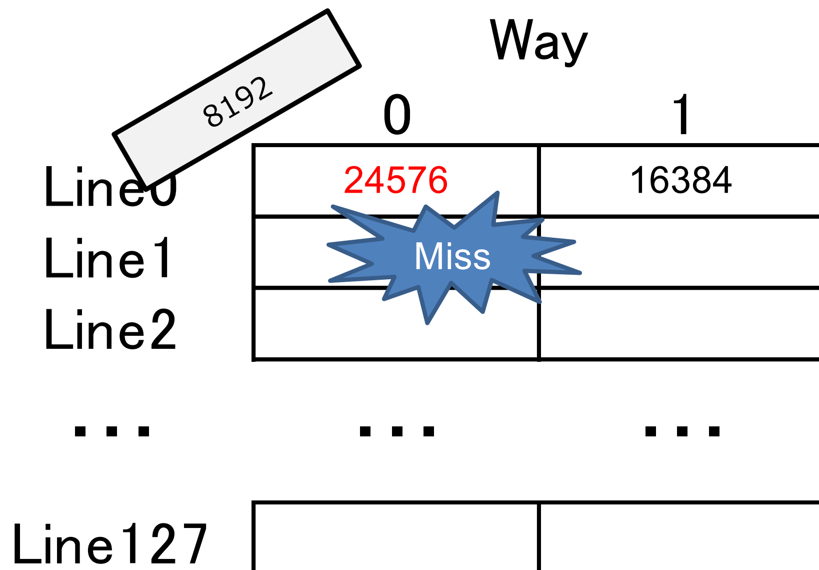


キャッシュの構造とラインアクセス

ループがi=1のとき

配列	アドレス	128B単位の ブロック番号	ライン番号	レジスタ
A(1)	1048576	8192	0	A(1)
B(1)	2097152	16384	0	B(1)
C(1)	3145728	24576	0	C(1)

```
DO i=1, 131072
   tmp1=A(1)
   tmp2=B(1)
   tmp3=C(1)
   ...
ENDDO
```

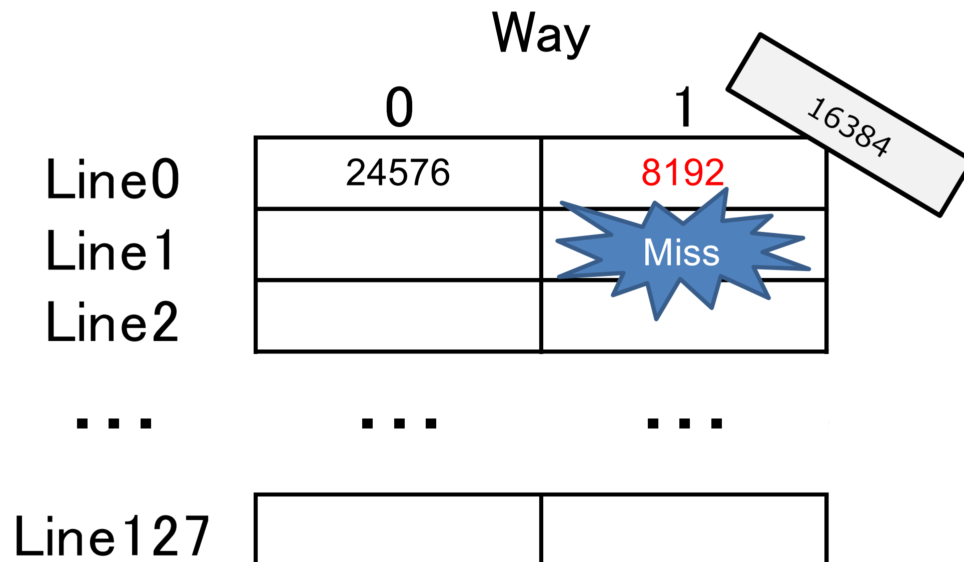


キャッシュの構造とラインアクセス

ループがi=2のとき

配列	アドレス	128B単位の ブロック番号	ライン番号	レジスタ
A(2)	1048584	8192	0	A(2)
B(2)	2097160	16384	0	B(2)
C(2)	3145736	24576	0	C(2)

```
DO i=1, 131072
   tmp1=A(2)
   tmp2=B(2)
   tmp3=C(2)
   ･･･
ENDDO
```

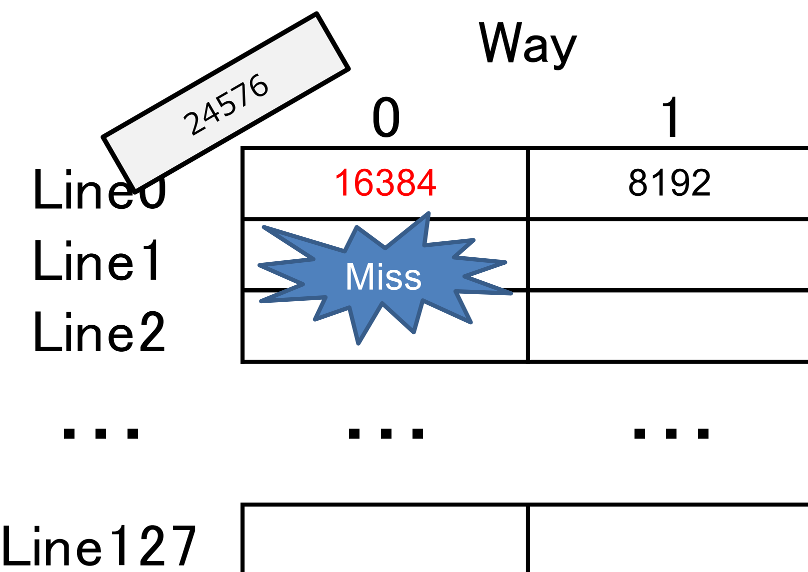


キャッシュの構造とラインアクセス

ループがi=2のとき

配列	アドレス	128B単位の ブロック番号	ライン番号	レジスタ
A(2)	1048584	8192	0	A(2)
B(2)	2097160	16384	0	B(2)
C(2)	3145736	24576	0	C(2)

```
DO i=1, 131072
   tmp1=A(2)
   tmp2=B(2)
   tmp3=C(2)
   ...
ENDDO
```

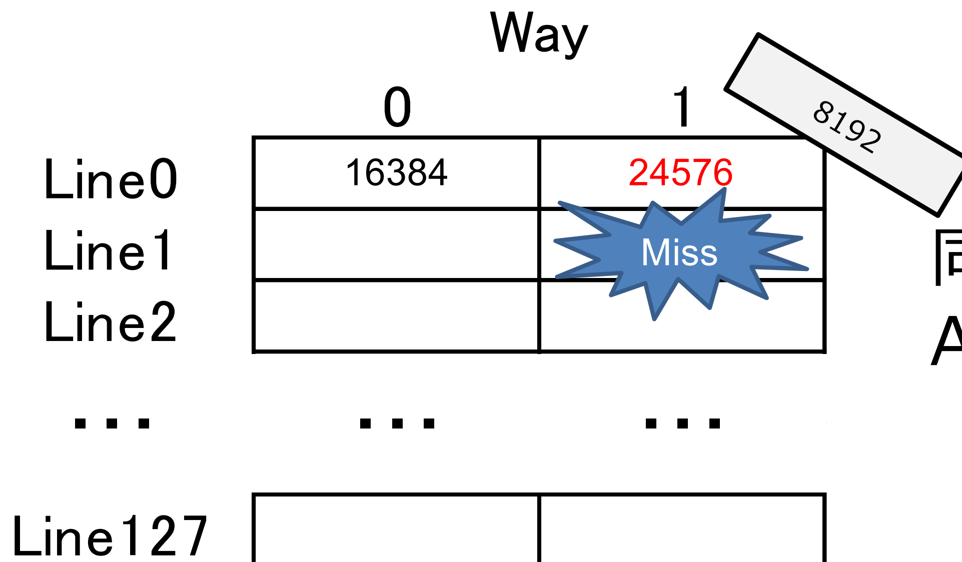


キャッシュの構造とラインアクセス

ループがi=2のとき

配列	アドレス	128B単位の ブロック番号	ライン番号	レジスタ
A(2)	1048584	8192	0	A(2)
B(2)	2097160	16384	0	B(2)
C(2)	3145736	24576	0	C(2)

```
DO i=1, 131072
   tmp1=A(2)
   tmp2=B(2)
   tmp3=C(2)
   ...
ENDDO
```



同じラインの入れ替えがひたすら発生
AとCはアクセスするたびにキャッシュミス

キャッシュスラッシング回避

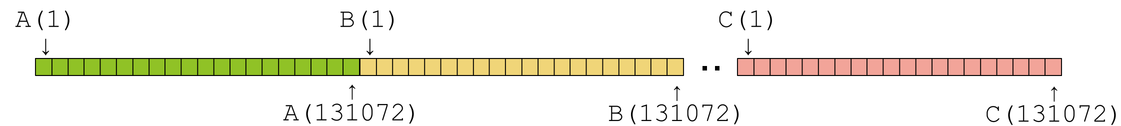
配列の先頭アドレスをずらそう

ここでは，Cだけ128バイト(1ライン分)ずらしてみる

ソース

```
real(8) :: A(131072)
real(8) :: B(131072+16)
real(8) :: C(130172)

DO i=1, 131072
   tmp1=A(i)
   tmp2=B(i)
   tmp3=C(i)
   ・・・
ENDDO
```

メモリ配置



先頭アドレス

LOC(A) = 1048576
LOC(B) = 2097152
LOC(C) = 3145856 (以前は3145728)

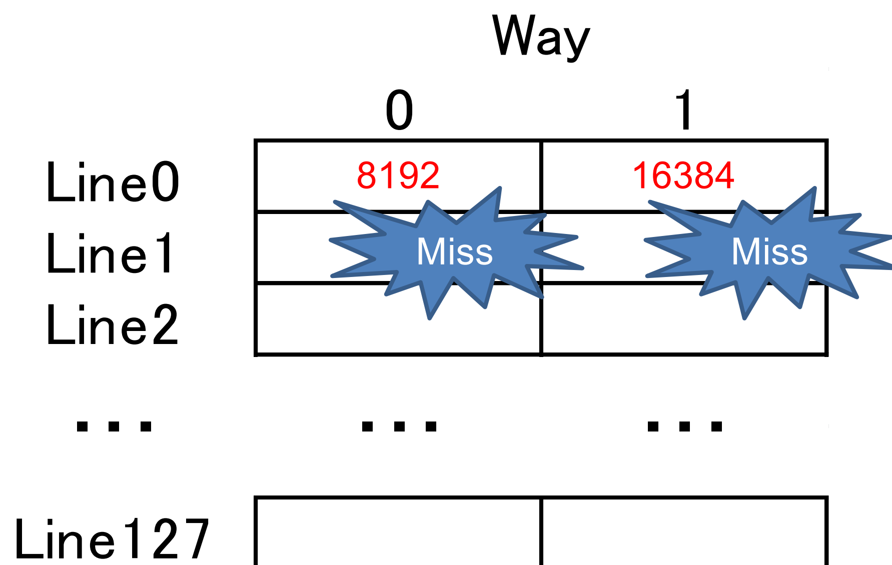
この例でループ変数iを追っかけてみよう

キャッシュの構造とラインアクセス

ループがi=1のとき

配列	アドレス	ブロック番号	128B単位の ライン番号	レジスタ
A(1)	1048576	8192	0	A(1)
B(1)	2097152	16384	0	B(1)
C(1)	3145856	24577	0	C(1)

```
DO i=1, 131072
   tmp1=A(1)
   tmp2=B(1)
   tmp3=C(1)
   ...
ENDDO
```

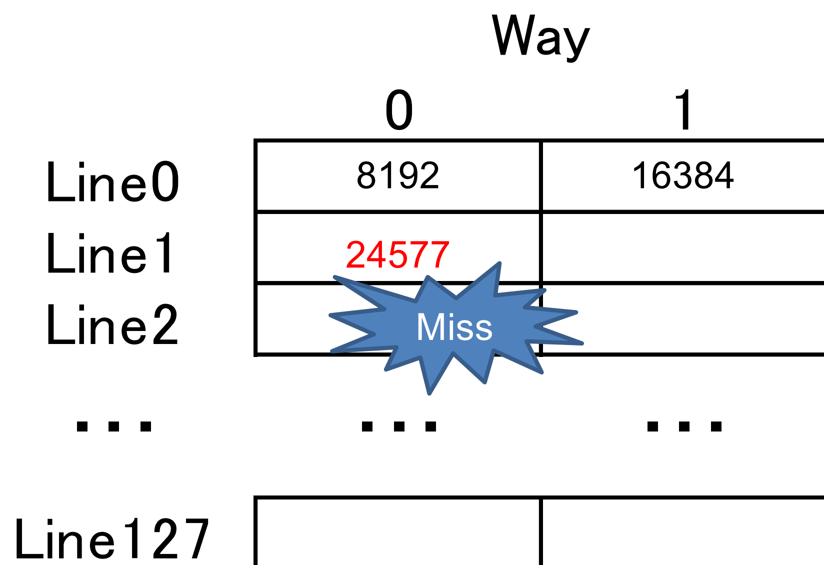


キャッシュの構造とラインアクセス

ループがi=1のとき

配列	アドレス	128B単位の ブロック番号	ライン番号	レジスタ
A(1)	1048576	8192	0	A(1)
B(1)	2097152	16384	0	B(1)
C(1)	3145856	24577	1	C(1)

```
DO i=1, 131072
   tmp1=A(1)
   tmp2=B(1)
   tmp3=C(1)
   ・・・
ENDDO
```

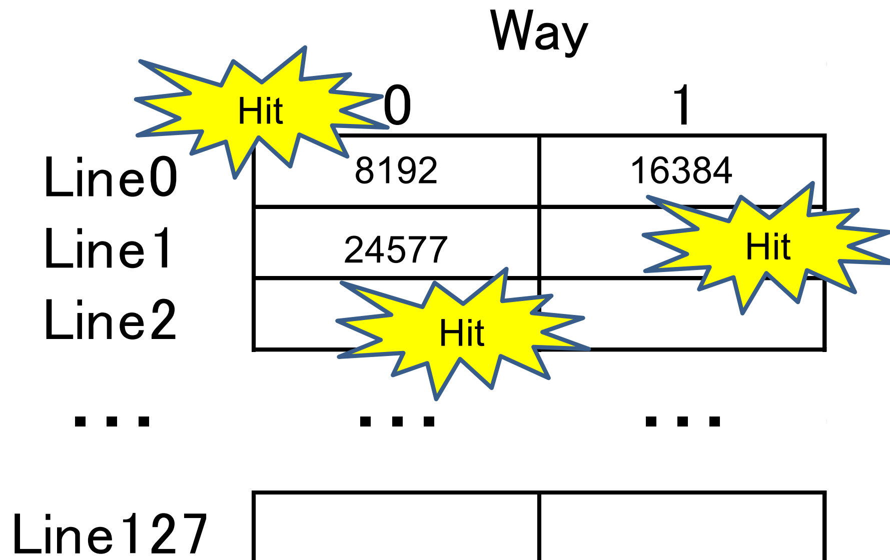


キャッシュの構造とラインアクセス

ループがi=2のとき

配列	アドレス	128B単位の ブロック番号	ライン番号	レジスタ
A(2)	1048584	8192	0	A(1)
B(2)	2097160	16384	0	B(1)
C(2)	3145864	24577	1	C(1)

```
DO i=1, 131072
   tmp1=A(1)
   tmp2=B(1)
   tmp3=C(1)
   ・・・
ENDDO
```

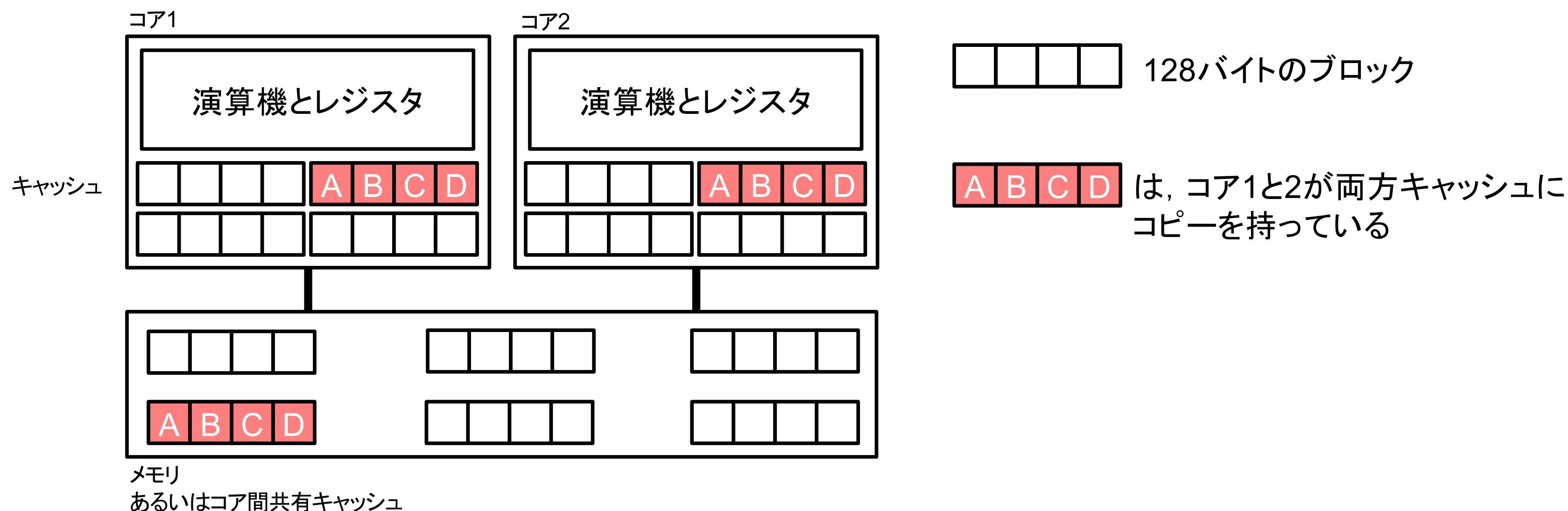


以降i=17までキャッシュミスなし

waveZ

False Sharing

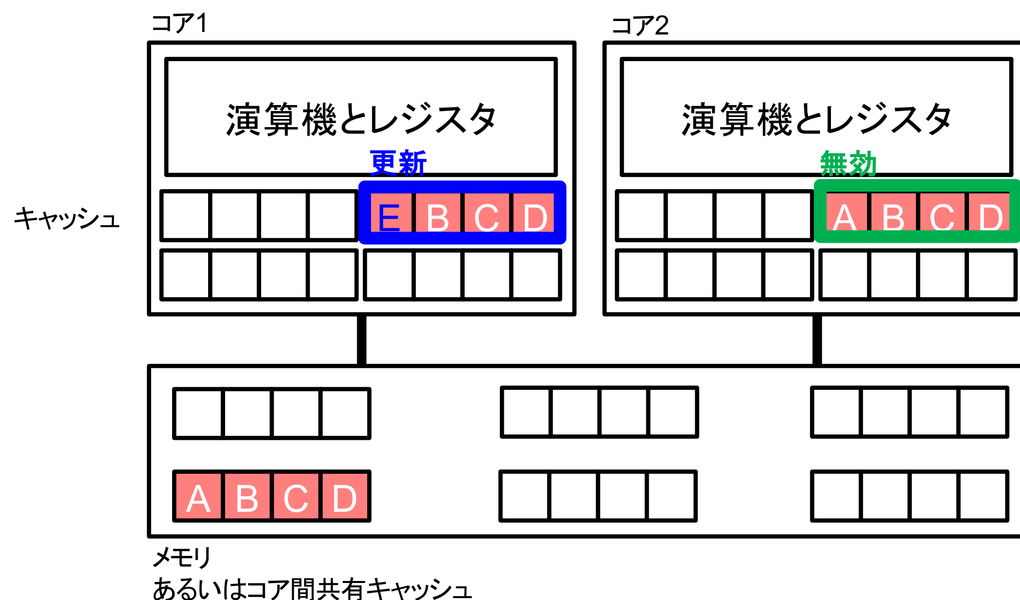
複数のコアが共有しているラインで、誰かが書き換えを行ったときに起きる現象



waveZ

False Sharing

複数のコアが共有しているラインで、誰かが書き換えを行ったときに起きる現象

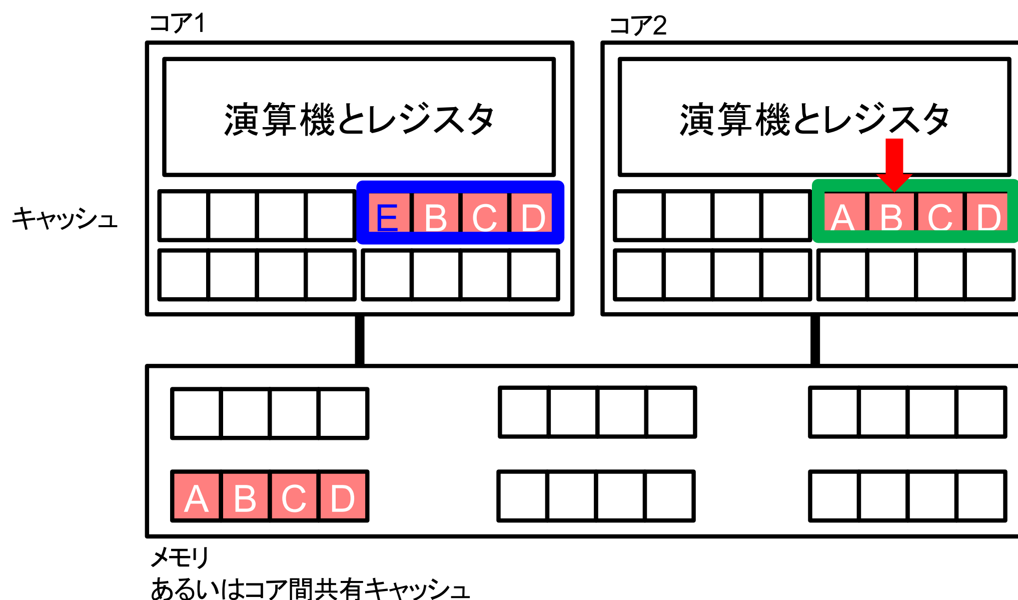


コア1がラインの内容を書き換えると
コア1では当該ラインに**更新フラグ**が
その他コアでは当該ラインに**無効フラグ**が
つく

waveZ

False Sharing

複数のコアが共有しているラインで、誰かが書き換えを行ったときに起きる現象

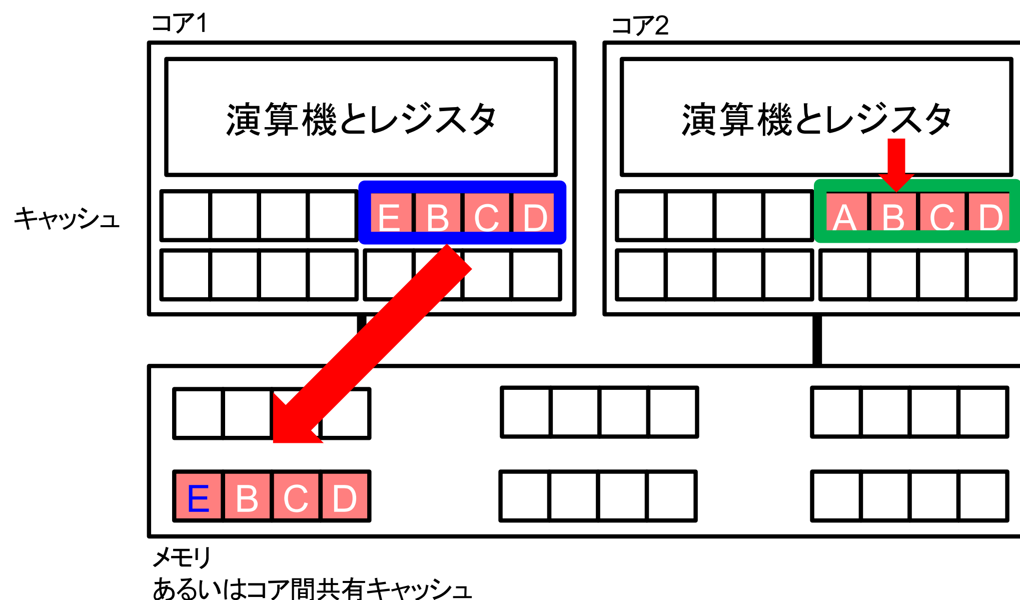


コア2がラインのBを**参照**するとき
Bが入っているラインは**無効フラグ**になっており値が変わっていることを検知

waveZ

False Sharing

複数のコアが共有しているラインで、誰かが書き換えを行ったときに起きる現象

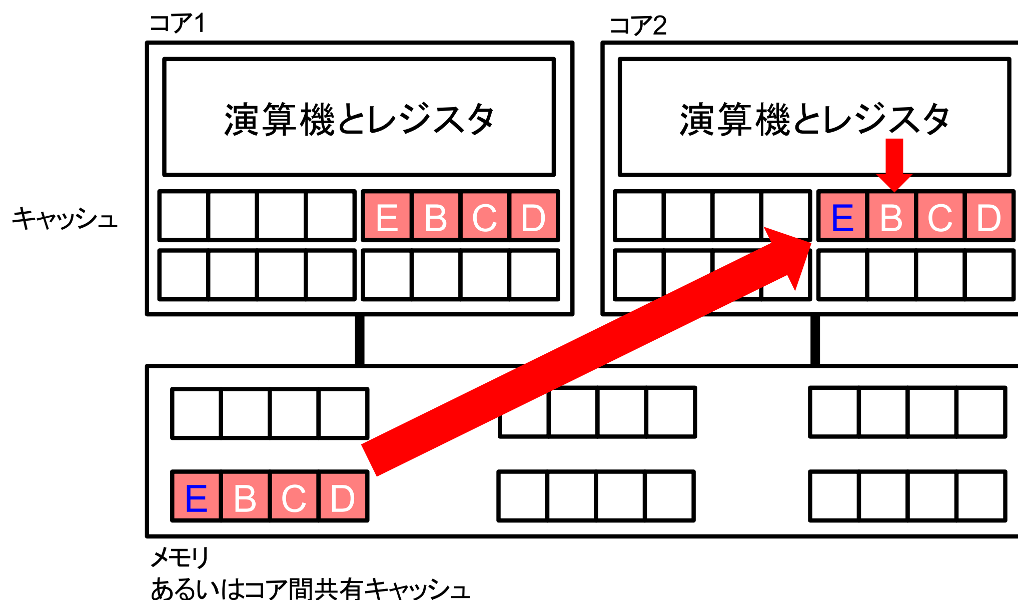


コア2がラインのBを**参照**するとき
Bが入っているラインは**無効フラグ**になっており値が変わっていることを検知

キャッシュデータの整合性を保つため，コア1のキャッシュからラインをメモリに書き戻し

False Sharing

複数のコアが共有しているラインで、誰かが書き換えを行ったときに起きる現象



コア2がラインのBを**参照**するとき
Bが入っているラインは**無効フラグ**になっており値が変わっていることを検知

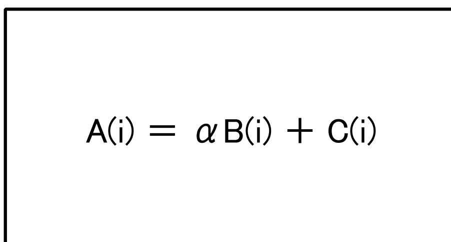
キャッシュデータの整合性を保つため，コア1のキャッシュからラインをメモリに書き戻し

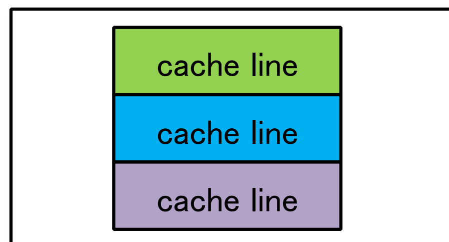
ストリームの考え方

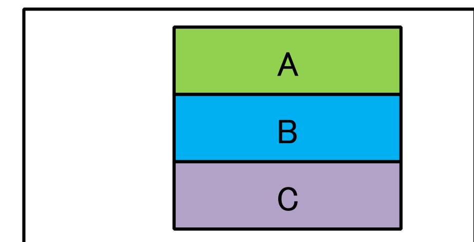
waveZ

```
do I=i, N
   A(i) = alpha * B(i) + C(i)
enddo
```

よくある計算ループ
Aには一方的に書き込まれるだけ



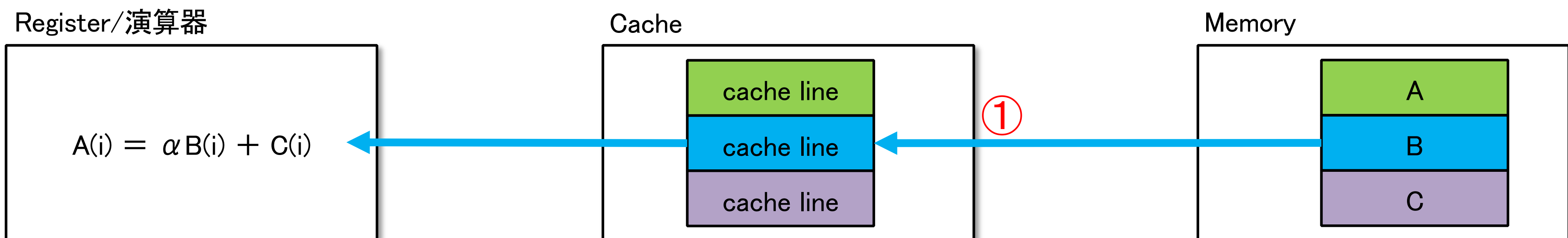




ストリームの考え方

```
do I=i, N
   A(i) = alpha * B(i) + C(i)
enddo
```

よくある計算ループ
Aには一方的に書き込まれるだけ

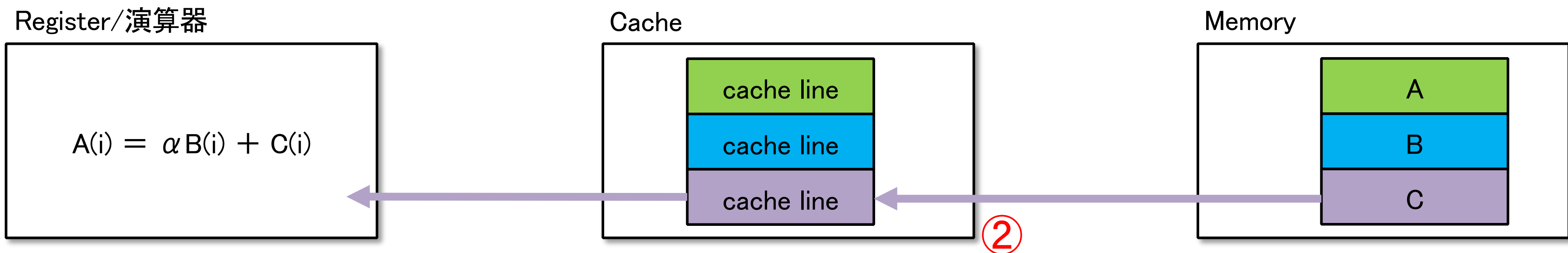


ストリームの考え方

waveZ

```
do I=i, N
   A(i) = alpha * B(i) + C(i)
enddo
```

よくある計算ループ
Aには一方的に書き込まれるだけ

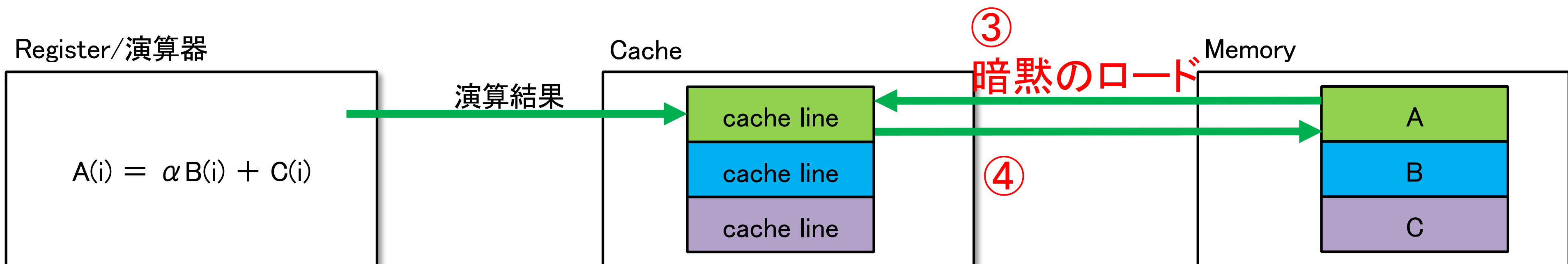


ストリームの考え方

waveZ

```
do I=i, N
   A(i) = alpha * B(i) + C(i)
enddo
```

よくある計算ループ
Aには一方的に書き込まれるだけ



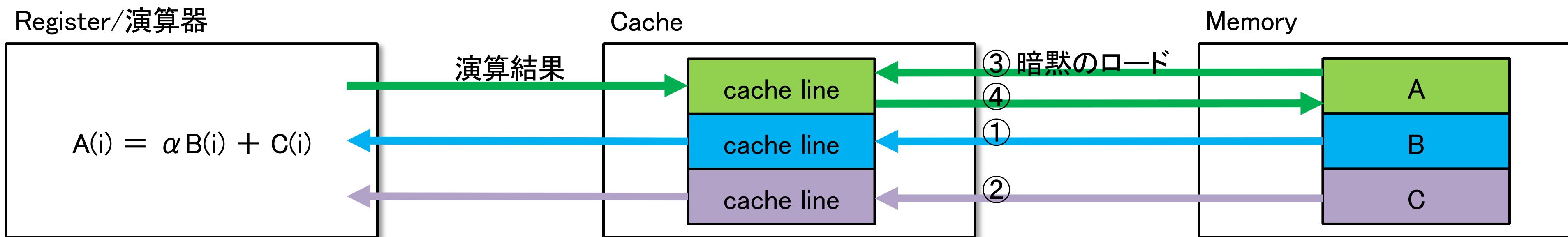
ストリームの考え方

waveZ

```
do I=i, N
   A(i) = alpha * B(i) + C(i)
enddo
```

よくある計算ループ
Aには一方的に書き込まれるだけ

メモリとのやり取り(矢印)が4本4ストリームという



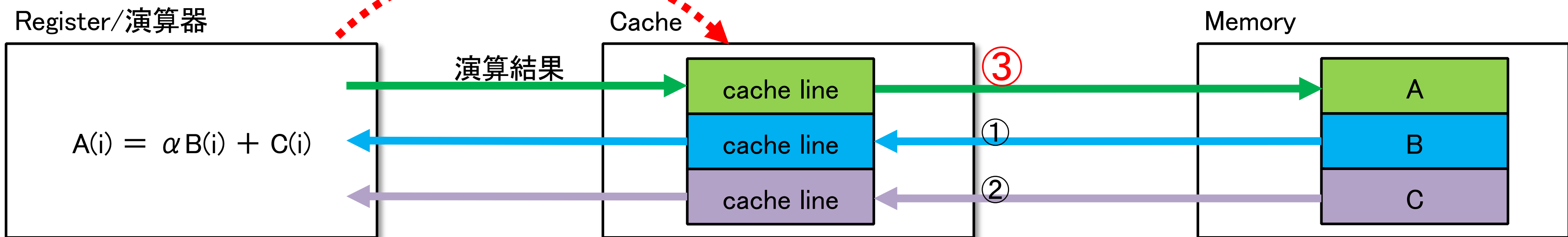
Z-Fill

waveZ

```
do I=i, N
   A(i) = alpha * B(i) + C(i)
enddo
```

よくある計算ループ
Aには一方的に書き込まれるだけ

一方的に書き込むAは一旦キャッシュに持って来なくてもよいのでは？
➡A64FXではZ-fillという機能がある（コンパイルオプション-Kzfill）



3ストリーム

Z-Fill

waveZ

zfillあり -Kzfill

キャッシュライン確保命令 ←DCZVA命令というものが実体

Register/演算器	Cache	Memory
A(i) ＝ αB(i) + C(i)	cache line	A
	cache line	B
	cache line	C

3stream

① ② ③

演算結果

zfillなし: DCZVA instruction	zfillあり: DCZVA instruction
8.00E+00	2.73E+07
8.00E+00	2.73E+07
8.00E+00	2.73E+07
8.00E+00	2.73E+07
8.00E+00	2.73E+07
8.00E+00	2.73E+07
8.00E+00	2.73E+07
8.00E+00	2.73E+07
8.00E+00	2.73E+07
8.00E+00	2.73E+07
8.00E+00	2.73E+07
8.00E+00	2.73E+07
9.60E+01	3.28E+08
9.60E+01	3.28E+08

DCZVA）
ARMv8の命令セットが持つ命令

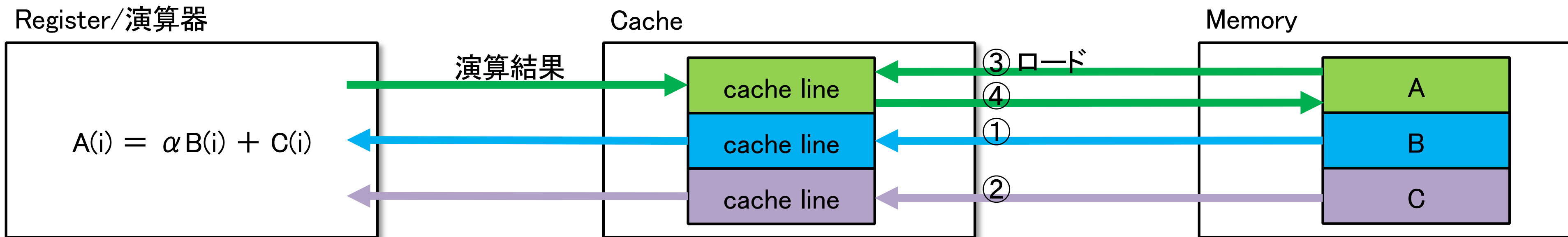
L2キャッシュに対して，指定したアドレスに対応するキャッシュラインを作り0埋めする

ストリームの考え方

waveZ

```
do I=i, N
   A(i) = A(i) + B(i) * C(i)
enddo
```

Aは書き込みだけでなく
参照もされている場合

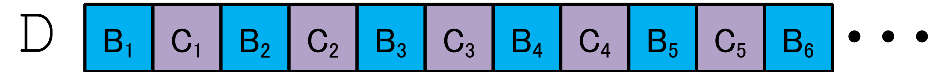


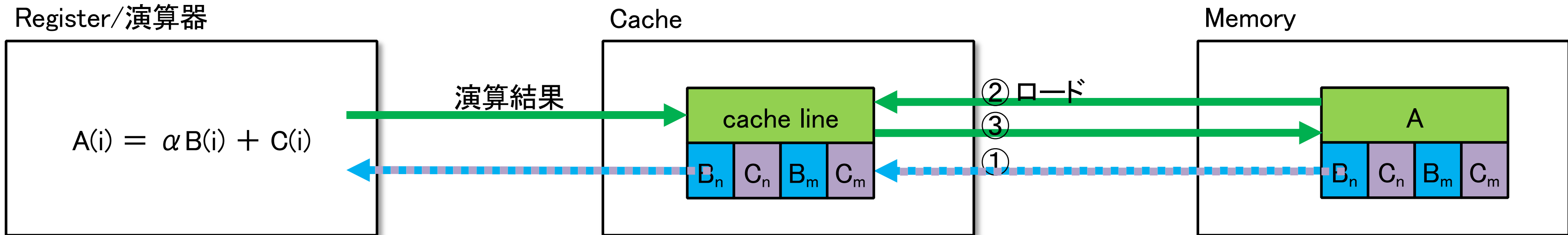
これは当然
4ストリーム必要

ストリームの考え方

wave Z

```
do I=i, N
   A(i) = A(i) + D(1,i) * D(2,i)
enddo
                  B(i)       C(i)
```

D 

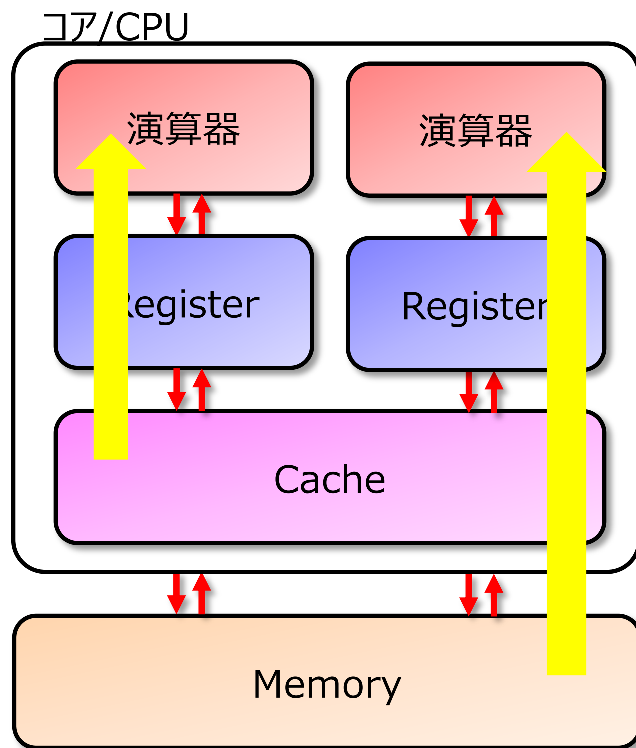


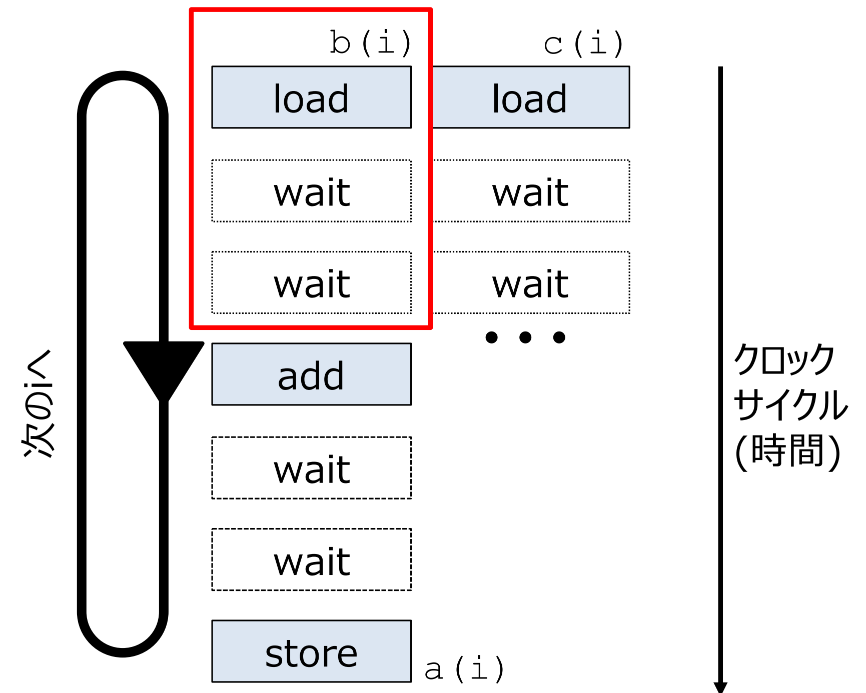
こうすると
3ストリームになる

プリフェッチ

waveZ

- ソフトウェアパイプラインはロード3に3サイクルかかるという例で解説
- 実際には，データがキャッシュになく，データをメモリまで取りにゆくと，メモリ待ちとなり，より時間がかかる

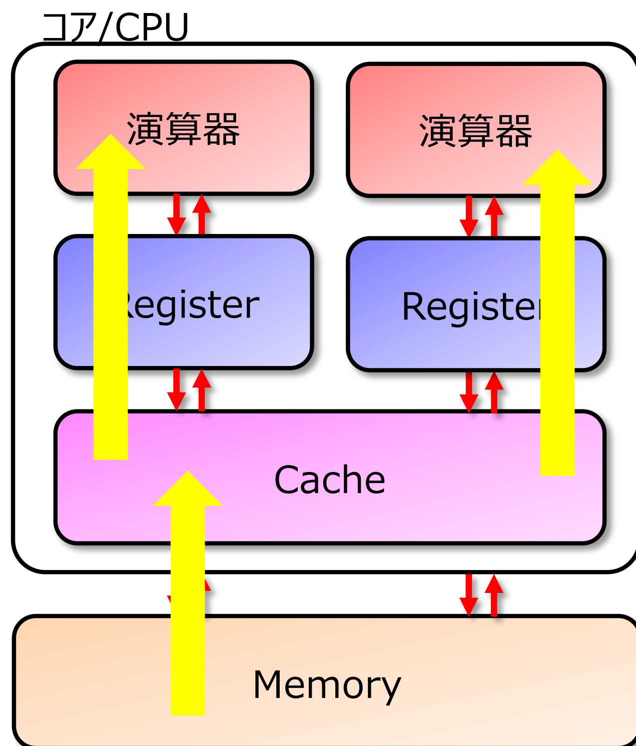




プリフェッチ

waveZ

- あらかじめ使うと分かっているデータは，ロード命令に先立ってメモリからキャッシュに持ってきておけばメモリ待ち時間はなくなり，アクセス高速化→ プリフェッチ

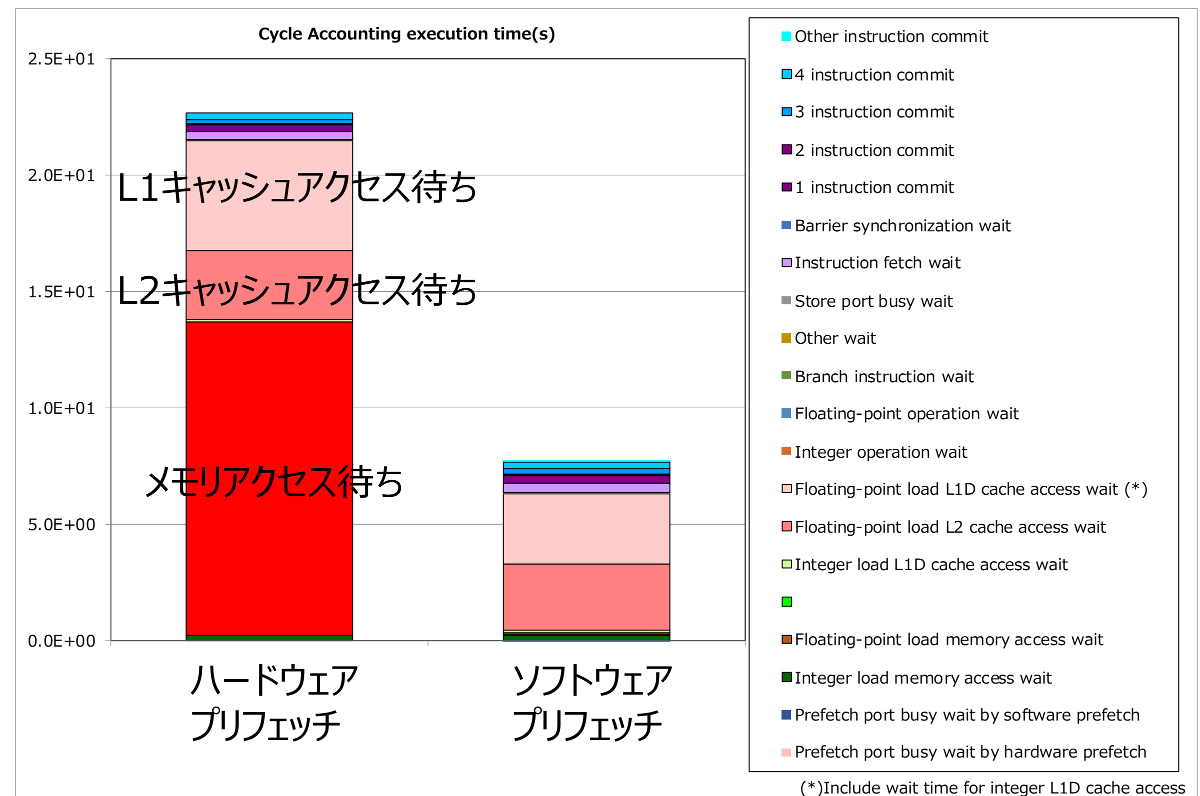


- ハードウェアプリフェッチ
  - CPUの中の専用の回路が，今までのアクセスパターンから，次にアクセスするであろうメモリ領域を予測し，キャッシュに持ってくる
  - ある程度簡単なアクセスパターン
  - 専用の回路は少ない
- ソフトウェアプリフェッチ
  - コンパイラがプログラムを解析，プリフェッチ命令をプログラム中に埋め込む
  - ある程度柔軟なアクセスパターン
  - プリフェッチ命令を入れただけできる

プリフェッチ

A64FX(富岳, FX1000, FX700)でのソフトウェアプリフェッチの利用

	HWPF	SWPF
L1D miss rate (/Load-store instruction)	67.53%	67.56%
L1D miss demand rate (%) (/L1D miss)	66.99%	4.41%
L1D miss hardware prefetch rate (%) (/L1D miss)	33.00%	32.99%
L1D miss software prefetch rate (%) (/L1D miss)	0.01%	62.60%
L2 miss rate (/Load-store instruction)	67.91%	67.98%
L2 miss demand rate (%) (/L2 miss)	44.07%	0.61%
L2 miss hardware prefetch rate (%) (/L2 miss)	59.85%	33.50%
L2 miss software prefetch rate (%) (/L2 miss)	0.00%	65.89%



まとめ

waveZ

- 単体性能最適化にまつわる、重要な概念・CPU内部で起きていることについて紹介しました
- CPUの性能を発揮するには、スレッド並列・SIMD・ソフトウェアパイプラインを適応することが必須で、それには、プロセス並列(MPI並列)と同様に、1プロセスの中でも、さまざまなレベルでの並列性を考慮する必要
- ロード・ストアを効率よく行うことも必要であり．キャッシュの動作を把握しておくことが必要
- 次回は単体性能最適化の事例とGPUのお話しをします

waveZ