

# 先進計算機構成論 04

---

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](mailto:shioya@ci.i.u-tokyo.ac.jp)

# 質問とか回答など

- GPUとCPUの違いがわかりました。これまでなんとなく、Pytorchなどでviewを使っていたのですが、その意味が分かりました。
- 今まで制御部分の大きさの観点からCPU、GPUを見たことがなかつたので非常に新鮮だった。

# 質問とか回答など

- cpuの性能が、上がりにくくなつたからマルチコア化する技術に対する研究が進んでいるみたいに聞いたのですが、コア数ってただ増やすだけじゃ性能上がらないんですよね？

# 質問とか回答など

- わからなかつたこと（質問）：FPGAの専用DSPを使えば効率がまだCPU・GPUより低いでしょうか？
- FPGAのメリットは、命令制御に必要な資源が不要なことと授業で紹介されていましたが、それによってどのような効果が得られるのでしょうか。例えば、処理速度が向上したりするのでしょうか。

# 質問とか回答など

- クロック信号の反転が多くなるほどに消費電力に負荷がかかることがわかった。それに関して今後根本的なbreak throughは起こりうるのだろうか。量子コンピュータではどうなっているのだろうか。
- 講義中にトランジスタのサイズを小さくするにも限界があるという話がありましたが、そういう意味ではCPUの性能(動作周波数)は将来的に頭打ちになるのでしょうか。

# 質問とか回答など

- 現在のトランジスタのサイズ程度ならばトンネル効果などの影響は全く考慮しなくていい程度なのでしょうか？それともある程度現段階で考慮されていて、より量子化学的影響が大きくなつていっても対応できる見通しが立っているのでしょうか

# 質問とか回答など

- ムーアの法則はいつ機能しなくなると予想されますか。また、TSMCがエネルギー効率は2年で倍増すると唱えているそうですが、ムーアの法則が機能しなくなった後はどのようにして発展していくと思われますか。

# 質問とか回答など

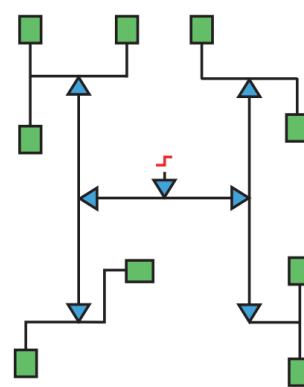
- 今回の講義では、便宜上電気回路（抵抗、コンデンサ）で表していますが、仲間はずれにされているコイルのリアクタンスや電磁誘導のインダクタンスやを応用して改良したりする研究はあるのでしょうか？  
(コイルは速度が遅いので反応速度は下がりますが、上手く使ったら消費電力を抑える  
(例えば毎回発振する回路であるクロックの電流を回収する、など)

# 質問とか回答など

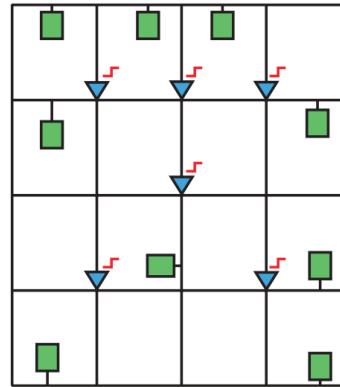
- skewを防ぐために、全FF-clock間の配線と同じ長さにしていると言っていましたが、それ以外にskewを防ぐ方法というのはないのでしょうか。
- VLSIの配線方法や配線アルゴリズムについてH-tree以外のものにも興味がわきました

図は以下より：

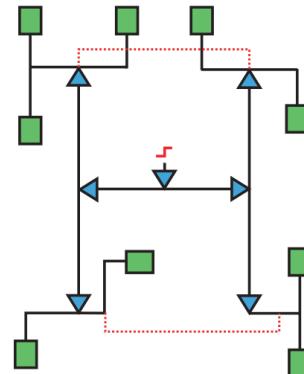
KIM, Youngchan; KIM, Taewhan. Algorithm for synthesis and exploration of clock spines. In: 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC). IEEE, 2017. p. 263-268.



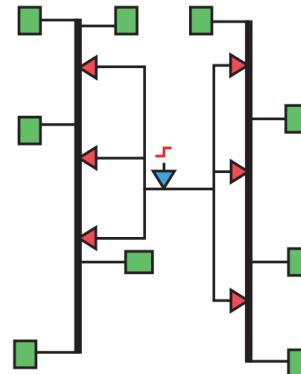
(a) Clock tree structure



(b) Clock mesh structure



(c) Clock tree structure with links



(d) Clock spine structure

- ◊ skew を無くすためには、クロックネットワーク全体で電圧のズレがなければ良い
- ◊ clock grid : メッシュ状の配線を使う
  - ノード同士を密に縦横で繋いでやれば電位差が生まれにくい
- ◊ clock spine : 太い配線を幹にして、そこから配る
  - 太い線の中では差が生まれにくいので、そこから配る

Fig. 1. Two extreme clock network structures are shown in (a) and (b). Two intermediate clock network structures are shown in (c) and (d). Clock spine structure in (d) delivers clock signal from clock source to clock sinks through top-level clock tree, clock spines, and stubs. At least two spine buffers have to be placed on each clock spine to maintain the tolerance to clock skew variability.

# 質問とか回答など

- IntelのCPUが銅線の代わりにコバルト線か何かを使っているために、パフォーマンスが良くないという噂を聞いたことがありますか、具体的にはどういうことなのでしょう？今回仰っていた電圧を下げる話が関わってくるのでしょうか？
- クロックの消費電力が大きいということが言われていましたが、クロックの使用を減らすということも考えられているのでしょうか。減らせるとするとどういったやり方になるのでしょうか。

## 質問とか回答など

- 自宅の空冷式PCではクロック周波数は3.4GHzだったが、水冷式などのより強力な冷却器を導入すれば、さらにクロック周波数を上げられるのだろうか？

<https://hwbot.org/> より：

## CPU Frequency xCPU Ranking

Member ranking		Team ranking								
	Score	User	Frequency	Hardware	Cooling	HW	GL			
1.	8722.78 MHz	The Stilt	8722.8 MHz	AMD FX-8370	LN2	35pts	161pts	55	63	13
2.	8709 MHz	AndreYang	8709 MHz	AMD FX-8150	LN2	52pts	140pts	40	31	16
3.	8659.64 MHz	Smoke	8659.6 MHz	AMD FX-8370	LN2	28pts	125pts	10	18	13
4.	8615.39 MHz	slamms	8615.4 MHz	AMD FX-8350	LN2	54pts	115pts	13	16	16
5.	8543.71 MHz	wytiwx	8543.7 MHz	Intel Celeron D 352	LN2	70pts	107pts	22	23	23
6.	8532.17 MHz	BenchBros	8532.2 MHz	AMD FX-8320	LN2	52pts	100pts	14	8	14
7.	8502.1 MHz	NickShih	8502.1 MHz	AMD FX-8350	LN2	51pts	94pts	10	7	10
8.	8470.74 MHz	Hicookie	8470.7 MHz	AMD FX-8350	LN2	48pts	89pts	15	14	14
9.	8448.78 MHz	_12_	8448 MHz	AMD FX-8320	LN2	46pts	85pts	6	8	8
10.	8431.91 MHz	CherV	8431.9 MHz	AMD FX-8350	LN2	47pts	84pts	6	0	0
11.	8429.38 MHz	macci	8429.4 MHz	AMD FX-8150	LHe	46pts	83pts	102	37	37
12.	8407.06 MHz	Atheros	8407.1 MHz	AMD FX-8320	LN2	43pts	82pts	4	3	3
13.	8406.34 MHz	Wizerty	8406.3 MHz	AMD FX-8150	LN2	43pts	81pts	15	6	6
14.	8370.93 MHz	alvinkenzo	8370.9 MHz	AMD FX-8350	LN2	45pts	80pts	9	3	3
15.	8366.83 MHz	The Silver	8366.8 MHz	AMD FX-8370	LN2	23pts	79pts	6	12	12
16.	8362.21 MHz	ivanqu0208	8362.2 MHz	Intel Celeron D 347	LN2	60pts	78pts	9	13	13
17.	8348.43 MHz	Ananerbe	8348.4 MHz	AMD FX-8350	LN2	44pts	77pts	6	5	5

# 質問とか回答など

- 浮動小数点数演算は、その複雑性によって、FPGA上で実装した際のLUTの消費量が激しいのだと思いますが、精度を多少捨てて固定小数点を利用する方式にした場合はある程度LUTを節約できたりするのでしょうか

# 質問とか回答など

- FPGAの話は初めて聞いたのでとても面白かったです。実際どのようにところで活用されているか気になりました。
- FPGAを初めて知った。真理値表の出力をあらかじめ持つておいて入力によって選択するという発想があるのか。

# 質問とか回答など

- コンピュータの性能が物理的な制約にぶち当たってムーア則が落ち着いてきながらも、GPGPU や TPU が現れたり、M1 チップが話題になったり、構成的な部分での変化によってコンピュータは今後も進化していくのかなあとと思いました。

# 質問とか回答など

- 余談の内容が面白かったです。「ムーアの法則は最近は成り立たない」という話を耳にしたことがありましたが、その理由がわかって良かったです。

# 質問とか回答など

- 12世代IntelCPUは高速なコアと低速なコアを両方搭載していますが、これはエネルギー密度問題への対処なのでしょうか.

# 質問とか回答など

- I also encountered the concept of pipeline in the process of learning R language, which can help me understand the logic of pipelining in this course.

# 命令パイプライン

---

# 今日の内容：命令パイプライン

1. シングル・サイクル・プロセッサの動作
  - ◊ パイプライン化を前提とした構造のものを使って復習
  - ◊ 全ての命令の処理が 1 サイクルで完結
2. 上記のパイプライン化
  1. 具体的にどうパイプライン化するか
3. ハザード

# 導入：工場のラインを考える



- ベルトコンベアのラインの上を製品が流れていく
  - ◊ 4人の人が、それぞれの工程の作業をおこなって完成
  
- 上のように1つしか製品をながさないと、
  - ◊ 各人は他の人が作業している間はヒマ

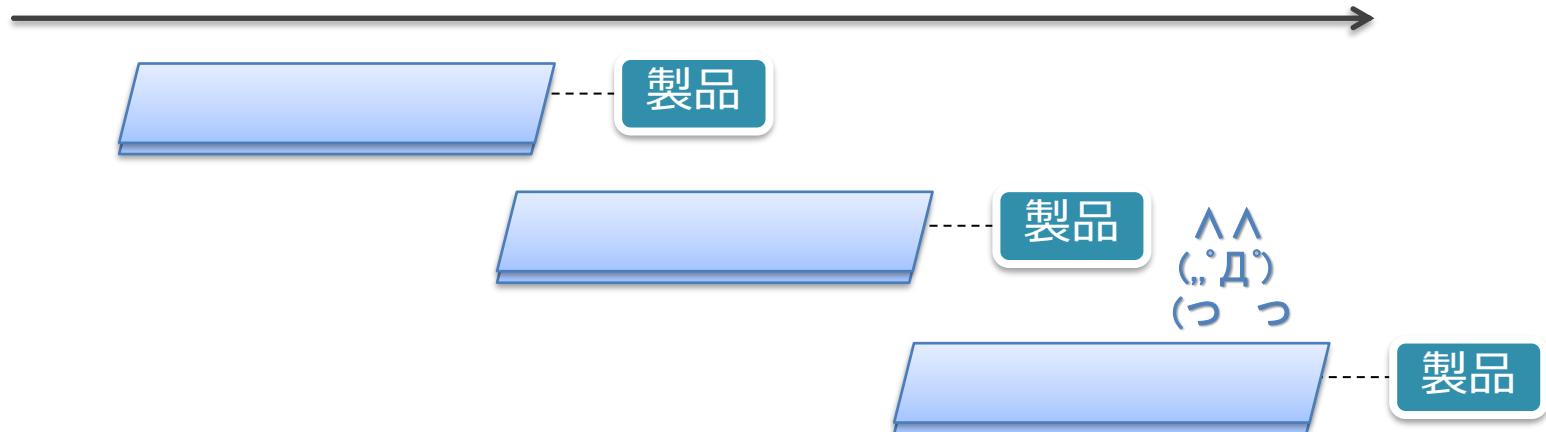
# 導入：工場のラインを考える



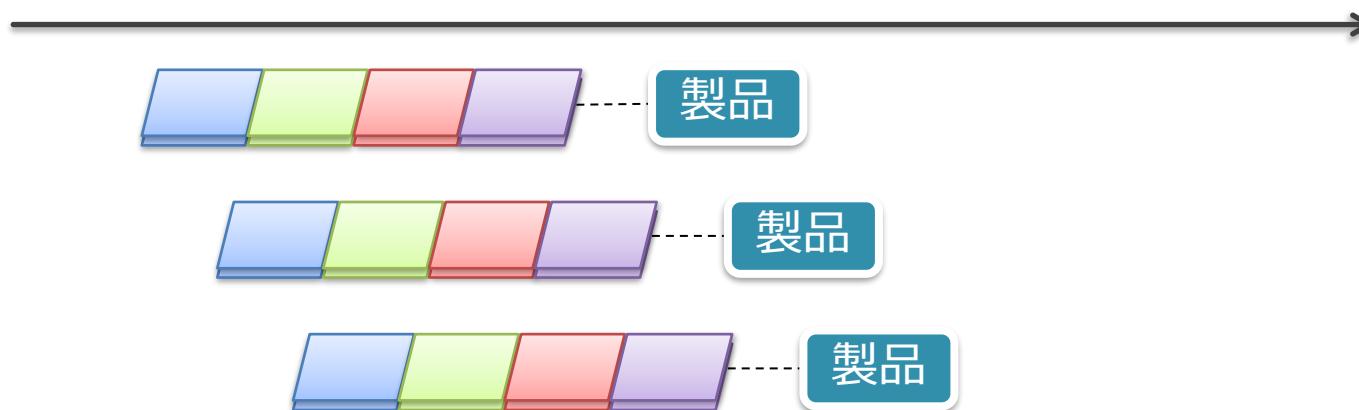
- 実際の工場：複数の製品を同時に流す
  - ◊ 各工程を並列して処理することによりスループットを向上
  - ◊ さっきの4倍の速度で製品ができあがっていく
- これが 命令パイプライン

# パイプライン化による性能向上

パイプライン化しない場合



パイプライン化した場合



# シングル・サイクル・プロセッサの動作

---

# もくじ

## 1. シングル・サイクル・プロセッサの動作

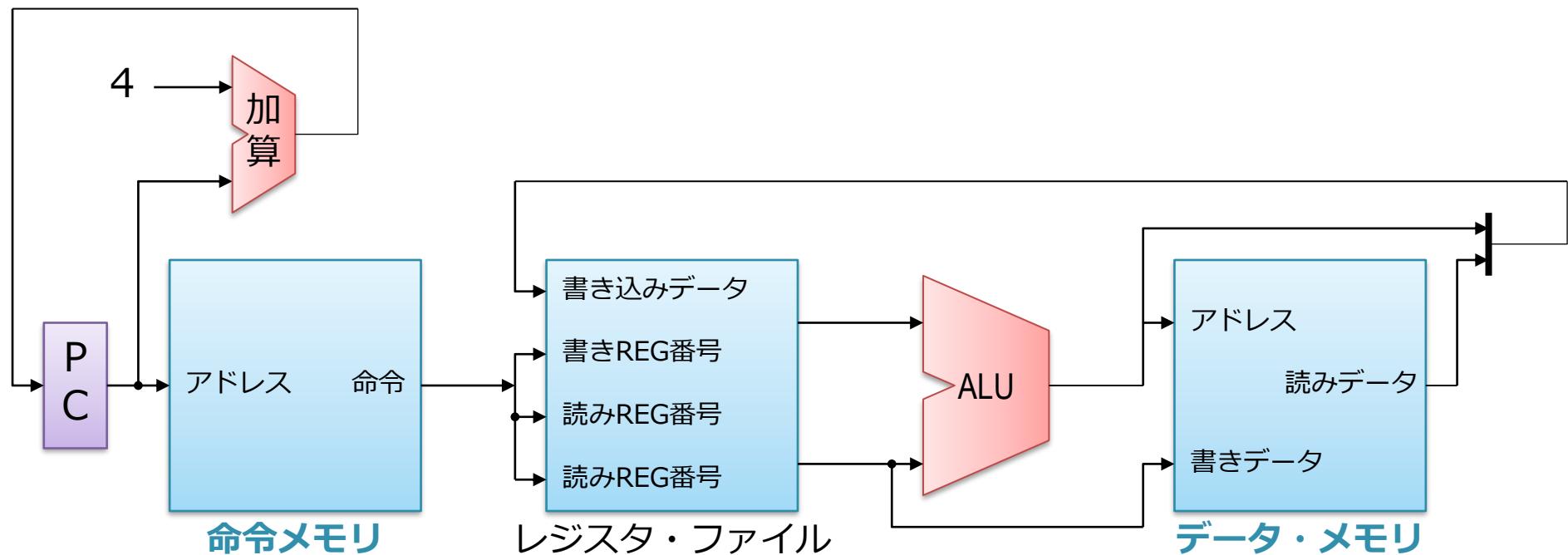
- ◊ パイプライン化を前提とした構造のものを使って復習
- ◊ 全ての命令の処理が 1 サイクルで完結

## 2. 上記のパイプライン化

1. 具体的にどうパイプライン化するか

## 3. ハザード

# ベースとなるシングル・サイクル・プロセッサ



- 以前説明したものとの違い：
  - ◊ メモリが命令メモリとデータメモリに別れている
  - ◊ 算術 & 論理演算, ロード, ストアのみを実行可能
  - 分岐とジャンプは, 簡単のために今は考えない

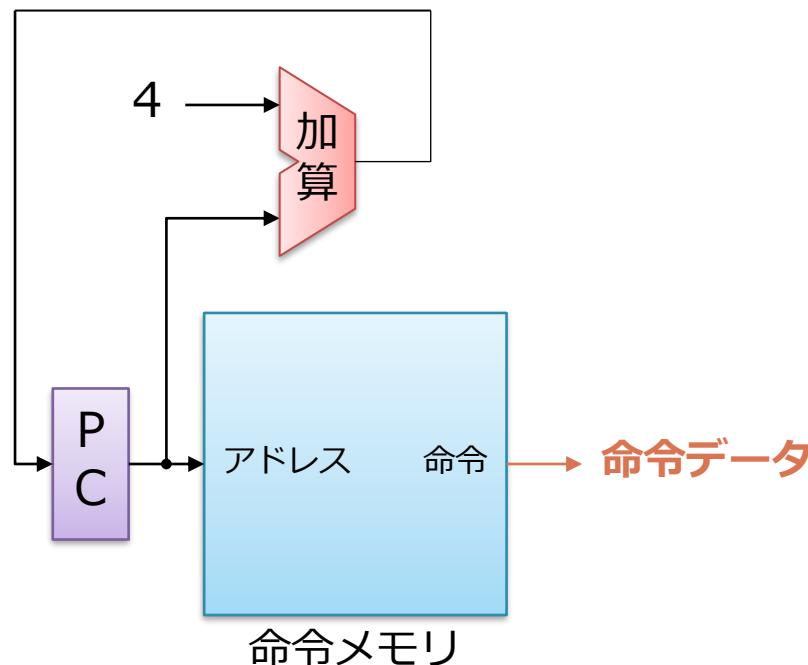
# 1命令の実行フェーズ

## ■ 実行フェーズ

1. フェッチ
2. デコード
3. レジスタ読み出し
4. 実行
5. レジスタ書き戻し

## ■ RISC-V の加算命令を実行する流れをざっとみる

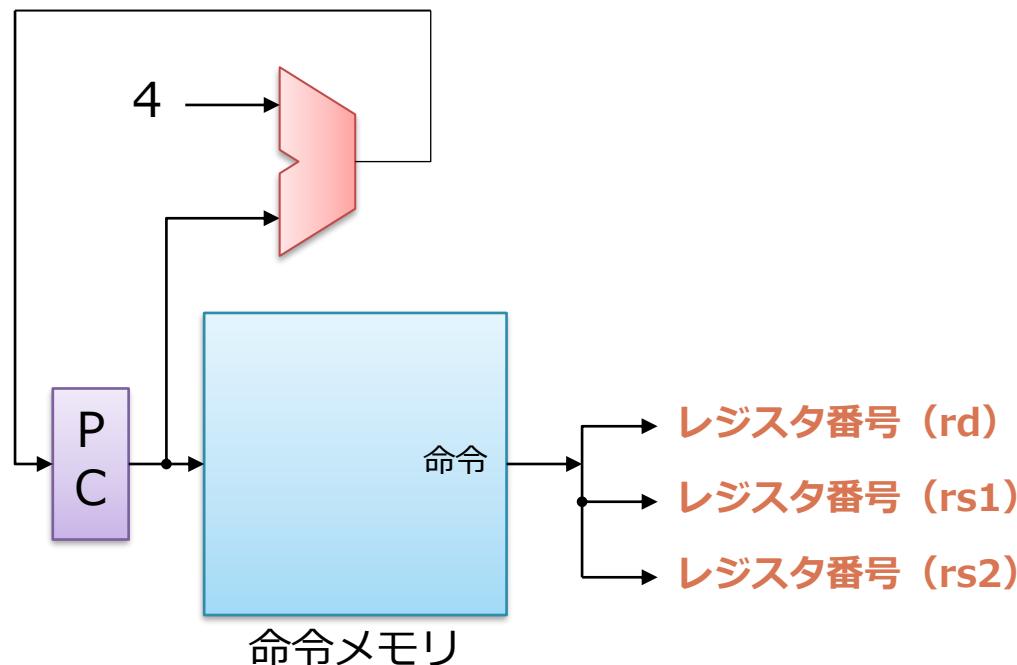
# 命令フェッチ



## ■ 命令メモリから命令を読み出す

- ◊ 命令メモリを順に読んでいくため, PC は毎サイクル加算される
- ◊ 足している 4 は, RSIC-V では命令の幅が 4 バイトだから
- ◊ 基本的に, この部分はどの命令でも変わらない

# 命令デコード

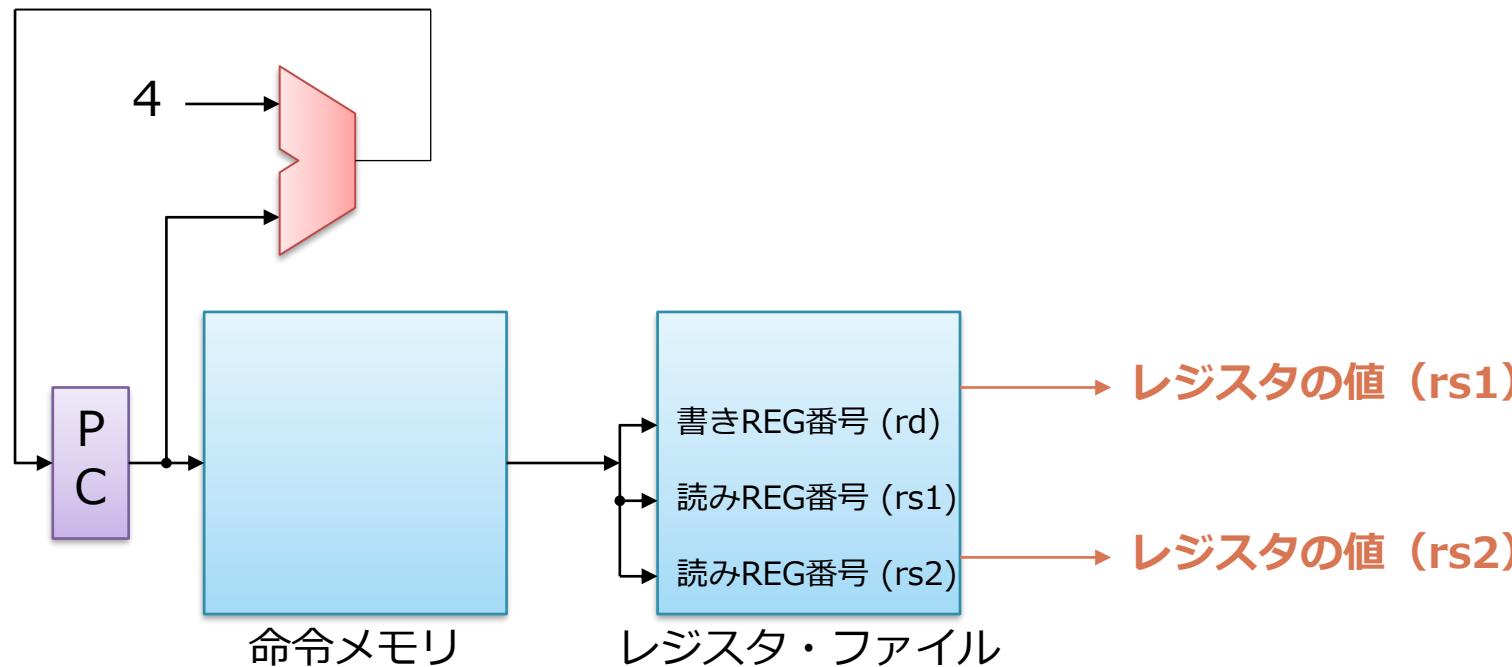


ADD :  $x[rd] \leftarrow x[rs1] + x[rs2]$

0000000	rs2	rs1	000	rd	0110011
---------	-----	-----	-----	----	---------

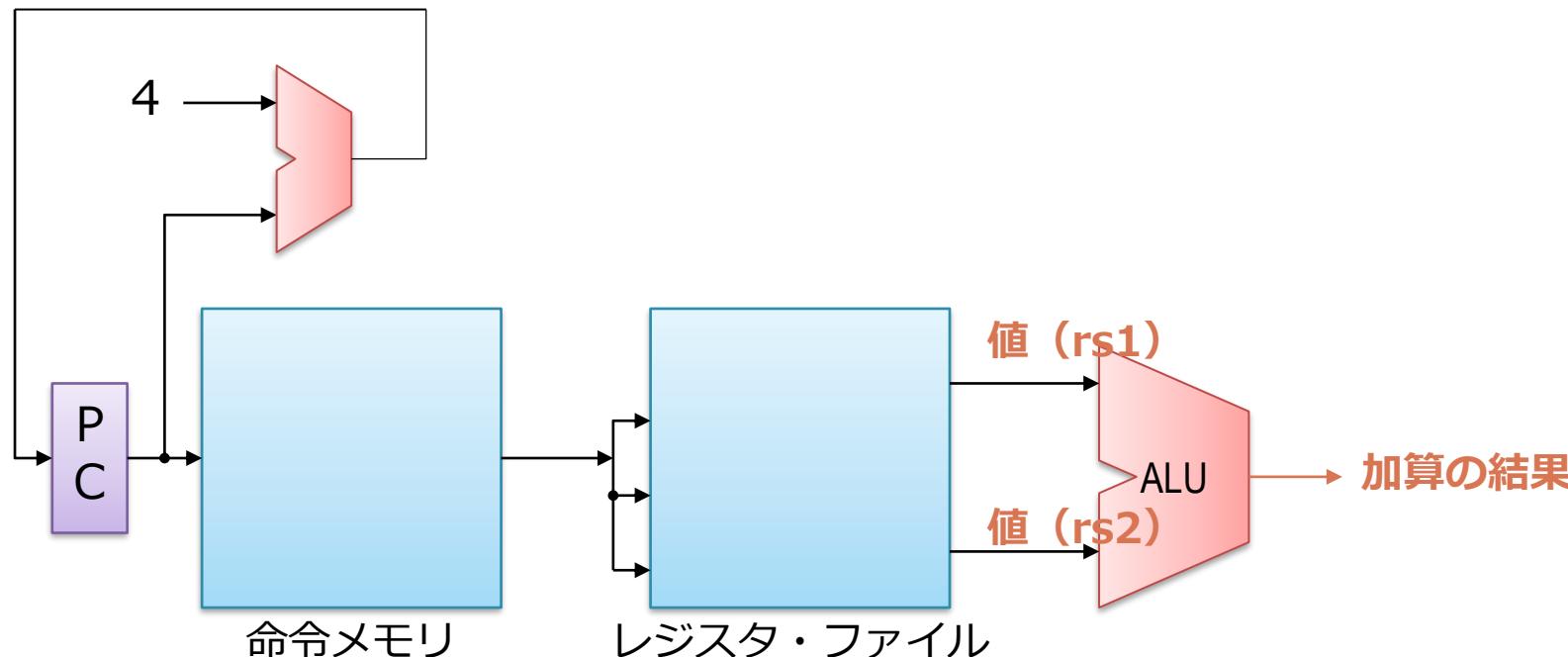
- 取り出した命令からレジスタ番号を表す部分のビットを取り出す
  - ◊ ソース (rs1, rs2) とディスティネーション (rd)

# レジスタ読み出し



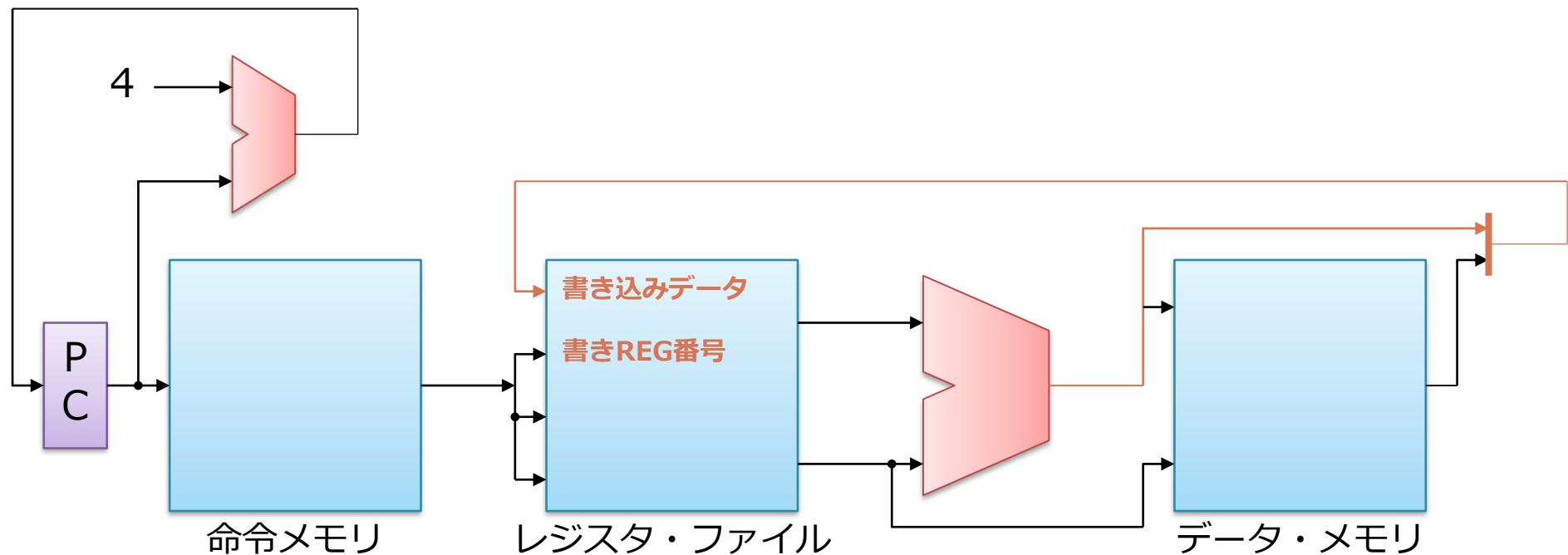
- デコードで得られたレジスタ番号を使って RF にアクセス
  - ◊ ソース・オペランドの値を読み出す

# 実行



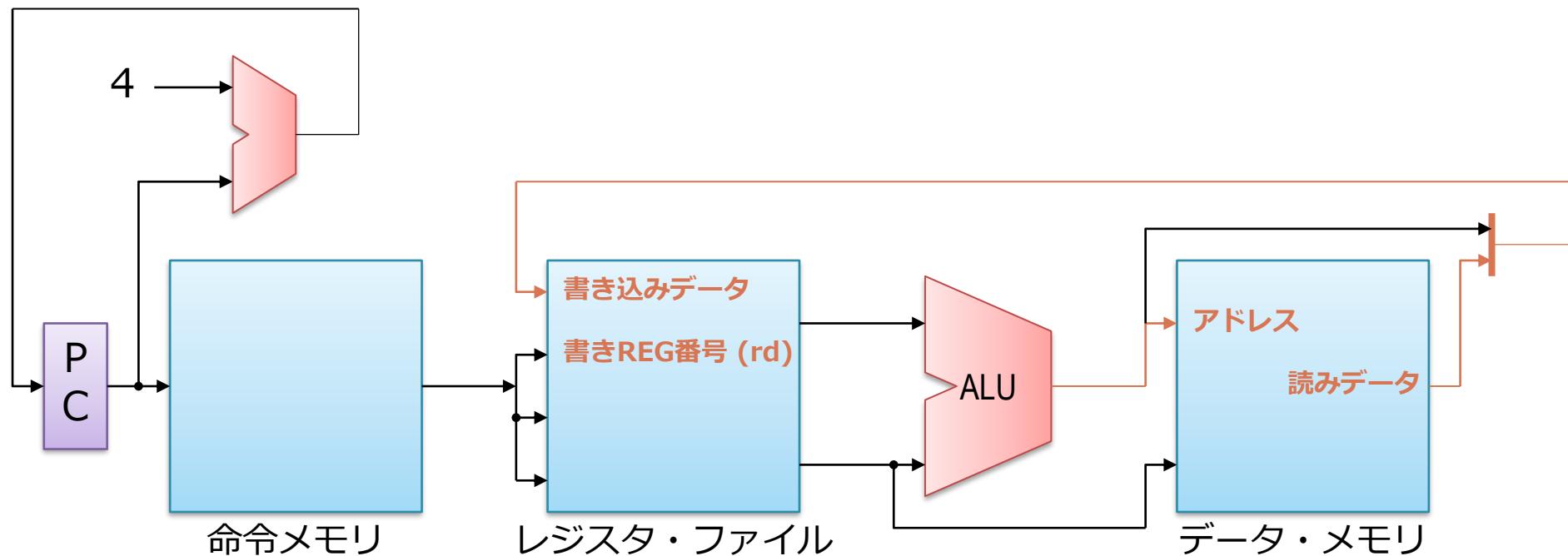
- RF から読みだした 2 つの値を加算

# レジスタ書き戻し



- 加算の結果をレジスタ・ファイルに書き戻す
  - ◊ データ・メモリには用がないので何もしない

# ロードの場合：メモリ・アクセスが加わる



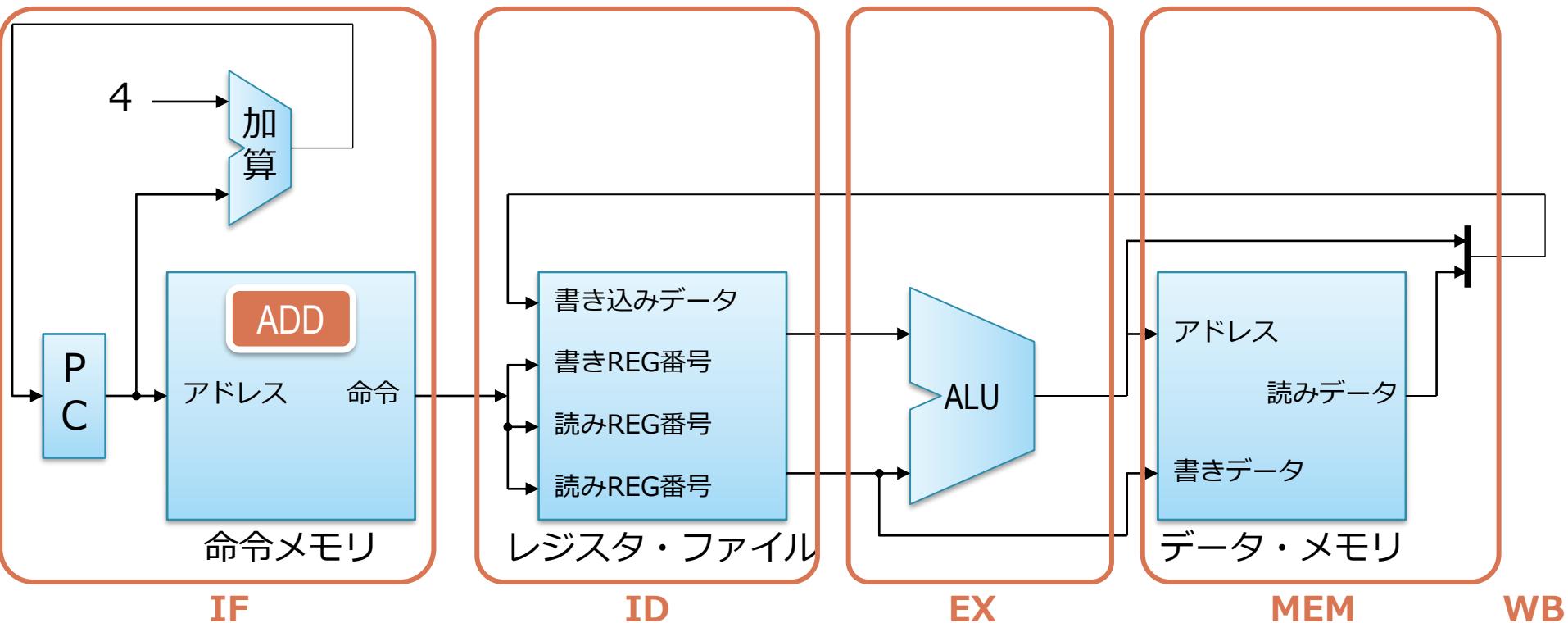
LW :  $x[rd] \leftarrow (x[rs1] + immediate)$

immediate[11:0]	rs1	010	rd	0000011
-----------------	-----	-----	----	---------

## ■ 加算命令との違い：

- ◊ アドレスの計算 ( $x[rs1] + immediate$ ) を ALU でやる
- ◊ 得られたアドレスでデータ・メモリにアクセス

# 各処理は基本的には左から右に流れる



- 特定のユニットで仕事をしている間、他の部分は遊んでいる
- パイプライン化
  - ◊ これをもとに、導入で話したように処理をオーバーラップさせる

# パイプライン化

---

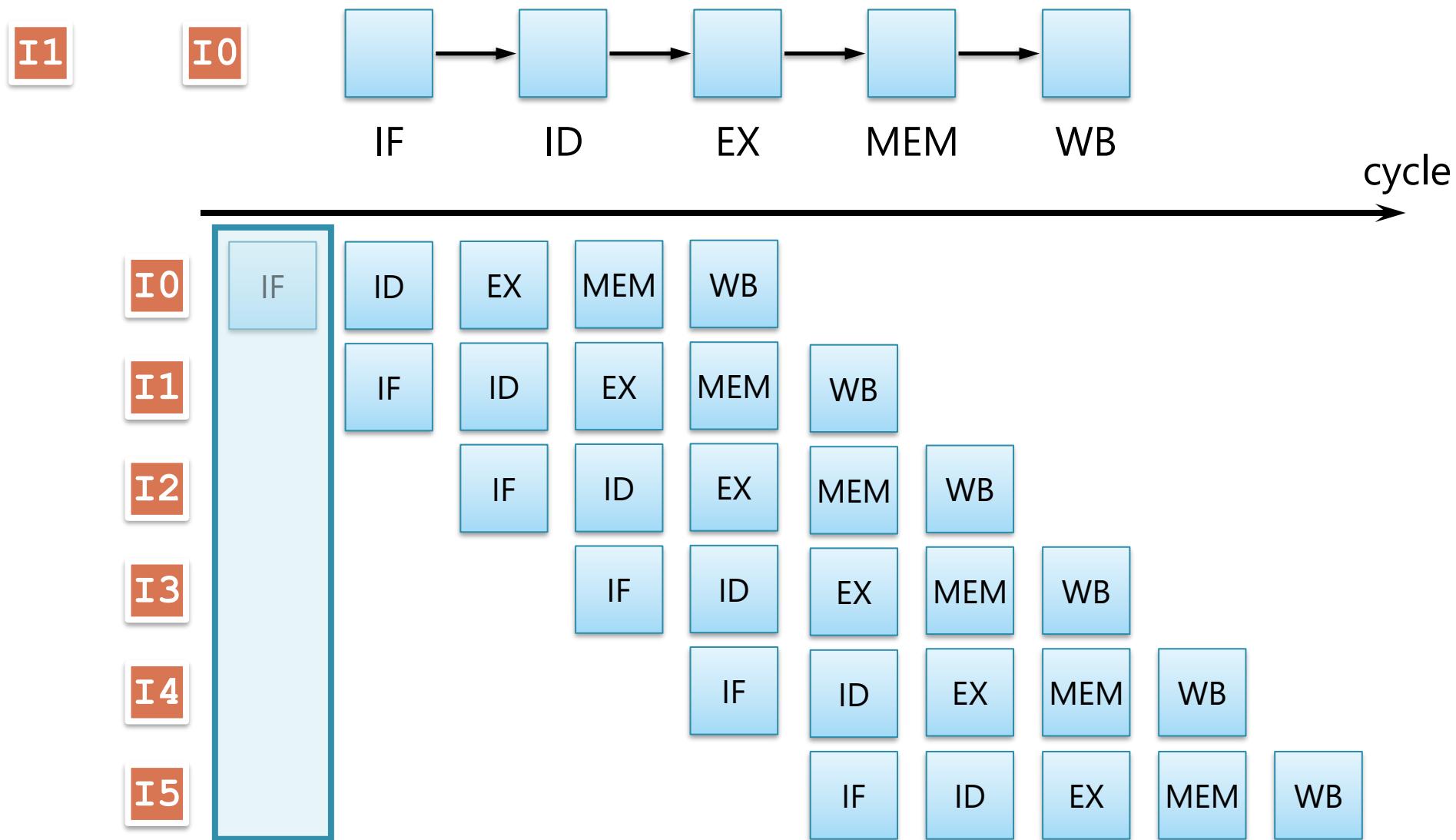
# もくじ

1. シングル・サイクル・プロセッサの動作
  - ◊ パイプライン化を前提とした構造のものを使って復習
  - ◊ 全ての命令の処理が 1 サイクルで完結
2. 上記のパイプライン化
  1. 具体的にどうパイプライン化するか
3. ハザード

# パイプライン化

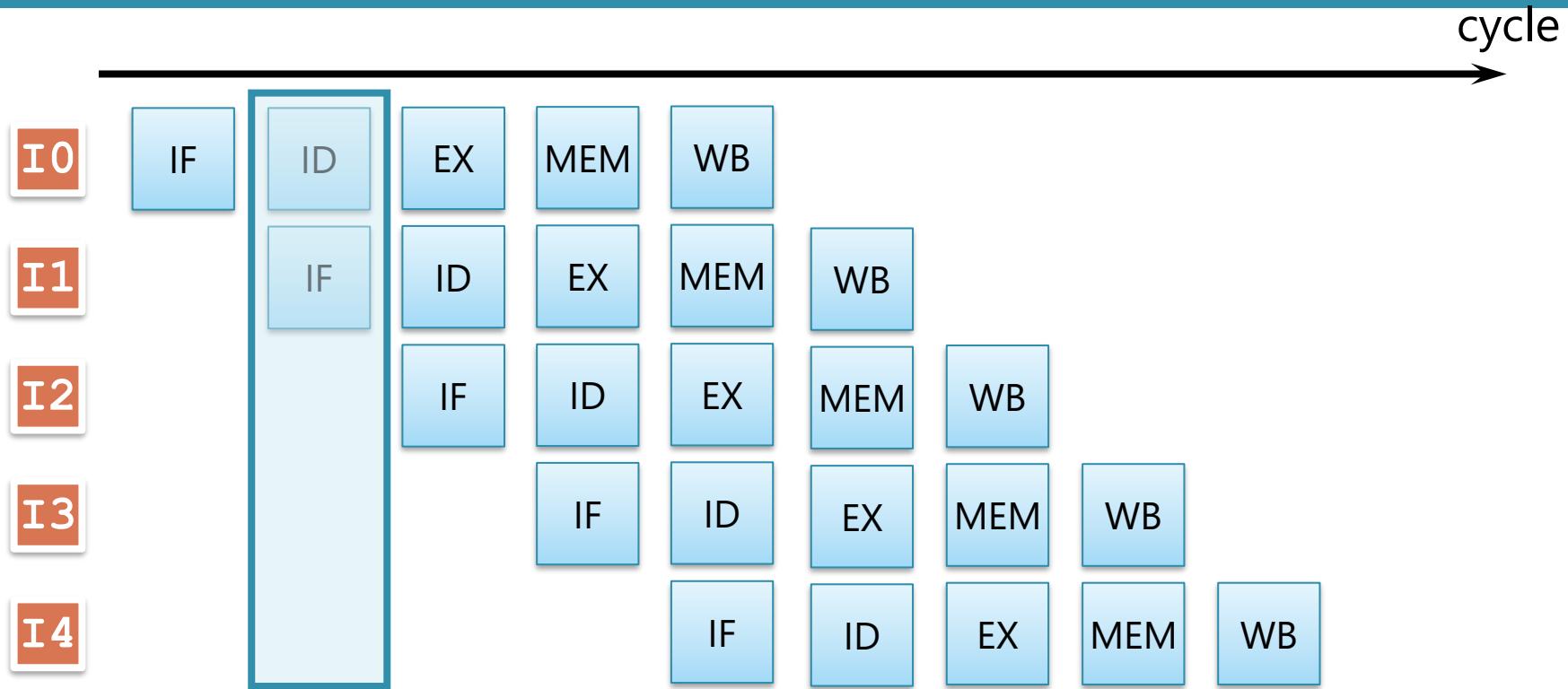
- 回路のまとまりをオーバラップさせる単位にする
  - ◊ この単位をステージと呼ぶ
- ステージ
  1. IF : 命令フェッチ
  2. ID : デコードとレジスタ読み出し
  3. EX : 実行
  4. MEM : メモリ・アクセス
  5. WB : レジスタ書き込み

# 命令パイプラインの実行の様子



# パイプライン・チャートの見方

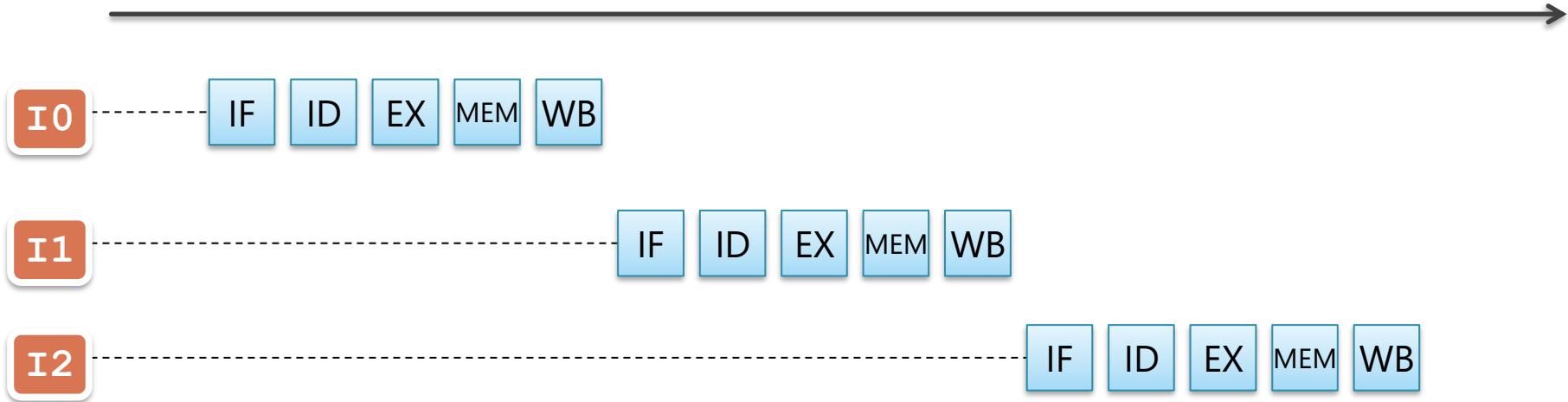
ここから先で多用されるので重要



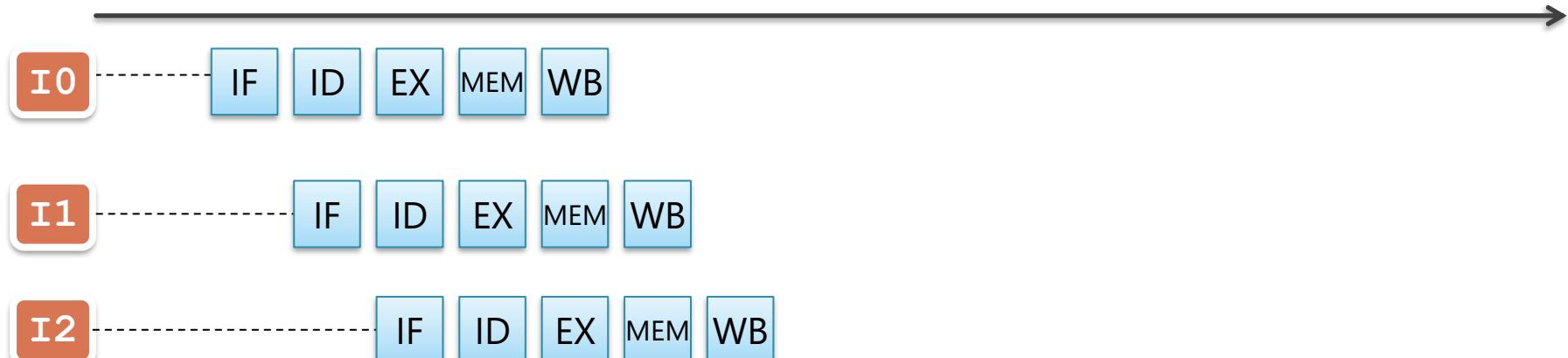
- ◇ 左から右にむかって時間は進む
- ◇ 上から下にむかって命令が実行順に置かれる
- ◇ 各ステージを表す四角は左側にある命令がその時そこにいることを示す
- 上記では2サイクル目に、I0 が ID に、I1 が IF で処理されている

# パイプライン化による性能（スループット）向上

パイプライン化しない場合



パイプライン化した場合



# 余談：実際の CPU を実行した場合のパイプライン

9ca4 r20 = lb(r19)	F 1 2   Rn 1   D 1   S1c   I 1 2 3   X 1 2   Wb 1   Cm 1
9ca8 sb(r20, r13)	F 1 2   Rn 1   D 1   Sw 1 2   S1c   I 1 2 3   X 1 2   Wb 1   Cm 1
9cac beq(r15, r7)	F 1 2   Rn 1   D 1   S1c   I 1 2 3   X Wb 1   f 1 2 3 4   Cm 1
9cb0 r20 = lb(r19)	F 1 2   Rn 1   D 1   S1c   I 1 2 3   X 1 2   Wb 1   f 1 2   Cm 1
9cb4 sb(r20, r13)	F 1 2   Rn 1   D 1   Sw 1   S1c   I 1 2 3   X 1 2   Wb 1   Cm 1
9cb8 beq(r15, r8)	F 1 2   Rn 1   D 1   S1c   I 1 2 3   X Wb 1   f 1 2 3 4   Cm 1
9cbc r20 = lb(r19)	F 1 2   Rn 1   D 1   S1c   I 1 2 3   X 1 2   Wb 1   f 1 2   Cm 1
9cc0 sb(r20, r13)	F 1 2   Rn 1   D 1   Sw 1   S1c   I 1 2 3   X 1 2   Wb 1   Cm 1
9cc4 bne(r15, r9)	F 1 2   Rn 1   D 1   S1c   I 1 2 3   X Wb 1   f 1 2 3   Cm 1
9cc8 r15 = lb(r19)	F 1 2   Rn 1   D 1   S1c   I 1 2 3   X 1 2   Wb 1   f 1 2   Cm 1
9ccc sb(r15, r13)	F 1 2   Rn 1   D 1   Sw 1 2   S1c   I 1 2 3   X 1 2   Wb 1   Cm 1
9cd0 r15 = andi(r11)	F 1 2   Rn 1   D 1   S1c   I 1 2 3   X Wb 1   f 1 2 3   Cm 1
9cd4 r15 = addi(r15)	[27050, 37429] F 1 2   Rn 1   D 1   Sw 1   S1c   I 1 2 3   X Wb 1   f 1 2   Cm 1
9cd8 r14 = add(r12, r14)	F 1 2   Rn 1   D 1   S1c   I 1 2 3   X Wb 1   f 1 2 3 4   Cm 1
9cdc r15 = slli(r15)	F 1 2   Rn 1   D 1   Sw 1   S1c   I 1 2 3   X Wb 1   f 1 2   Cm 1
9ce0 sb(r29, r14)	F 1 2   Rn 1   D 1   S1c   I 1 2 3   X 1 2   Wb 1   f 1   Cm 1
9ce4 r15 = srli(r15)	F 1 2   Rn 1   D 1   Sw 1   S1c   I 1 2 3   X 1 2   Wb 1   f 1   Cm 1
9ce8 r14 = srli(r11)	F 1 2   Rn 1   D 1   S1c   I 1 2 3   X Wb 1   f 1 2 3 4   Cm 1
9cec r13 = addi(r18)	F 1 2   Rn 1   D 1   Sr   S1c   I 1 2 3   X Wb 1   f 1 2 3   Cm 1
9cf0 r14 = andi(r14)	F 1 2   Rn 1   D 1   Wku   S1c   I 1 2 3   X Wb 1   f 1 2   Cm 1
9cf4 bgeu(r17, r15)	F 1 2   Rn 1   D 1   Sw 1   S1c   I 1 2 3   X Wb 1   f 1   Cm 1
9b28 r15 = slli(r15)	F 1 2   Rn 1   D 1   Sw   S1c   I 1 2 3   X Wb 1   f 1 2   Cm 1
9b2c r15 = add(r15, r28)	F 1 2   Rn 1   D 1   Sw 1   S1c   I 1 2 3   X Wb 1   f 1   Cm 1
9b30 r15 = lw(r15)	F 1 2   Rn 1   D 1   Sw 1 2   S1c   I 1 2 3   X 1 2   Wb 1   Cm 1
9b34 r14 = slli(r14)	F 1 2   Rn 1   D 1   Sw   Wku   S1c   I 1 2 3   X Wb 1   f 1 2   Cm 1
9b38 r14 = add(r6, r14)	F 1 2   Rn 1   D 1   Sw 1   S1c   I 1 2 3   X Wb 1   f 1   Cm 1

■ 塩谷が開発している RISC-V CPU (RSD) の実行を可視化したもの

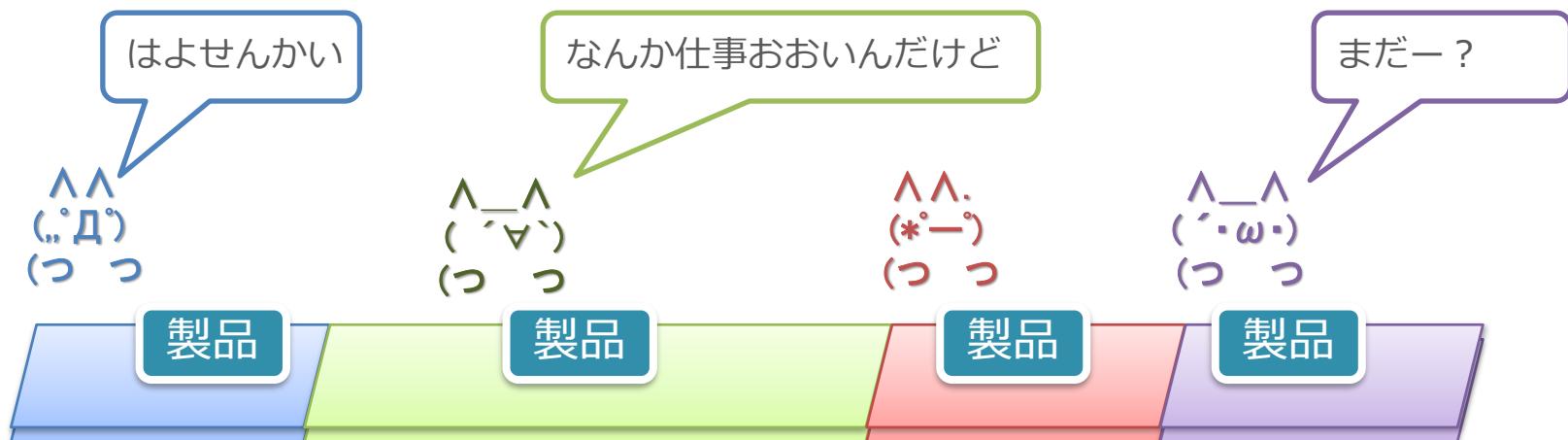
- ◇ <https://github.com/rsd-devel/rsd>
- ◇ out-of-order 実行をしているので、途中からプログラム順とは異なるタイミングで実行が進んでいる

# パイプライン化の効果

- レイテンシ (latency) : 短くならない (か, やや延びる)
  - ◊ 一続きの処理が始まってから終わるまでにかかる時間
  - ◊ この場合, 1命令の始まりから終わりまでの処理時間
  - ◊ 原理的に短くならない (ステージ間にFFが入る分のびる)
- スループット (throughput) : ステージ数倍だけ上がる
  - ◊ 単位時間当たりの処理量
  - ◊ この場合, 単位時間あたりに実行される命令数

# ステージを「どこで」切るか

- 大きな回路のまとまりをステージにする
  - ◊ 回路のまとまりが大きい → 遅延も大きい
- この遅延の大きさが揃っていないと、綺麗にうごかない
  - ◊ パイプライン全体は、一番遅いステージの遅延にあわせて動く
  - ◊ 他の人が仕事が終わったからと言って、先に送れない
- 良くない例：緑の人だけ仕事が多いので、全体が動かせない



# ステージを「どこで」切るか

## ■ ステージ

1. IF : 命令フェッチ
2. ID : デコードとレジスタ読み出し
3. EX : 実行
4. MEM : メモリ・アクセス
5. WB : レジスタ書き込み

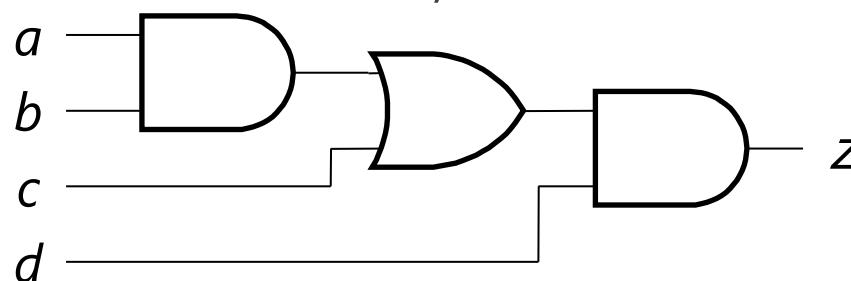
## ■ 上記では、デコードとレジスタ読み出しが ID ステージにまとめられている

- ◊ デコードにかかる遅延はほとんどない
- ◊ 読み出した命令からオペランドを取り出すのは、単に信号線を繋ぐだけで良い

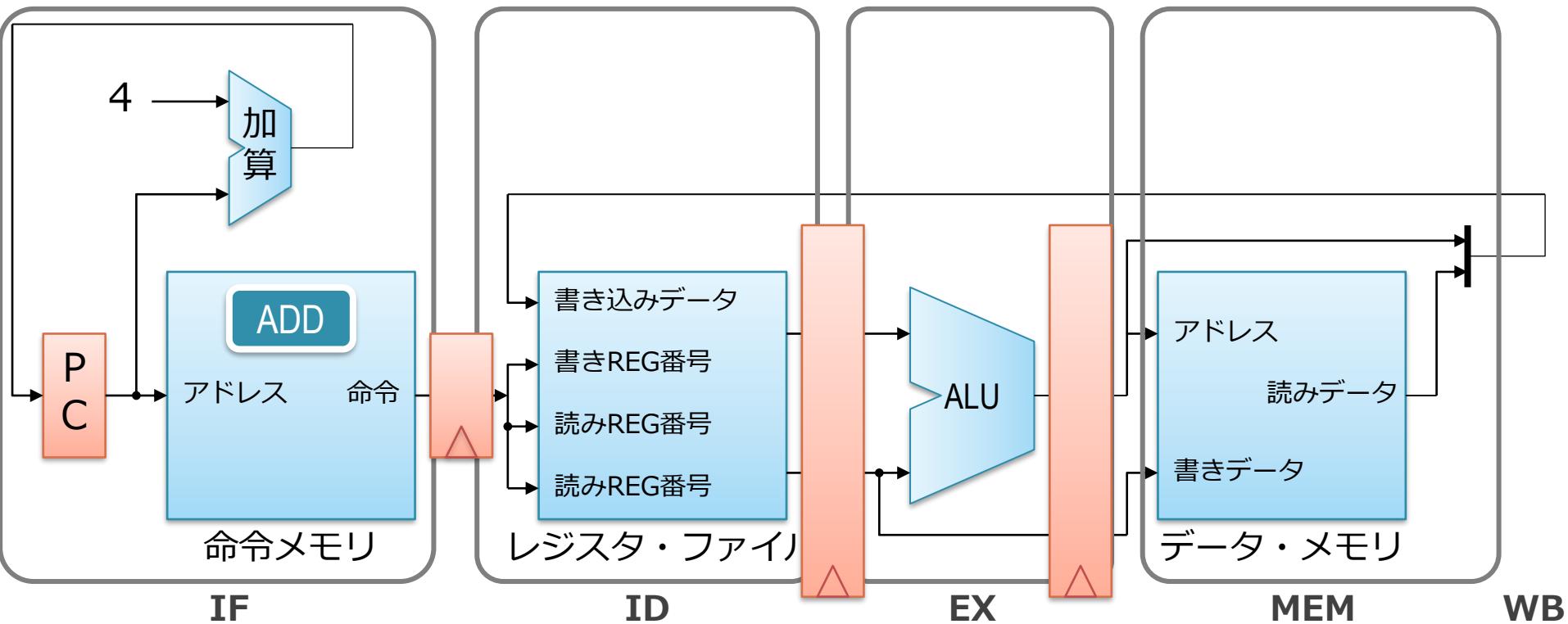
# ステージを「どうやって」切るか

- シングル・サイクル・プロセッサの回路に適当に間隔をあけて命令を流せばよいというものもない

1. 各ステージを完全に同じ長さにすることは凄く難しい
  - ◊ 同じ長さ=同じ遅延=全く同じ段数の組み合わせ回路
2. 長いステージであっても信号は絶えず変化する可能性がある
  - ◊ 短いパスから順に出力に反映される
  - ◊ たとえば下の回路で  $a, b, c, d$  が全て変化したとすると、まず  $d$  の変化が  $z$  に反映し、次に  $d$  が…



# パイプライン化（オーバーラップ）の実現方法



- ◊ 各ステージの間に、D-FF（オレンジの四角）をいれる
  - WB の書き込みについては、レジスタ・ファイル 자체がクロックに同期して書き込みが行われるので D-FF は不要
- ◊ 各ステージの処理が早く終わっても、次のクロックまでは D-FF で信号の伝搬は止まる

# 余談：非同期回路やウェーブ・パイプライン

- クロックによる同期化を使わずにパイプラインを作る方法もあるにはある
- やり方：
  1. 色々な方法でステージ間の遅延の大きさを気合いで揃える
  2. 一定間隔でデータを流す
- 設計 & 動作させることがすごく難しいので、主流ではない
  - ◊ 特に、高速動作がかなり難しい

# ハザード

---

# もくじ

1. シングル・サイクル・プロセッサの動作
  - ◊ パイプライン化を前提とした構造のものを使って復習
  - ◊ 全ての命令の処理が 1 サイクルで完結
2. 上記のパイプライン化
  1. 具体的にどうパイプライン化するか
3. ハザード

# ハザード

## 1. 構造ハザード

1. 構造ハザードとはなにか？
2. その解決方法

## 2. 非構造ハザード

- a. データ・ハザード
- b. 制御ハザード

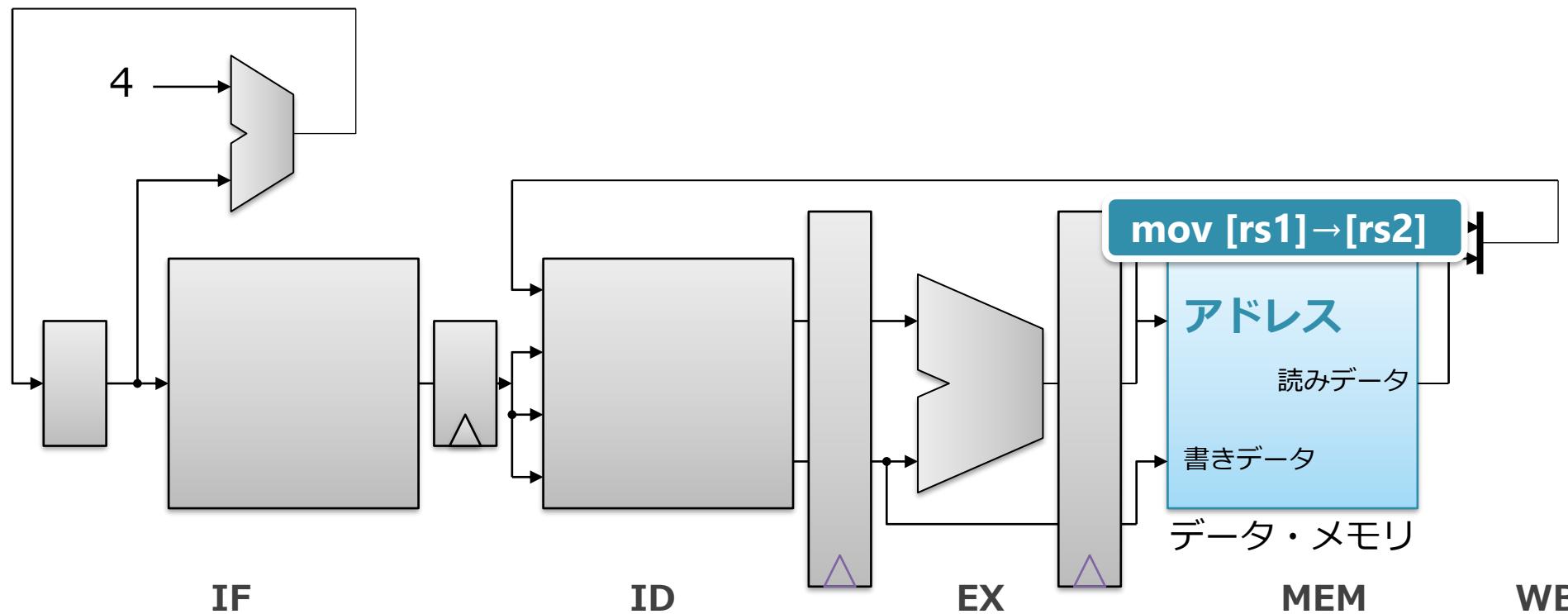
# 構造ハザード

- ハード資源の不足により、パイプラインがうまく動作しないこと
- いくつかの例を使った説明、解消方法について解説

# 構造ハザードの例 1：メモリ間 mov

- 例 1：仮に  $\text{mov } [\text{rs1}] \rightarrow [\text{rs2}]$  のような命令があったとする
  - ◊  $\text{rs1}$  で指定されるアドレスのメモリの値を読んで、
  - ◊  $\text{rs2}$  で指定されるアドレスのメモリに書き込む
- 実際に、x86 にはこのような命令がある

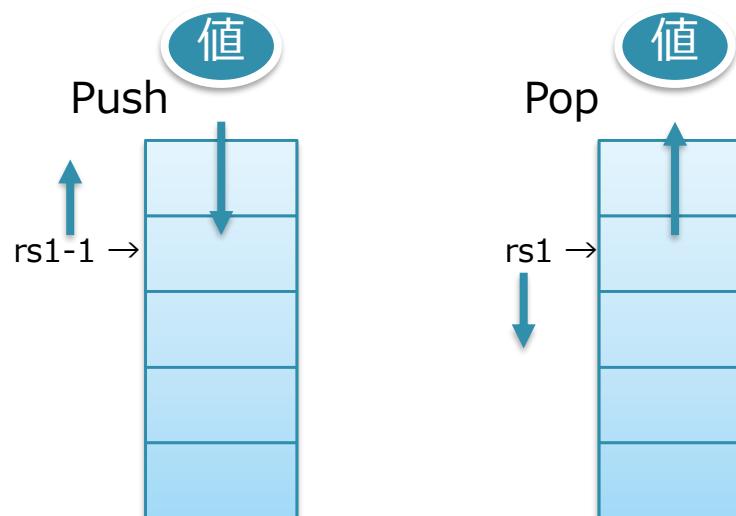
# **mov [rs1]→[rs2] // [rs1]→[rs2]へのコピー**



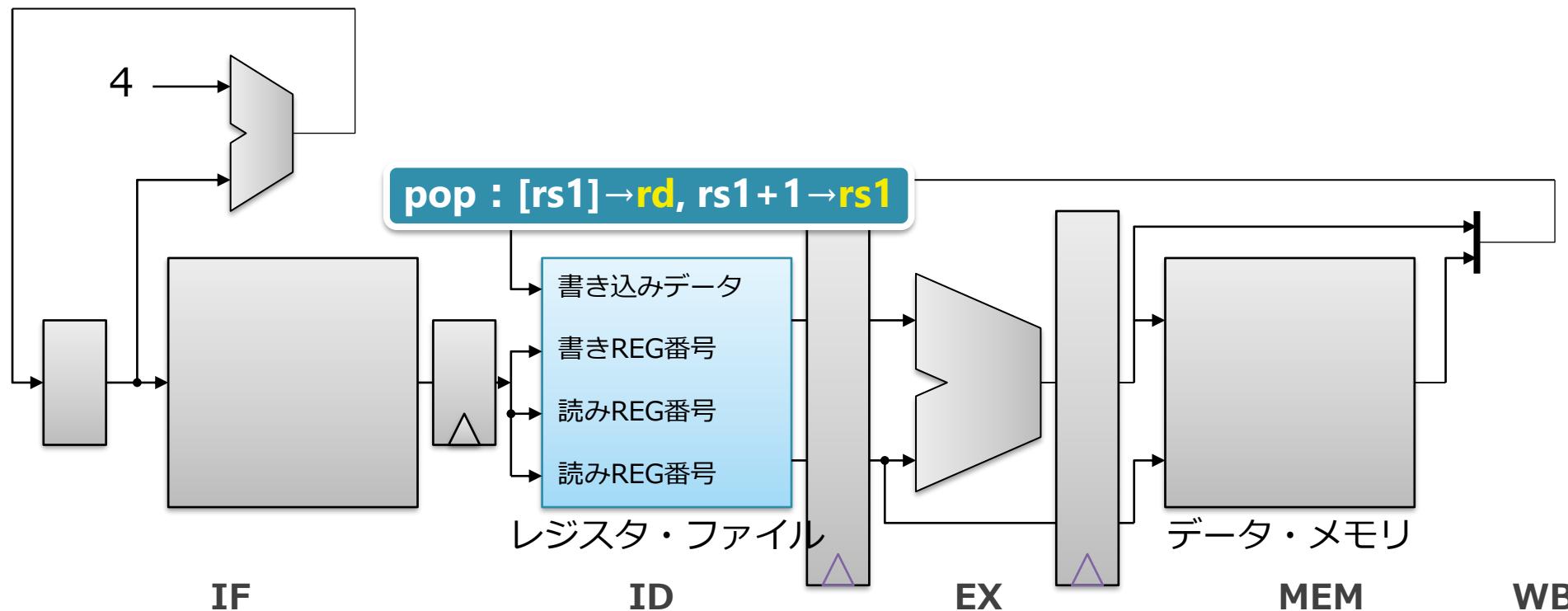
- メモリをあるサイクルに同時に読んで書く必要がある
  - ◊ しかし、データ・メモリのアドレスの口は1つしかない
  - ◊ MEMステージでデータ・メモリの読みと書きが同時にできない

# 構造ハザードの例 2 : push/pop

- x86 や ARM ではスタック操作のための push/pop 命令がある
  - ◊ push :  $rs1-1 \rightarrow rd$ ,  $r2 \rightarrow [rd]$ 
    1. スタック・ポインタ（のレジスタ）をデクリメントし,
    2. それをアドレスにしてメモリに値を書き込む
  - ◊ pop :  $[rs1] \rightarrow rd$ ,  $rs1+1 \rightarrow rs1$ 
    1. スタック・ポインタをアドレスにして値を読む
    2. スタック・ポインタをインクリメント



# $\text{pop} : [\text{rs1}] \rightarrow \text{rd}, \text{rs1} + 1 \rightarrow \text{rs1}$



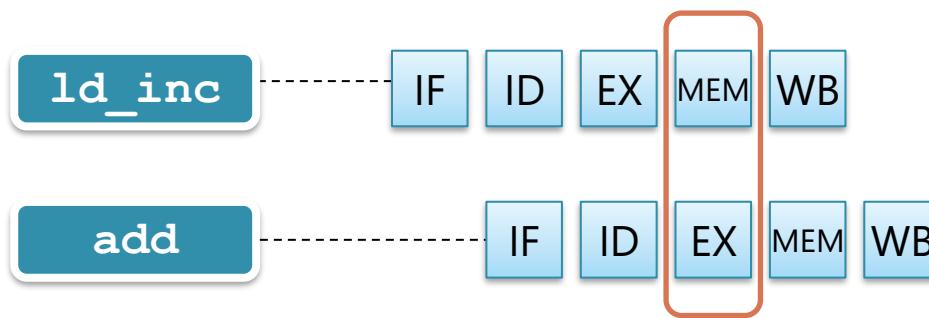
- WB ステージでレジスタに  $\text{rd}$  と  $\text{rs1}$  の2つを書き込む必要がある
  - ◊ レジスタ・ファイルへの書き込みは、同時に2つはできない

# 構造ハザードの例 3

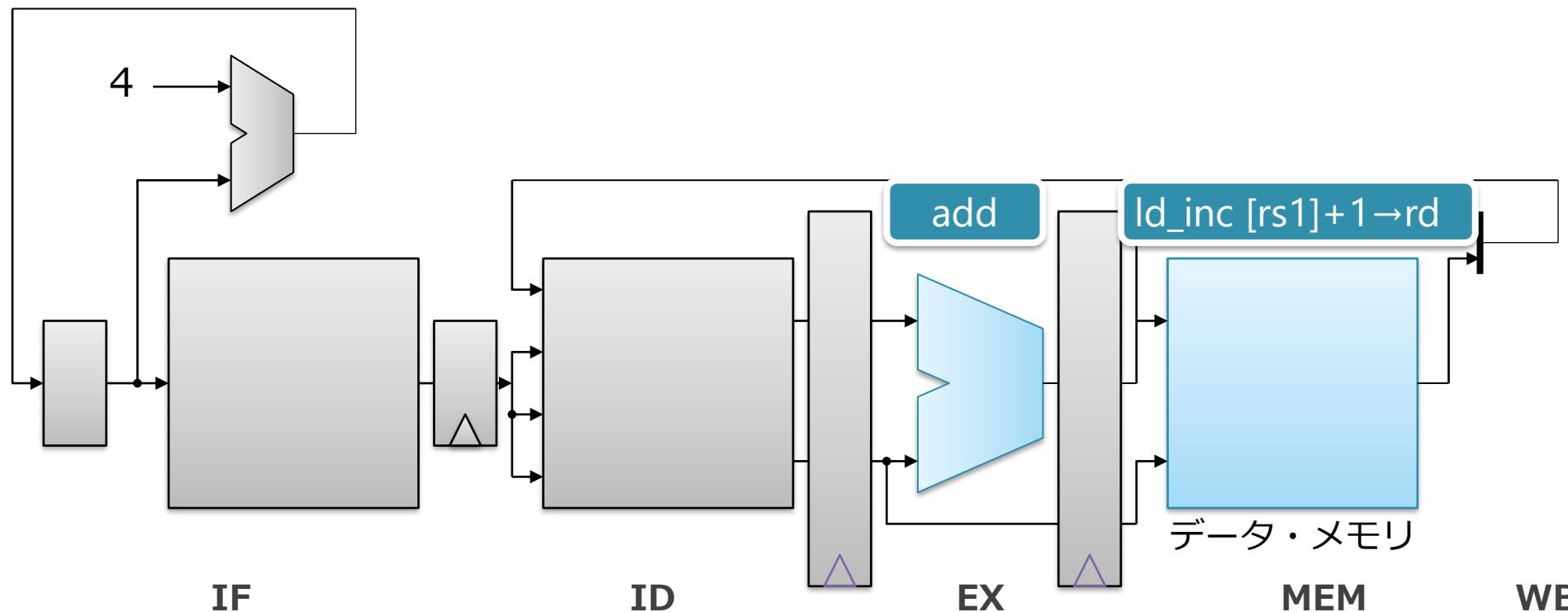
- 使用資源の異なるステージ間のぶつかりでも起きる
  - ◊ これまでの例は、同じステージ内で資源が足りない例
- **ld\_inc [rs1]+1→rd** のような命令があったとする
  1. rs1 の指すアドレスからメモリを読む
  2. 読んだ値にさらに + 1 してから rd に書く
- 一見、資源は足りているようだが…
  - ◊ レジスタの読み書きは 1 つずつしかない
  - ◊ メモリも 1 力所を読むだけ
  - ◊ 加算も 1 回行うだけ

# 構造ハザードの例 3

- $ld\_inc [rs1]+1 \rightarrow rd$  と add が連続した場合：
  - ◊  $ld\_inc$  で、 MEM ステージから読んだ値を加算しようとしても、
  - ◊ そのサイクルは後続の add が演算器を使っているので使用できない



# $ld\_inc [rs1]+1 \rightarrow rd$ と add が連続した場合



- EX ステージ以外では、演算器にはアクセスできない
  - ◊ 他の命令が使っている可能性がある

# 構造ハザードの解決方法

## ■ 解決方法

1. ハードウェアの増強
2. 時分割処理
3. マイクロ命令への変換

# 解決方法 1：ハードウェアの増強

## ■ ハードウェアを増強する

- ◇ mov [rs1]→[rs2]
  - 複数箇所のメモリを同時に読み書きできるように
- ◇ pop
  - レジスタに 2 つ同時に書き込むように
- ◇ Id\_inc [rs1]+1→rd
  - MEM ステージに専用の加算器を追加

# 解決方法 1：ハードウェアの増強

- 利点：オーバーヘッドをいとわなければ、基本これで解決
- 欠点：回路規模が増える
  - 1. 機能の増強量に比例した回路が必要
    - なにも考えないで対応していくと、ものすごい数の回路になる
    - 例：ARM は全 16 レジスタを一気にメモリに書ける命令がある
  - 2. 機能の増強量に対して、線形より大きなオーダーで回路規模が増える場合もある
    - 加算器などなら、増やした数の分だけ線形に回路が増える
    - メモリやレジスタは、同時に読み書きできる数の 2 乗で回路が大きくなる（今後の講義で説明）

# 構造ハザードの解決方法

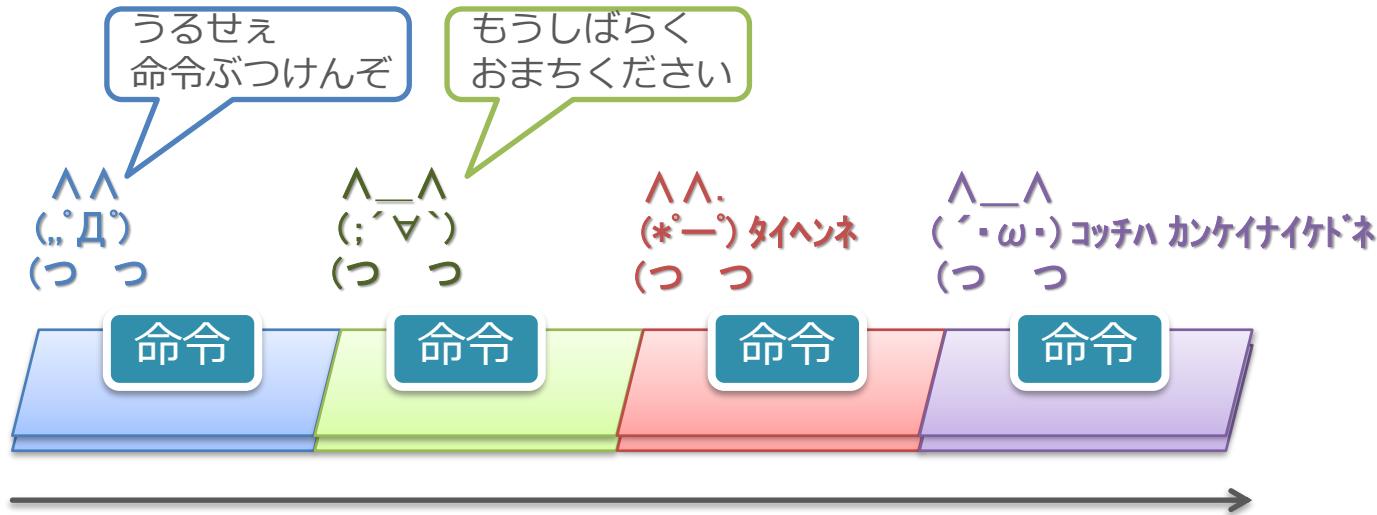
## ■ 解決方法

1. ハードウェアの増強
2. 時分割処理
3. マイクロ命令への変換

# 解決方法 2 : 時分割で処理

- 構造ハザードの原因：
  - ◊ ハードウェア（の機能）が足りない
- パイプラインを止めて、複数のサイクルをかけて処理する
  - ◊ mov [rs1]→[rs2]
    - メモリを読んだあと、次のサイクルで書きこむ
  - ◊ pop
    - 1つレジスタに書いたあと、次のサイクルで書き込む
  - ◊ Id\_inc [rs1]+1→rd
    - Id\_inc が MEM で値を読んだら、次のサイクルで +1

# なぜパイプラインを止めるのか

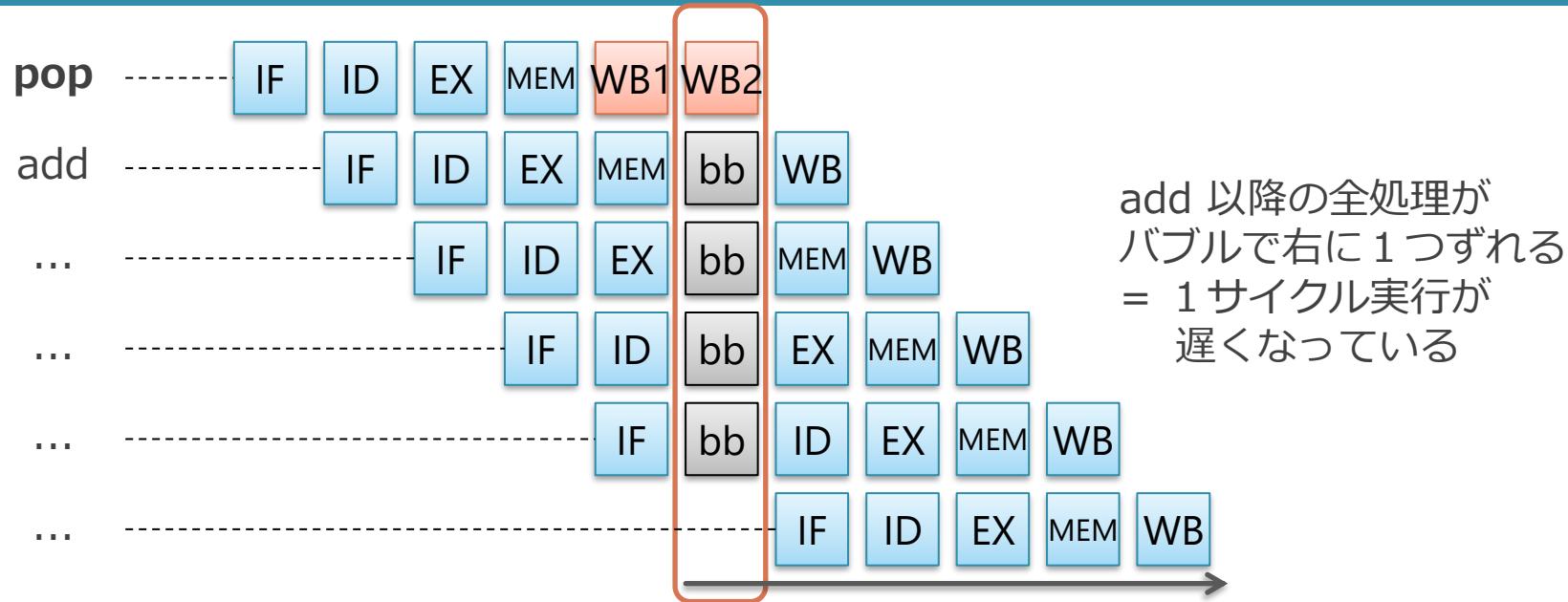


- 上流を止めないと破綻する
  - ◊ (;`^V^)` が複数サイクルをかけて仕事をしている場合、命令はそこにとどまり続ける
  - ◊ その間は上流をとめないと命令をおく場所がないし、依存関係がまもられない
- (\*ー\*) より下流は流れていっても、この場合は問題ない

# パイプラインを止めること

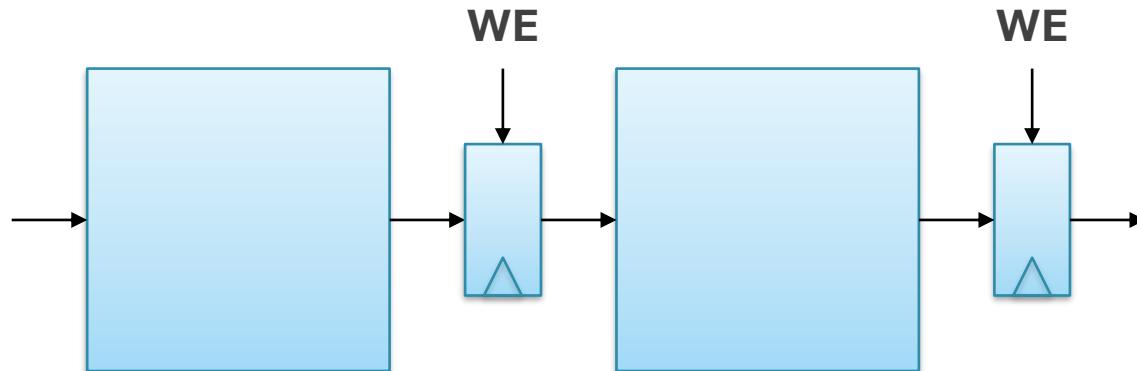
- パイプラインを止めるとことを「ストール」や「インターロック」という
  - ◊ 本や人によって、意味や使い方が微妙に統一されていない
  - ◊ この講義では、以降はストールで統一

# ストールの動作



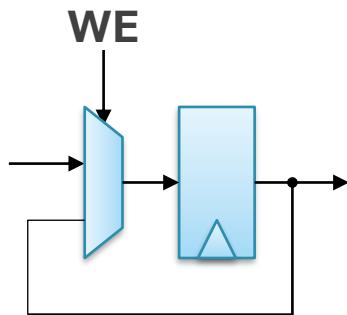
- pop : 1 つレジスタに書いたあと、次のサイクルで書き込む
  - ◊ WB1 と WB2 の 2 サイクルで書き込む
  - ◊ WB2 の間は上流を全て止める
- パイプライン・チャート上では上記のようになる
  - ◊ 止める原因の命令の下が全部右にずれる
  - ◊ ずれた部分の空き (bb) を「バブル」とよぶ

# ストールの実現方法



- 回路的には、Write Enable (WE) つきの D-FF を使う
  - ◊ WE が 0 のサイクルは書き込みが行われない
  - ◊ ストールさせたい時は、そのステージの WE を 0 に

# WE つき D-FF の実現方法



- たとえば D-FF とマルチプレクサで作れる
  - ◊ WE が 0 の時は、その時の自分自身の出力を書き込む

# 構造ハザードの解決方法

## ■ 解決方法

1. ハードウェアの増強
2. 時分割処理
3. マイクロ命令への変換

# 解決方法 3 : マイクロ命令への変換

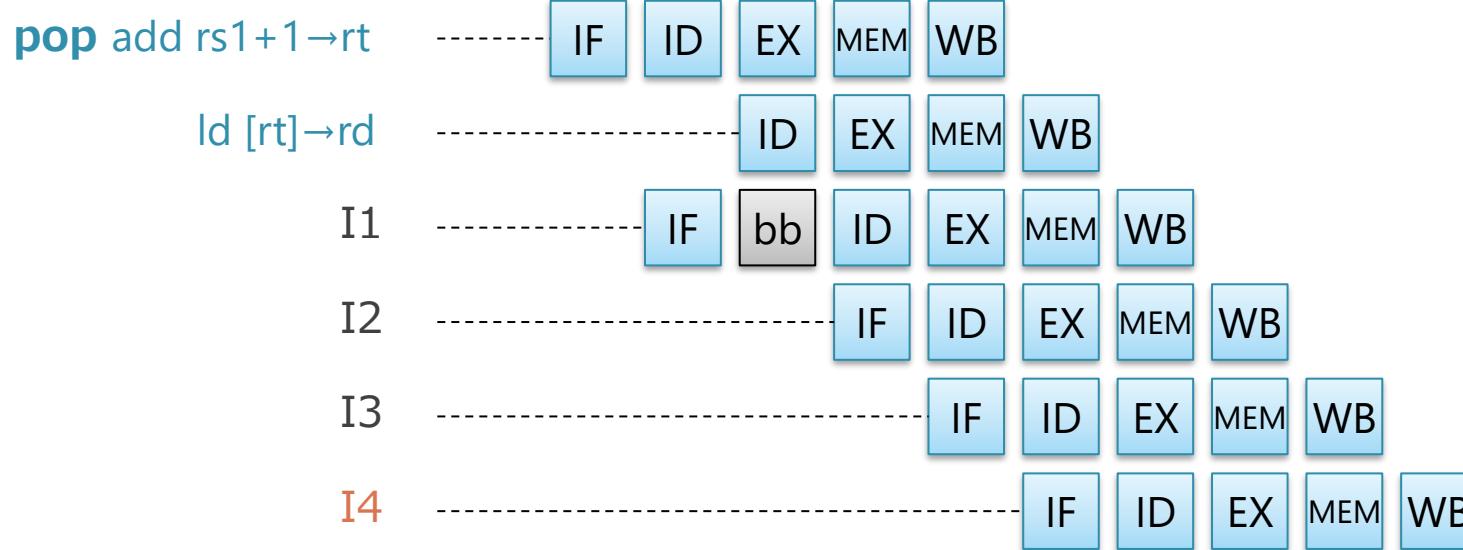
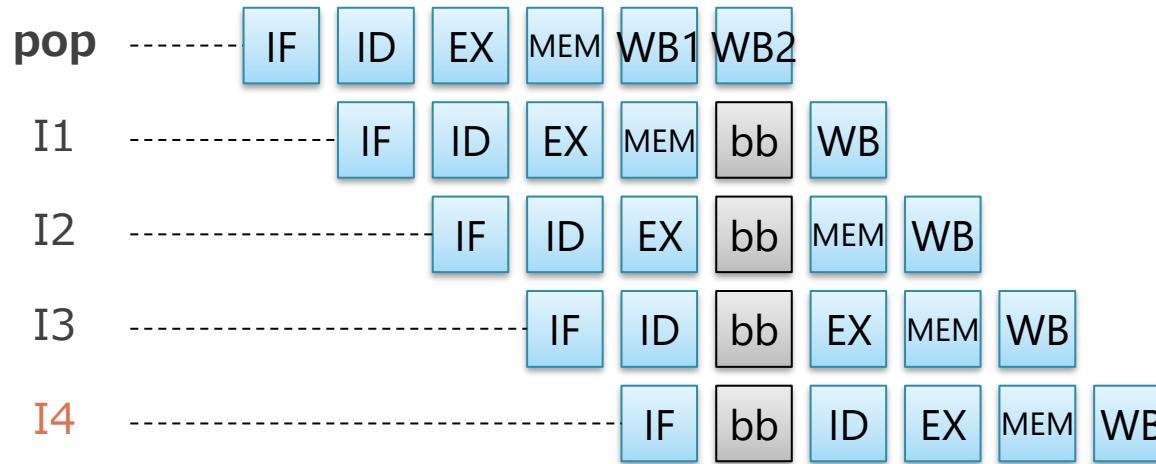
- 複数のマイクロ命令に分解して実行
  - ◊ マイクロ命令 : CPU の内部でのみ使われる命令
    - プログラマからは全く見えない
  - ◊ マイクロ命令は、構造ハザードを起こさないよう設計しておく
- 現代の x86 や ARM は、主にこの方法を採用している

# マイクロ命令への変換の例

- mov [rs1]→[rs2]
  1. ld [rs1]→rt
  2. st rt→[rs2]
- pop
  1. add rs1+1→rt
  2. ld rt→rd
- ld\_inc [rs1]+1→rd
  1. ld [rs1]→rt
  2. add rt+1→rd

rt はプログラマから見えない CPU 内部にある中間結果を保持するレジスタ

# 時分割処理とマイクロ命令への分解の比較 I4 が終わる時間は変わらない



■ ID でマイクロ命令に分解 = デコードで時分割処理している

# マイクロ命令への分解の利点

- 処理時間が変わらないのなら、なぜこんな複雑なことをするのか？
- 分解後は、構造ハザードのことを一切考えなくてよくなるから
  - ◊ mov, pop, Id\_inc が連續で来た場合、どう止めたらよいのか？
    - 止めるべきステージの場所はさまざま
    - 組み合わさると意味がわからない
  - ◊ マイクロ命令に分解してしまえば、ID ステージでのストールのみ考えれば良い

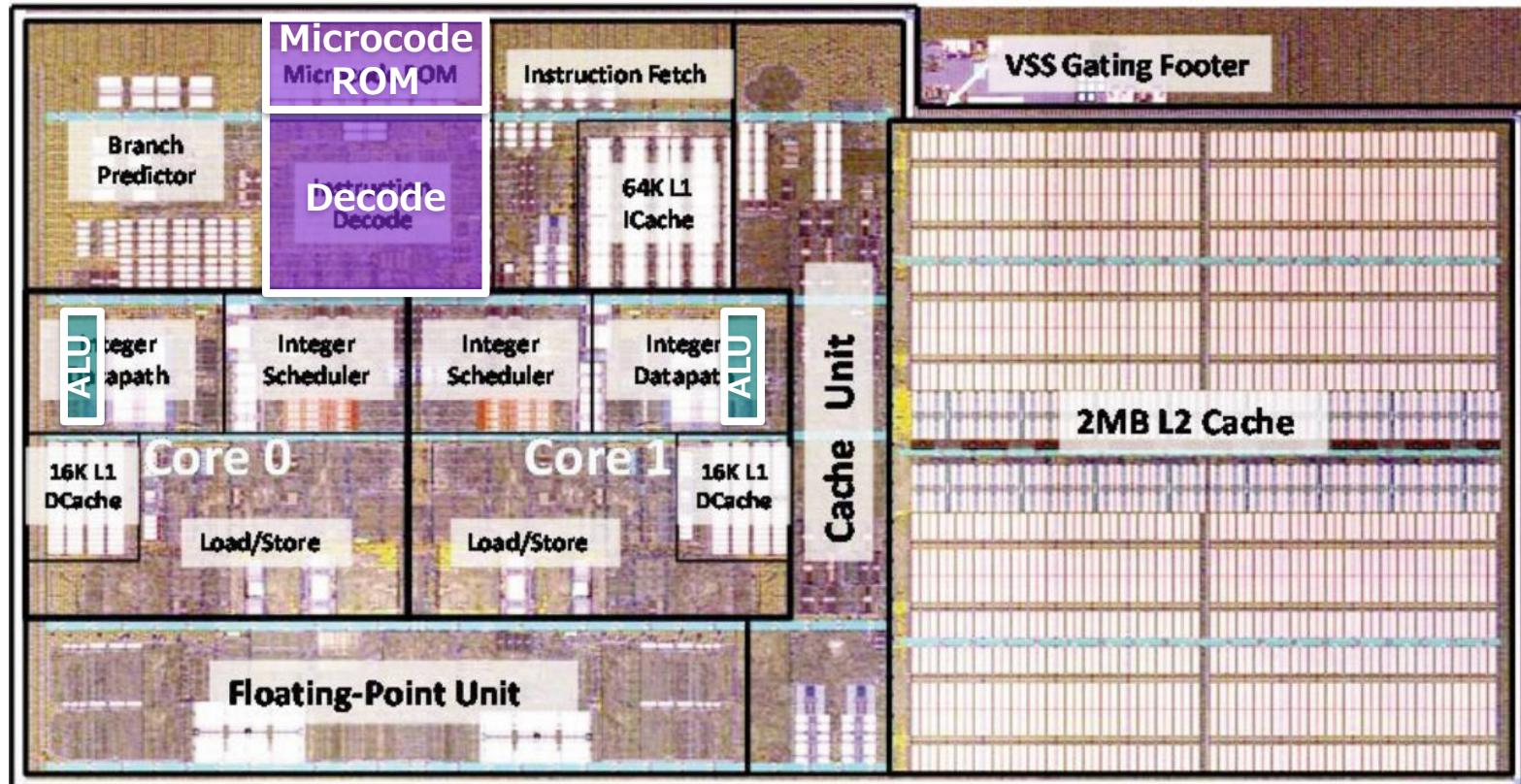
# マイクロ命令への分解の利点

- 内部の設計をクリーンにできる
  - ◊ スーパスカラ（パイプラインを複数並列に並べる）などでは、こうしないと複雑すぎて無理
- = 内部を刷新しつつ、プログラムの互換性を保てる

# マイクロ命令への分解の欠点

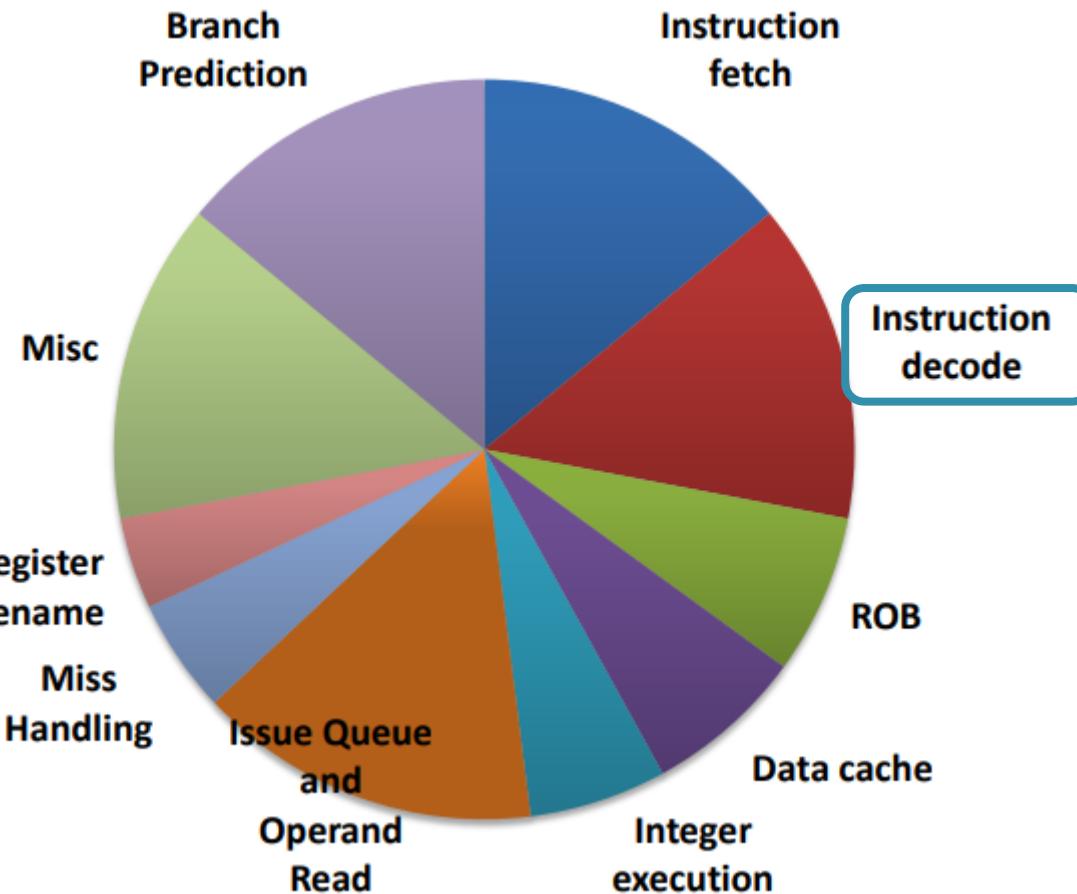
- 分解（デコード）がマジで大変
- 基本的には、ひたすらパターン・マッチング
  - ◊ でかい真理値表がいる
  - ◊ 本当に複雑なものは、メモリで出来たテーブルも使う

# AMD Bulldozer のチップ写真



- Tim Fischer<sup>1</sup> , Srikanth Arekapudi<sup>2</sup> , Eric Busta<sup>1</sup> , Carl Dietz<sup>3</sup> , Michael Golden<sup>2</sup> , Scott Hilker<sup>2</sup> , Aaron Horiuchi<sup>1</sup> , Kevin A. Hurd<sup>1</sup> , Dave Johnson<sup>1</sup> , Hugh McIntyre<sup>2</sup> , Samuel Naffziger<sup>1</sup> , James Vinh<sup>2</sup> , Jonathan White<sup>4</sup> , Kathryn Wilcox, Design Solutions for the Bulldozer 32nm SOI 2-Core Processor Module in an 8-Core CPU, ISSCC 2011 より

# ARM Cortex-A15 の消費電力の割合



- NVIDIA Tegra 4 Family CPU Architecture より

# マイクロ命令への分解の他の利点

- CPU にバグがあったときに、後からパッチが当てられる
  - ◊ バグを「エラッタ」とも呼ぶ
- 動作がおかしい命令を、他のバグってない命令の列で置き換える
  - ◊ 分解に使う表は、あとから書き換えられるようになっている
  - ◊ Microcode ROM というのがそれ

# インテルの Core シリーズのエラッタのリスト 地味に結構バグってる



## Errata

### SKZ1 A CAP Error While Entering Package C6 May Cause DRAM to Fail to Enter Self-Refresh

**Problem:** A CAP (Command/Address Parity) error that occurs on the command to direct DRAM to enter self-refresh may cause the DRAM to fail to enter self-refresh although the processor enters Package-C6.

**Implication:** Due to this erratum, DRAM may fail to be refreshed, which may result in uncorrected errors being reported from the DRAM.

**Workaround:** None Identified.

**Status:** For the Steppings affected, refer the *Summary Tables of Changes*.

### SKZ2 PCIe® Lane Error Status Register May Log False Correctable Error

**Problem:** Due to this erratum, PCIe® LNERRSTS (Device 0; Function 0; Offset 258h; bits [3:0]) may log false lane-based correctable errors.

**Implication:** Diagnostics cannot reliably use LNERRSTS to report correctable errors.

**Workaround:** None Identified

**Status:** For the Steppings affected, refer the *Summary Tables of Changes*.

### SKZ3 In Memory Mirror Mode, DataErrorChunk Field May be Incorrect

**Problem:** In Memory Mirror Mode, DataErrorChunk bits (IA32\_MC7\_MISC register MSR(41FH) bits [61:60]) may not correctly report the chunk containing an error.

**Implication:** Due to this erratum, this field is not accurate when Memory Mirror Mode is enabled.

**Workaround:** None Identified.

**Status:** For the Steppings affected, refer the *Summary Tables of Changes*.

### SKZ4 Intel® RDT MBM Does Not Accurately Track Write Bandwidth

**Problem:** Intel® RDT (Resource Director Technology) MBM (Memory Bandwidth Monitoring) does not count cacheable write-back traffic to local memory. This will result in the RDT MBM feature under counting total bandwidth consumed.

**Implication:** Applications using this feature may report incorrect memory bandwidth.

**Workaround:** None Identified.

**Status:** For the Steppings affected, refer the *Summary Tables of Changes*.

### SKZ5 PCIe® Port May Incorrectly Log Malformed\_TLP Error

**Problem:** If the PCIe port receives a TLP that triggers both a Malformed\_TLP error and an ECRC\_TLP error, the processor should only log an ECRC\_TLP error. However, the processor logs both errors.

**Implication:** Due to this erratum, the processor may incorrectly log Malformed\_TLP errors.

**Workaround:** None Identified

**Status:** For the Steppings affected, refer the *Summary Tables of Changes*.

### SKZ6 Short Loops Which Use AH/BH/CH/DH Registers May Cause Unpredictable System Behavior

**Problem:** Under complex micro-architectural conditions, short loops of less than 64 instructions that use AH, BH, CH or DH registers as well as their corresponding wider register (e.g. RAX, EAX or AX for AH) may cause unpredictable system behavior. This can only happen when both logical processors on the same physical processor are active.

**Implication:** Due to this erratum, the system may experience unpredictable system behavior

Number	Steppings		Status	Errata
	U-0	M-0		
SKZ10	X	X	No Fix	With eMCA2 Enabled a 3-Strike May Cause an Unnecessary CATERR# Instead of Only MSMI
SKZ11	X	X	No Fix	CMCI May Not be Signaled for Corrected Error
SKZ12	X	X	No Fix	CSRs SVID And SDID Are Not Implemented For Some DDRIO And PCU devices
SKZ13	X	X	No Fix	Register Broadcast Read From DDRIO May Return a Zero Value
SKZ14	X	X	No Fix	Intel® CMT Counters May Not Count Accurately
SKZ15	X	X	No Fix	Intel® CAT May Not Restrict CacheLine Allocation Under Certain Conditions
SKZ16	X	X	No Fix	Intel® PCIe® Corrected Error Threshold Does Not Consider Overflow Count When Incrementing Error Counter
SKZ17	X	X	No Fix	IIO RAS VPP Hangs During The Warm Reset Test
SKZ18	X	X	No Fix	Processor May Hang on Complex Sequence of Conditions
SKZ19	X	X	No Fix	Intel® PCIe® Root Port Electromechanical Interlock Control Register Can Be Written
SKZ20	X	X	No Fix	System Hangs May Occur When IPQ And IRQ Requests Happen At The Same Time
SKZ21	X	X	No Fix	Masked Bytes in a Vector Masked Store Instructions May Cause Write Back of a Cache Line
SKZ22	X	X	No Fix	ERROR_N[2:0] Pins May Not be Cleared After a Warm Reset
SKZ23	X	X	No Fix	Intel® PCIe® Slot Presence Detect And Presence Detect Changed Logic Not PCIe® Specification Compliant
SKZ24	X	X	No Fix	Debug Exceptions May Be Lost or Misreported When MOV SS or POP SS Instruction is Not Followed By a Write to SP
SKZ25	X	X	No Fix	Incorrect Branch Predicted Bit in BTS/BTM Branch Records
SKZ26	X	X	No Fix	DR6_B0-B3 May Not Report All Breakpoints Matched When a MOV/POP SS is Followed by a Store or an MMX Instruction
SKZ27	X	X	No Fix	Intel® PT TIP.PGD May Not Have Target IP Payload
SKZ28	X	X	No Fix	The Corrected Error Count Overflow Bit in IA32_MCO_STATUS is Not Updated When The UC Bit is Set
SKZ29	X	X	No Fix	SMRAM State-Save Area Above the 4GB Boundary May Cause Unpredictable System Behavior
SKZ30	X	X	No Fix	VM Exit May Set IA32_EFER.NXE When IA32_MISC_ENABLE Bit 34 is Set to 1
SKZ31	X	X	No Fix	x87 FPU Exception (#MF) May Be Signaled Earlier Than Expected
SKZ32	X	X	No Fix	POPCNT Instruction May Take Longer to Execute Than Expected
SKZ33	X	X	No Fix	Load Latency Performance Monitoring Facility May Stop Counting
SKZ34	X	X	No Fix	Intel® Processor Trace PSB+ Packets May Contain Unexpected Packets
SKZ35	X	X	No Fix	Performance Monitoring Counters May Undercount When Using CPL Filtering
SKZ36	X	X	No Fix	Intel® PT ToPA PMI Does Not Freeze Performance Monitoring Counters
SKZ37	X	X	No Fix	Performance Monitoring Load Latency Events May Be Inaccurate For Gather Instructions
SKZ38	X	X	No Fix	CPUID TLB Associativity Information is Inaccurate
SKZ39	X	X	No Fix	Vector Masked Store Instructions May Cause Write Back of Cache Line Where Bytes Are Masked
SKZ40	X	X	No Fix	Incorrect FROM_IP Value For an RTM Abort in BTM or BTS May be Observed
SKZ41	X	X	No Fix	PEBS Record After a WRMSR to IA32 BIOS_UPDT_TRIG May be Incorrect
SKZ42	X	X	No Fix	MOVNTDQA From WC Memory May Pass Earlier Locked Instructions
SKZ43	X	X	No Fix	#GP on Segment Selector Descriptor that Straddles Canonical Boundary May Not Provide Correct Exception Error Code
SKZ44	X	X	No Fix	Intel® PT OVF Packet May be Lost If Immediately Preceding a TraceStop

# Windows Update でこっそり更新されてたりもする

## Intel 製マイクロコードの更新プログラムの概要

適用対象: Windows Server 2019, all versions, Windows 10, version 1809, Windows 10, version 1803, 詳細

### Intel 製マイクロコードの更新プログラム

マイクロソフトは、スペクター バリアント 2 (CVE 2017-5715 ["ブランチ ターゲット インジェクション"]) に関する Intel による検証済みのマイクロコードの更新プログラムをリリースしています。

次の表は、Windows バージョン別のサポート技術情報一覧です。サポート技術情報には、リリースされている Intel 製マイクロコードの更新プログラムが CPU 別に記載されています。

サポート技術情報番号と説明	Windows のバージョン	Source
KB4100347 Intel 製マイクロコードの更新プログラム	Windows 10 Version 1803、Windows Server Version 1803	Windows Update、Windows Server Update Services、Microsoft Update カタログ
KB4090007 Intel 製マイクロコードの更新プログラム	Windows 10 Version 1709 および Windows Server 2016 Version 1709	Windows Update、Windows Server Update Services、Microsoft Update カタログ
KB4091663 Intel 製マイクロコードの更新プログラム	Windows 10, version 1703	Windows Update、Windows Server Update Services、Microsoft Update カタログ

- <https://support.microsoft.com/ja-jp/help/4093836/summary-of-intel-microcode-updates> より

# 余談：命令の歴史

- RISC-V や MIPS などでは、パイプライン実行を最初から想定
  - ◊ 小さいハードで構造ハザードが起きにくいよう設計されている
  - ◊ RISC (Reduced Instruction Set Computer) と呼ばれる
  - ◊ RISC-V は、「5代目の RISC」という名前
- x86 は、登場時はパイプライン化を考えていなかった
  - ◊ 当時は、小数の命令でたくさんのことができるのが正義
  - ◊ しかし、そのままではパイプライン化は困難
  - ◊ CISC (Complex Instruction Set Computer) と呼ばれる
- ARM の R は RISC の R なのだが、ARM は結構 CISC ぽい
  - ◊ ARM : Advanced RISC Machine
  - ◊ 構造ハザードを凄い勢いで起こす命令が多い

# 余談：命令の歴史

- マイクロ命令への分解により、x86 や ARM はこの問題を（一応）克服
  - ◊ 回路規模やエネルギーにおける代償は大きい
  - ◊ 互換性が維持できるので、商業上重要
- x86 や ARM は、64bit バージョンを作る際に命令の内容をかなり整理した
  - ◊ パイプラインが作りやすくなっている
  - ◊ 富岳では ARM 32bit を切り捨てており（多分）、大分楽になっているはず

# 構造ハザードのまとめ

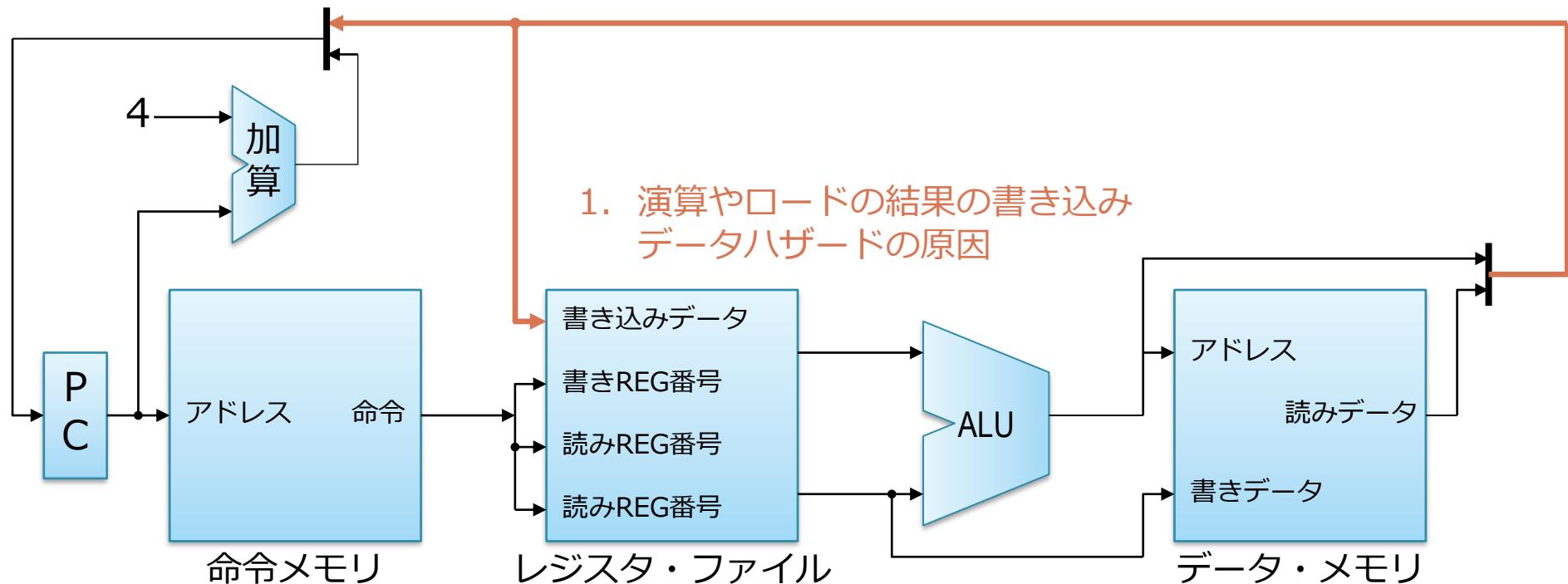
- 構造ハザード：ハード資源の不足に起因
- 解決方法
  - 1. ハードウェアの増強
  - 2. 時分割処理
  - 3. マイクロ命令への変換
- パイプライン・ストール

# ハザード

1. 構造ハザード
2. 非構造ハザード：バックエッジに由来
  - a. データ・ハザード
  - b. 制御ハザード

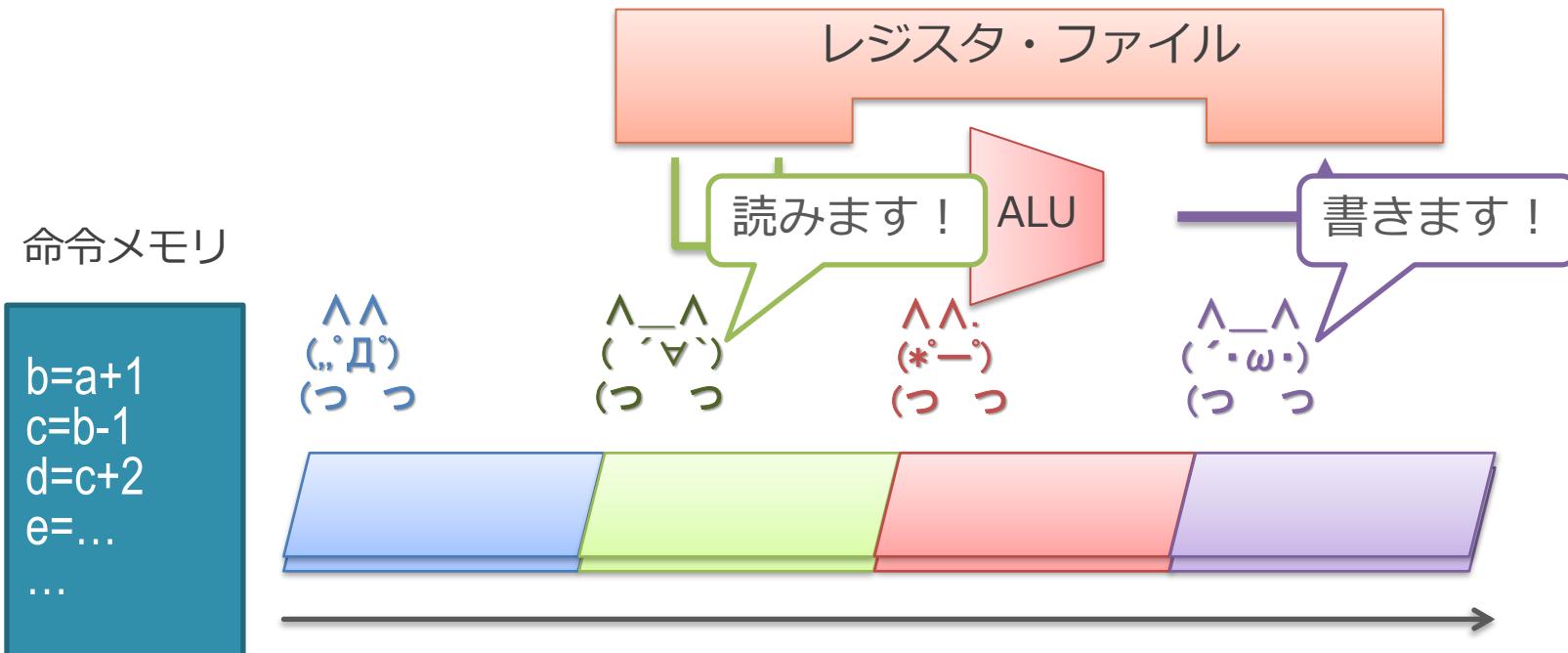
# バックエッジ：逆方向（右から左）にいく信号

2. 分岐結果の PC への反映  
制御ハザードの原因



- バックエッジがあるため、命令を単純に流せない場合がある
  - ◊ 工場のラインのように、一方向に流せない

# データ・ハザード



#### ■ レジスタ・ファイルへのアクセス

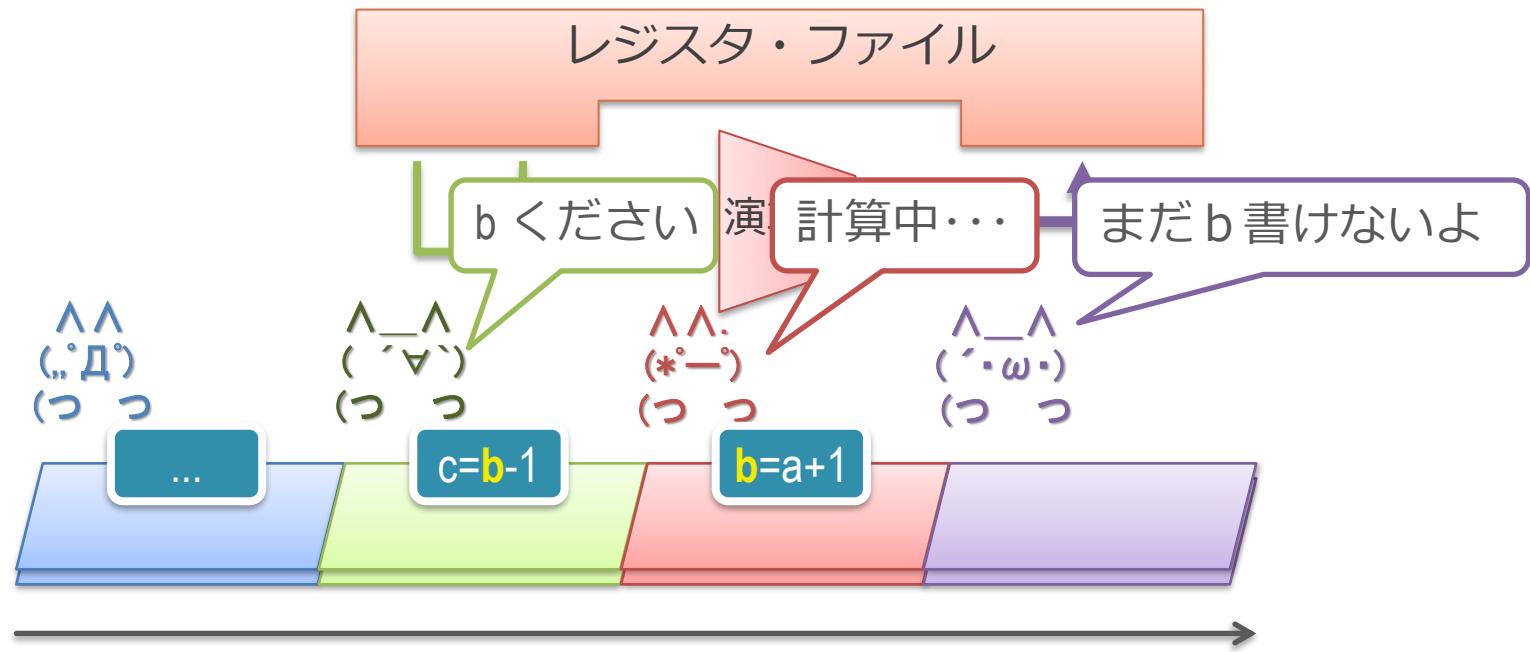
- ◇ 演算の入力は(‘ $\forall$ ’) の人がレジスタ・ファイルから読み出す
  - ◇ 演算の結果は(‘ $\cdot\omega\cdot$ ’) の人がレジスタ・ファイルに書き込む

# データ・ハザード

命令メモリ  
(プログラム)

**b=a+1**  
**c=b-1**

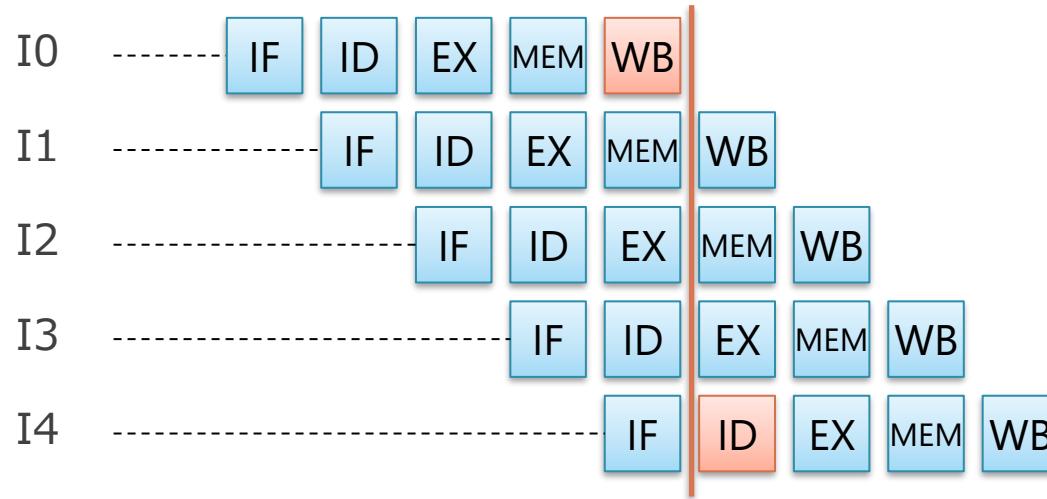
...  
...  
...



■ 直前の命令の結果を使う命令が現れた場合 :

- ◊ ( '∀' ) の人が **b=a+1** の結果を読もうとしても,
- ◊ (\*°—°) の人がまだ計算中でレジスタ・ファイルに b が書けていない
- ◊ ( '・ω・' ) の人が計算結果をかけるのはさらに次のサイクル

# データ・ハザード



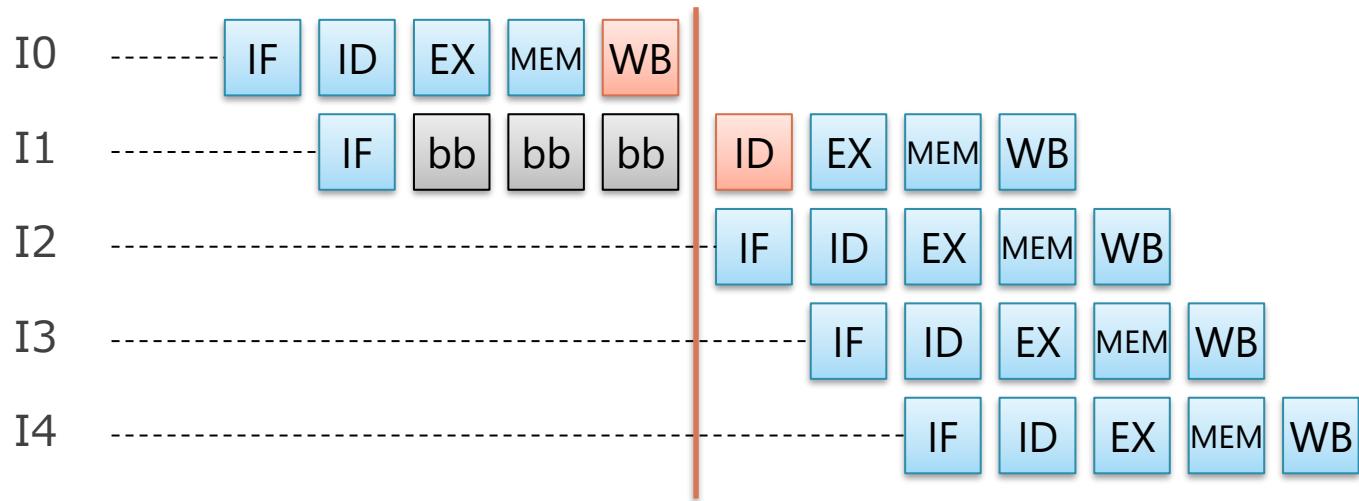
- I0 の WB が終わるまで、その結果はレジスタに書き込まれない
  - ◊ I4 までは、その値がレジスタから得られない
  - ◊ ID ステージでレジスタを読むため

# データ・ハザードの解消方法

## ■ 解消方法

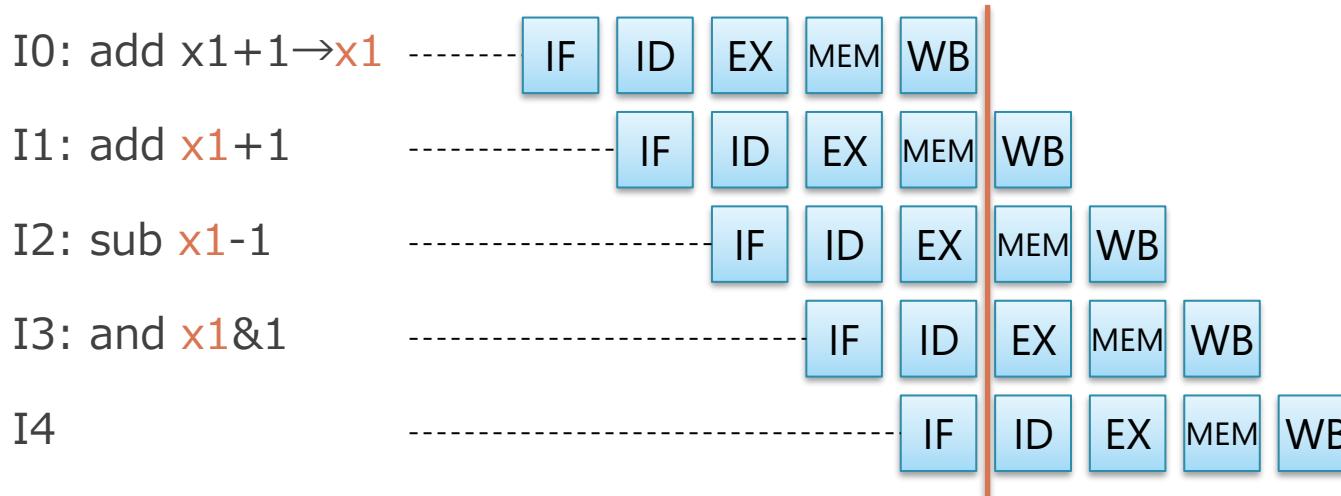
1. ストールさせる
2. 遅延スロット（なにもしない）
3. フォワーディング
4. マルチスレッディング

# 1. ストールさせる



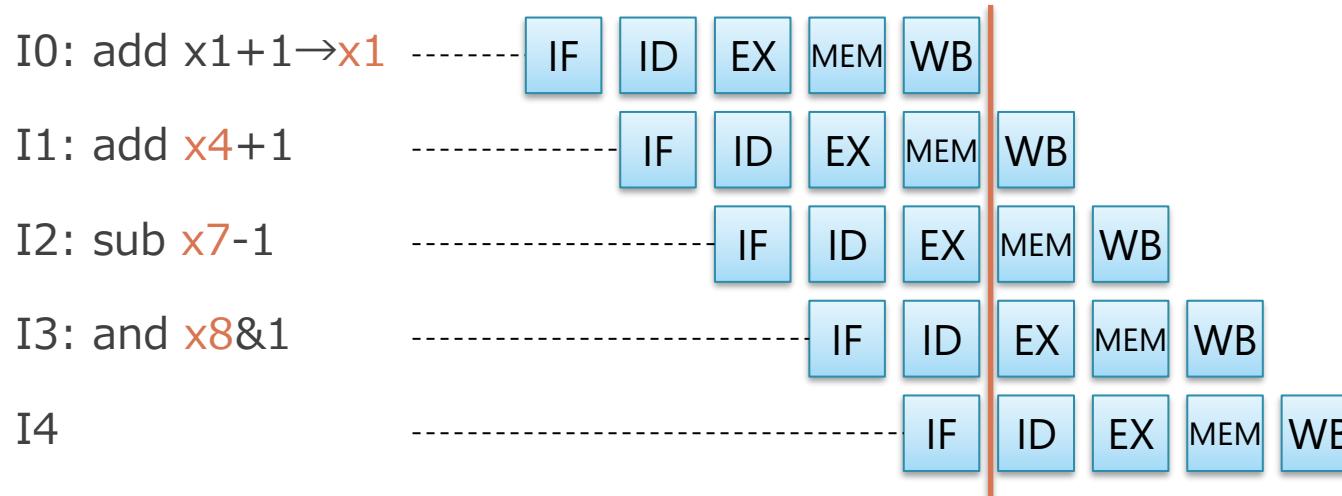
- I0 の WB が終わるまで、後続の命令を遅らせる
  - ◊ I1 の ID が、I0 の WB の右にくるまでストール
  - ◊ I1 は I0 の結果を使える
- 欠点：とても遅くなる

## 2. 遅延スロット (なにもしない)



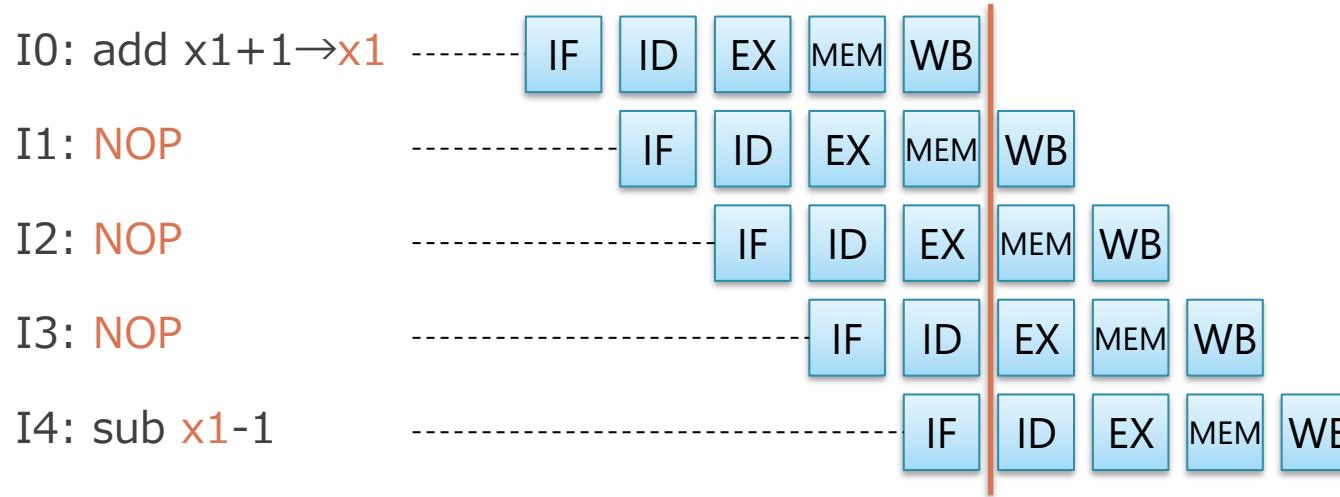
- 特になにも対策せず,  
「ある命令の結果は、数命令先まで見えない」と言う仕様にする
  - ◊ 上の例だと I1, I2, I3 は、I0 の結果は見えない
  - ◊ I1, I2, I3 には、I0 で add する前の値が見え続ける

## 2. 遅延スロット (なにもしない)



- ここに I0 の結果を使わない命令を入れれば、性能低下はない
  - ◊ この部分を「遅延スロット」と呼ぶ
  - ◊ この場合、遅延スロットが 3 命令分ある
  - ◊ コンパイラががんばって入れる
  - ◊ 人力でアセンブリ言語で頑張ることもある

# NOP の挿入



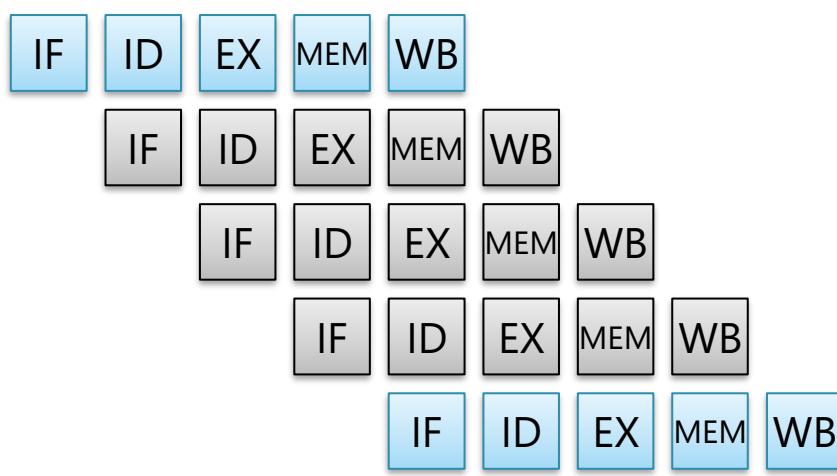
- もしそのような命令がない場合、
  - ◊ NOP (No Operation) と呼ぶ何もしない命令をいれる
  - ◊ これもコンパイル時にいれておく必要がある
- 上の例は、 $x_1$  に 1 を足した結果を使う以外の処理がなかった場合

# 遅延スロットの利点

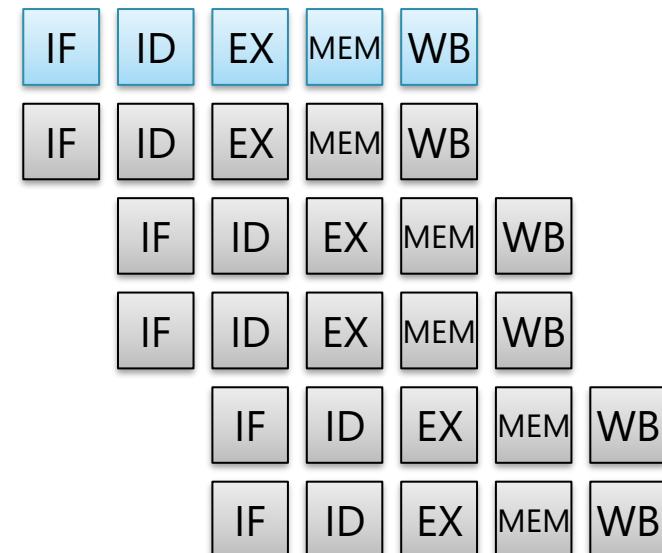
- 利点：
  - ◊ なにもしないので、ハードは最も単純
  - ◊ 並列にできる命令があれば、性能も下がらない

# 遅延スロットの欠点

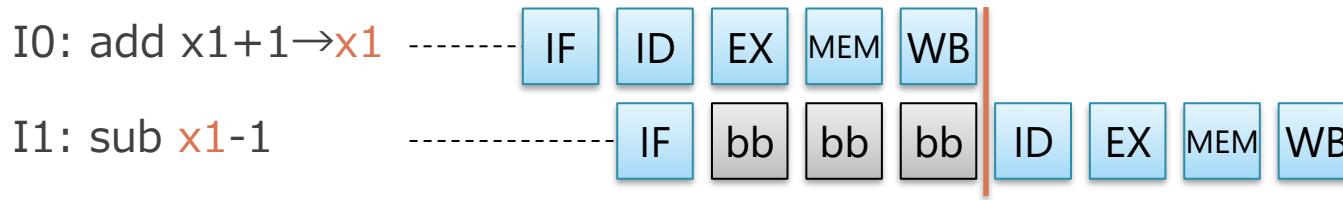
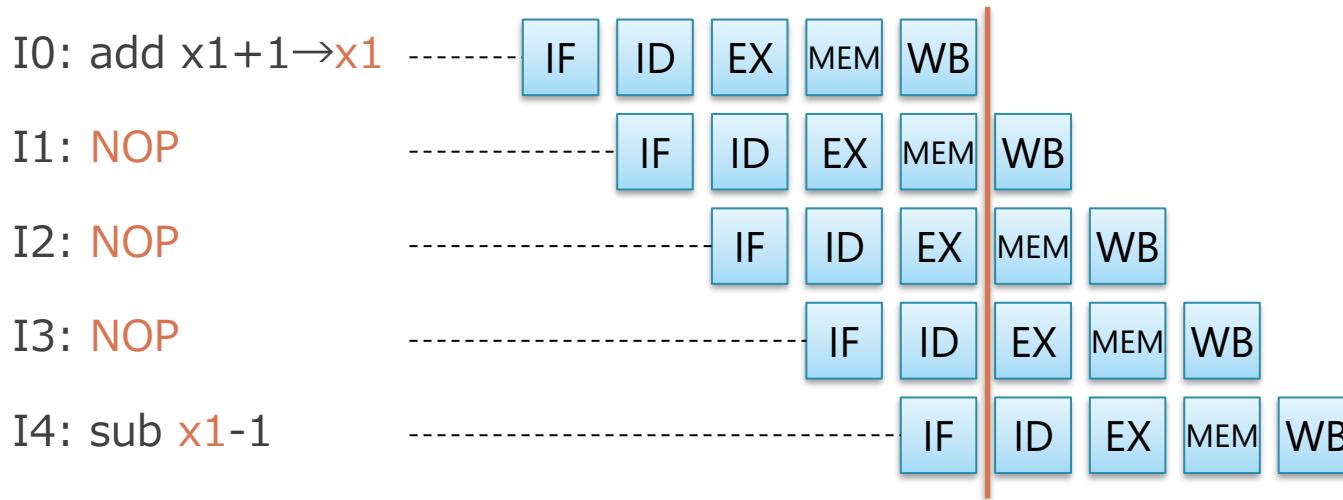
- 欠点：「仕様」なので、一度決めると変えられない
  - ◊ 後からパイプラインの段数を変えると互換性がなくなる
    - クロックをあげるために、段数を増やせない
  - ◊ 複数の命令を同時処理しようとしたときにも互換性がなくなる
  - ◊ MIPS では遅延スロットが 1 命令分、仕様として存在
    - 互換性のためにこれを忠実に再現するため後年は逆に複雑化



2 命令同時処理すると、遅延スロットが増える



# 遅延スロットの欠点 2



- 欠点 2 : 並列してできる命令が常にあるとは限らない
  - ◊ NOP を入れるしかなくなる
  - ◊ 実質ストールしてバブルを入れるのと同じになってしまう

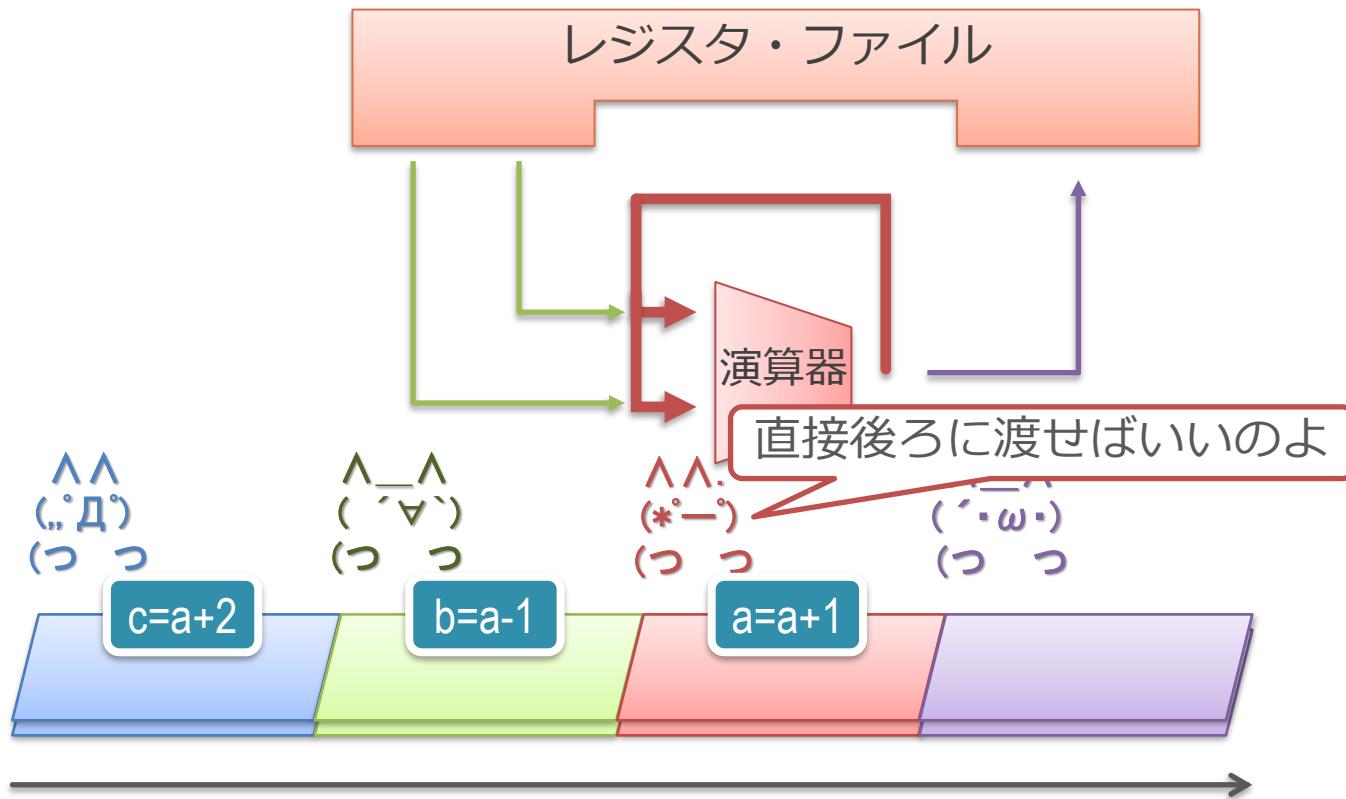
# データ・ハザードの解消方法

## ■ 解消方法

1. ストールさせる
2. 遅延スロット（なにもしない）
3. フォワーディング
4. マルチスレッディング

# フォワーディング

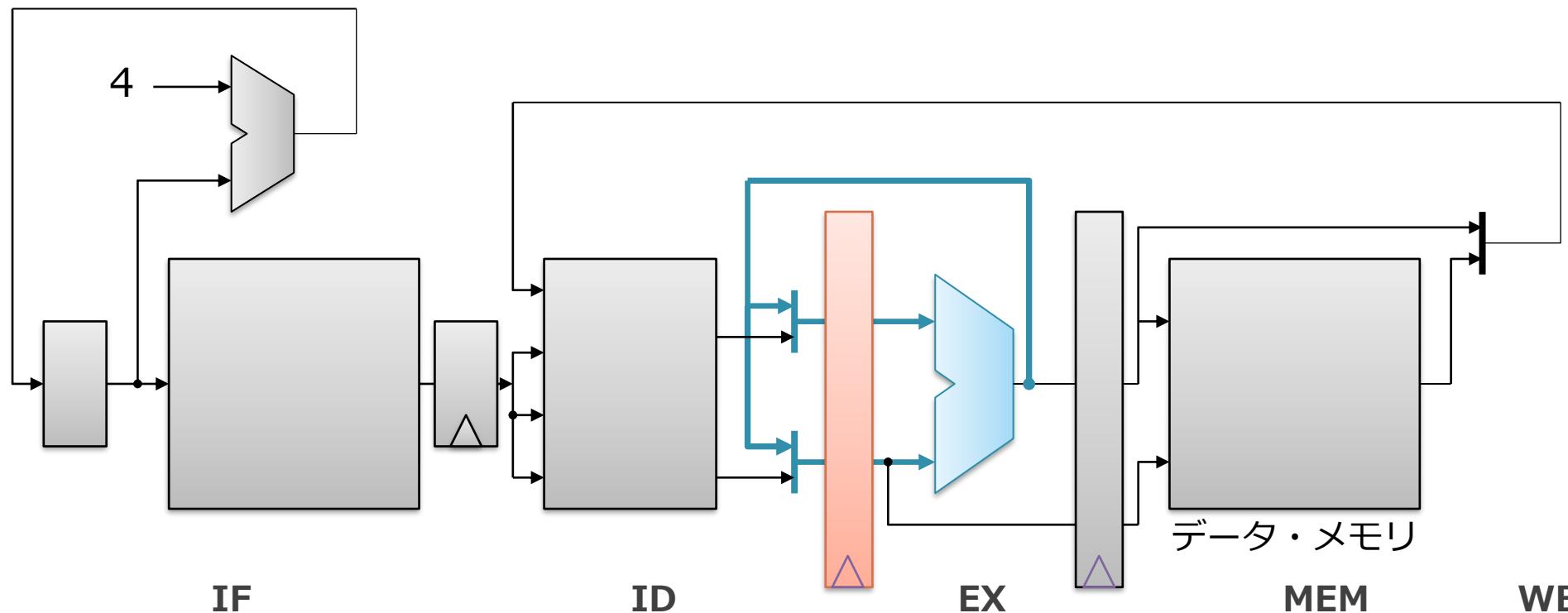
```
a=a+1  
b=a-1  
c=a+2  
e=...  
...
```



## ■ フォワーディング（バイパスとも呼ぶ）

- ◊  $(\cdot\circ\cdot)$  の人が、次のサイクルにも結果を使えるようレジスタに書くと同時に手元に結果をおいておく

# フォワーディングの回路



- 演算器の結果を、フィードバック
  - ◊ レジスタ・ファイルからの読み出し結果と選択して入力に

# フォワーディングの利点

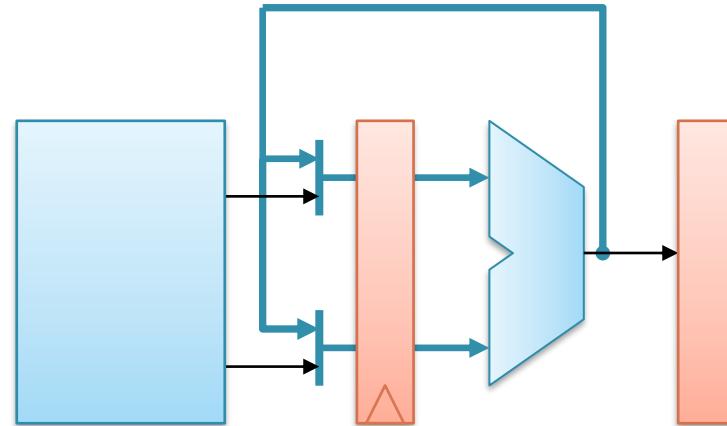
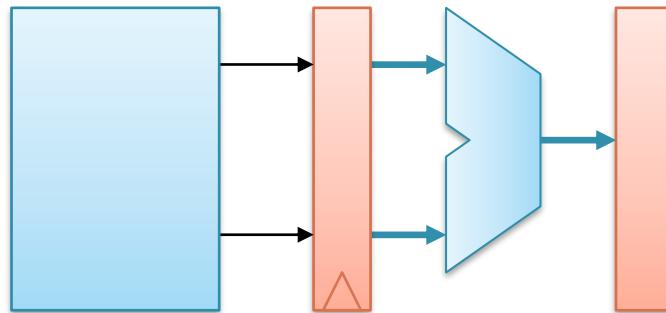
- 利点：
  - ◊ 依存関係がある命令が連續できてもパイプラインを動かし続けられる
  - ◊ バブルを発生させることがない

# フォワーディングの問題



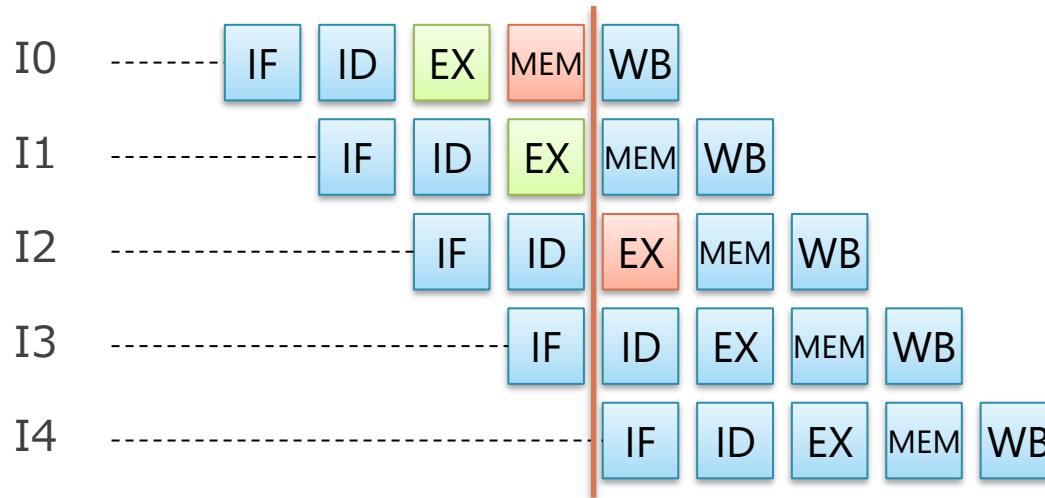
- 演算器とフォワーディングを含むステージは1サイクルで処理する必要がある
  - ◊ 分割すると、バブルを入れてるので同じになってしまう
  - ◊ できればここはパイプライン化したくない
    - (FP 演算等の複雑なものは、やむなくパイプライン化している)
- CPU 全体のクリティカル・パスになりやすい
  - ◊ クロック周波数がここで決まることが多い

# フォワーディングの問題



- フォワーディングは、この演算器の部分の遅延を増やしてしまう
  - ◊ クロック周波数の低下につながる

# ロードについては、完全に解決はできない



- ロードではデータ・メモリを読むまでその値は取れない
  - ◊ 次の命令は、MEM より後に EX がこないといけない
    - I1 は、I0 のロード結果が見えない
  - ◊ この部分はストールや遅延スロットでなんとかすることがおおい

# データ・ハザードの解消方法

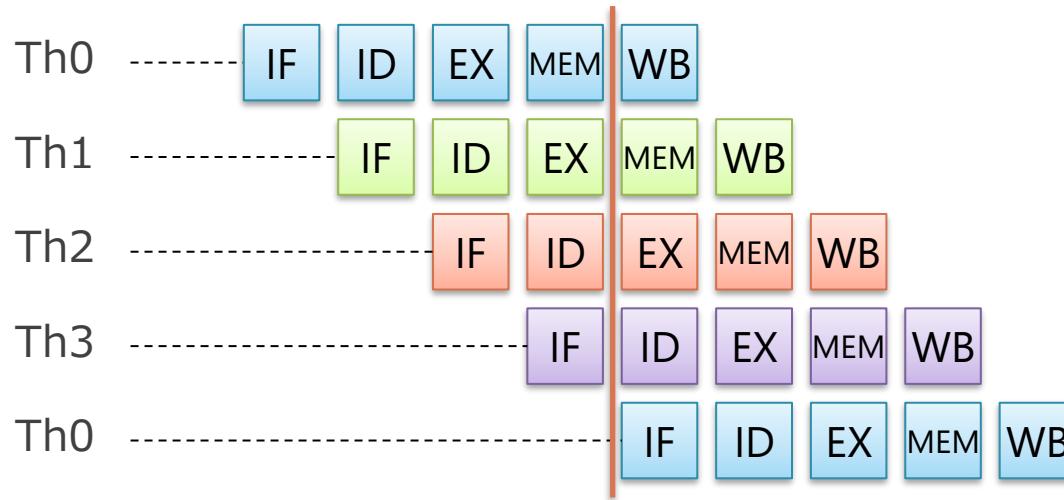
## ■ 解消方法

1. ストールさせる
2. 遅延スロット（なにもしない）
3. フォワーディング
4. マルチスレッディング

# マルチスレッディング

- 広義のマルチスレッド：
  - ◊ コンテキスト（PC やレジスタ）を複数持つこと
- ソフトウェアにおけるマルチスレッド
  - ◊ pthread とか
  - ◊ 複数のコンテキストが並列して動作
- ハードウェアのマルチスレッド
  - ◊ ひとつの CPU 内に複数のコンテキストを複数持つ
  - ◊ 次ページの方法は「細粒度マルチスレッディング」と呼ぶ
    - ハードウェアのマルチスレッドは、他にもいろいろある

# マルチスレッディング



- Th0 から Th3 までの4つのスレッドの命令を順に実行
  - ◊ 各スレッドは独立しているので、お互いの結果を読むことはない
- Th0 に戻ってくる頃には、前回の Th0 の結果が書き込まれている

# マルチスレッディングの利点と欠点

- 利点：
  - ◊ 他の方法のような問題が起きない
    - 理想的にはバブルも発生せず、クロックも落ちない
  - ◊ 演算器をパイプライン化しても性能に影響がない
    - 他のスレッドを実行して時間をつぶしていればよい
- 欠点：
  1. 動かすスレッドがない場合は、止めておくしかない
    - GPU 等ではスレッドが大量にあるので、問題とならない
    - GPU ではループの各周がスレッドになっている
  2. スレッド数分のレジスタを持つ必要があるのでハードが大きい

# データ・ハザードのまとめ

## ■ 解消方法

1. ストールさせる
2. 遅延スロット（なにもしない）
3. フォワーディング
4. マルチスレッディング

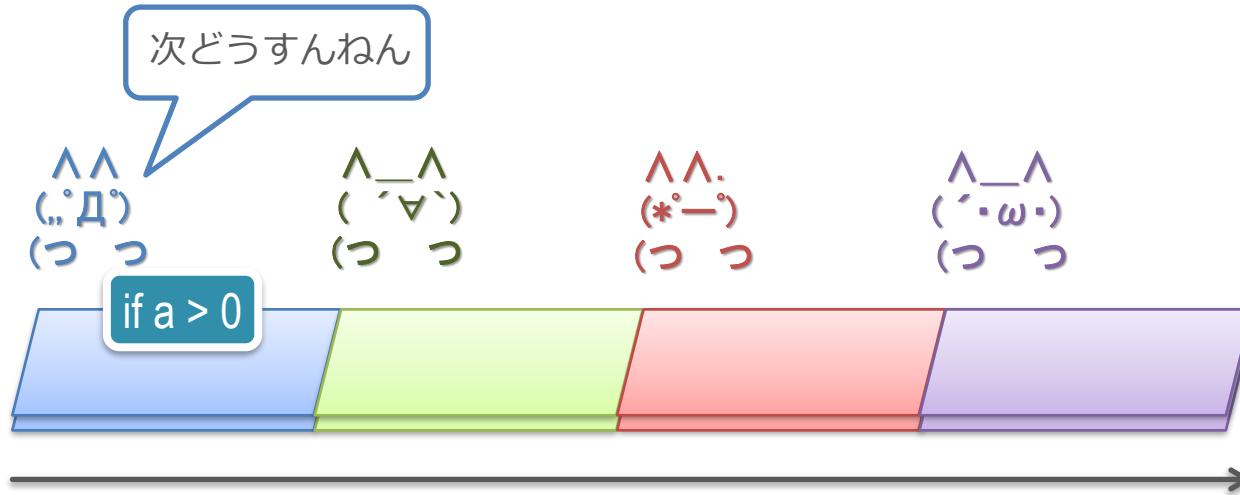
# ハザード

1. 構造ハザード
2. 非構造ハザード：バックエッジに由来
  - a. データ・ハザード
  - b. 制御ハザード

# 分岐命令の処理と制御ハザード

命令メモリ

```
if a > 0:  
    a=a+1  
else:  
    a=a-1
```



- 「`if a > 0`」の結果は最終段の(`'・ω・`)の人まで反映出来ない
  - ◊ 先頭は次に `a=a+1` と `a=a-1` のどちらを取り込めばいいのかわからない

# 制御ハザードの解消方法

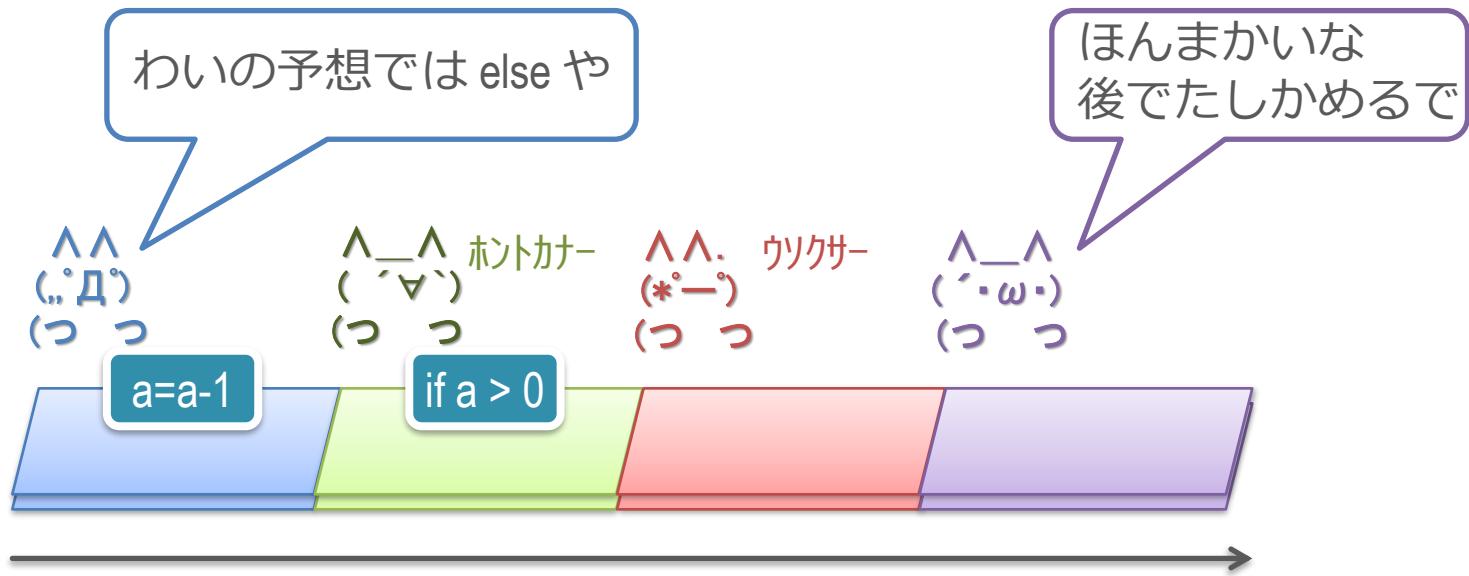
## ■ 解消方法

1. ストールさせる
2. 遅延スロット（なにもしない）
3. マルチスレッディング
  - 上記は、基本的にデータ・ハザードと同様にして適用できる
  - フォワーディングは制御ハザードでは意味的に無理
4. 分岐予測による投機実行

# 分岐予測

プログラム

```
if a > 0:  
    a=a+1  
else:  
    a=a-1
```



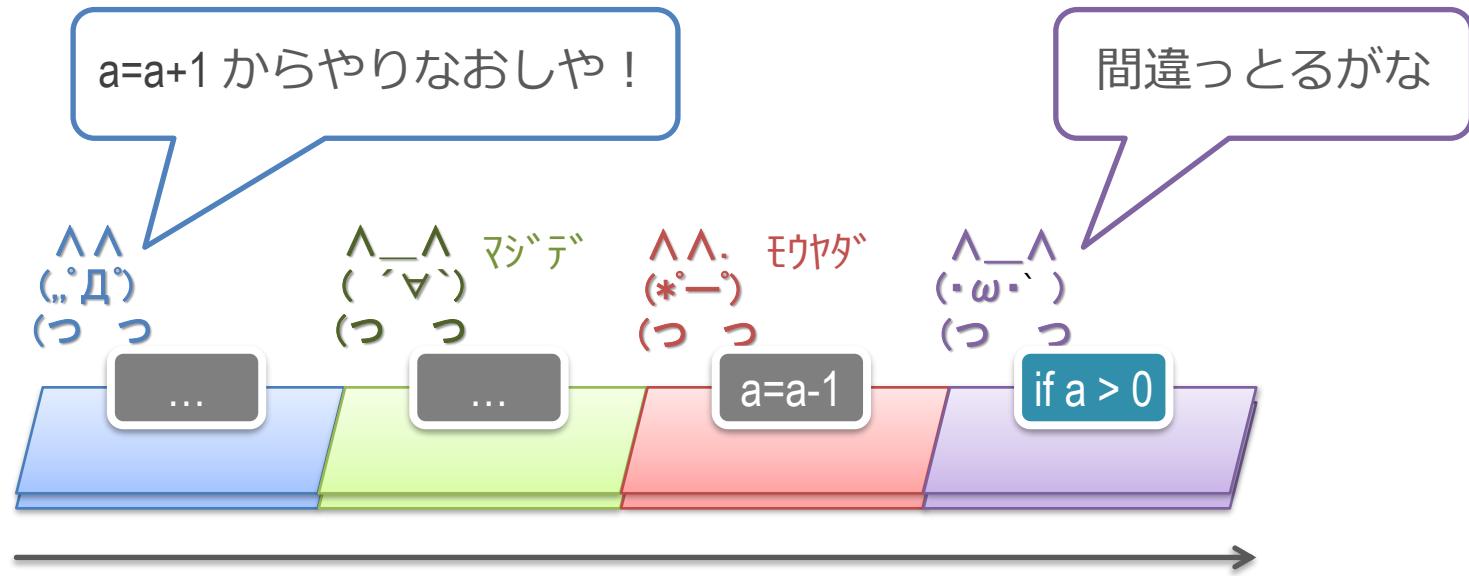
## ■ 動作

- ◇ 「`if a > 0`」の結果を予測して、命令を取り込む
  - 前回はこっちに行ったので、次もこっちに違いないとかで予測
- ◇ あとから予測が正しいか確認する

# 分岐予測ペナルティ

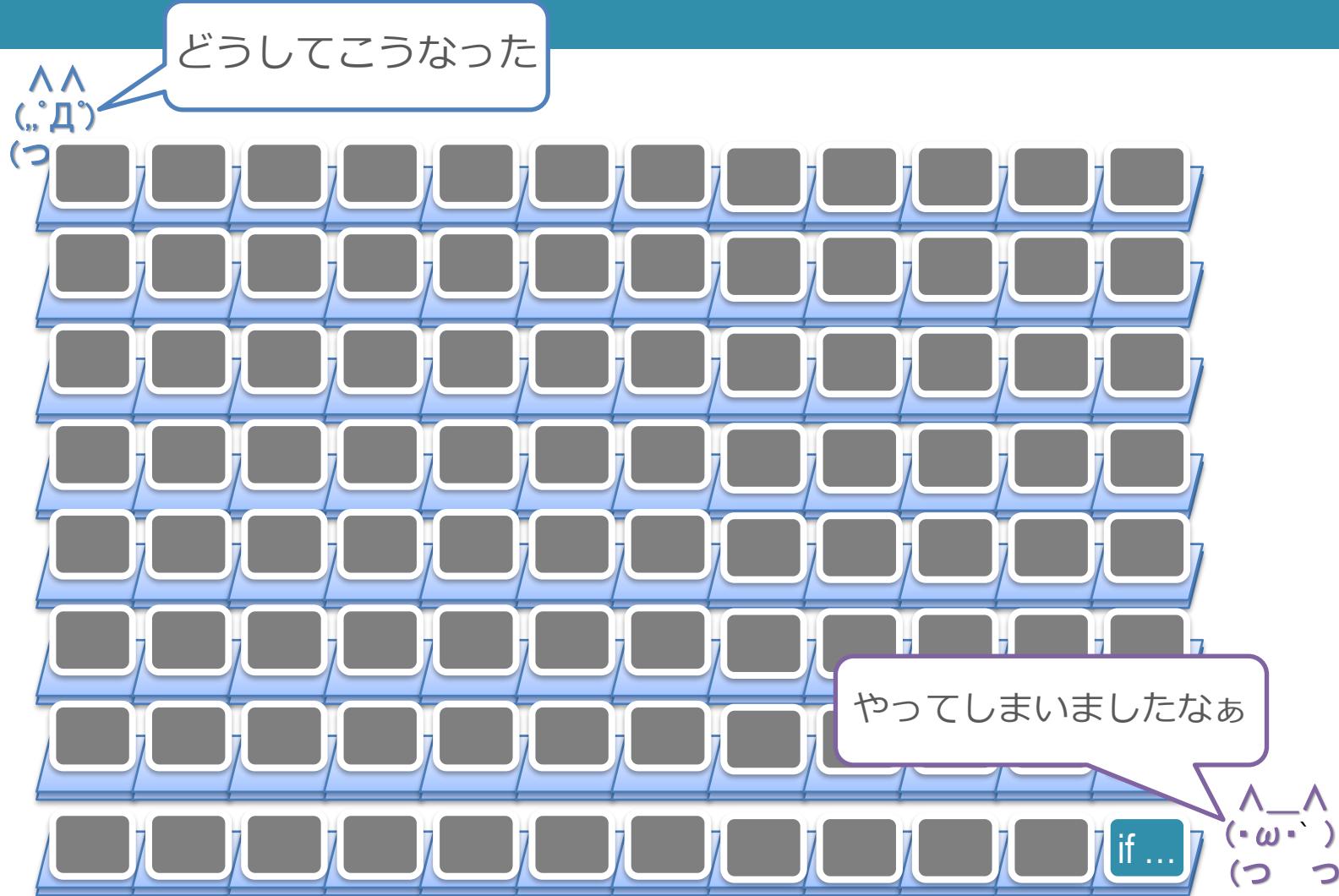
プログラム

```
if a > 0:  
    a=a+1  
else:  
    a=a-1
```



- 予測が間違っていた場合、以降の処理を取り消してやり直す
- この図では、無駄になるのは3命令分

# 大規模な高性能プロセッサの場合



- 取り消しは最悪数十命令以上に
  - ◊ IBM POWER8 だと、8命令同時 × 10数段

# 今日のまとめ

- パイプライン
- 各種のハザードと解消方法
  - ◇ 構造ハザード
  - ◇ 非構造ハザード
    - データ・ハザード
    - 制御ハザード
- 来週は分岐予測の具体的な方法など

# 出欠と感想

- 本日の講義でよくわかったところ、わからなかつたところ、質問、感想などを書いてください
  - ◊ 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:
- 意見や内容へのリクエストもあつたら書いてください
- LMS の出席の締め切りは来週の講義開始までに設定してあります
  - ◊ 仕様上「遅刻」表示になりますが、特に減点等しません
  - ◊ 来週の講義開始までは感想や質問などを受け付けます