

先進計算機構成論 08

東京大学大学院 情報理工学系研究科 創造情報学専攻

塩谷 亮太

shioya@ci.i.u-tokyo.ac.jp

質問や感想への回答

- hash perceptron は, $HL=0$, $HL=4$, $HL=8$, $HL=16$ からのアウトプットは一つの数という認識であっているでしょうか? そうすると, 「関係のある履歴にだけ着目する」という perceptron の考え方とはやや違う感じがしました.
- ◇ パーセプトロン予測器とハッシュパーセプトロン予測器が大分考え方が違うのはその通り
- 関係のある二つの履歴が x_1 , x_5 のように, 2つのHLテーブルにまたがる場合の挙動などはどうなるのでしょうか.
- ◇ 多分, その2つのテーブルの重みが大きくなるのだと思う

質問や感想への回答

- 前回の質問の続きですが、グローバル予測が成立するのが非直観的というのはグローバルな履歴は単射ではないからと感じたからです。(A->D->Gも00ですしB->E->Hも00なので何かおかしくなりそうだなと感じたからです)
よく考えたら実行列が長くなればそれだけ癖が出るので単射とみなせそうだなと思いました。

質問や感想への回答

- 分岐予測器を3つ以上を用い、並列的に予測をし、結果の多数決を取ることでアンサンブル効果など期待できないでしょうか？例えば Hash perceptron + BATAGE + Wormhole という感じで組み合わせるイメージです。また、単純パーセプトロンを複数並列にして、その多数決でも実現可能だと思います。どうでしょうか？
- 補足: 単純パーセプトロン同士は、例えば学習率をそれぞれ違うものにしておく（単純な勾配降下法？）、初期値を変えておく、更新のタイミング・データをズラす、などで差別化が可能のはずです。
 - ◇ アンサンブル予測器は実際結構ある
 - ◇ 多数決よりも、前回その状況で正解を言ったやつを選ぶことが一般的
 - ◇ 使用する記憶容量あたりの精度が重要だが、その点でいうと冗長性が高く（別の記憶素子で同じ事を憶えている）効率がまいいち

質問や感想への回答

- 機械学習を用いた予測器の設計というのは、分岐予測を分類問題として扱うということですか？
- ◇ まず最初に思いつく方法はそれだけど、どうだろう？

質問や感想への回答

- 一時期分岐無しプログラミングにハマっていましたが、苦勞のわりに普通に分岐させた方が速かったり、そもそも苦勞せずともコンパイラが最適化で分岐を無くしたりしてくれるので、やめました。
- ◇ はまると超速になる事もあるが、そういうケースはまれかも
- ◇ 最近の CPU は実は内部で自動でやってくれる場合がある

短距離ジャンプを行う分岐命令の条件付き命令への変換

IBM POWER8 processor core microarchitecture より

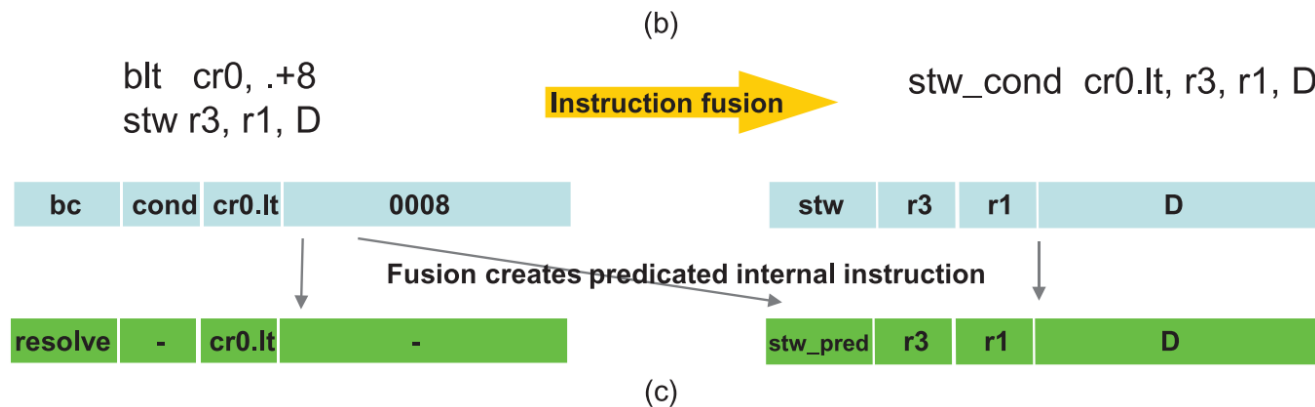


Figure 4

Instruction fusion in the POWER8 processor can be used for a variety of purposes that improve performance. (a) Two dependent instructions are transformed into two independent internal operations. (b) Two dependent instructions are transformed into a single internal operation. (c) A branch is transformed into predicated execution.

C	Assembly	Executed MicroOps
<pre> int max = 0; int maxid = -1; for (i = 0; i < n; i++) { if (x[i] >= max) { max = x[i]; maxid = i; } } </pre>	<pre> loop: lw x2, 0(a0) bge x1, x2, skip mv x1, x2 mv a1, t0 skip: addi a0, a0, 4 addi t0, t0, 0x1 j loop: </pre>	<pre> loop: lw x2, 0(a0) set.ge x1, x2 p.mv x1, x2 p.mv a1, t0 addi a0, a0, 4 addi t0, t0, 0x1 j loop: </pre>

Figure 3: Short forwards branch appearing in a loop to find the max of an array. The C source, assembly, and decoded μ OPs are shown.

質問や感想への回答

- パーセプトロン予測機の単純な構造でそれなりの予測をできることが意外でした。重みの学習が収束するにはどれほどの時間がかかるのでしょうか。
- ◇ 数回程度のアクセスで収束させないと使えない

質問や感想への回答

- 分岐予測の中でも、間接分岐の予測は難しいということを理解することができました。基本的には「前回飛んだ先に今回も飛ぶ」ということですが、このルールに従った場合どの程度予測が当たるものなのでしょうか。
- ◇ 関数ポインタや仮想関数，要素数が多い switch-case があんまり出てこない → その場合ほぼ当たる
- ◇ Python や Ruby のような言語ランタイムのインタプリタでは仮想命令の処理に間接分岐がめっちゃ使われる
 - Threading と呼ばれるプログラムの書き方を使うと，上記の方法でも結構あたるようになって性能があがる

質問や感想への回答

- 関節分岐の予測は難しいということだったのですが、if-else ifがどれくらいになったら関節分岐にするといいみたいなものってあるのでしょうか。3-4個だと関節分岐にするよりif-else-ifで関数直書きの方がいいとかあるんでしょうか
- ◇ 最近結構高度な間接分岐予測器がつまれているので、大概とりあえずほっといた方が速い
- ◇ コンパイラも勝手に判断してくれる事が多い（case の個数が少ないと if-else にしてくれる）

質問や感想への回答

- コンデンサ (負の電荷) が1bitの情報を持っている。放電すると情報が実質的に消失することは理解しました。またコイルの話になってしまうのですが、コイル + コンデンサ の組み合わせにすることで、コイルのリアクタンスを使って電荷の流出スピードを軽減できると思ったのですが、似たような話がありますか？
- (通常なら Q [クーロン] 流れるところを、コイルによって電荷の流出を阻害して $0.5Q$ [クーロン] に抑え、コンデンサには $0.5Q$ [クーロン] 残る...みたいなイメージです。)"
- ◇ リアクタンスを使った話は聞いたことがないが、流出は配線やトランジスタからよりも、コンデンサそのものの全体から周囲の絶縁体を通して起きるのでちょっと難しそう

- 物理的な制約からメモリの構造がおおよそ決まってしまうということでしたが、AMDやIntelがどのように差別化しているのかが気になりました。
- ◇ 個々のメモリ（キャッシュ）の中身自体はあまり差別化されないと思う

- メモリについて授業で紹介されていましたが、ストレージクラスメモリとメインメモリの違いは何でしょうか。

- メモリへの攻撃は主にどんな目的で行われるのでしょうか。

質問や感想への回答

- メモリや回路の理解は、webサービスやシステムを作る上で直結した知識ではないと思うのですが、どのような点で重要でしょうか。
 - ◇ 直結していないのはその通りだと思う。
 - ◇ パフォーマンスが重要になってくる場合は、背景の知識として効いてくることはあると思う。

質問や感想への回答

- メモリ読み出しの時のプリチャージの仕組みを聞き逃してしまったのですが、なぜプリチャージが必要なのでしょう
- 1. ビット線に電荷をプリチャージ
- 2. アクセストランジスタを接続
- 3. メモリセルのコンデンサに電荷が
 - ある → そのまま
 - ない → ビット線にプリチャージされた電荷がコンデンサに流れ込み電圧下降
- ◇ (※ 実際はプリチャージは $\frac{1}{2}$ の電圧で行うのでちょっと違う

- 基本的にハードウェアは冷却した方が周波数を上げられたりメリットばかりだと思っていたのですが、今回の漏電速度の低下を利用したデータ盗用のようにデメリットが生じる場合は他にもあるのでしょうか？

- DRAMのコンデンサの大きさは、CMOSトランジスタと比べてどの程度の大きさなんでしょうか。

質問や感想への回答

- メモリの物理的な性質を利用するCold Boot Attackの話がとても面白かったです。
変な質問かもしれないのですが、メモリの格子状の構造を3次元にして、(配線の長さ/メモリ量)を小さくしたりすることはできないのでしょうか?”
- ◇ コンデンサの容量を縦方向に穴掘って稼いでるので、3次元方向に積むのは結構大変
- ◇ メモリのチップ自体を重ねることは最近によくある

質問や感想への回答

- 少し授業おいていかれているように感じます、すみません...

前回の内容

1. 分岐予測の続き
2. メモリ

今日の内容

1. 命令の並列実行の基本
2. データ依存
3. 静的命令スケジューリング
4. 動的命令スケジューリング

命令の並列実行

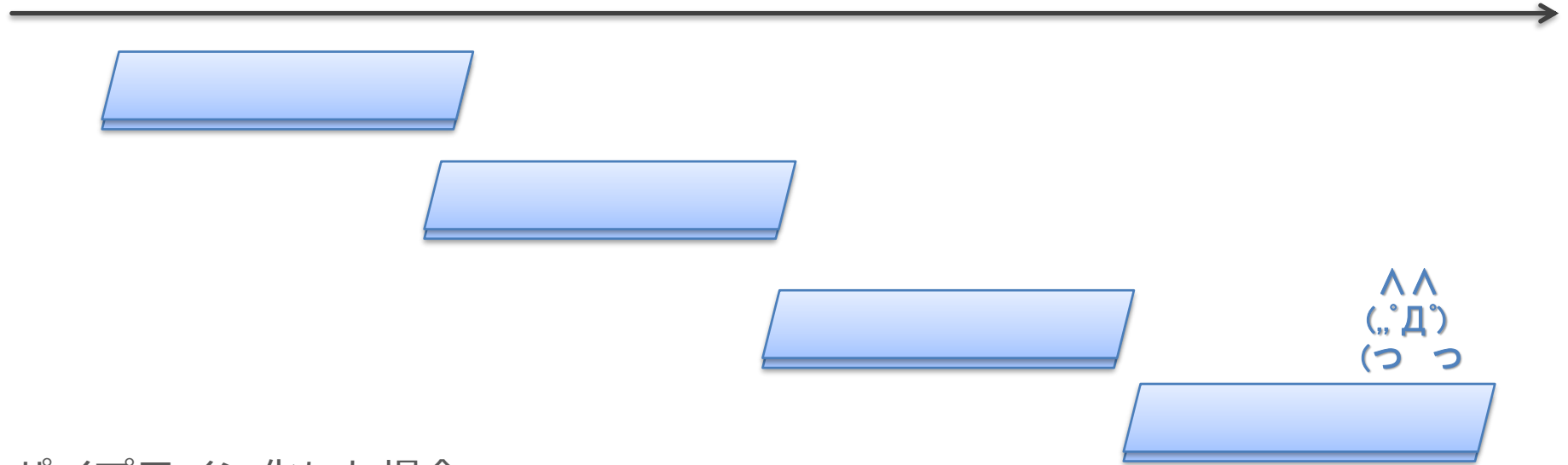
1. スカラ・プロセッサ
2. スーパスカラ・プロセッサ

スカラ・プロセッサ

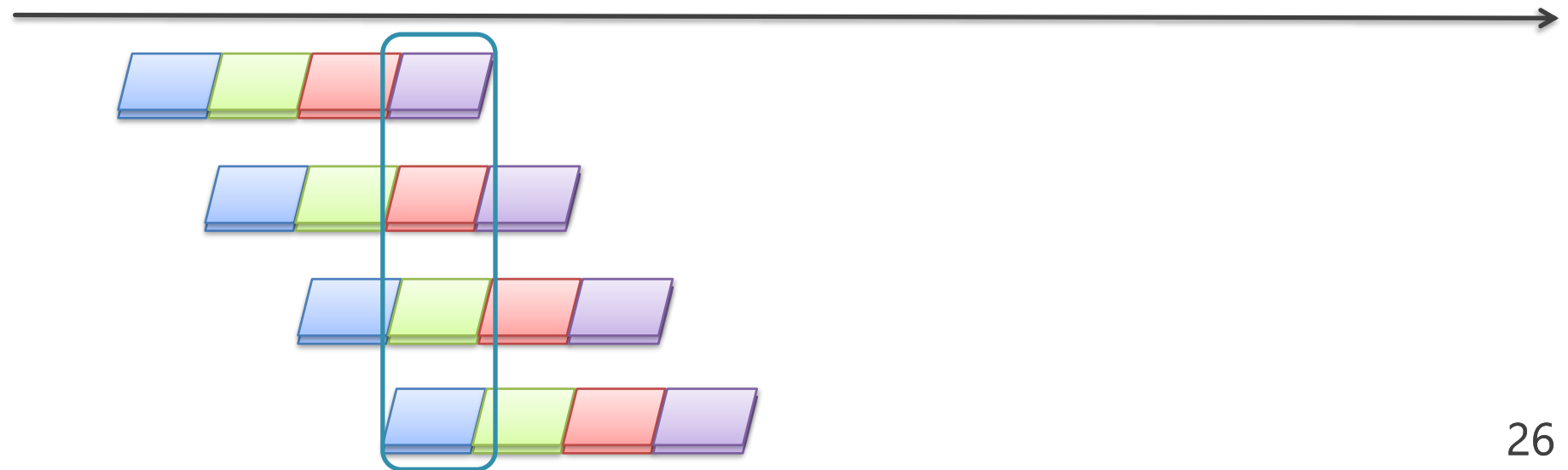
- 1 クロック・サイクルあたりに単一の命令を実行するプロセッサ
- パイプライン化：
 - ◇ 1 つの命令に関わる処理を分割して、毎サイクル並列に実行
 - ◇ パイプライン化されると「単一の命令を実行」になっていない？
 - 1 クロック・サイクルあたりでみると、単一の命令を処理

パイプライン化

パイプライン化しない場合

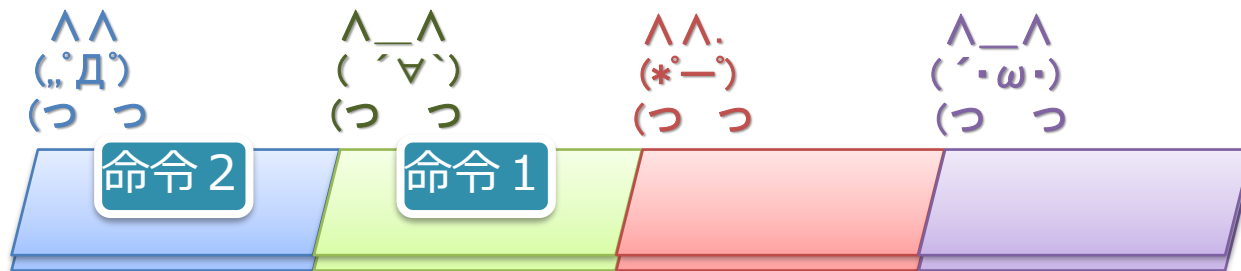


パイプライン化した場合



パイプライン化による性能向上の限界（復習）

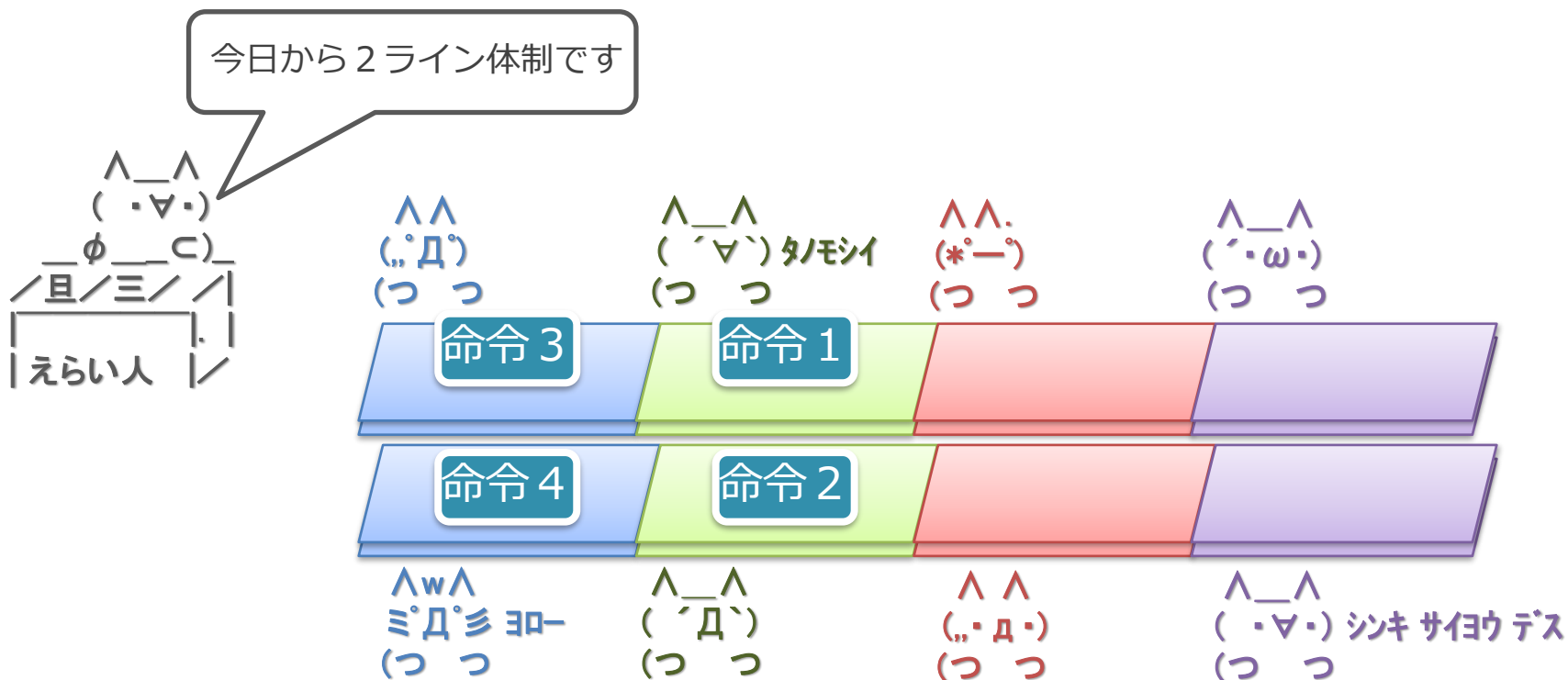
- パイプライン化による性能向上には限界がある
 - ◇ 回路的な理由による周波数向上の限界
 - D-FF の遅延
 - 電力と熱
 - ◇ アーキテクチャ的な理由による実効性能の限界
 - バックエッジによる実効性能の低下
 - （命令スケジュールを行うプロセッサ固有の性能低下もある



スーパスカラ・プロセッサ (Superscalar processor)

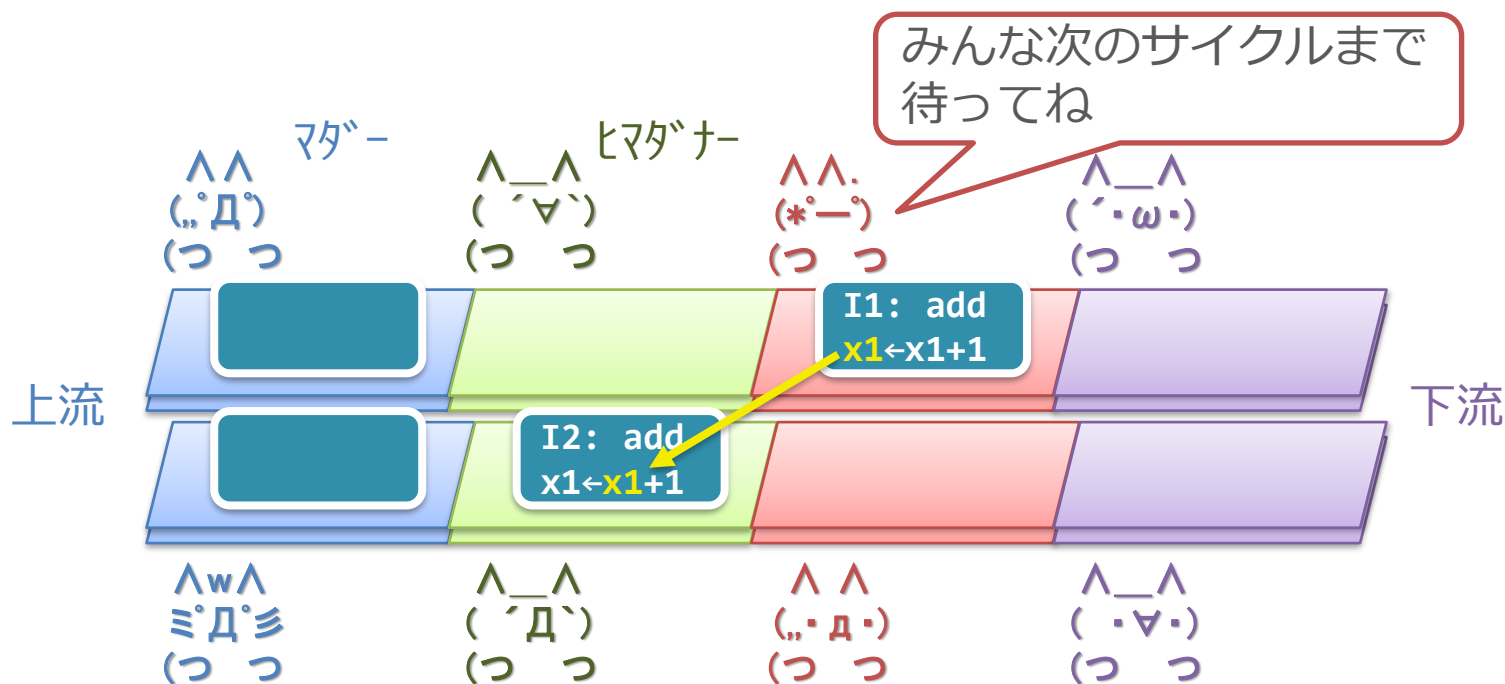
■ スーパスカラ・プロセッサ

- ◇ パイプラインや関連する演算器などを複数並べる
- ◇ 複数の命令を並行して処理して性能を向上



単純なスーパースカラ・プロセッサの動作

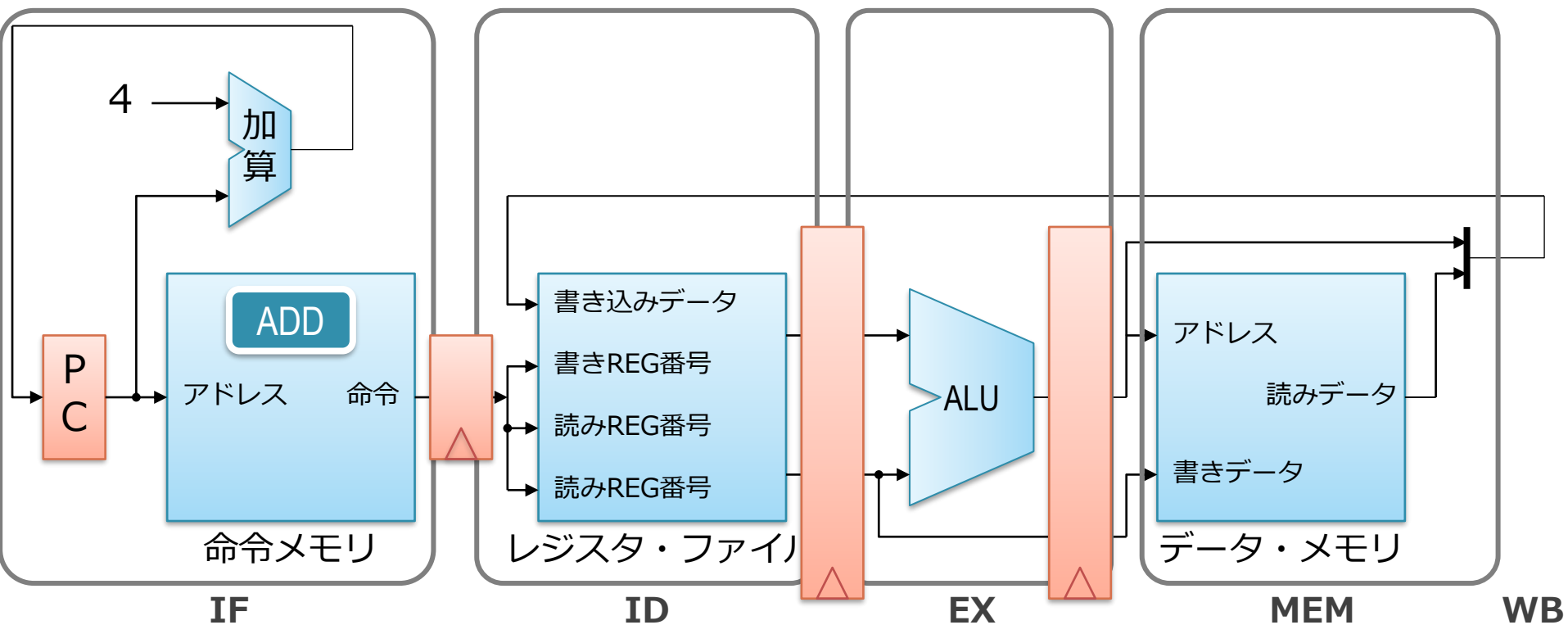
- 同時にフェッチしてきた命令間に依存がない場合は並列に実行
 - ◇ もし依存がある場合は、後続の命令全てを待たせて処理
 - パイプラインの上流側は全てストールさせる
 - ◇ プログラムの意味を保つため
 - 下の図だと I1 と I2 を並列に計算したらおかしくなる



パイプラインの復習

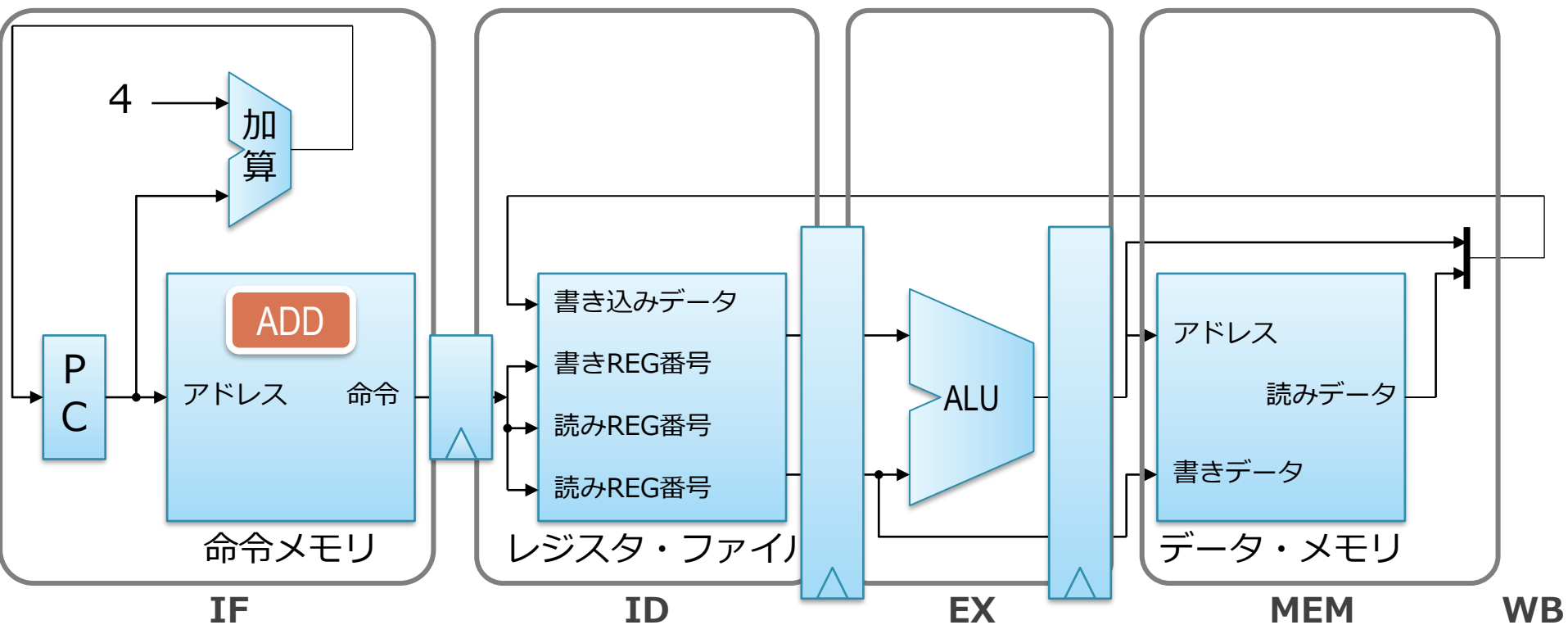
1. IF (**i**nstruction **f**etch)
2. ID (**i**nstruction **d**ecode)
3. EX (**e**xecution)
4. MEM (**m**emory)
5. WB (**w**rite **b**ack)

パイプライン化されたプロセッサのブロック図



- ◇ 各ステージの間に, D-FF (オレンジの四角) を入れる
 - WB の書き込みについては, レジスタ・ファイル自体がクロックに同期して書き込みが行われるので D-FF は不要
- ◇ 各ステージの処理が早く終わっても, 次のクロックまでは D-FF で信号の伝搬は止まる

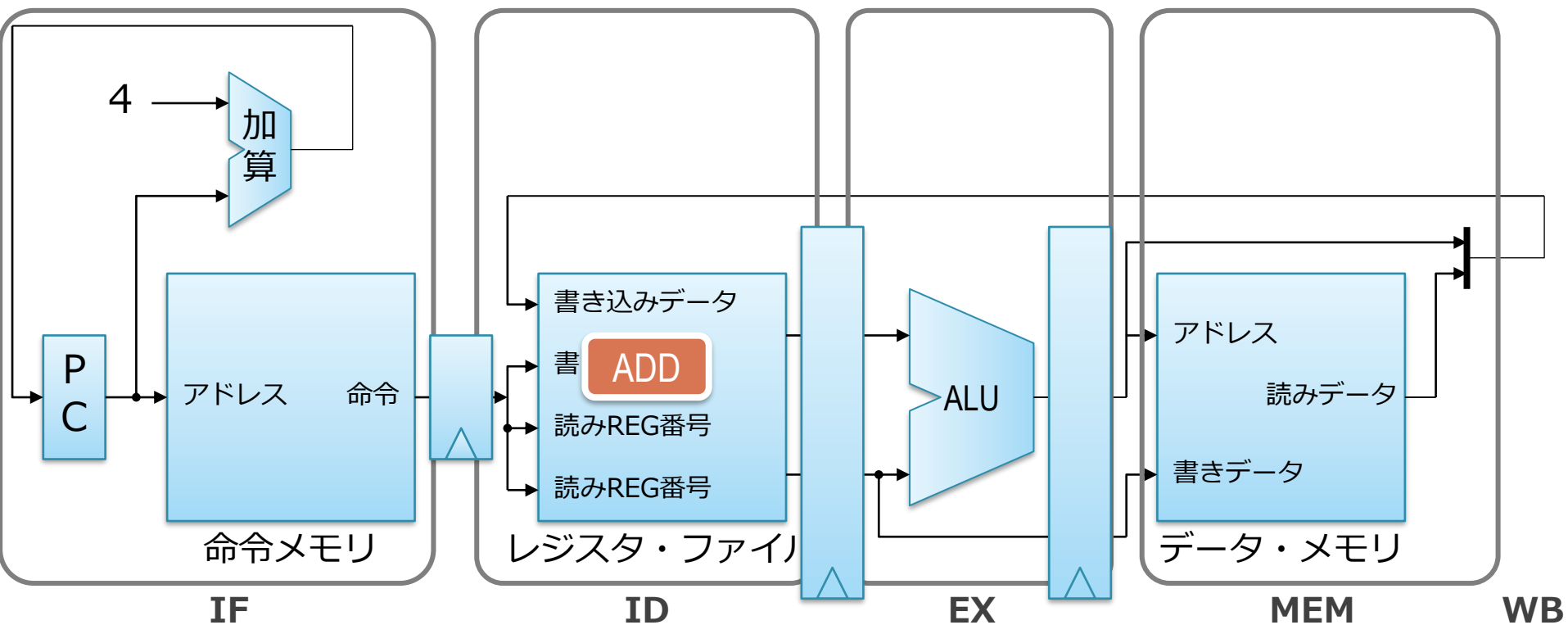
フェッチ



■ IF (instruction fetch)

◇ 命令をメモリから取り出す（フェッチするという）

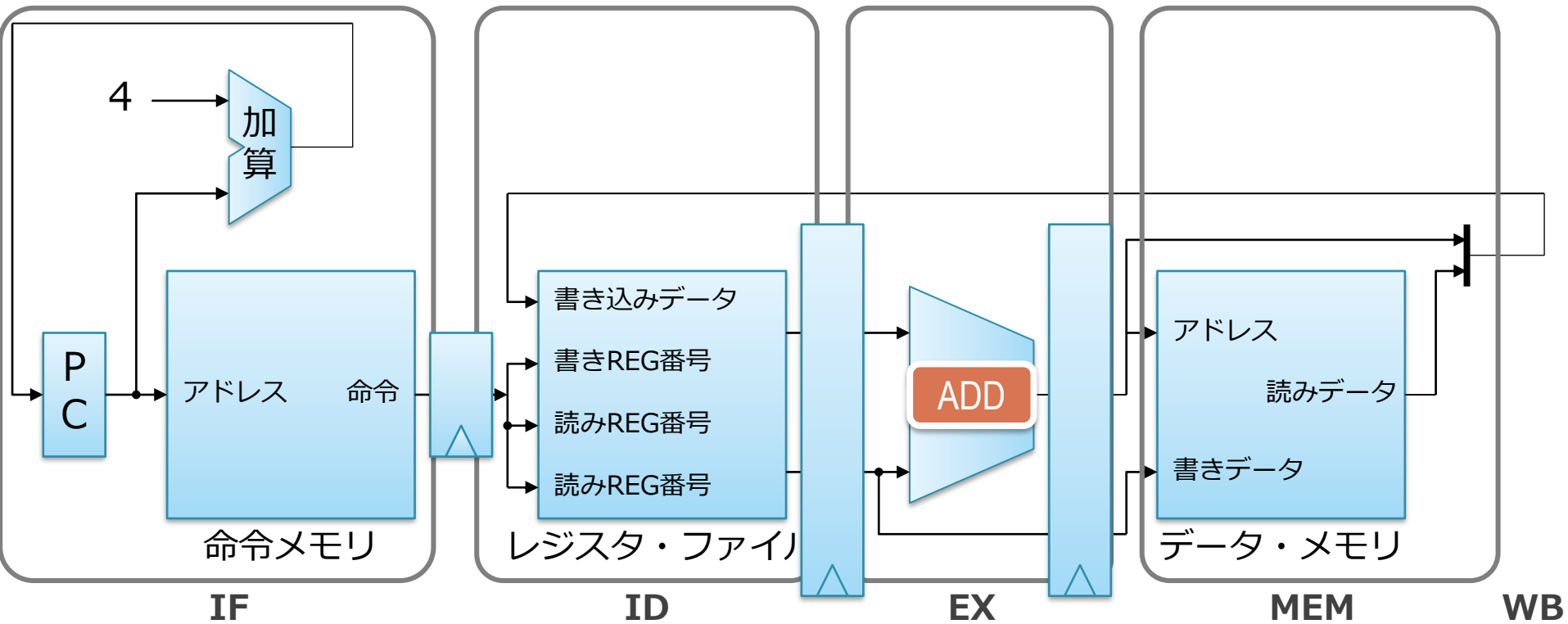
デコード



■ ID (instruction **d**ecode)

- ◇ 取り出した命令の解析（デコードという）をする
- ◇ デコードしてレジスタ番号などを取り出し、レジスタを読み出す

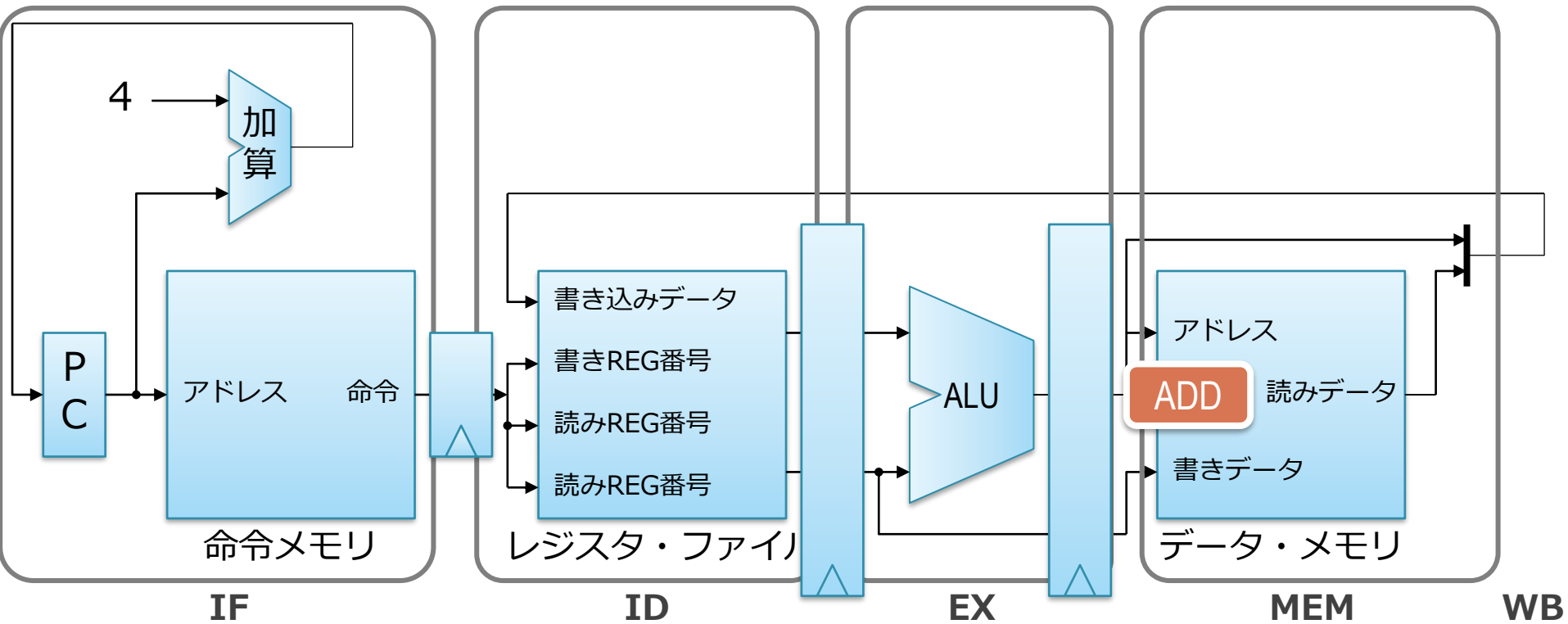
実行



■ EX (execution)

◇ 演算器で加減算や論理演算などを実行する

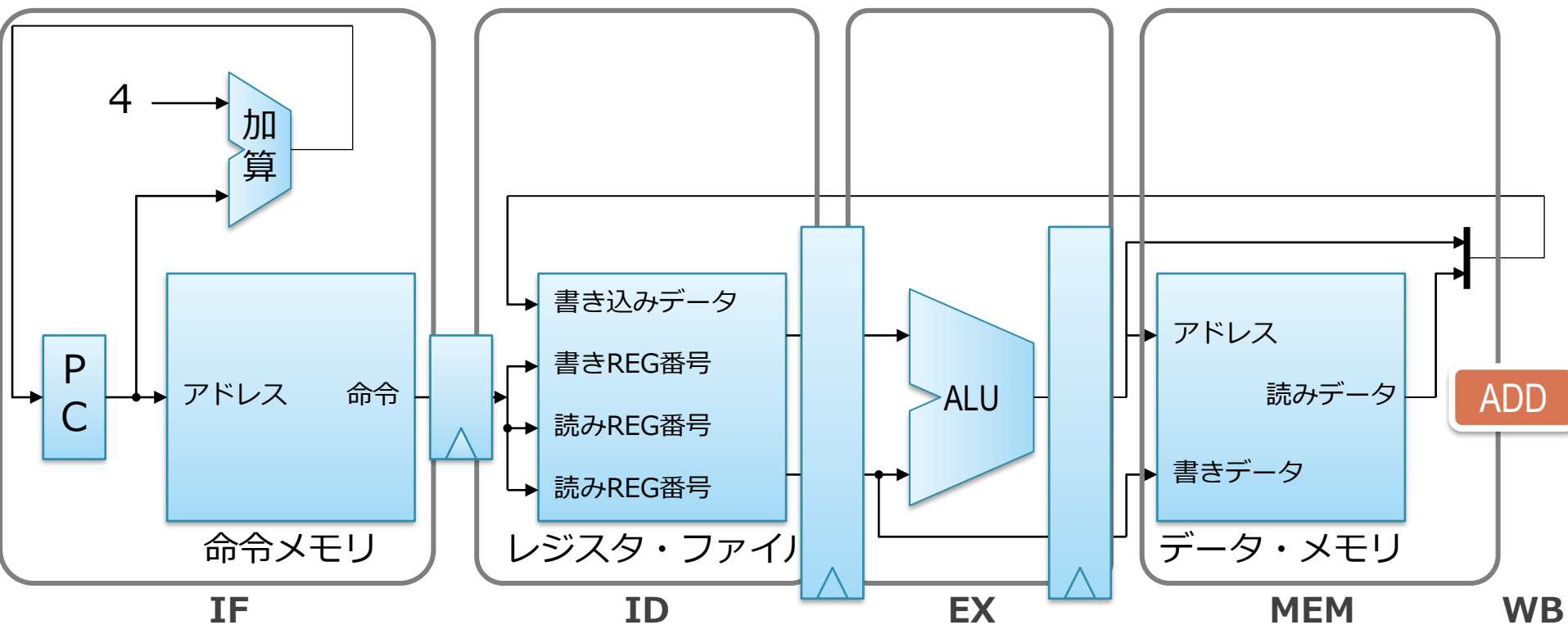
メモリアクセス



■ MEM (**m**emory)

◇ データメモリにアクセス

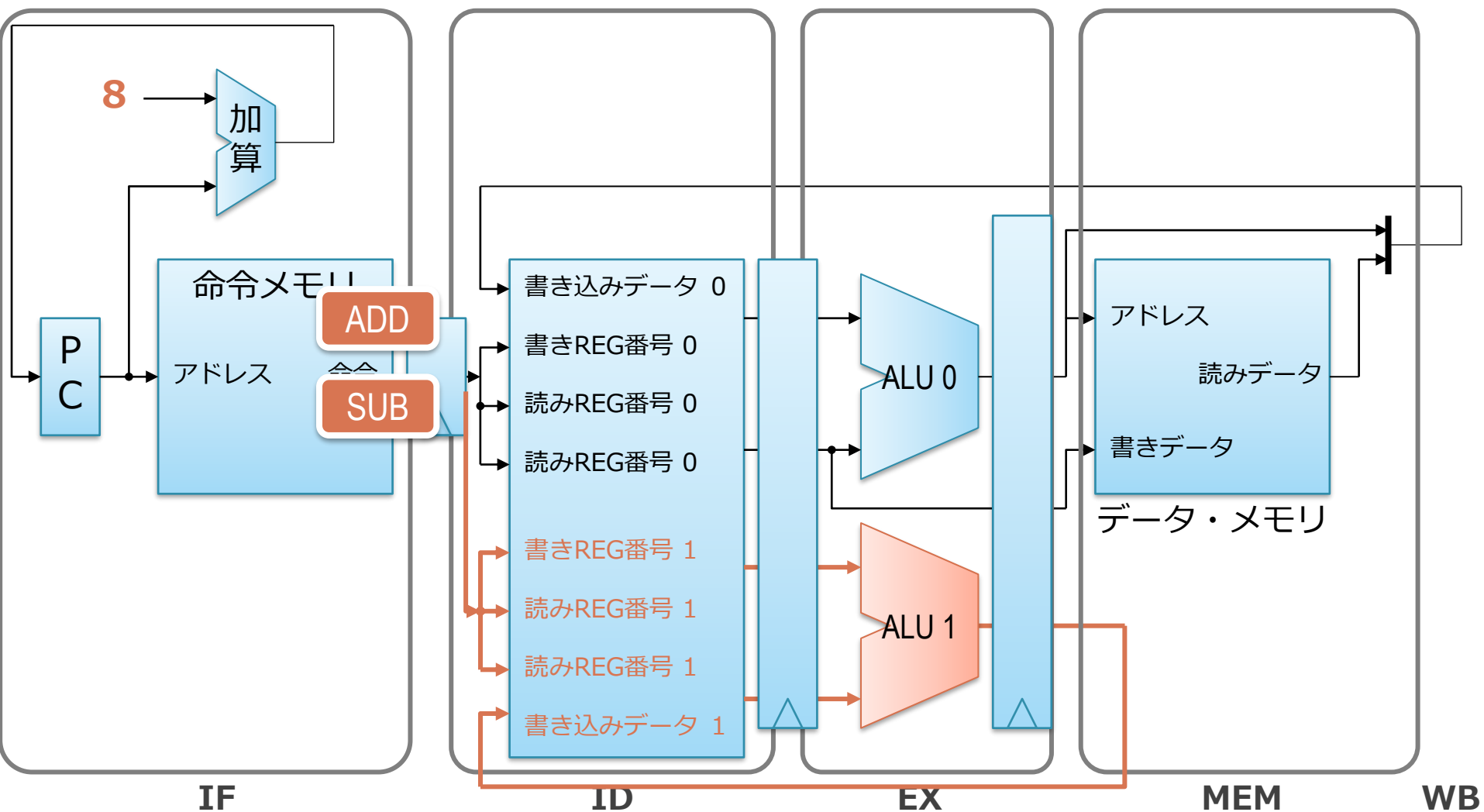
書き戻し



■ WB (**w**rite **b**ack)

◇ EX や MEM で得られた値をレジスタに書き戻す

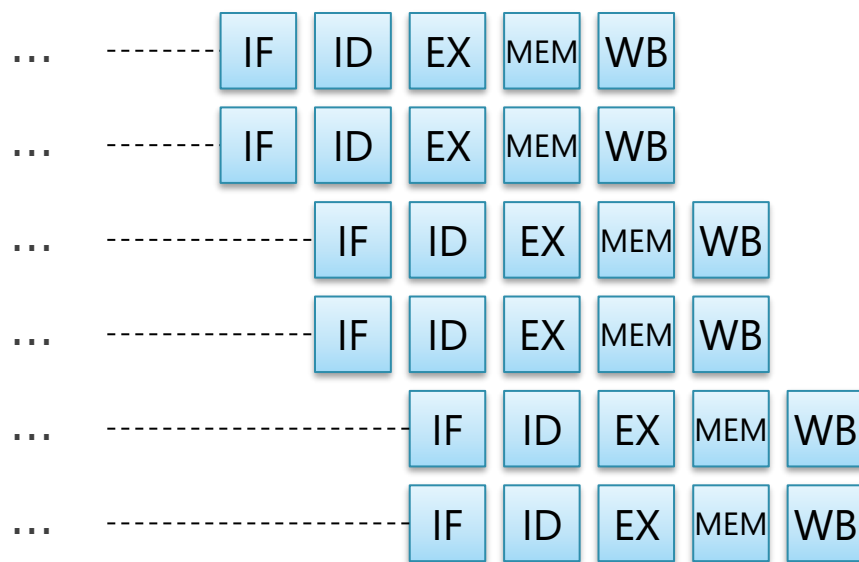
単純な 2-way スーパースカラ・プロセッサの例



- ◇ フェッチ, レジスタ・アクセス, ALU を2命令分に拡張 (赤線)
- ◇ この例では, データ・メモリは1つのまま (並列実行に制限がある)

単純なスーパースカラによる性能向上

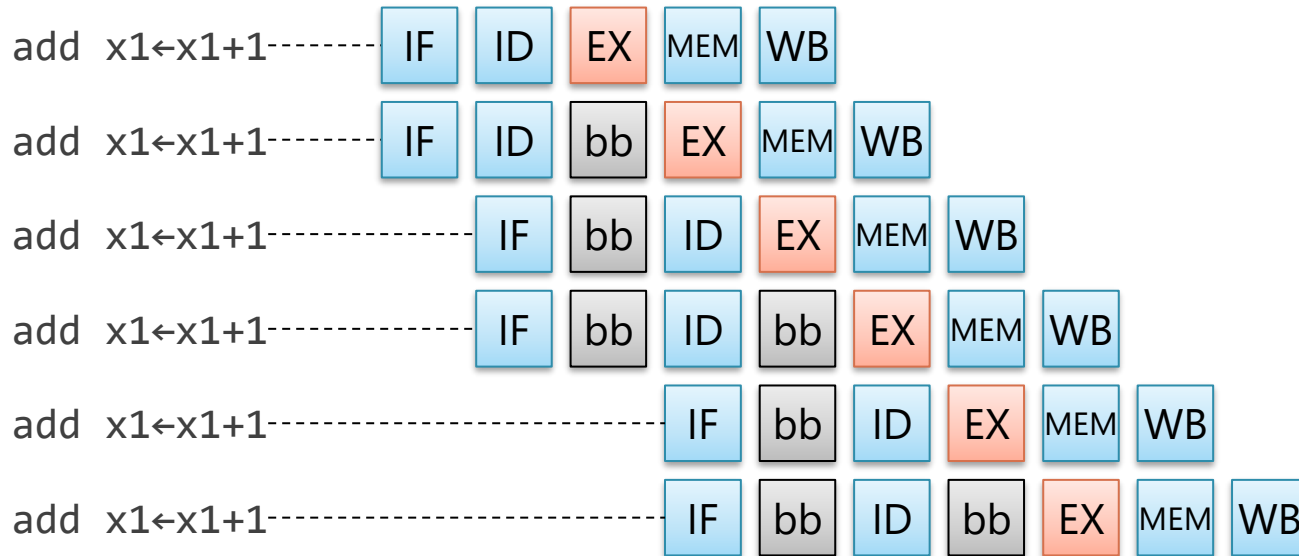
- 理想的には、並列に用意した資源の分だけ性能が向上
 - ◇ 2-way → 性能は2倍
 - ◇ 下の図は、理想的にパイプラインが回った場合



スーパースカラによる並列実行の制約

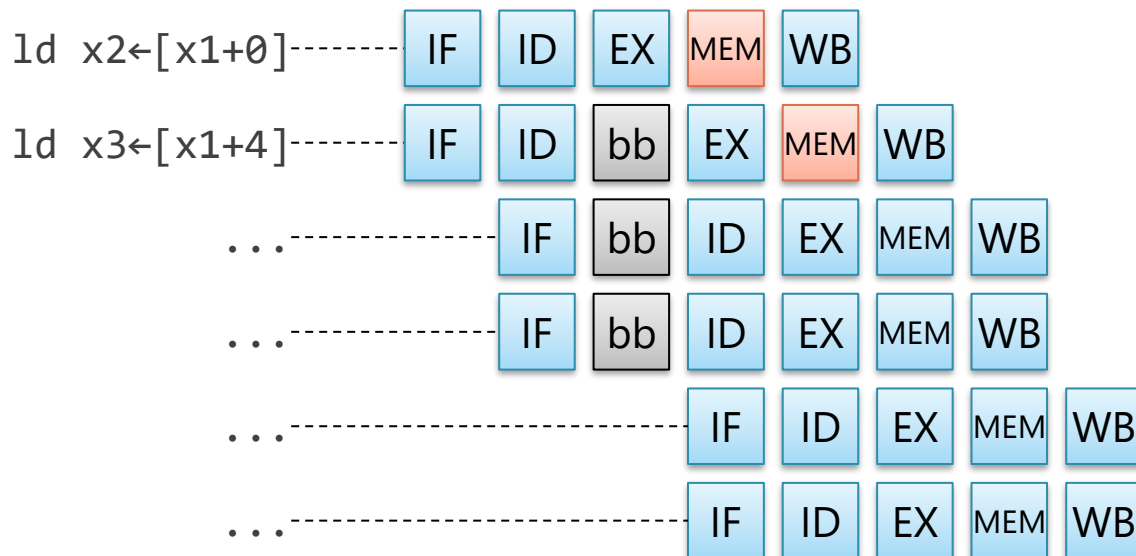
- 実際はさまざまな制約があり，そんなに性能はあがらない
 - ◇ 2-way なら数割ぐらいの向上
- 典型的な制約の例：
 1. 同時にフェッチされた命令間に依存がある場合
 2. 構造ハザードが起きる場合
 3. 同時にフェッチされた命令内に分岐があり，他に飛ぶ場合

1. 同時にフェッチされた命令間に依存がある場合



- 最悪の場合：上記のように全ての命令間に連続に依存があるとき
 - ◇ 演算が逐次的に行われるようにバブルが入る
 - ◇ スカラ・プロセッサから全く性能があがらない

2. 構造ハザードが起きる場合



- 例：先ほどのブロック図のように、メモリは1つしかない場合
 - ◇ ロード命令は1サイクルに1つしか実行できない
 - ◇ 上記のように、ロードが連続するとバブルが入る
- 回路規模が大きい & 使用頻度が低い演算器はパイプライン間で共有されることが多い = 複数同時に来ると止まる
 - ◇ 乗算器, 除算器, 超越関数の演算器など

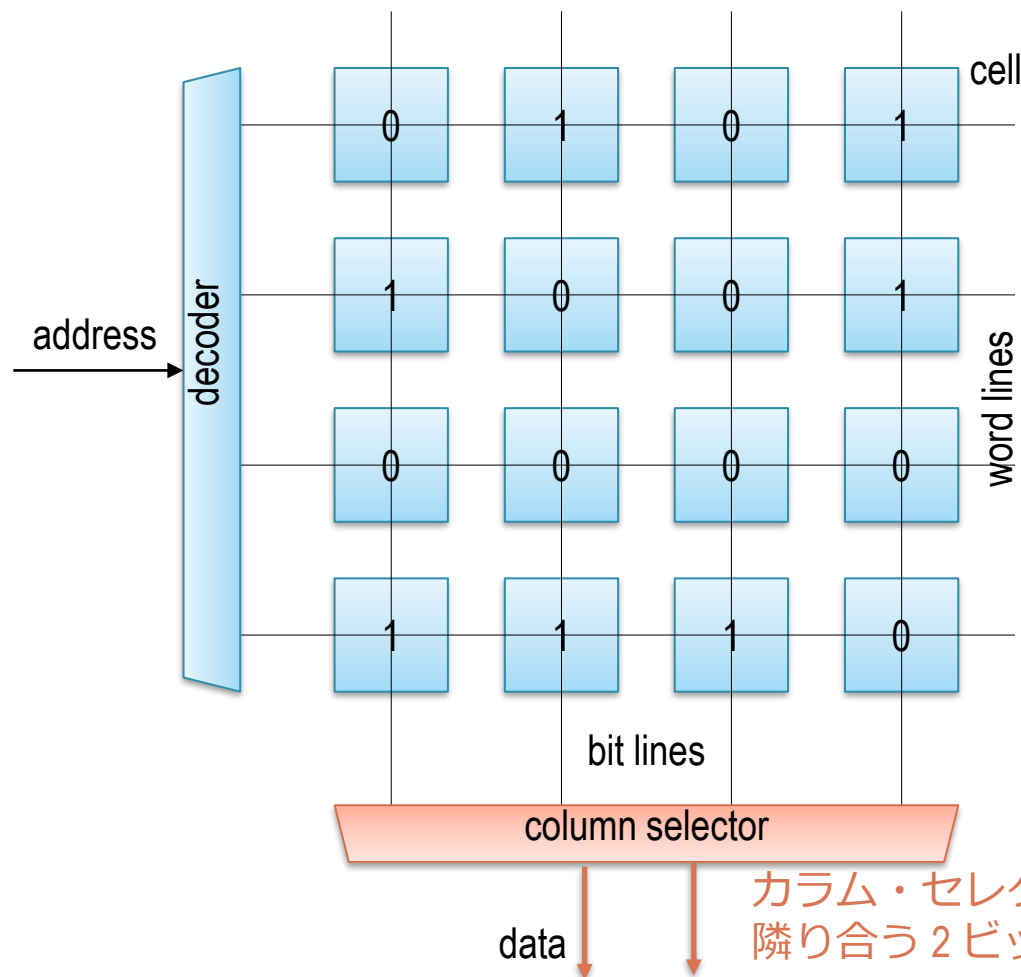
3. 同時にフェッチされた命令内に分岐があり、他に飛ぶ場合

- 1 命令目が分岐命令で成立する（と予測された）場合
 - ◇ 命令メモリが 1 ポートしかない場合，そこでフェッチが途切れる
- 例：下記のようなコードの場合

```
0x1000: bne ..., 0x1100
0x1004: ...
...
0x1100: add
```
- PC が今 0x1000 の場合，bne と add をまとめてフェッチしたい
 - ◇ そのためには，0x1000 と 0x1100 の 2 場所を読む必要がある
 - ◇ さらに，分岐予測では 2 個先までアドレスを予測する必要がある

メモリでは連続箇所（連続した命令）を一気に読むのは一般に簡単

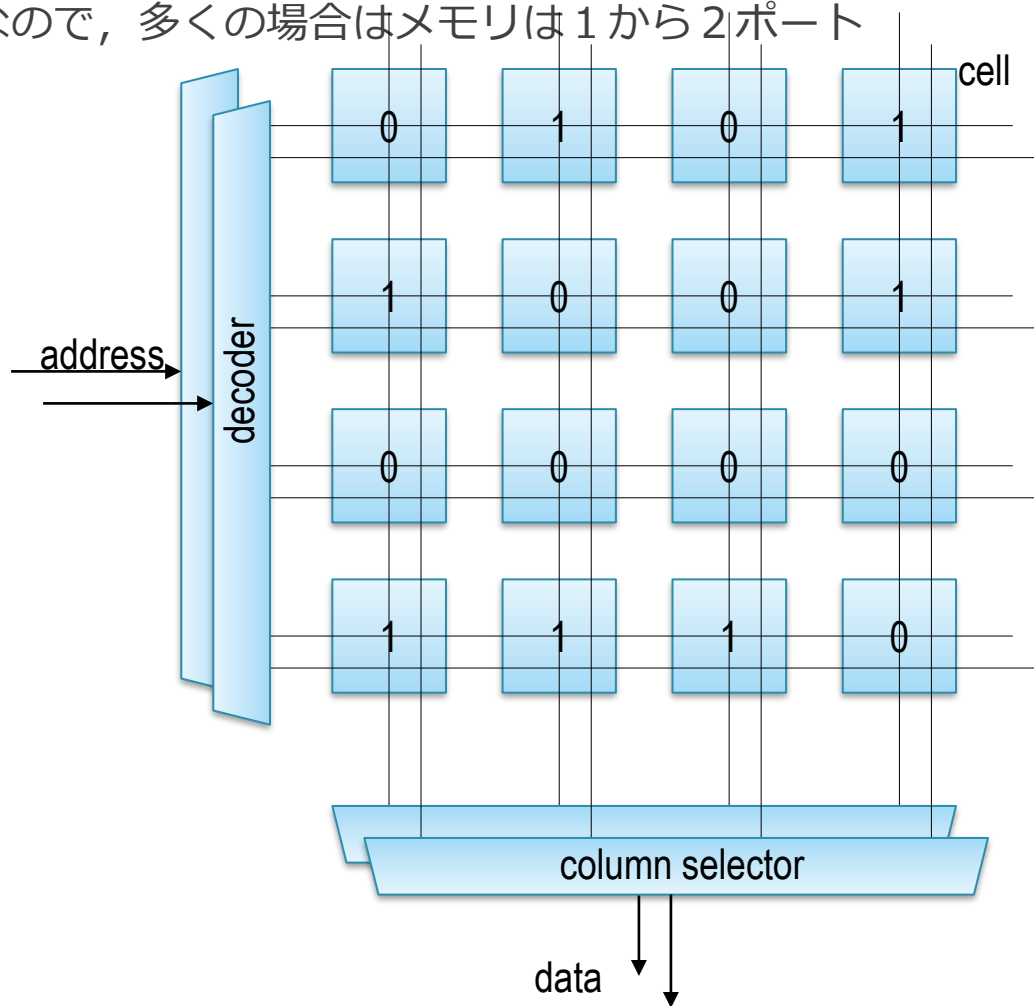
- ◇ もともと行単位で一気に読んでるため
 - カラムセクタでずらせばよい
- ◇ 他に、もっと大きな単位でマルチバンク化という方法が常に適用できる



カラム・セクタを、4ビットから隣り合う2ビットを選ぶように変更

任意の複数箇所を同時に読む = マルチポート・メモリが必要に

- ◇ 連続箇所を一気に読むのは簡単だが、独立した2カ所は大変
 - マルチポート・メモリはポート数の2乗で面積が大きくなる
 - 下の図では縦横の線の数がそれぞれ倍に
- ◇ なので、多くの場合はメモリは1から2ポート



3. 同時にフェッチされた命令内に分岐があり、他に飛ぶ場合

- 分岐をまたいでフェッチをしたい場合
 - ◇ 分岐の飛び元と飛び先の、2箇所をメモリから同時に読む必要がある
 - マルチポート・メモリが必要で回路の増大を招く
 - ◇ さらに、分岐予測では2個先までアドレスを予測する必要がある
 - 出来なくはないが、これもまた回路の増大を招く
- 実際には分岐にあたるとそこでフェッチを止めるのが普通

単純なスーパースカラによる並列実行のまとめ

- これまでに説明したような単純なスーパースカラではあまり大きな性能向上が期待できない
 - ◇ 2-way なら数割ぐらいの向上
- 同時実行幅を増やしていても、何かの制約ですぐ止まる
 - ◇ n 命令のうち 1 つでもひっかかってたらダメ

同時実行幅を増やしていても、何かの制約ですぐ止まる

■ どうする？

◇ 1. 構造ハザード

→ ユニットを増やす

◇ 2. データ依存

→ **命令スケジューリング（後述）**

◇ 3. 分岐をまたぐ場合

→ 上に比べればあまり影響がないので放置

□ 分岐命令は4命令に1回ぐらいの出現なので、4並列ぐらいまでは顕在化しにくい

余談：「スーパスカラ・プロセッサ」という言葉

- 広義の「スーパスカラ・プロセッサ」
 - ◇ パイプラインや演算器を複数備え、複数命令を同時実行できるもの
- 単に「スーパスカラ・プロセッサ」と書いた場合、
後述する「out-of-order 実行を行うスーパスカラ・プロセッサ」の意味でも使われることがある

今日の内容

1. 命令の並列実行
- 2. データ依存**
3. 静的命令スケジューリング
4. 動的命令スケジューリング

命令間の依存関係

■ 命令のスケジューリング

- ◇ プログラムの意味を変えずに、命令の実行順を並び変えること
- ◇ これによって並列に実行できる命令を増やす

■ プログラムの意味が変わらない = 依存関係をくずさない

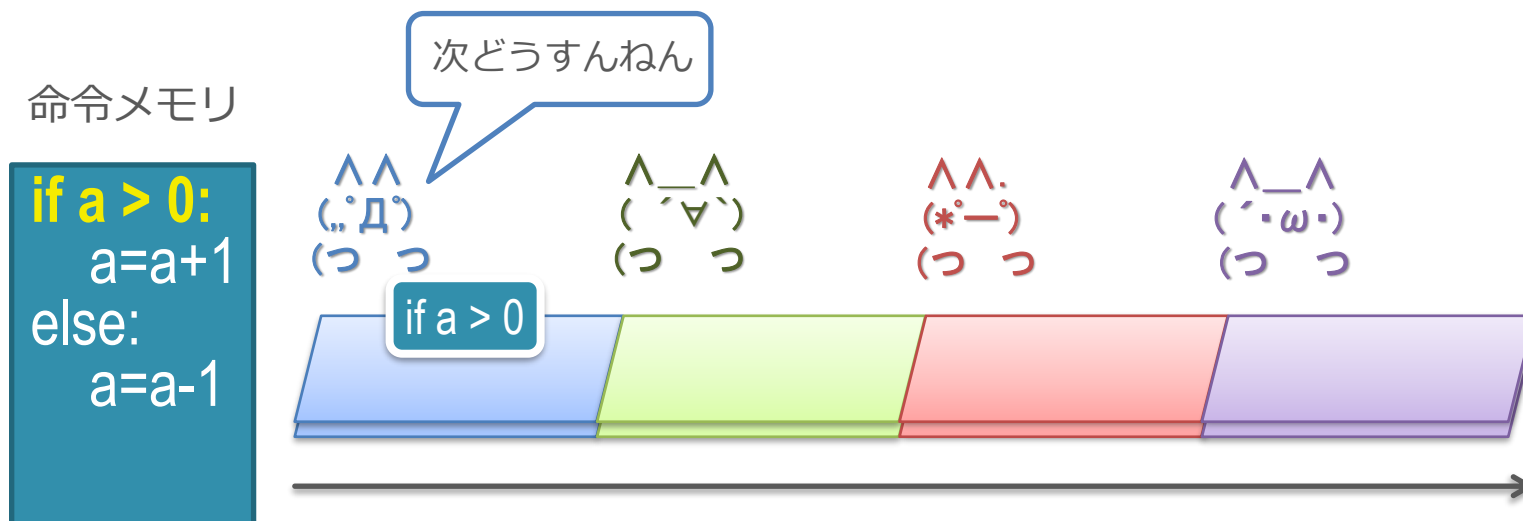
- ◇ 以降のスライドでは、スケジューリングの背景として命令間の依存関係を整理しておく

命令間の依存関係

1. 制御依存
2. データ依存
 1. 真の依存
 2. 偽の依存

制御依存

- 分岐とその後ろにある命令間の依存
 - ◇ 分岐命令の後ろにある命令は，分岐先がわかるまで実行不能
 - ◇ 分岐先が確定するまでどこを実行すれば良いか不明なため
- 分岐予測による投機実行により，効果的に解決できる



データ依存

1. 真の依存


- ◇ フロー依存 : RAW (read after write)

2. 偽の依存

- ◇ 逆依存 : WAR (write after read)
- ◇ 出力依存 : WAW (write after write)

真の依存：フロー依存

RAW (read after write)

- 文字通り, 同じレジスタを「書いた後に読む」際の依存
 - ◇ 「真の依存」, 「フロー依存」, 「RAW」は呼び方が違うだけでおなじものを指している
 - ◇ 一般に「データの依存関係」と言われたら思い浮かべるもの
- 真の依存の例：I1 が終わらないと I2 は実行できない
I1: add **x1** ← x2 + 1

I2: add x3 ← **x1** + 1

偽の依存 1 : 逆依存

WAR (write after read)

- 同じレジスタを「読んだ後に書く」

- ◇ 真の依存（書いた後に読む）と方向が逆

- 逆依存の例：

I1: add x2 ← x1 + 1



I2: add x1 ← x3 + 1

- I1 と I2 の間には真の依存は存在しない

- ◇ 「←」の右の入力部分だけみると、順番を入れ替えても問題ない

- ◇ しかし、もしスケジューリングして I2 を先にやると x1 が破壊されてしまう

偽の依存 2 : 出力依存

WAW (write after write)

- 同じレジスタを「書いた後に書く」

- 出力依存の例 :

I1: add **x1** ← x2 + 1



I2: add **x1** ← x3 + 1

- 逆依存と同様に, I1 と I2 の間には真の依存は存在しない
 - ◇ 「←」の右辺にある入力部分だけをみると, 順番を入れ替えても問題ない
 - ◇ しかし, スケジュールして I2 を先にやると I1 により x1 が破壊されてしまう

真の依存と偽の依存

- 真の依存は原理的に取り除きようがない
- 偽の依存は有限のレジスタを使い回すことによって発生する
 - ◇ いろいろ取り除きようがある

偽の依存の解消の例

- たとえばレジスタがたくさんあれば、事前に取り除ける

- ◇ 逆依存

I1: add x2←x1+1  I1: add x2←x1+1
I2: add x1←x3+1 I2: add x4←x3+1


- ◇ 出力依存

I1: add x1←x2+1  I1: add x1←x2+1
I2: add x1←x3+1 I2: add x4←x3+1

- 実際にはレジスタは無限に大きくできない

- ◇ 記憶回路の容量と速度はトレードオフがある

余談：値予測

- 真の依存を超えて実行を行う値予測（value prediction）という手法も研究されている
 - ◇ 依存元命令の演算結果自体を予測して，実行を先に進める
 - ◇ （この講義では基本的には真の依存は超えられないものとして話をします）
- 関数呼び出し時にメモリに退避させたレジスタの値を復帰させる場合などは結構精度高く予測できたりする
 - ◇ 一時期下火だったが，また研究されだした
 - ◇ 近い将来に実際に搭載されると思う
- I2 は値予測によって予測された x1 の値を使って I1 より先に実行


```
I1: add x1 ← x2 + 1
I2: add x3 ← x1 + 1
```

今日の内容

1. 命令の並列実行
2. データ依存
- 3. 静的命令スケジューリング**
4. 動的命令スケジューリング

静的命令スケジューリング

■ 静的命令スケジューリング

- ◇ コンパイラにより、並列実行できるように命令を並びかえる方法

■ 静的 vs. 動的

- ◇ 静的：

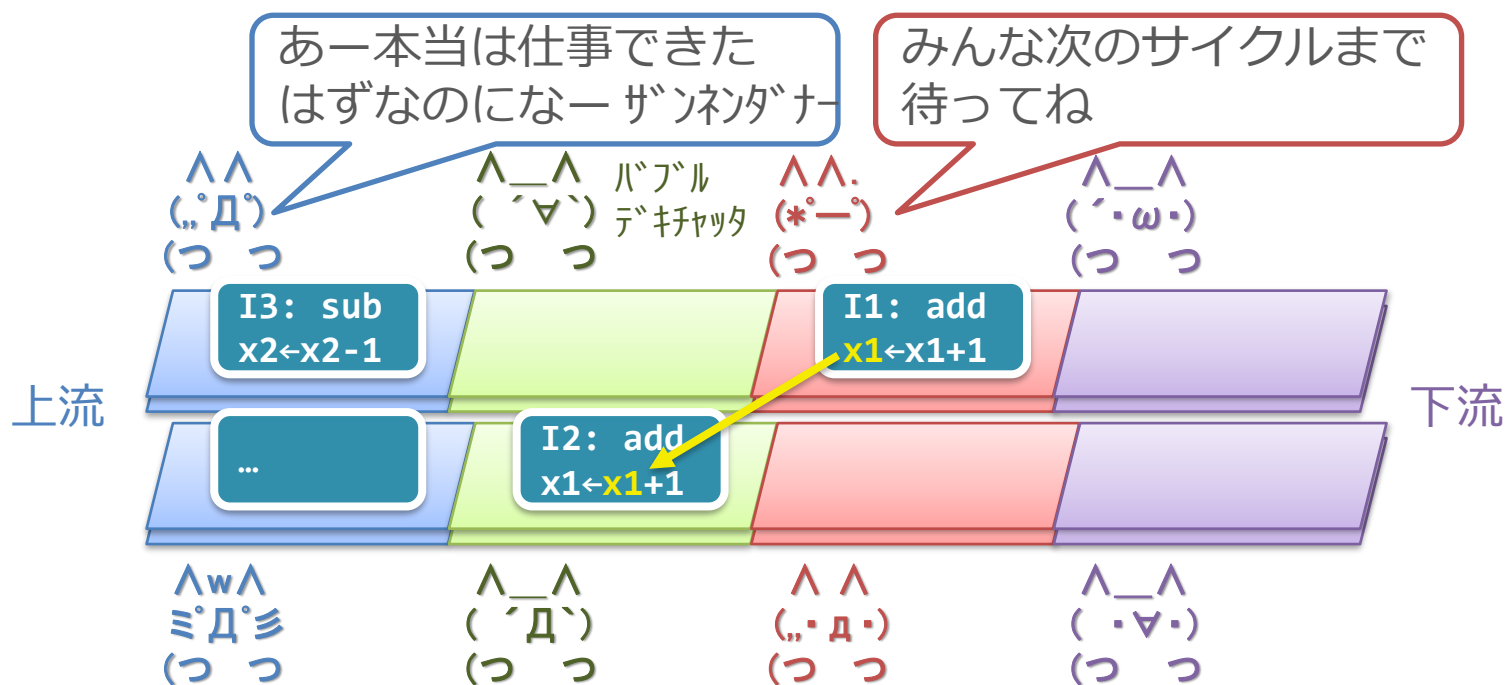
- 事前に並び替えておくので、CPU からみると実行順は変化しない

- ◇ 動的：

- CPU が実行時に並び替える

単純なスーパースカラでの実行の例

- 下記のコードでは I1 と I2 には真の依存があるが、I3 は無関係
 - ◇ しかし、上流が全部とまるので、I3 も実行できない
 - ◇ I1: add x1←x1+1
 - I2: add x1←x1+1
 - I3: sub x2←x2-1




静的スケジューリングによる解決

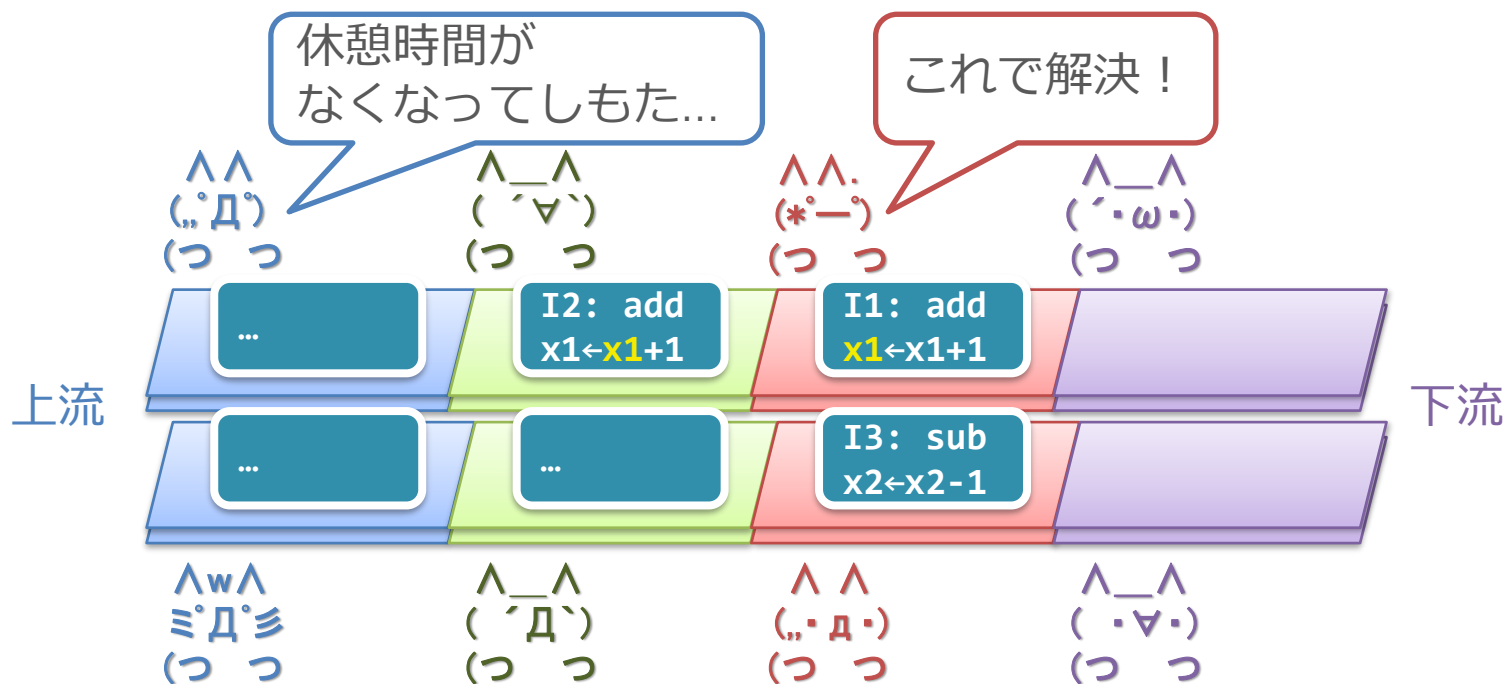
- I2 と I3 を入れ替えておけば、パイプラインはとまらない

◇ I1 と I3 が同時にパイプラインに投入される

◇ I1: add $x1 \leftarrow x1 + 1$
I2: add $x1 \leftarrow x1 + 1$
I3: sub $x2 \leftarrow x2 - 1$



I1: add $x1 \leftarrow x1 + 1$
I3: sub $x2 \leftarrow x2 - 1$
I2: add $x1 \leftarrow x1 + 1$



VLIW : Very Long Instruction Word

- 静的スケジューリングを前提とした CPU のアーキテクチャ
- 通常の命令相当の操作を複数まとめたものを 1 つの命令とする
 - ◇ 命令セットの仕様として, 1 つの VLIW 命令内では依存関係を持たないようにする
 - ◇ = フェッチした後は必ずそれらは並列実行できる

◇ I1: add x1←x1+1
I2: add x2←x2+1
I3: sub x3←x3-1
通常の命令セット

I1:

add x1←x1+1
add x2←x2+1
sub x3←x3-1

VLIW ではこれで 1 命令

VLIW の利点と問題点

■ 利点：

- ◇ ハードウェアがすごく簡単

- スーパースカラでは依存があったら止める機構があった

- VLIW では仕様として命令内に依存は発生しないので，不要

■ 問題点：

1. 性能向上に限界がある

2. 互換性がとりにくい

1. 性能がいまいち出ない

- VLIW は静的スケジューリングに全面的に頼っている
 - ◇ しかし、それで出来る並び替えは結構自由度が低い

静的スケジューリングが難しい例 1

- 例1：分岐を乗り越えた並び替えは難しい
(不可能とはいっていない)

- ◇ 3行目のメモリ・アクセスを1行目の位置まで引き上げることは困難
- ◇ うかつにやると例外が起きて落ちる

```
1: i = i + 1
2: if (flag)
3:     a = *ptr; // flag が false の時は ptr は NULL
```

静的スケジューリングが難しい例 2

- 例 2 : ポインタ参照の順番を入れ替えるのは難しい
(不可能とはいっていない)

- ◇ 2 行目と 3 行目のメモリ・アクセスを入れ替えることは困難
- ◇ うかつにやると意味が変わる

```
1: func(STRUCT* s, int* a){  
2:     s->a = 1;  
3:     int b = *a;  // a は s->a を指してる可能性がある
```

余談：C 言語などでのポインタ経由アクセス

- 以下では, `a` と `c` のために2回分のロード命令が生成される
 - ◇ 間にグローバル変数へのアクセスが入ると, 一回 `*ptr` をロードしてレジスタに置いた値が使い回せない
 - ◇ オブジェクトへのメンバへのアクセスでも同じことがおきる
 - ◇ ローカル変数に1回コピーしてからアクセスしたほうが速い
- ```
int g = 0;
func(int* ptr){
 int a = (*ptr) + 1;
 int b = (*ptr) + 2; // 最適化されて上のロード結果を使用
 g = 1; // ptr が g を指している可能性がある
 int c = (*ptr) - 1;
```

# 互換性がとりにくい

- 静的に CPU の挙動を仮定して命令をスケジュールする
  - ◇ = その仮定をくずれるとまずい
- 要因：
  1. 並列実行幅が固定されている
  2. 実行タイミングを仮定してスケジュールされている

# 1. 並列実行幅が固定されている

- 仕様として「N 命令相当を 1 つの VLIW 命令 とする」としている
  - ◇ 性能を上げるために N を後から増やそうと思っても増やせない
  - ◇ 既存のコードが動かなくなってしまう

- たとえば N を 2 から 4 にすると互換性がとれない

◇ I1: `add x1←x1+1`  
`add x2←x2+1`

I2: `sub x1←x1-1`  
`sub x2←x2-1`

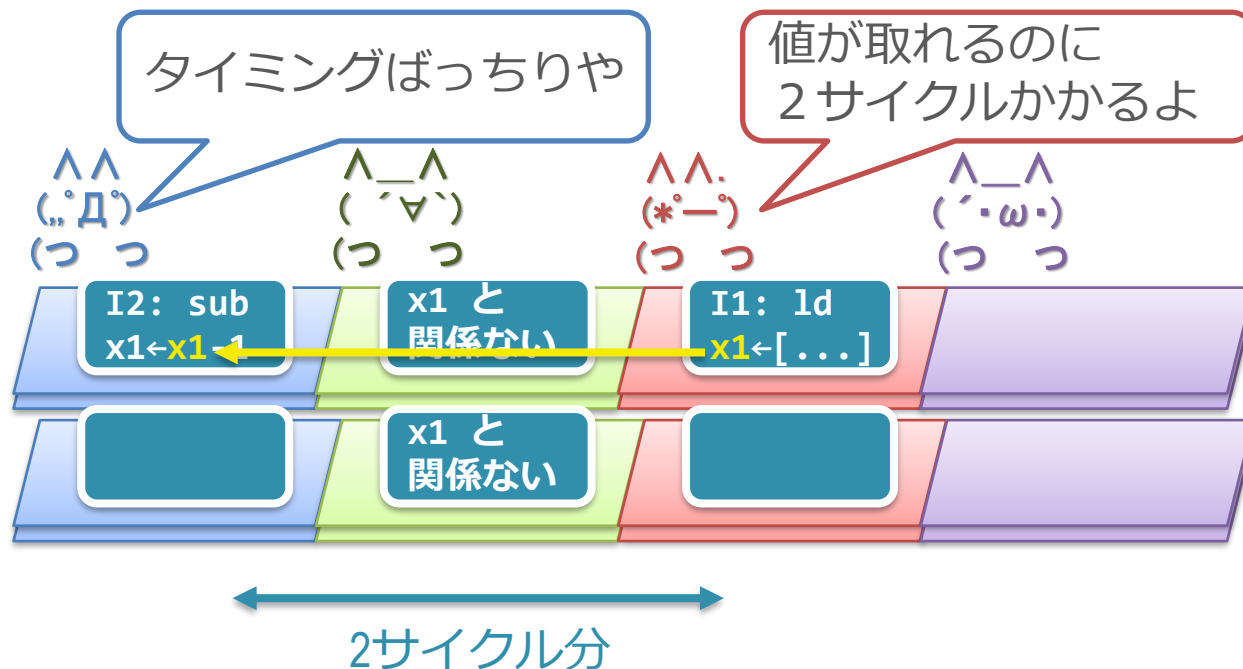
ある VLIW バージョン 1

I1: `add x1←x1+1`  
`add x2←x2+1`  
`sub x1←x1-1`  
`sub x2←x2-1`

ある VLIW バージョン 2  
そのまま実行すると仕様違反

## 2.実行タイミングを仮定してスケジュールされている

- 複数サイクルかかる命令は、それに合わせてスケジュールされる
  - ◇ 「M サイクル後に結果が使用できる」前提でパイプラインが止まらないように事前に命令が並べてある
- 以下では、I1: ld の値が使えるタイミングで I2 が実行できるよう両者を離してある



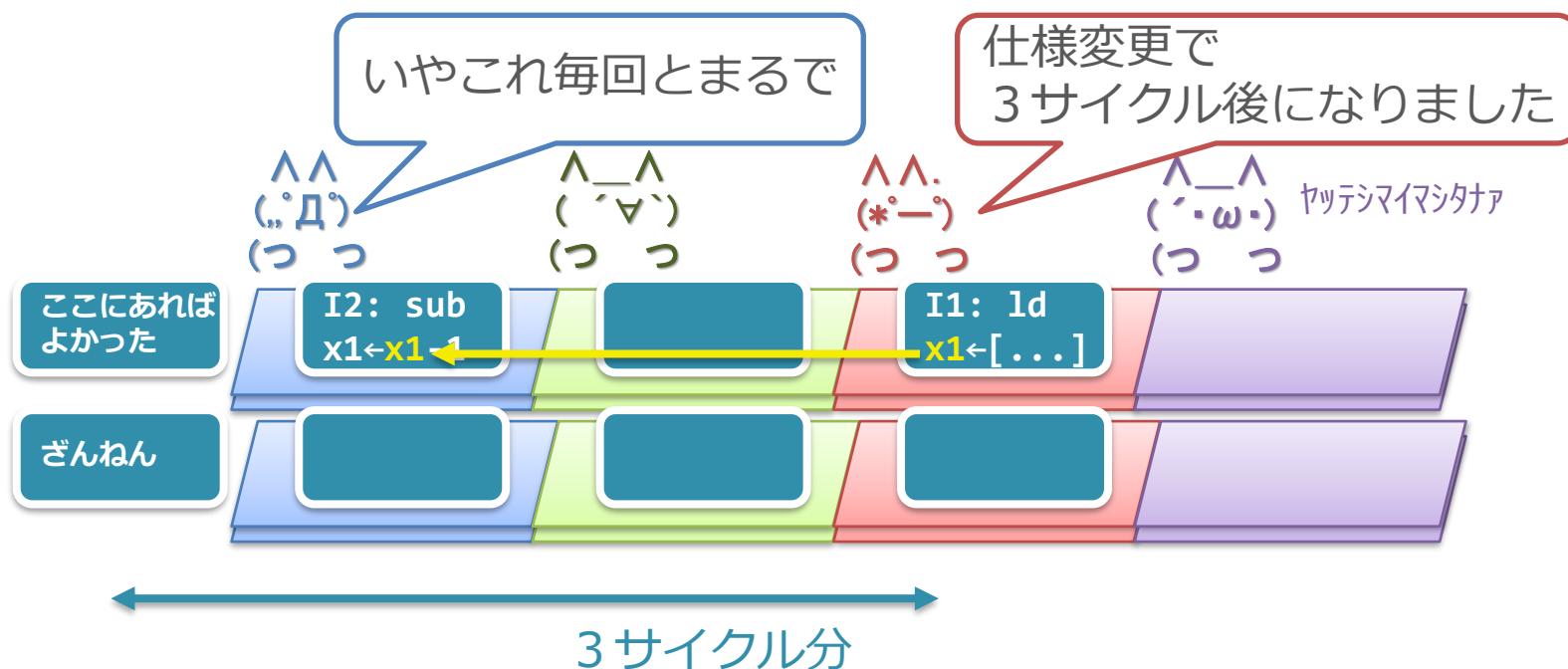


## 2. 実行タイミングを仮定してスケジュールされている

- M を変化させにくい
  1. 短くなる = 既存のコードは恩恵を受けられない
  2. 長くなる = 毎回バブルが発生して性能ががたおちする
    - 想定したタイミングに入力が揃わない
- たとえば、次世代の以下のような CPU 作る場合を考える：
  1. 乗算器を改良してレイテンシが短くなった
  2. キャッシュを倍にしてヒット率をあげたがレイテンシが多少伸びた

# キャッシュのレイテンシが伸びた場合

- Id のレイテンシが 2 から 3 に変更となった場合
  - ◇ 2 サイクルにジャストで合わせて I2 をスケジューリングしておくと、毎回バブルが発生することに



# 実行タイミングを仮定してスケジュールされている ことの他の問題

- そもそも実行時にレイテンシが動的に変化する場合是对応困難
  - ◇ キャッシュのヒットとミスが場合によってかわるようなロード
- コンパイラではあらかじめヒットかミスを仮定してスケジュール
  - ◇ プロファイラで事前に特性をとって、それに基づくことである程度緩和はできる

# VLIW の例 : Intel Itanium

- インテルと HP で作った VLIW プロセッサ
  - ◇ 2000 年代前半ぐらいまでは x86 からこれに移行しようとしていた
  - ◇ EPIC アーキテクチャと言われる命令セットを持つ
- これまで述べたような VLIW の問題を緩和するような機構を色々投入
  - ◇ 命令セットの互換性をとりながら同時実行幅を増やす
  - ◇ 分岐を跨いだロードの移動をハードで支援

# Intel Itanium の性能

- しかし, x86 よりも全然性能がでなかった
  - 1. 静的スケジューリングの限界
  - 2. レイテンシを仮定したコード
  - 3. クロックが上げられなかった
    - 1. 2. に関連して, キャッシュ・アクセスのステージ数を増やしてクロックを上げることができない
    - 2. VLIW の問題緩和の機構のせいで返って複雑化

# Intel Itanium の末路

1. 当時 32 ビットから 64 ビットへの移行の要求が高まっていた
  - ◇ 主にメモリ使用量を増やすため
    - 32 ビットのアドレスで表せるのは 4GB まで
  - ◇ Itanium はこのための 64 ビット CPU でもあった
2. インテルは互換 CPU の製造開発を許したくなかった
  - ◇ しかし既に与えたライセンスは取り消せない
  - ◇ 64 ビット世代で内容を刷新して今度は独占を目指した
3. AMD が独自に x86-64 を策定
  - ◇ Itanium がさっぱり性能でないので、MS が見切りをつけて Windows の x86-64 対応を開始
4. 後追いでインテルも x86-64 の CPU を開発
  - ◇ Itanium は一応製造されているが、2021 年に最終出荷で終了

# VLIW は全くダメなのか？

- 以下のような場面であれば有用
  - ◇ 絶対性能よりも、ハードが小さいこと（電力）の要求が高い
  - ◇ 動作させるソフトウェアが限られている
    - 互換性が問題になりにくい
- 典型的には、組み込み CPU が該当
  - ◇ 簡単なハードでそこそこの性能がほしい時に有用
- CPU を作る学生実験で性能出したい場合なんかでも有望
  - ◇ 実装が簡単 & 課題となる少数のプログラムさえ速ければよい
  - ◇ 人力静的スケジュールで最適化する
    - いろんな仮定をぶちやぶれる

# 今日の内容

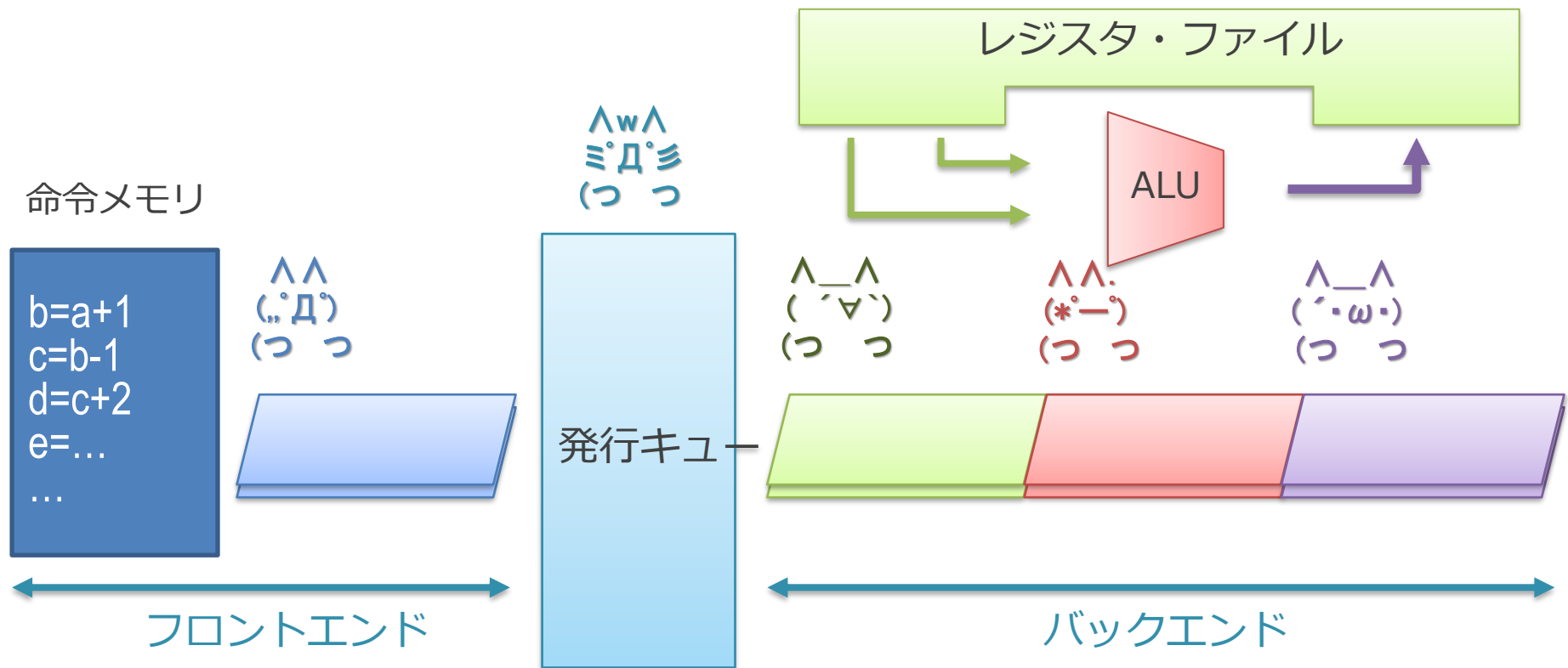
1. 命令の並列実行
2. データ依存
3. 静的命令スケジューリングと VLIW
4. 動的命令スケジューリング (のさわり)



# 動的命令スケジューリング

- CPU により, うまく並列実行できるように命令を並びかえる方法
  - ◇ 静的 : 事前に並び替えておくので, CPU からみると変化しない
  - ◇ 動的 : CPU が実行時に並び替える
- スカラ/スーパスカラとは直行した概念
  - ◇ ...ではあるが, 普通は動的スケジューリングを行う CPU はスーパスカラ
  - ◇ スカラで動的スケジューリングをやってもあまり意味がないから
- 現在主流の CPU は, 基本的にみなこのタイプ

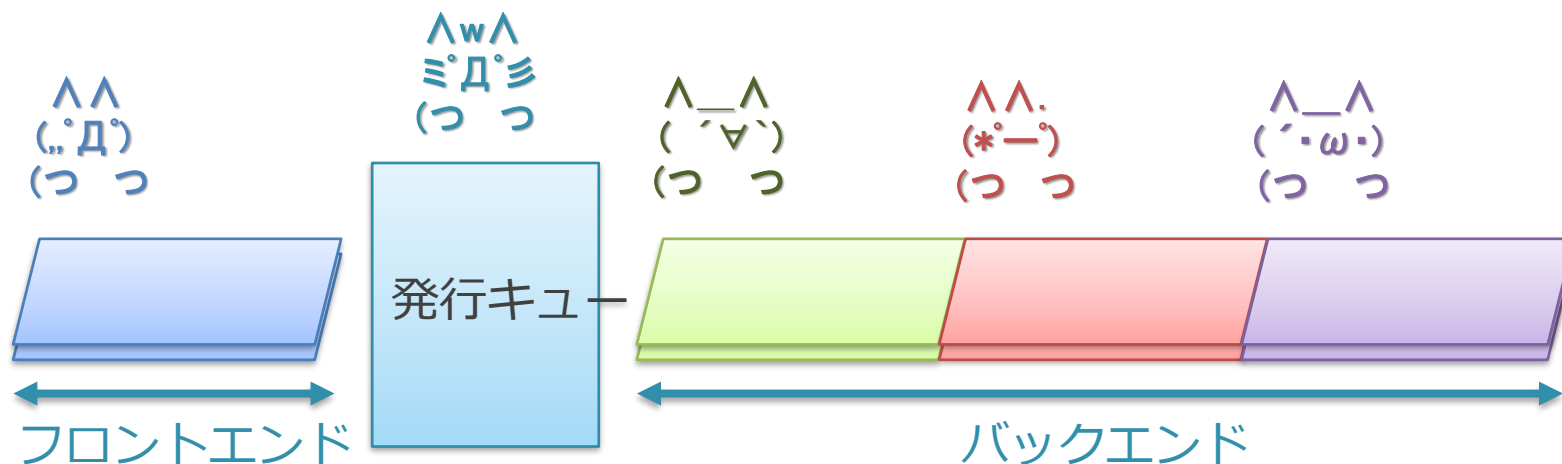
# 動的命令スケジューリングを行う CPU の構造



## ■ 発行キューによって前後に分離された構造を持つ

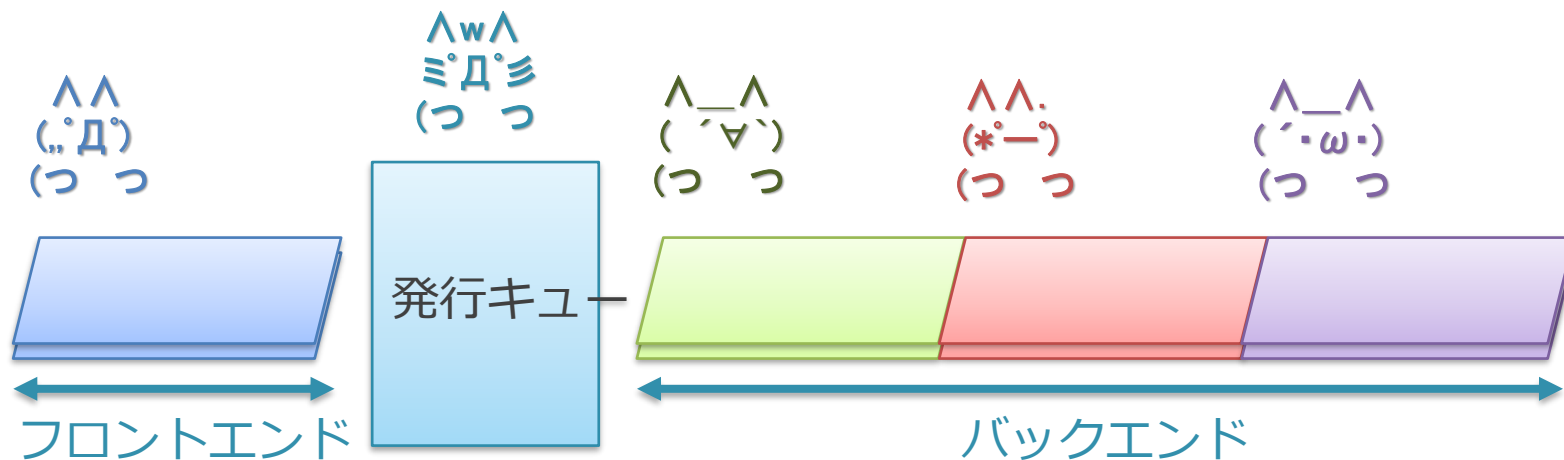
1. フロントエンド : 命令を供給
2. 発行キュー : 命令の待ち合わせ
3. バックエンド : 命令を実行

# 大ざっぱな動作



1. フロントエンドで命令を順にフェッチ
2. 発行キューに投入
3. 実行可能なものから順にバックエンドに命令を送信
4. レジスタを読んで演算器で実行し書き戻す

# 言葉の定義 1



- ◇ ディスパッチ：フロントエンドから発行キューに命令をいれること
- ◇ 発行 (issue)：発行キューからバックエンドに命令を送ること
- ◇ 完了 (complete)：バックエンドで命令の処理が終わること
- ◇ 微妙にこのあたりの用語は文献ごとに統一されていないので注意
  - インテルは昔からかたくなにディスパッチと発行を逆の意味で使う

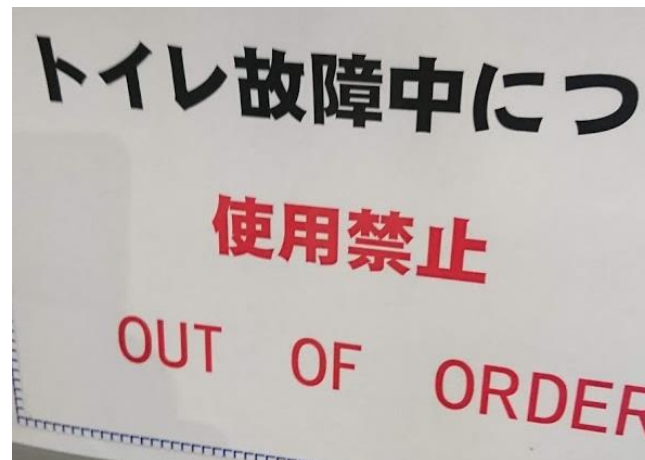
# 言葉の定義 2

- 並び替えに関係する用語：

- ◇ in-order : プログラム順のこと

- ◇ out-of-order (OoO) : 上記とは違う順番のこと

- (一般には, 公共性の高い機器が故障してることを言うらしい)



# 今日の内容

1. 命令の並列実行
2. データ依存
3. 静的命令スケジューリングと VLIW
4. 動的命令スケジューリング（のさわり）

# 出欠と感想

- 本日の講義でよくわかったところ, わからなかったところ, 質問, 感想などを書いてください
  - ◇ LMS の出席を設定するので, そこにお願いします
  - ◇ パスワード:
- 意見や内容へのリクエストもあったら書いてください
- LMS の出席の締め切りは来週の講義開始までに設定してあります
  - ◇ 仕様上「遅刻」表示になりますが, 特に減点等しません
  - ◇ 来週の講義開始までは感想や質問などを受け付けます