

先進計算機構成論 09

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

出席について

- 出席の締め切りが講義開始時に設定していたつもりでしたが、本日0時になっていました
 - ◇ 16時に再設定しています
- 過去にも2回ほどそう言うときがあったので、覚えがある人はメール等で申告してください

- 動的スケジューリングについて少し疑問があり、発行キューで命令を並び替えると思うが、そのアルゴリズムというかその工程にかかる時間は依存関係によるストール等に比べてどれほどの時間オーダーで小さいものなののでしょうか。

◇ 本日説明します

- スーパースカラプロセッサには制約があって性能が上がりづらいとの話がありましたが、実際のコンピュータでは採用されていないのでしょうか
- ◇ 上がりづらいのは VLIW

- 分岐予測の性能が悪いとバブルが発生しやすいという認識でよろしいですか.

◇ 概ねその通り

- VLIWは、消費電力や互換性の低いソフトウェアにおいて有用と紹介されていましたが、実際FPGAに実装されたりしないのでしょうか。
- ◇ FPGA 上などのソフトプロセッサとして実装するならいいかも
- ◇ ただしコンパイラの開発は大変

- C++で例外を出さない関数にはnoexcept修飾子つけた方が、実行順序を入れ替えやすいのでコンパイラの最適化の幅が広がるという話を聞いたことがあります。実際にそれがどの程度効いてくるのかは分かりませんが。
 - ◇ noexcept 自体はもともと move constructor の実現のためだったかも
 - ◇ 命令レベルの並べ替えの話よりも、もう少し上位の話かも

質問や感想

- RISC-Vのカスタム命令でニューラルネットワークを高速化する研究をやりたいんですが（PULP-NNみたいなもの）、カスタム命令をシミュレーション可能かつ高速なプロセッサシミュレーターは何かいいものはありませんか？
- ◇ たぶん、真面目に実行結果までシミュレーションしないタイプのトレースベースのシミュレータが簡単かつ高速で良いと思う
 - 実行結果をシミュレーションするもの：
 - * gem5, SESC, 鬼斬
 - トレースベース：
 - * ChampSim, Sniper
- ◇ ただし、ベースとするプロセッサ構成などにもよるので、別途相談してください

- 以前、別のコンピュータ系の講義を受けた際に「命令を並べ替える方法には、VLIWとスーパースカラがある」というようにこの2つを並列に並べて議論していたのですが、これは、この先生が「スーパースカラ=out of order実行を行うスーパースカラプロセッサ」と認識されていた、という理解で合っているでしょうか
- ◇ それはそうだと思う
- ◇ VLIW とスーパースカラがあると言うよりは、コンパイラによる静的な方法とハードによる動的な方法がある、とした方が良いと思う

- I've forgot the source but I think there's a myth that in C++, using an infinite while with break is better than using for loops. Is that true from the hardware perspective?
 - ◇ Today's hardware can execute programs written both ways at similar speeds.
 - ◇ Today's compilers can optimize code written both ways to nearly the same instruction sequence.

- Maybe I missed this part during the lecture. What is the difference between VLIW architecture and superscalar processor?
- ◇ VLIW executes instructions statically scheduled by a compiler, while an out-of-order superscalar processor dynamically schedules instructions during execution.

- 分岐命令が内部的に条件付き命令に変換されているというのが面白かったです。ここでの内部的というのはハードウェアがやっているとのことでしょうか？

◇ その通り

- 一方で、c言語をコンパイルするときの最適化で、例えばループアンローリング等があると思うのですが、コンパイラがやる最適化とハードウェアがやる最適化はどう分けられているのでしょうか。
- ◇ 基本的に、「動かしてみないとわからない」「単純な」ようなことはハードにやらせると良い
- ◇ 分岐の変換：
 - 「分岐予測が外れる分岐である」は実行時にしかわからない
 - 条件付き実行への変換は、そこまでは複雑でない
- ◇ 冗長部分式の削除，定数伝播：
 - そう言うのはハードには難しい
 - （とも言い切れなくなってきた・・・

- spectre, meltdownのどちらか忘れましたが、out of order実行を原因とするプロセッサの脆弱性があったと思うのですが、これはスーパースカラやVLIW等でも発生したりはしないのでしょうか
- ◇ ありえなくはないが、基本的に in-order なスーパースカラや VLIW では難しいと思う

- glibcでは`likely()`等のマクロを使って、到達しやすいパスやしにくいパスを明示しているようです(が、それが実際にどうコンパイルされるのかはよく知りません)
- ◇ 命令がジャンプせずに直進させるように命令列が生成できると速くなるが、そこに効くきがする

- 悪い効率、コストを甘受してでもマルチポートなどの解決策を使うことはありますか？
 - ◇ これはあり得る

- 昔の講義で、「VLIWは歴史的にはスーパースカラ方式の欠点を解決するものとして登場した」と聞いたのを思い出しました。やはり、静的な並列化の限界や互換性がないことが実用的に厳しかったんですかね。。。◇ それもあるが、「スーパースカラ方式の欠点」は、「その当時の良く考えず特に工夫しなかった場合の欠点」であり、大概の部分は頑張ったら克服された

- 静的スケジューリングが難しい例で挙げられていたことはrustでは避けられるような気がしました。(コンパイラがハードウェア依存になったり他の問題もいっぱいあるので難しいことには変わらないですが...)“
- ◇ 分岐や関数呼び出しを跨いで命令を移動させるのはツライと思う

- 静的命令スケジューリングをコンパイラで行う場合、命令間の依存関係をどのように検知しているのでしょうか？
 - ◇ コンパイラは自分で命令列を生成しているので、依存関係は最初から知っている
- データの依存関係と聞いて有向グラフを思い浮かべました。実際にグラフの考え方が使われることはあるのでしょうか。
 - ◇ めっちゃ使う

- 現在のPCのプロセッサは基本スーパースカラという認識であってますか？

◇ そうです

- だんだんと自分の中で各機構がブラックボックス化してきました。とくにハザード検出などどうやっているのか複雑な印象です。．．

◇ 今日は少しその話題もあります

- 静的スケジューリングは、スーパースカラの並列数に最適化の規模が依存すると思うのですが、主要なコンパイラはどうしているのでしょうか?

◇ パラメータとして与えていると思う

- 業界（最？）大手が他の会社だけでなく学会一般とも違う用語の使い方をしているというのは面白い話だと思いました。どのような経緯があるのでしょうか。
- ◇ めいめい勝手な呼び方をしたら一貫性がないまま色々定着してしまった
- ◇ 設計データの機密性が高く内側で閉じた世界を作っている事が多いので、あんまり困らないのかも

- 月並みな感想ですが、日頃のプログラミングで楽をさせてもらっていることを痛感しました。

前回の内容

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

今回の内容

- 動的命令スケジューリング

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
 - ◇ 静的命令スケジューリング
 - 事前にコンパイラなどで命令を並び替えておく
 - CPU からみると実行順は変化しない
 - ◇ 動的命令スケジューリング
 - CPU が実行時に命令の実行順を並び替える
 - (以降では動的スケジューリングとも呼ぶ)

動的命令スケジューリング

■ スカラとスーパスカラ

- ◇ スカラ： 1 サイクルに単一の命令を実行

 - 基本的に in-order

- ◇ スーパスカラ： 1 サイクルに複数の命令を同時実行

 - in-order と out-of-order がある

■ 動的スケジューリングは上記とは直行した概念

- ◇ ...ではあるが、普通は動的スケジューリングを行う CPU はみんなスーパスカラ

- ◇ スカラで動的スケジューリングをやっても、複雑さの割にあまり性能がでないからだと思う

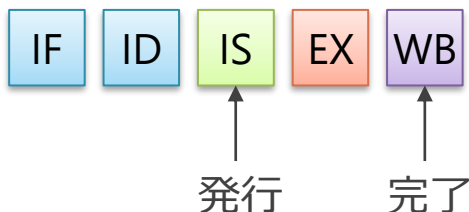
■ 現在主流の高性能な CPU は、基本的にみなこのタイプ

言葉の定義 1

- 発行 (issue)
 - ◇ 演算器に命令を送信すること
- 完了 (complete)
 - ◇ 演算が終わり, レジスタ・ファイルへ結果を書き込むこと

スカラのパイプラインを使って説明

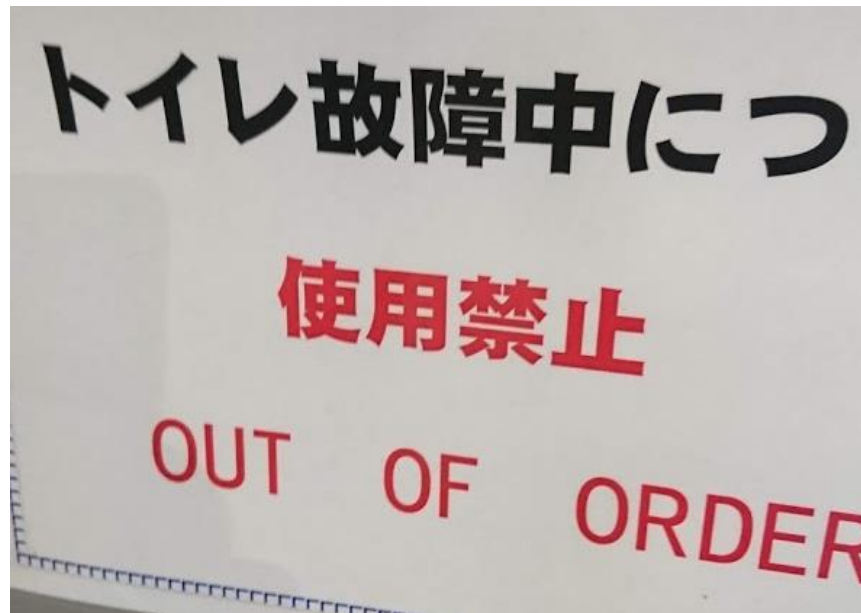
- 以降で使うパイプラインの構造
 - ◇ 簡単にするため、MEM ステージはない
 - ◇ ID ステージの後に、発行のための IS (issue) ステージを設ける
- レジスタは ID の時点では読まず、IS で読む
 - ◇ 説明の簡単化のため、このスライドではそうすることにする
 - ◇ 「発行」はあくまで演算器へ命令を送り出すことであり、本来はレジスタ読み出しの意味を含まない



言葉の定義 2

■ 並び替えに関係する用語：

- ◇ in-order (InO) : プログラム内の実行順のこと
- ◇ 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)



動的スケジューリングを行う CPU の分類

- 発行と完了の順序によって分類できる
 - ◇ それぞれ in-order に行うか out-of-order (OoO) に行うか
- 上記分類の結果としてスケジューリング能力が決まる
 - ◇ 命令の並び替えを許すか？
 - ◇ 偽の依存を超えて並び替えを許すか？
 - ◇ (真の依存には必ず従う必要がある

真の依存と偽の依存

1. 真の依存 (RAW: Read after Write) :

I1: add **x1** ← x2 + 1

I2: add x3 ← **x1** + 1

2. 逆依存 (WAR: Write after Read) :

I1: add x2 ← **x1** + 1

I2: add **x1** ← x3 + 1

3. 出力依存 (WAW: Write after Write) :

I1: add **x1** ← x2 + 1

I2: add **x1** ← x3 + 1

スケジューリング方法の分類

簡単・低性能



1. in-order 発行/in-order 完了

- ◇ プログラム順に従って実行
(動的スケジューリングなし)

2. in-order 発行/out-of-order 完了

- ◇ 真の依存と偽の依存を守る範囲で並び替えて実行

3. out-of-order 発行/out-of-order 完了

- ◇ 真の依存のみを守って並び替えて実行

複雑・高性能

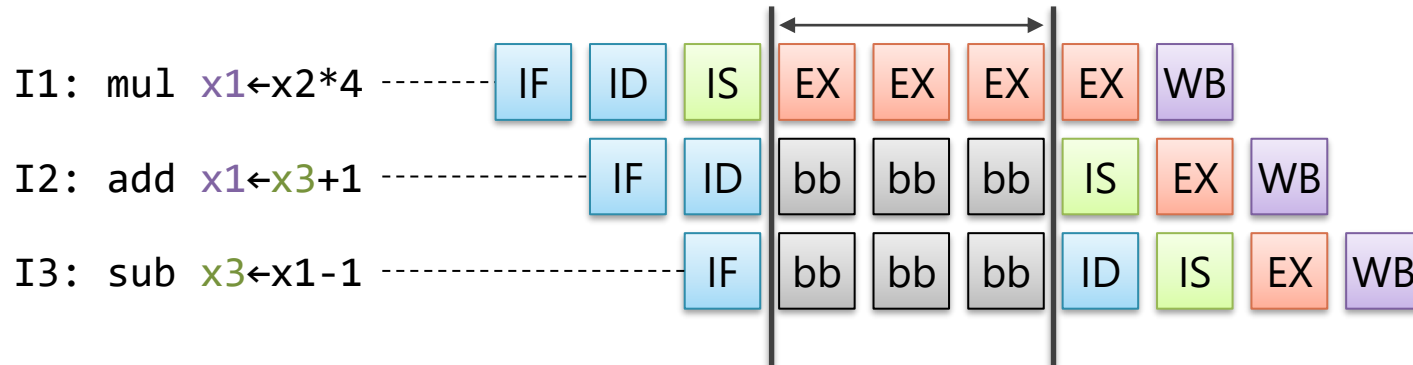
(out-of-order 発行/in-order 完了の組み合わせは
おそらく意味がない)

スケジューリング方法の分類

1. **in-order 発行/in-order 完了**
2. in-order 発行/out-of-order 完了
3. out-of-order 発行/out-of-order 完了

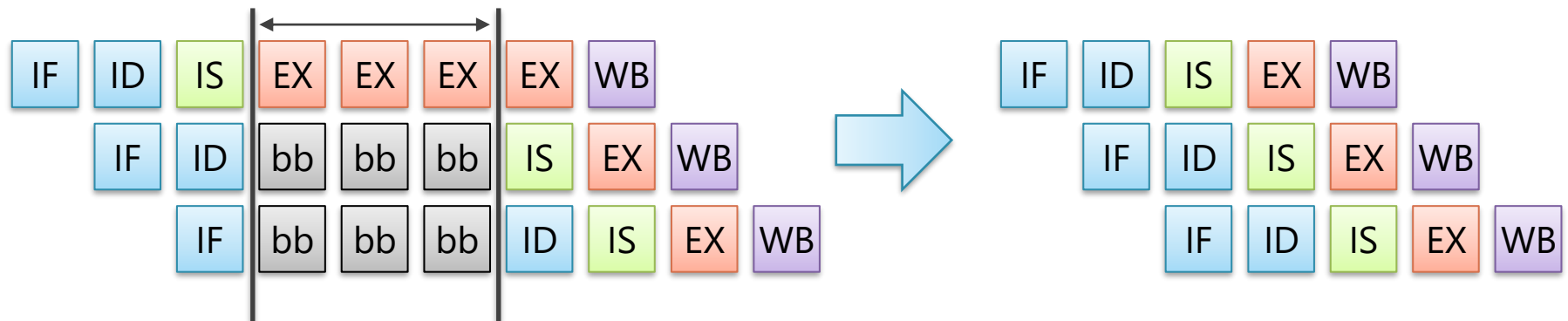
in-order 発行/in-order 完了

- 動作：レイテンシが長い命令が来たらパイプラインをストール
 - ◇ 依存関係とかに関係なく，問答無用でとめる
- 通常の命令よりレイテンシが伸びる分だけとめる
 - ◇ 下記の場合，mul の遅延は 4 サイクルなので $4-1=3$ サイクル



単純なスカラ・パイプラインの動作と同じ

- レイテンシが長い命令現れた場合には，世界中が止まる
 - ◇ = 1 サイクルの命令のみでパイプラインで実行してるのと等価
- あらゆる追い越しは原理的に起きず，真&偽 の依存は常に守られる



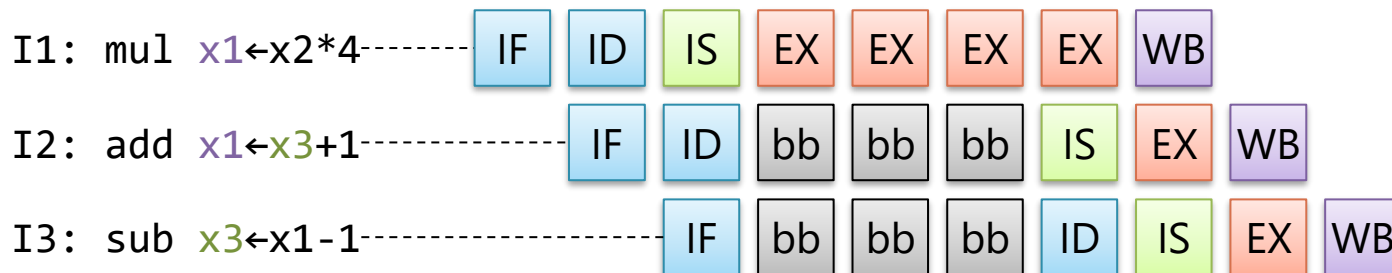
in-order 発行/in-order 完了

1. 発行が in-order

- ◇ ある命令の IS が終わった後に、次の命令の IS（とWB）が来る
- ◇ 逆依存（WAR）が守られる
 - write-after-read の違反が起きない

2. 完了が in-order

- ◇ ある命令の WB が終わった後に、次の命令の WB が来る
- ◇ 出力依存（WAW）が守られる
 - write-after-write の違反が起きない



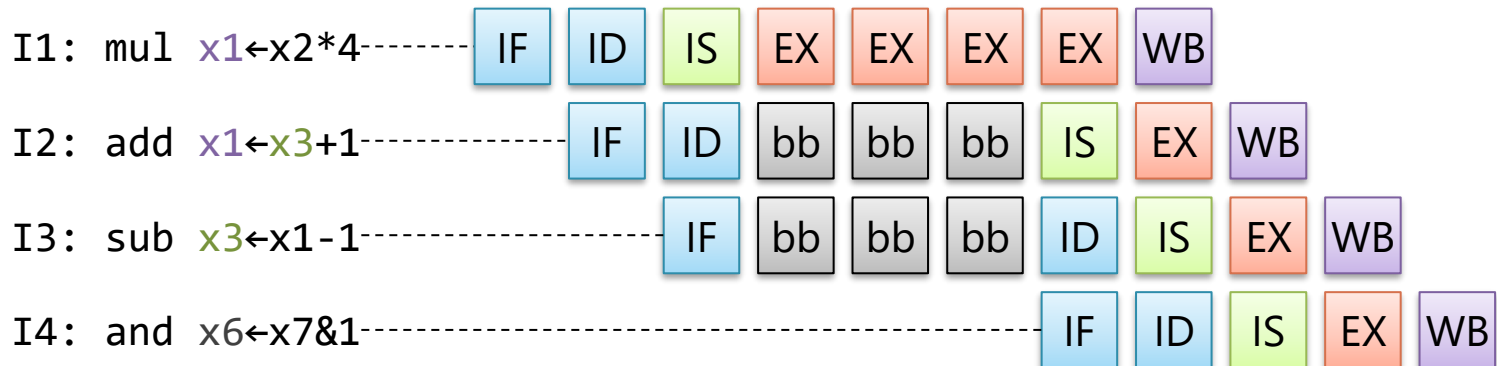
in-order 発行/in-order 完了

■ 性能は最も低い

- ◇ mul と依存がない命令であっても, mul の完了を待たされる
- ◇ しかし, 最も単純

■ I1 に対し, 以下は全員待たされる

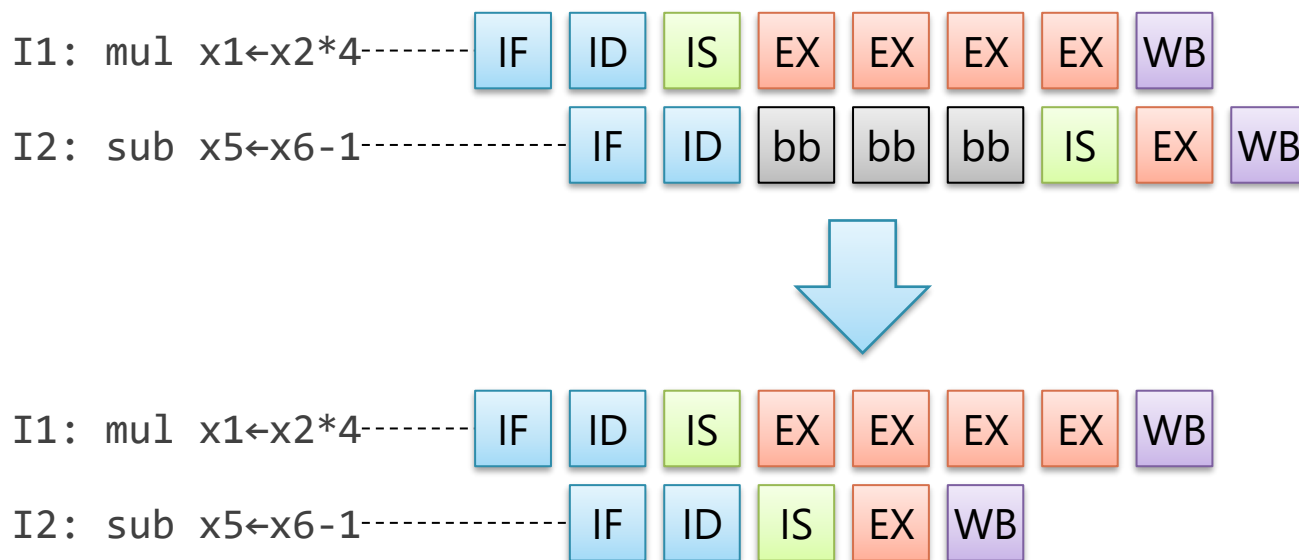
- ◇ I2: 出力依存
- ◇ I3: 逆依存
- ◇ I4: 依存なし



動的スケジューリングを行うプロセッサの分類

1. in-order 発行/in-order 完了
2. in-order 発行/out-of-order 完了
3. out-of-order 発行/out-of-order 完了

モチベーション：依存がない命令を裏で実行したい



■ I2 は I1 に対して真の依存も偽の依存もない

◇ したがって, I1 の完了を待たずに I2 を発行しても動作の正しさは保たれる

◇ なんとかして, I2 を I1 の裏で並列に実行したい

■ 方針：

◇ 命令毎に依存の有無を確かめながら適宜ストール or 発行を行う

スコアボードによるストールの判定

■ スコアボード (Scoreboard)

- ◇ レジスタや演算器などの状態を保持するテーブル群
 - 各資源が現在、使用可能か不能かを保持
- ◇ 今回は簡単にレジスタが使用可能かだけを考えることにする

■ レジスタ状態 :

- ◇ 各レジスタの値が現在利用可能かを 1 bit のフラグで保持
- ◇ 1 なら現在使用可能, 0 なら実行結果の書き込み待ち
 - (文献によってはは 1/0 が反転していることもある)

レジスタ・ファイル スコアボード

x1	0x123	1
x2	結果待ち	0
...		...

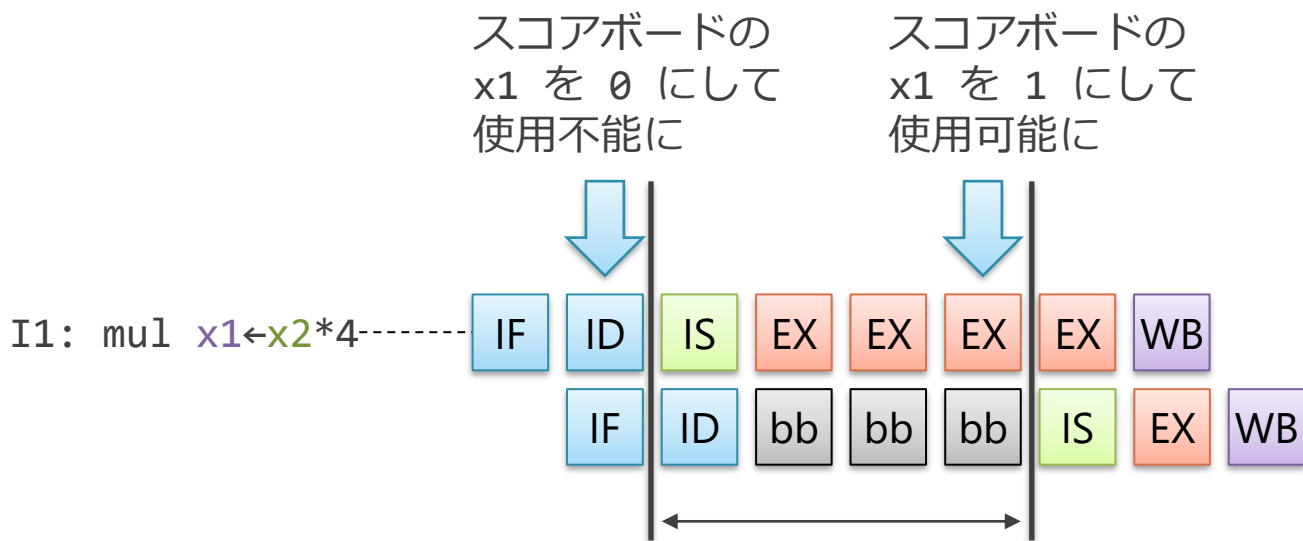
スコアボードの更新とストール

■ 更新：

- ◇ 長いレイテンシの命令がデコードされた場合、それが終わるまでそのディスティネーションを使用不能にする

■ スコアボードが以下のいずれかの場合にパイプラインをストール

1. ソース・オペランドが使用不能
2. ディスティネーション・オペランドが使用不能



スコアボードによるストールの判定

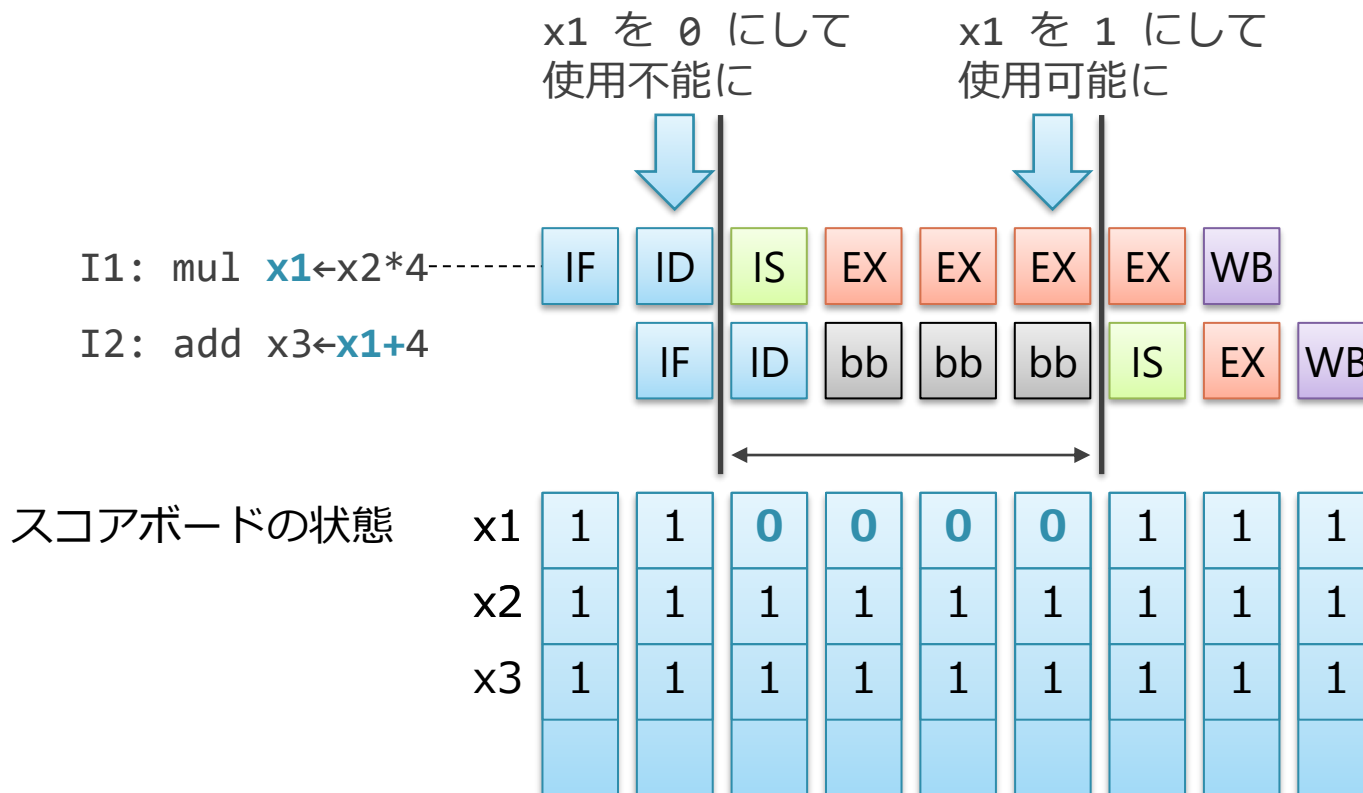
1. 真の依存がある場合
2. 出力依存がある場合
3. 逆依存がある場合
4. 依存がない場合

- 上記それぞれの場合に,
ソースとディスティネーションの状態を元に説明

1. 真の依存がある場合

■ I2 のデコード時 :

1. ソースの x1 が使用可能になるまでストール
□ 真の依存が守られる
2. ディスティネーションの x3 は使用可能なので問題ない

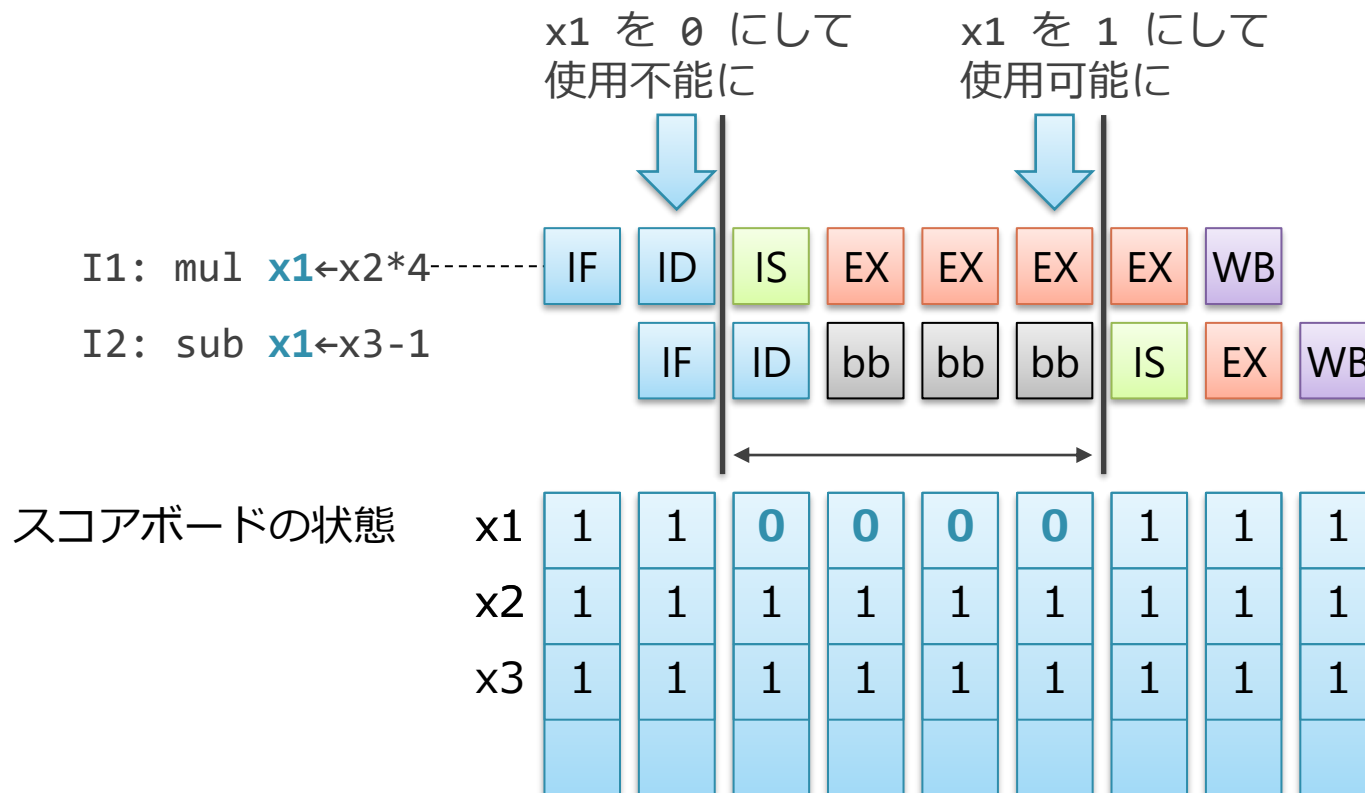


2. 出力依存（WAW）がある場合

■ I2 のデコード時：

1. ソースの x3 は使用可能なので問題ない
2. ディスティネーションの x1 が使用可能になるまで待つ

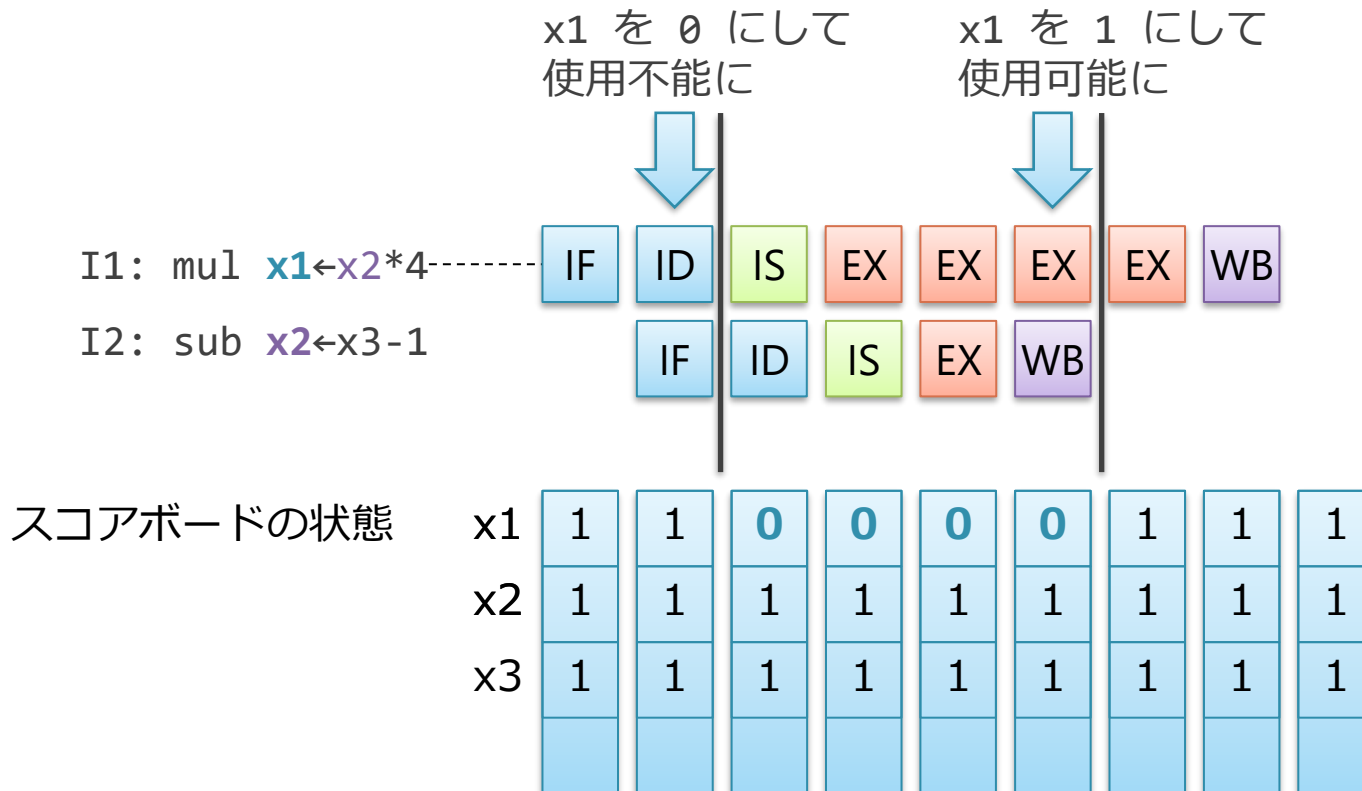
□ 出力依存が守られる = 同じレジスタへの書き込みが in-order



3. 逆依存 (WAR) がある場合

■ I2 のデコード時 :

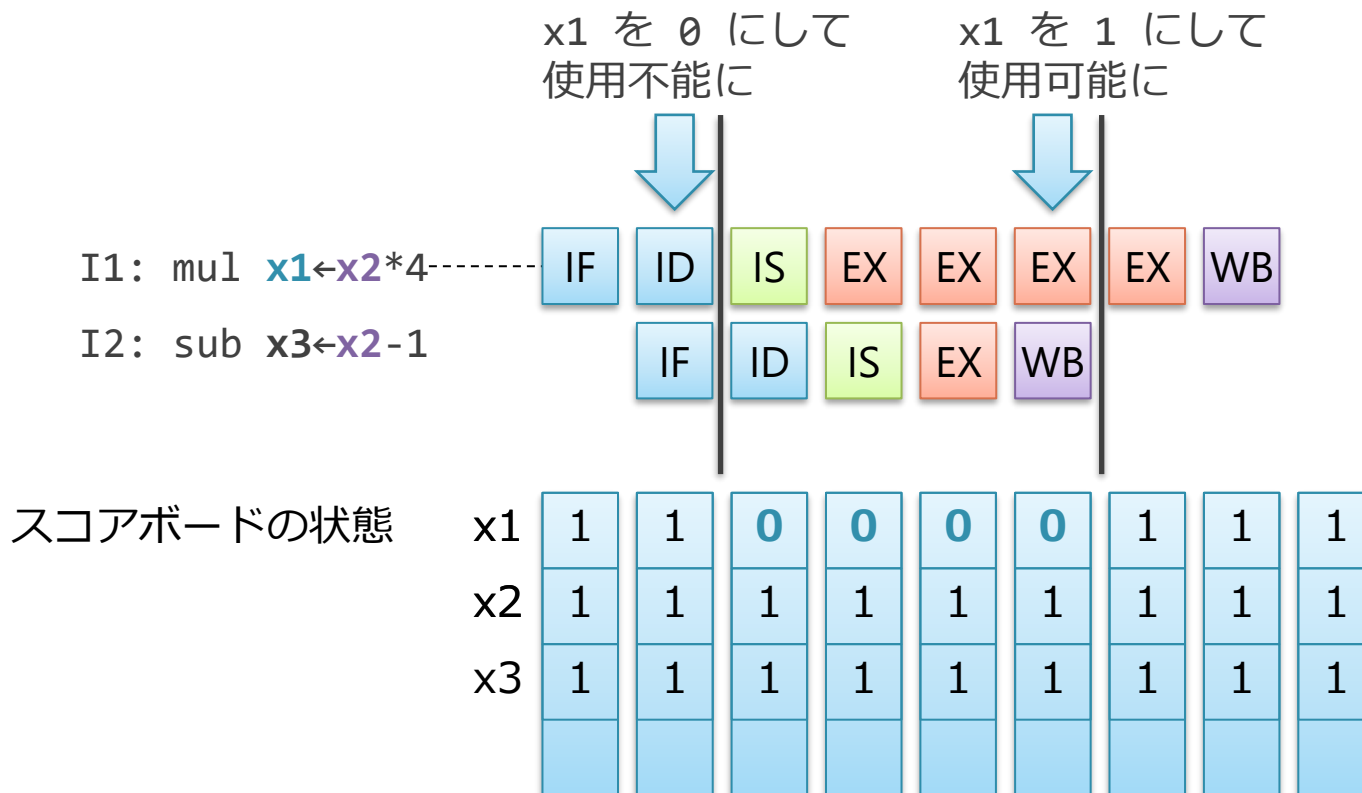
- ◇ ソース x3 とディスティネーション x2 共に使用可能なので, そのまま発行
- ◇ WB の時系列が逆転するが, 最終的に辻褃はあう



4. 依存がない場合

■ I2 のデコード時：

- ◇ ソース x3 とディスティネーション x3 共に使用可能なので、そのまま発行
- ◇ WB の時系列が逆転するが、最終的に辻褄はあう



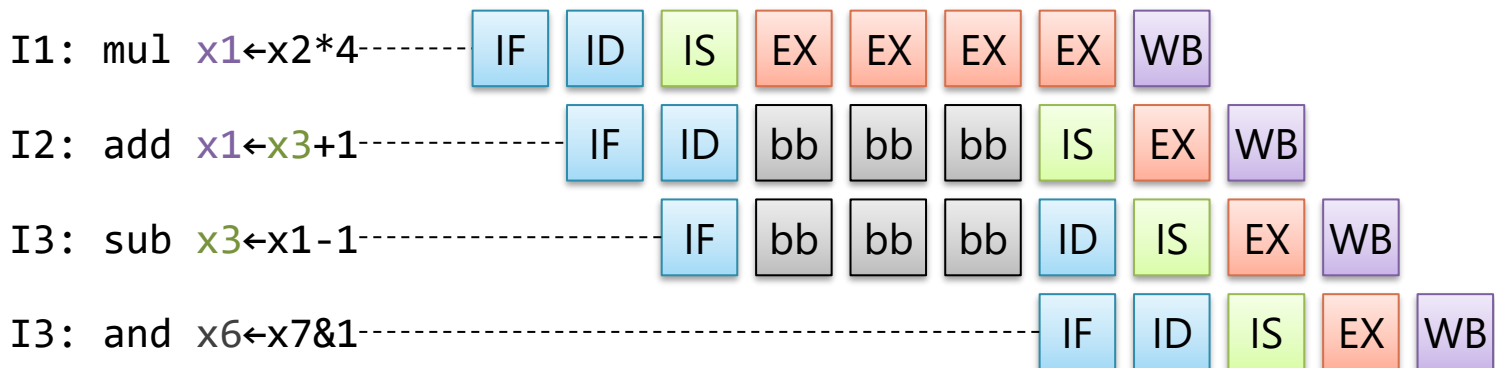
in-order 発行/out-of-order 完了

■ 発行が in-order :

- ◇ 偽を含めて何らかの依存があると, パイプライン全体をストール
 - ある命令の発行は, 前の命令の発行を追い越せない
- ◇ 下図では, IS は必ず右下に伸びる

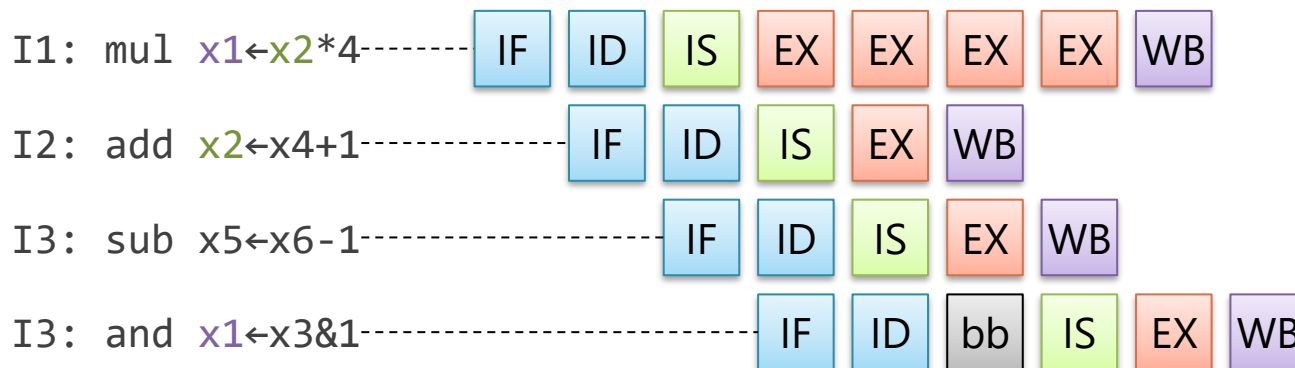
■ これにより, 逆依存は常に守られる

- ◇ IS と次の命令の IS が in-order
- ◇ = IS と次の命令の WB も in-order



in-order 発行/out-of-order 完了

- 完了が out-of-order :
 - ◇ スコアボードでオペランドが使用可能なら発行
 - 結果として完了は out-of-order に
 - ◇ 下図では, WB のタイミングは, 命令の順序とは無関係



in-order 発行/out-of-order 完了

- 一般に「in-order CPU」と言った場合はこのこと
 - ◇ 一部でこれを「out-of-order 実行」と呼んでいる場合があるが普通は違う
 - ◇ in-order 発行/out-of-order 完了 が正確
- 比較的単純な構成で、そこそこ性能がでる
 - ◇ 依存がなければ命令を追いついて実行できる
 - コンパイラが理想的にコードを吐けば、高い性能が出せる
 - ◇ スコアボードによる制御は比較的単純

動的スケジューリングを行うプロセッサの分類

1. in-order 発行/in-order 完了
2. in-order 発行/out-of-order 完了
3. out-of-order 発行/out-of-order 完了

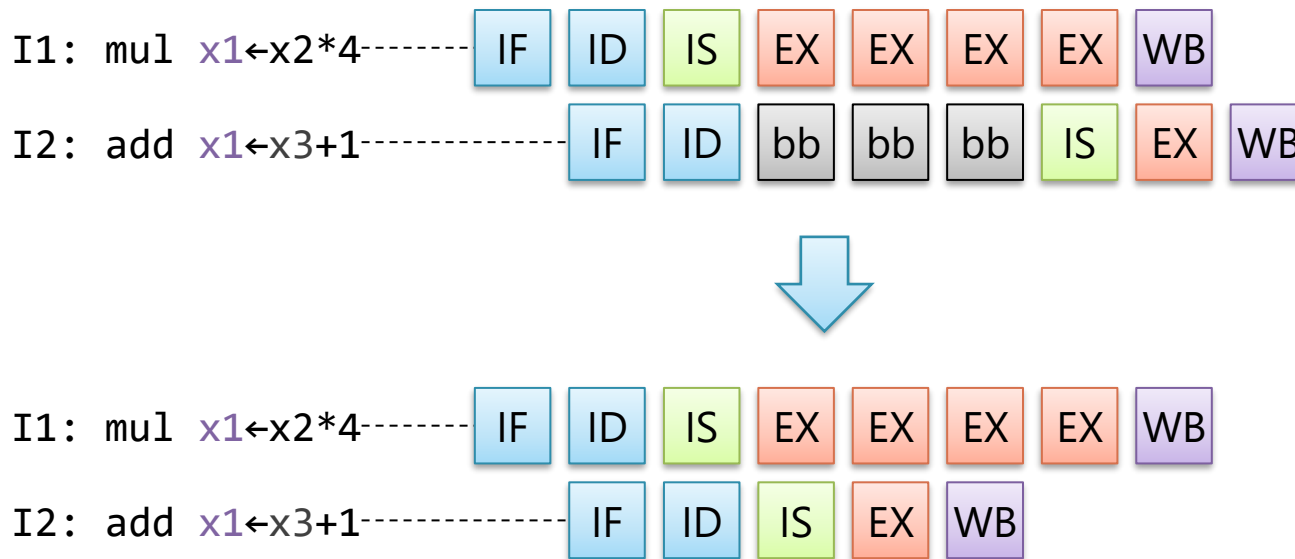
out-of-order 発行/out-of-order 完了

1. レジスタ・リネーム
2. out-of-order 発行機構

- in-order 発行/ out-of-order 完了による, 下記をなんとかしたい
 1. 出力依存 (WAW) があると止まってしまう
 2. なんらかの依存がある命令があるとそこでパイプライン全体が止まってしまう

1. 出力依存（WAW）があると止まってしまう

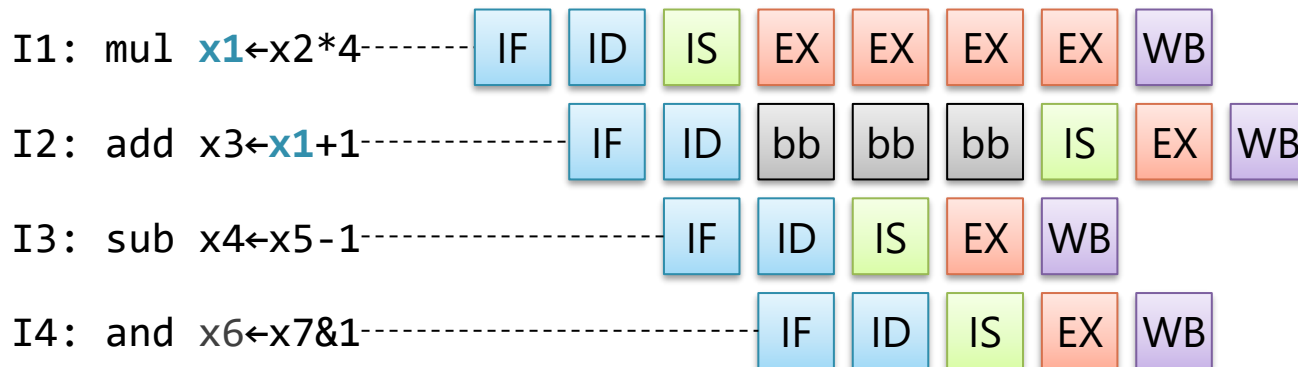
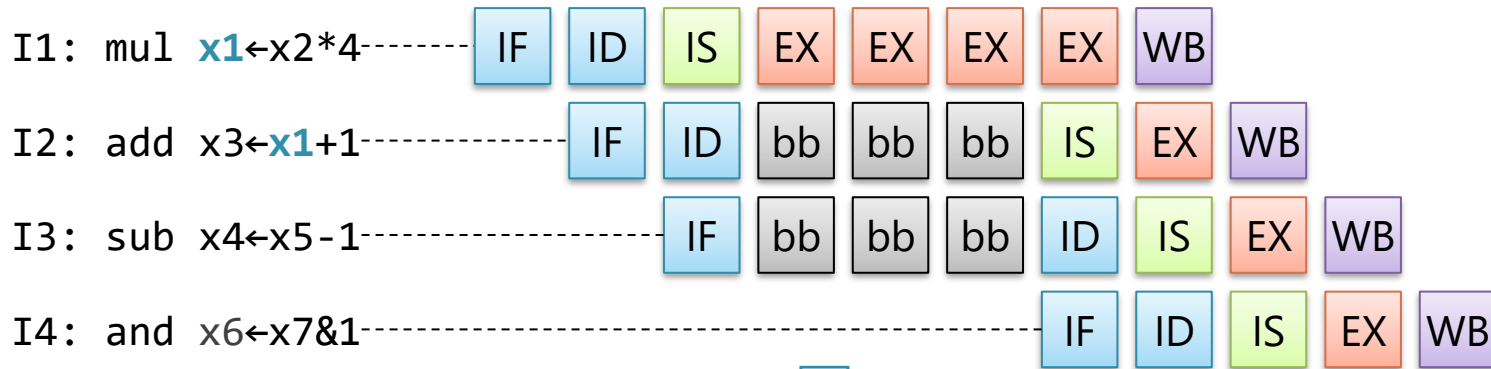
- 出力依存 = 同じレジスタに書き込むことで発生する依存
 - ◇ I2 の演算自体は I1 と関係なくできる
 - ◇ 同じ場所書き込むので、順序を揃えなければいけない
 - これをなんとかしたい



2. なんらかの依存がある命令があるとそこでパイプライン全体が止まってしまう

■ I3 と I4 は I1 と無関係なので、裏で実行したい

◇ 真の依存がない場合は、発行を out-of-order に行いたい

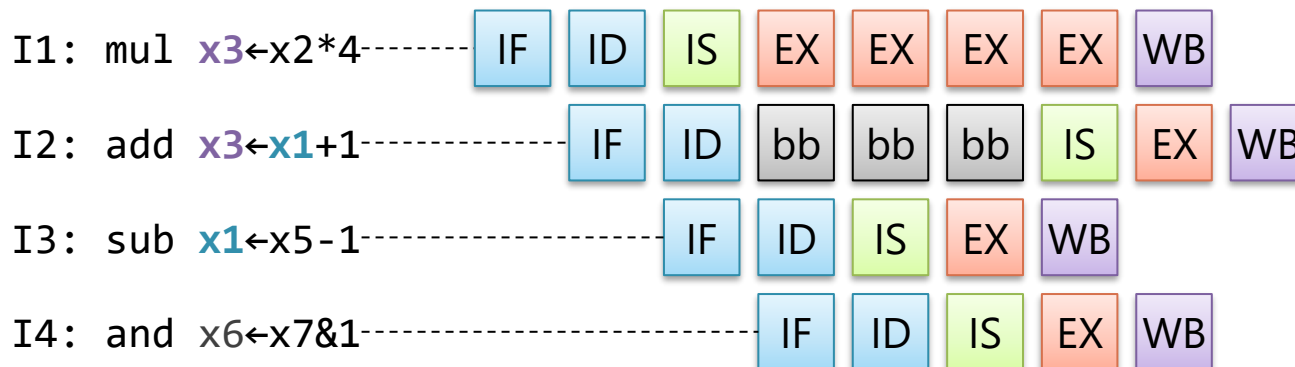


発行を out-of-order にした場合

- 発行を out-of-order にすると, 逆依存 (WAR) を守れない

◇ I3 が I2 よりも先に完了してしまう

- 発行を in-order にすることにより 逆依存 の違反を防いでいたとも言える



out-of-order 発行を行う方式

1. トマスロ方式

- ◇ IBM System/360 で初めて実装された
- ◇ トマスロ (Tomasulo) のアルゴリズムとして知られる

2. 物理レジスタ方式

- ◇ MIPS R10000 で初めて実装された (と思う)

■ 現在主流なのは後者

- ◇ 構造がより単純で設計しやすく性能も出やすいから
- ◇ 文献では2者の関係があまり書かれていないが、
これらはかなり構造が違う

■ とりあえず以降では後者の物理レジスタ方式の説明を行う

レジスタ・リネーム

- 目的：出力依存と逆依存を取り除く

- ◇ 真の依存にのみ従って発行を行うことができるように

- 方針：レジスタの名前を付け替える

- ◇ 偽の依存の原因 = 同じレジスタの使い回し

- ◇ ディスティネーションに専用のレジスタを動的に毎回新しく割り当てる

- レジスタ番号がかぶらないので、
他の命令との間で出力依存や逆依存は生じなくなる

I1: mul **x3**←x2*4

I2: add **x3**←**x1**+1

I3: sub **x1**←x5-1

I4: and x6←x7&1



I1: mul **p20**←p12*4

I2: add **p21**←x11+1

I3: sub **p22**←p15-1

I4: and **p23**←p17&1

レジスタ・リネーム

■ 論理レジスタ：

- ◇ 命令セットで定義されているレジスタ
- ◇ プログラマから見える

■ 物理レジスタ：

- ◇ レジスタ・リネームによって割り当てられる内部のレジスタ
- ◇ 通常論理レジスタの数倍程度の数を用意する
- ◇ プログラマからは見えない

I1: mul **x3**←x2*4

I2: add **x3**←**x1**+1

I3: sub **x1**←x5-1

I4: and x6←x7&1



I1: mul **p20**←p12*4

I2: add **p21**←x11+1

I3: sub **p22**←p15-1

I4: and **p23**←p17&1

レジスタ・リネームのための機構

■ RMT (Register Map Table)

- ◇ 論理レジスタと物理レジスタの対応関係を保持する表
- ◇ インテル用語だと RAT (Register Alias Table)
 - こっちの方が最近をよく見るかもしれない

■ フリーリスト

- ◇ 現在使用できる物理レジスタ番号のリスト
 - FIFO で構成する
- ◇ ここから物理レジスタを確保して割り当てる
- ◇ 使用し終わった物理レジスタ番号は、フリーリストに返却

■ リネーム・ステージ

- ◇ デコード・ステージの後に存在
- ◇ 上記を使って、レジスタをリネームする

レジスタ・リネームの動作（１）

I1: mul $x3 \leftarrow x2 * 4$ IF ID **RN**
I2: add $x3 \leftarrow x1 + 1$ IF ID
I3: sub $x1 \leftarrow x5 - 1$ IF
I4: and $x6 \leftarrow x7 \& 1$

RMT

x1	10	10	10
x2	11	11	11
x3	12	12	20

フリーリスト

20	20	
23	23	23
25	25	25
...

■ I1 のリネーム (ディスティネーション)

- ◇ フリーリストから p20 を取り出し、ディスティネーションに割り当て
- ◇ RMT の x3 を書き換える

レジスタ・リネームの動作（２）

I1: mul x3 ← x2 * 4
p20 ← p11 * 4

I2: add x3 ← x1 + 1

I3: sub x1 ← x5 - 1

I4: and x6 ← x7 & 1

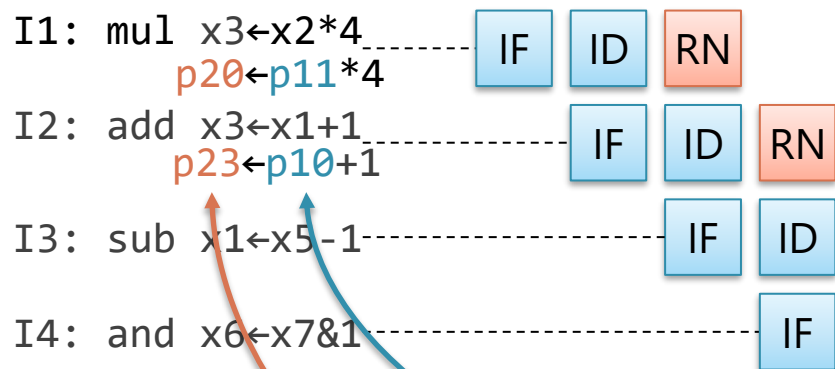
RMT	x1	10	10	10
	x2	11	11	11
	x3	12	12	20
	

フリーリスト

20	20	
23	23	23
25	25	25
...

- I1 のリネーム（ソース）
- ◇ RMT の x2 を参照して、
x2 を p11 に付け替える

レジスタ・リネームの動作（3）



RMT

x1	10	10	10	10
x2	11	11	11	11
x3	12	12	20	23
...

フリーリスト

20	20		
23	23	23	
25	25	25	25
...

■ I2 のリネーム

- ◇ フリーリストから p23 を取り出し
ディスティネーションに割り当て
- RMT の x3 を書き換える
- ◇ RMT の x1 を参照して,
ソース を p10 に書き換える

レジスタ・リネームのまとめ

- 各命令のディスティネーションに専用のレジスタを与える
 - ◇ レジスタ番号がかぶらないので、
他の命令との間で出力依存や逆依存は生じなくなる
 - ◇ RMT やフリーリストにより実現
- リネームを済ました後は、真の依存だけを考えて発行すればよい

out-of-order 発行/out-of-order 完了

1. レジスタ・リネーム
2. **out-of-order 発行機構**
 1. スケジューラの作り方
 2. 性能への影響
 3. トマスロ方式との違い

in-order 発行と out-of-order 発行の違い

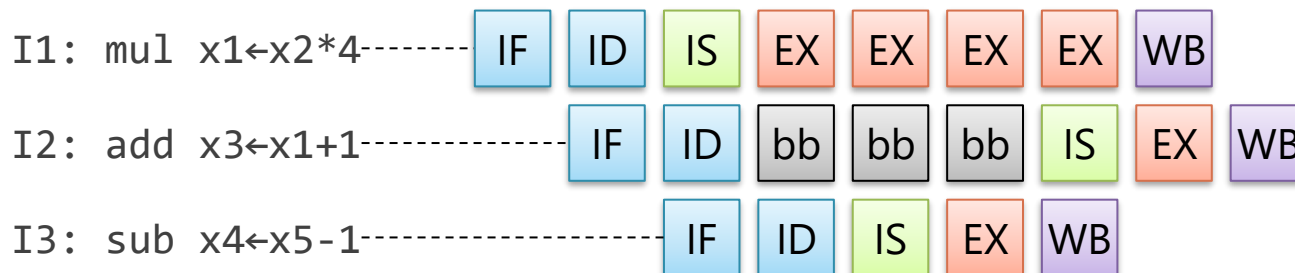
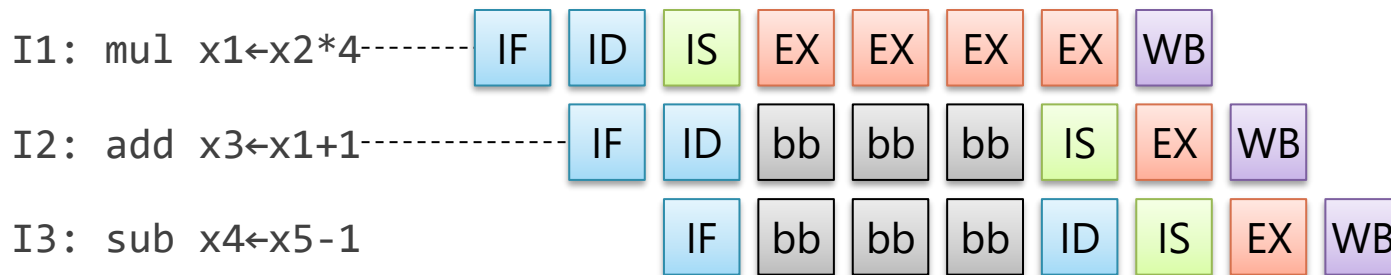
in-order 発行 :

◇ IS ステージにいる 1 命令 の依存だけを考えれば良い

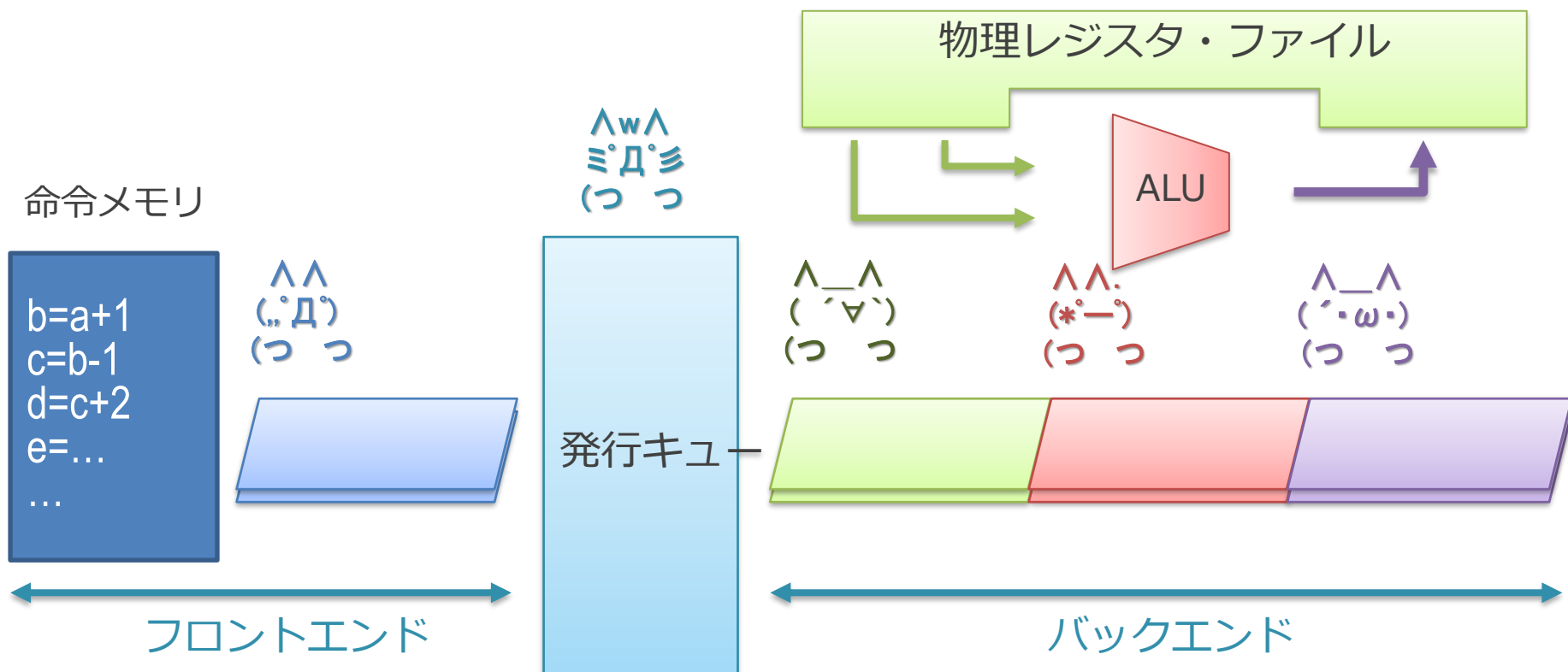
out-of-order 発行 :

◇ 発行待ち命令 (I2) をどこかおいておく必要 = バッファがある

◇ I2 の依存を監視して満たされたら, そのタイミングで発行



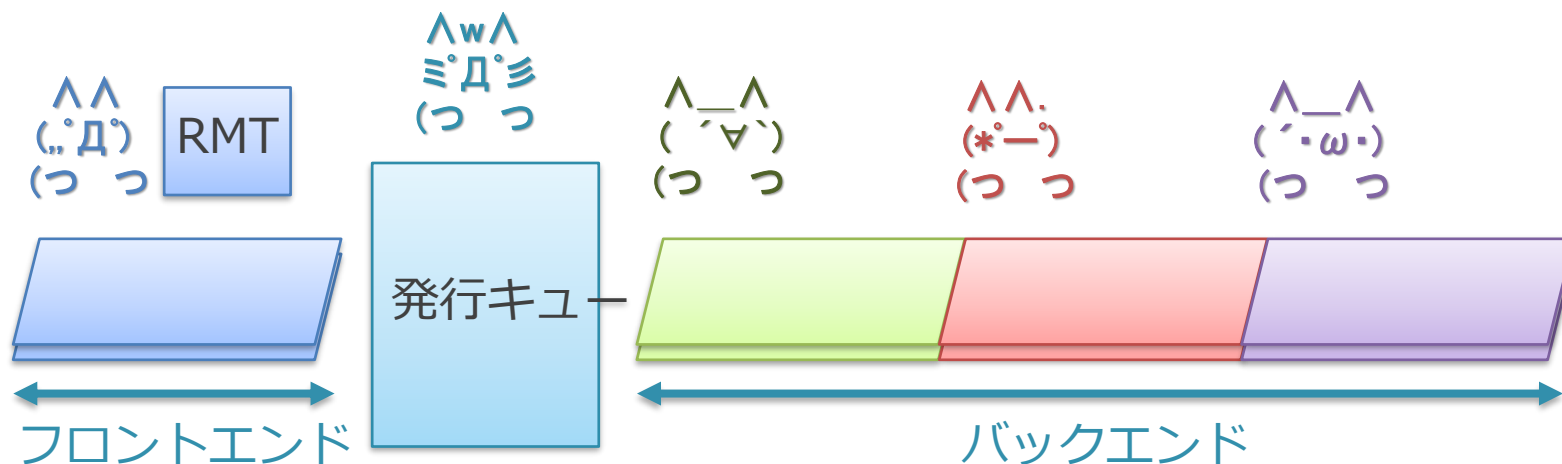
out-of-order 発行を行う CPU の構造



■ 発行キューによって前後に分離された構造を持つ

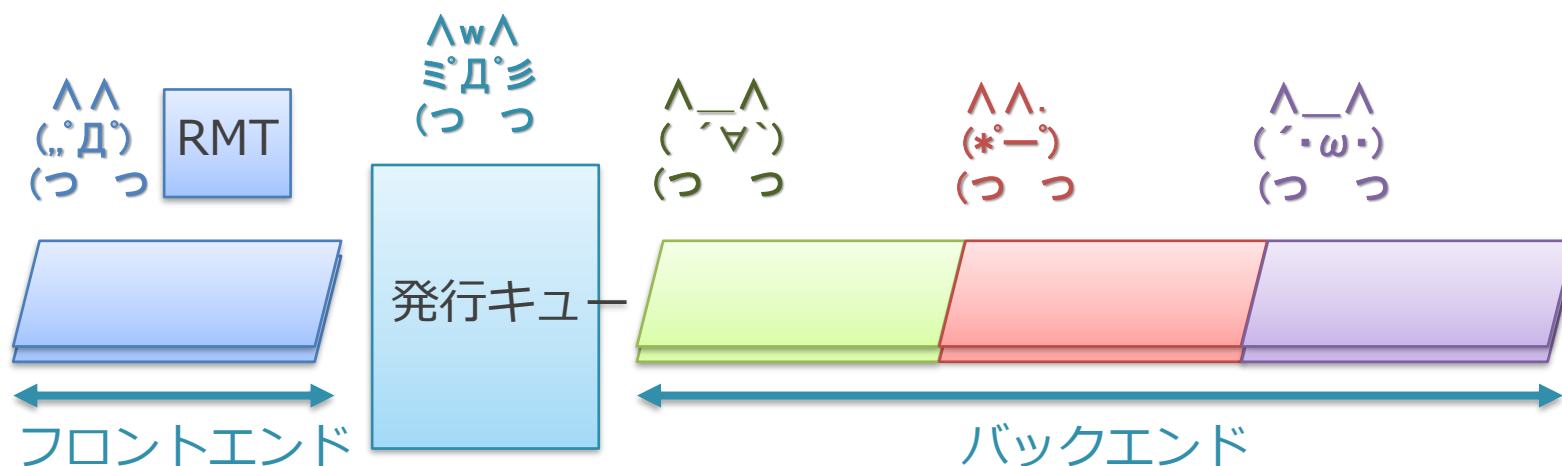
1. フロントエンド: 命令をフェッチ, リネーム
2. 発行キュー: 発行待ち命令の待ち合わせのバッファ
3. バックエンド: 命令を実行

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してリネーム
2. 発行キューにディスパッチ
3. 発行可能なものから順にバックエンドに命令を発行
4. レジスタを読んで演算器で実行し書き戻す

ディスパッチ

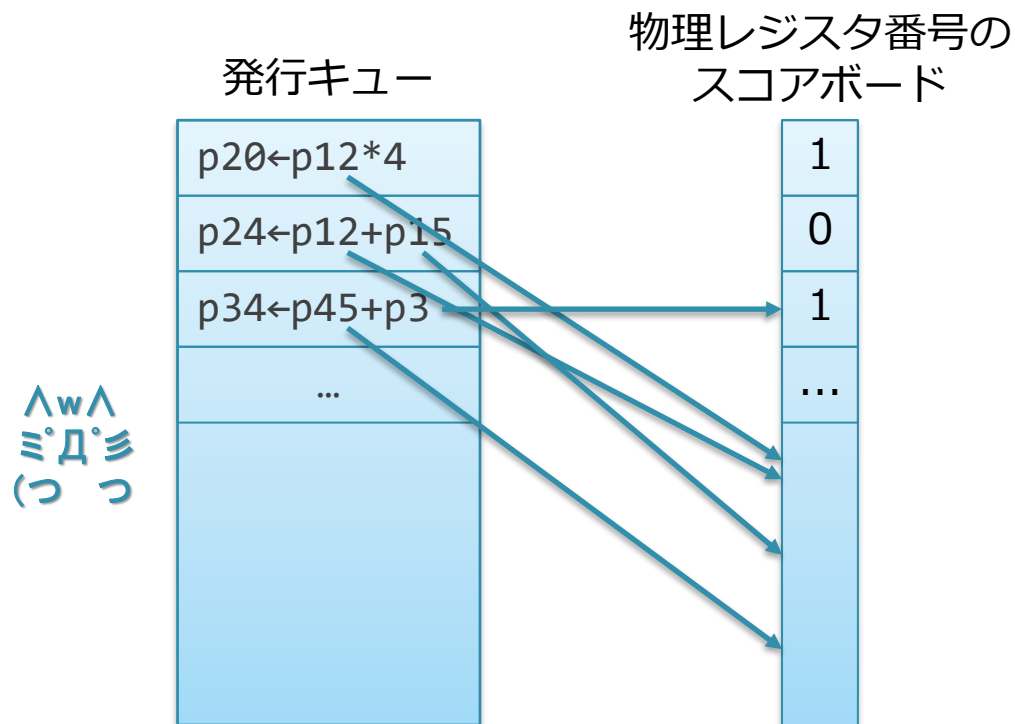


- 別にある発行キューのフリーリストから空きエントリ番号を確保
 - ◇ 物理レジスタのフリーリストの場合と同様
 - ◇ 発行したら返却

バックエンドへの命令の発行

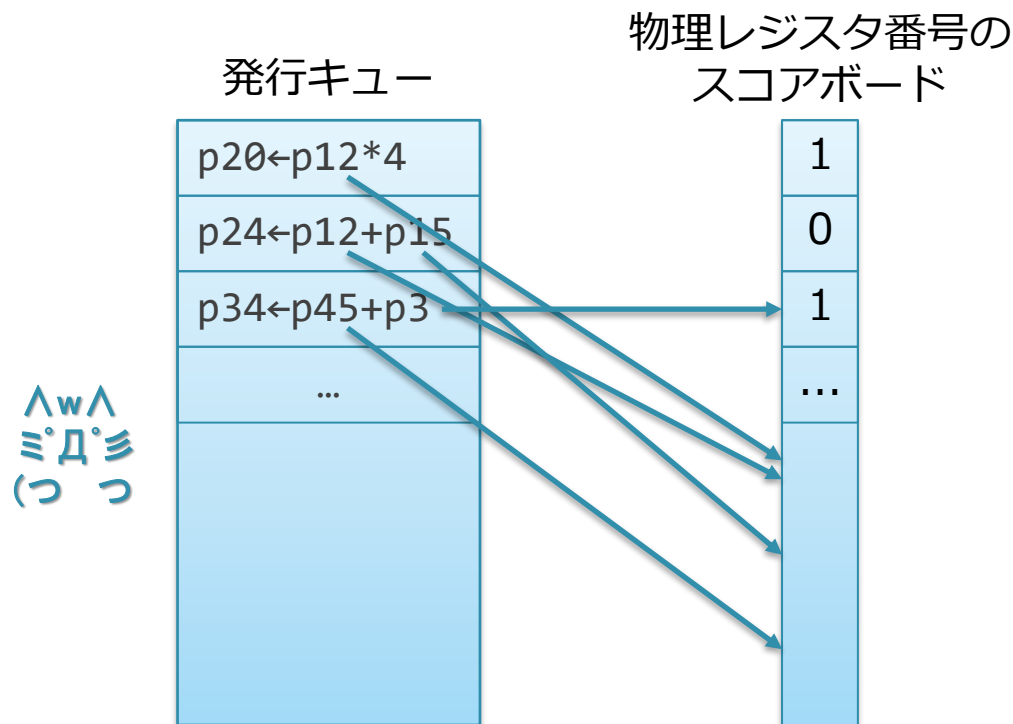
- スケジューラ（発行キュー）の作り方
 1. スコアボードを使った方法
 2. 連想検索を使う方法
 3. 行列を使う方法

発行キューのナイーブな実装



- 発行可能なものから順にバックエンドに命令を発行
- ナイーブには、スコアボードで実装できる
 - ◇ 発行キューの各エントリの命令がそれぞれスコアボードを監視

ナイーブな実装の問題

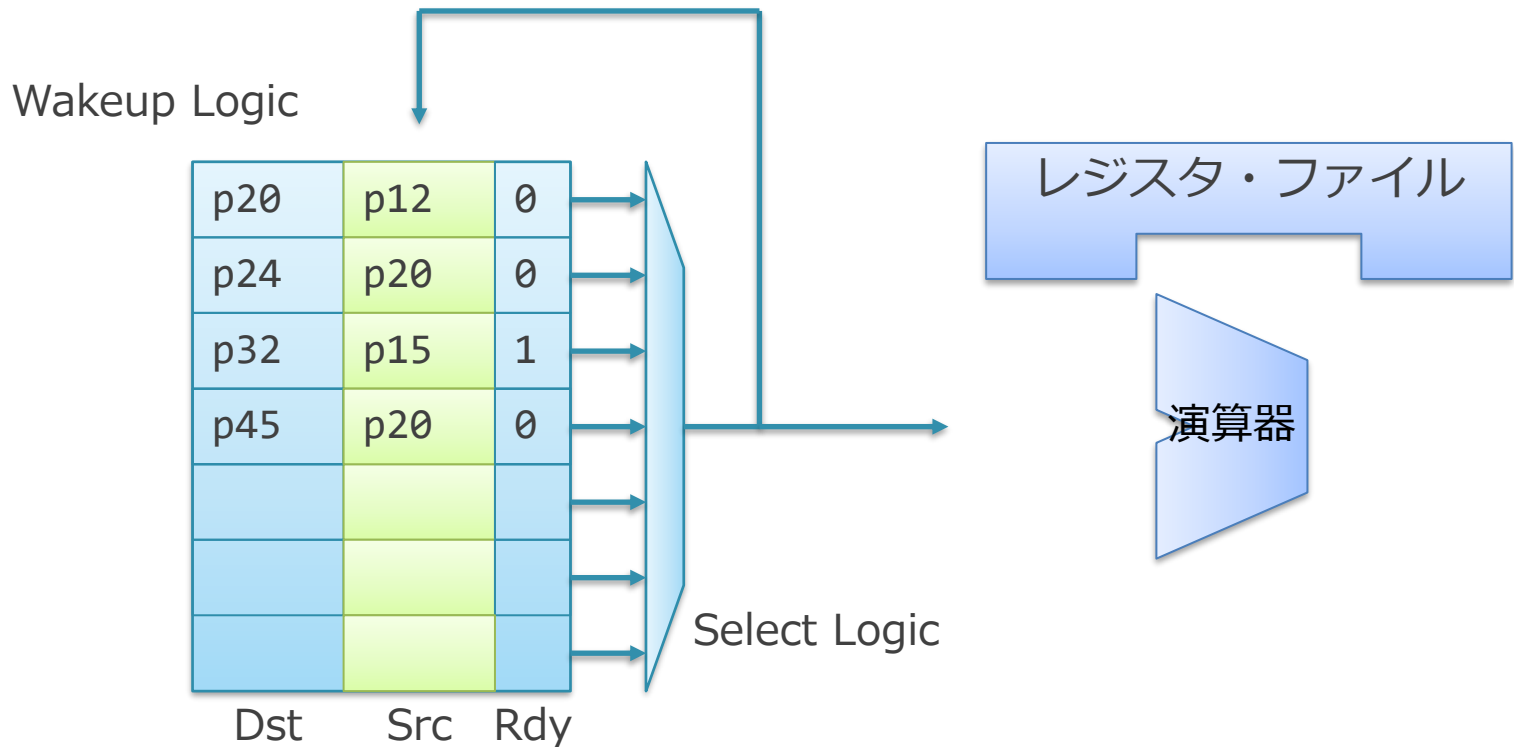


- スコアボードによる実装はスケールしない
 - ◇ 発行キューのエントリ数：数十から100命令
 - ◇ スコアボードはソース・オペランドの数だけ独立に読まれる
= ものすごい数のポート数（ $100 * 3$ とか）になる
 - ◇ メモリのサイズはポート数の二乗に比例するのでヤバイ

バックエンドへの命令の発行

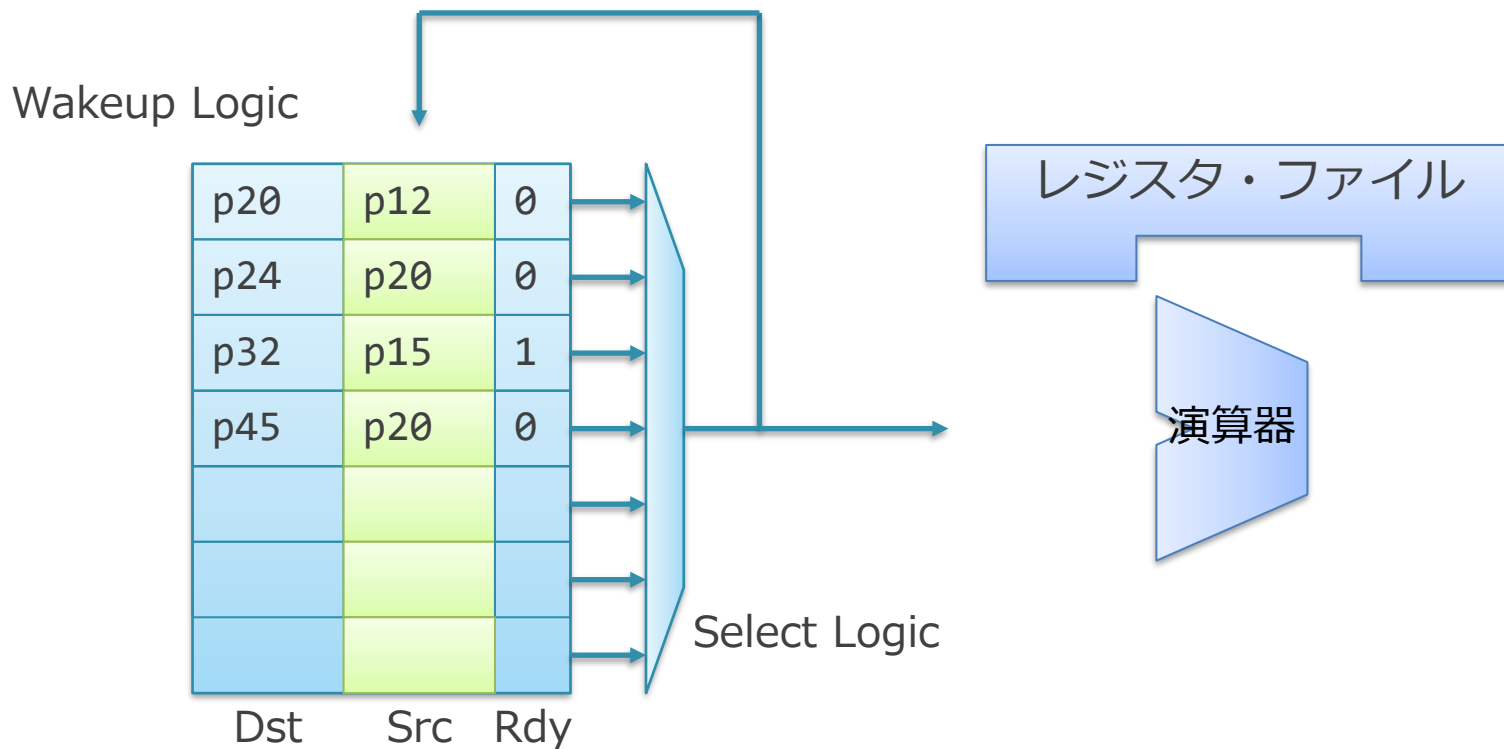
- スケジューラの作り方
 1. スコアボードを使った方法
 2. 連想検索を使う方法
 3. 行列を使う方法

連想検索による実装



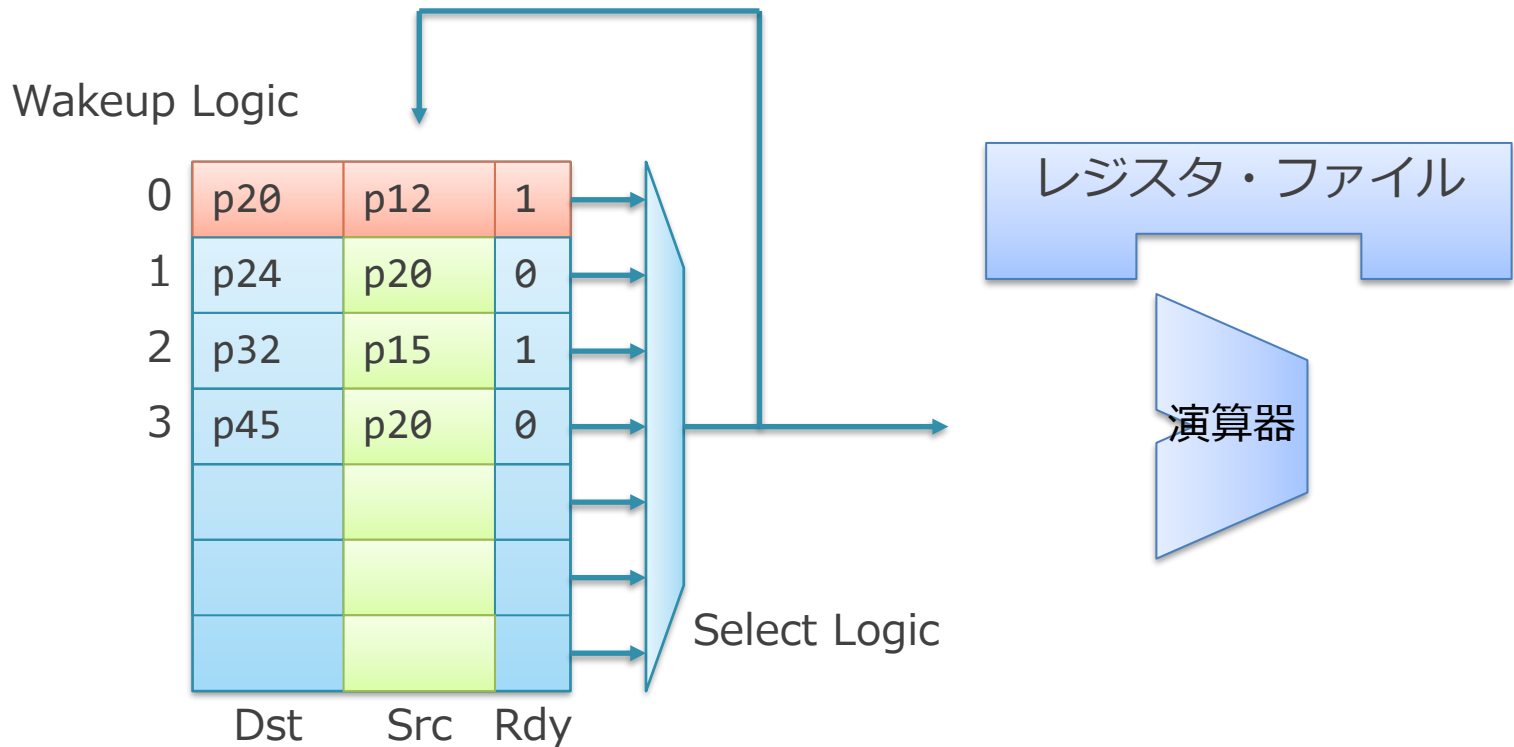
- Wakeup Logic : 寝ている（発行待ち）命令を起こす機構
 - ◇ Dst : ディスティネーションの物理レジスタ番号
 - ◇ Src : ソースの物理レジスタ番号（簡単のためここでは1つ）
 - ◇ Rdy : 依存元命令が実行され，発行準備ができているか

連想検索による実装



- Select Logic : 起きた命令から発行する命令を選ぶ機構
 - ◇ 発行準備ができている命令 (Rdy=1) から 1 つを選んで発行
 - ◇ インテル用語だと Picker

発行キューの動作（１）：Select



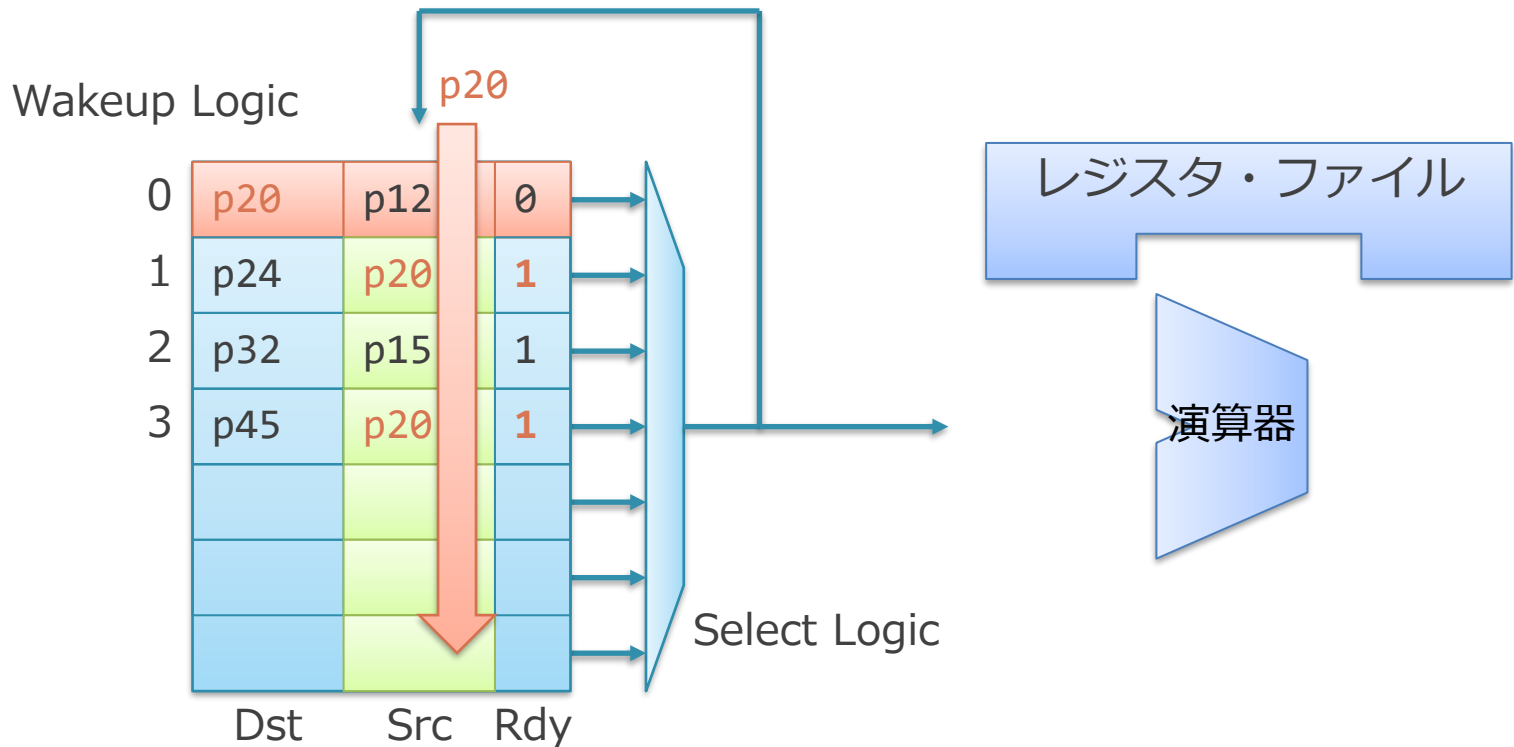
- 現在エントリ 0 番と 2 番で発行準備ができています

- ◇ Select Logic により, 0 番を選択

- ◇ 0 番の内容（命令）がバックエンドに発行される

- （理想的にはプログラム内の命令順に優先度を付けて選ぶが、ハードが複雑なので適当に番号が若いものを選ぶことも多い

発行キューの動作（２）：Wakeup



- Select して発行した命令のディスティネーション番号を Wakeup Logic 全体にブロードキャスト
 - ◇ 各エントリのソースオペランドの番号と一致比較を行う
 - ◇ 一致した場合は真の依存が満たされた = Rdy を 1 に (wakeup)

バックエンドへの命令の発行

- 発行キューの作り方
 1. スコアボードを使った方法
 2. 連想検索を使う方法
 3. 行列を使う方法

行列を使った方式

■ モチベーション：連想検索を使った方式の問題

◇ 遅延が大きい

□ 命令のセレクト後に物理レジスタ番号を読み出す必要がある

◇ 消費電力が大きい

□ ものすごい数の比較器が同時に稼働している

行列を使った方式

■ マトリクス・スケジューラ

- ◇ Masahiro Goshima et al., A High-Speed Dynamic Instruction Scheduling Scheme for Superscalar Processors, MICRO 2001
- ◇ Sassone, Peter G. et al., Matrix Scheduler Reloaded, ISCA 2007

■ 行列状の構造を使って wakeup を行う

- ◇ 各命令が依存元命令のビットベクタもつ
- ◇ レジスタ番号を介さずに命令 to 命令で直接 wakeup
- ◇ 少なくともインテルや IBM の CPU ではこれを使っている

マトリクス・スケジューラ

I_0 ld $p11 = *(p10)$

I_1 sll $p12 = p11 \ll 1$

I_2 sll $p13 = p11 \ll 2$

I_3 add $p14 = p12 + 1$

		producer				rdy
		I_0	I_1	I_2	I_3	
consumer	I_0					1
	I_1	1				0
	I_2	1				0
	I_3		1			0

- 行, 列がそれぞれ発行キューのエントリに対応
 - ◇ 行 : コンシューマ 列 : プロデューサ
- プロデューサとコンシューマの交点により依存関係を表す
 - ◇ 各行は, その命令が依存している列に 1 を立てる

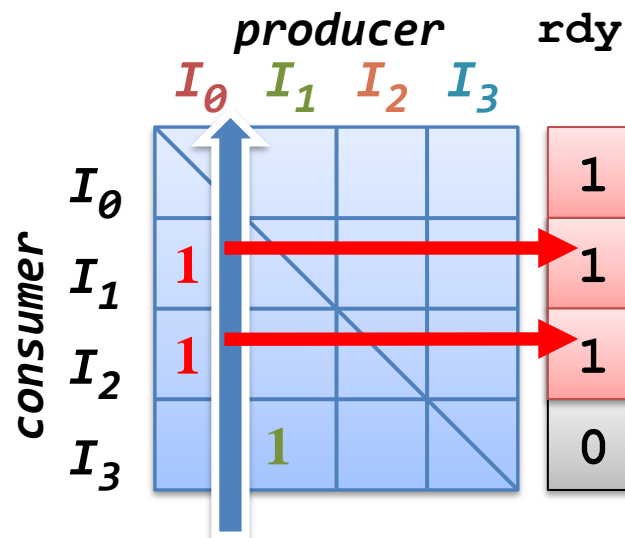
マトリクス・スケジューラ

I_0 ld $p11 = *(p10)$

I_1 sll $p12 = p11 \ll 1$

I_2 sll $p13 = p11 \ll 2$

I_3 add $p14 = p12 + 1$



■ Wakeup の動作

- ◇ 発行された命令の列をアサートする
- ◇ 各行のその列のビットが立っていたら rdy を 1 に

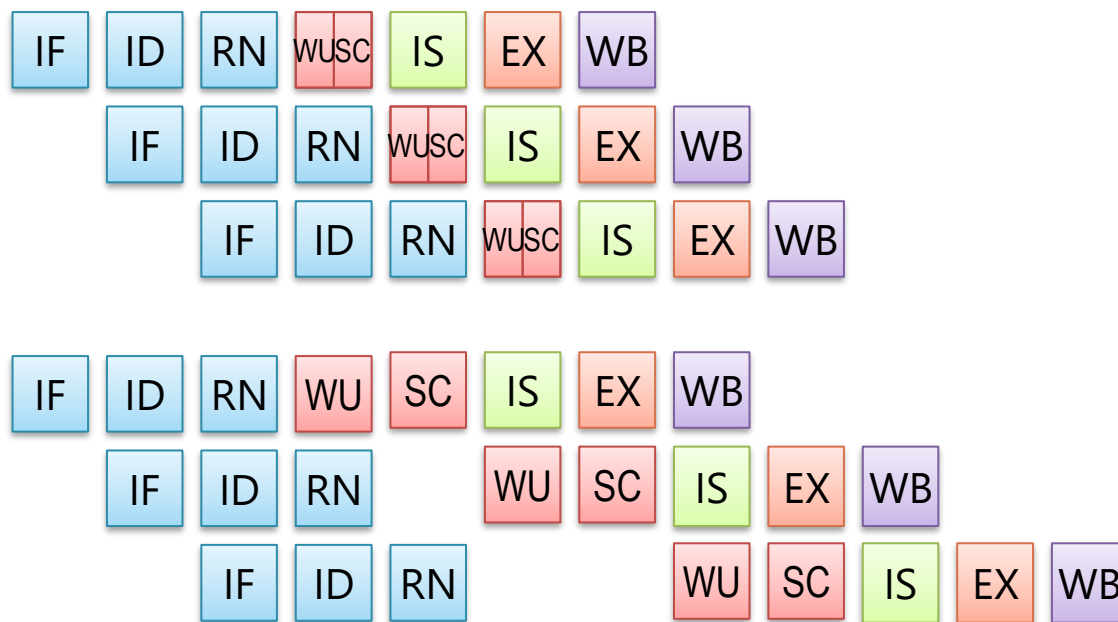
■ RAM の読み出しに近い構造で実現できる

out-of-order 発行/out-of-order 完了

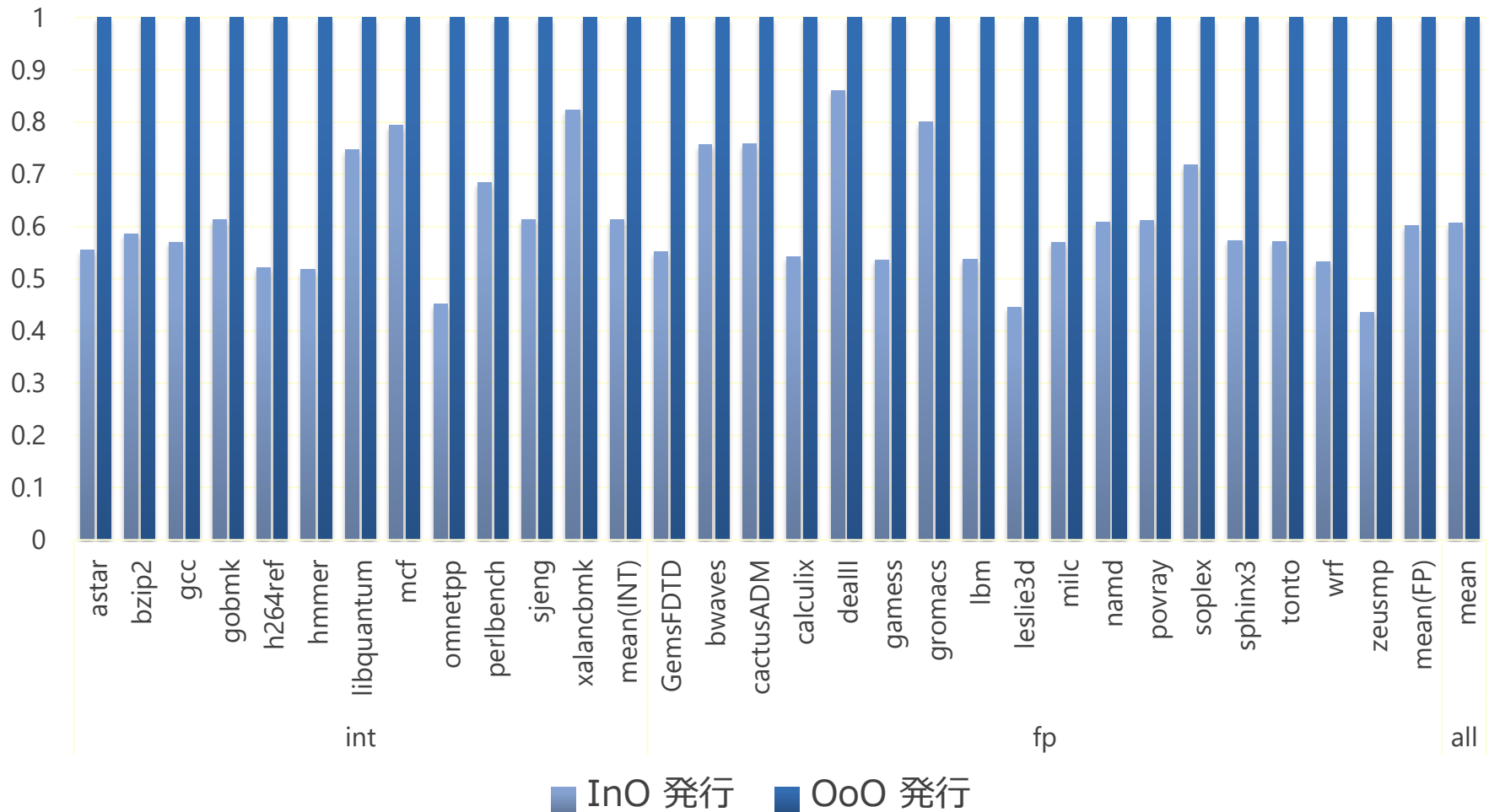
1. レジスタ・リネーム
2. out-of-order 発行機構
 1. スケジューラの作り方
 2. **性能への影響**
 3. トマスロ方式との違い

発行キューの特性

- Wakeup-Select は 1 サイクルで終わる必要がある
 - ◇ ここをパイプライン化すると、依存がある命令の実行が遅くなる
- 広範囲へのブロードキャストや選択などを伴う
 - ◇ クリティカルパスになりやすい



in-order 発行と out-of-order 発行の性能 (IPC) SPEC CPU 2006 より



■ OoO 発行の CPU の性能で正規化

◇ InO 発行の CPU の性能は、平均で OoO 発行の60%程度

out-of-order 発行/out-of-order 完了

1. レジスタ・リネーム
2. out-of-order 発行機構
 1. スケジューラの作り方
 2. 性能への影響
 3. トマスロ方式との違い

Out-of-order 発行の方式

- ここまで話をしてきたのは物理レジスタ方式
 - ◇ 現在はこれが主流
 - ◇ 比較のためにトマスロ方式も説明

トマスロ方式

■ 構成要素：

◇ 論理レジスタ・ファイル

- 論理レジスタ番号でアクセスされる
- レジスタ値 or タグが置かれる
- タグ：命令を識別する ID

◇ リザーベーション・ステーション

- ソースオペランドのレジスタ値 or タグが置かれる
- 発行キューと同様に値の待ち合わせを行う

トマスロ方式のブロック図

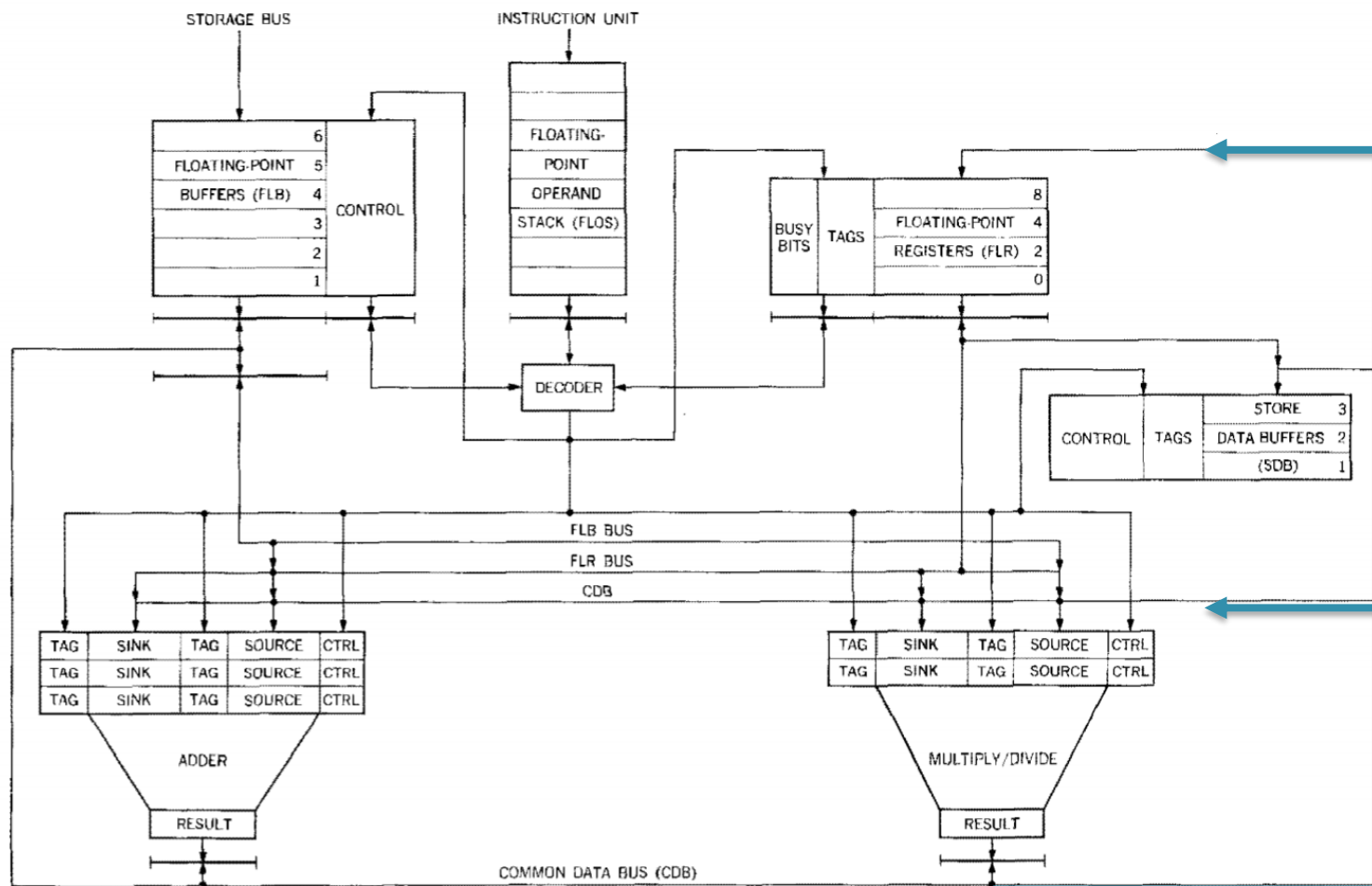


Figure 4 Data registers and transfer paths, including CDB and reservation stations.

- An Efficient Algorithm for Exploiting Multiple Arithmetic Units, IBM Journal of Research and Development, 11(1):25-33, 1967.

トマスロ方式の動作

■ フロントエンド：

1. 論理レジスタ・ファイルを読む
 - もし ready なら値が取れる
 - もし not-ready ならタグが取れる
 - * こっちの場合は RMT に似ている
2. 自分のタグを論理レジスタ・ファイルに書き込む
 - 現在自分は演算中でまだ値はないことを示す
 - いずれ自分が結果を返すことを示すために、識別用のタグを書いておく

トマスロ方式の動作

■ バックエンド：

1. 読めた値 or タグをリザベーション・ステーションに登録
2. ソースが揃ったものから演算器に発行
3. 完了したら,
 1. リザベーション・ステーションに結果をブロードキャスト
 2. 論理レジスタ番号に結果を書き込む

トマスロ方式におけるリネームとスケジュール

■ リネーム

- ◇ 論理レジスタを読んだ際のタグへの置き換えと
リザベーション・ステーションへのコピーにより実現

■ スケジュール

- ◇ リザベーション・ステーションでの待ち合わせにより実現

トマスロ方式と物理レジスタ方式の違い

- トマスロ方式は, in-order 発行を行う CPU からの延長
 - ◇ 論理レジスタ・ファイルを拡張して, 現在待ち合わせ中の場合はそれをタグとして書いておく
 - ◇ 実行結果のデータそのものをバッファ (リザベーション・ステーション) で待ち合わせる
- 物理レジスタ方式は, データとスケジュールを完全に分離
 - ◇ まずリネームをしてしまう
 - ◇ リネーム結果の真の依存にのみ従ってスケジュール

性質の違い

■ トマスロ方式

- ◇ リネームとスケジュールがちょっとまじってる
 - マトリクス・スケジューラとか適用できないと思う
- ◇ レジスタの値は複数箇所に複製されて存在
 - 論理レジスタ・ファイル, リザベーション・ステーション
- ◇ データはフロントエンドとバックエンドでアクセス

■ 物理レジスタ方式

- ◇ リネームとスケジュールはそれぞれ独立に行われる
- ◇ レジスタの値は単一の物理レジスタ・ファイルに格納
- ◇ データはバックエンドでのみアクセス

- 基本的に後者を理解していればよいが、教科書は大概前者で書かれているので違いをまとめておく

out-of-order 発行の残りの話題

- 例外への対応
- ロード・ストアへの対応

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 (なんか一言書いてね)
 - ◇ 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