はじめに

- 前回は、マシン命令を処理する仕組み（機構）について講義しました。
- 今回と次回は、マシン命令処理の高速化について考えます。

- 教科書の対応範囲は、以下の通りです。
  - 5.2.2項（a）～（d）（p.135～139）

命令実行サイクル（おさらい）

- 1個のマシン命令は、以下のステージを順に経て処理を行います。
  1. 命令フェッチ（IF: Instruction Fetch）
  2. 命令解読（D: instruction Decode）
  3. オペランドフェッチ（OF: Operand Fetch）
  4. 演算実行（EX: EXecution）
  5. 結果格納（WB: Write Back）
  - 前回の講義で用いた例では、必ずしも上記のステージが実行フェーズと1対1対応していませんでしたが、1命令の処理に複数サイクルを要していました。

- もっと速く（短い時間で）処理できないでしょうか？

パイプライン処理（Pipelining）

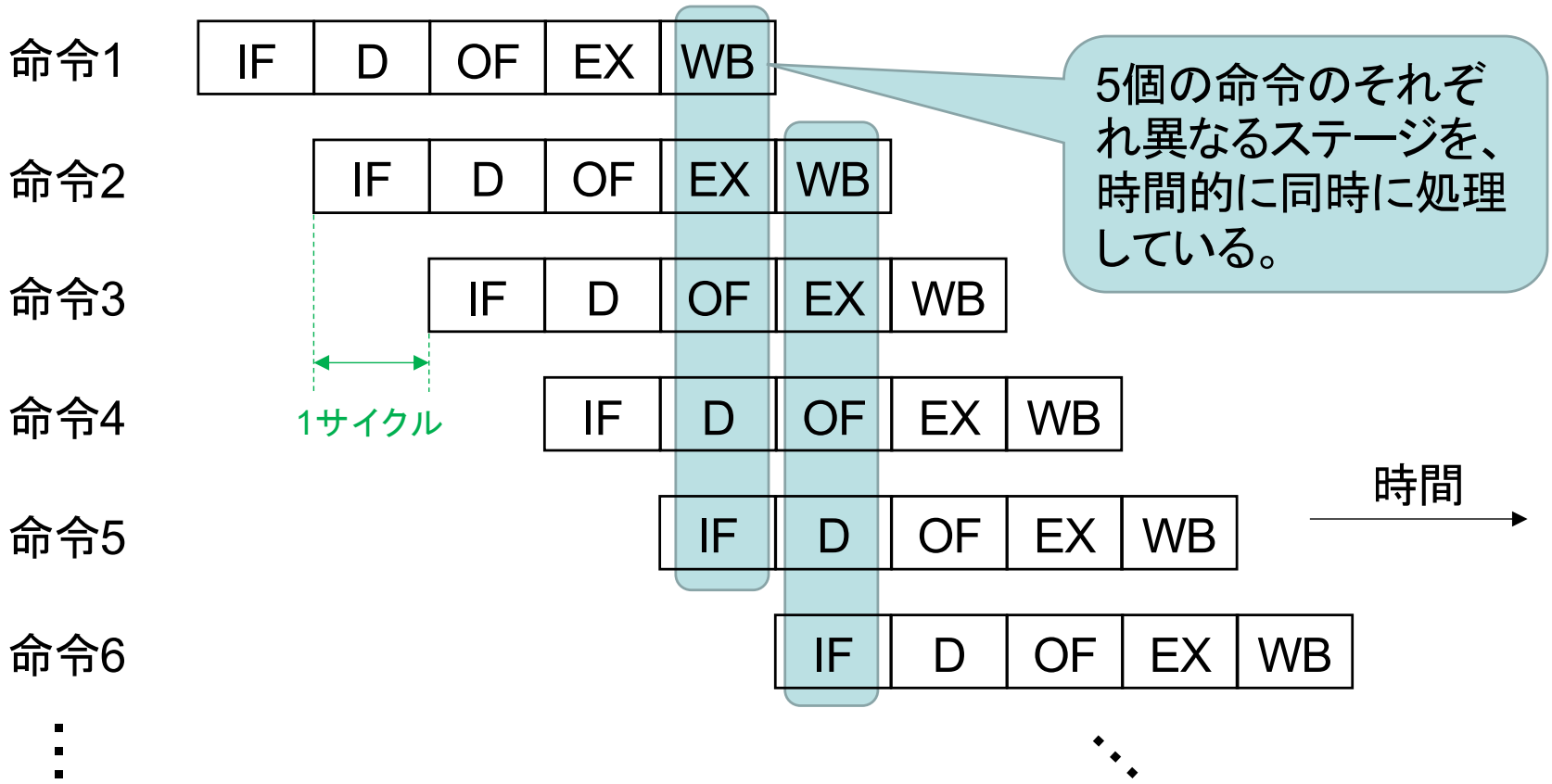
- プロセッサの高速化の1つの手法が、パイプライン処理（pipelining）です。
  - 処理を細かい過程（stage）に分割して、流れ作業で処理します。
- このパイプライン処理を命令の処理に適用したのが、命令パイプライン（処理）方式です。
  - ちなみに、演算処理に適応したものを演算パイプライン（処理）方式といいます。昔のスーパーコンピュータは、演算パイプライン方式を採用したベクトルコンピュータでしたが、本科目では扱いません。

命令パイプライン方式

- 基本原理
  - 命令実行サイクルをパイプライン(pipeline)という複数ステージ(stage; 過程)に分割します。
    - 各ステージでの処理時間がほぼ同じになるように分割します（そうしなければうまく高速化できません）。

    注） 命令実行サイクルでもステージという言葉を用いていましたが、命令パイプラインのステージと必ずしも一致しません。
  - ステージごとに独立に処理（稼働）できるハードウェア機構を装備します。
  - それらのステージおよびそのステージでの処理を担当するハードウェア機構を時間的・空間的にオーバラップして(overlap; 重なり合わせて)稼働させます。

- 流れ作業のイメージ
  - 例として、命令実行サイクルのステージと命令パイプラインのステージを一致させた場合で考えます。
    - PC更新は命令フェッチと同時に行うものとします。



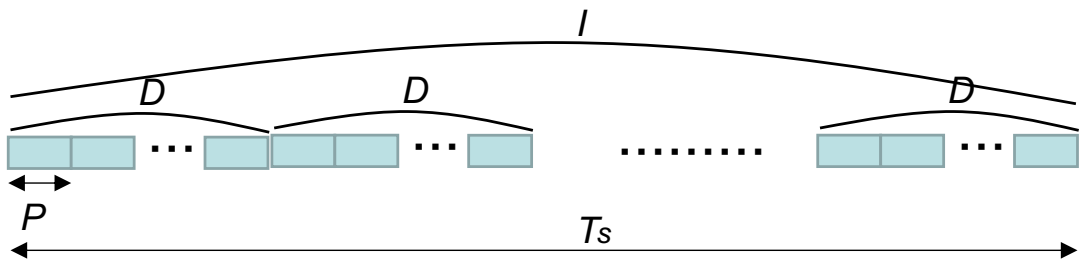
- 特徴・効果
  - 複数のステージから構成されるパイプラインを見かけ上並列に実行させます。
    - 時間と空間の両方で多重化しています。
      - 命令処理を時間的に分割することにより、1本のパイプラインで複数の命令を処理しています。
      - 各ステージ（IF、D、など）の処理を独立に、かつ、同時に行っています。
    - 命令の処理に要する時間を隠蔽（hide）しています。
      - 前ページの例では、1命令実行するのに5サイクルかかっており、この点では、命令パイプラインで処理しない場合に比べてまったく高速化できていません。しかし、同時に他の命令の処理を行うことで、その命令の処理にかかる時間を隠しています。
      - これにより、命令処理のスループット（throughput）を増加させています。

一般的な定義は、単位時間当たりの仕事量。ここでは、単位時間当たりに処理する命令数。

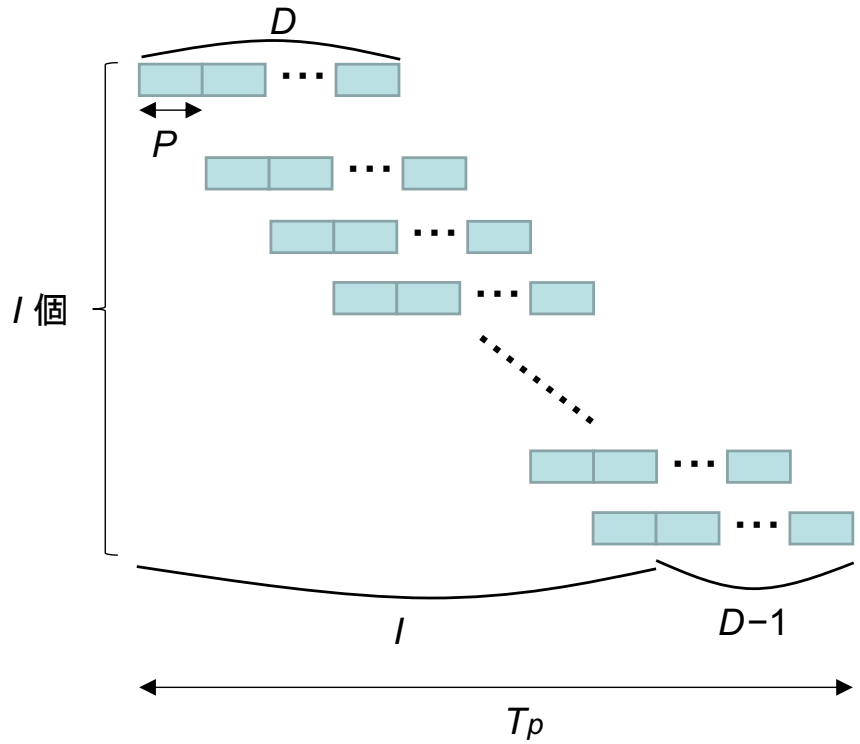
パイプライン処理の理論的性能

- 以下の定義を用いて、パイプライン処理によって性能が何倍になるかを見積もります。
  - パイプラインの深さ（ステージ数）： D
  - パイプラインピッチ（1サイクルの時間）： P [秒]
  - 処理する命令数： I

- 逐次実行の場合の全処理時間T_sを求めます。
  - $T_s = I \times (D \times P)$ [秒]



- パイプライン処理の場合の全処理時間T_pを求めます。
  - $T_p = (I + (D - 1)) \times P$ [秒]



- よって、性能向上比（高速化率）Rは
  - $R = \frac{1/T_p}{1/T_s} = \frac{T_s}{T_p} = \frac{I \times D}{I + D - 1} = \frac{D}{1 + \frac{D-1}{I}}$
  - ここで、I が十分大きければ $\frac{D-1}{I} \approx 0$ より $R \approx D$

- つまり、性能はほぼ D倍に高速化できます。
  - ちなみに、現在の主要な商用マイクロプロセッサの命令パイプライン段数は10～25です。
  - さて、D倍の性能が出るのは、パイプラインがスムーズに流れる場合のみです。実際にはD倍にならない状況も生じ得ますが、それについてはもう少し後で説明します。

- また、1命令当たりの処理に要するサイクル数 *CPI*（Cycles Per Instruction）は
  - $CPI = \frac{T_p/P}{I} = \frac{I+D-1}{I} = 1 + \frac{D-1}{I}$
  - ここでも、I が十分大きければ $\frac{D-1}{I} \approx 0$ より $CPI \approx 1$

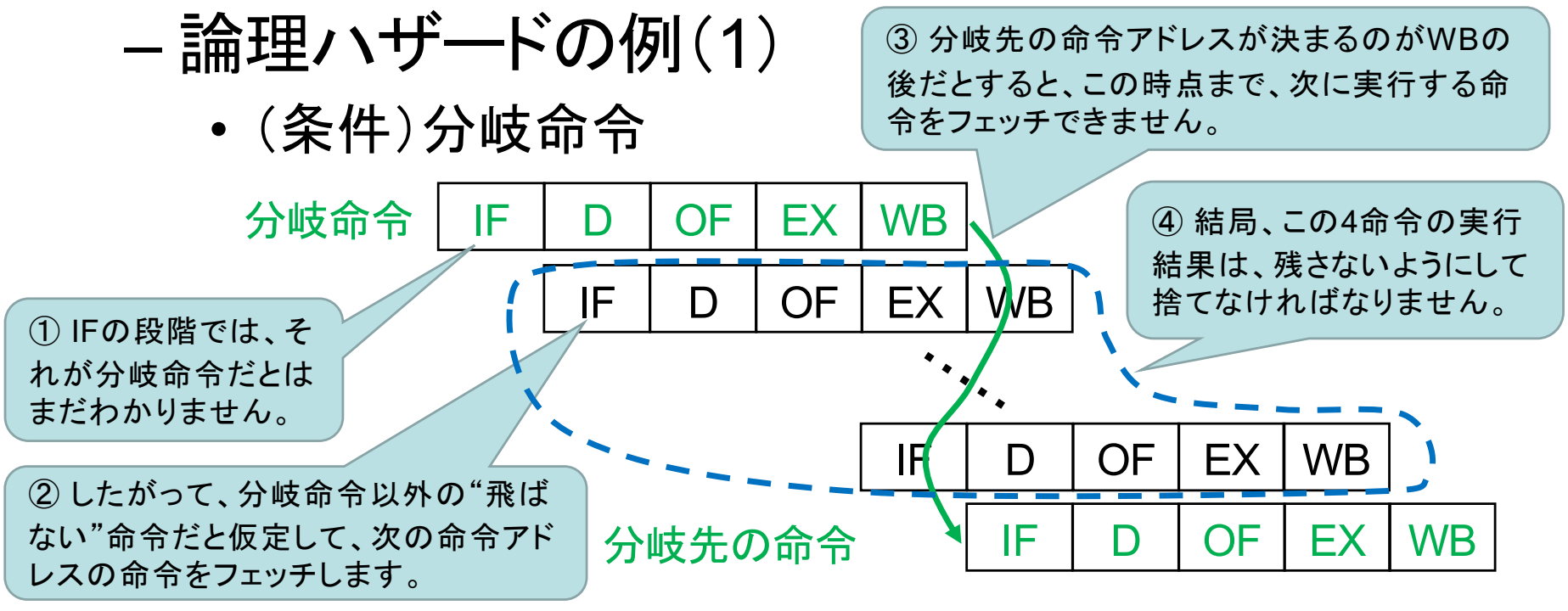
- つまり、1命令の処理を開始してから完了するまでにはDサイクルかかりますが、見かけ上は、1命令を1サイクルで処理していることになります。
  - これは計算するまでもなく、T_p を求めたときの図から明らかですが。。。

パイプラインの乱れ

- パイプライン中で命令を次のステージへ進めることができない状況をパイプラインハザード（pipeline hazard）と言います。

  – パイプラインハザードには以下の2種類があります。
    - 論理ハザード
      – 命令間の依存関係によって生じるハザード
    - 構造ハザード
      – パイプラインステージの構成に起因して生じるハザード
      – 本質的には、複数のステージで、同じ資源を利用しようとする（競合）ことによって生じる

- パイプラインハザードの例
  - 以下に、どのような状況でパイプラインハザードが生じるかを見ていきます。
    - 以下の例では、命令パイプラインはIF、D、OF、EX、WB、の5ステージで構成するものとします。
  - 論理ハザードの例(1)
    - (条件)分岐命令



- 結局、分岐命令と分岐先の命令の間の4サイクルはパイプライン処理せずに逐次処理したのと同じ効果しかありません。
- このような分岐命令によって生じる命令間の依存関係を制御依存と言います。

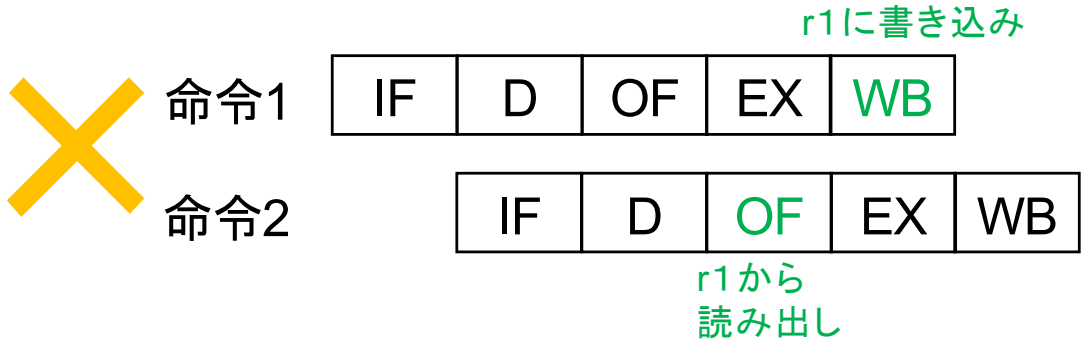
– 論理ハザードの例(2)

- データの生成・使用の順序制約
  - 以下の2命令を順に実行する場合を考えます。

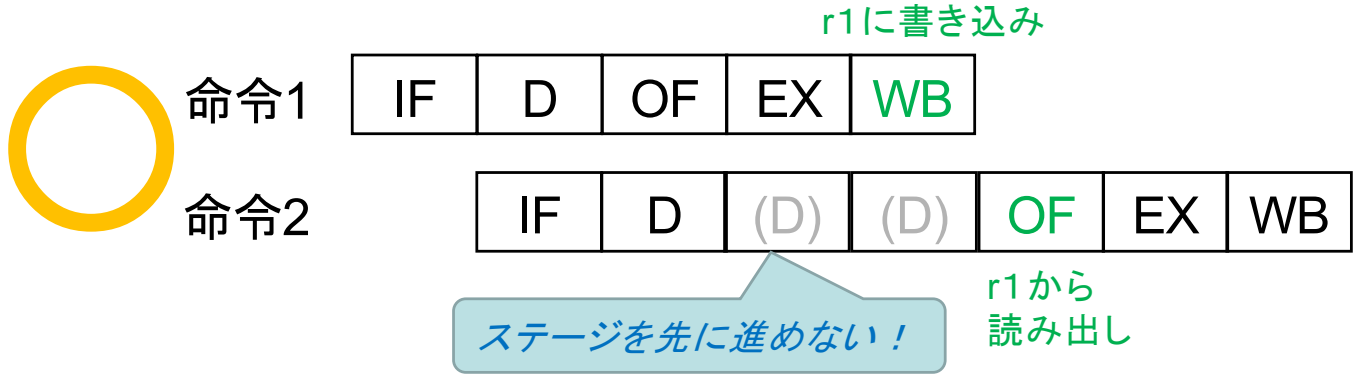
    命令1:　　add　r2, r3, r1　　(r1 ← r2 + r3)

    命令2:　　add　r1, r4, r5　　(r5 ← r1 + r4)
  - 命令1で生成した値(レジスタr1に格納)を命令2で使用しています。つまり、r1の使用に関して命令2が命令1に依存しています。
  - よって、命令1が加算結果をr1に書き込むまで、命令2がr1を読み出してはいけません。

– つまり、以下のような処理を行ってはいけない。



– 以下のように、正しいr1の値が読めるように、命令2をDステージに留めます。



– パイプラインをこのように制御することを**インタロック制御**（interlock control）と言います。

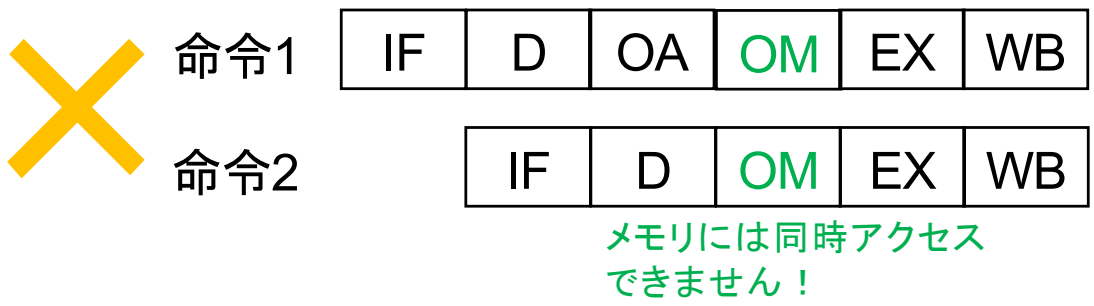
– 結局、パイプラインの2サイクル分を無駄にしてしましました。

– 構造ハザードの例

- メモリアクセスの競合
  - 以下の2命令を順に実行する場合を考えます。

    命令1:　add　r1, 8(r9), r1　　(r1 ← r1 + Mem[r9+8])

    命令2:　add　r3, (r4), r5　　(r5 ← r3 + Mem[r4])
  - 命令1の第2オペランドのアドレッシングモードはベース、命令2の第2オペランドのアドレッシングモードは間接、が指定されています。
    - » オペランドフェッチにおいて、アドレス計算を行うステージをOA、メモリにアクセスするステージをOM、と表記することにします。
  - そのままパイプライン処理しようとすると



– このケースでも、命令2がOMステージに入るのを1サイクル遅らせるようにインタロック制御しなければなりません。

– パイプライン中にバブルが生じるとパイプライン処理の効果（高速化）が低減します。
  » バブル（bubble）：パイプラインとして動作はしているものの、（インタロック制御の結果などによって）有益な仕事をしていないステージを指します。
– 特に、アドレッシングモードによってオペランドフェッチのステージが伸び縮みする（ステージ数が変化する）と、バブルも増え、また、パイプライン制御自体も複雑にします。
  • 高速化を目的として考案した命令パイプライン制御方式なのに、実際には理想通りに高速化できません。
  • そこで。。。（次ページの話題に移ります）

CISCとRISC

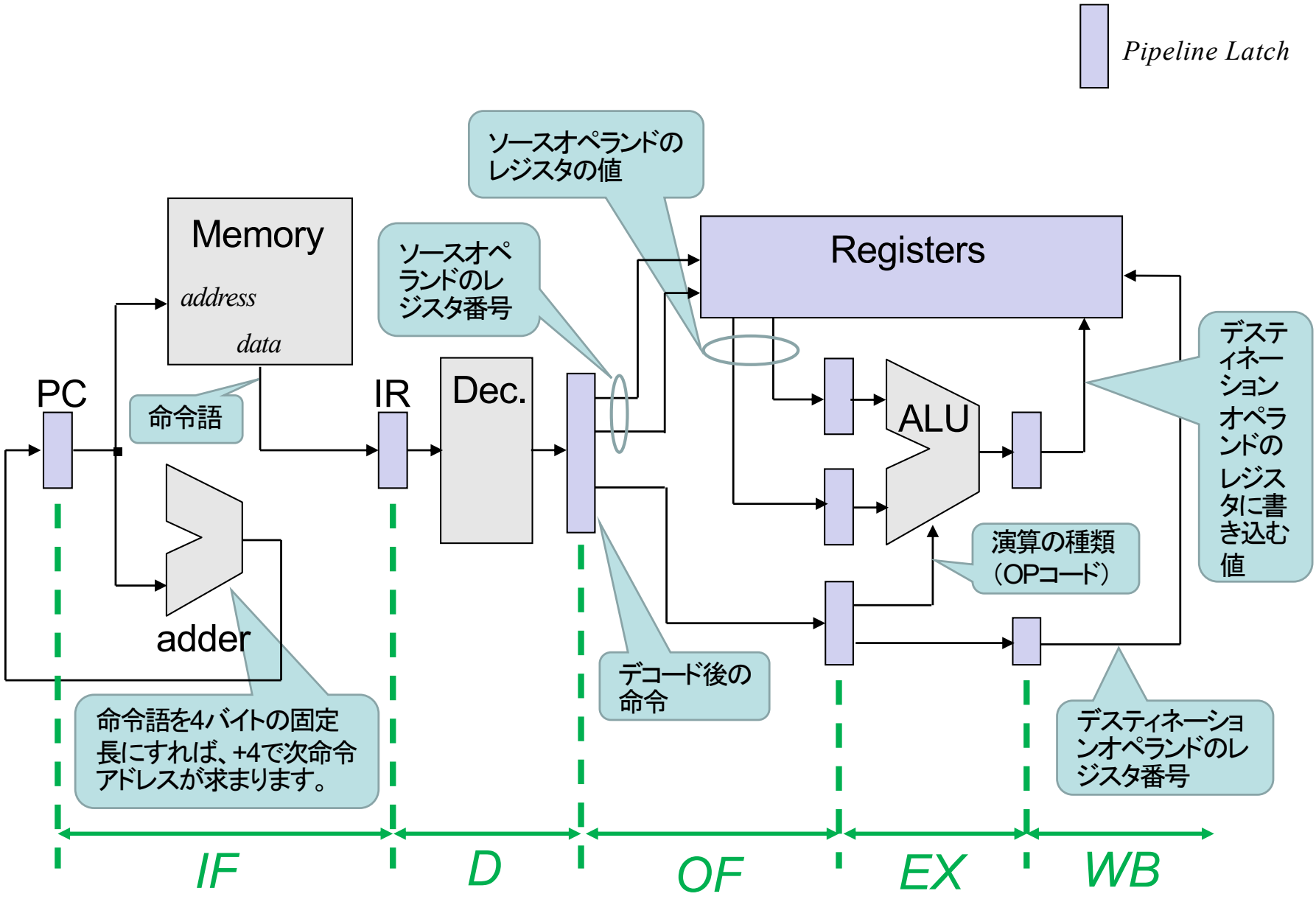
- 複雑な（高機能な）命令からなる命令セットを持つプロセッサでは、パイプラインがスムーズに流れません。
  - 1980年代になって、マシン命令を高機能化することによって性能を上げようとしてきたそれまでのやり方に、方向転換が必要になりました。
- では、逆に、パイプラインがスムーズに流れるような命令セットとはどのようなものでしょうか？
  - その答えとして提唱されたのが、RISC（Reduced Instruction Set Computer）です。

- RISCでは、
  - 命令の機能や形式を単純化しました。
    - これによって、命令パイプラインを流れやすくする（バブルを減らす）とともに、パイプラインのピッチも小さくできると主張しました。
  - 複雑な処理は単純な命令を組み合わせて使用することでカバーすることにしました。もちろん、めったに使わない複雑な命令は備えません。
  - それまで主流であった高機能な命令を持つコンピュータをCISC（Complex Instruction Set Computer）と名付けました。この分野での“complex” は悪口に当たります。
    - コンピュータアーキテクチャの世界では、「RISCかCISCか？」で大論争が起こりました。

- それで、現在は？
  - 皆さんのパソコンなどで使用しているx-86の命令セットはCISCです。
    - しかし、CISCではコンピュータの性能を上げることはできませんし、現在もまだ主流で使われている、ということなどなかったでしょう。
    - 実は、命令セットはCISCですが、内部で、マシン命令をRISC風の内部命令に置き換えて（変換して）います。
    - つまり、見た目はCISCなのですが、内部のハードウェアは完全なRISCのプロセッサです。
      - マイクロプログラム制御方式ではありませんが、それに似た考え方を用いてRISC風の内部命令に変換することにより今日までのプロセッサの高速化を達成してきたのです。

ハードウェア構成

- ここでは、命令パイプライン処理を行う極めて簡単なプロセッサのハードウェア構成を示します。
  - パイプラインステージが、IF、D、OF、EX、WB、の5段パイプラインとします。
  - 演算はレジスタ間でのみ行うものとします。
- パイプラインステージの境界には、パイプラインラッチ（パイプラインレジスタ）を置きます。
  - パイプラインラッチの記憶内容がステージへの入力となり、ステージ内の組み合わせ回路で処理した結果が、次段のパイプラインラッチの入力になります。


Pipeline Latch
ソースオペランドの
レジスタの値
Memory
address
data
ソースオペ
ランドのレ
ジスタ番号
Registers
デステ
ィネー
ション
オペラ
ンドの
レジス
タに書
き込む
値
PC
命令語
IR
Dec.
ALU
adder
演算の種類
（OPコード）
デコード後の
命令
命令語を4バイトの固定
長にすれば、+4で次命令
アドレスが求まります。
デスティネーショ
ンオペランドのレ
ジスタ番号
IF
D
OF
EX
WB