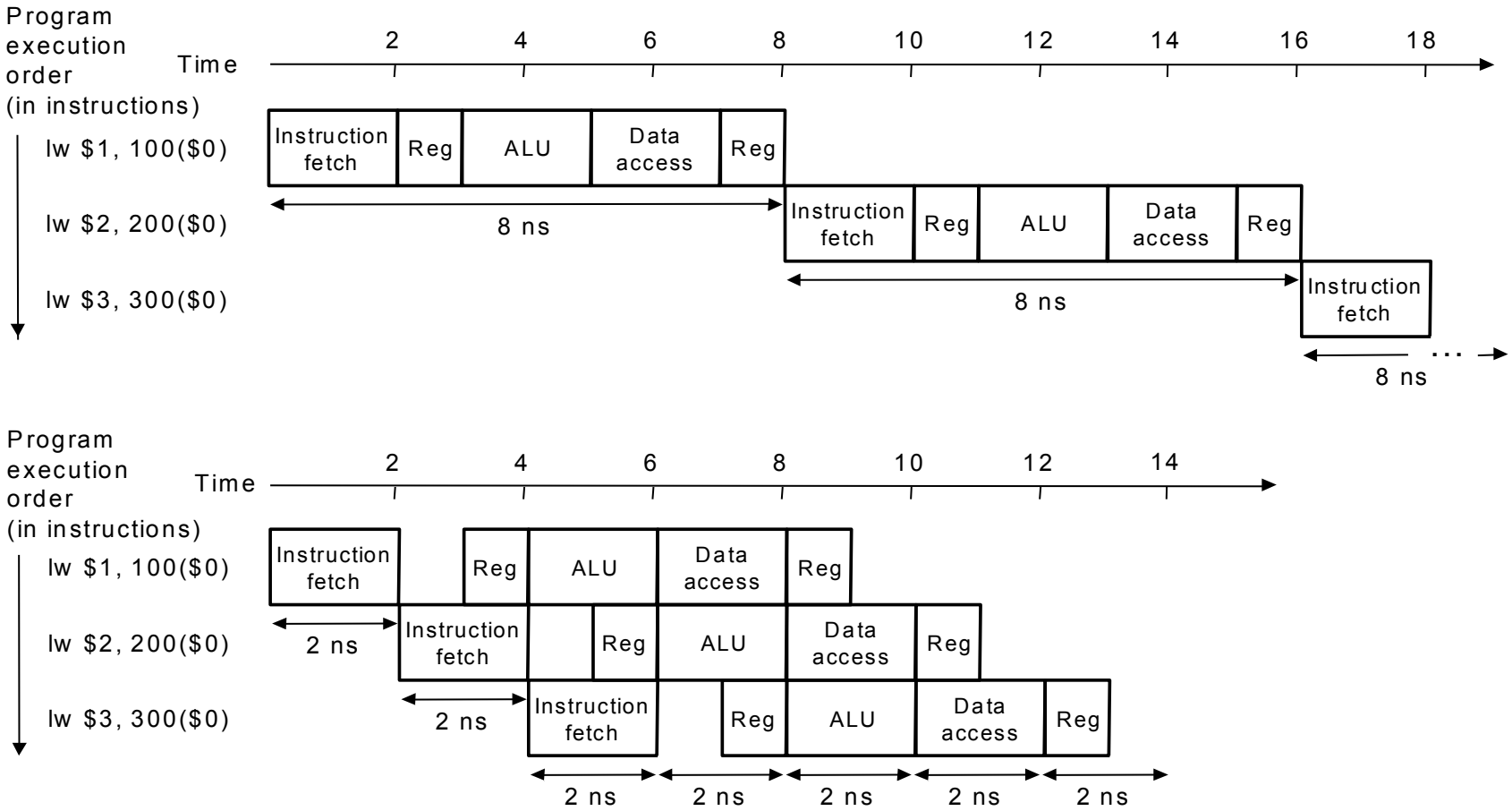
2012/07/2, 7/6
「パイプライン・パイプライニング」

パイプライン・パイプライニング(1) (Pipeline, Pipelining)

- マルチサイクル実装は、シングルサイクルと比較して、CPIが著しく高い
  - クロックの上昇にはさまざまな限界→速度向上の限界
- 一つの命令の各ステップ同士は、依存性がある→逐次的に実行する必要がある
- 複数の命令の各ステップは、並列に行えるか?
  - 複数の連続する命令の各ステップを、重ねて実行する
  - 例: ある命令をフェッチし、第二ステップでデコードしているときに、次の命令のフェッチを行う (ステージ)
  - すべての命令の実行ステップにおいて、次の命令の直前の命令ステップを重ねて実行する→パイプライニング
    - c.f. スーパースカラ
  - 理想的には、CPI = 1 (Q: CPI < 1は可能?)

パイプライニング(2)

- 複数の命令を並列にオーバーラップして実行して、命令実行のスループット(処理速度)を向上することにより、性能を上げる



理想的な並列性＝スピードアップはパイプライン中のステージ数。これは達成可能だろうか?

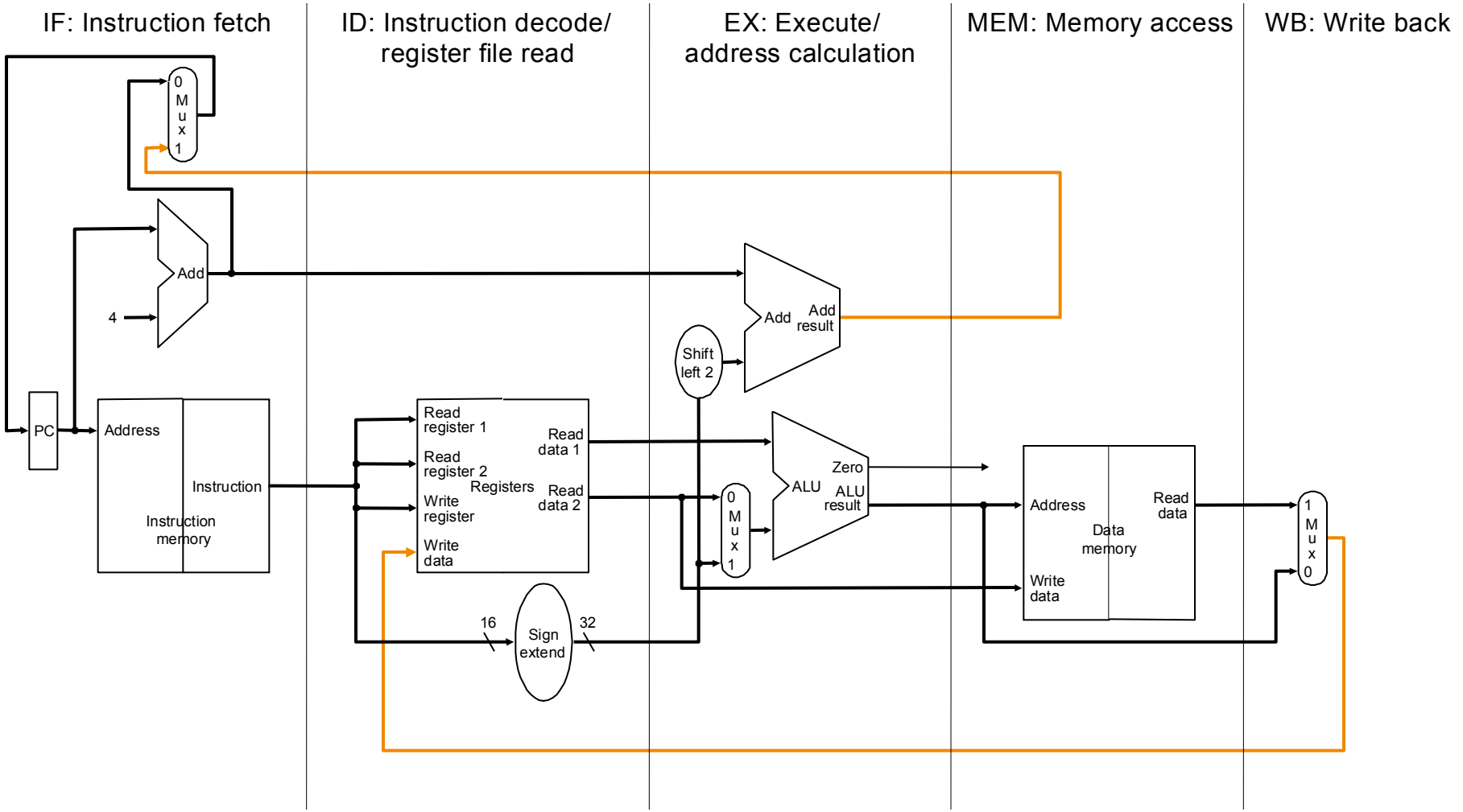
パイプライニングの特性

- 容易にする性質 ← RISCプロセッサの特性
  - 全ての命令長が同一
  - 命令形式が少ない
  - メモリオペランドがロード･ストア命令にしか現れない

- 困難にする性質(障害)→ パイプラインハザード (pipeline hazard)
  - 構造ハザード (structural hazards): メモリは同時にはアクセスできない
  - 制御ハザード (control hazards): 分岐があるときは、制御がどちらに移るかわからない←分岐が確定しないと、パイプライニングができない
  - データハザード (data hazards): ある命令の入力が、直前の命令の結果に依存

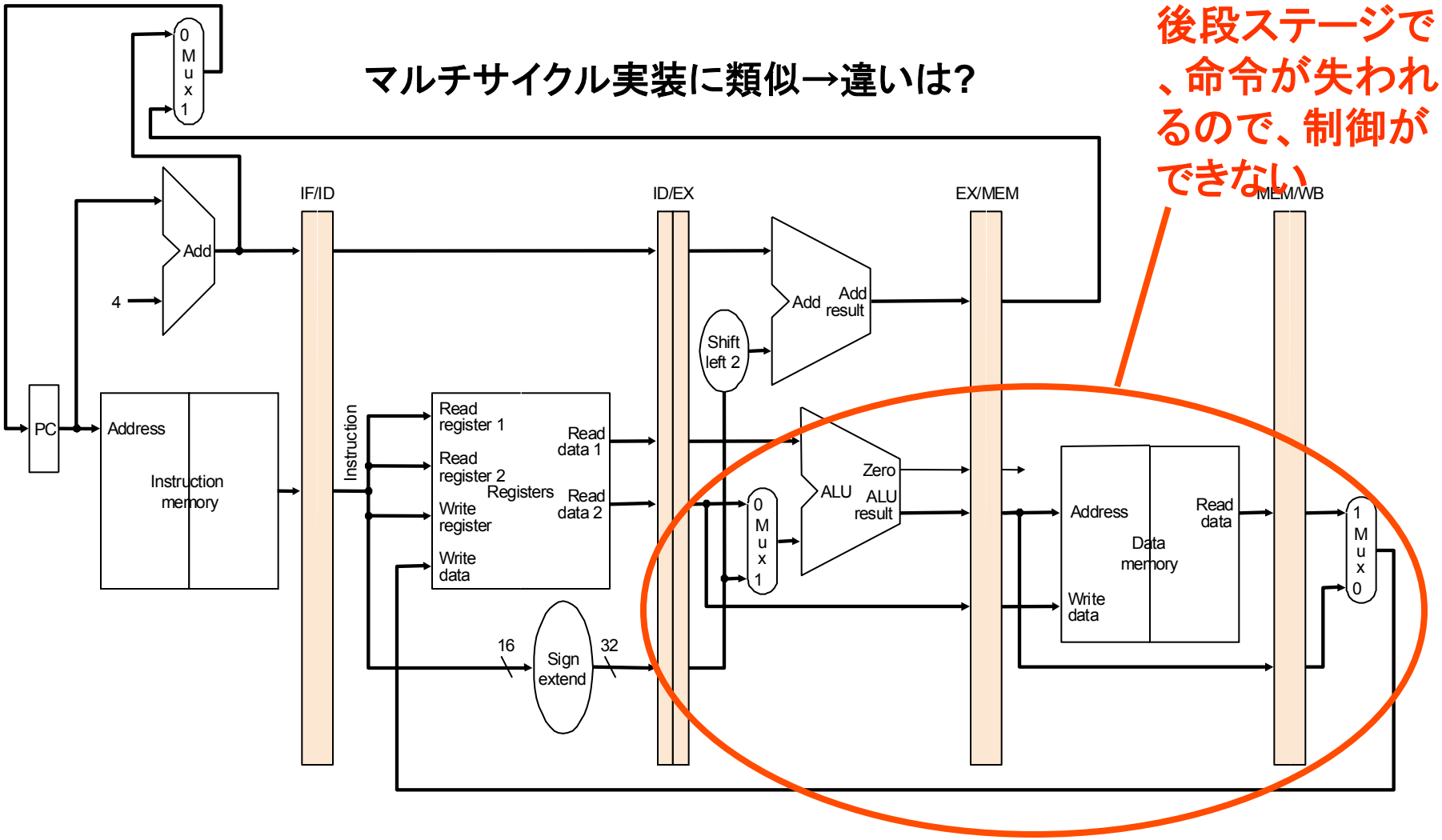
- 現代のプロセッサでは、さらに困難な問題がある:
  - 例外処理
  - out-of-order 実行、speculative実行など (この授業ではとりあげない)

パイプライニングの基本的なアイデア-ステージ分割



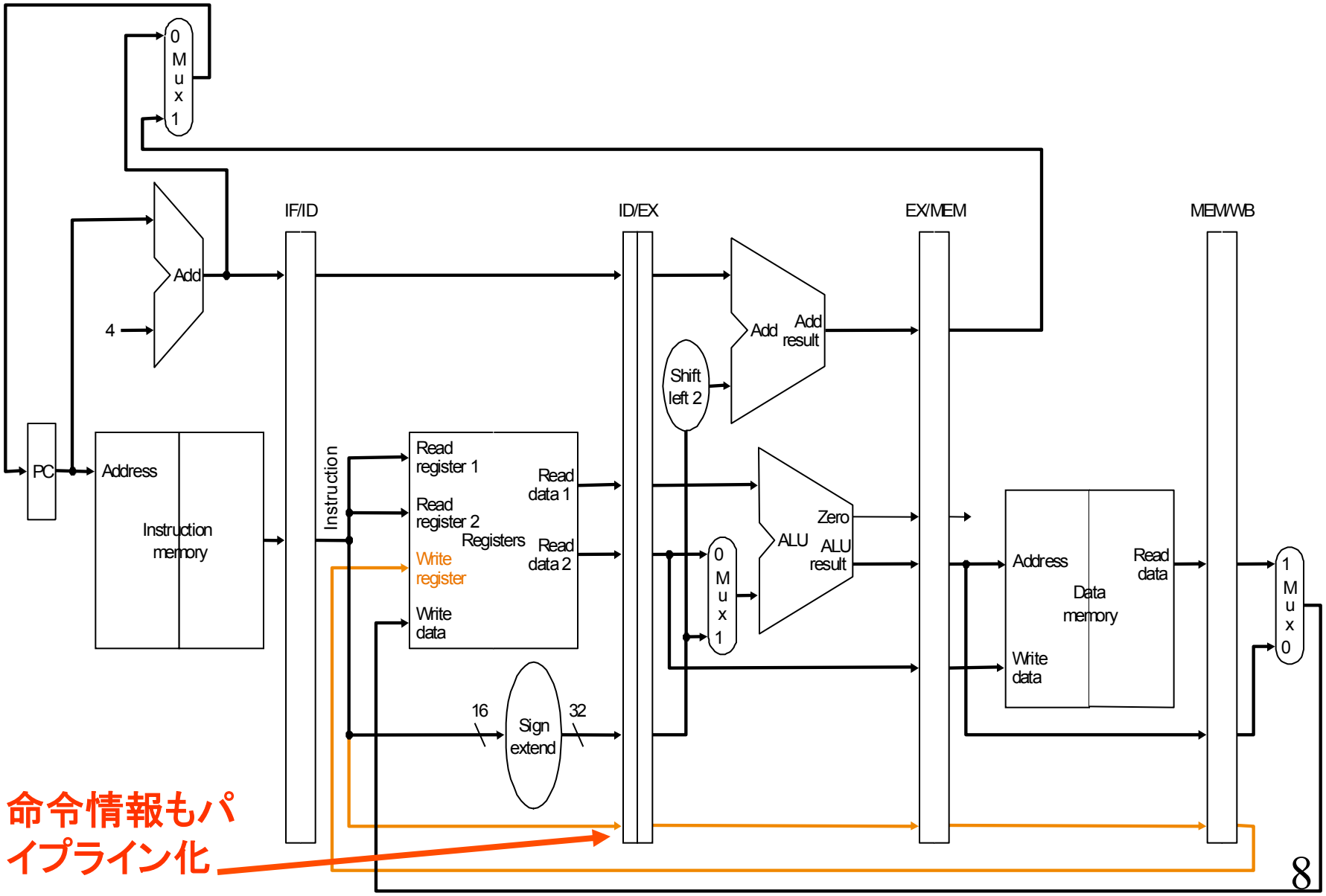
- *シングルサイクル実装を、それぞれのクロック毎に実行する「ステージ」(Stage)に分割、命令は複数のステージで実行*
- *データパスをステージに分割するにはどのようにすれば良いか?*

パイプライン化されたデータパス

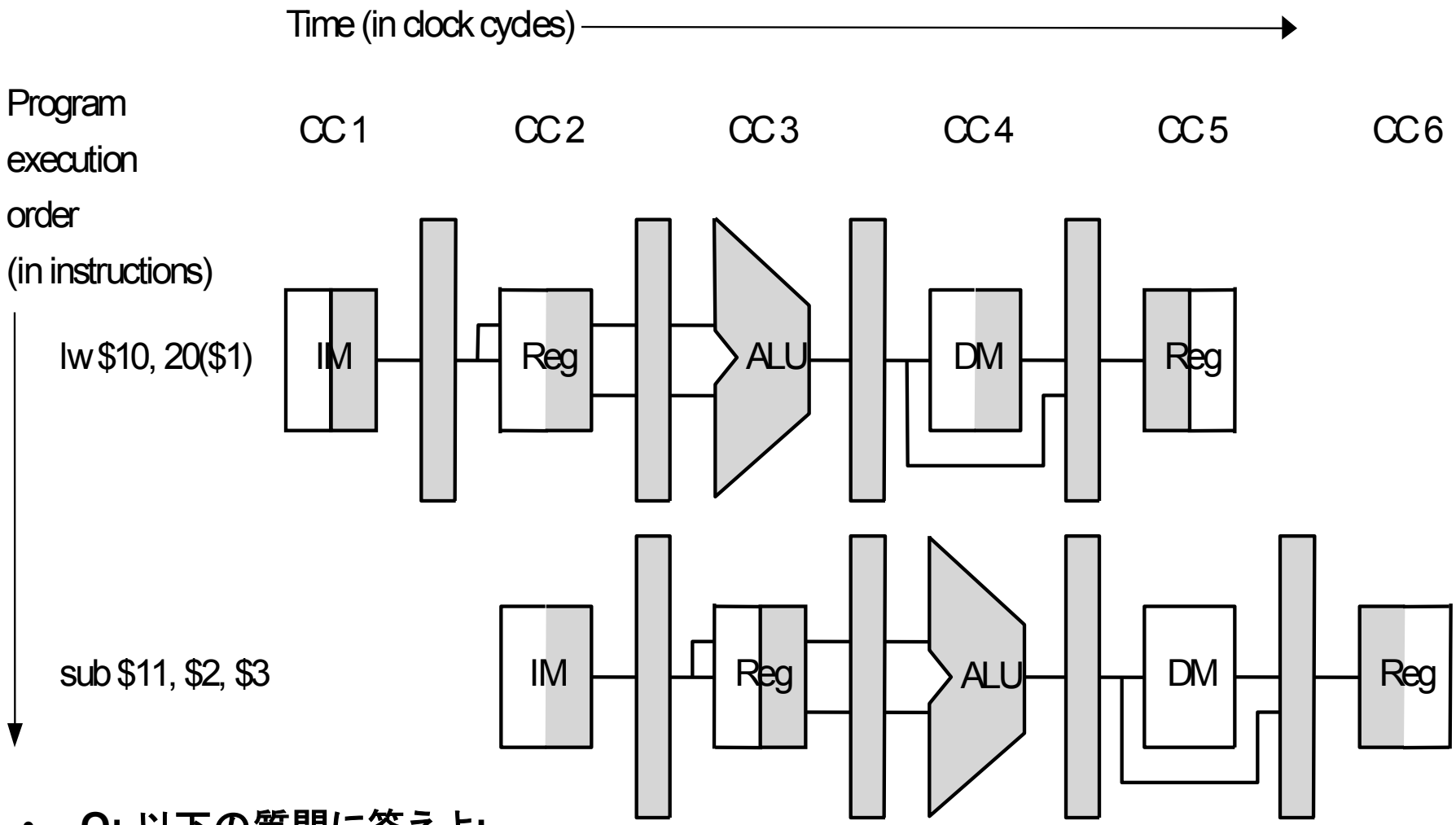


依存性がない場合も、どのような命令を実行すると問題が生じるか?

修正されたデータパス

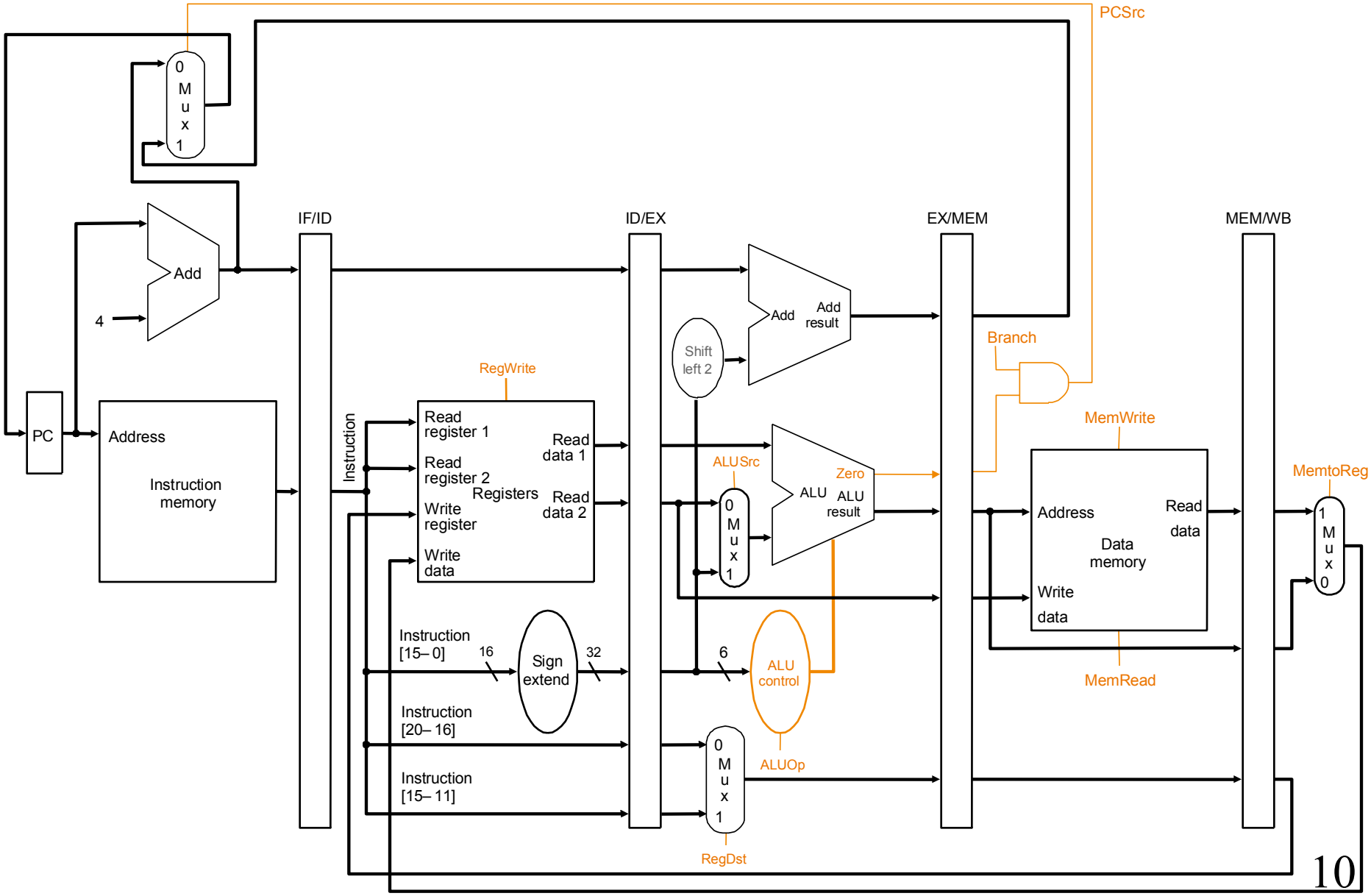

0
M
u
x
1
Add
4
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
Add
result
Shift
left 2
PC
Address
Instruction
memory
Instruction
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
0
M
u
x
1
ALU
Zero
ALU
result
Address
Data
memory
Read
data
Write
data
1
M
u
x
0
16
Sign
extend
32
命令情報もパイプライン化

パイプラインのグラフィカルな表現



- **Q: 以下の質問に答えよ:**
  - **このプログラムを実行するのに、何サイクルかかる?**
  - **第4サイクルにおいて、ALUは何を実行している?**

パイプラインの制御


PCSrc
0
M
u
x
1
Add
4
IF/ID
ID/EX
EX/MEM
MEM/WB
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALUSrc
ALU
Zero
ALU result
Branch
MemWrite
MemtoReg
Address
Data memory
Read data
Write data
MemRead
Instruction [15–0]
16
Sign extend
32
6
ALU control
ALUOp
Instruction [20–16]
Instruction [15–11]
RegDst

パイプラインの制御

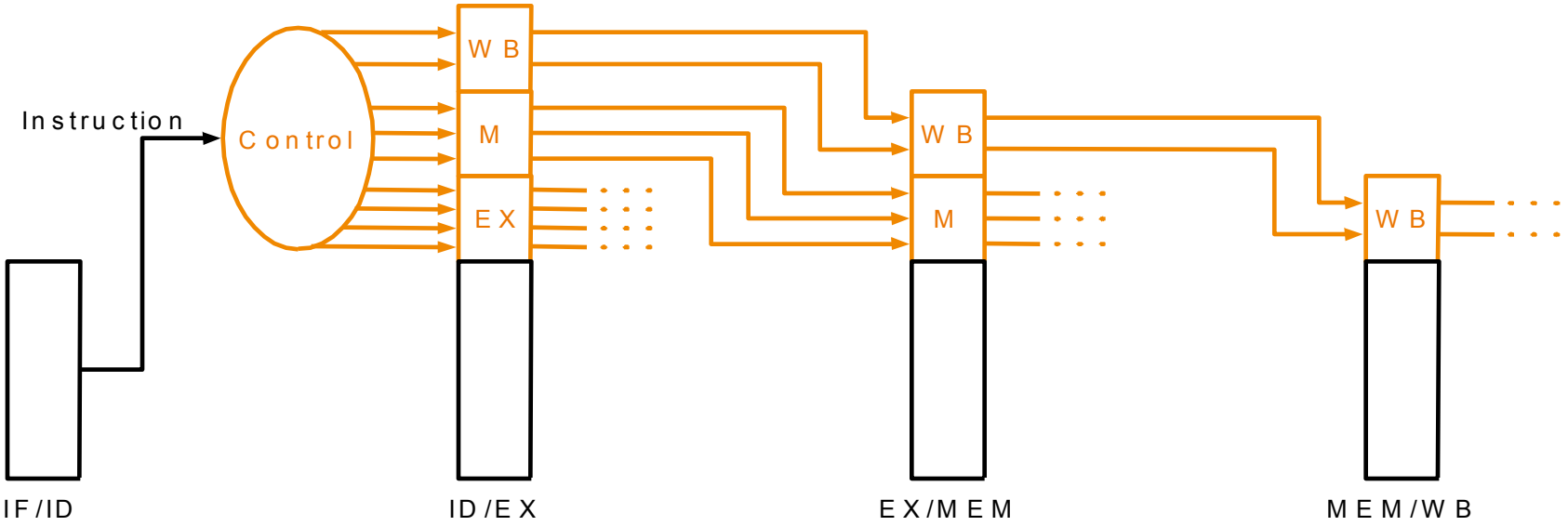
- MIPSでは5ステージのパイプライン。それぞれのステージでは、何をどのように制御する必要があるか?
  - 命令フェッチとPCのインクリメント
  - 命令デコード / レジスタ読み出し
  - 命令実行
  - メモリ書き込み
  - レジスタ書き込み (Write Back)

- 制御はずっと複雑になる。どのように制御を行うか?
  - 1個所で集中管理?
  - 有限状態機械を用いるべき?

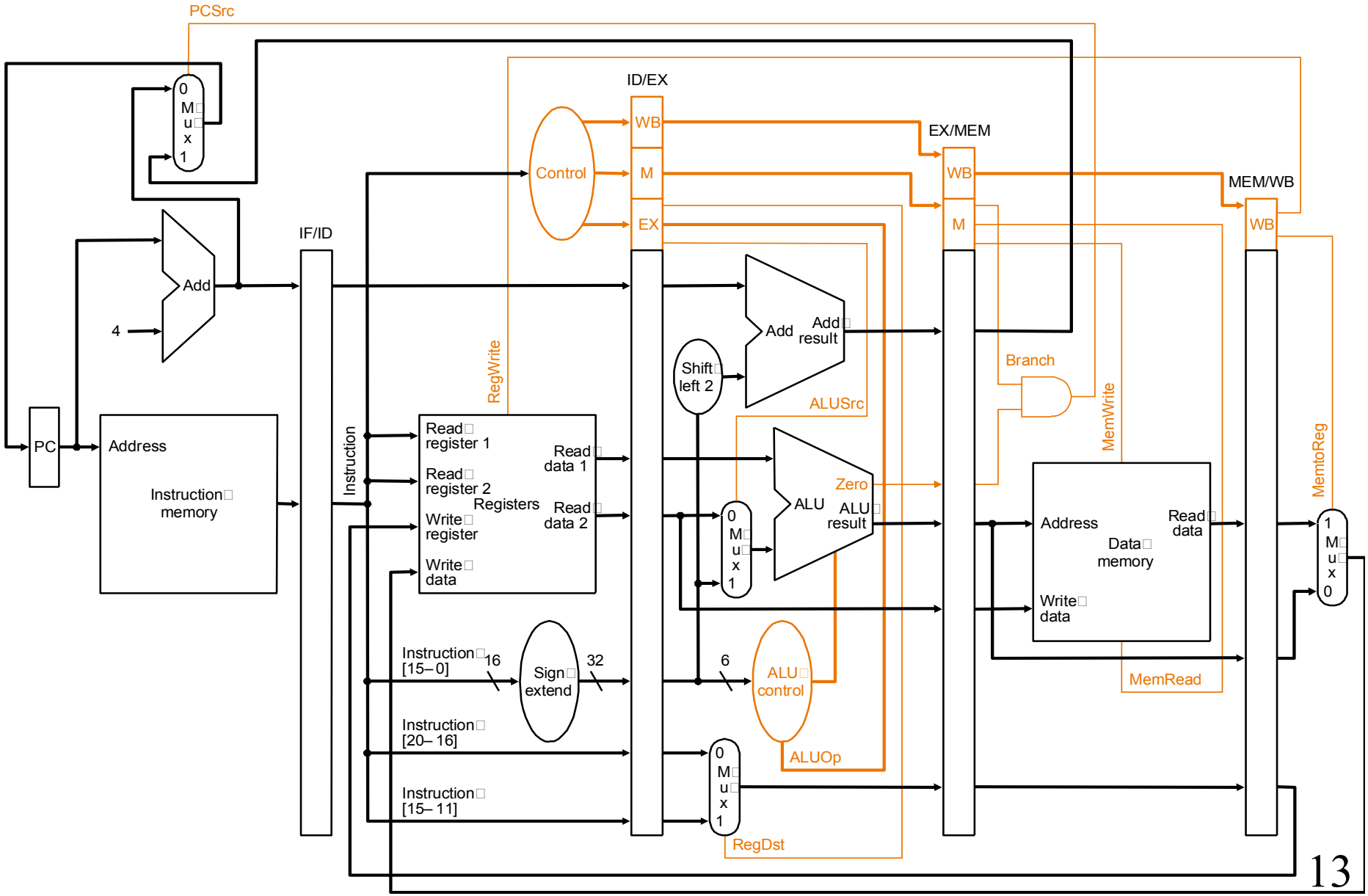
パイプラインの制御(3)

- 先の命令同様、制御信号をデータと同じようにパイプライン上を伝達させる

Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

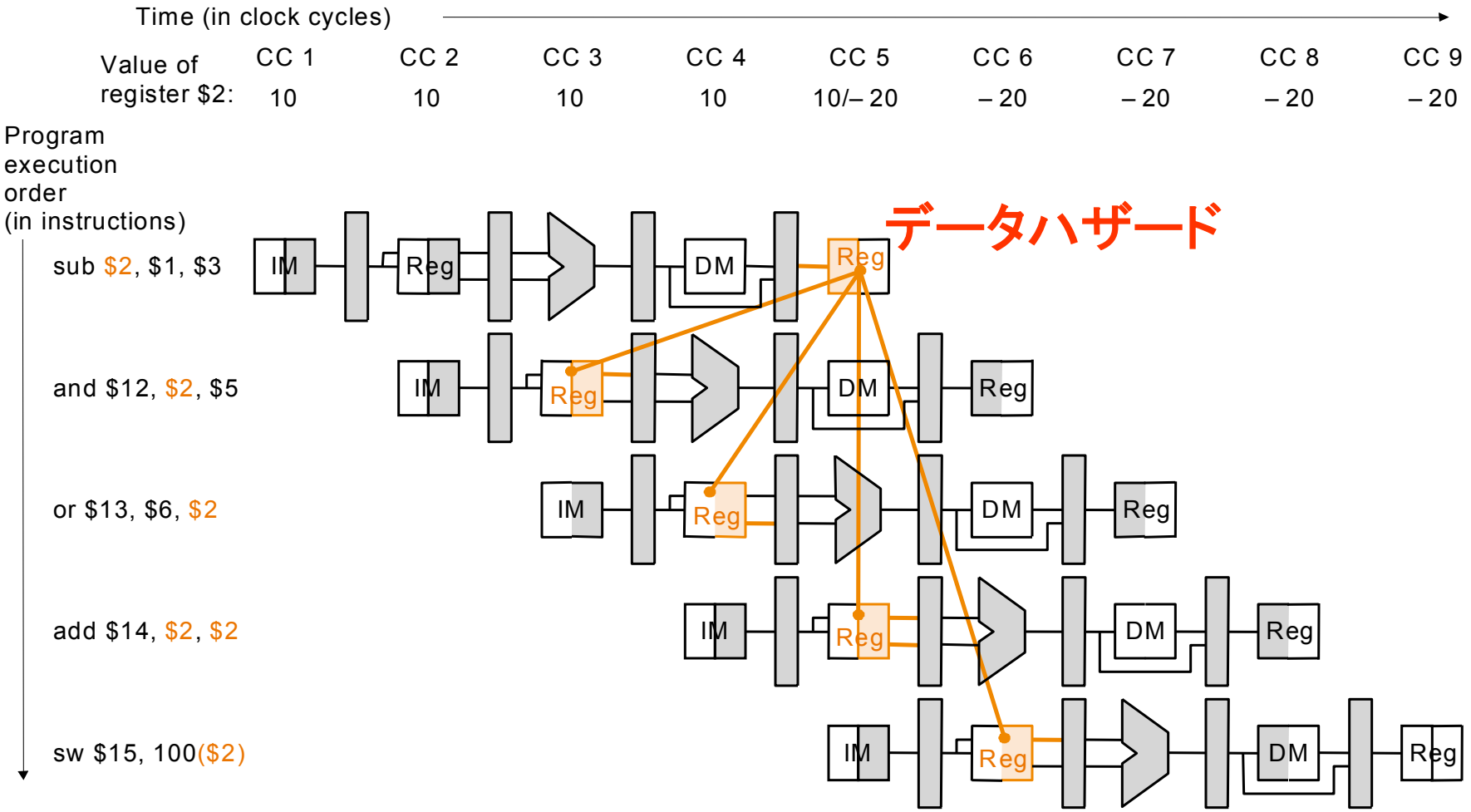


制御を含んだデータパス



データパス内のデータの依存性

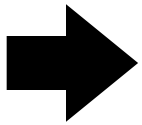
- データハザード: ある命令の実行が完了する前に、次の命令を実行する問題点
  - 「過去」に戻るデータの依存性により、データハザードが生じる



ソフトウェアによる解決

- コンパイラによる解決→データハザードが起きないように保証
- NOP (Non-Operation、無効命令)を適切な個所に挿入
- どこに挿入すればよいか?

```
sub   $2, $1, $3
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
```



```
sub   $2, $1, $3
nop
nop
nop
and   $12, $2, $5
nop
  :
```

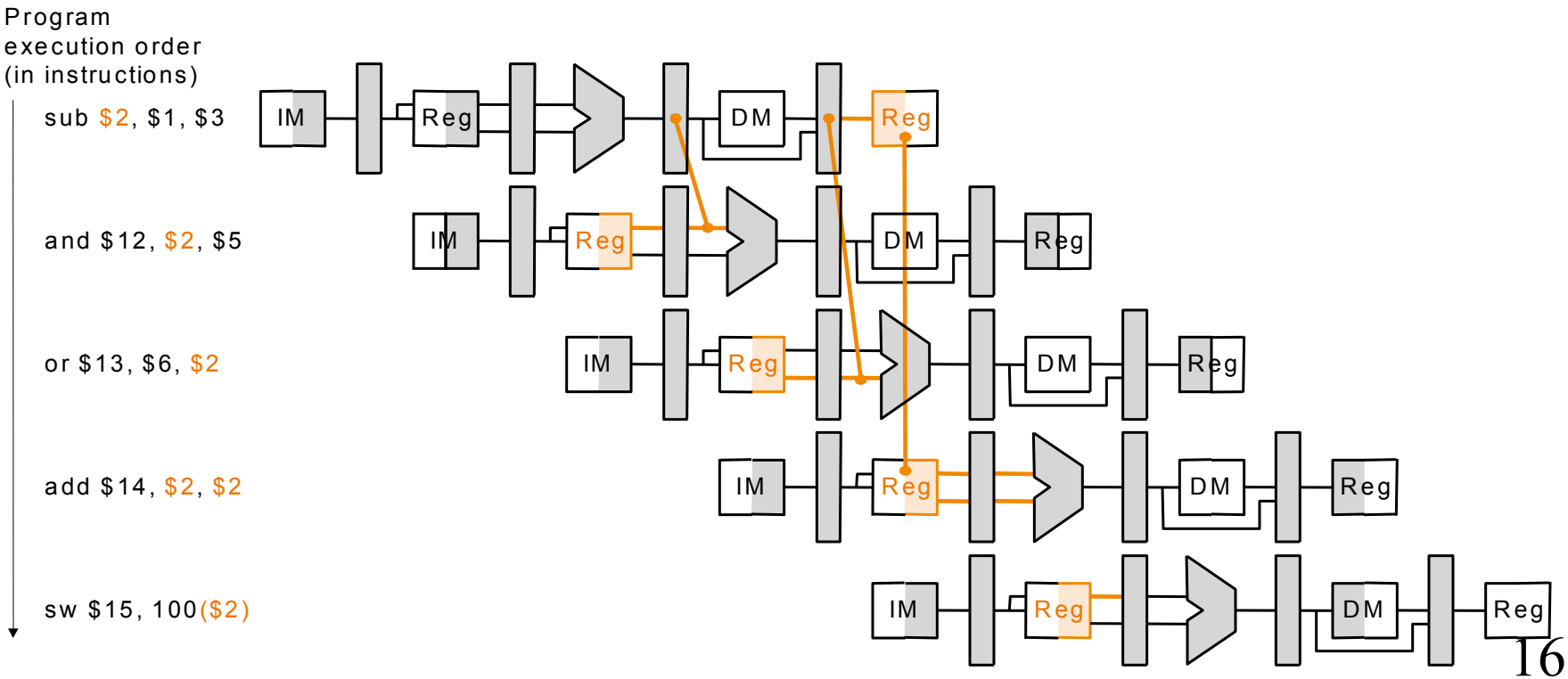
- 問題: これにより極端に遅くなる

解決策: フォワーディング

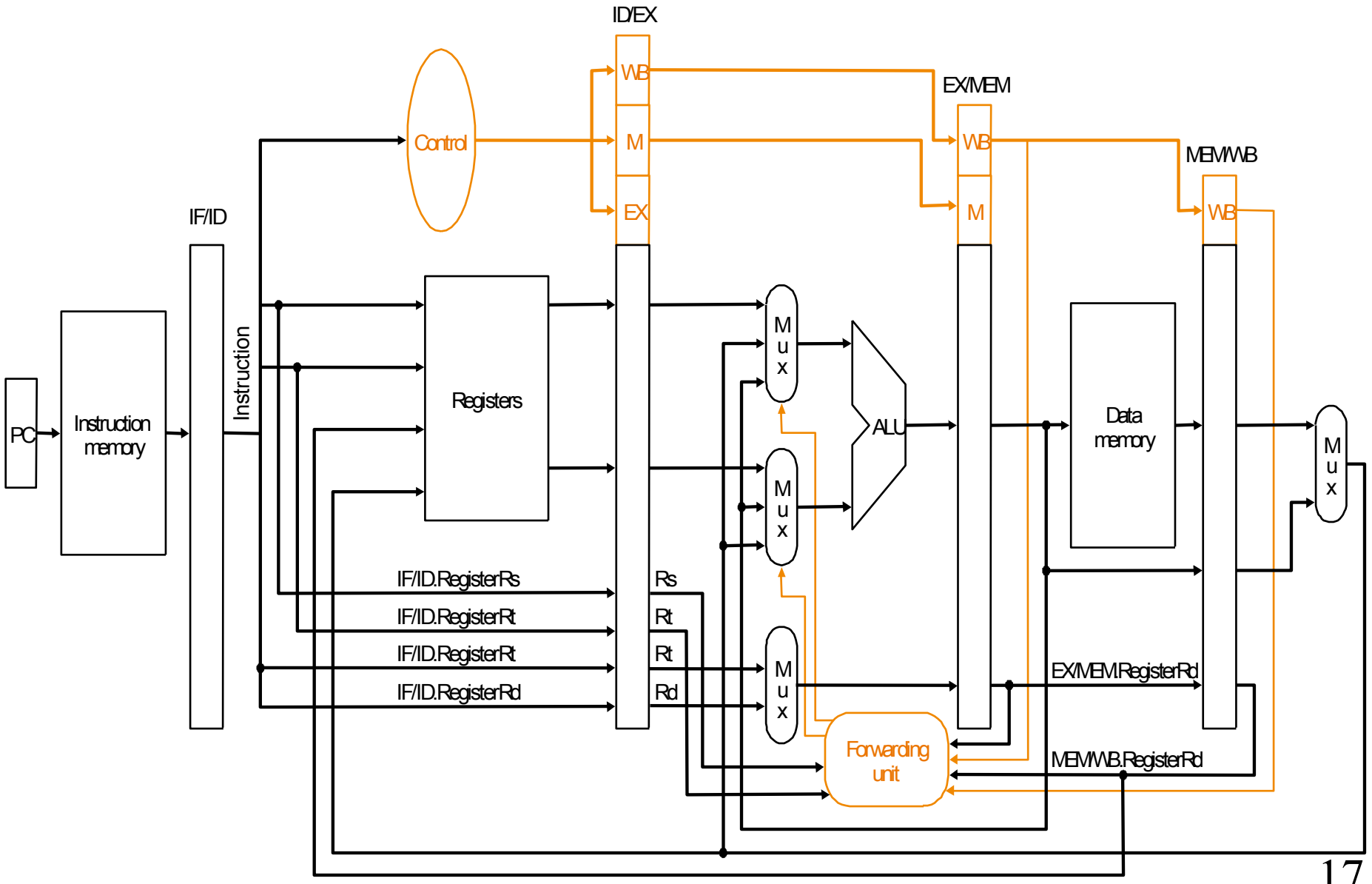
- 結果をレジスタに書き戻されるのを待たないで一時的な結果を直接用いる
  - 同じレジスタに対する読み書き→レジスタフォワーディング
  - ALU フォワーディング

Time (in clock cycles) →

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2 :	10	10	10	10	10/– 20	– 20	– 20	– 20	– 20
Value of EX/MEM :	X	X	X	– 20	X	X	X	X	X
Value of MEM/WB :	X	X	X	X	– 20	X	X	X	X

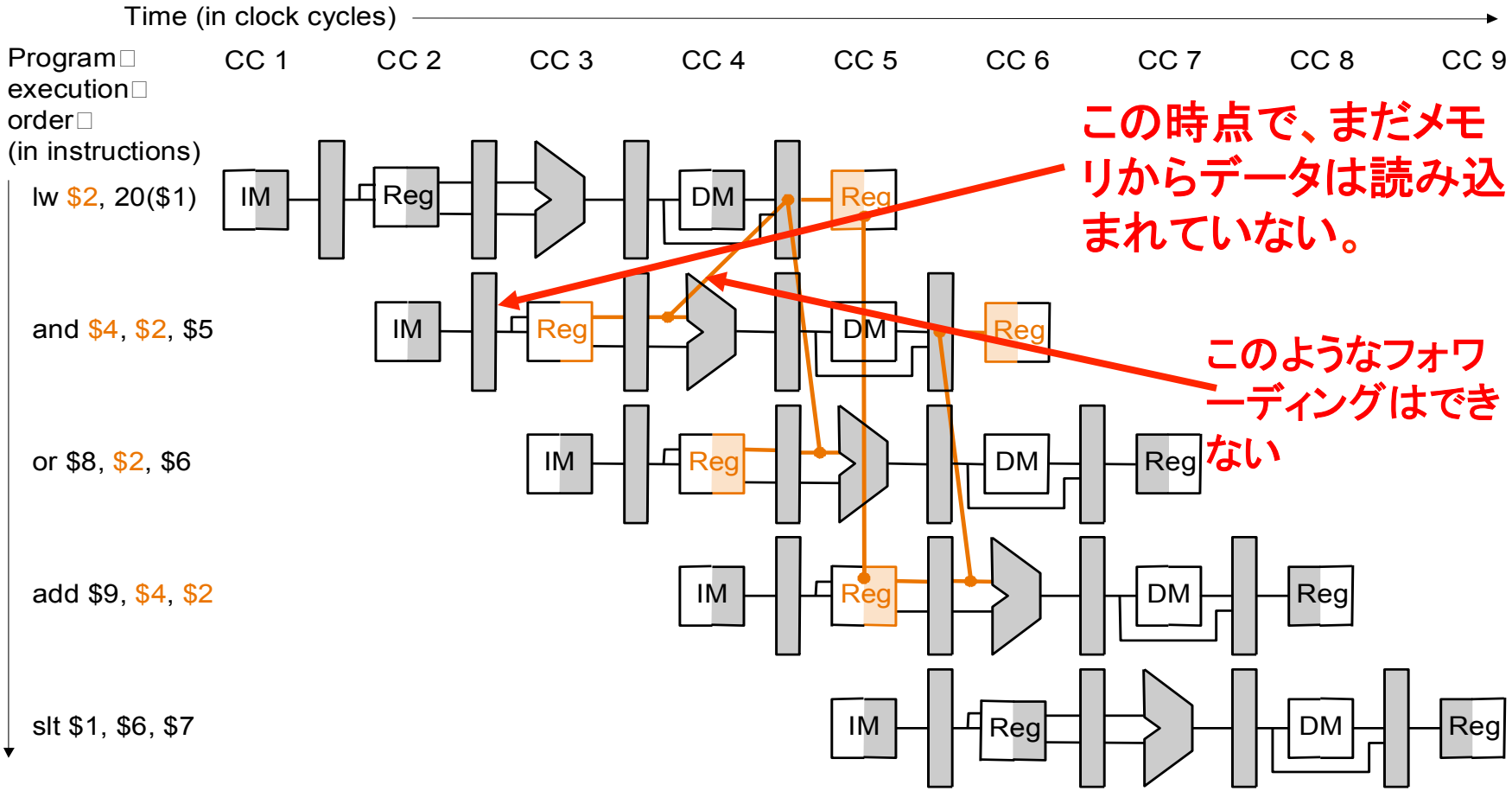


フォワーディングの実装



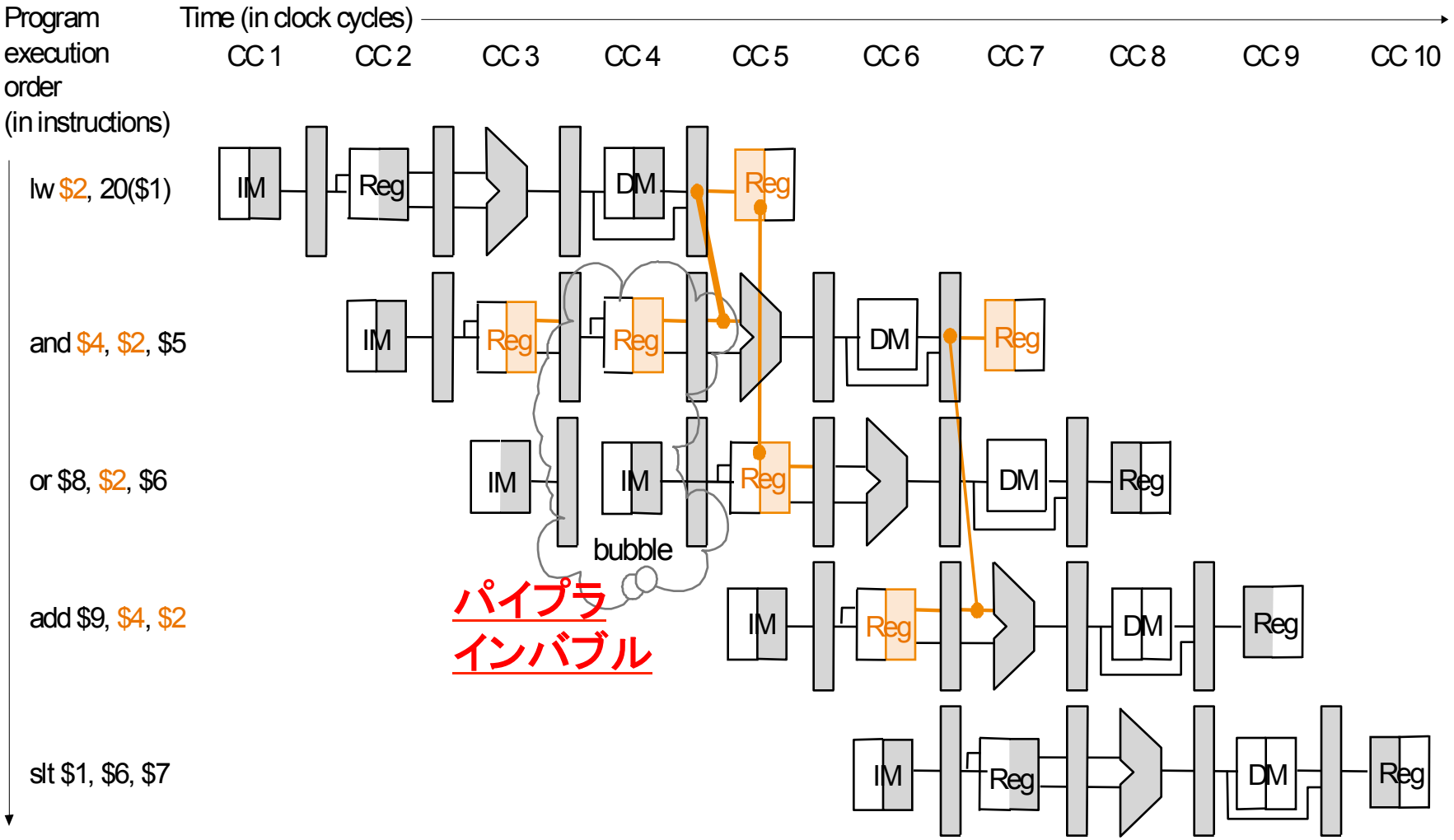
フォワーディングの限界

- フォワーディングを行っても、Load word はハザードを起こす可能性がある:
  - 例: ある命令が、特定のレジスタへのロード命令の直後にその内容を読んで使用しようとする場合→ ロードを待たせる(ストールさせる)必要あり
  - そこで、ロード命令を「ストール」するハザードの検出回路が必要



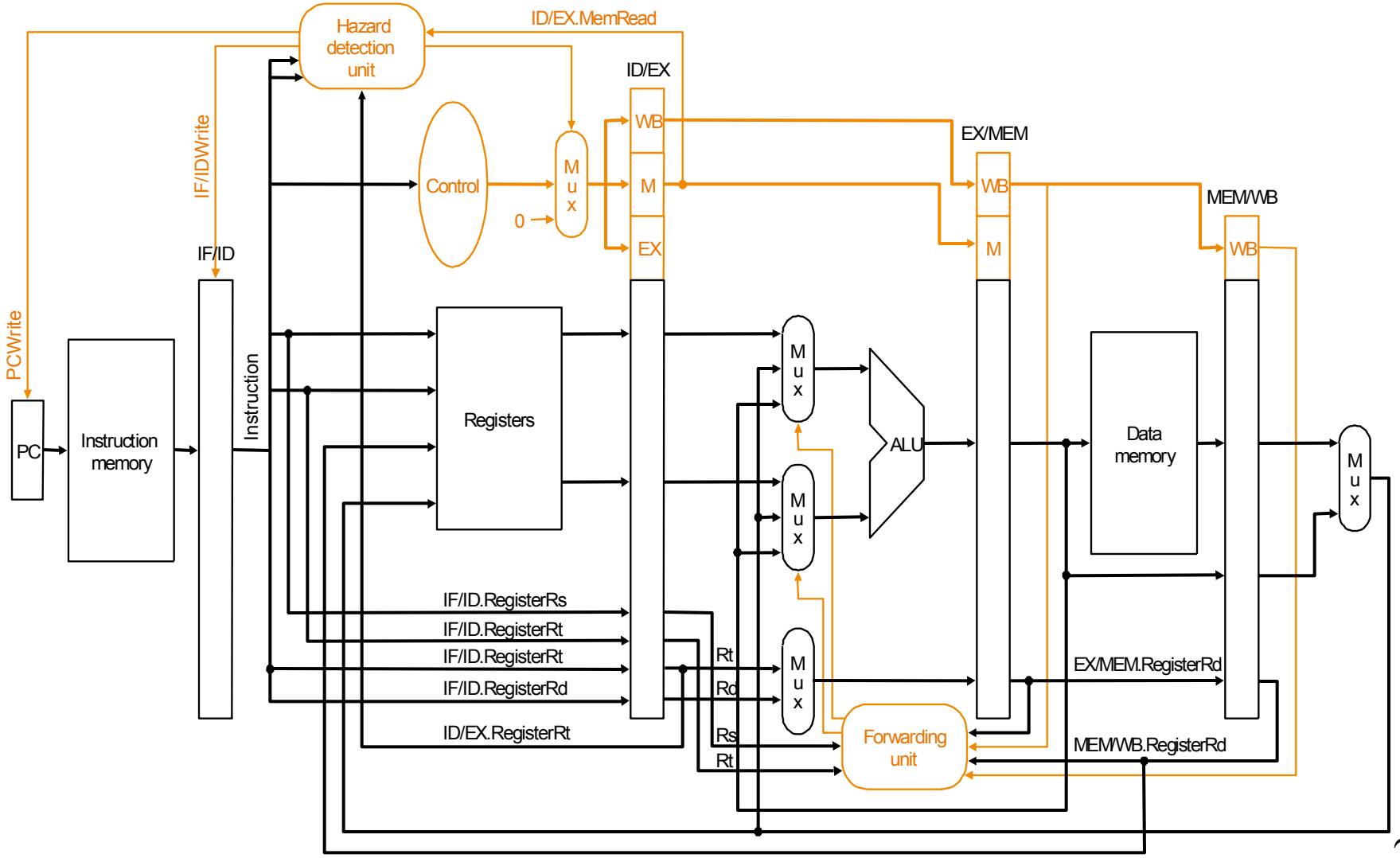
パイプラインストールとパイプラインバブル

- パイプラインを止める(ストール)←同じステージに命令を留めておく



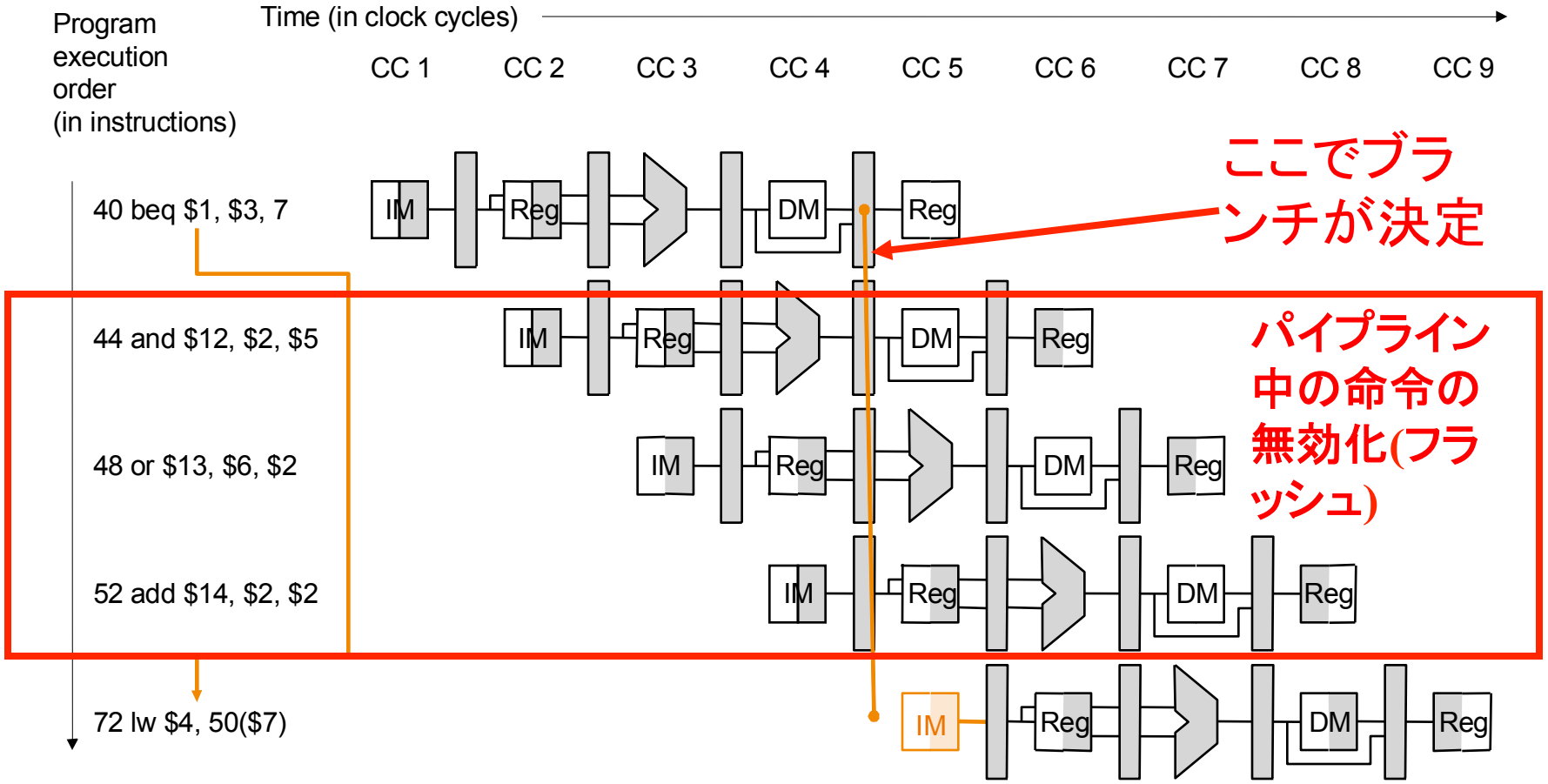
ハザード検出回路によるストール

- 書き込みを行わない命令を先行させることによりストールする



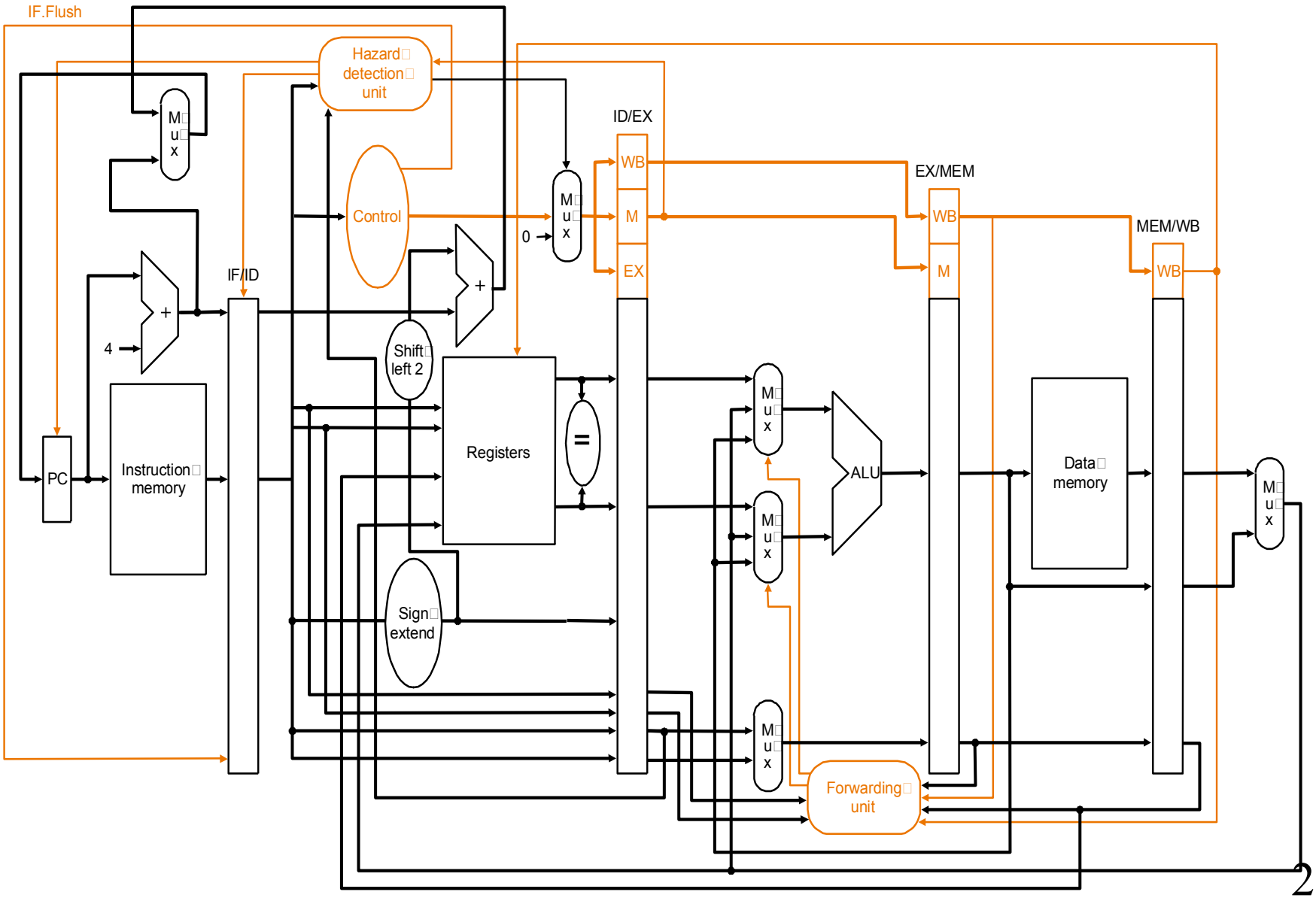
分岐ハザード

- 分岐を決定できるときには、他の命令がパイプライン中に流れている!
  - 途中まで実行した命令を、パイプラインをフラッシュ(flush)することにより、無効化する必要がある



- ここでは、ブランチは「しない」方に予測→外れたらフラッシュ必要

パイプラインのフラッシュ(Pipeline Flush)



さらなる性能の向上

- ストールの減少→コンパイラによる命令の並べ替え(命令スケジューリング) :

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t2, 0($t1)
sw $t0, 4($t1)
```

- 分岐の最適化
  - ディレイスロット (分岐の(メモリ上での)次の命令を必ず実行)
  - ハードウェア分岐予測 (Branch Target Buffer)
  - コンパイラによる分岐の除去

- スーパスカラ: 複数の命令を同一サイクルで複数の演算ユニットで実行
  - DEC Alpha 21264: 9 ステージパイプライン、最大 6 命令同時発行
  - Pentium Pro/II/III: 11-14ステージパイプライン、最大 3 命令同時発行
  - Pentium IV: 20-31 ステージ (スーパーパイプライン)、最大 3 命令同時発行
    - (このあたりが限界➔Core2など最近は短縮する傾向に、なぜ?)

- アーキテクチャの進歩も重要だが、それを生かすコンパイラやランタイムも同様に重要
  - これらなしには、RISCは性能を発揮できない