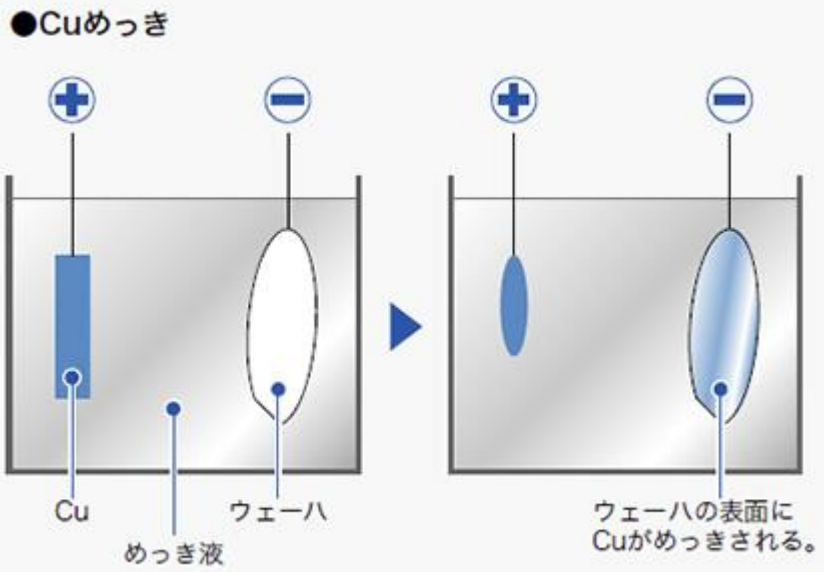
先進計算機構成論 09

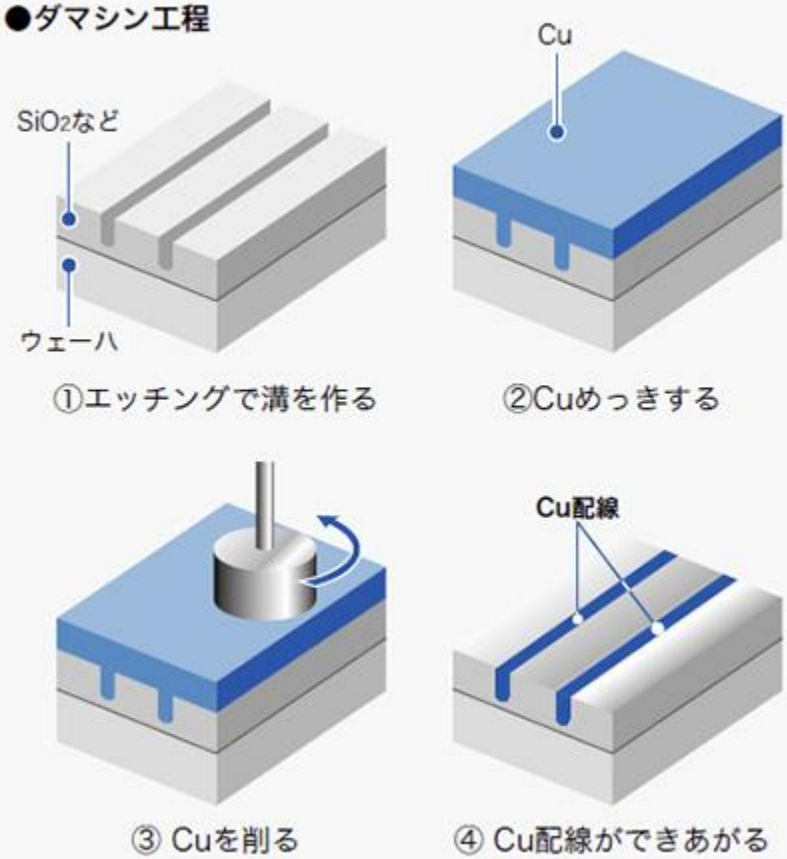
東京大学大学院 情報理工学系研究科 創造情報学専攻
塩谷 亮太
shioya@ci.i.u-tokyo.ac.jp

質問や感想

- 基盤の配線の話で、銅を塗って焼くみたいな話がありましたが、スピンコーティングをしてレジスタで処理するという話で合っているのでしょうか？ コンピュータとは関係ない物理系出身でサンプル作成でレジスタやスピンコーティングをしていたので関係があるのかなと疑問に思いました。

図は https://www.screen.co.jp/spe/technical/guide/cu より





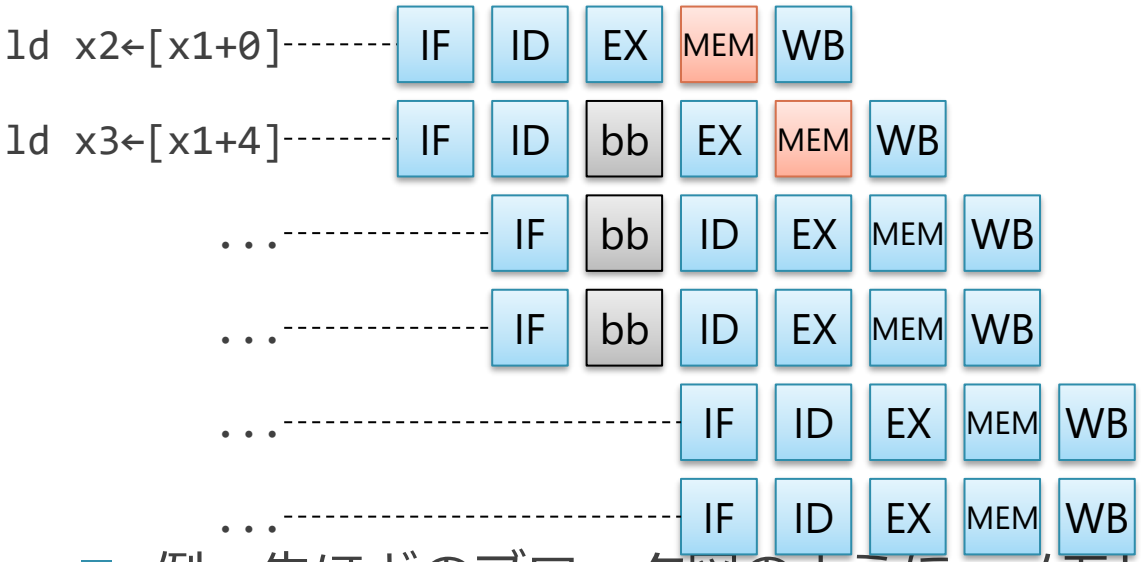
質問や感想

- 静的スケジューリングはあまり性能を出せなかったという点について、最近だと勝手にポインタを触れなくなったりJITコンパイルする言語なら互換性の問題が気にならなくなったりしてある程度静的スケジューリングの欠点が回避できるようになってきていると思うのですが、どこかで活かされていたりしないのでしょうか？結局動的スケジューリングされてしまうので意味が薄い感じですか？

質問や感想

- 構造ハザードがどんなものか忘れてしまっていたので，スーパースカラの構造ハザードが起きる場合 (p36) の内容がよくわかりませんでした.

2. 構造ハザードが起きる場合
（便宜上メモリステージは EX で行うとしてます）



- 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（つまり１個しかない） = 複数同時に来ると止めるしかない
  - ◇ 乗算器，除算器，超越関数の演算器など

質問や感想

- VLIWで実際には何命令ほどを1命令として扱っているんでしょうか？
- 聞き飛ばしてしまったかもしれませんが、Itaniumとx86-64の違いをもう一度説明して欲しいです。Itaniumは独自にライセンスを得るために開発しようとしたものの性能が出なかったCPUで、x86-64は既存のCPUを64bit用に改良しただけで新しくライセンスは与えられないという認識であっていますか。

質問や感想

- 実際のcpuでにレジスタの範囲内で偽の依存関係を解消しているのでしょうか。

- VLIWで数命令を1命令に集約する際に，依存関係のないものを寄せ集めるのはそこそこな計算コストが必要だと感じたのですが，実際にはどのような操作で数命令を集約しているのでしょうか．単純に先頭からn個ずつで区切っていって，依存関係があったらその部分を入れ替えるようなことを行っているのでしょうか.

質問や感想

- 先週の，if とかよりも min を使ったほうが良い？ というのは，min/max は CPU 命令があるからそれを使うと分岐自体が無くなって嬉しい？ という意図でした
  （が，もう一度軽く調べてみたら min を返してくれる CPU 命令が見当たらなかった．記憶違いだった…？）"

質問や感想

- AMDがx86-64を策定したのにどうして一般向けにはintelのほうが主流なんでしょうか？
- 現在のIntel製x86 CPU(Xeon、Corei7など)は、「out of order実行を行うスーパースカラプロセッサ」という理解で正しいでしょうか。
- 今後は組み込みCPUもx86-64が主流になるんですね

質問や感想

- superscalarのscalarって数学のスカラーと関係あるんですかね？
- ゲームの話で思い出しましたが、PS5は今までのPS4と異なりSSD搭載だそうです
  どうして今になって移行したんでしょうかね…
  読み込み時間よりHDDのデータの長期保存を今まで優先してたんでしょうか"

質問や感想

- 自分はセキュリティを専門にしたいのですが、分野としてハードウェアのセキュリティを選択することの将来性についてどう思われますか？またこの視点でチェックした方がいい論文やジャーナル、カンファレンスなどがあれば教えていただきたいです。
- スーパースカラプロセッサは全てout-of-order実行のものを指してると思い込んでました。

質問や感想

- p.64について，例えば，Cでは関数に構造体を渡す場合，ポインタで渡した方が値のコピーがなく高速だと思っていたのですが，ローカル変数にコピーすることによるこのトレードオフはどのようになっていますか.

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;
  ```
  ```

質問や感想

- コンパイラによる命令スケジューリングの具体的なやり方というのがいまいちイメージ出来ないのですが、どうやって命令間の依存関係を捉えたりそれらを並び替えたりしているのでしょうか

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

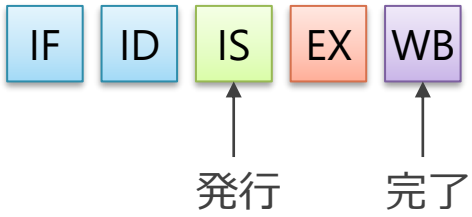
- スカラとスーパスカラ
  - スカラ： １サイクルに単一の命令を実行
  - スーパスカラ：１サイクルに複数の命令を同時実行
- 動的スケジューリングは上記とは直行した概念
  - …ではあるが，普通は動的スケジューリングを行う CPU はみんなスーパスカラ
  - スカラで動的スケジューリングをやっても，複雑さの割にあまり性能がでないからだと思う
- 現在主流の高性能な CPU は，基本的にみなこのタイプ

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

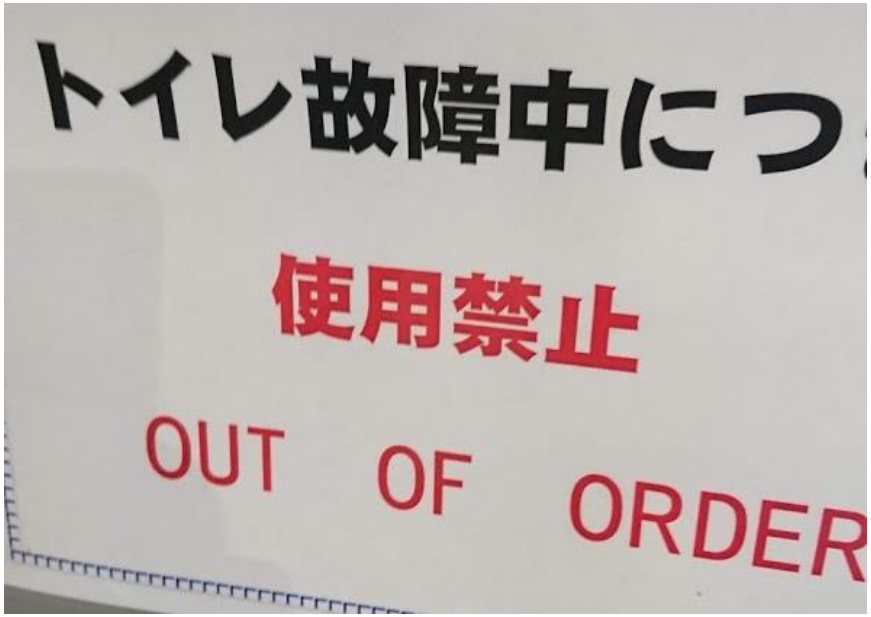
スカラのパイプラインを使って説明

- 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

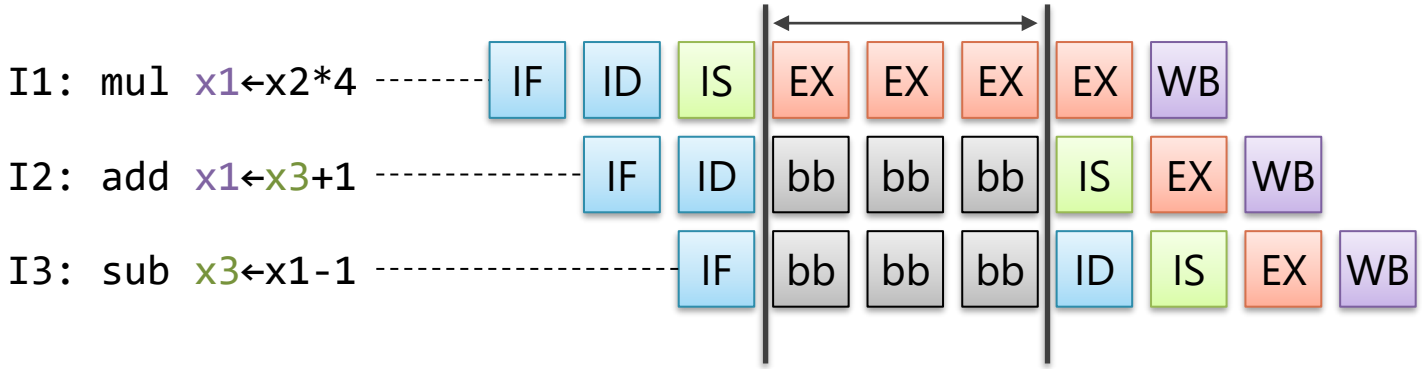
（out-of-order 発行/in-order 完了の組み合わせは
おそらく意味がない

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

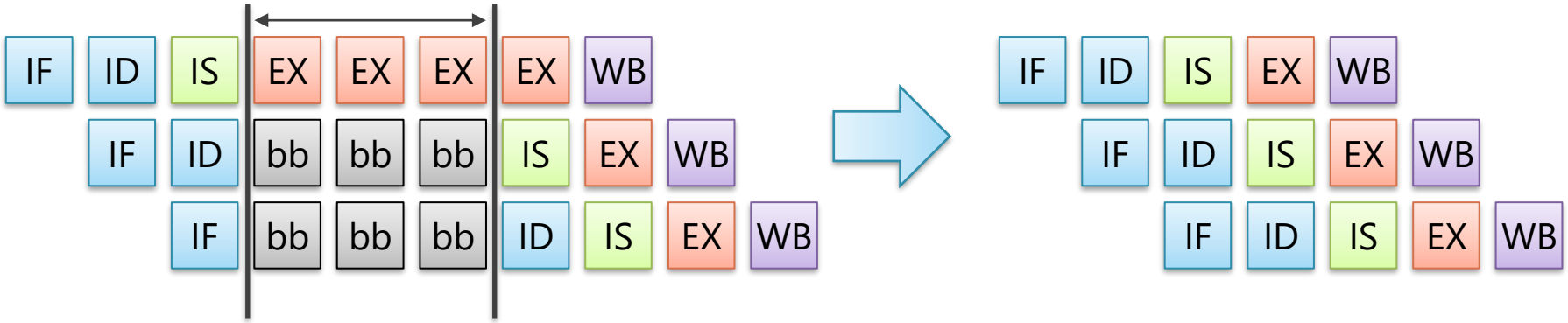
in-order 発行/in-order 完了

- 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

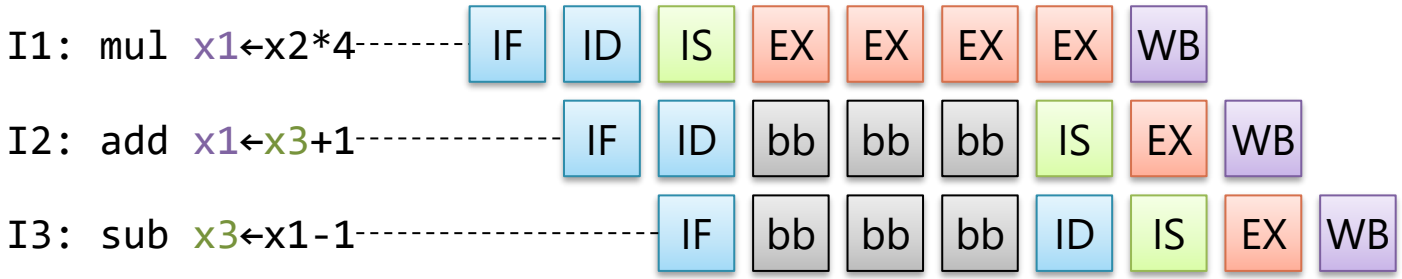
- レイテンシが長い命令現れた場合には，世界中が止まる
  - ◇ = １サイクルの命令のみでパイプラインで実行してるのと等価
- あらゆる追い越しは原理的に起きず，真&偽 の依存は常に守られる



in-order 発行/in-order 完了

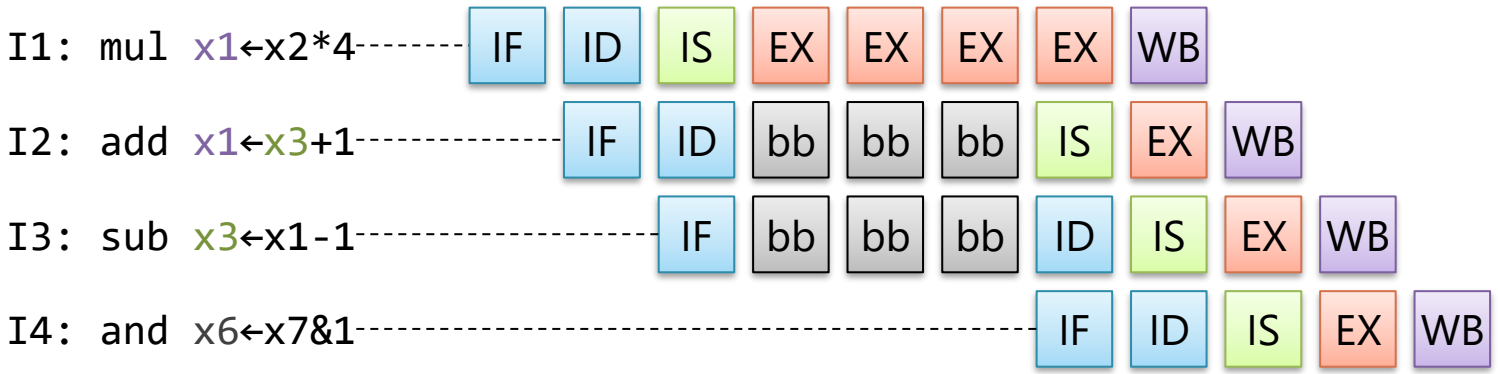
1. 発行が in-order
   - ◇ ある命令の IS が終わった後に，次の命令の IS（とWB）が来る
   - ◇ 逆依存（WAR）が守られる
     - □ write-after-read の違反が起きない

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

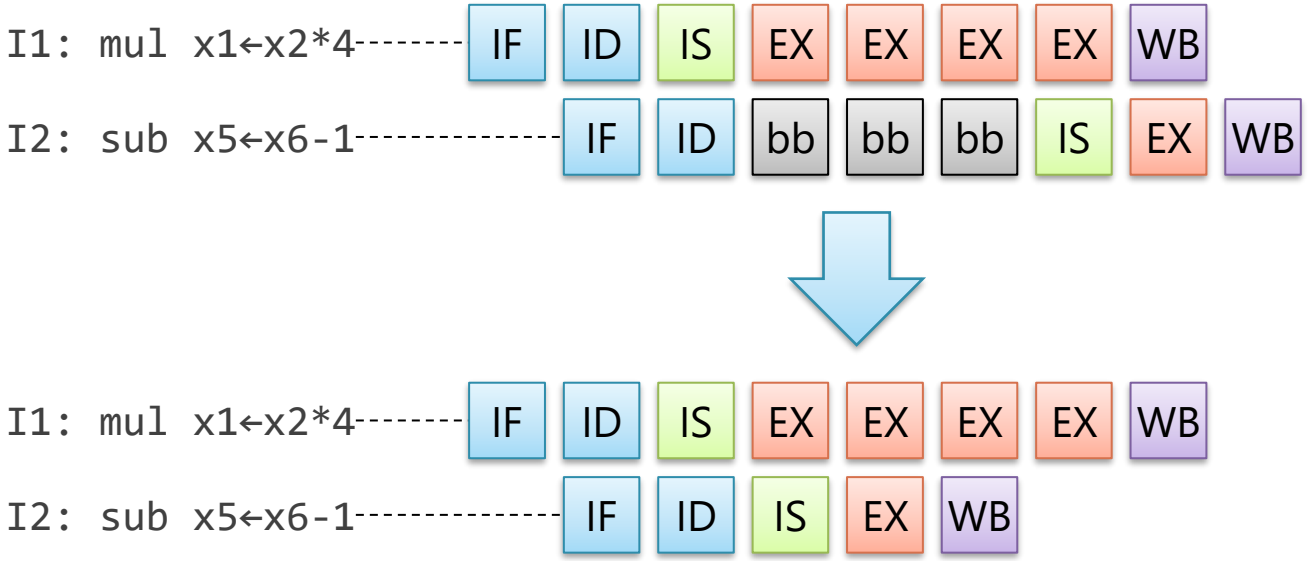
- 性能は最も低い
  - ◇ mul と依存がない命令であっても，mul の完了を待たされる
  - ◇ しかし，最も単純
- I1 に対し，以下は全員待たされる
  - ◇ I2: 出力依存
  - ◇ I3: 逆依存
  - ◇ I4: 依存なし



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

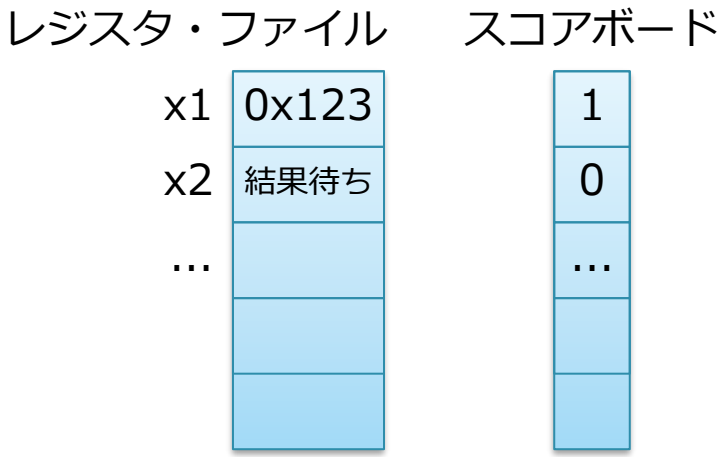


- I2 は I1 に対して真の依存も偽の依存もない
  - ◇ したがって，I1 の完了を待たずに I2 を発行しても動作の正しさは保たれる
  - ◇ なんとかして，I2 を I1 の裏で並列に実行したい
- 方針：
  - ◇ 命令毎に依存の有無を確かめながら適宜ストール or 発行を行う

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

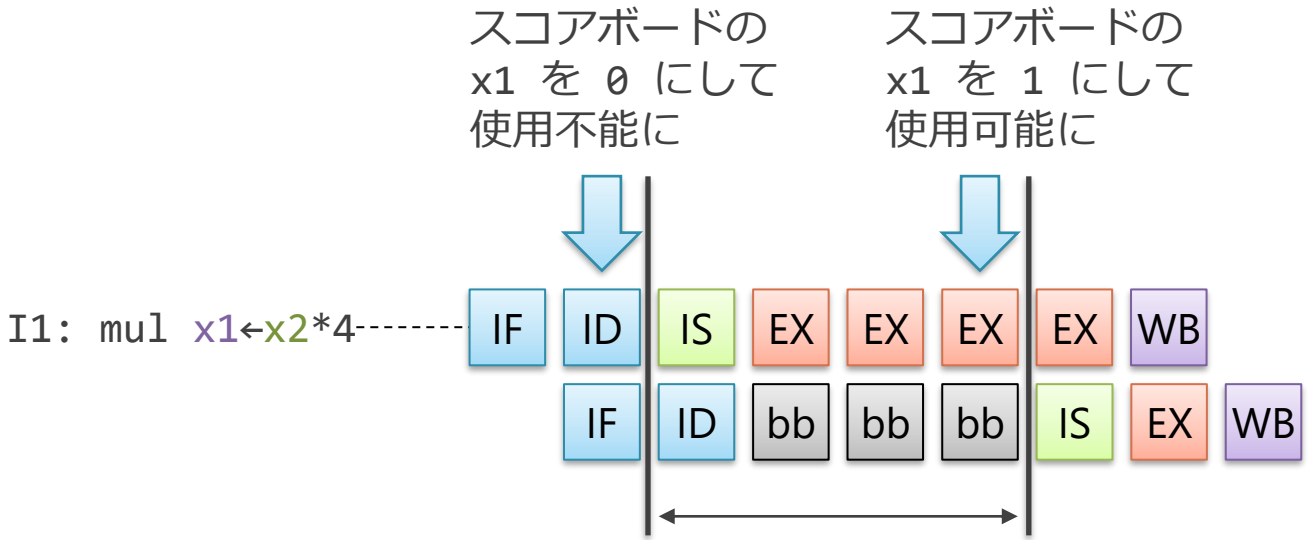
- レジスタ状態：
  - ◇ 各レジスタの値が現在利用可能かを 1 bit のフラグで保持
  - ◇ 1 なら現在使用可能，0 なら実行結果の書き込み待ち
    - □ （文献によっては 1/0 が反転していることもある）



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



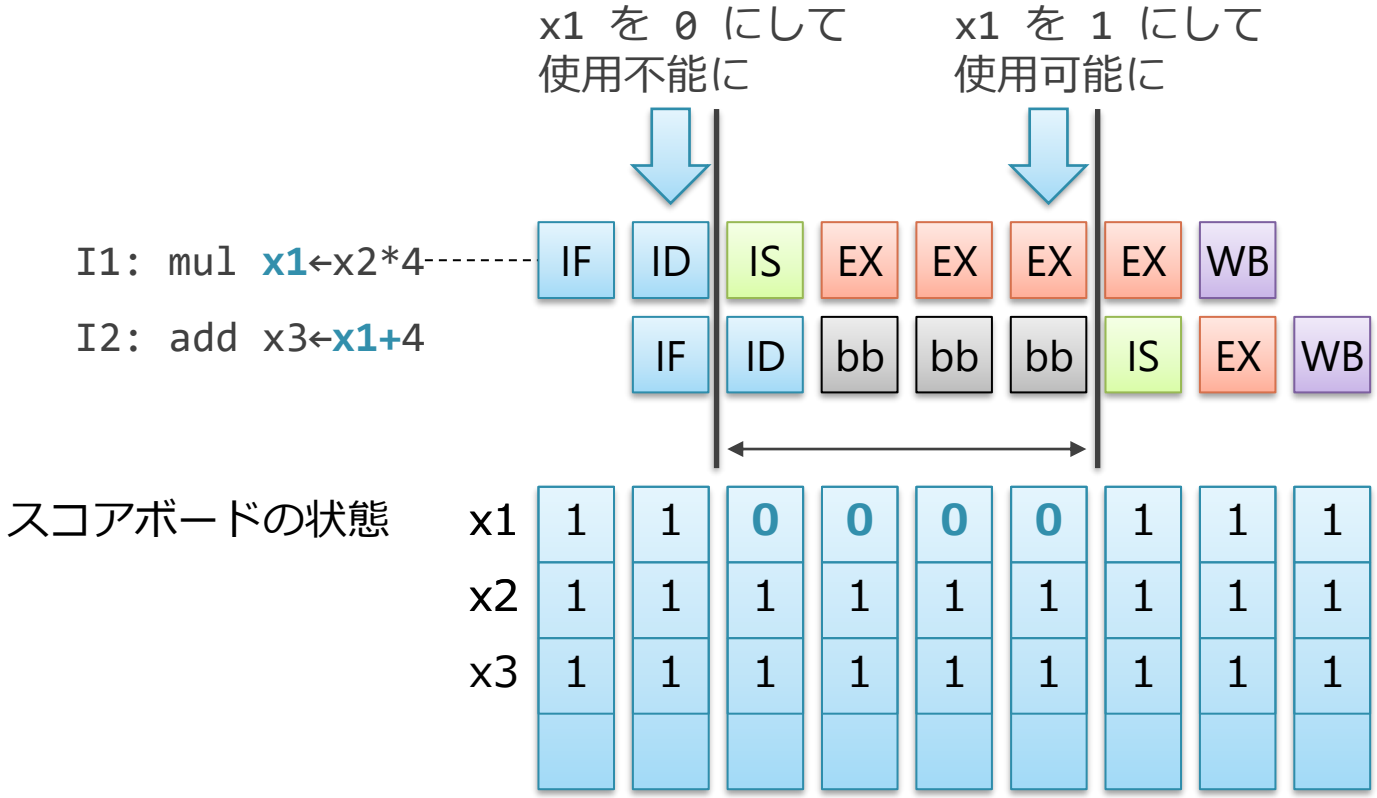
スコアボードによるストールの判定

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

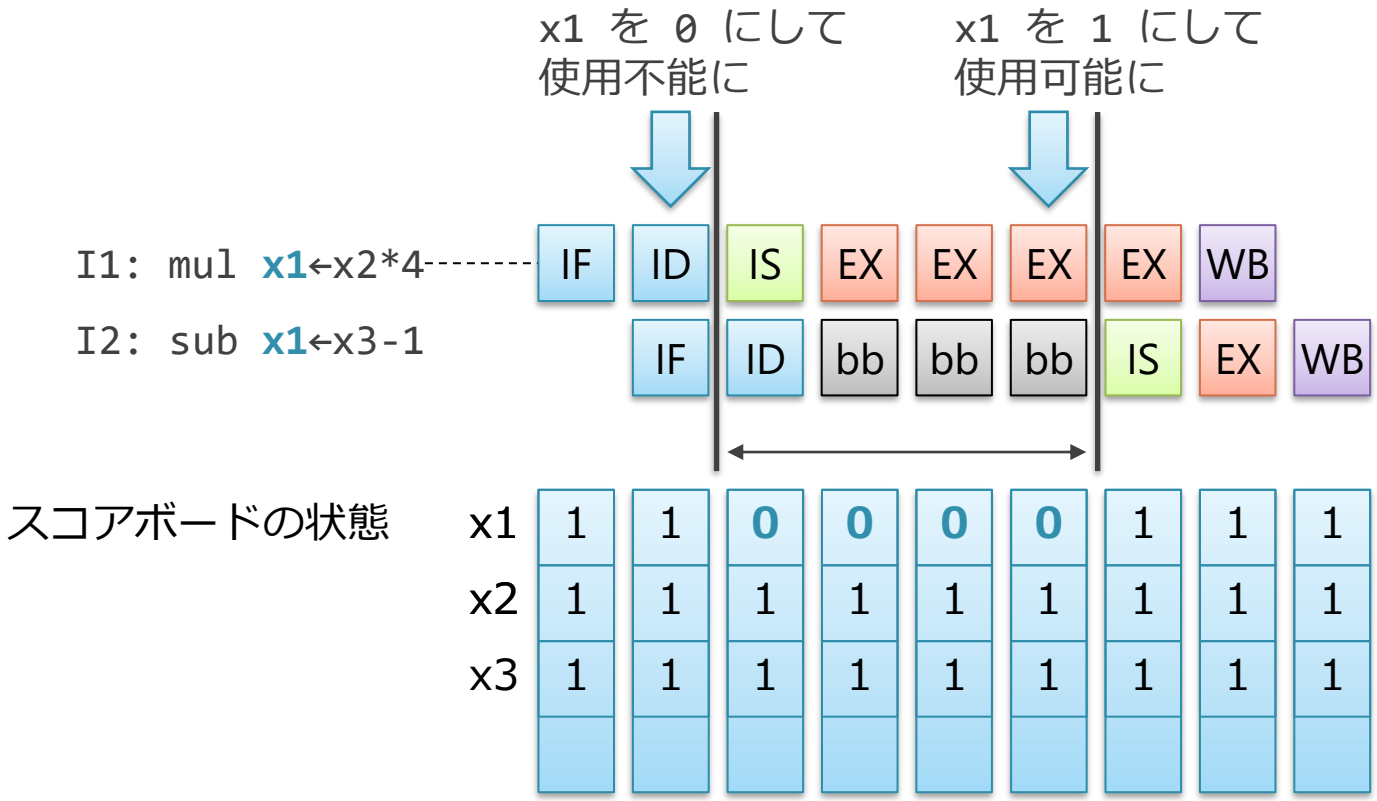
1. 真の依存がある場合

- 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



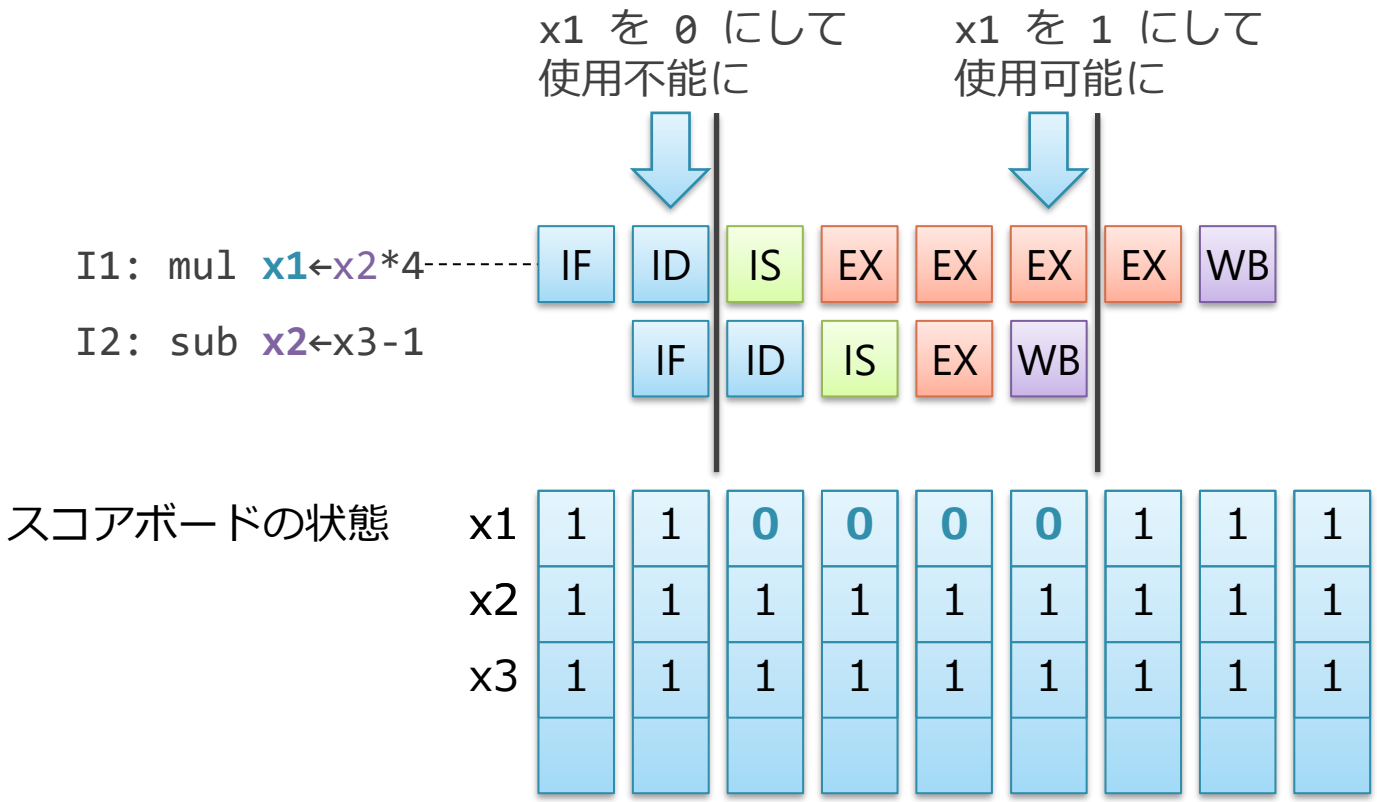
2. 出力依存（WAW）がある場合

- 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



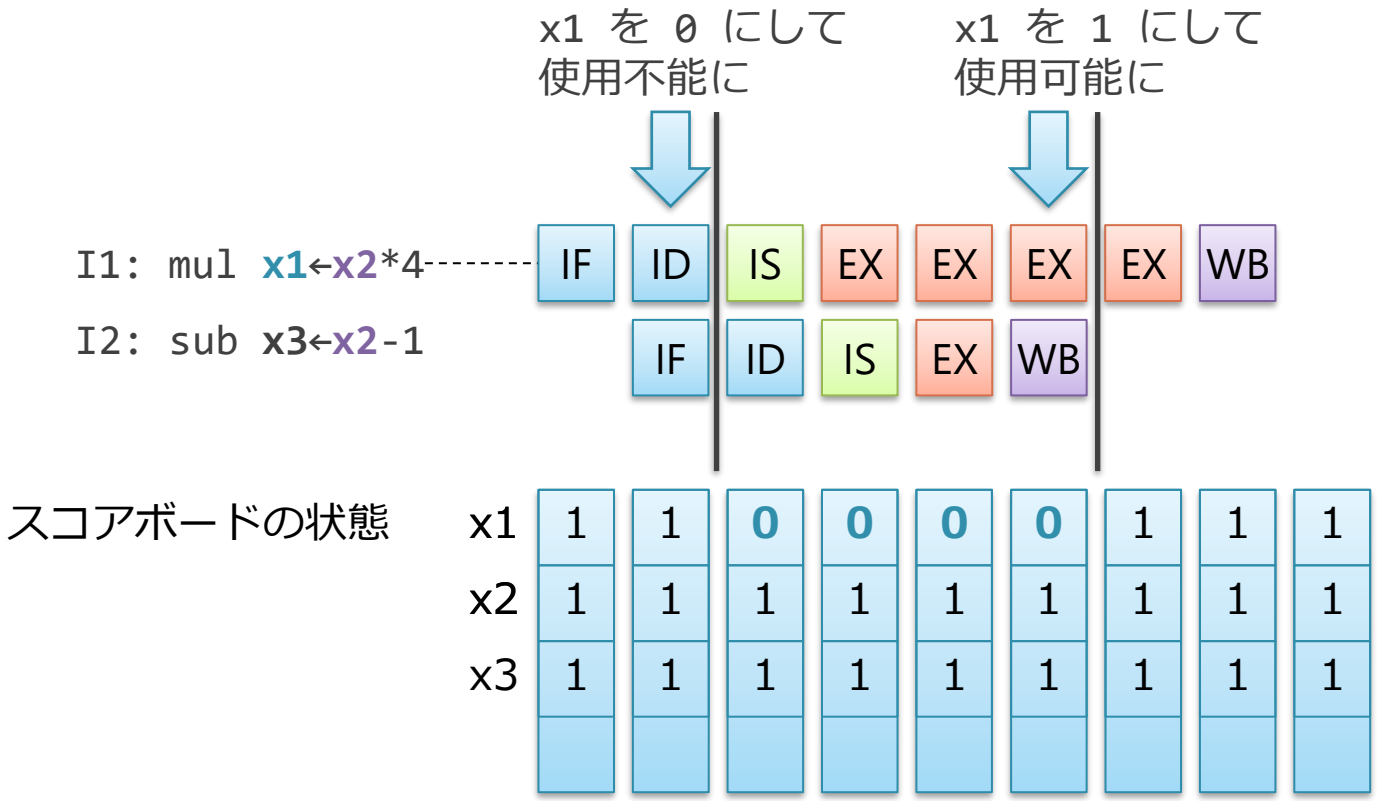
3. 逆依存（WAR）がある場合

- 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，最終的に辻褄はあう



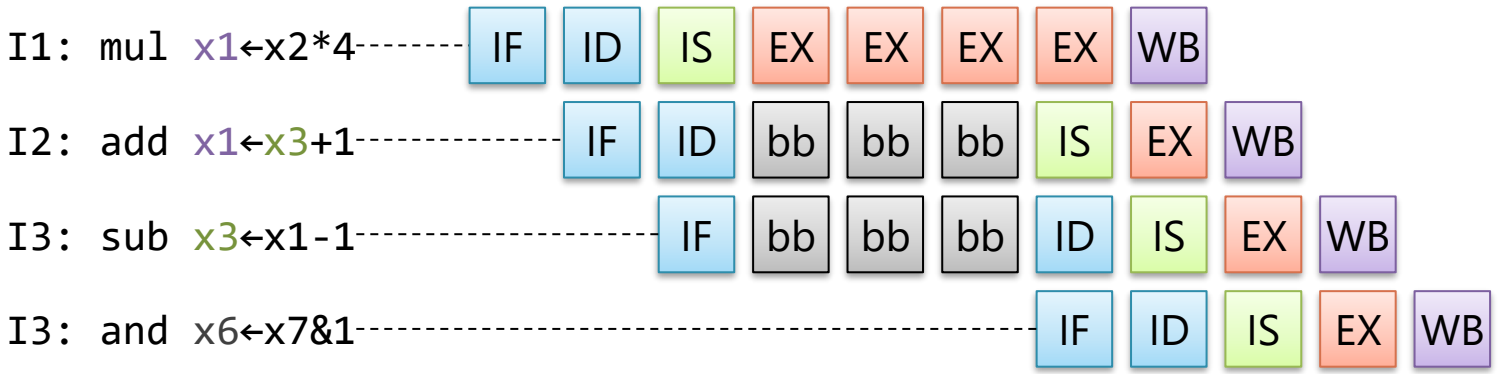
4. 依存がない場合

- 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



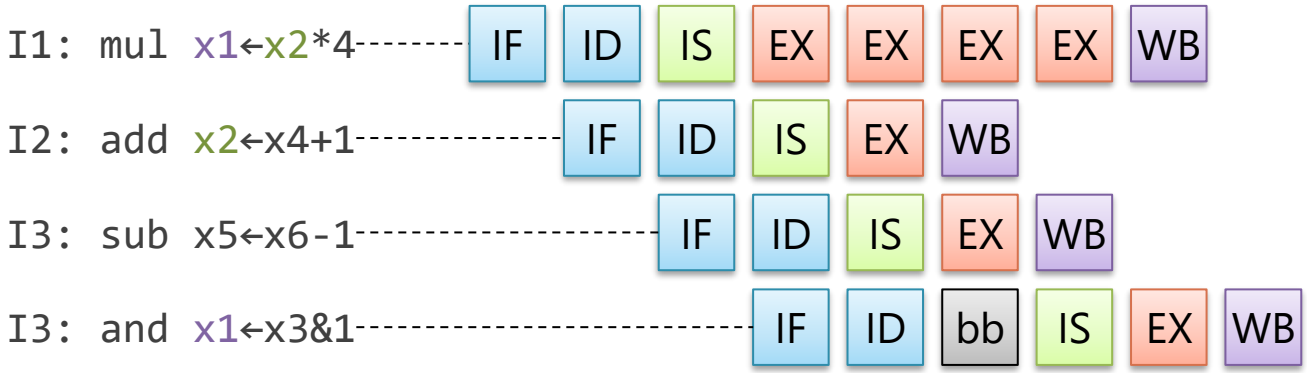
in-order 発行/out-of-order 完了

- 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」と言った場合はこれのこと
  - 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越して実行できる
    - 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

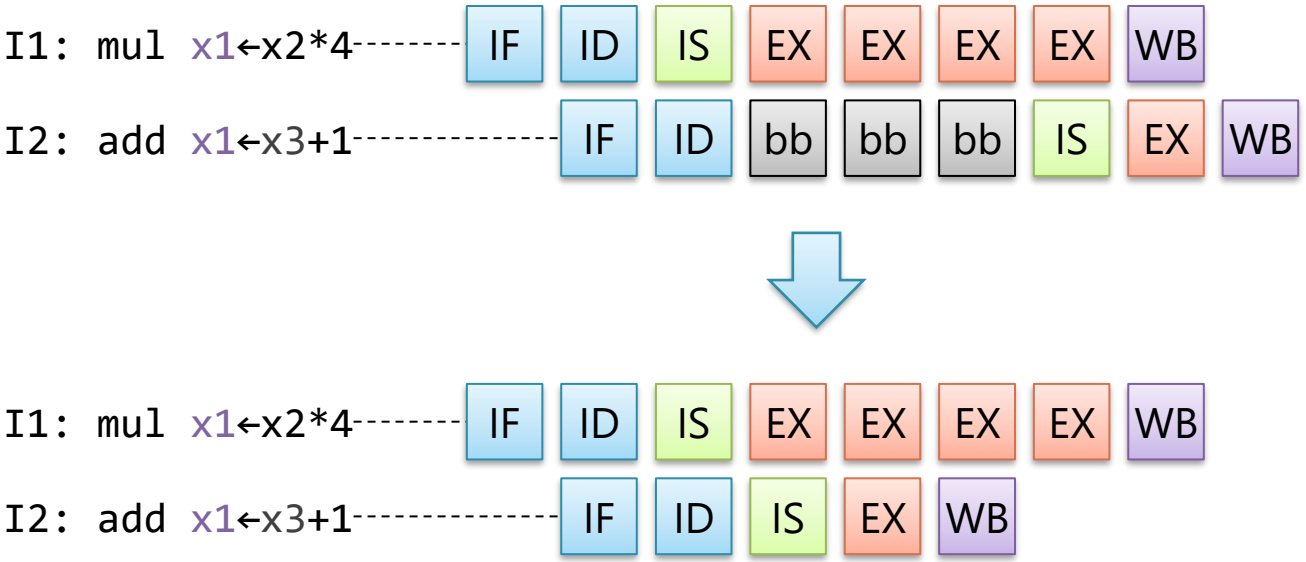
1. レジスタ・リネーム
2. out-of-order 発行機構

モチベーション

- 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

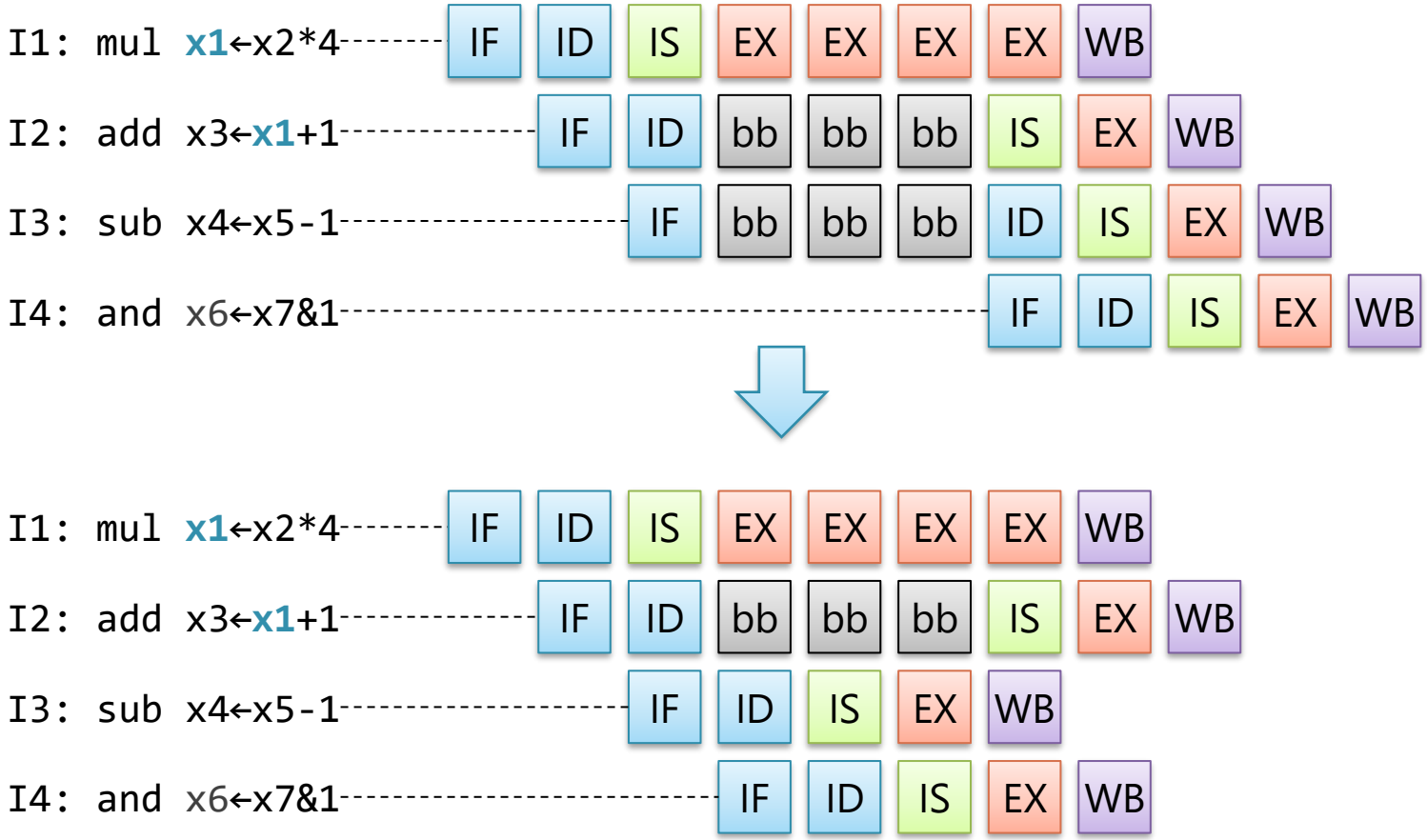
1. 出力依存（WAW）があると止まってしまう

- 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
  - 同じ場所に書き込むので，順序を揃えなければいけない
    - これをなんとかしたい



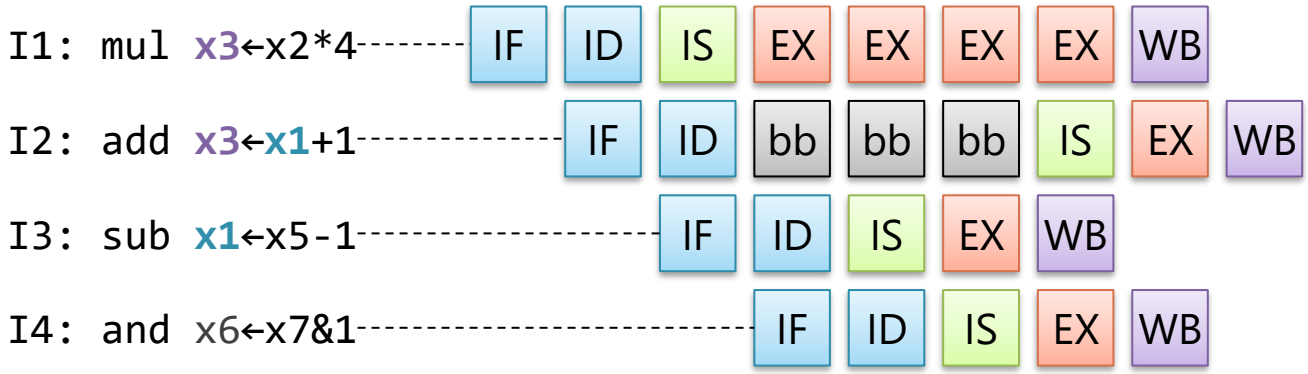
2. なんらかの依存がある命令があるとそこでパイプライン全体が止まってしまう

- 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）

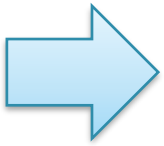
- ■ 現在主流なのは後者
  - ◇ 構造がより単純で設計しやすく性能も出やすいから
  - ◇ 文献では２者の関係があまり書かれていないが，
    これらはかなり構造が違う

- ■ 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

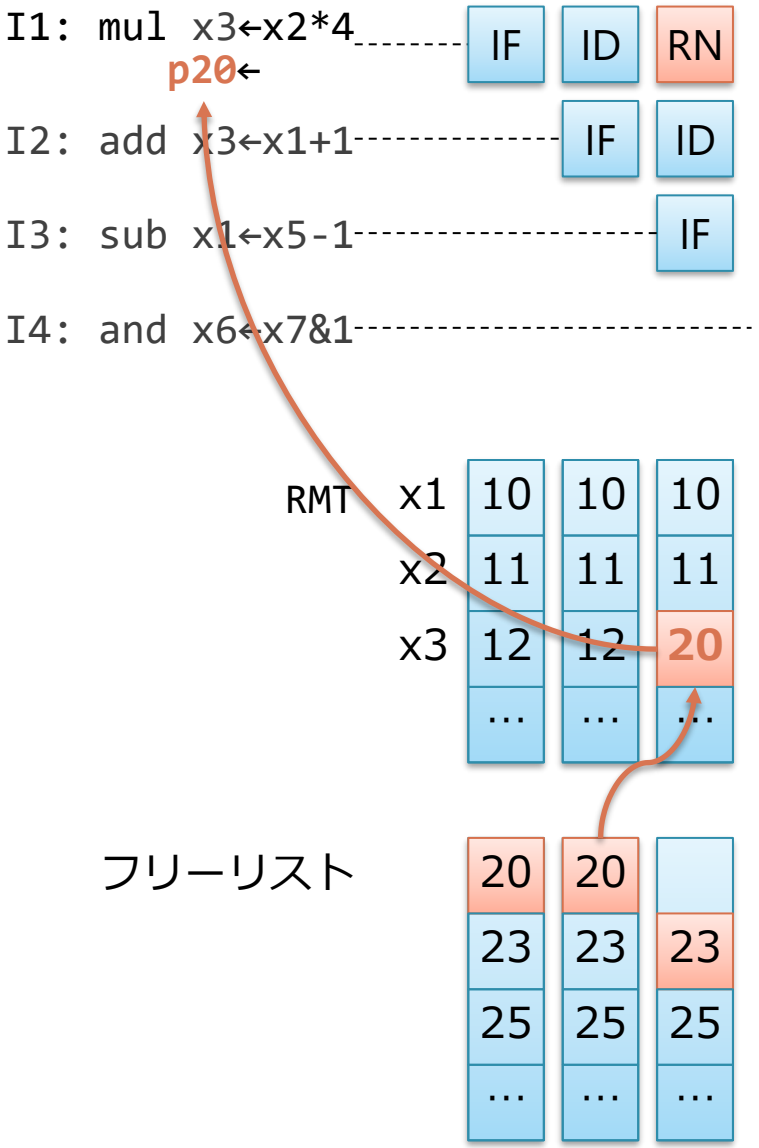
```
I1: mul x3←x2*4
I2: add x3←x1+1
I3: sub x1←x5-1
I4: and x6←x7&1
```

```
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

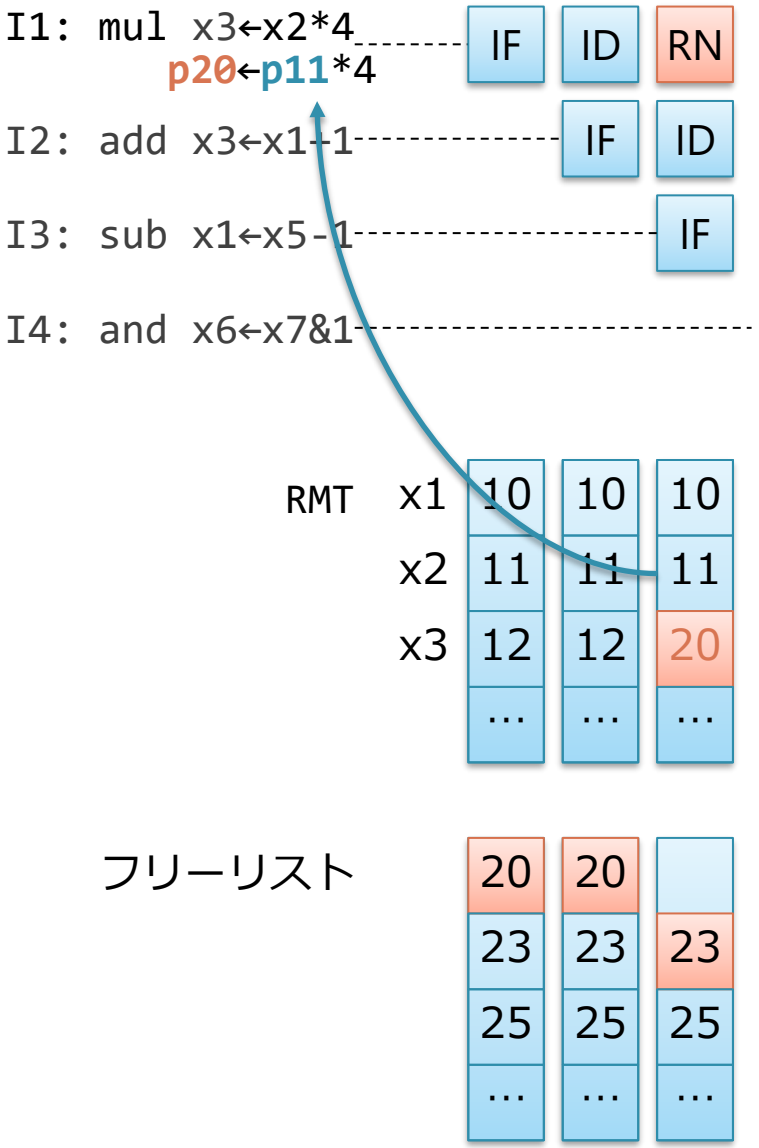
- RMT（Register Map Table）
  - 論理レジスタと物理レジスタの対応関係を保持する表
  - インテル用語だと RAT（Register Alias Table）
    - こっちの方が最近はよく見るかもしれない
- フリーリスト
  - 現在使用できる物理レジスタ番号のリスト
    - FIFO で構成する
  - ここから物理レジスタを確保して割り当てる
  - 使用し終わった物理レジスタ番号は，フリーリストに返却
- リネーム・ステージ
  - デコード・ステージの後に存在
  - 上記を使って，レジスタをリネームする

レジスタ・リネームの動作（１）



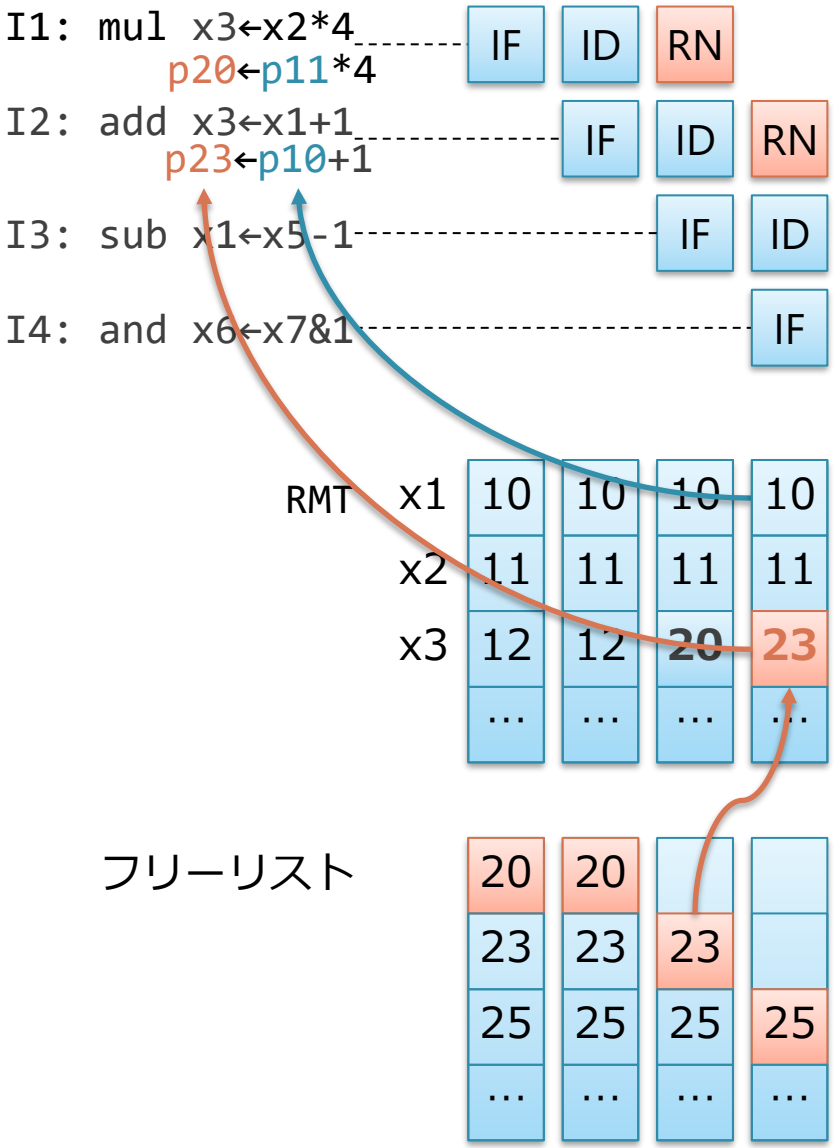
- I1 の リネーム（ディスティネーション）
  - フリーリストから p20 を取り出し，ディスティネーションに割り当て
  - RMT の x3 を書き換える

レジスタ・リネームの動作（２）



- I1 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）



- 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

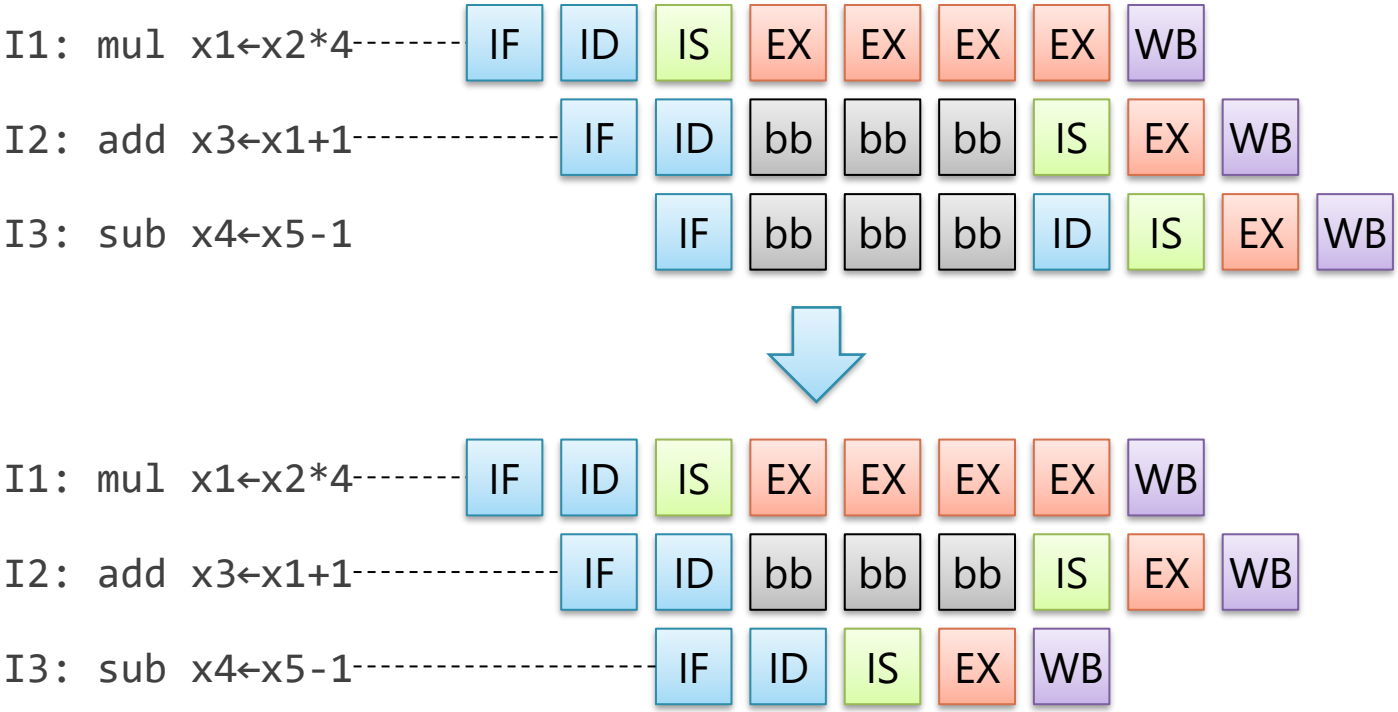
- 各命令のディスティネーションに専用のレジスタを与える
  - ◇ レジスタ番号がかぶらないので，
    他の命令との間で出力依存や逆依存は生じなくなる
  - ◇ RMT やフリーリストにより実現
- リネームを済ました後は，真の依存だけを考えて発行すればよい

out-of-order 発行/out-of-order 完了

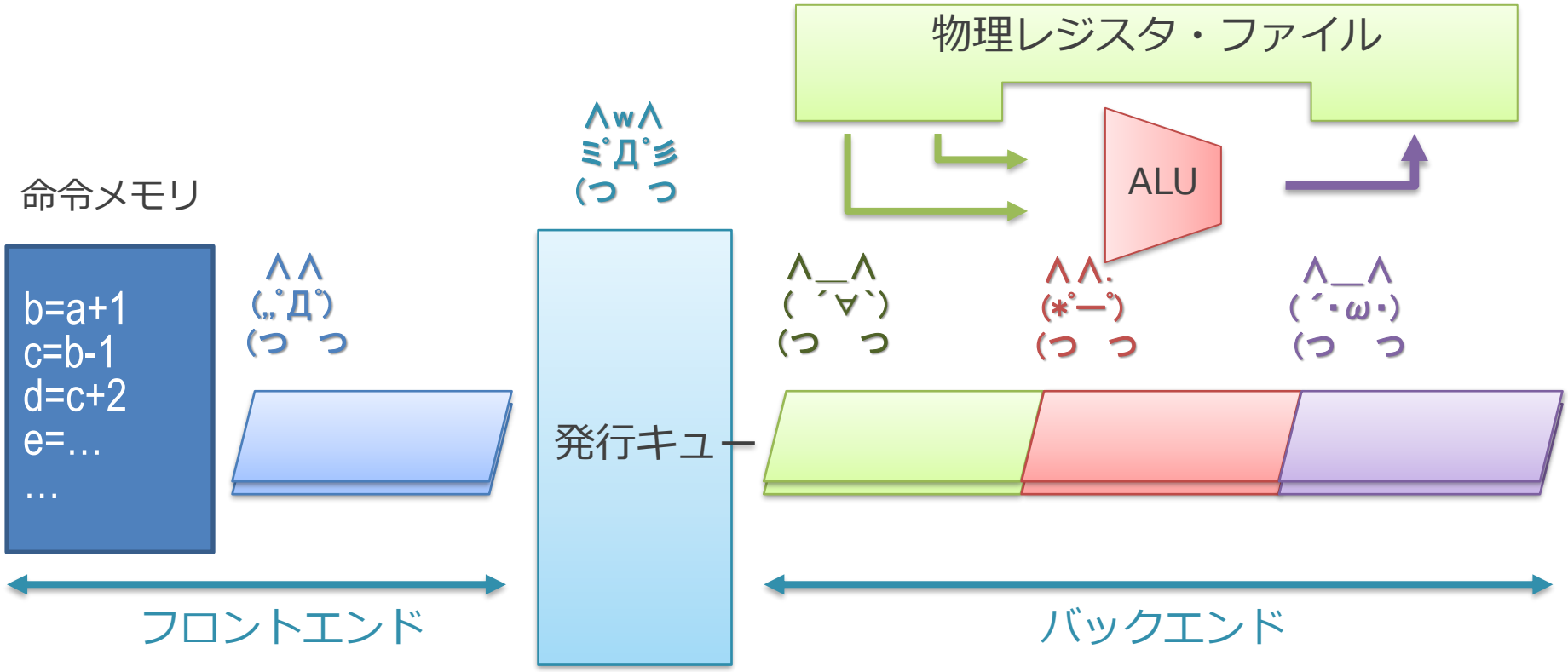
1. レジスタ・リネーム
2. **out-of-order 発行機構**
    1. スケジューラの作り方
    2. 性能への影響
    3. トマスロ方式との違い

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 = バッファがいる
  - ◇ 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

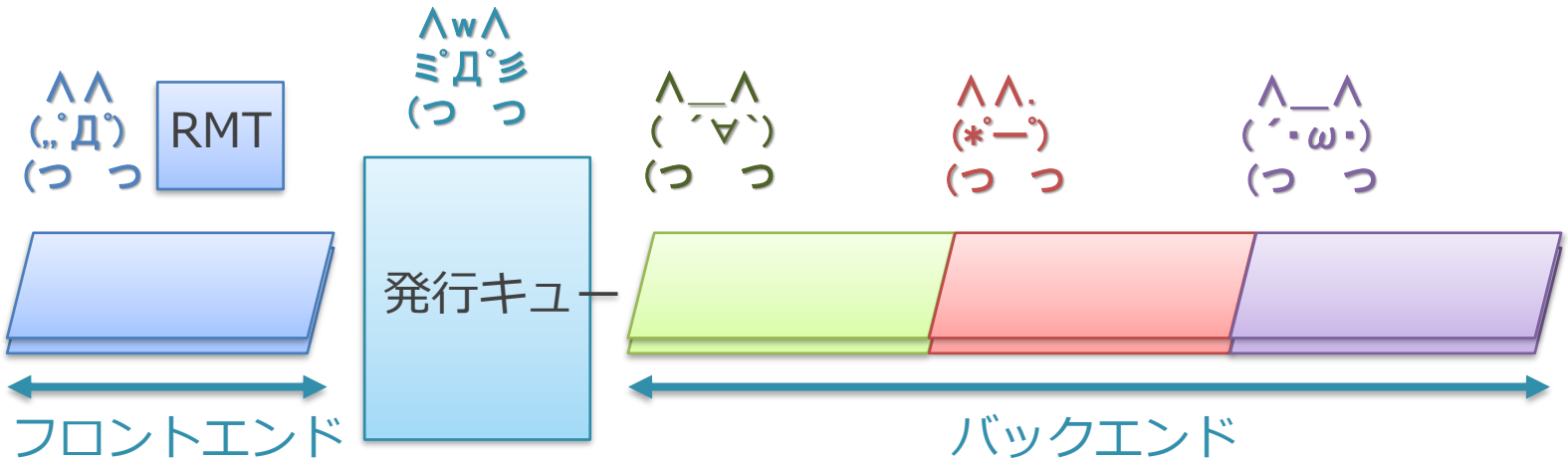


out-of-order 発行を行う CPU の構造



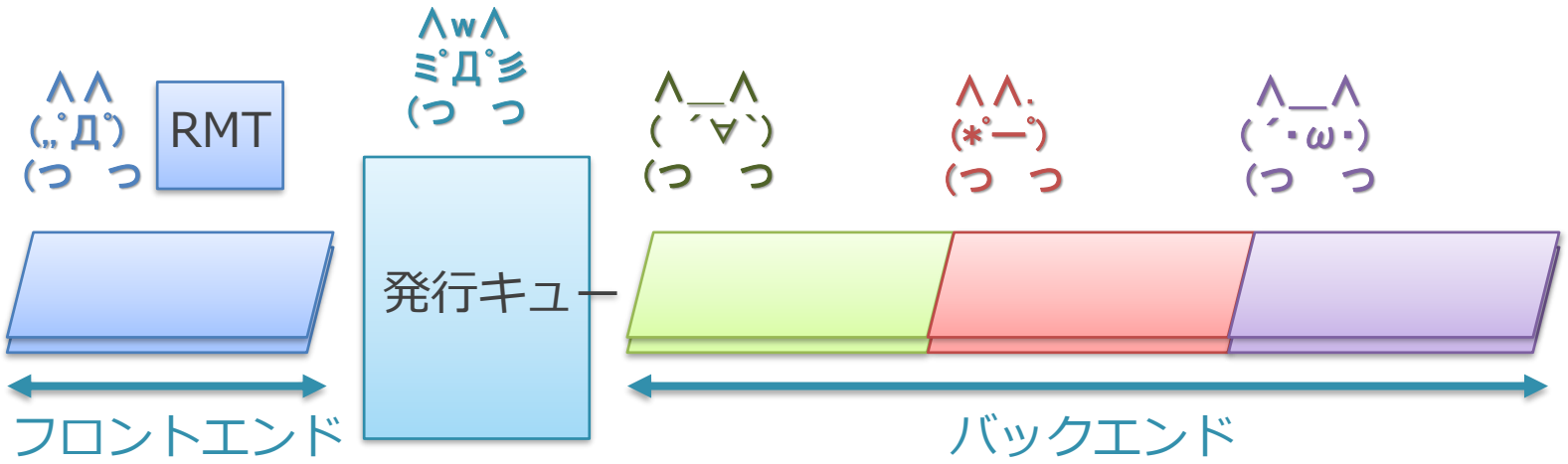
- 発行キューによって前後に分離された構造を持つ
  1. フロントエンド：　命令をフェッチ，リネーム
  2. 発行キュー：　発行待ち命令の待ち合わせのバッファ
  3. バックエンド：　命令を実行

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

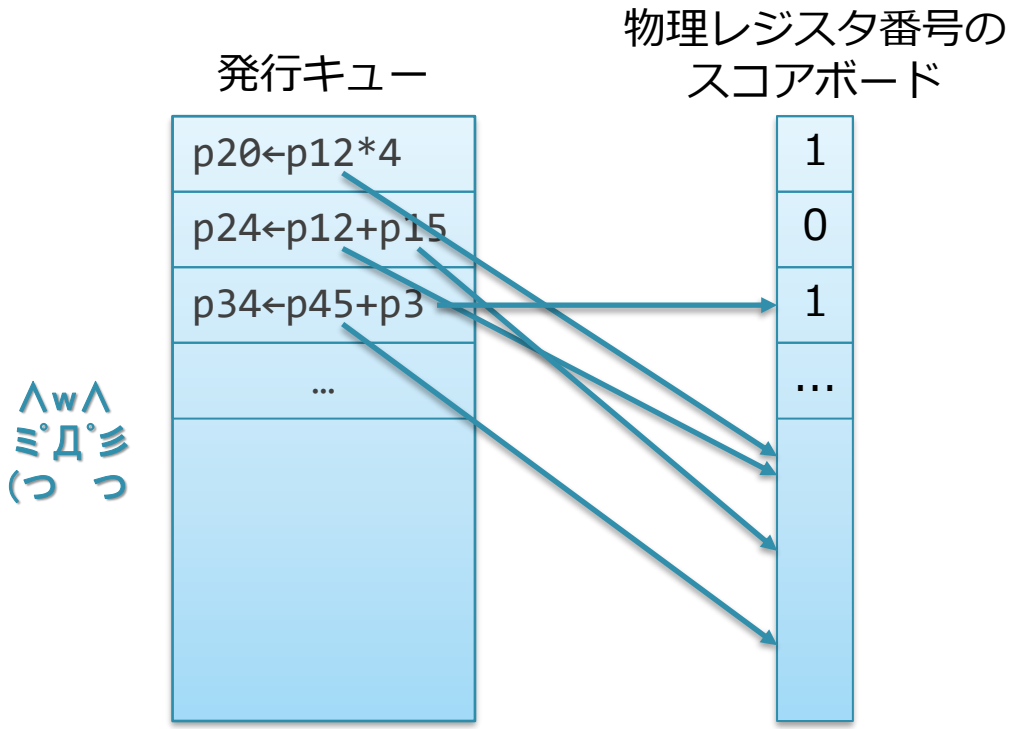


- 別にある発行キューのフリーリストから空きエントリ番号を確保
  - ◇ 物理レジスタのフリーリストの場合と同様
  - ◇ 発行したら返却

バックエンドへの命令の発行

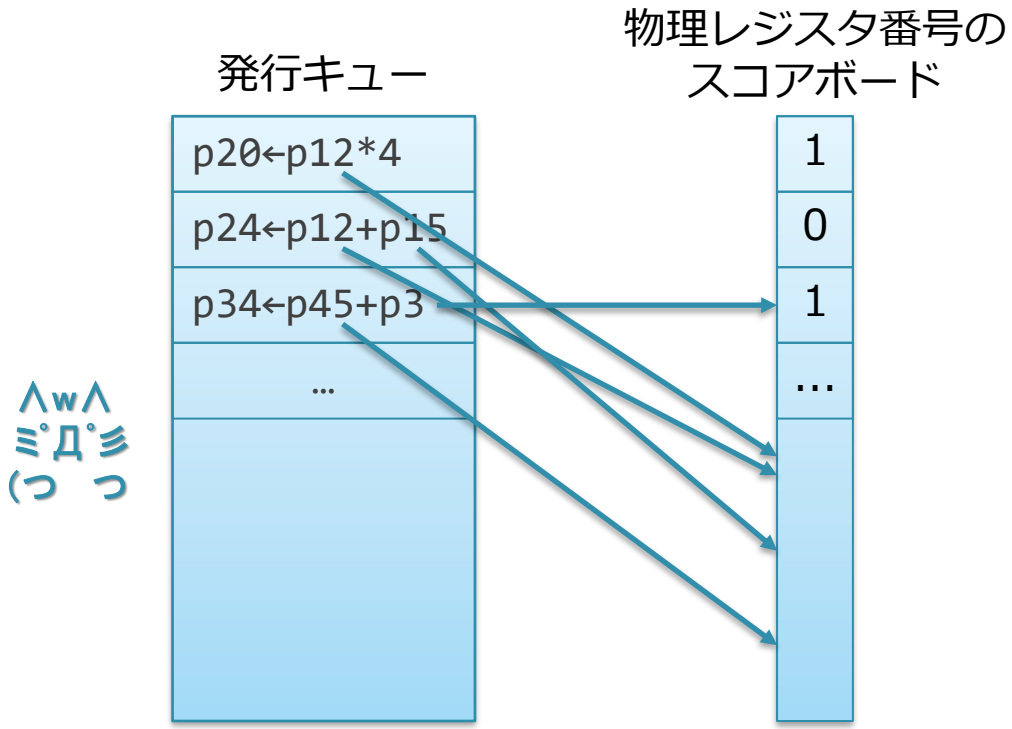
- スケジューラ（発行キュー）の作り方
  1. スコアボードを使った方法
  2. 連想検索を使う方法
  3. 行列を使う方法

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

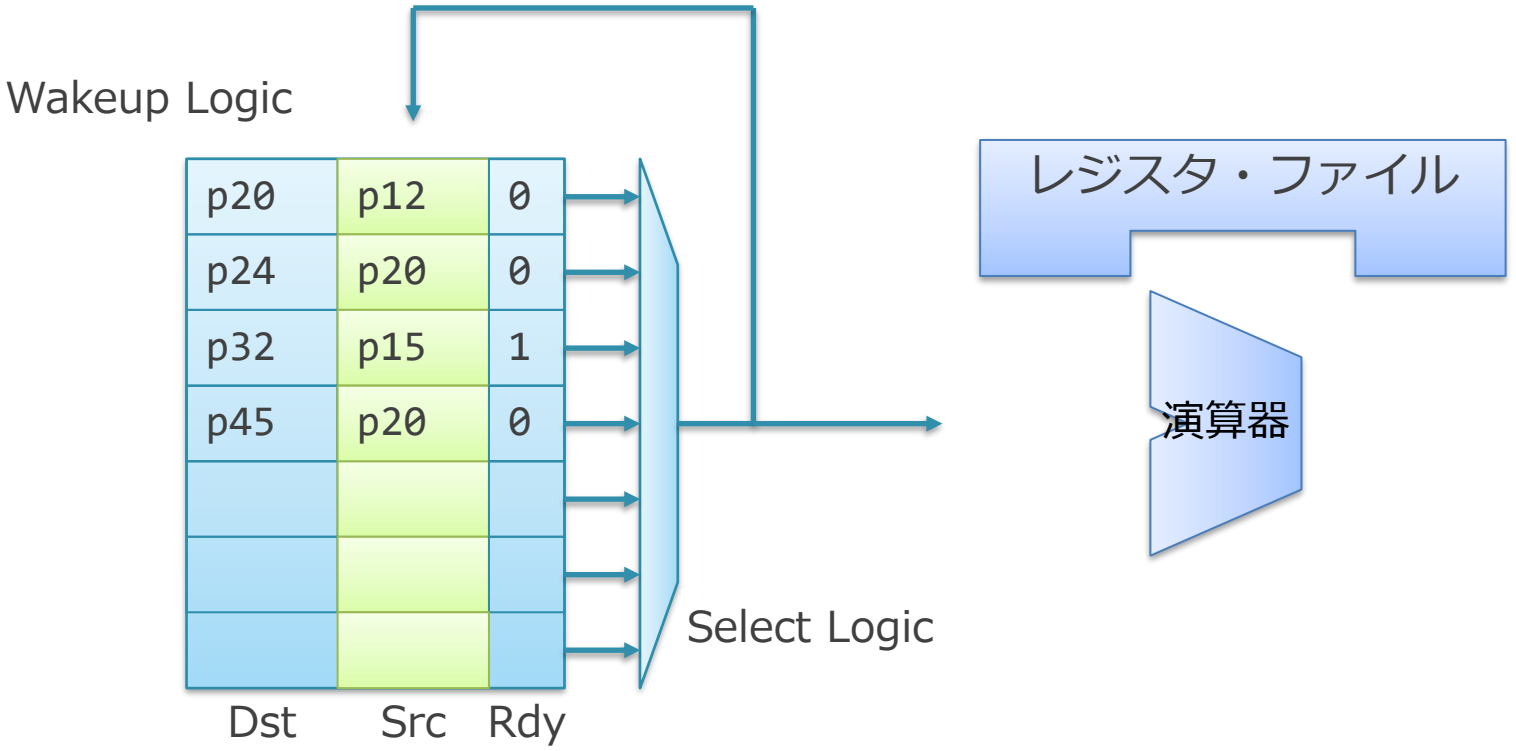


- スコアボードによる実装はスケールしない
  - ◇ 発行キューのエントリ数：数十から100命令
  - ◇ スコアボードはソース・オペランドの数だけ独立に読まれる
    = ものすごい数のポート数（100*3 とか）になる
  - ◇ メモリのサイズはポート数の二乗に比例するのでヤバイ

バックエンドへの命令の発行

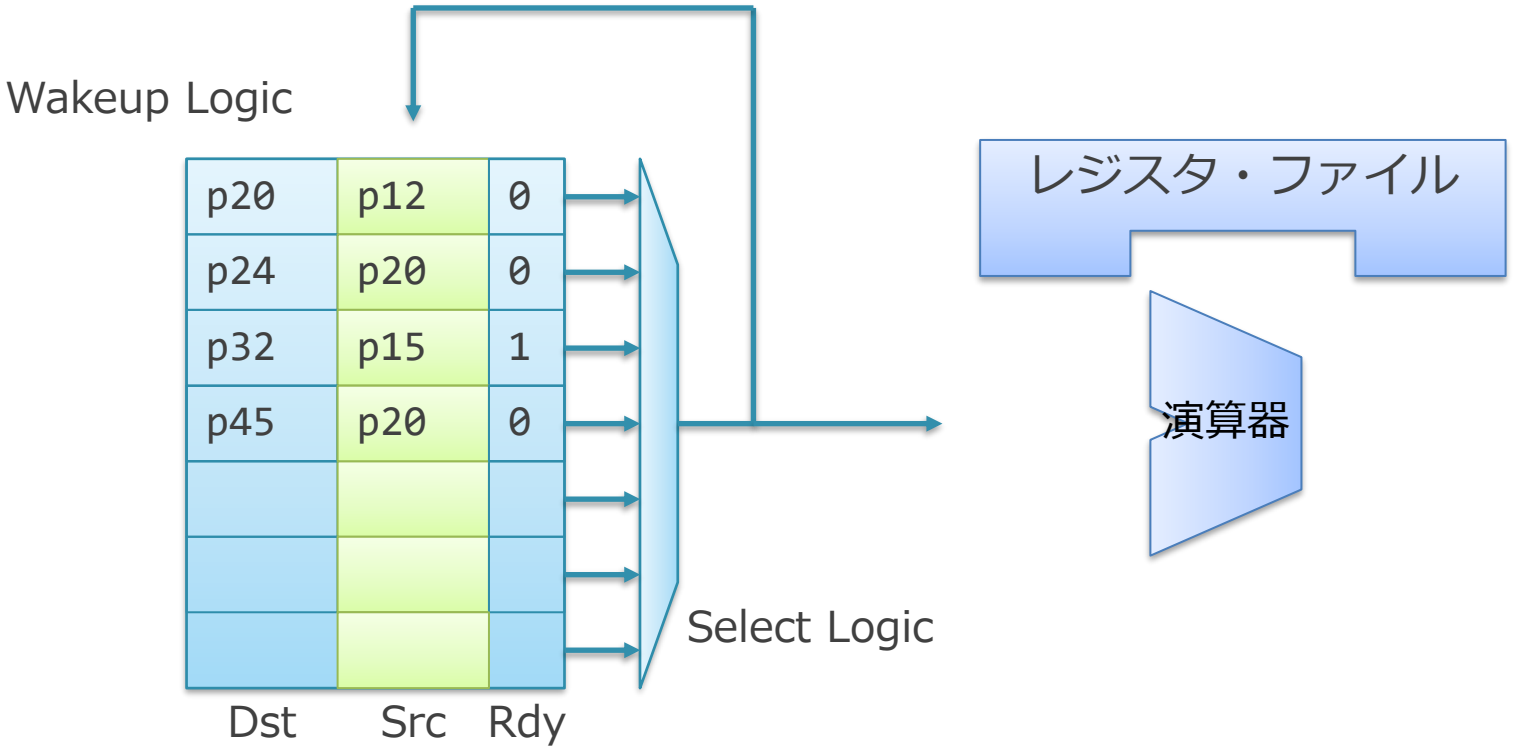
- スケジューラの作り方
  1. スコアボードを使った方法
  2. 連想検索を使う方法
  3. 行列を使う方法

連想検索による実装



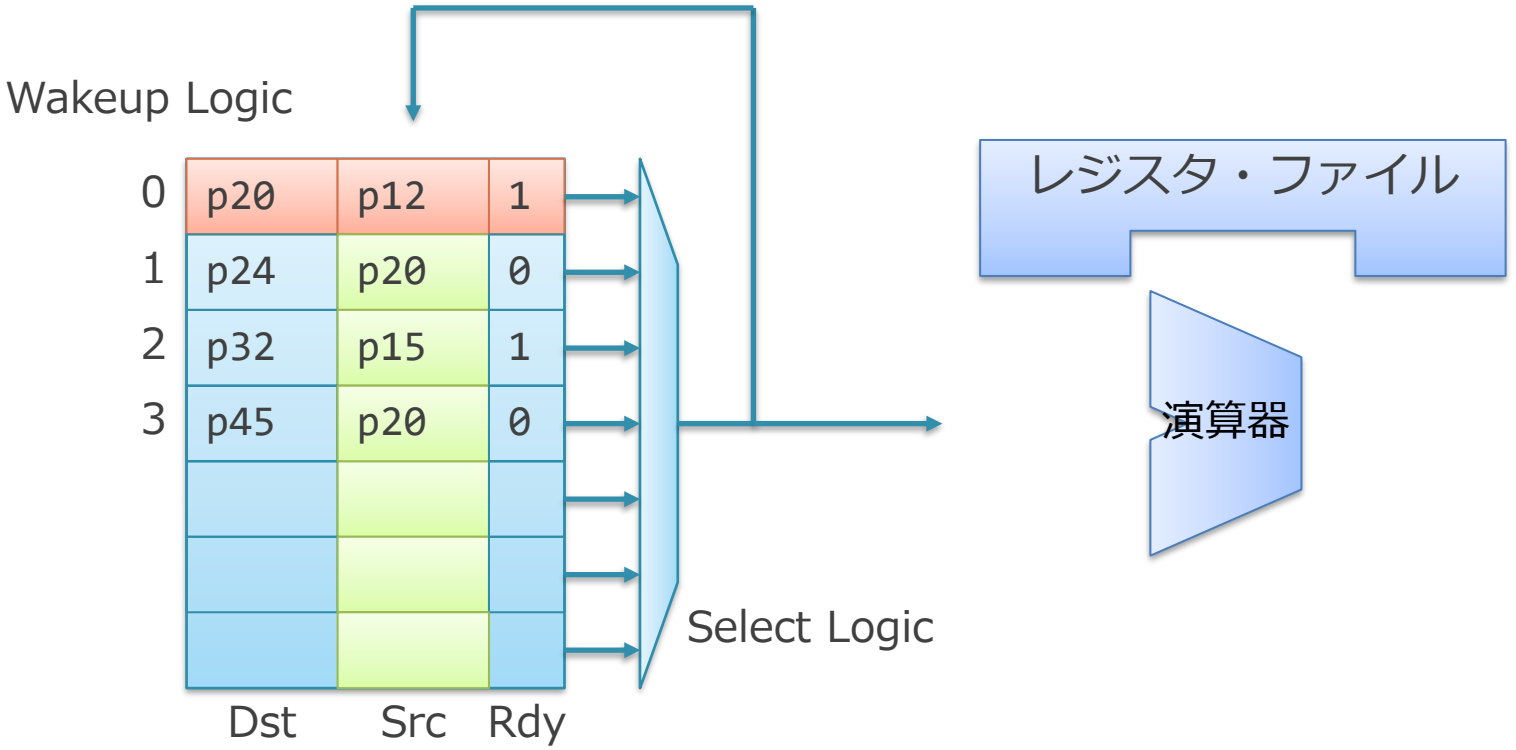
- Wakeup Logic：寝ている（発行待ち）命令を起こす機構
  - ◇ Dst：ディスティネーションの物理レジスタ番号
  - ◇ Src：ソースの物理レジスタ番号（簡単のためここでは１つ）
  - ◇ Rdy：依存元命令が実行され，発行準備ができているか

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

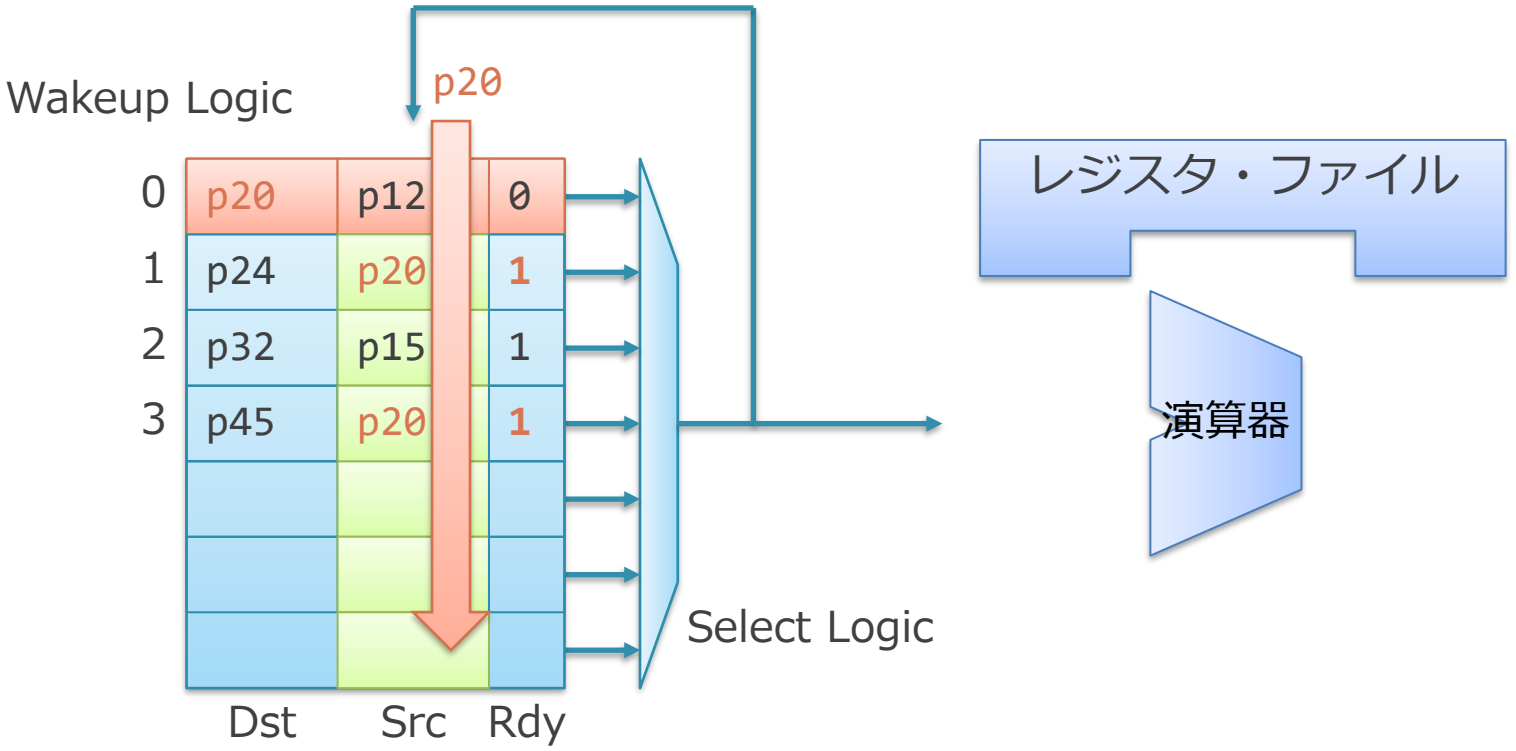
発行キューの動作（１）：Select



- 現在エントリ 0 番と 2 番で発行準備ができている
  - ◇ 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

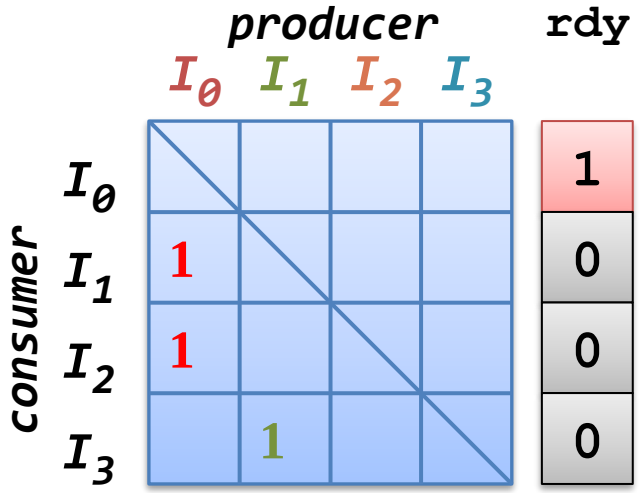
- モチベーション：連想検索を使った方式の問題
  - 遅延が大きい
    - 命令のセレクト後に物理レジスタ番号を読み出す必要がある
  - 消費電力が大きい
    - ものすごい数の比較器が同時に稼働している

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

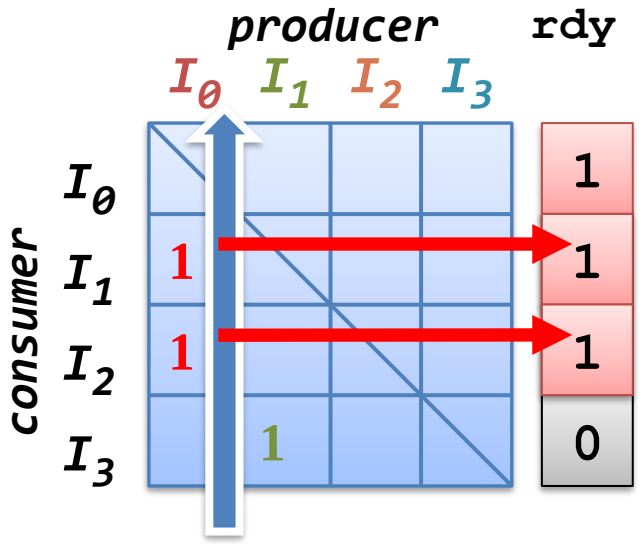
```
I0  ld   p11 = *(p10)
I1  sll  p12 = p11 << 1
I2  sll  p13 = p11 << 2
I3  add  p14 = p12 + 1
```



- 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

```
I0  ld   p11 = *(p10)
I1  sll  p12 = p11 << 1
I2  sll  p13 = p11 << 2
I3  add  p14 = p12 + 1
```



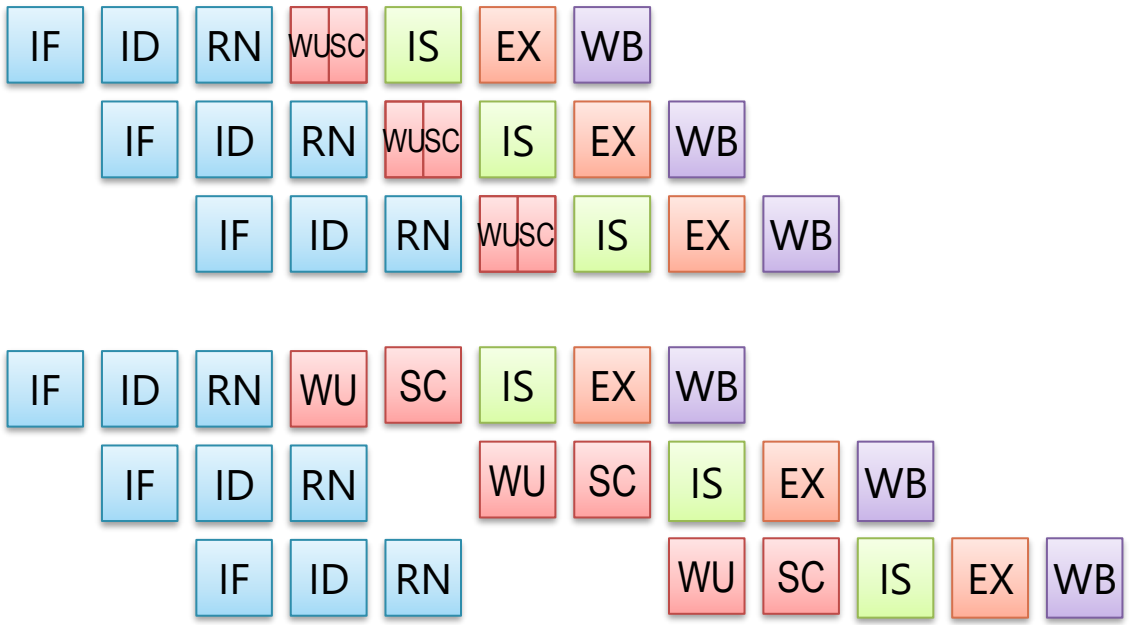
- Wakeup の動作
  - ◇ 発行された命令の列をアサートする
  - ◇ 各行のその列のビットが立っていたら rdy を 1 に
- RAM の読み出しに近い構造で実現できる

out-of-order 発行/out-of-order 完了

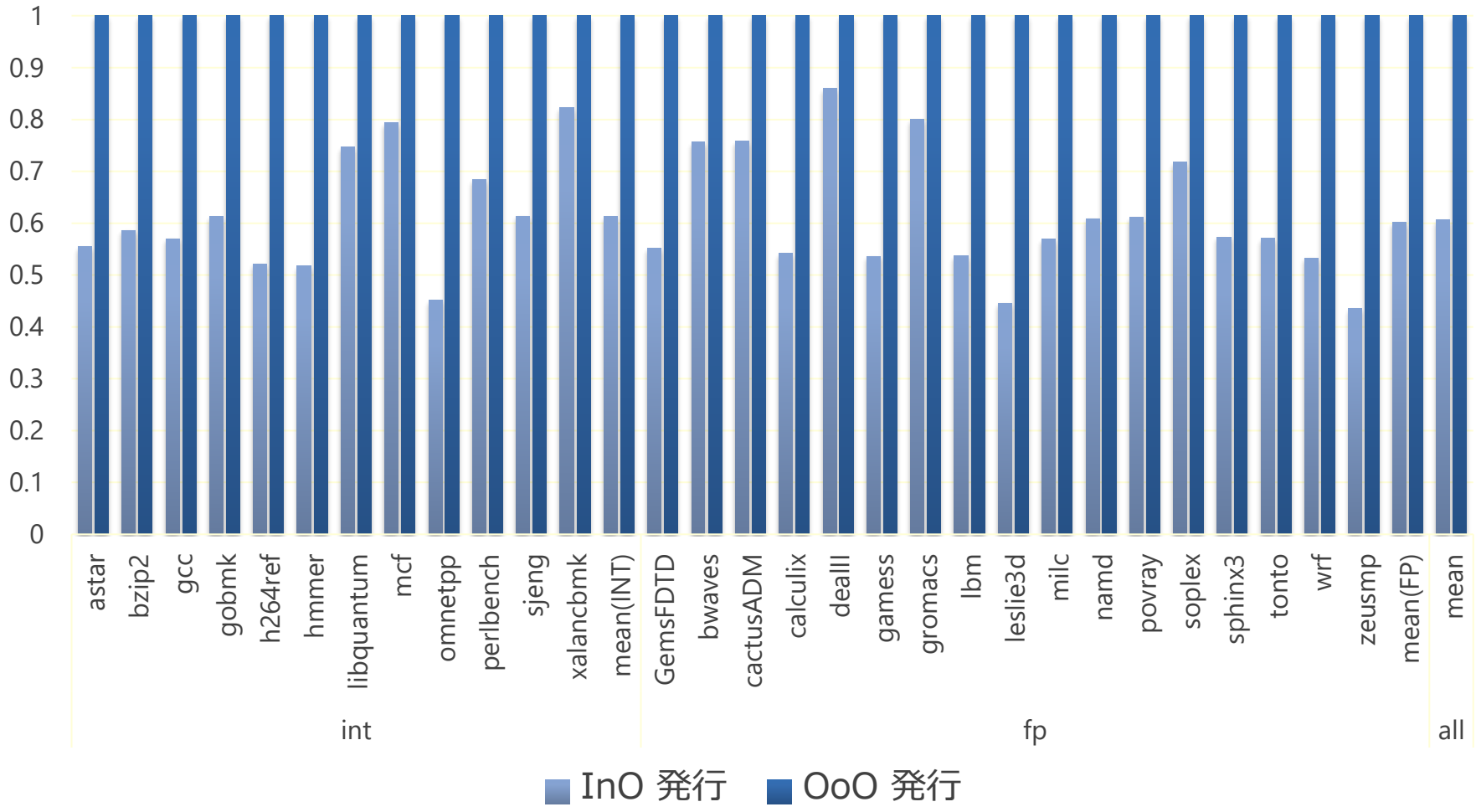
1. レジスタ・リネーム
2. out-of-order 発行機構
    1. スケジューラの作り方
    2. **性能への影響**
    3. トマスロ方式との違い

発行キューの特性

- Wakeup-Select は 1 サイクルで終える必要がある
  - ◇ 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）SPECCPU 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

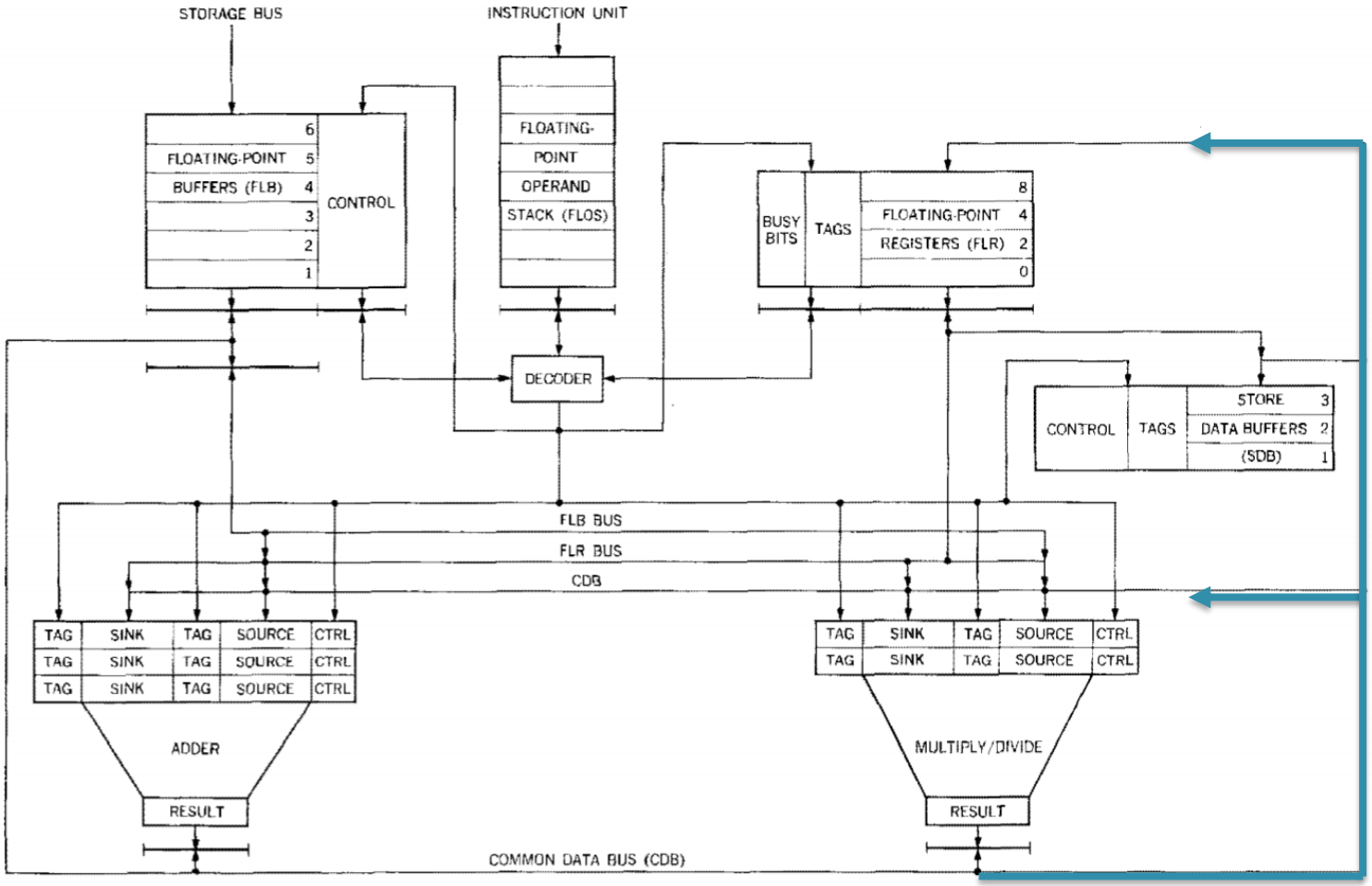


Figure 4 Data registers and transfer paths, including CDB and reservation stations.

- 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：wakeup
- 意見や内容へのリクエストもあったら書いてください