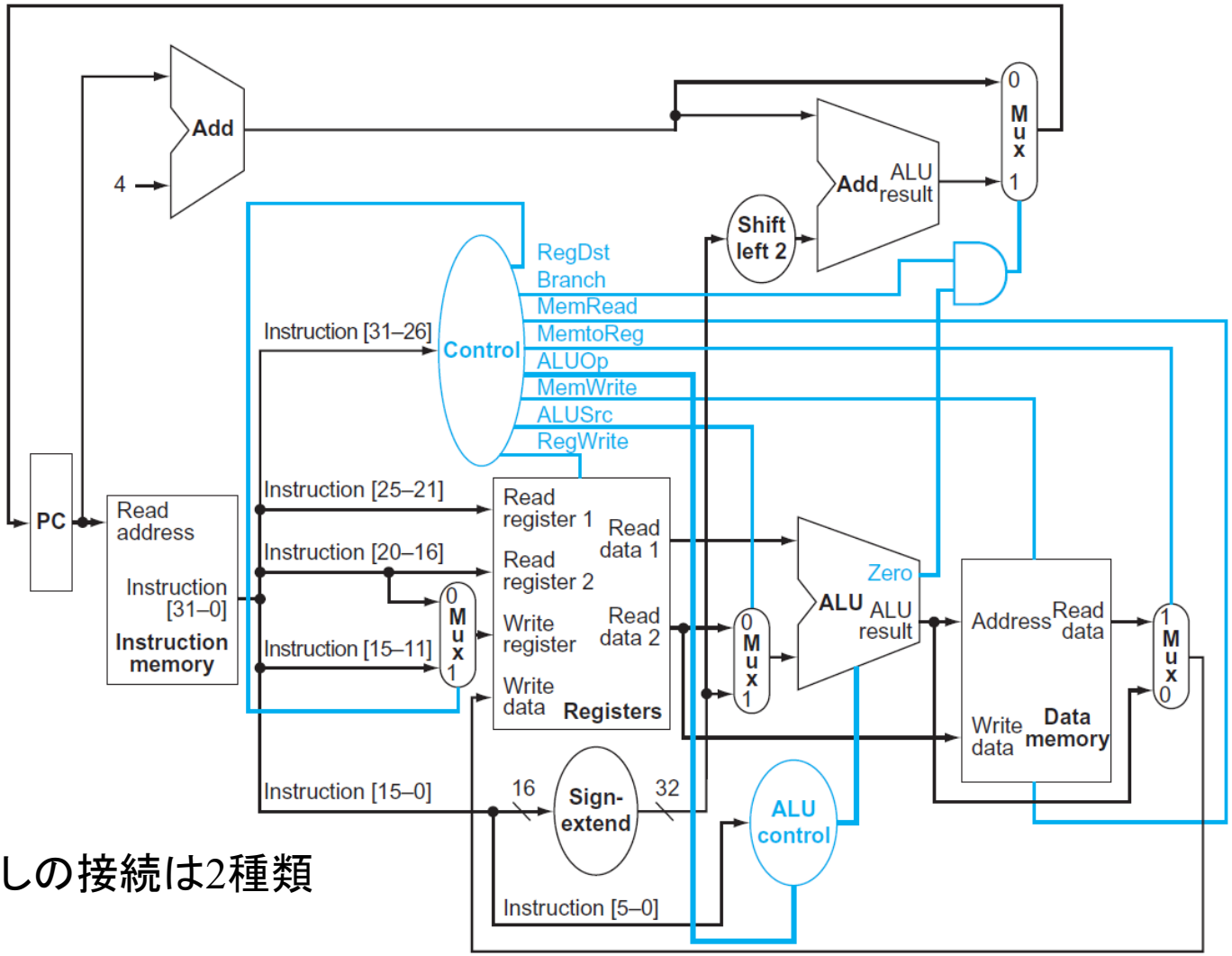
第7回
2018/12/25
プロセッサのマルチサイクル実装とパイプライン

図の多くはPatterson, Hennessy: Computer organization and design 5th editionより引用

前回解説したMIPSプロセッサ



機能ブロックどうしの接続は2種類

- データパス
- 制御

前回のプロセッサ実装の問題点

「シングルサイクル設計」と呼ばれ、一命令の実行が1クロックに相当

- 命令実行がたった1クロックですんでくれるなら良いように見える・・・
- 常にClock per instruction (CPI) = 1

実際には、以下の理由で用いられない

- 命令種類ごとに必要な時間が違うのに、最も長いものにクロックサイクルを合わせなければならない
  - 例: addの遅延 > andの遅延
- 各機能ブロック (ALU, レジスタ、メモリ…)の、働いている時間の割合が低い

→ マルチサイクル化＆パイプライン

マルチサイクル (Multicycle)による設計

- 複数のクロックサイクルで、一つの命令を実行

一命令の実行を、ステージに分割して考える

- 命令フェッチ(IF):メモリから命令(32bitデータ)を取り出す
- 命令デコード(ID):命令種類を解釈し、必要なレジスタを読み出す
- 実行(EX): ALUなどを用いた演算を行う
- メモリアクセス(MEM):メモリのデータへアクセス
- 書き込み(WB): 結果をレジスタに書き込む

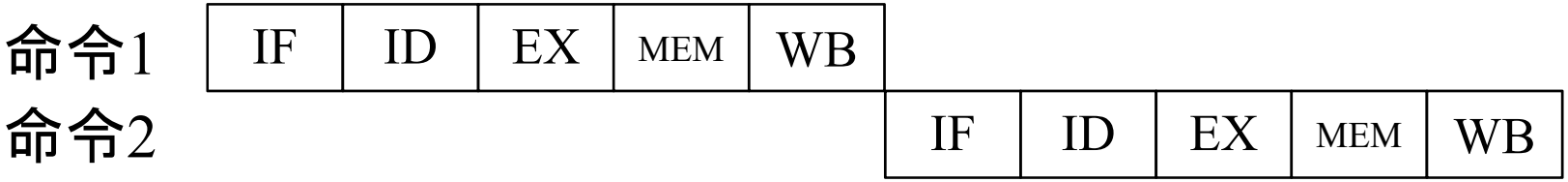
(本講義の例では)命令を
クロックサイクル 5つ分で実行

実際のプロセッサではさらに複雑になりうるが(20段、不定サイクル数など)、以上が基本

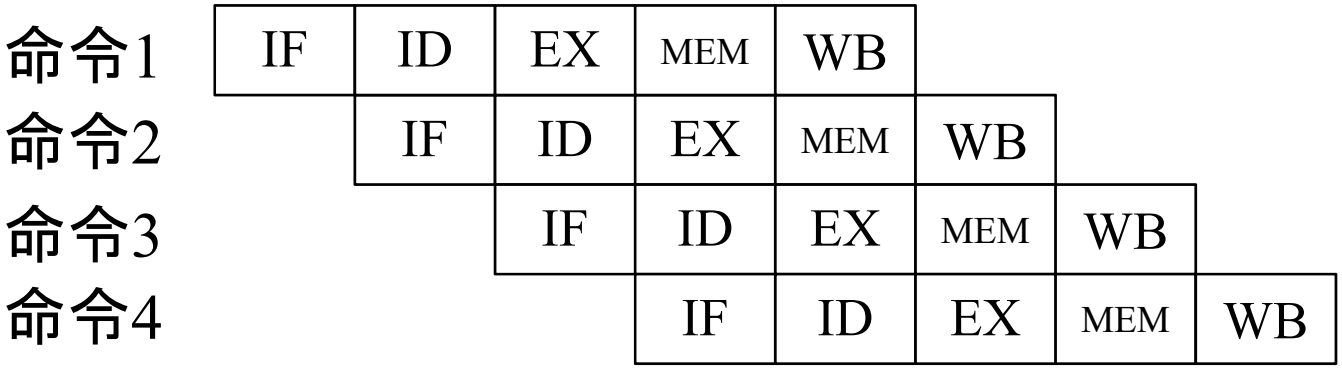
パイプラインの考え方

- 例：エスカレーターには、1人目が降りる前に2人目が乗ることができる
- 例：工場の流れ作業。刺身を切る人⇒皿に載せる人⇒皿を箱に入れる人・・・

パイプラインなし



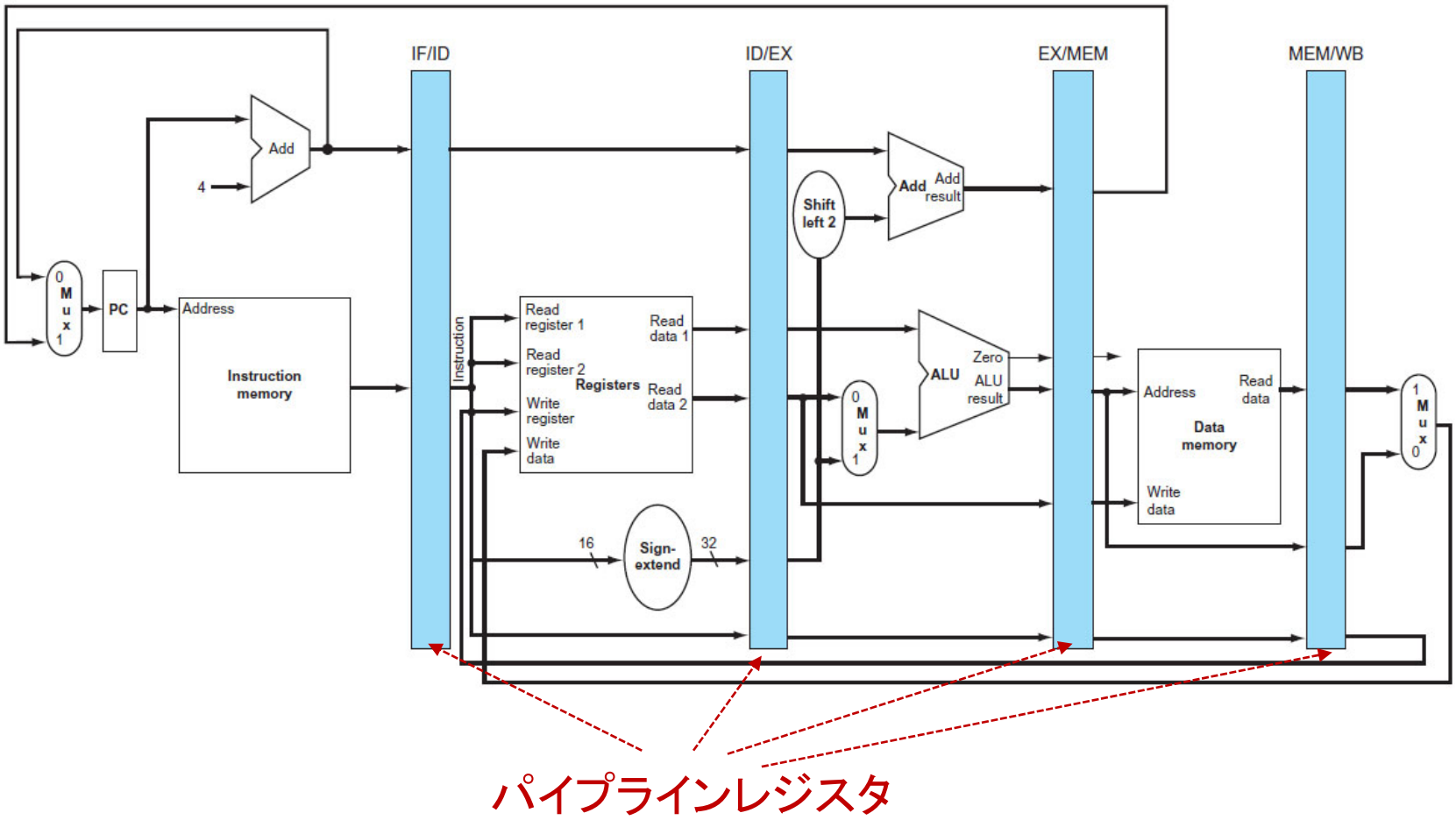
パイプラインあり



命令実行のスループットを向上させるのがねらい … ハードウェアではどう実現？

マルチサイクル・パイプラインのあるプロセッサ

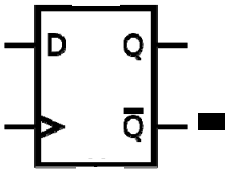
命令に関する情報は、PCを起点として(ほぼ)左から右へ



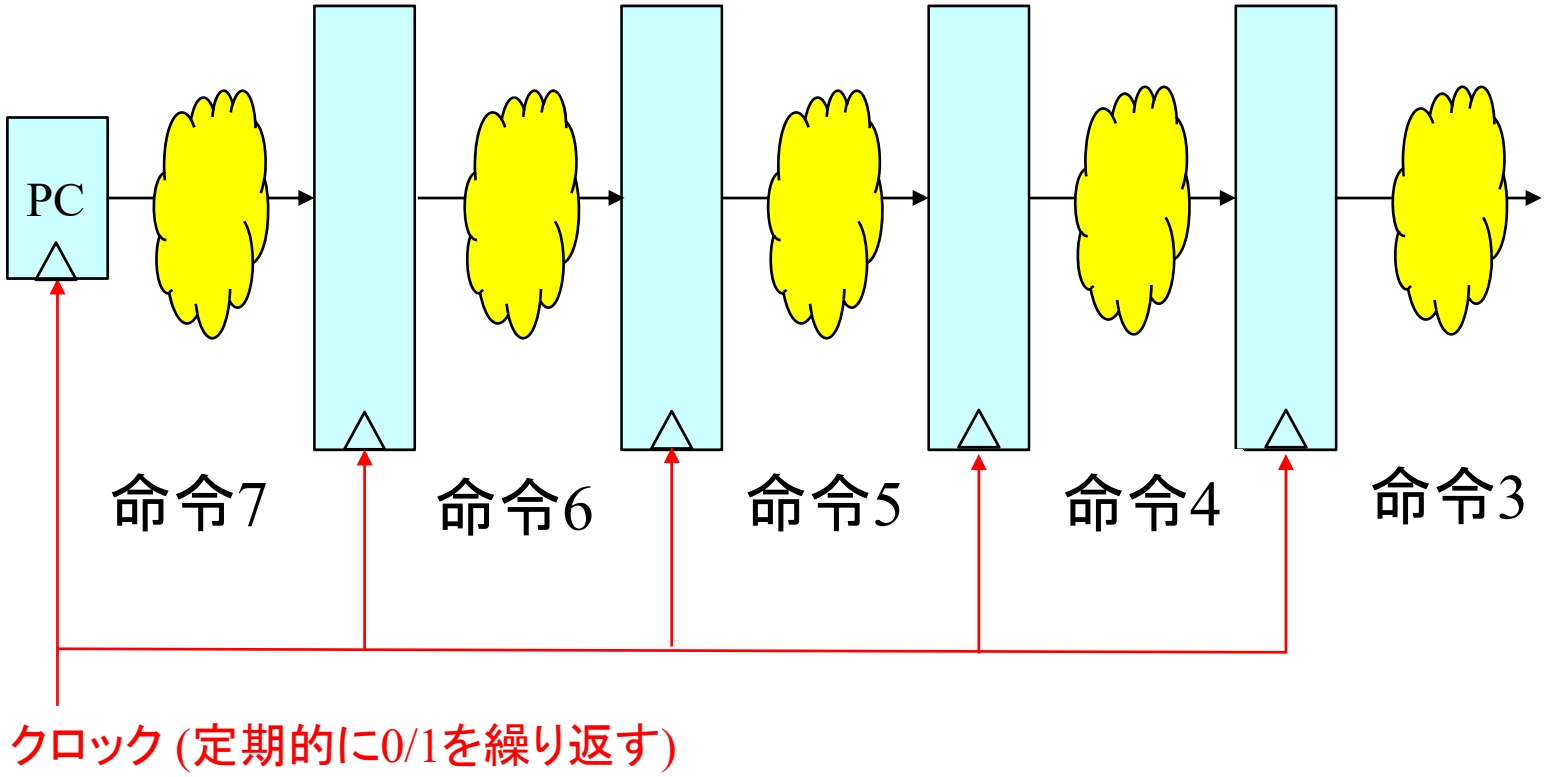
※制御(パス・ユニット)は省略

パイプラインレジスタ

Dフリップフロップ



- 基本的に、データを左から右へ流すだけのレジスタ
- D-フリップフロップなどの同期回路(状態を持つ順序回路の一種)でできている
- パイプラインのステージとステージを区切る役割



- PCとパイプラインレジスタたちには、同じクロックが与えられる (同期している)

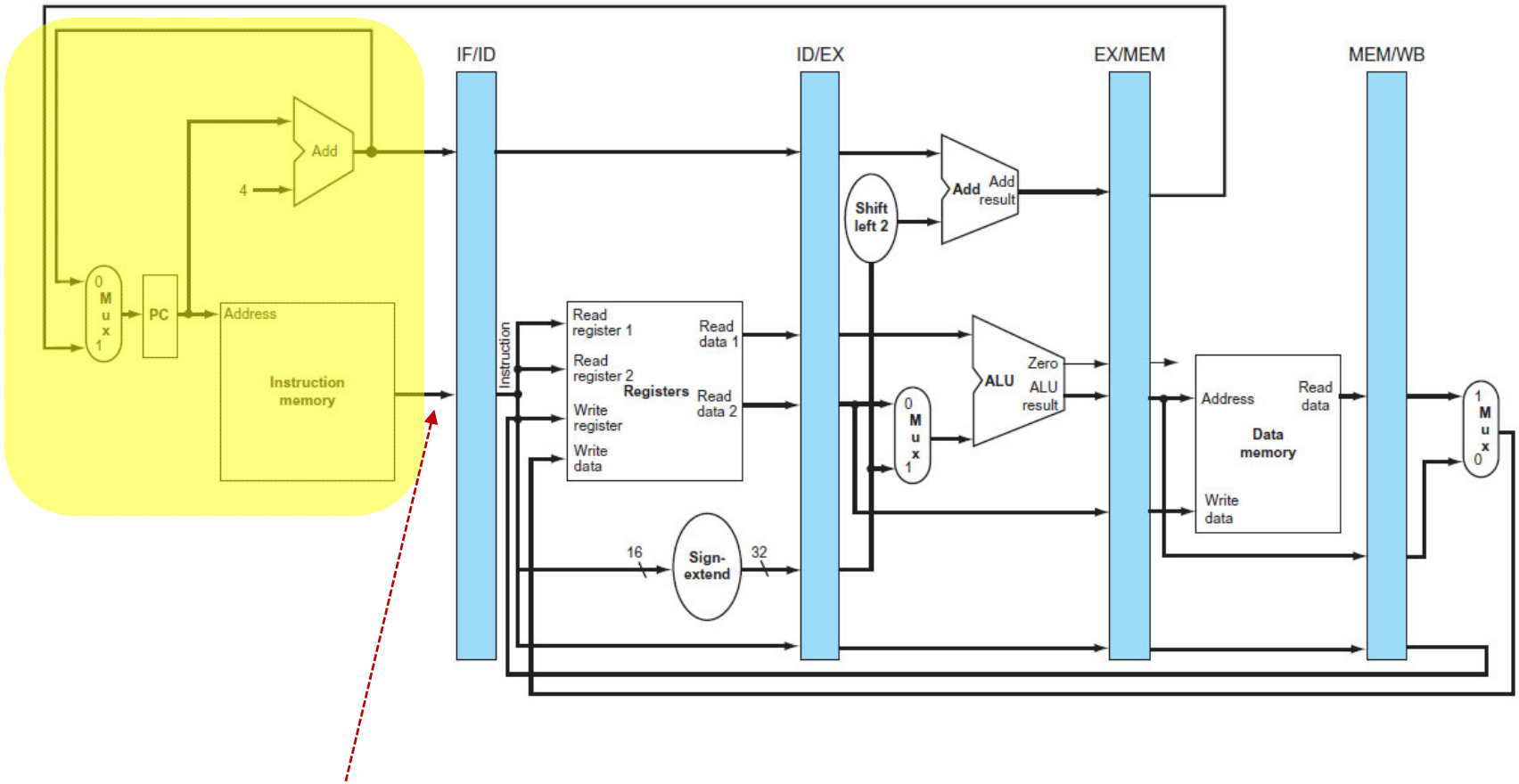
→ 1クロックごとに、命令の情報は左から右へ

第 1ステージ: 命令フェッチ(IF)

- PCが指し示すメモリの番地から命令を読み込んで、命令レジスタに書き込む
- PCに4を加算して、結果を再びPCに書き込む(次の命令のアドレス)
- 以下の疑似コードで動作を説明できる:

```
IR = Memory[PC];
PC = PC + 4;
```

第 1 ステージ: 命令フェッチ(IF)の動作



IR(32bit命令データ)。IFとIDを区切るパイプラインレジスタは、少なくともIRのデータを保持する

第 2 ステージ: 命令デコード(ID)
レジスタ値のフェッチも同時に行えるのがMIPSの特徴

R形式

op	rs	rt	rd	shamt	funct

- 命令語のビットフィールドで指定されるrs レジスタと rt レジスタを読み込んでおく
  - rdについては今は放置して次ステージへ渡す (先週との違い)

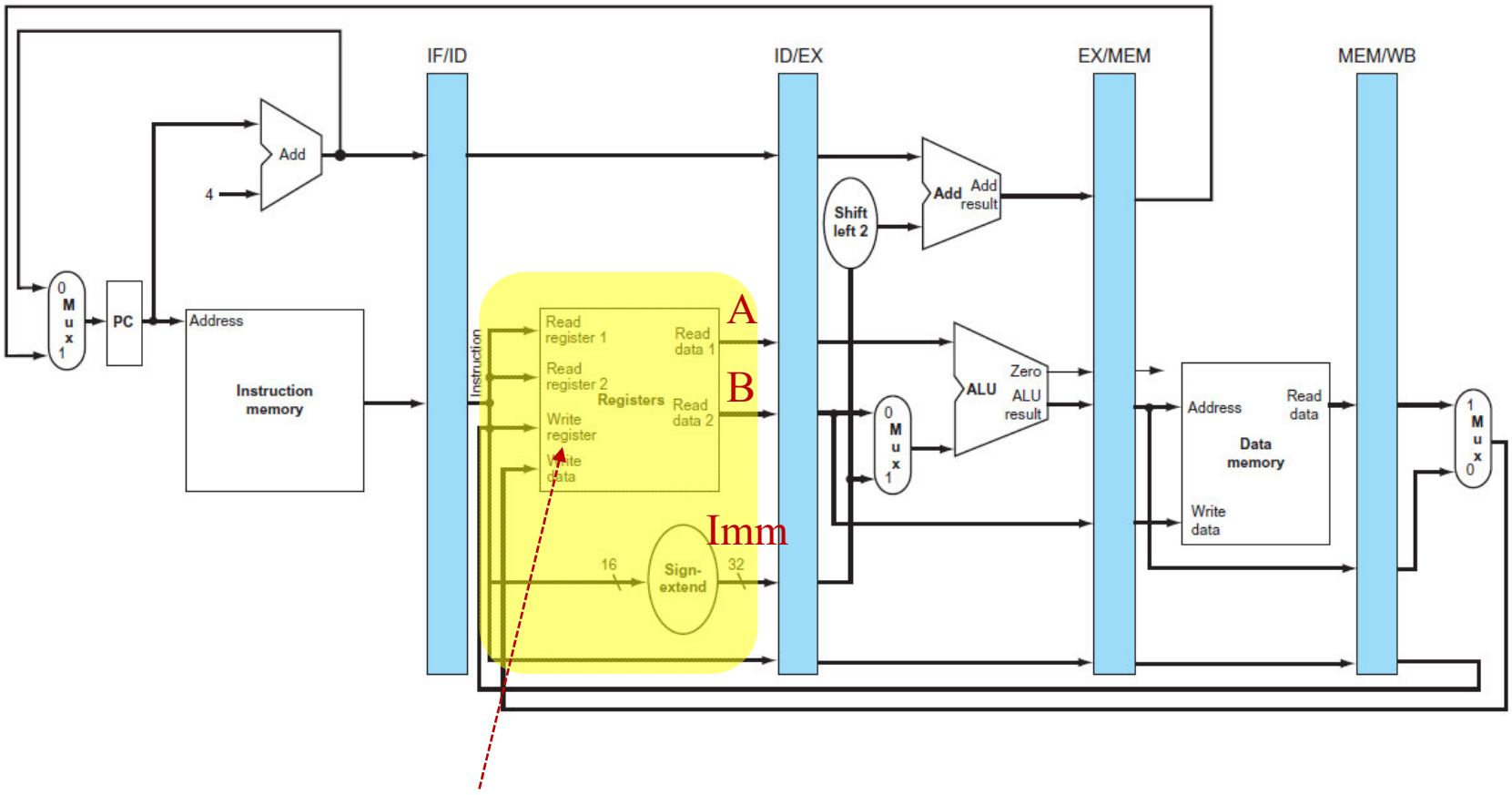
I形式

op	rs	rt	16 bit immediate

- 命令がI形式の場合に備えて、16bit即値を読み込んで、符号拡張しておく
  - I形式ではないかもしれないが気にしない
- 疑似コードでは:

```
A = Reg[IR[25-21]]; (命令レジスタのビットフィールド25-21)
B = Reg[IR[20-16]];
I = sign-extend(IR[15-0]);
```

第 2 ステージ: 命令デコード(ID)の動作



Write register/Write dataはまだ使わない

第 3 ステージ: 命令実行 (EX)

- このステージでは、命令種類によって、以下の異なる動作を行う
  - このためには制御が必要だが、未説明

(1-a) R-形式:

```
ALUOut = A op B;
```

(1-b) メモリ参照 (I-形式):

```
ALUOut = A + I;
```

(1-c) beq命令 (条件つきブランチ、I-形式):

```
ALUOut = A - B;
Zero = (A-B==0)? 1: 0;
```

(1-a), (1-b), (1-c)はどれもALUを使う。

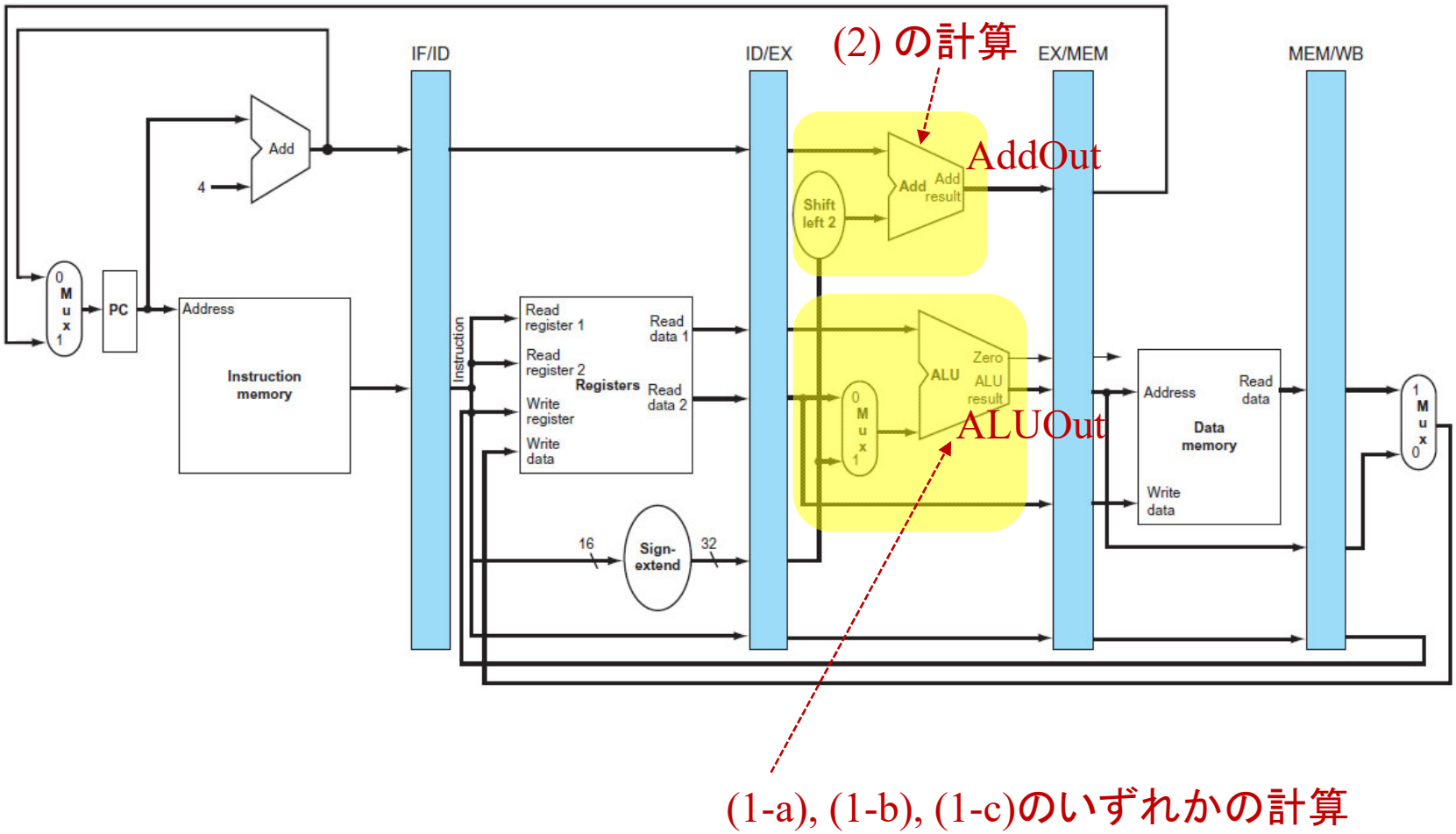
- 並行して、下記の計算を行っておく (ALUとは別のAdderを使う)

```
AddOut = PC + Imm << 2;
```

[Q] この計算は何命令のためか？

[Q] ALUと別にAdderを用意するのはなぜ？

第 3 ステージ: 命令実行(EX)の動作



第 4 ステージ: メモリアクセス(MEM)

命令種類によって異なる動作

(a) ロード命令もしくはストア命令の場合

```
(1) MDR = Memory[ALUOut];
```

又は

```
(2) Memory[ALUOut] = B;
```

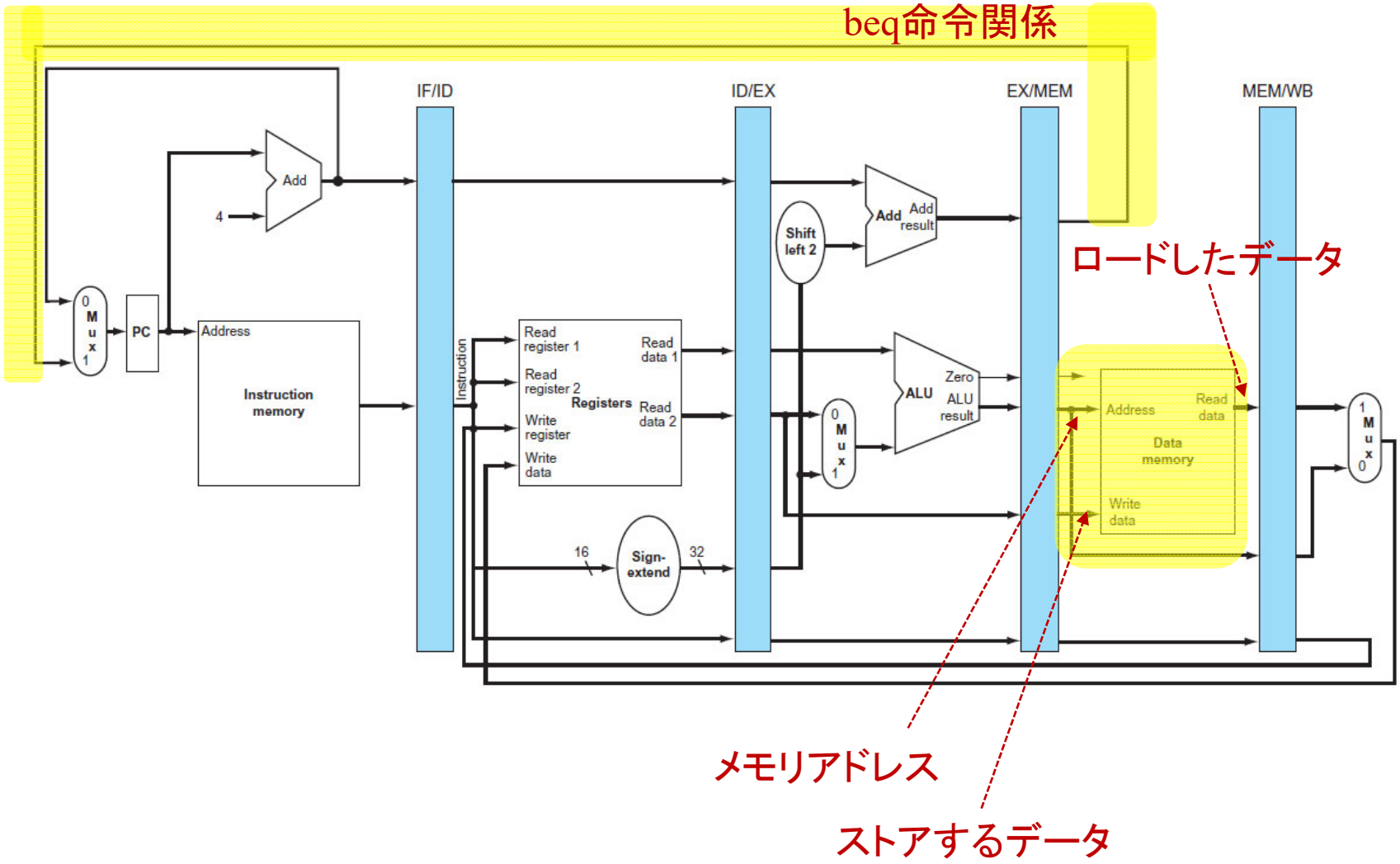
(b) beq命令の場合

```
if (Zero) PC = AddOut;   // 制御が必要
```

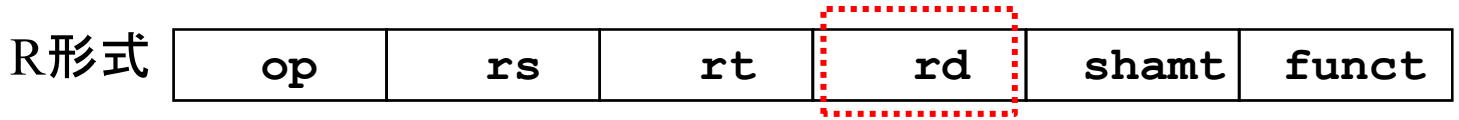
(c) R-形式命令の場合

なにもせず、 ALUOutを次ステージへ

第 4 ステージ: メモリアクセス(MEM)の動作

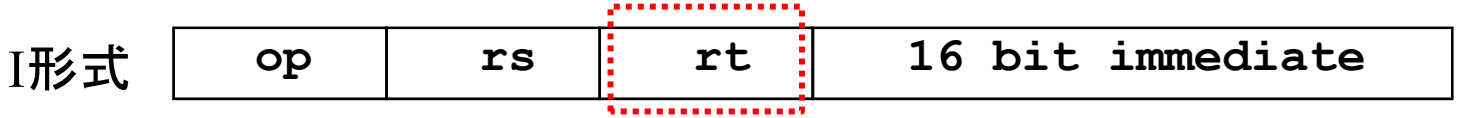


第 5 ステージ: 書き込み (WB, Write-back)



R形式命令の場合:

- Reg[IR[15-11]]= MDR;



ロード命令(I形式)の場合:

- Reg[IR[20-16]]= MDR;

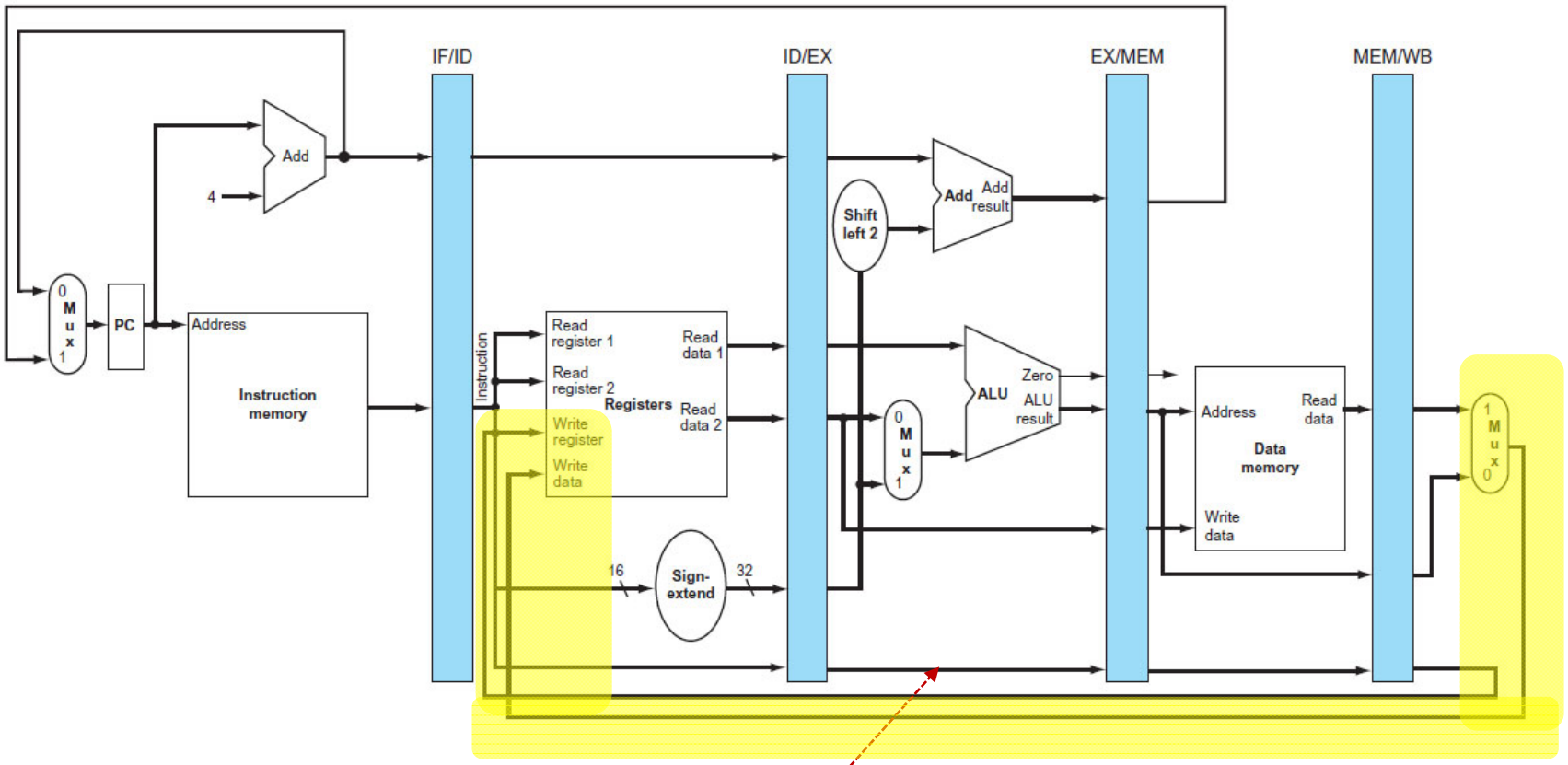
それ以外の命令の場合:何もしない

課題:

- IR(命令)のうち、使いたいbitが異なる
- そもそもIRを入力に使えるのは第2ステージのみ

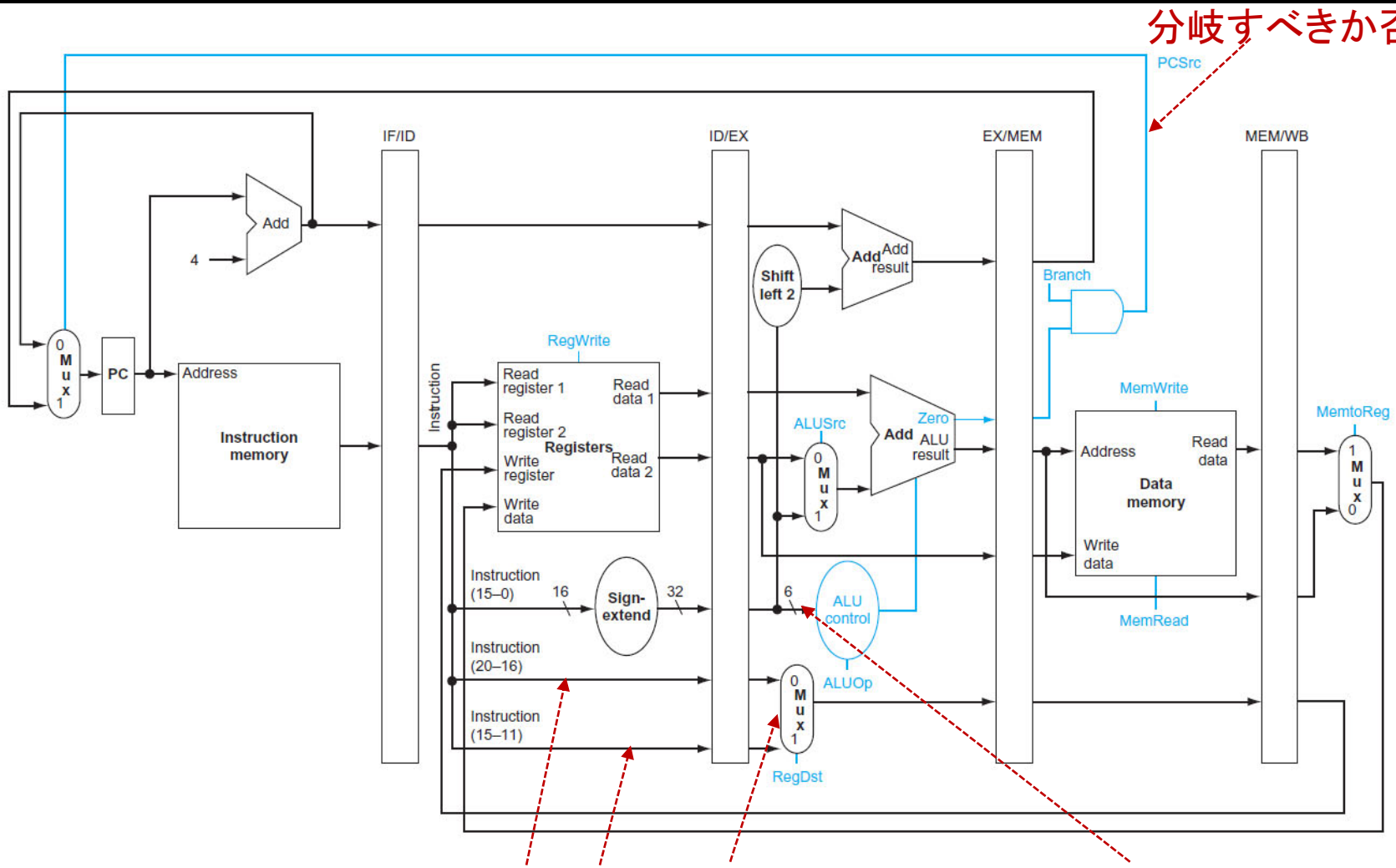
  [Q] 第2ステージから第5ステージに直接データパスをつなぐと何が悪い?

第 5 ステージ: 書き込み(WB)の動作 (1)



レジスタ番号を持ち越したい
実はまだこの図に足りない部分があった・・・

制御(一部)を加えたプロセッサ構成



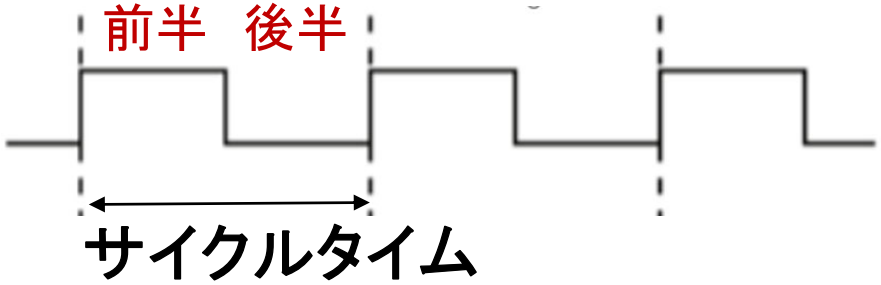
「各ステージは分離している」は本当か？

- レジスタファイルへの操作
  - 第2ステージで読み込み
  - 第5ステージで書き込み
- 何が問題？ → 1サイクル内で下記が起こる
  - ある命令(アドレス10000)によるレジスタ書き込みと「同時」に、3つ後の命令(アドレス10012)によるレジスタ読み込みが起こる
  - データハザード(次回)という問題の一種

- この場合は、レジスタファイルの仕組みの工夫で対応可能
  - クロックサイクルの前半では書き込みのみ可能
  - クロックサイクルの後半では読み込みのみ可能

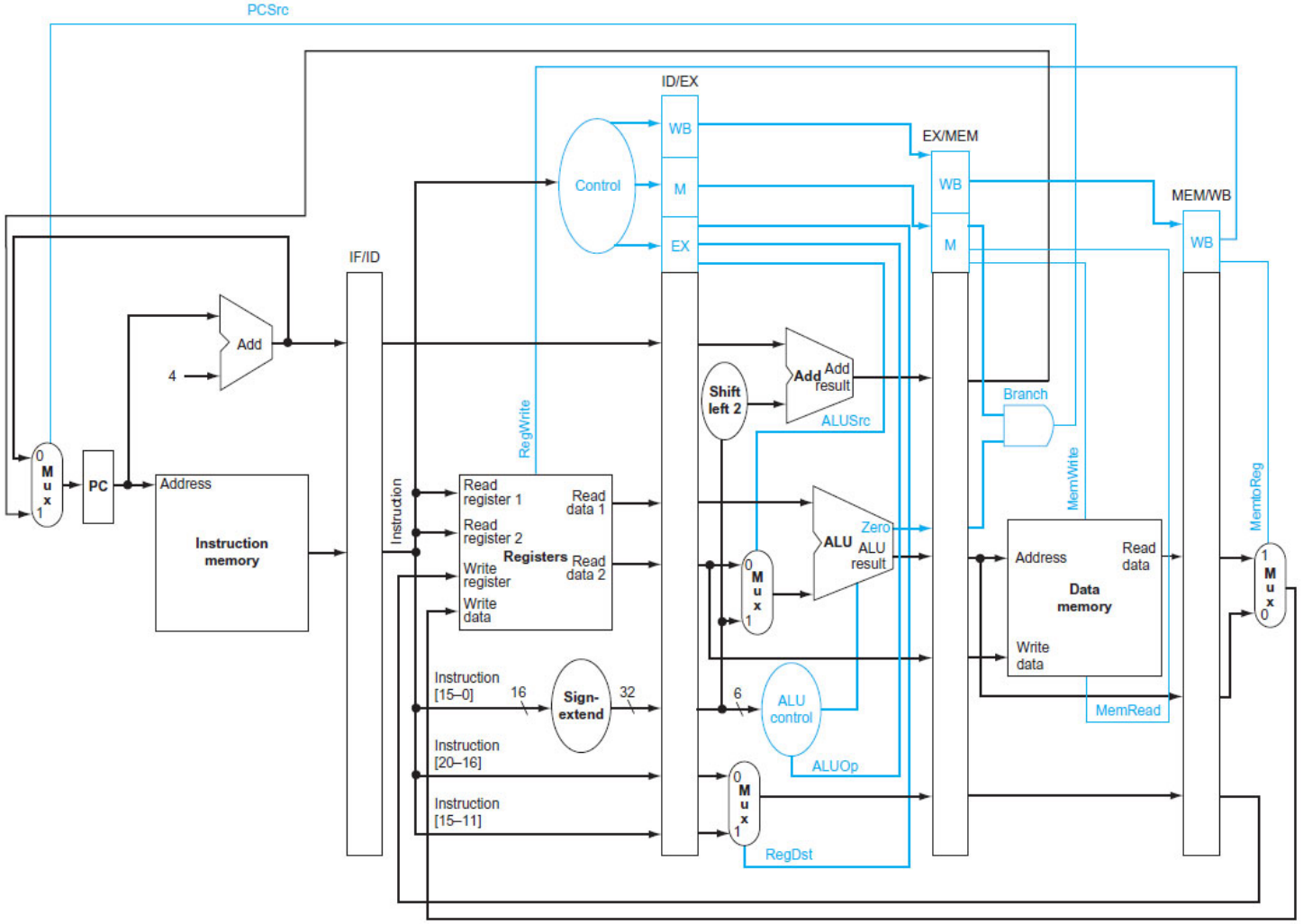
  という仕組みにしておく

C



同様に、メモリ, PCも仕組みが工夫されているとする

制御を加えたプロセッサ構成 (今日の最終版)



主制御ユニットが第2ステージに、ALU制御ユニットが第3ステージに置かれた
制御データ(WBの中にRegWriteなど)もシフトレジスタに追加

今回のまとめ

- 前回のプロセッサを改良して、マルチサイクル＆パイプラインへ対応した
- 前回のシングルサイクルとの違い
  - 1命令＝複数サイクル
    - Clock per instruction(CPI)が高い→これ自身は良くない
  - 1命令が実行途中に次の命令を初めてしまう
    - 命令をステージに分け、パイプライン実行
    - ハードウェア上は、ステージ間はパイプラインレジスタで区切られる
- 複雑化してもこれらを行いたい利点

→ クロック周波数を圧倒的に速くできる

→ 機能ユニットの稼働率を上げることができる

- 次回：パイプラインの複雑さはこんなものではない → ハザードの問題
  - beqで「分岐しよう」と決意したときにはもう3命令くらい進んでいる
  - $s3 = $s1+$s2; $s5 = $s3+$s4 は今日の仕組みでは正しく実行できない