
第8回
2019/01/11
プロセッサのパイプライン化によるハザード問題

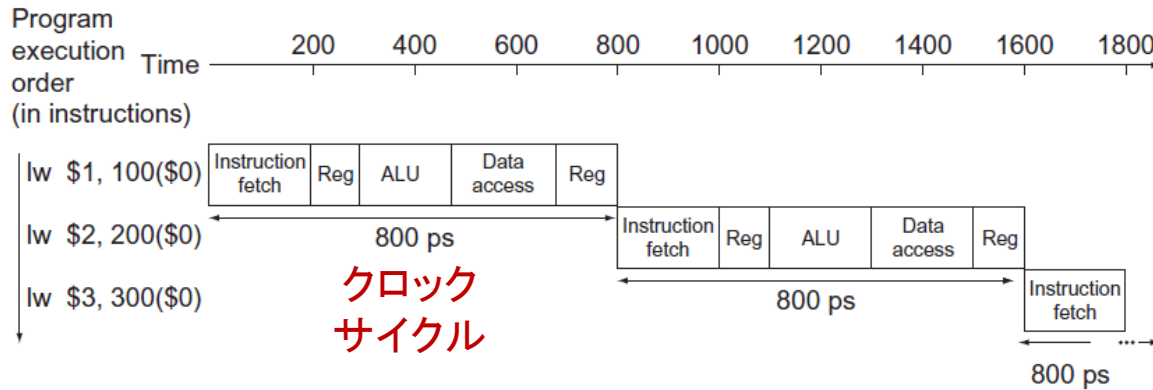
図の多くはPatterson, Hennessy: Computer organization and design 5th editionより引用

パイプラインについての復習(1)

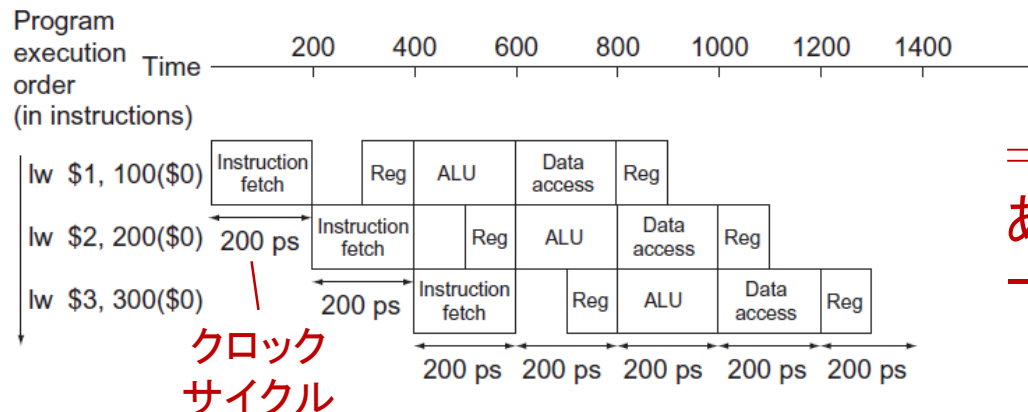
- プロセッサのパイプライン化

- 目的は、クロック周波数を向上させて、全体の性能向上を図るため
- 1命令の実行を複数段階に分けて、複数クロックかけて実行する
 - 授業の例では、IF, ID, EX, MEM, WBの5ステージ
- ある命令と後続命令の実行を時間的に重ねる

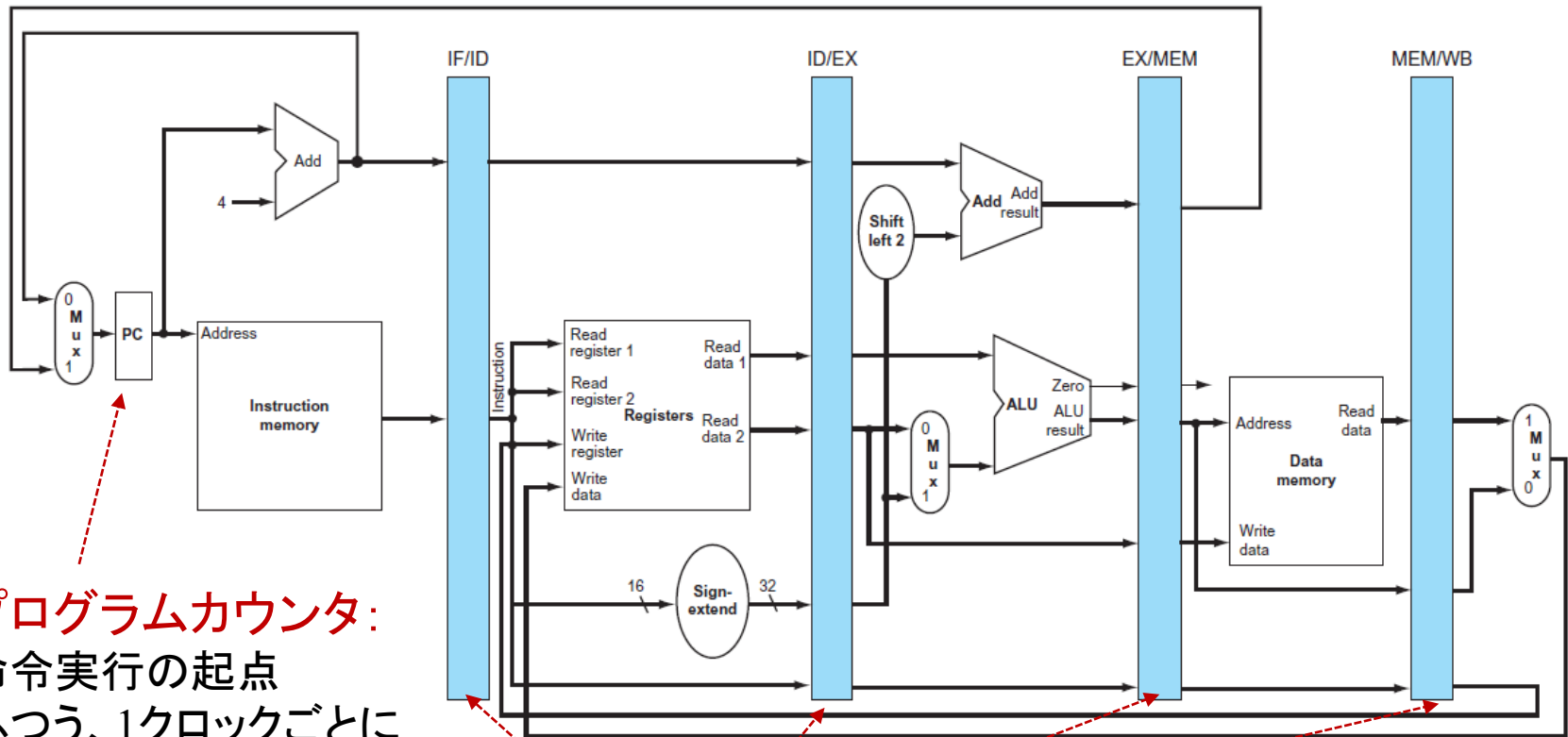
シングル
サイクル



マルチ
サイクル+
パイプライン



パイプラインについての復習(2)



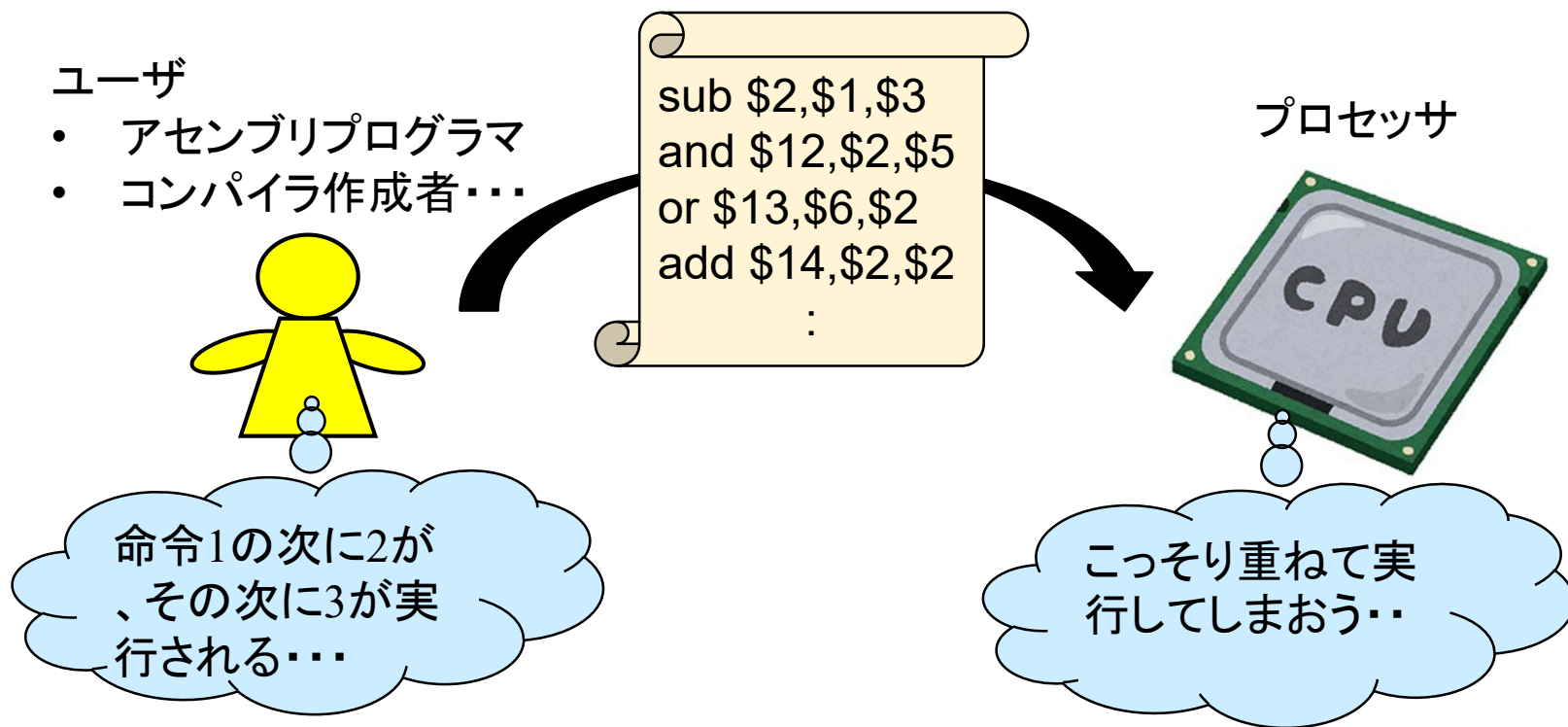
プログラムカウンタ:
命令実行の起点
ふつう、1クロックごとに
4ずつ増えていく

パイプラインレジスタ: ステージとステージの区切りに置かれる
データを一時せきとめて、1クロックごとに左から右へ流す役割

パイプラインが引き起こしうる問題: ハザード(Hazard)

- パイプラインは、現代のプロセッサのほとんどで採用される重要技術
→ しかし、**ハザード**という問題に対応する必要があり、この解決を行うハードウェア技術が必要である

ハザードを大まかにいうと: 以下のような、プロセッサユーザとプロセッサ内部とのギャップによって引き起こされる問題 → 解決は、原則**プロセッサ側の責任**

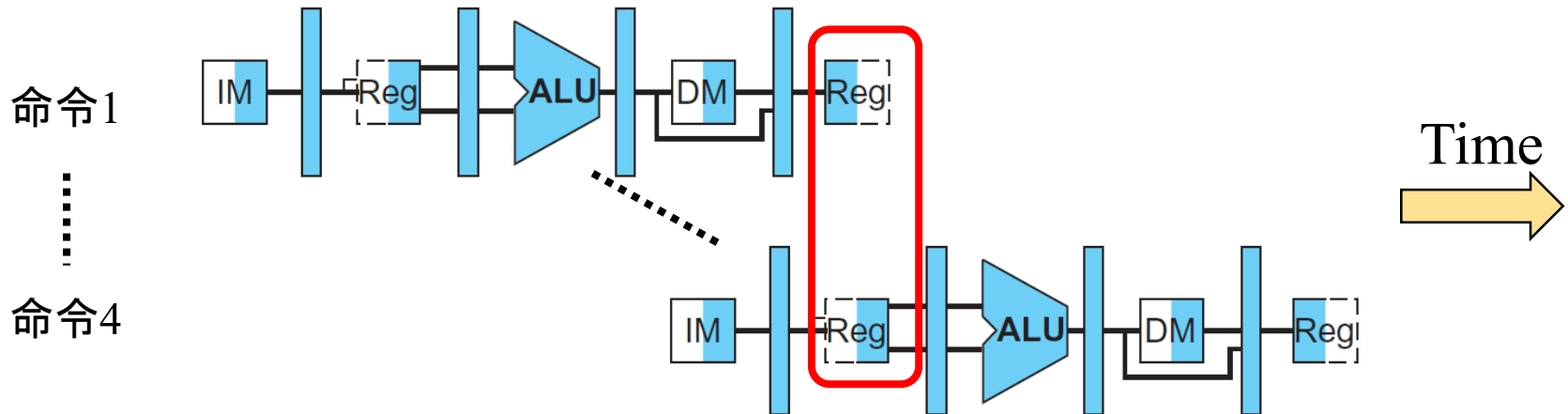


ハザードの種類

- **構造ハザード (Structural Hazard)**
 - 同時に実行される命令の組み合わせに、ハードウェアの構造が対応できない事態
 - 例: 第2ステージIDと、第5ステージWBの両方でレジスタファイルを利用
- **データハザード (Data hazard)**
 - ある命令で必要なデータを、先行命令が終わっていないなどにより、利用できない事態
 - 例: `sub $2,$1,$3 ; and $12,$2,$5`
- **制御ハザード (Control hazard)**
 - 分岐命令により引き起こされる問題
 - 例: `j, beq`命令によりジャンプする必要があるが、そのことが分かった時にはもう後続命令の実行が始まっていた

構造ハザード

- 同時に実行される命令の組み合わせに、ハードウェアの構造が対応できない事態。ある機能ユニットが1命令実行で複数回使われるなど
- 例：第2ステージIDと、第5ステージWBの両方でレジスタファイルを利用



- 解決法の一つ：クロックサイクルを2つに分けて考える
 - 前半ではレジスタへの書き込みのみが、後半では読み込みのみを可能にする

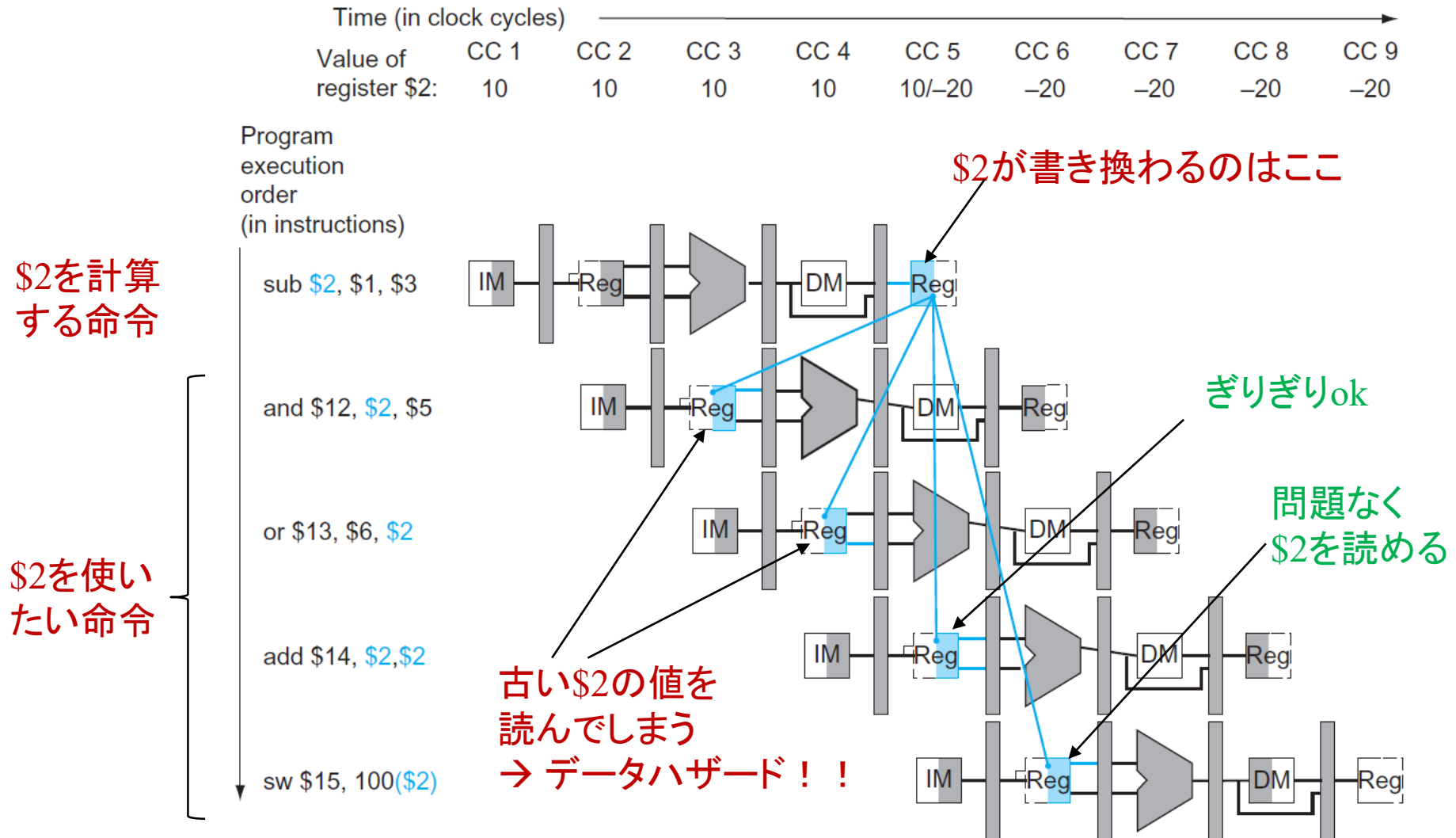
他の例：

- 第1ステージIFと第4ステージMEMの両方でメモリを使用
 - (本講義の例では起こらないが)複数のステージで同じALUを使う設計だと起こりうる
- 原則、各ユニットの構造の工夫などで解決する

データハザード

- ある命令で必要なデータを、先行命令が終わっていないなどにより、利用できない事態
 - 計算結果をレジスタに格納した「直後に」同じレジスタを読みたい、など
- 例：以下のプログラムを実行したい。初期状態で、\$1=0, \$2=10, \$3=20とする
 - sub \$2, \$1, \$3 ← この命令により、\$2は10から-20へ変更
 - and \$12, \$2, \$5
 - or \$13, \$6, \$2
 - add \$14, \$2, \$2
 - sw \$15, 100(\$2)
- パイプラインのないプロセッサであれば、何の心配もなく実行できるが...
- パイプラインのあるプロセッサでは、データハザードが起きうる！

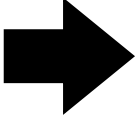
データハザードの起こる理由



非現実的な方法: ソフトウェアによる対応

プロセッサがハザードを解決してくれないとしたら？

- コンパイラ等ががんばって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、適切な数、挿入

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

- これには複数の問題:
 - コンパイラが複雑に&アセンブリプログラミングが極端に困難
 - プロセッサ内部(例えばパイプライン段数)が変更されたら、同じ機械語プログラムが動かないかも
 - 後述の方法より速度が遅い

ハードウェアによるデータハザードへの対応：検知と解決

- データハザードの**検知**

- 命令間に**依存関係(dependency)**があるか調べる必要

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

→ \$2を介して依存関係あり
→ **データハザードの可能性あり！**

```
sub    $2, $1, $3  
and    $12, $4, $5
```

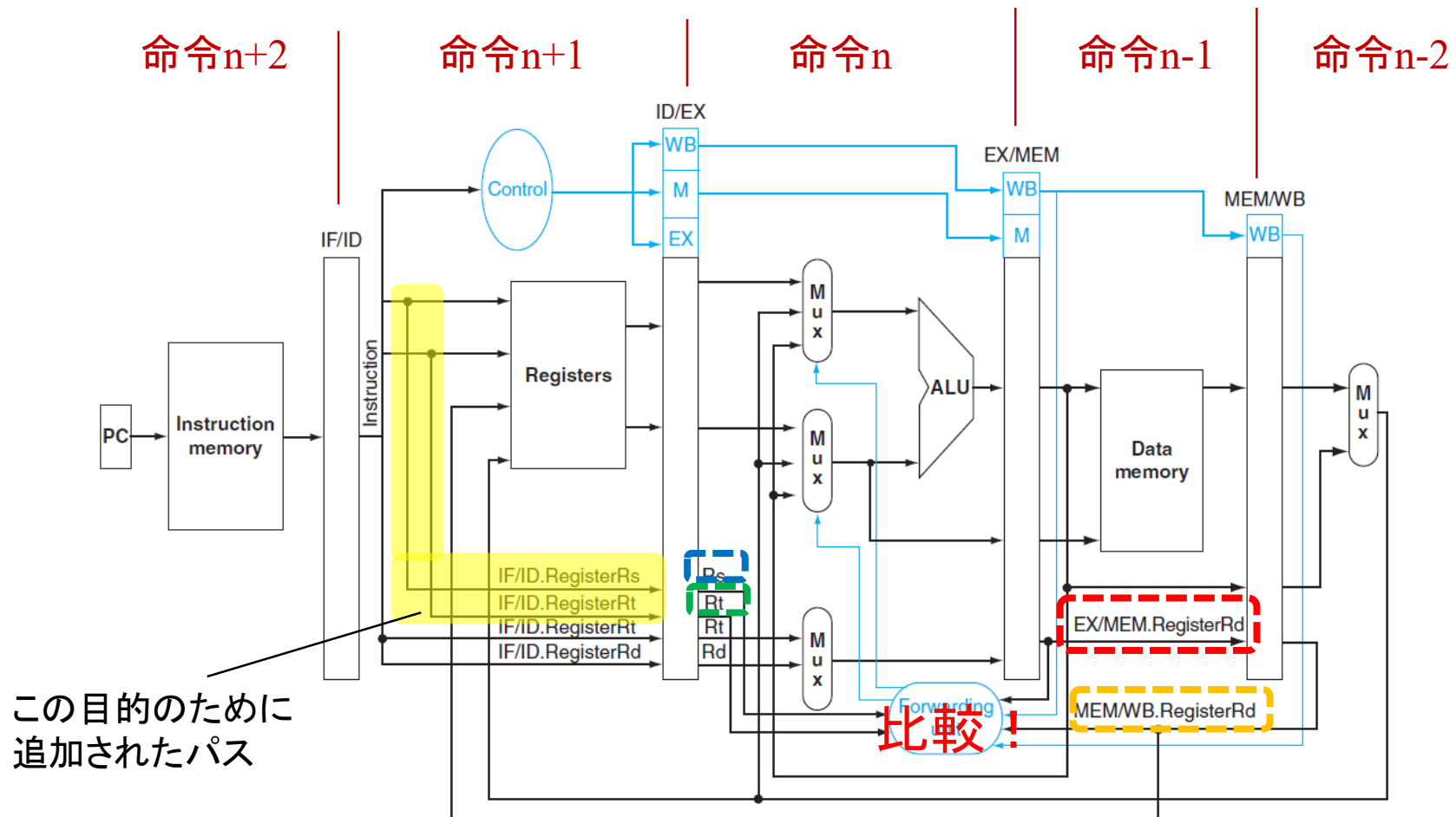
→ 依存関係なし
→ データハザードの可能性なし

- データハザードの**解決**

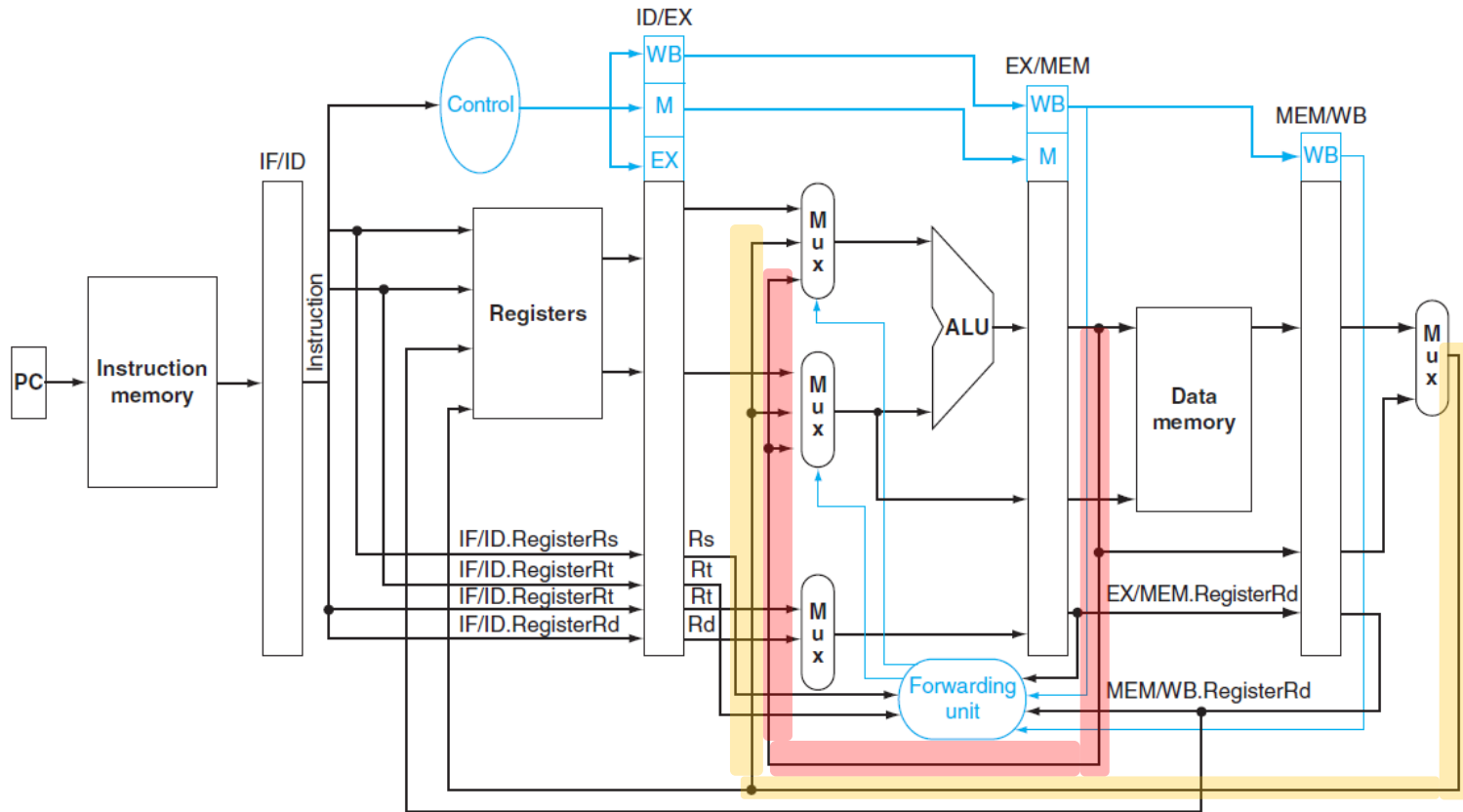
- フォワーディング (forwarding)
- ストール (stall)

データハザードの検知

- 知りたいこと: ある命令nのRsもしくはRtが、下記のいずれかと同じか? (簡単のためR形式命令のみ考慮)
 - 1つ先行した命令n-1のRd
 - 2つ先行した命令n-2のRd

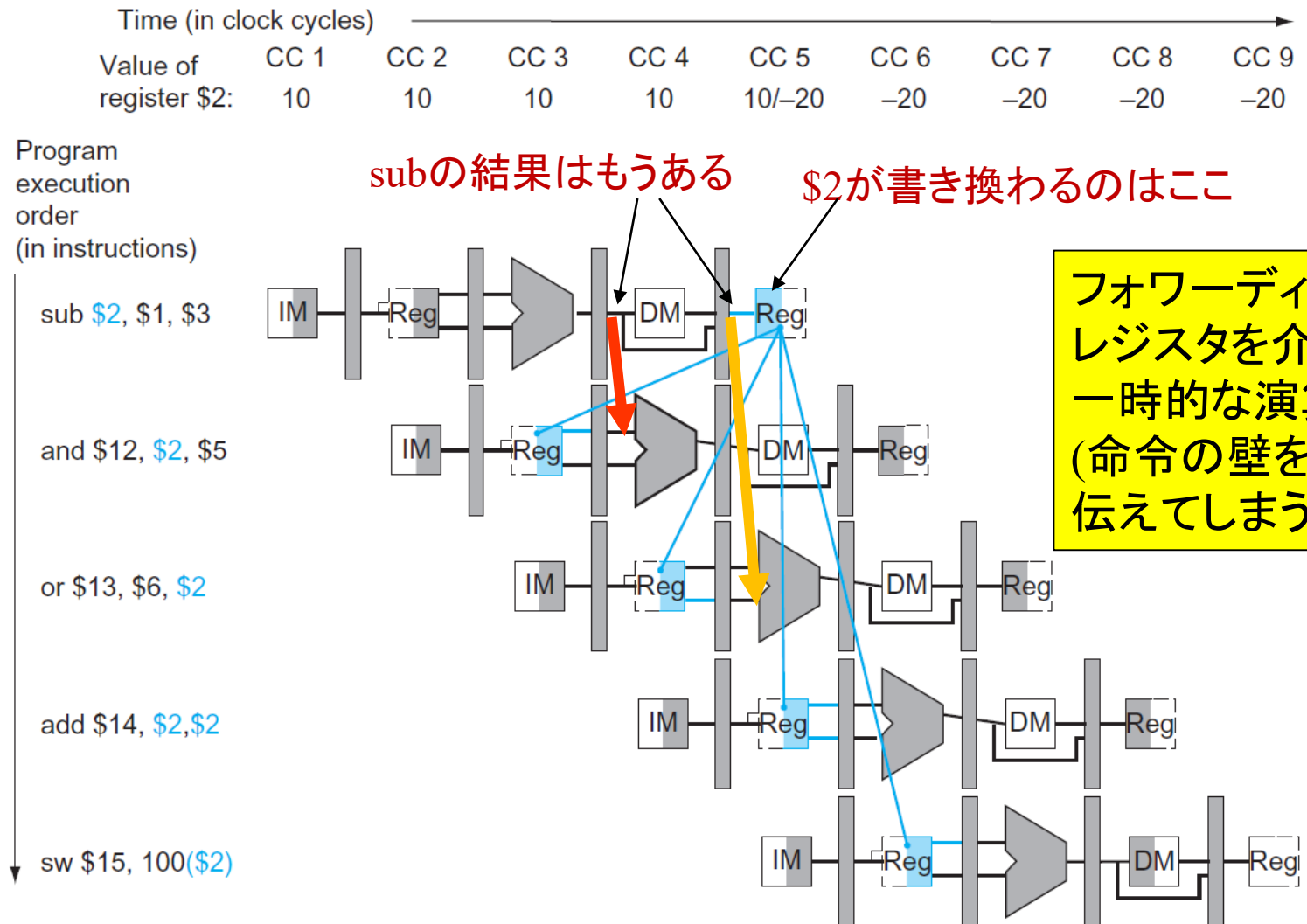


データハザードの解決(1) フォワーディング



- Forwarding unitがハザードを検知したら、
 - 命令n-2のステージ5(WB)のMux出力もしくは、
 - 命令n-1のステージ4(MEM)の入力を、命令nのステージ3(EX)のALU入力に与える

フォワーディングでうまくいく例

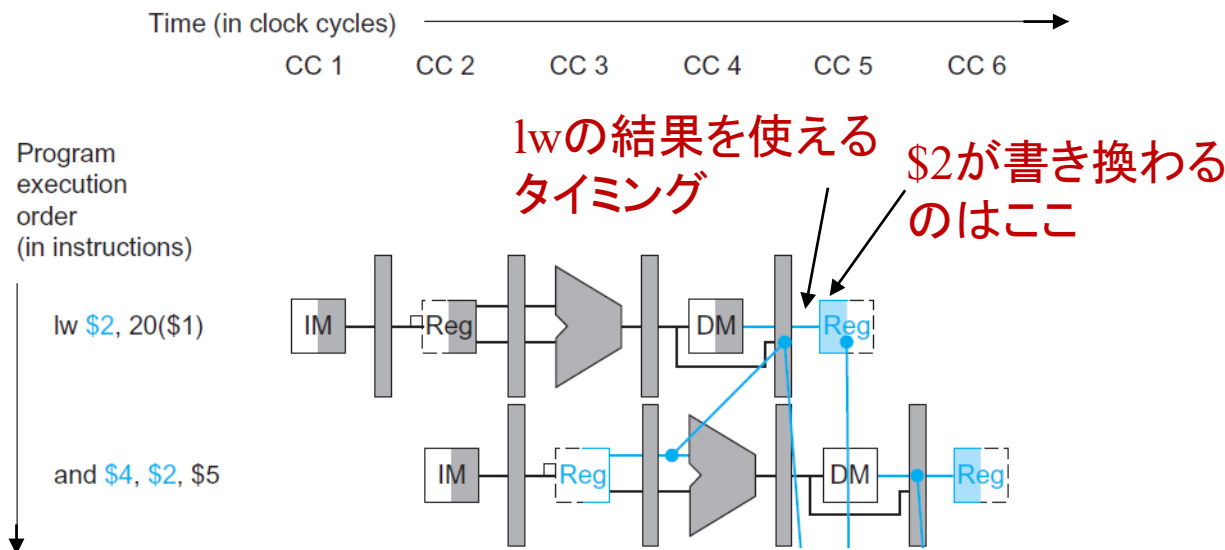


フォワーディングの限界と、解決(2)ストール

- 前の例では、2番目の演算の前までには1番目の演算結果データは出ていた
- より本質的に、データが間に合わないケースでは、フォワーディングでは不足
例: メモリロードの直後に結果を利用

lw \$2, 20(\$1)

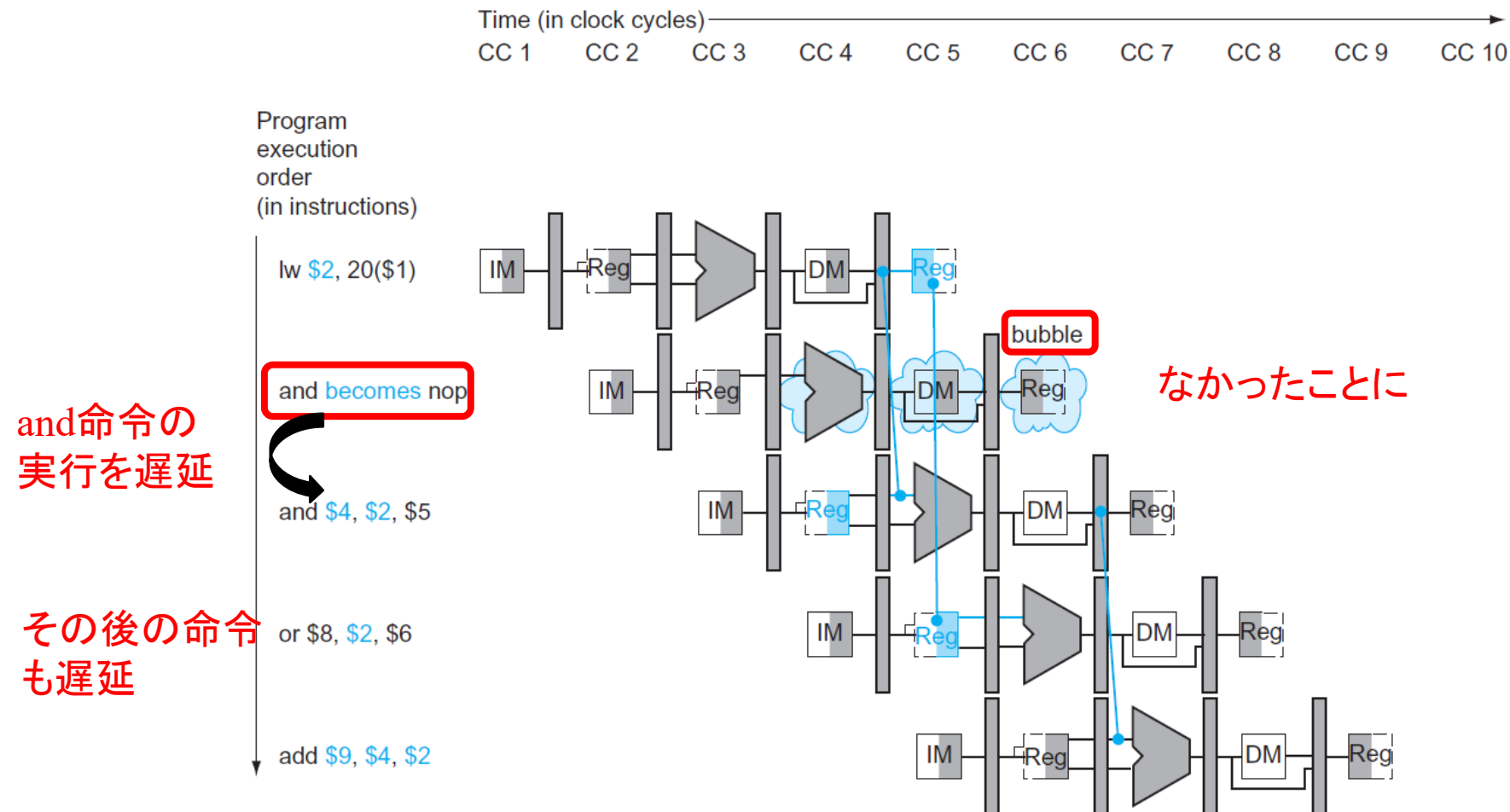
and \$4, \$2, \$5



- 命令1のステージ5から命令2のステージ3にフォワードするのは、時間をさかのぼるので不可能
 - よりパイプライン段数が多い・複雑なプロセッサでは、フォワーディングが使えるケースがさらに減る
- 別の解決法である、**ストール(stall)**が必要
- やりたいことは、後続命令を進行を遅らせること

ストールにより行いたいこと

- 「時間をさかのぼったデータ伝達」を排除するために、後続命令の進行を遅らせる



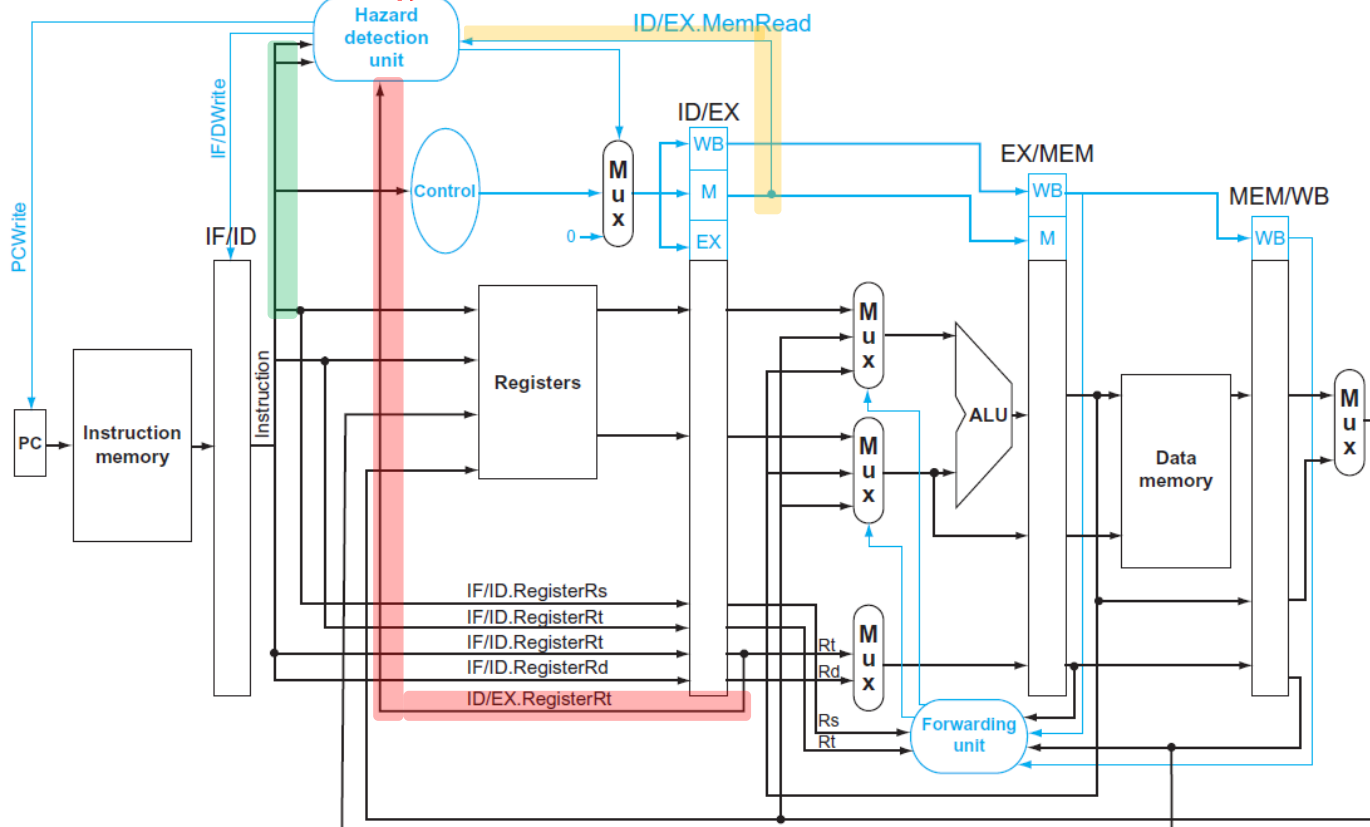
データハザードの検知 (追加)

- 以下の条件を検知するユニットの追加が必要

(命令n-1の種類=="lw" &&

(命令nのRd == 命令n-1のRt || 命令nのRs == 命令n-1のRt))

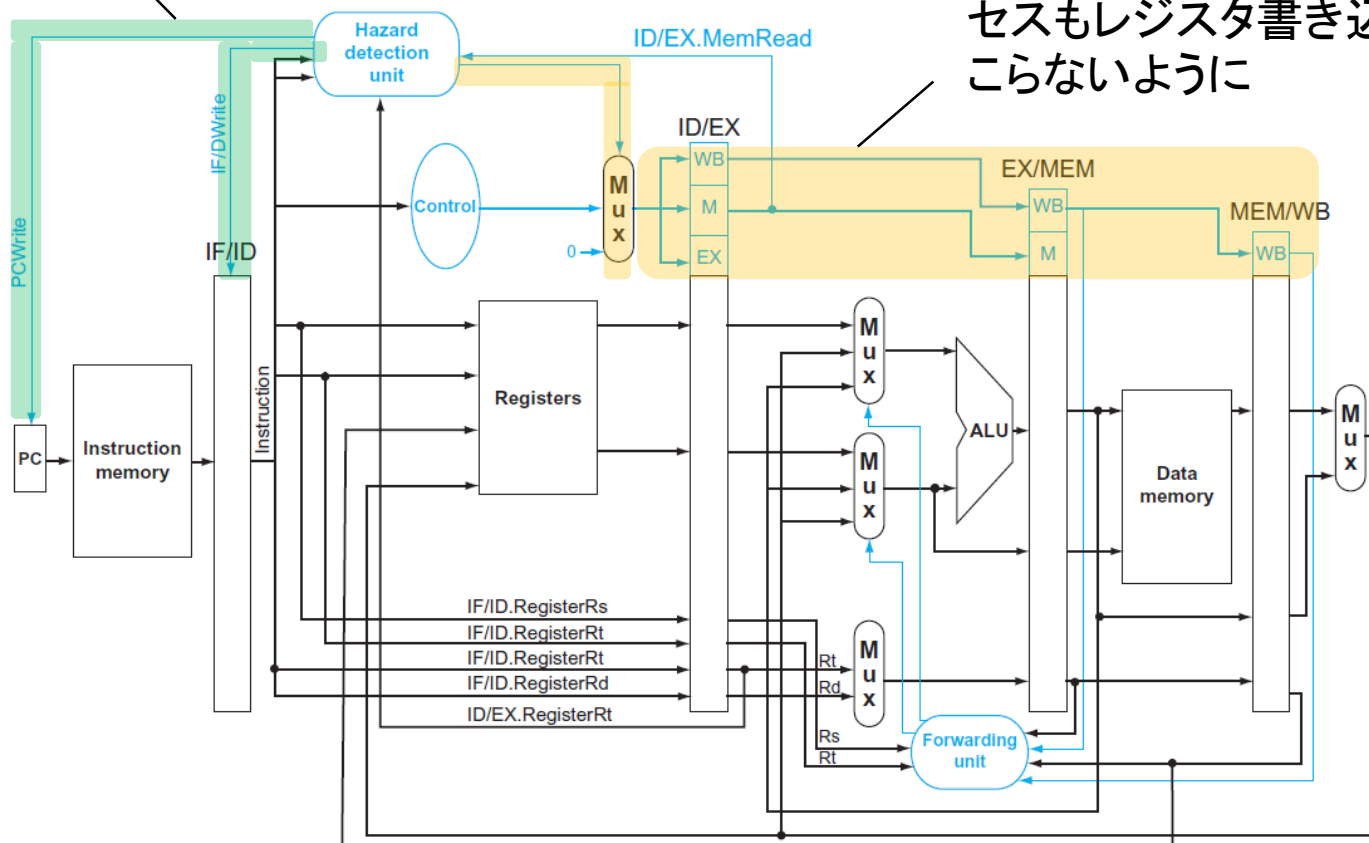
比較！



データハザードの解決(2) ストールの実現

- 後続命令の進行を1clock止める
- シフトレジスタがデータを左から右に流すのを停止
- $PC = PC + 4$ も停止

- この先の命令実行(例では消えるand)を無効にする
- 制御線を0にして、メモリアクセスもレジスタ書き込みも起こらないように



フォワーディング対ストール

- ストールのほうが万能だが、プログラムの進行を遅らせる → 速度性能が下がる
 - フォワーディングのみですむケースならCPI=1は保たれる
 - もし平均で10命令あたり1クロックのストールが起こると、CPI=1.1
- ストールさせるときも、フォワーディングの併用により、クロックの無駄を減らせる

制御ハザード

- 分岐命令により引き起こされる問題
- 分岐命令の復習
 - j L1 L1が示す命令へジャンプ
 - beq \$1, \$3, L2 \$1と\$3が等しいとき、L2へジャンプ。違えば次命令へ

```
sub $10,$4,$8
beq $1,$3,L2
and $12,$2,$5
or $13,$2,$6
add $14,$4,$2
slt $15,$6,$7
:
L2:
lw $4,50($7)
```

やや機械語に
近づけると

命令のアドレス

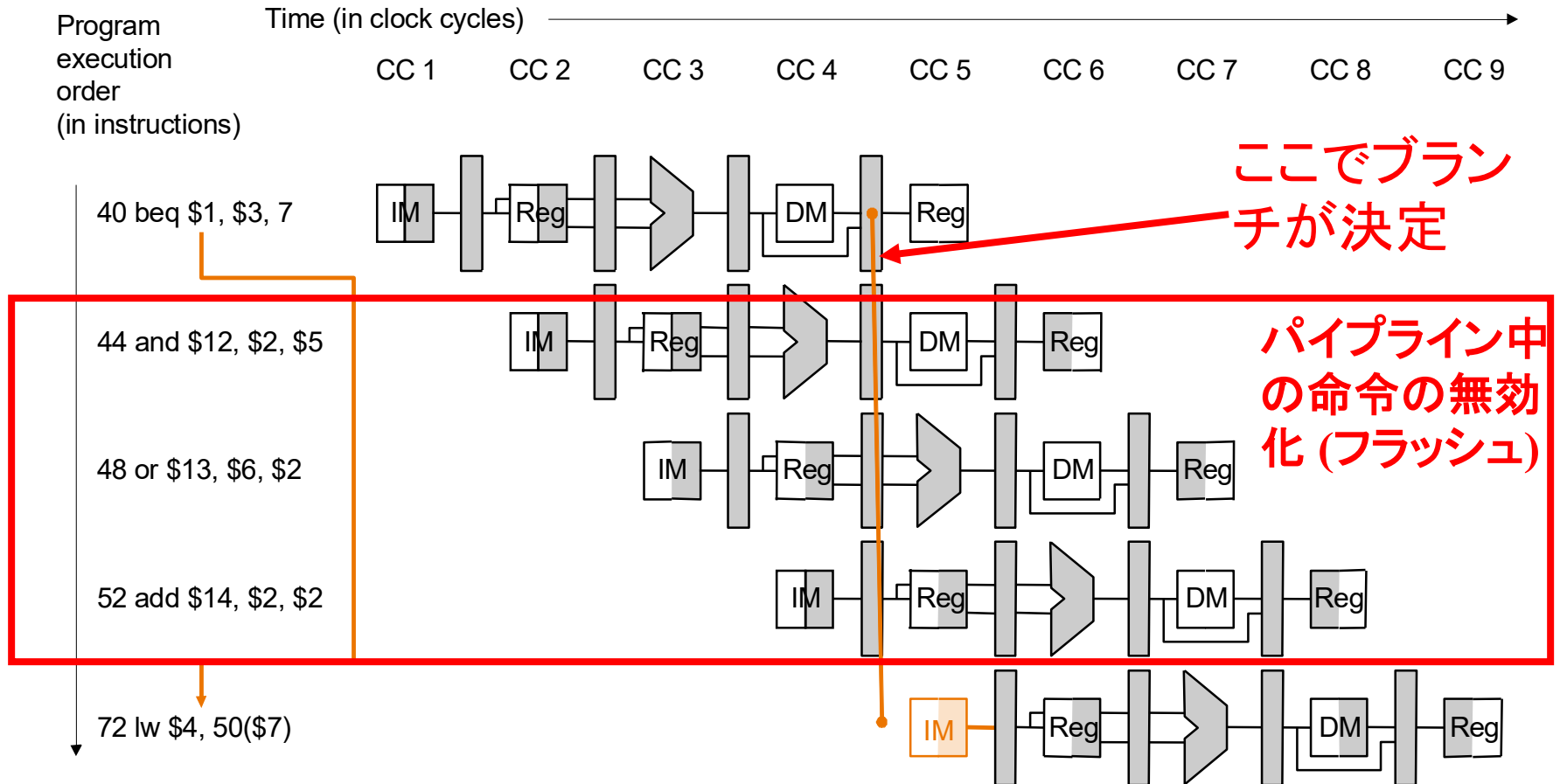
```
36 sub $10,$4,$8
40 beq $1,$3,7
44 and $12,$2,$5
48 or $13,$2,$6
52 add $14,$4,$2
56 slt $15,$6,$7
:
:
72 lw $4,50($7)
```

このbeqの意味

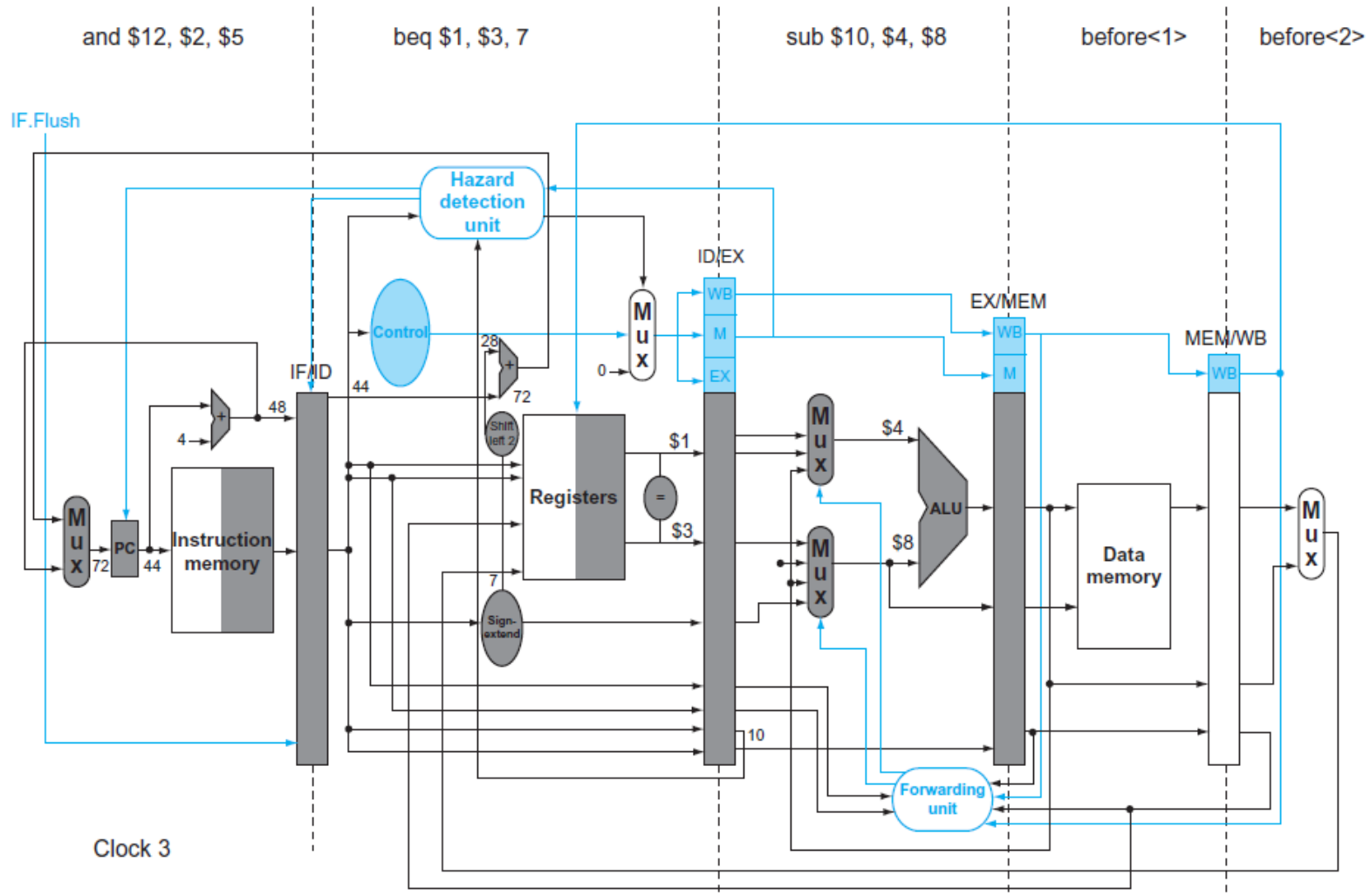
if (\$1==\$3) PC=40+4+(7x4)=72

制御ハザードの例

- 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



制御ハザードの検知と解決のハードウェア



ここまでのまとめ

- ・ ハザード:「命令は順々に実行される」という実行モデルと、プロセッサ内部でパイプラインを行っていることとのギャップにより引き起こされる
 - 構造ハザード
 - データハザード
 - 制御ハザード

それぞれ、ハードウェアの回路追加によって解決策がある

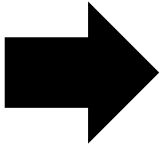
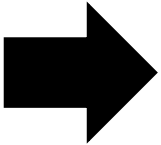
Intelプロセッサのパイプライン

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/ Speculation	Cores/ Chlp	Power	
Intel 486	1989	25 MHz	5	1	No	1	5	W
Intel Pentium	1993	66 MHz	5	2	No	1	10	W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29	W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75	W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103	W
Intel Core	2006	2930 MHz	14	4	Yes	2	75	W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	Yes	1	87	W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	Yes	8	77	W

2005年までは、クロックの高速化 & パイプラインステージ数が増大
それ以降は、クロックやや減 & ステージ数減。代わりにコア数増

ソフトウェアによる工夫

- ハザードの解決はハードウェアで行われるので、ソフトウェアはパイプラインの存在を知らなくても正しく動く
- しかし、パイプラインの存在を知っていると性能を向上可能(かもしれない)
 - ソフトウェアパイプライン (software pipelining): 近年のコンパイラの多くが採用

a=b+e; c=b+f;		lw \$t1, 0(\$sp)		lw \$t1, 0(\$sp)
		lw \$t2, 4(\$sp)		lw \$t2, 4(\$sp)
		add \$t3, \$t1, \$t2		<u>lw \$t4, 8(\$sp)</u>
		sw \$t3, 12(\$sp)		add \$t3, \$t1, \$t2
		lw \$t4, 8(\$sp)		sw \$t3, 12(\$sp)
		add \$t5, \$t1, \$t4		add \$t5, \$t1, \$t4
		sw \$t5, 16(\$sp)		sw \$t5, 16(\$sp)

命令の入れ替えだけで何が変わる? → ストールの削減

実際のプロセッサでの技術

これまで学んだ技術(パイプライン、ハザード解決)を基本にしつつ、さらなる高速化技術が満載！

- 実際のプロセッサでは、各命令実行のクロック数は違うことにも注意
- スーパースカラ (superscalar): 複数命令の動的同時発行
 - ※ 発行(issue)=命令の実行開始
- アウトオブオーダー(out-of-order)実行
 - 後続命令が先行命令を追い越しうる
- 分岐予測(branch speculation)
 - ある条件分岐(MIPSのbeq)の条件は、あたりやすいか外れやすいかを記憶
 - 先ほどの制御ハザードの解決例では、「おそらく分岐しない」を仮定していたと言える
- マイクロオペレーション(micro-operation)変換
 - IntelのようなCISC ISAのプロセッサで用いられる
 - 1機械語命令 ⇒ 複数の単純な(RISC的な)内部命令へ変換してから実行

なお、以上の技術は「1コア内」の話。複数コアについての話は次々回以降