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

質問とか回答など

- 一つだけ質問があり、ステージでIDはinstruction decodeというのはわかったのですが、WBは何の略でしょうか？"
- H-treeは例のように正方形が最もノードを多くすることができると思うのですが、実際の基盤のように端にクロックがある場合はどのような形状にしているのですか？例えば、二次元ではなく三次元的に配線の距離を調整しているのですか？"

質問とか回答など

- "ステージの区切りについて、デコードにほとんど時間がかからない場合が例になっていましたが、デコードに時間がかかるような命令セットの場合、各ステージにかかる時間の調整はどのように行っているのでしょうか？今回の例を見ていると、微調整はなかなか難しそうに感じました。"

- 工学・産業界でFPGAが盛んに取り入れられているのはどのような分野でしょうか？

質問とか回答など

- 質問なのですが，クロックの消費電力が大きくなる一因として「毎サイクルCPU全体をクロック同期する」ことが挙げられていましたが，CPU全体をクロック同期する必要がない例や部分的に同期を外すことで消費電力を抑える手法などは存在するのでしょうか.

質問とか回答など

- あまりFPGAに明るくないのですが、FPGA自体の構成のアルゴリズムを改善？することによりFPGAで作成した回路の効率化もできるのかなと考えました。そういった研究などあれば教えていただきたいです。

- 今回の講義の範囲ではないでしょうが、今後GPUのアーキテクチャについての話が聞けたらありがたいです。

質問とか回答など

- FPUにこれだけ大きな面積・消費電力がかかるのは知らなかったです。なんならALUで浮動小数点の演算もやっているのかと思っていました。何故そんなにFPUにトランジスタが必要なのか疑問に思いました。

- プログラミングをする際、パイプライン処理に適したコードの書き方はありますか？コンパイラの最適化に任せてしまっていいのでしょうか

質問とか回答など

- FPUにこれだけ大きな面積・消費電力がかかるのは知らなかったです。なんならALUで浮動小数点の演算もやっているのかと思っていました。何故そんなにFPUにトランジスタが必要なのか疑問に思いました。

- プログラミングをする際、パイプライン処理に適したコードの書き方はありますか？コンパイラの最適化に任せてしまっていいのでしょうか

質問とか回答など

- 深層学習などでGPUを使っていると，入力データあたりの処理時間が徐々に短くなっていくのですが，なぜですか．ハザードの対策が関係あるのでしょうか．
- 一般に並列化されるのは演算だが、制御を並列にできるとどうなるだろうとふと思った。でも並列にできない部分のことをそもそも制御と呼んでいるような気もする

質問とか回答など

- 今はスライド中で何かしている人を絵で表現したいなら、いらすとやの素材が選択されているのが大半なので、いってしまえば時代錯誤的なAAをなぜ使っているのかが気になりました。
- モナーとかギコとかかなり懐かしいです。
- ムーアの法則はまだ成り立っているのでしょうか。成り立っているならいつまで続くのでしょうか。

質問とか回答など

- CPU, GPU, FPGAのそれぞれの特徴は知っていましたが，トレードオフのまとめは文献などでも読んだことがなかったので，今回の講義でこれらのハードウェアが出現してきたのが腑に落ちました.

余談：The 1st Instruction Prefetching Championship

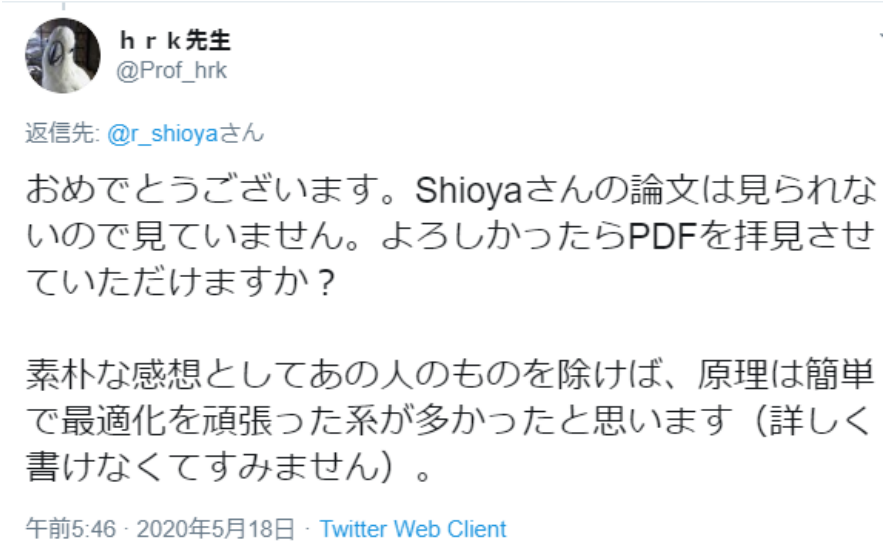
- 命令プリフェッチ：
  - 命令キャッシュ：
    - メイン・メモリは大きくて遅いので，その一部を保持する小型高速のキャッシュがついている
    - 命令を入れておくための専用のキャッシュ
  - プリフェッチ：
    - 将来アクセスされるアドレスを予測して，あらかじめキャッシュに先読みしておくこと

- The 1st Instruction Prefetching Championship
  - 命令プリフェッチのアルゴリズムの大会
  - https://research.ece.ncsu.edu/ipc/

The 1st Instruction Prefetching Championship

- ISCA の併設として開催
  - ◇ コンピュータ・アーキテクチャ分野で１番むずい国際会議の１つ
  - ◇ 提出するのは C++ の実装+論文で，査読有り
  - ◇ ６月初頭に発表

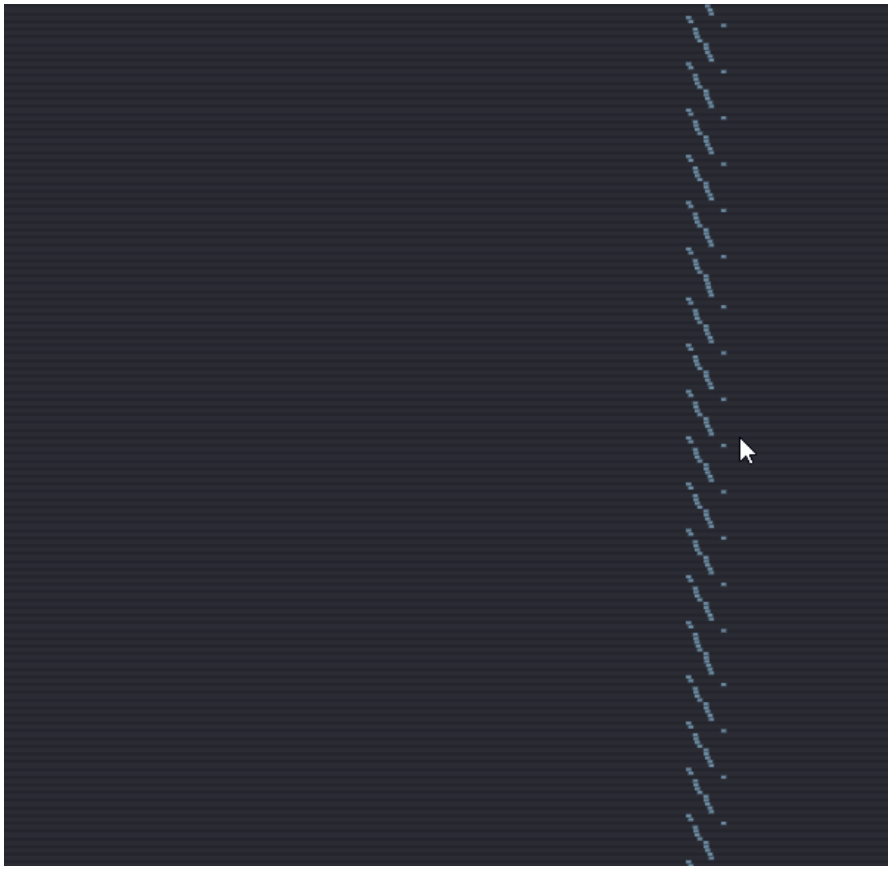
- 平木先生（元 コンピュータ科学&創造情報）もプログラム委員
  - ◇ ツイッターで書いたらリプがいただけた



D-JOLT prefetcher

- 塩谷は坂井入江研の学生さんらと組んで出場
  - D： 出川の D（妥協）
  - J：入江の I の次
  - O：中村の N の次
  - L：小泉の K の次
  - T：塩谷，坂井の S の次
- ４月から GW はこれをやっていたらほぼ潰れた
  - 内容はそのうち解説できたらしたい

余談：The 1st Instruction Prefetching Championship



- メモリアクセス可視化ツール「さざなみ」
  - ◇ これを使って色々解析

環境は公開されているので，みんなもやってみよう
（大会の募集はもう終わってますが）

The 1st Instruction Prefetching Championship

Home Organizers **Infrastructure** Rules & Criteria

Infrastructure

Framework

IPC1 will use the Champsim simulator, which was originally based off of the simulator used in the 2nd Data Prefetching Championship. Champsim is available at:

https://github.com/ChampSim/ChampSim

We also recommend that you join the Champsim Google Group for support and to provide feedback about the simulator:

https://groups.google.com/forum/#!forum/champsim

You can join the group by sending an empty mail to champsim+subscribe@googlegroups.com

Traces

IPC1 will use the traces available at:

https://drive.google.com/file/d/1qs8t8-YWc7ILoYbjbH_d3lf1xdoYBznf/view?usp=sharing

IPC1 will only evaluate single-core performance. Traces will run for 50 million warmup instructions, plus another 50 million evaluation instructions, or until the end of the trace.

In addition to the public trace list, your instruction prefetcher will also be evaluated with a secret trace list.

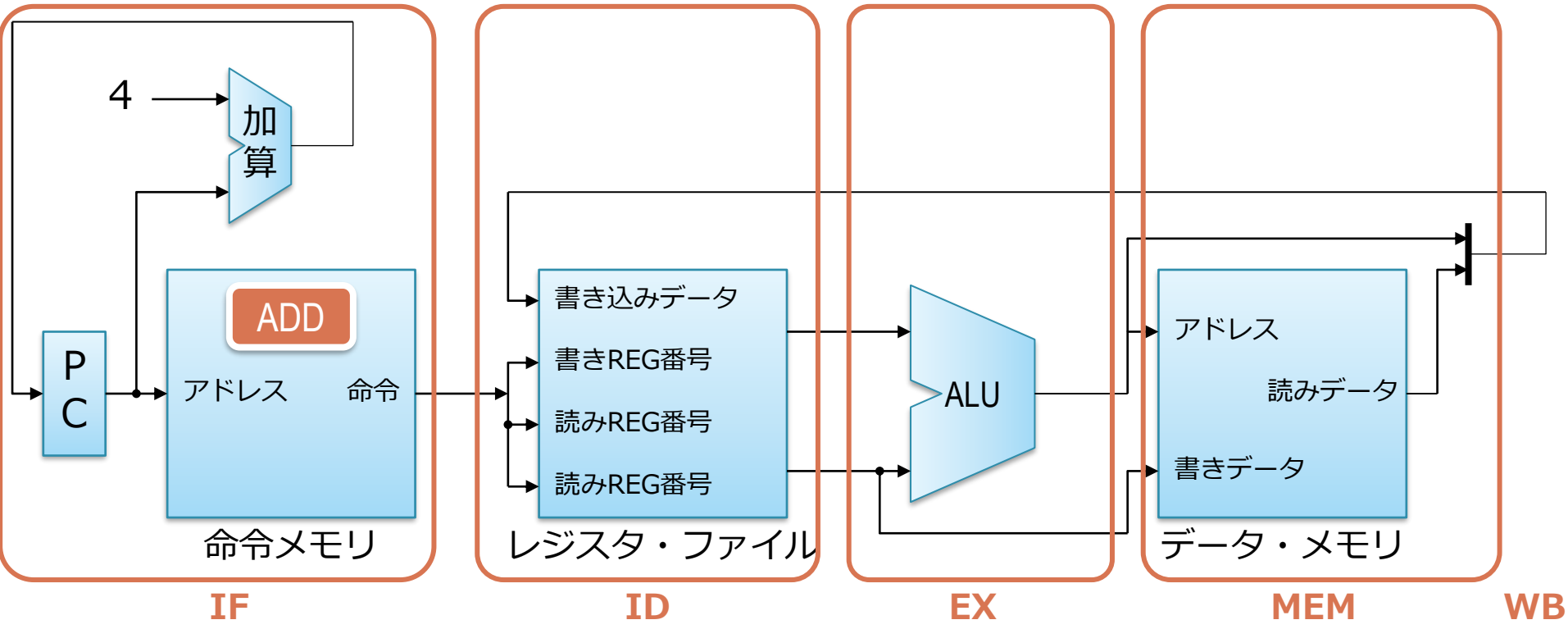
前回の内容

1. 回路の消費エネルギー
2. 命令パイプラインの基礎

今日の内容

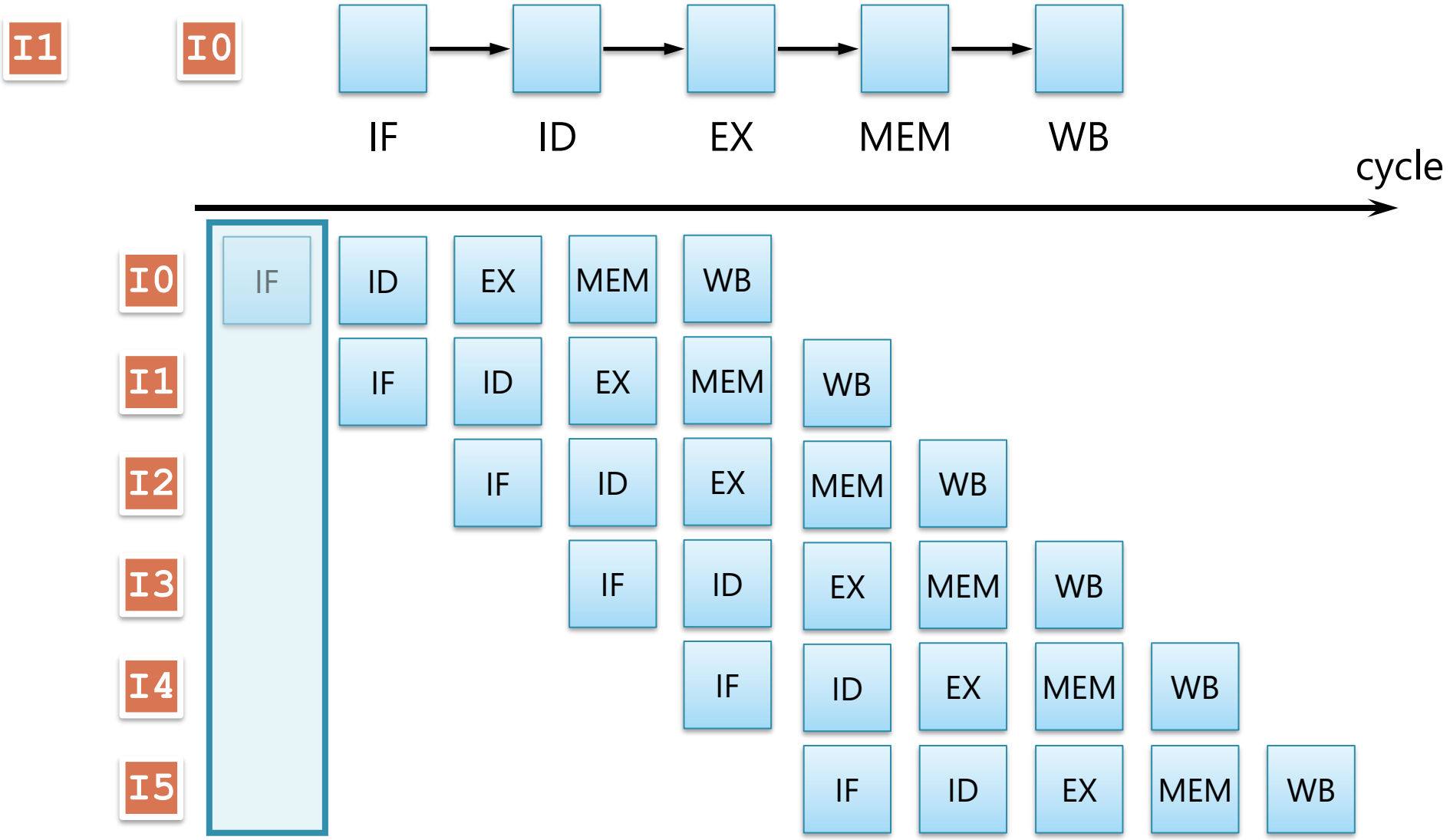
1. 命令パイプラインの詳細
    1. 前回の復習から
2. ハザードとその解決方法

各処理は基本的には左から右に流れる

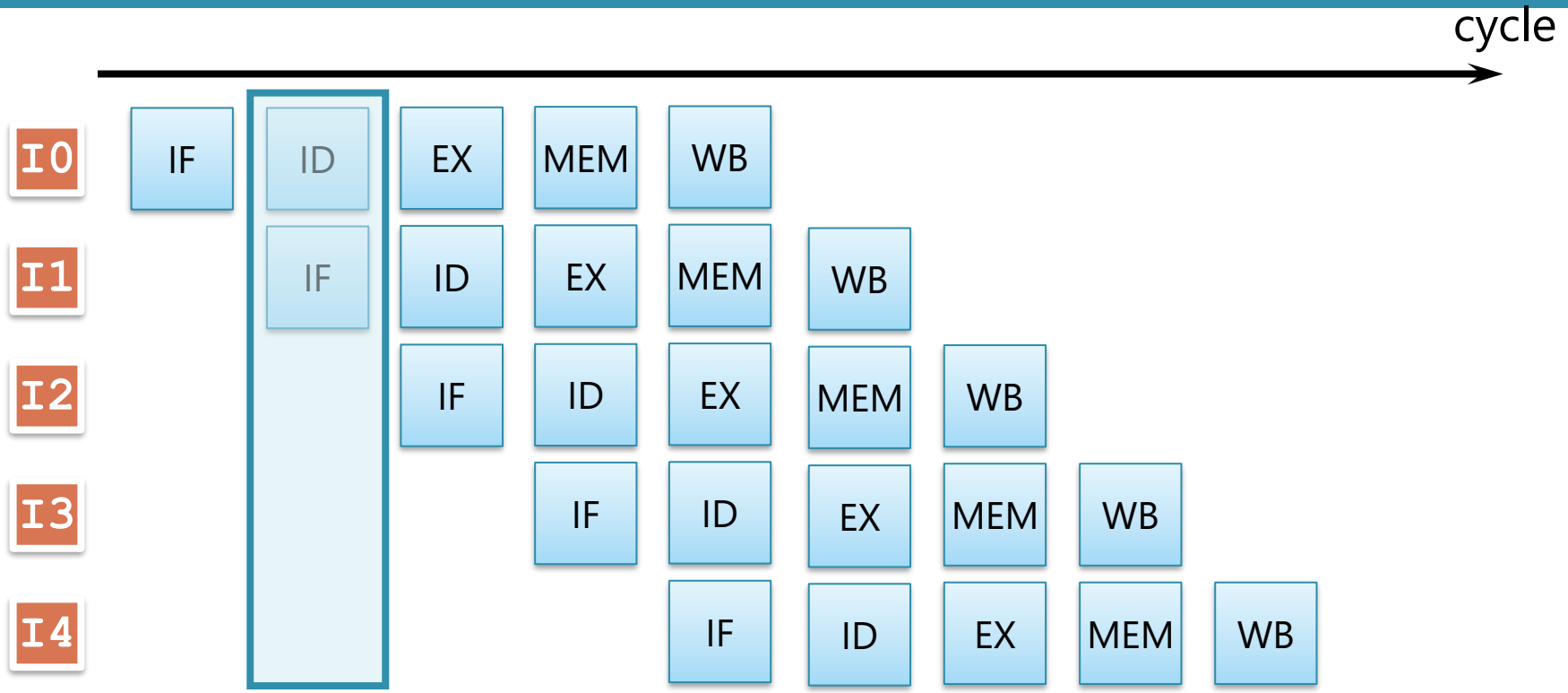


- ◇ 特定のユニットで仕事をしている間，他の部分は遊んでいる
- ◇ パイプライン化
  - □ これをもとに，導入で話したように処理をオーバーラップさせる

命令パイプラインの実行の様子

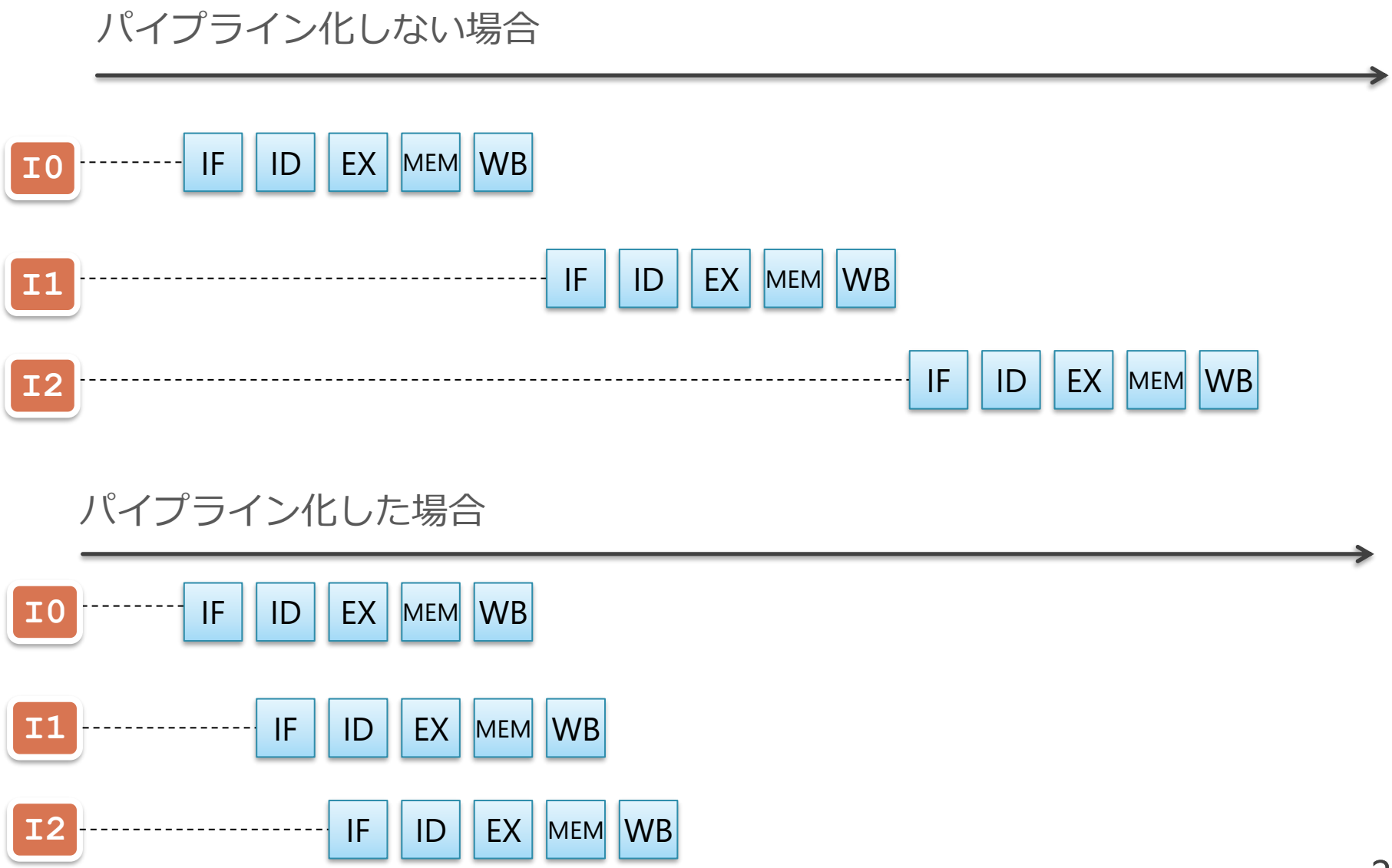


パイプライン・チャートの見方
ここから先で多用されるので重要

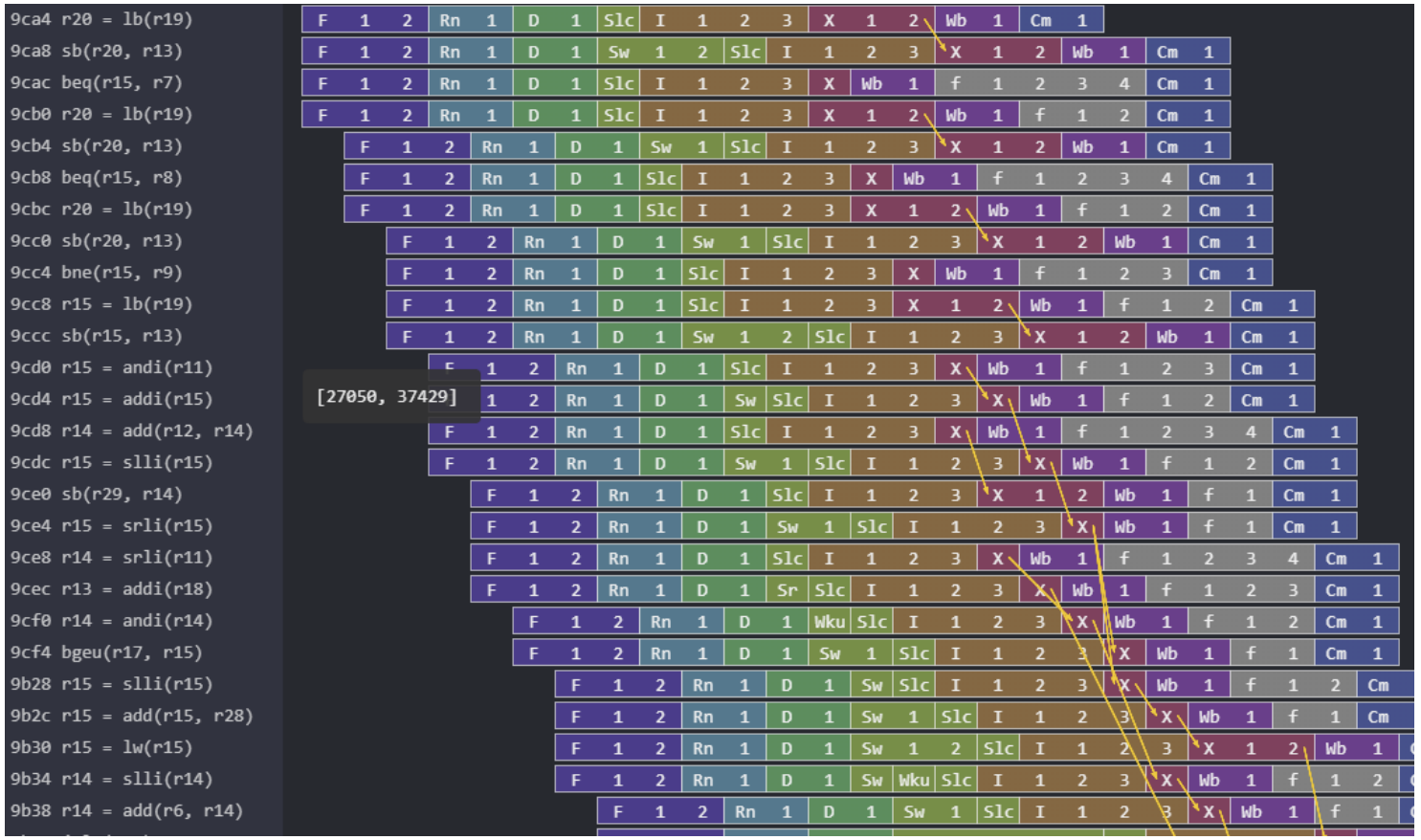


- 左から右にむかって時間は進む
- 上から下にむかって命令が実行順に置かれる
- 各ステージを表す四角は左側にある命令がその時そこにいることを示す
  - 上記では２サイクル目に，I0 が ID に，I1 が IF で処理されている

パイプライン化による性能（スループット）向上



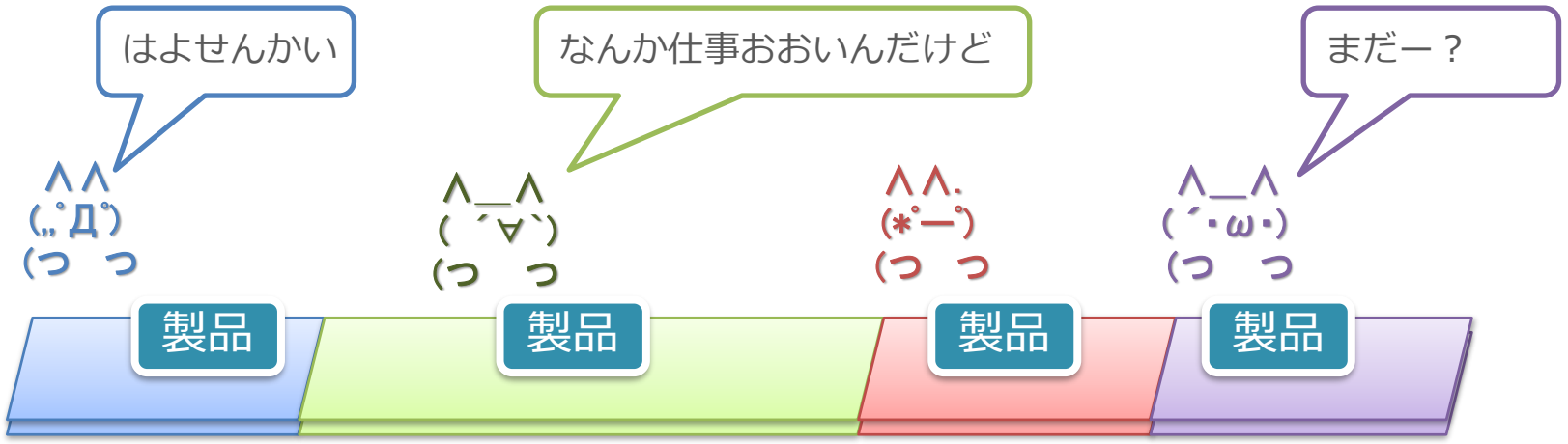
余談：実際の CPU を実行した場合のパイプライン



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

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

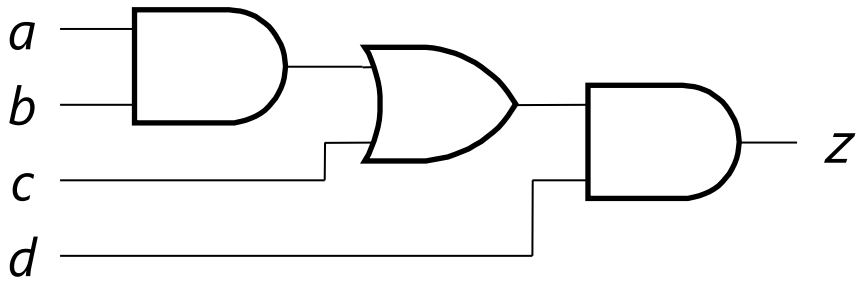


ステージを「どうやって」切るか

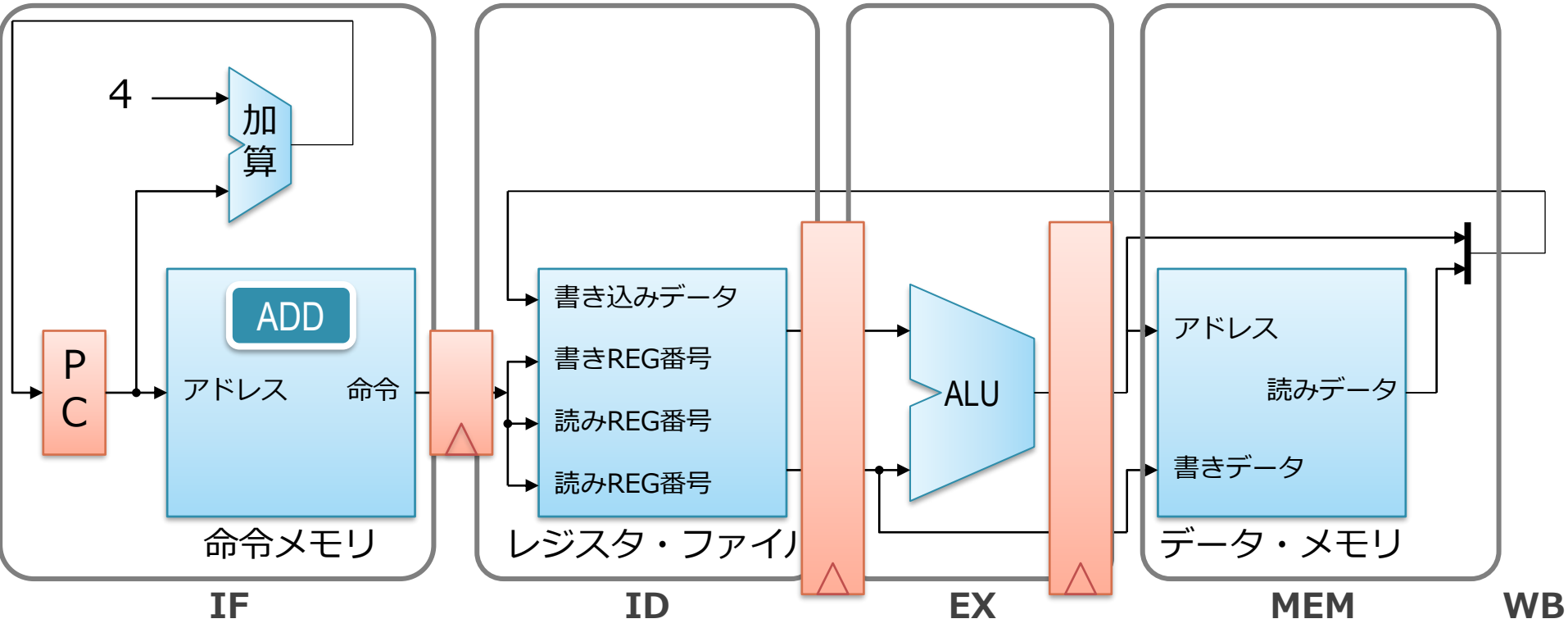
- 適当に間隔をあけて流せばよいというものもない

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・ファイル自体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

- パイプライン・ハザード：
  - ◇ パイプライン動作を妨げる要因

- 分類：
  1. 構造ハザード ：　　ハード資源の不足による
  2. 非構造ハザード：　　バックエッジによる
     a. データ・ハザード：　データ依存
     b. 制御ハザード ：　　制御依存（分岐命令）

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

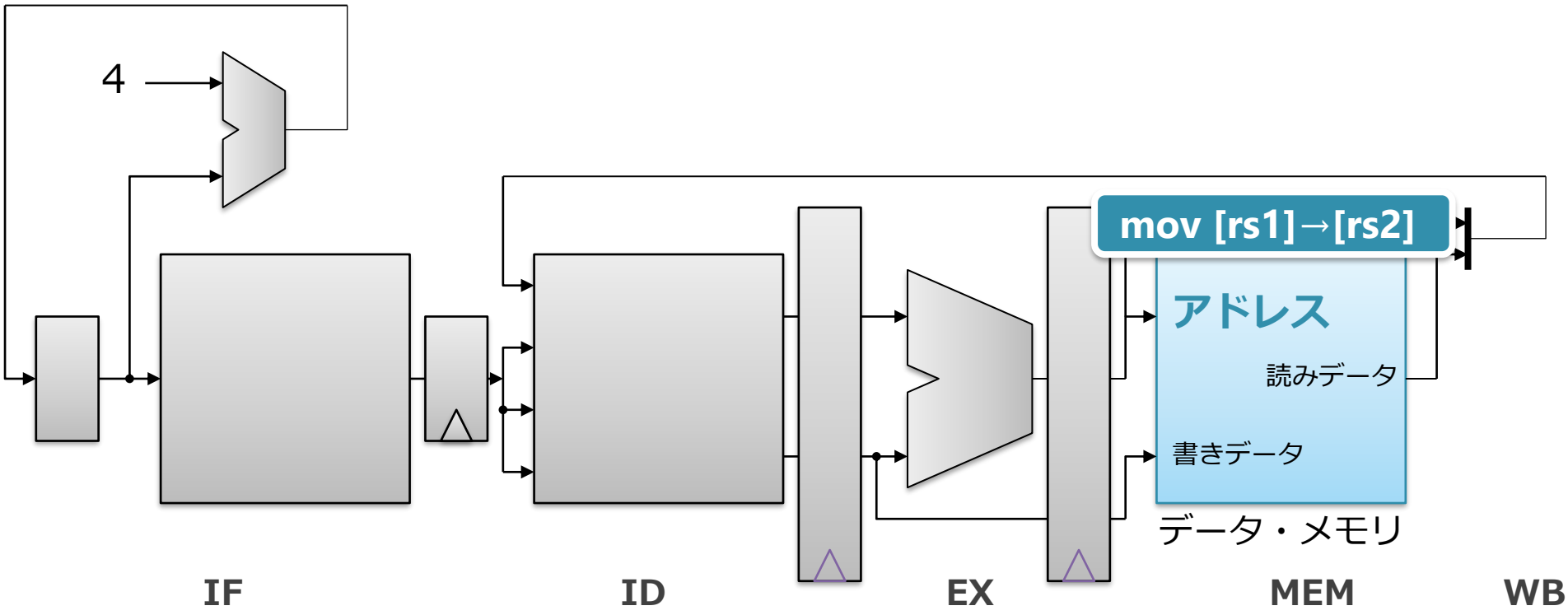
構造ハザード

- 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 mov [rs1]→[rs2] のような命令があったとする
  - ◇ rs1 で指定されるアドレスのメモリの値を読んで，
  - ◇ 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

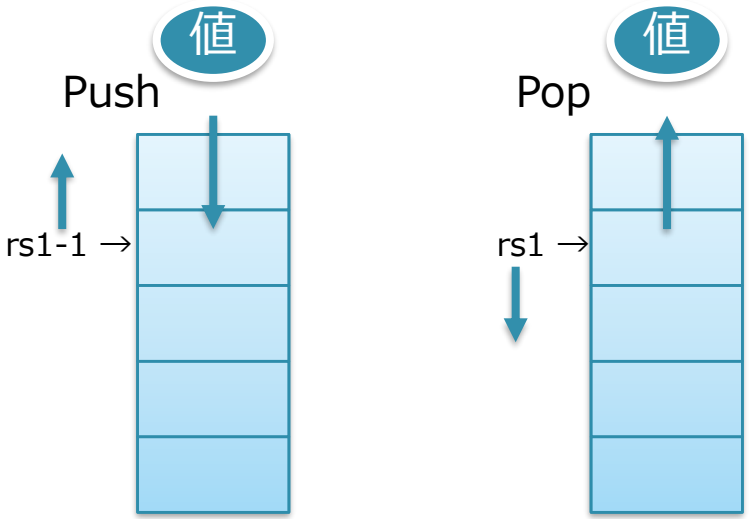
mov [rs1]→[rs2] // [rs1]→[rs2] へのコピー



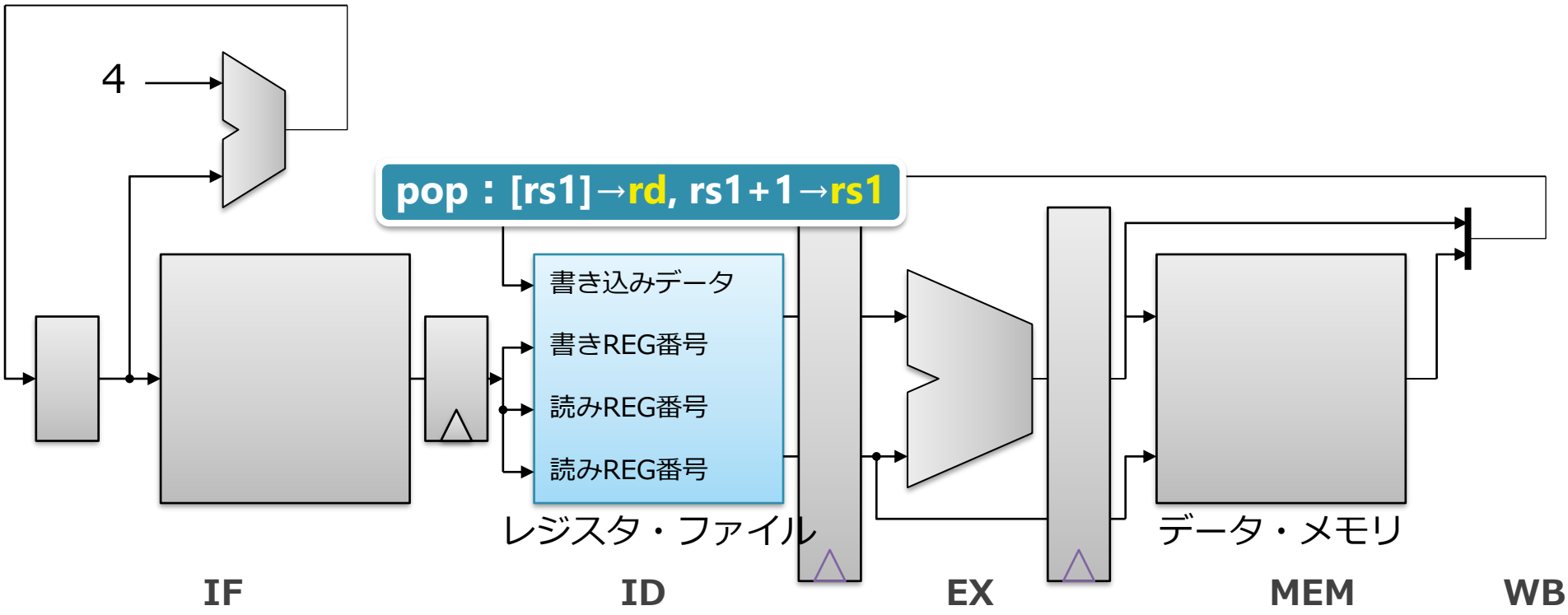
- メモリをあるサイクルに同時に読んで書く必要がある
  - ◇ しかし，データ・メモリのアドレスの口は1つしかない
  - ◇ MEM ステージでデータ・メモリの読みと書きが同時にできない

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→rd,　r2→[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]→rd, rs1+1→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



pop：[rs1]→rd, rs1+1→rs1



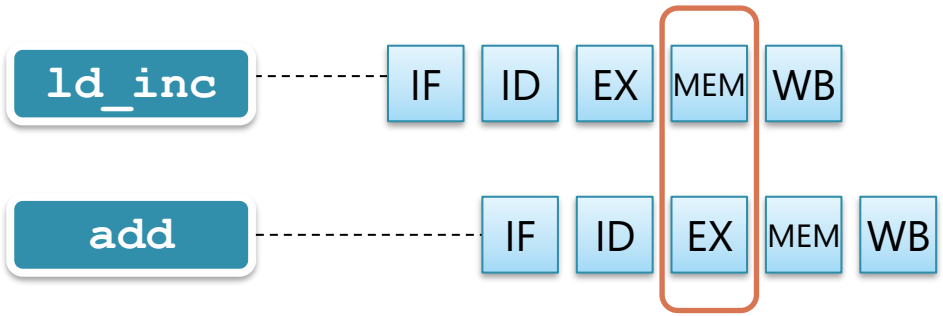
- WB ステージでレジスタに rd と rs1 の2つを書き込む必要がある
  - ◇ レジスタ・ファイルへの書き込みは，同時に２つはできない

構造ハザードの例３

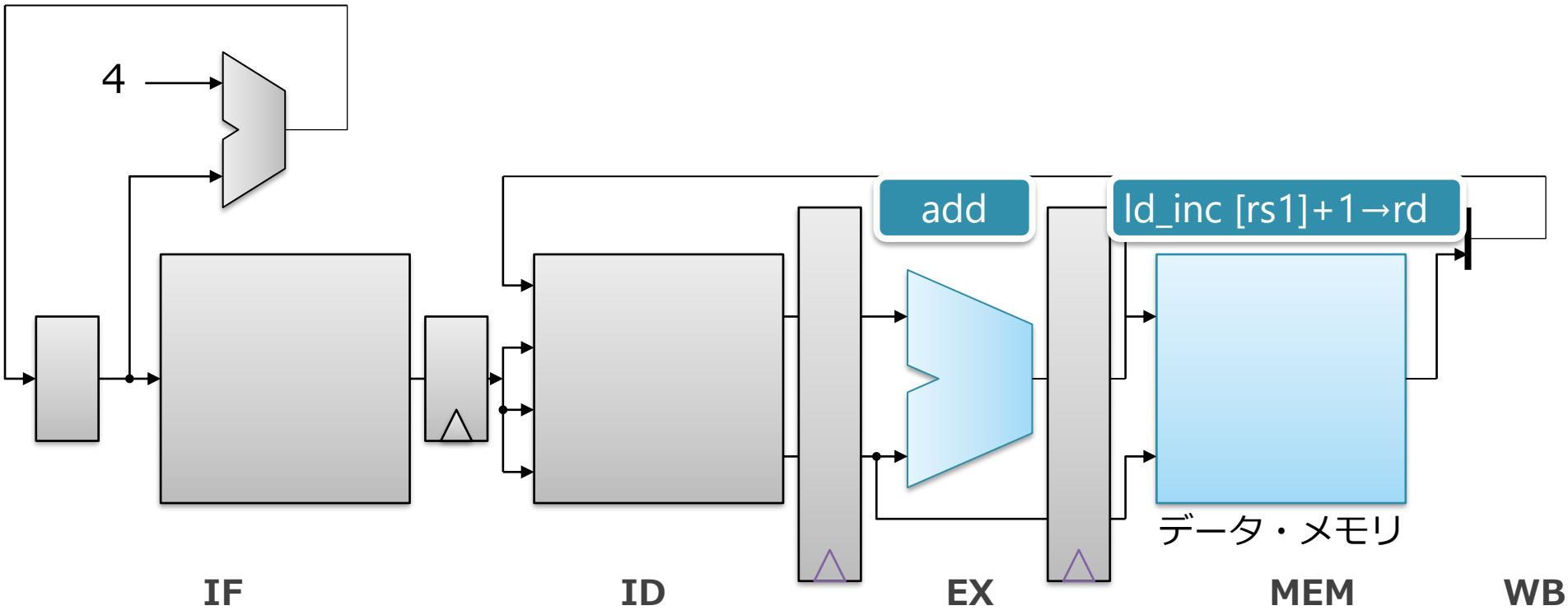
- 使用資源の異なるステージ間のぶつかりでも起きる
  - ◇ これまでの例は，同じステージ内で資源が足りない例
- **`ld_inc [rs1]+1→rd`** のような命令があったとする
  1. rs1 の指すアドレスからメモリを読む
  2. 読んだ値にさらに+１してから rd に書く
- 一見，資源は足りているようだが･･･
  - ◇ レジスタの読み書きは１つずつしかない
  - ◇ メモリも１カ所を読むだけ
  - ◇ 加算も１回行うだけ

構造ハザードの例３

- ld_inc [rs1]+1→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→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つ同時に書き込めるように
  - ld_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）

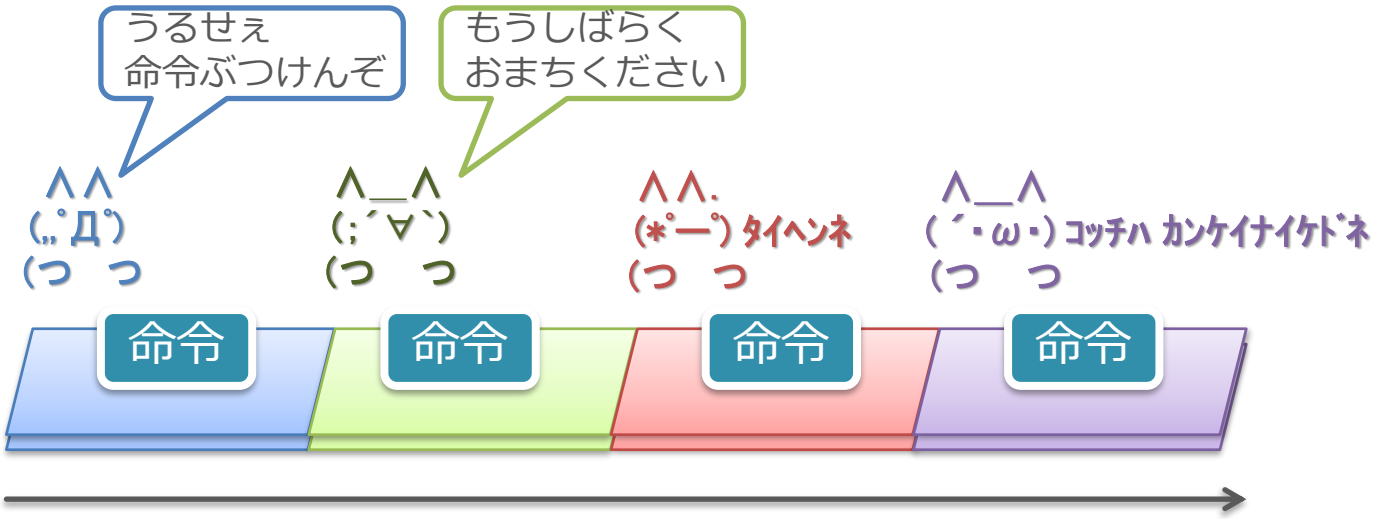
構造ハザードの解決方法

- 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
  - ld_inc [rs1]+1→rd
    - ld_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

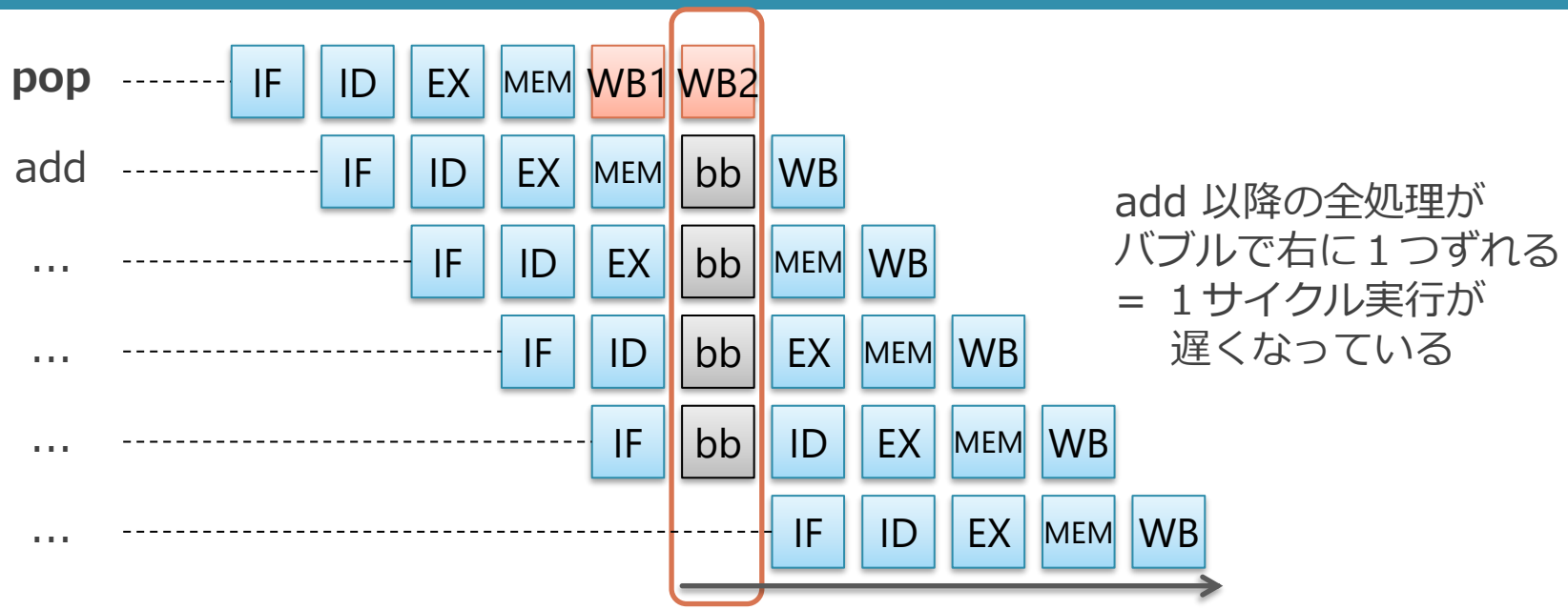


- 上流を止めないと破綻する
  - ◇ (;´∀`) が複数サイクルをかけて仕事をしている場合，命令はそこにとどまり続ける
  - ◇ その間は上流をとめないと命令をおく場所がないし，依存関係がまもられない
- (*ﾟ―ﾟ) より下流は流れていっても，この場合は問題ない

パイプラインを止めること

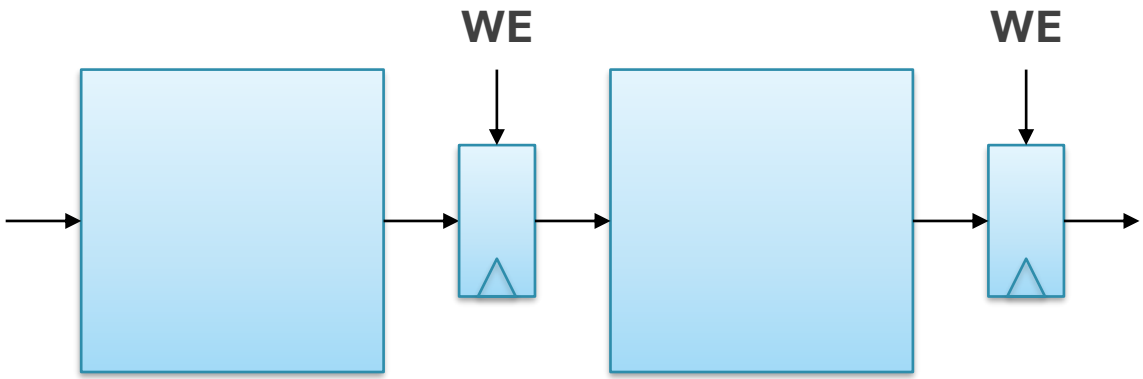
- パイプラインを止めるとことを「ストール」や「インターロック」という
  - ◇ 本や人によって，意味や使い方が微妙に統一されていない
  - ◇ この講義では，以降はストールで統一

ストールの動作



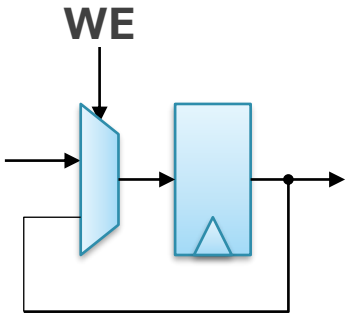
- pop： 1 つレジスタに書いたあと，次のサイクルで書き込む
  - ◇ WB1 と WB2 の 2 サイクルで書き込む
  - ◇ WB2 の間は上流を全て止める
- パイプライン・チャート上では上記のようになる
  - ◇ 止める原因の命令の下が全部右にずれる
  - ◇ ずれた部分の空き（bb）を「バブル」とよぶ

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

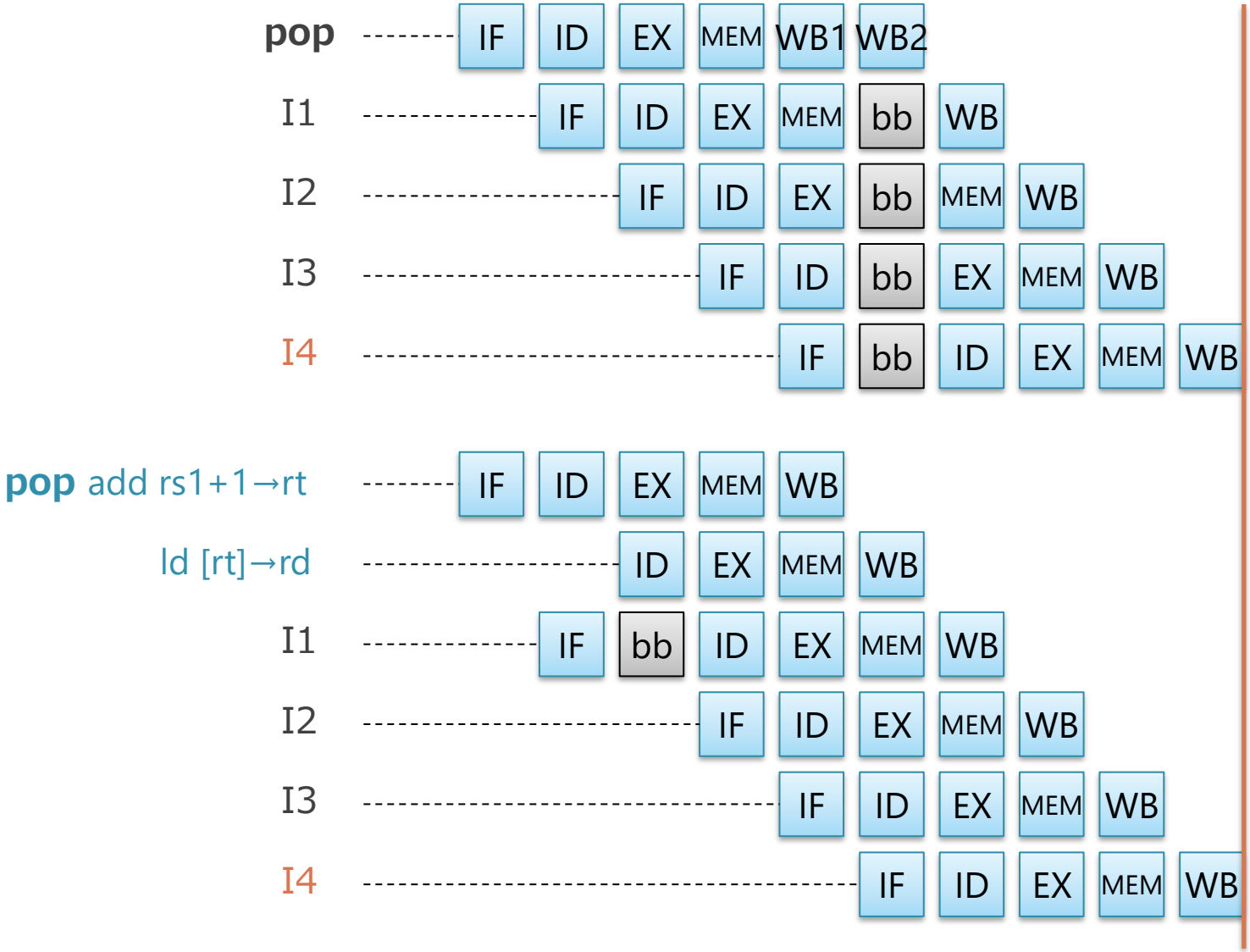
- mov [rs1]→[rs2]
  1. ld [rs1]→rt
  2. st rt→[rs2]

- 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，ld_inc が連続で来た場合，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

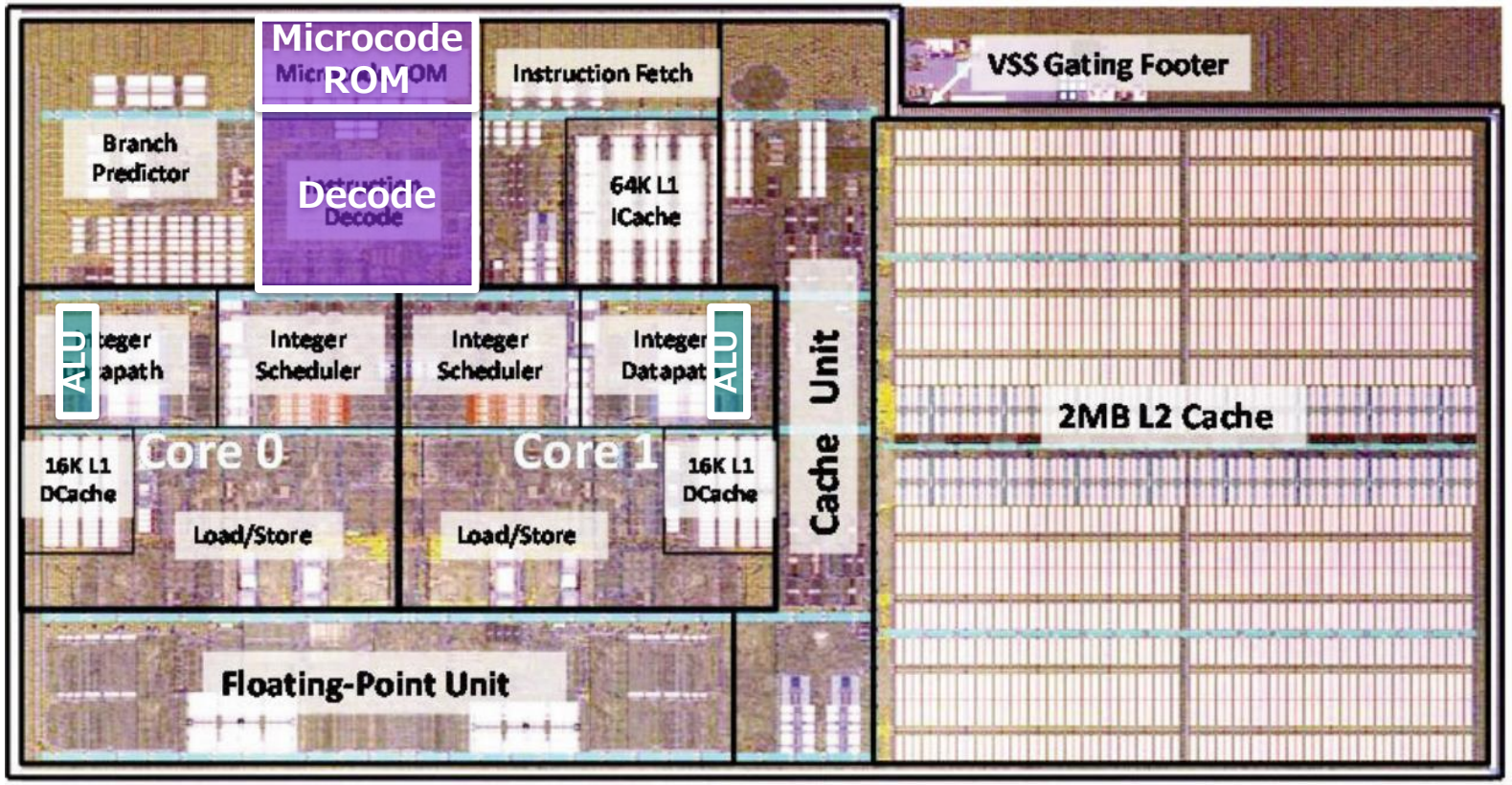
マイクロ命令への分解の利点

- 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

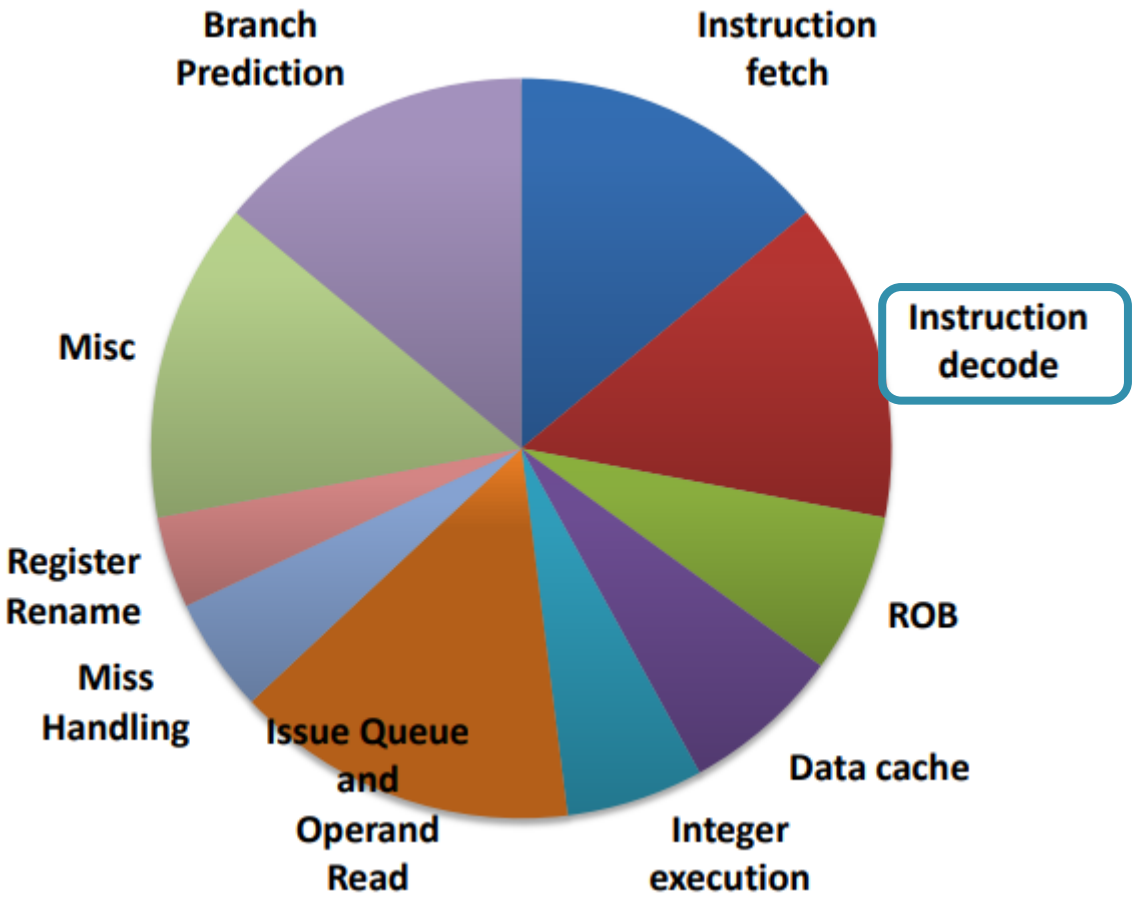
- 分解（デコード）がマジで大変
- 基本的には，ひたすらパターン・マッチング
  - ◇ でかい真理値表がいる
  - ◇ 本当に複雑なものは，メモリで出来たテーブルも使う

AMD Bulldozer のチップ写真



- Tim Fischer1 , Srikanth Arekapudi2 , Eric Busta1 , Carl Dietz3 , Michael Golden2 , Scott Hilker2 , Aaron Horiuchi1 , Kevin A. Hurd1 , Dave Johnson1 , Hugh McIntyre2 , Samuel Naffziger1 , James Vinh2 , Jonathan White4 , 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_ MC0_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

- https://pcper.com/wp-content/uploads/2017/06/6cfe-6th-gen-x-series-spec-update.pdf より

Windows Update でこっそり更新されていたりもする

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"]) に関連する 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

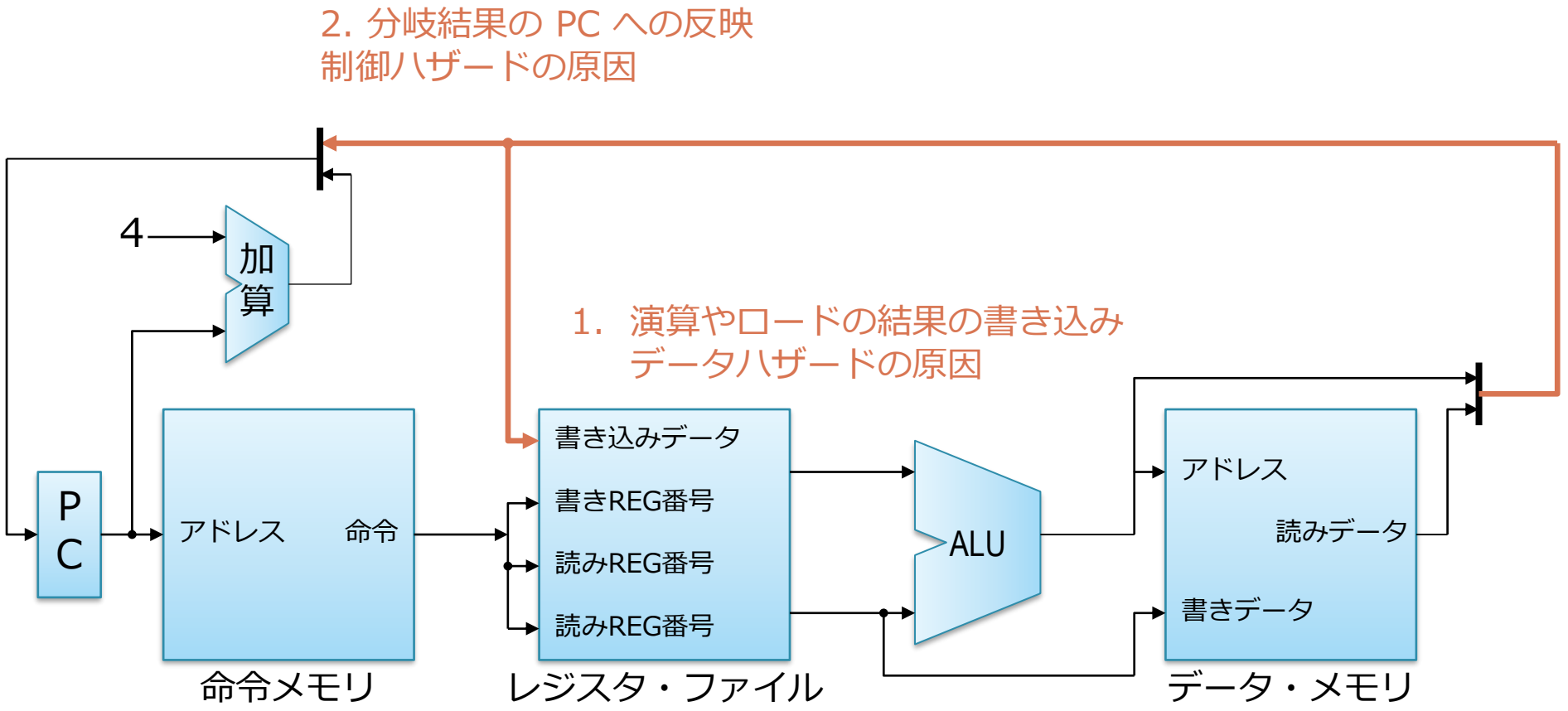
構造ハザードのまとめ

- 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

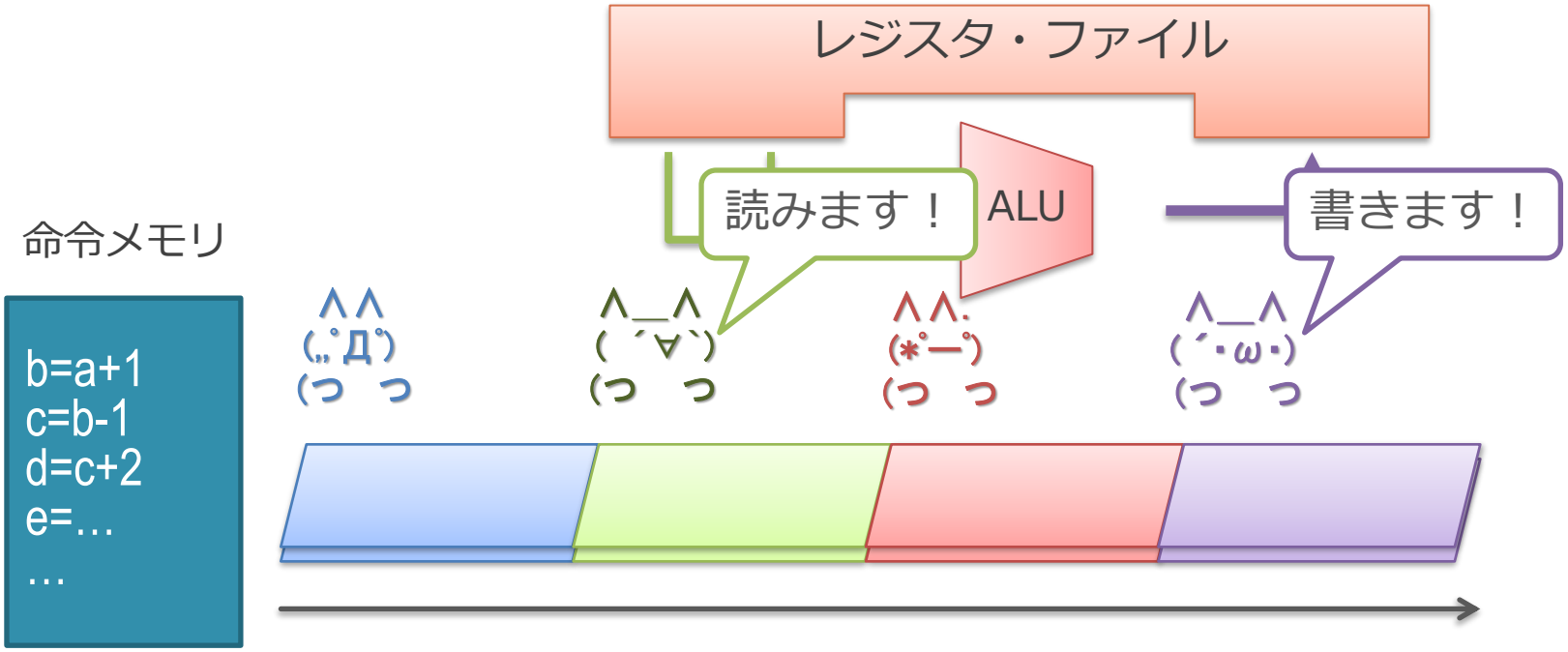
1. 構造ハザード
2. **非構造ハザード：バックエッジに由来**
   a. データ・ハザード
   b. 制御ハザード

バックエッジ：逆方向（右から左）にいく信号



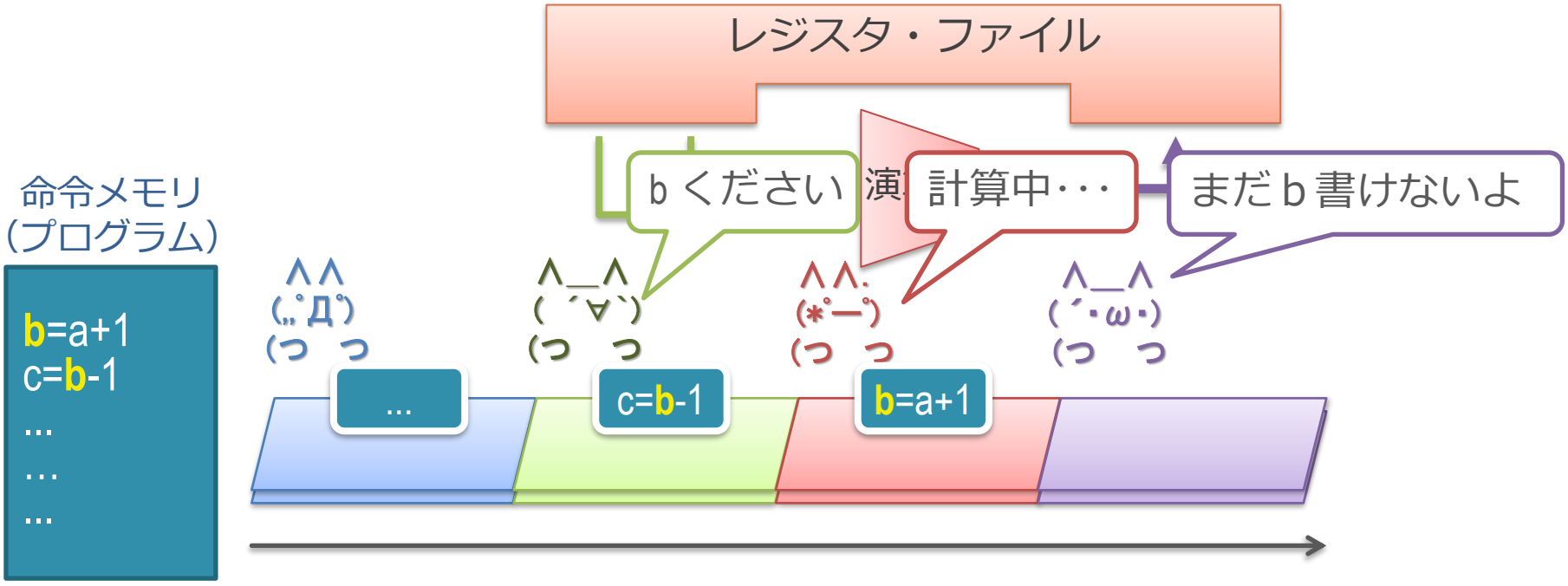
- バックエッジがあるため，命令を単純に流せない場合がある
  - ◇ 工場のラインのように，一方向に流せない

データ・ハザード



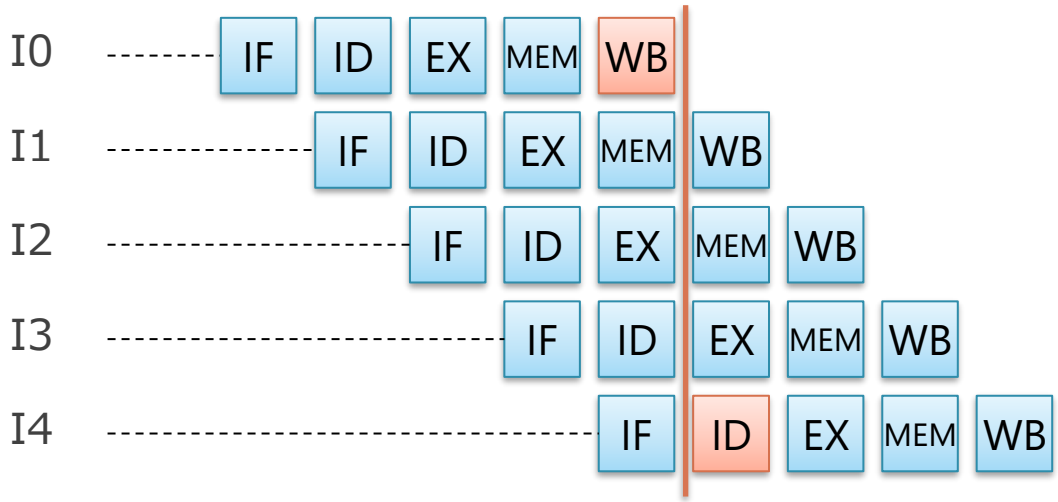
- レジスタ・ファイルへのアクセス
  - ◇ 演算の入力は(´∀`) の人がレジスタ・ファイルから読み出す
  - ◇ 演算の結果は(´･ω･) の人がレジスタ・ファイルに書き込む

データ・ハザード



- 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,
  - ◇ (*゜ー゜) 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

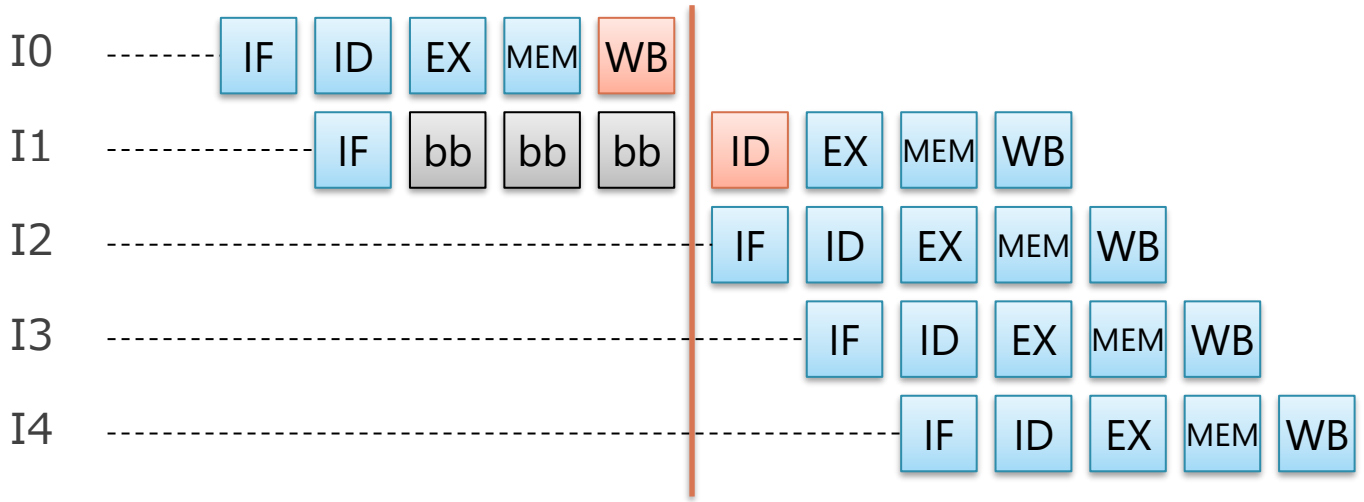


- I0 の WB が終わるまで，その結果はレジスタに書き込まれない
  - ◇ I4 までは，その値がレジスタから得られない
  - ◇ ID ステージでレジスタを読むため

データ・ハザードの解消方法

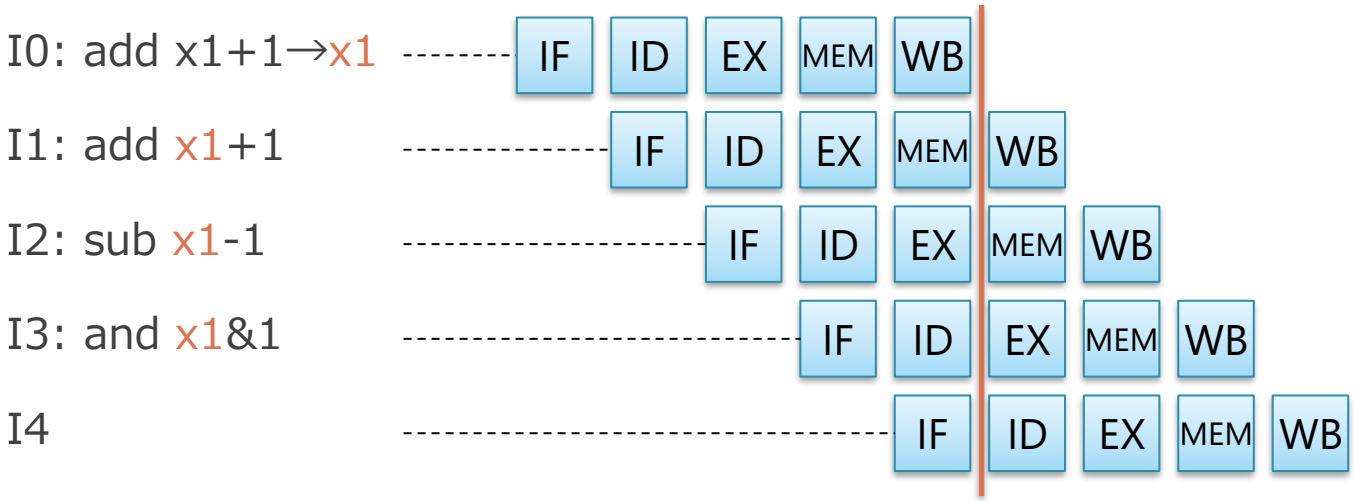
- 解消方法
  1. ストールさせる
  2. 遅延スロット（なにもしない）
  3. フォワーディング
  4. マルチスレッディング

1．ストールさせる



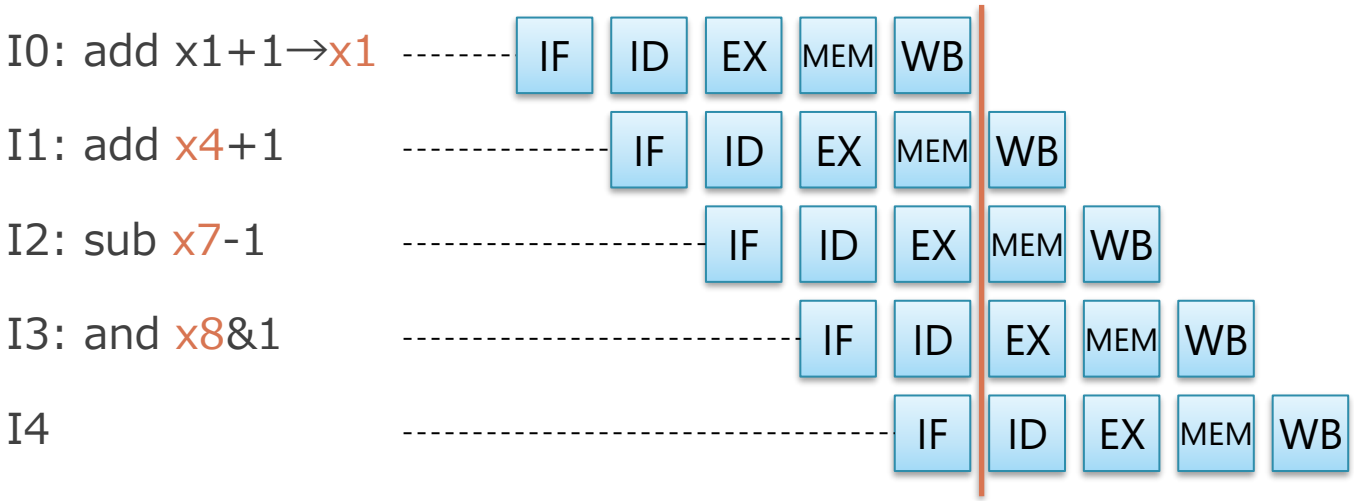
- I0 の WB が終わるまで，後続の命令を遅らせる
  - ◇ I1 の ID が，I0 の WB の右にくるまでストール
  - ◇ I1 は I0 の結果を使える
- 欠点：とても遅くなる

2. 遅延スロット（なにもしない）



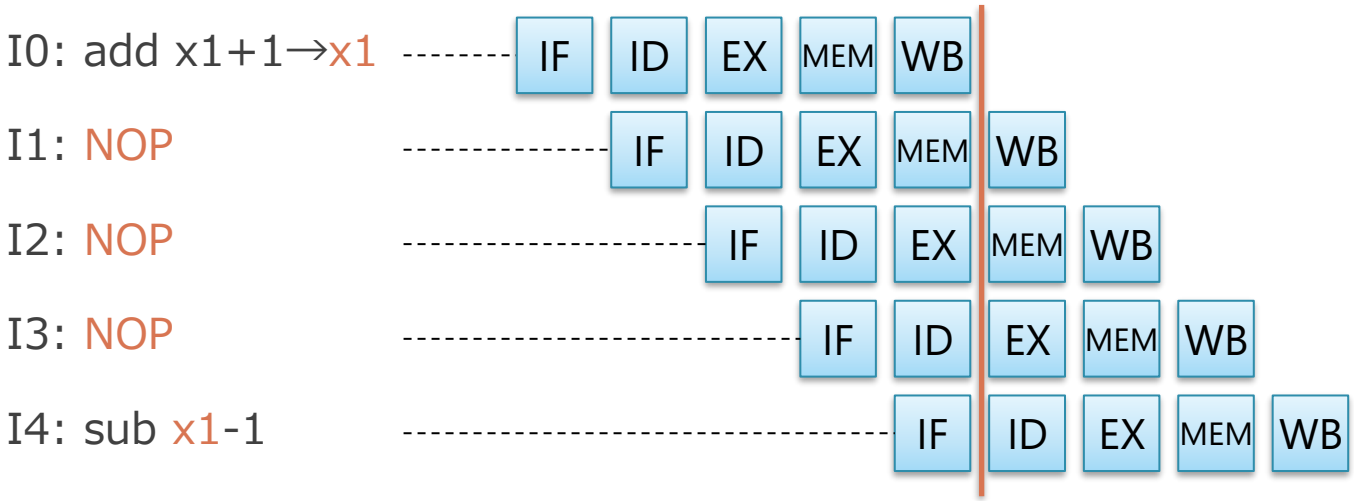
- 特になにも対策せず，
  「ある命令の結果は，数命令先まで見えない」と言う仕様にする
  - ◇ 上の例だと I1，I2，I3 は，I0 の結果は見えない
  - ◇ I1，I2，I3 には，I0 で add する前の値が見え続ける

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



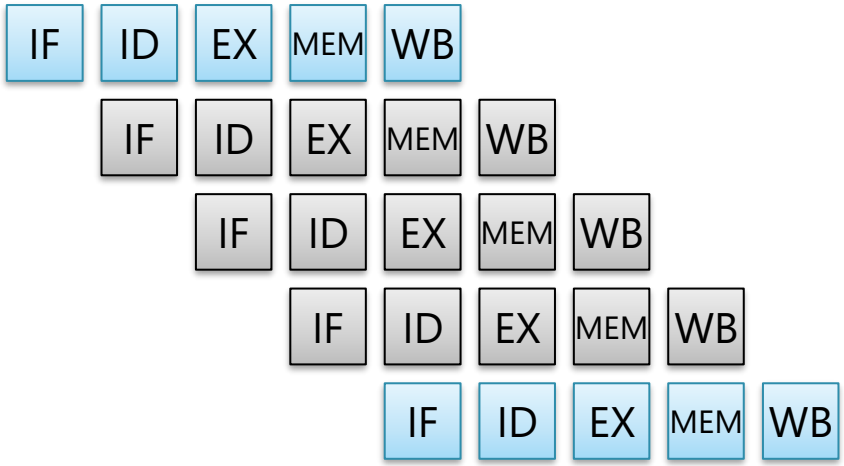
- もしそのような命令がない場合，
  - ◇ NOP（No Operation）と呼ぶ何もしない命令をいれる
  - ◇ これもコンパイル時にいれておくる必要がある
- 上の例は，x1 に 1 を足した結果を使う以外の処理がなかった場合

遅延スロットの利点

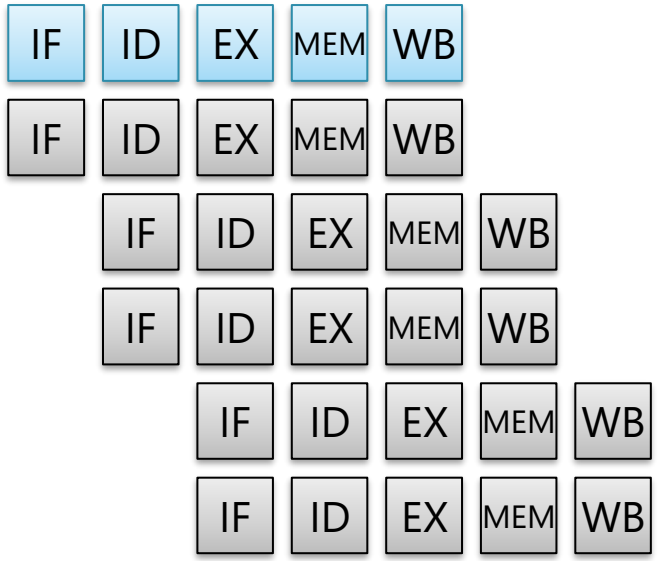
- 利点：
  - なにもしないので，ハードは最も単純
  - 並列にできる命令があれば，性能も下がらない

遅延スロットの欠点

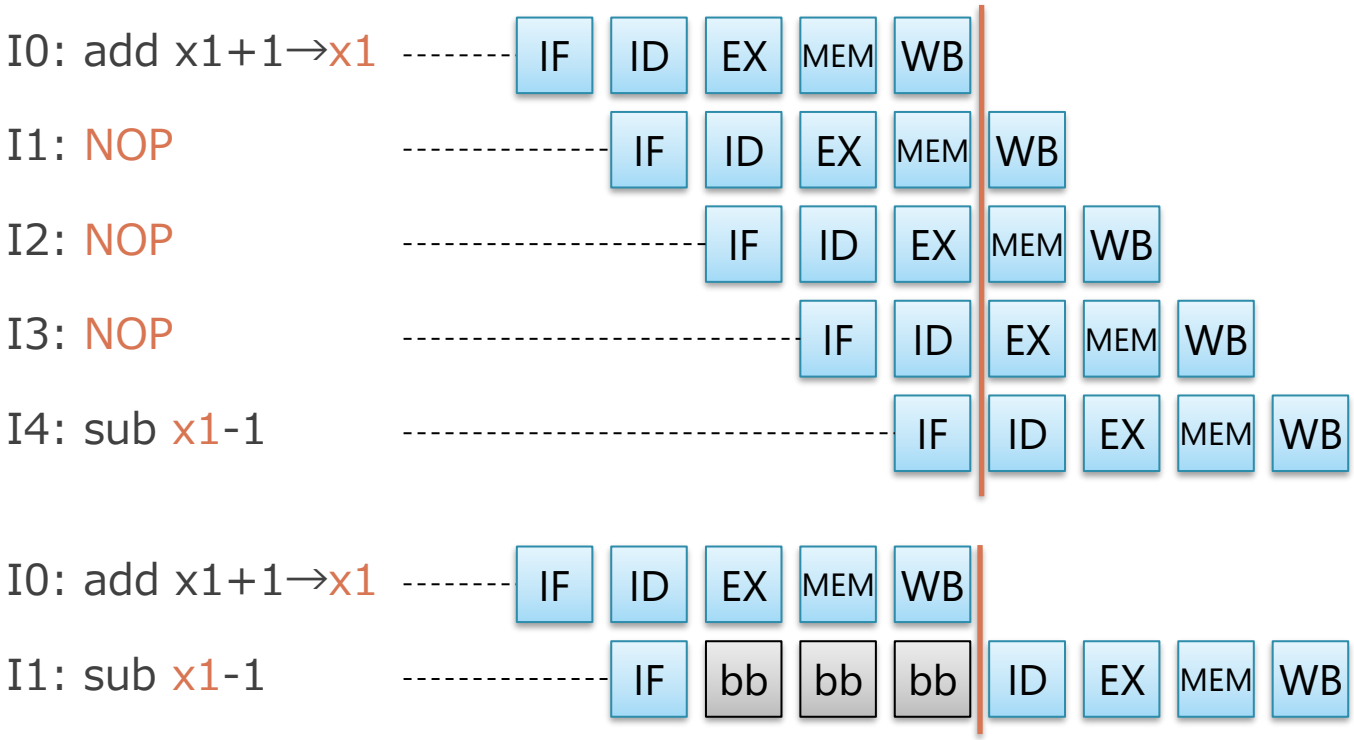
- 欠点：「仕様」なので，一度決めると変えられない
  - ◇ 後からパイプラインの段数を変えると互換性がなくなる
    - □ クロックをあげるために，段数を増やせない
  - ◇ 複数の命令を同時処理しようとしたときにも互換性がなくなる
  - ◇ MIPS では遅延スロットが 1 命令分，仕様として存在
    - □ 互換性のためにこれを忠実に再現するため後年は逆に複雑化



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

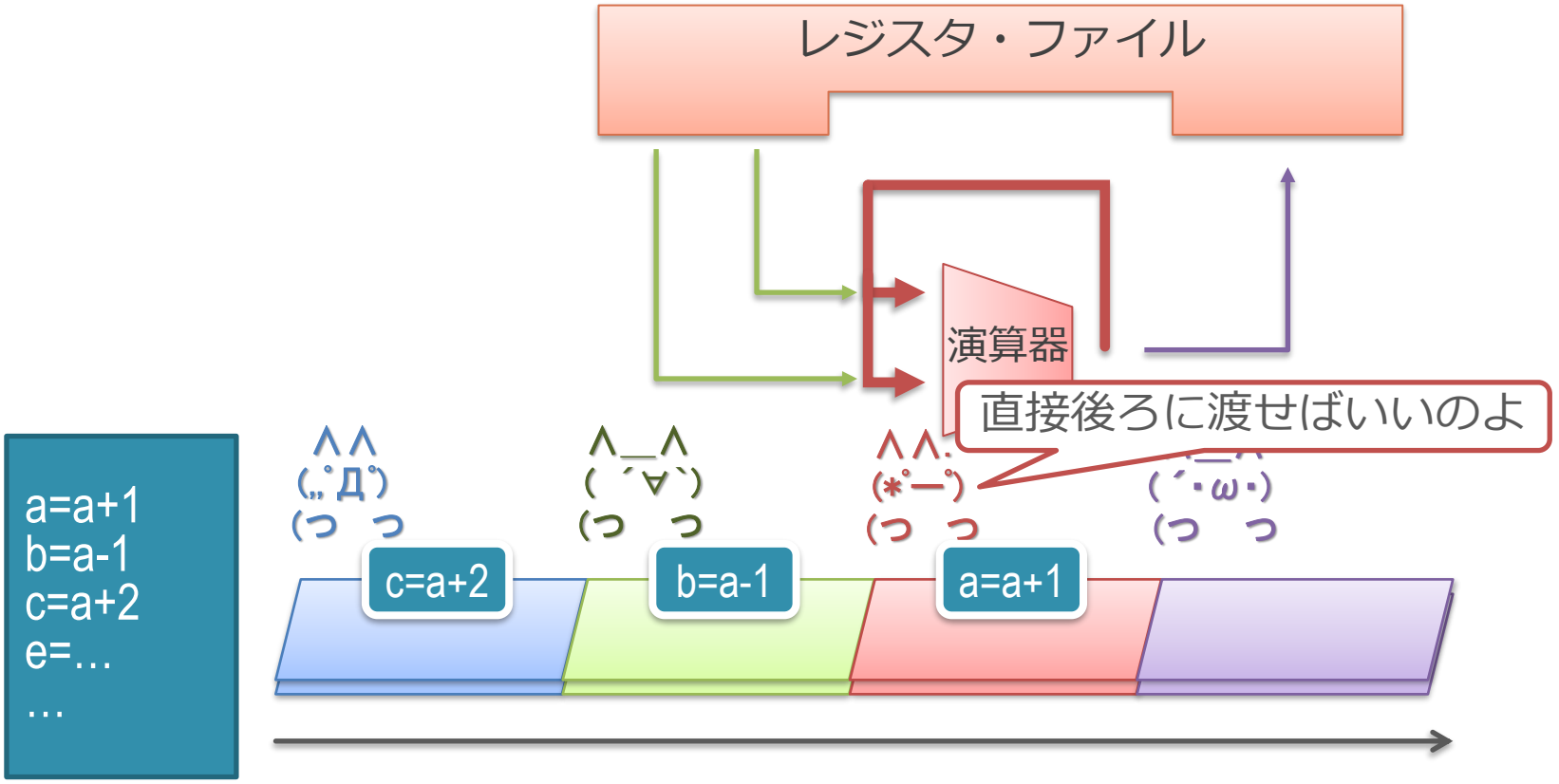


- 欠点２：並列してできる命令が常にあるとは限らない
  - ◇ NOP を入れるしかなくなる
  - ◇ 実質ストールしてバブルを入れるのと同じになってしまう

データ・ハザードの解消方法

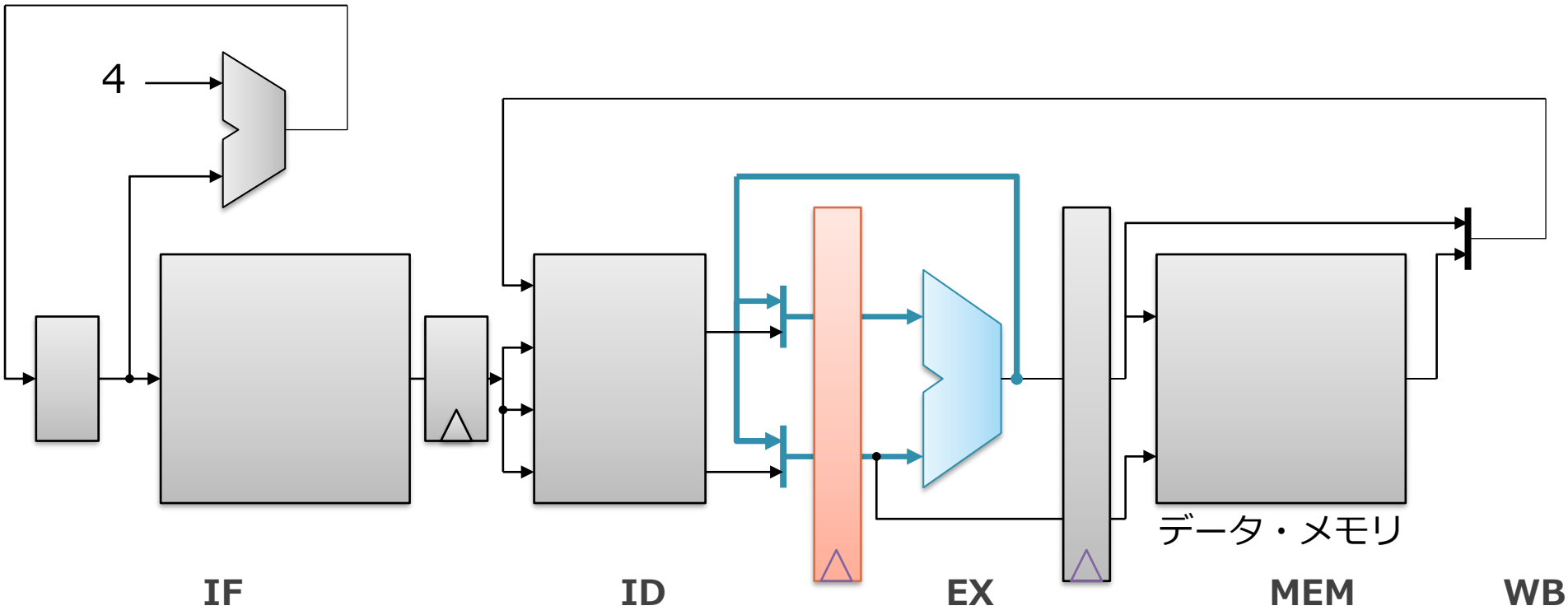
- 解消方法
  1. ストールさせる
  2. 遅延スロット（なにもしない）
  3. **フォワーディング**
  4. マルチスレッディング

フォワーディング



- フォワーディング（バイパスとも呼ぶ）
  - (*ﾟーﾟ) 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

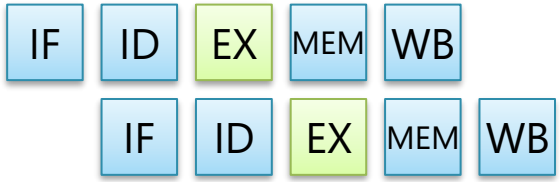


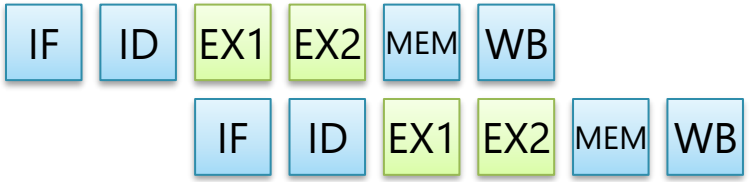
- 演算器の結果を，フィードバック
  - ◇ レジスタ・ファイルからの読み出し結果と選択して入力に

フォワーディングの利点

- 利点：
  - ◇ 依存関係がある命令が連続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

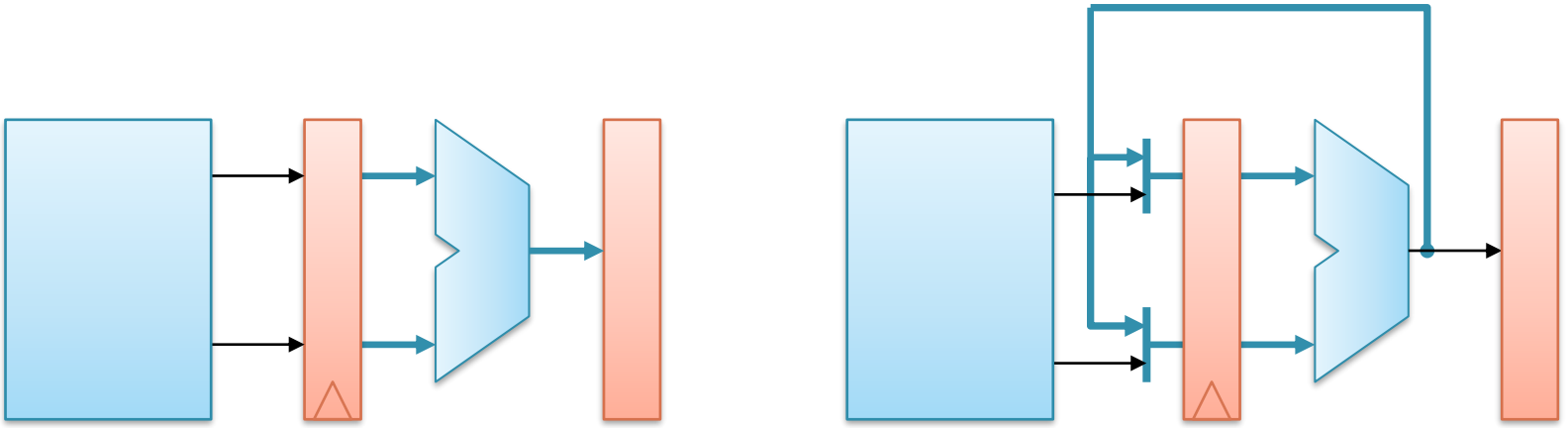
フォワーディングの問題





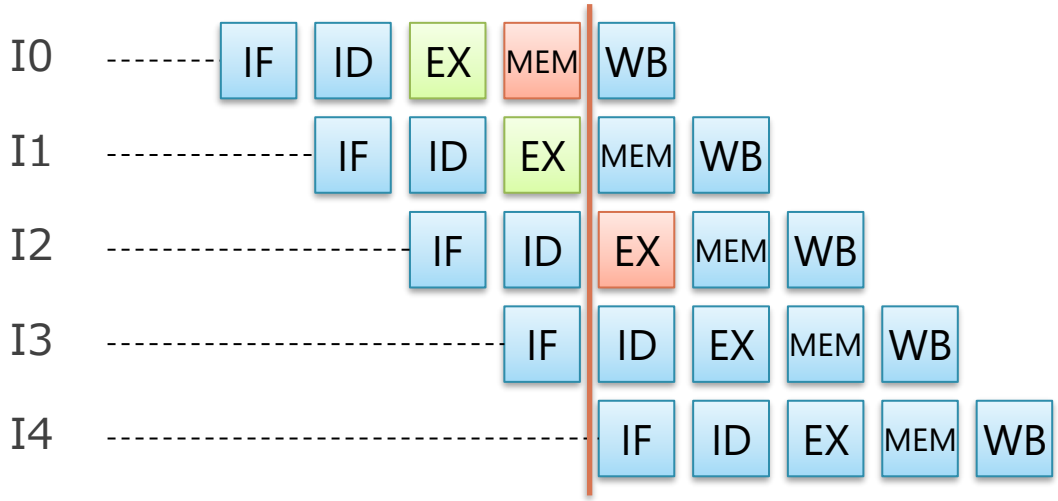
- 演算器とフォワーディングを含むステージは１サイクルで処理する必要がある
  - 分割すると，バブルを入れてるのと同じになってしまう
  - できればここはパイプライン化したくない
    - （FP 演算等の複雑なものは，やむなくパイプライン化している
- CPU 全体のクリティカル・パスになりやすい
  - クロック周波数がここで決まることが多い

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

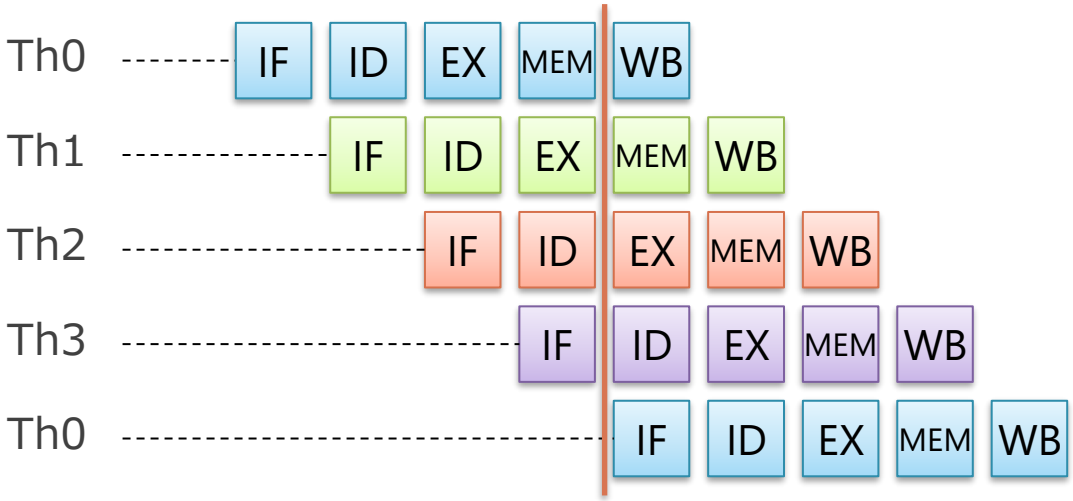
データ・ハザードの解消方法

- 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おきない
    - □ 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

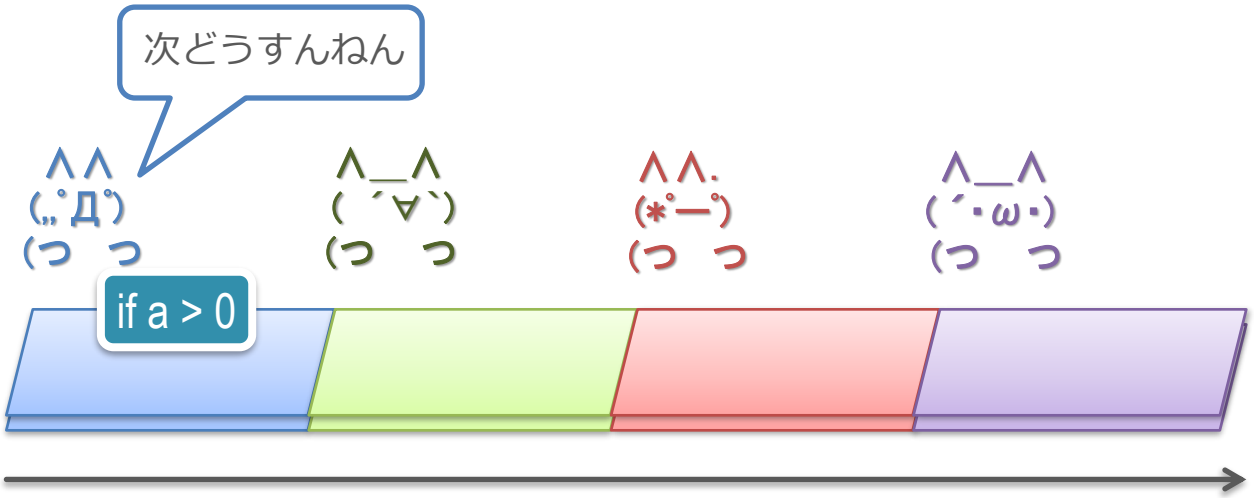
1. 構造ハザード

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

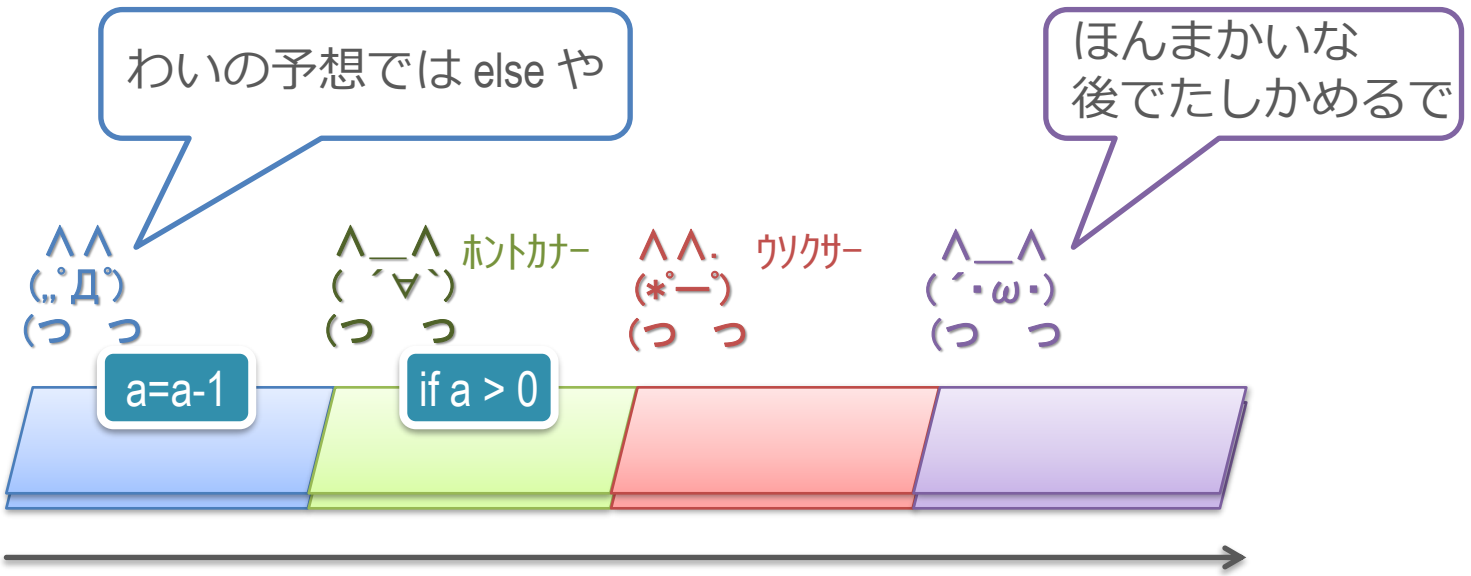
- 解消方法
  1. ストールさせる
  2. 遅延スロット（なにもしない）
  3. マルチスレッディング
     - 上記は，基本的にデータ・ハザードと同様にして適用できる
     - フォワーディングは制御ハザードでは意味的に無理

  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
```

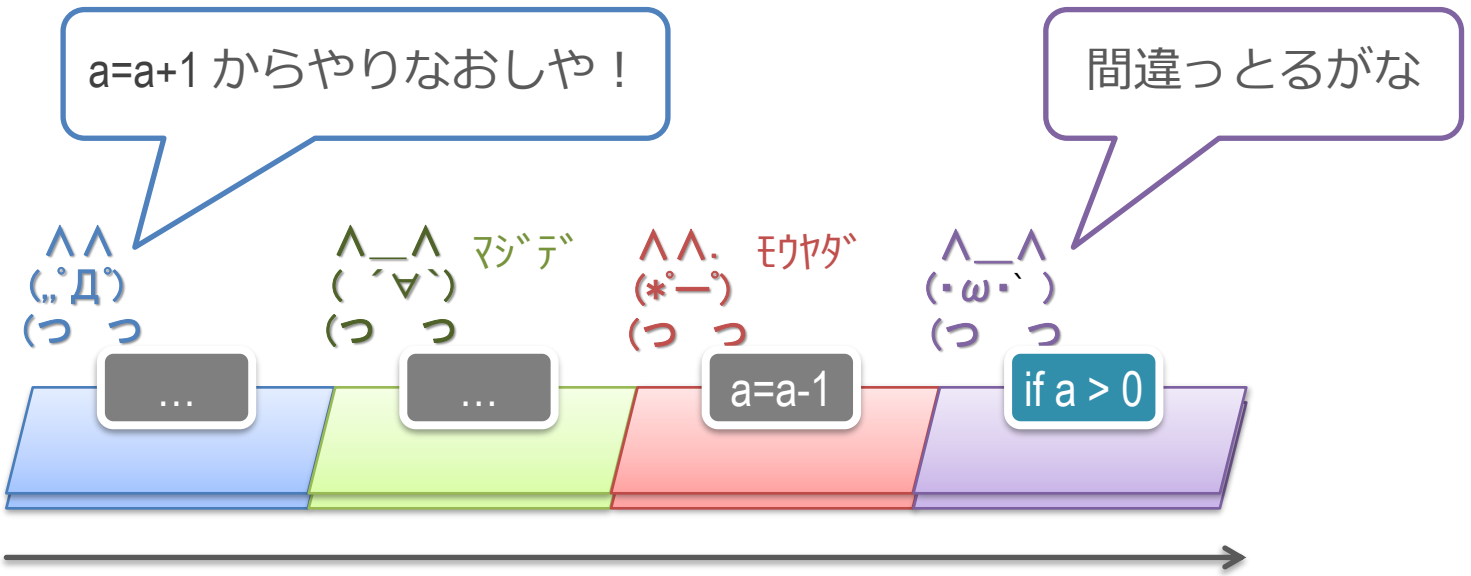


- 動作
  - 「if a > 0」の結果を予測して，命令を取り込む
    - 前回はこっちに行ったので，次もこっちに違いないとかで予測
  - あとから予測が正しいか確認する

分岐予測ペナルティ

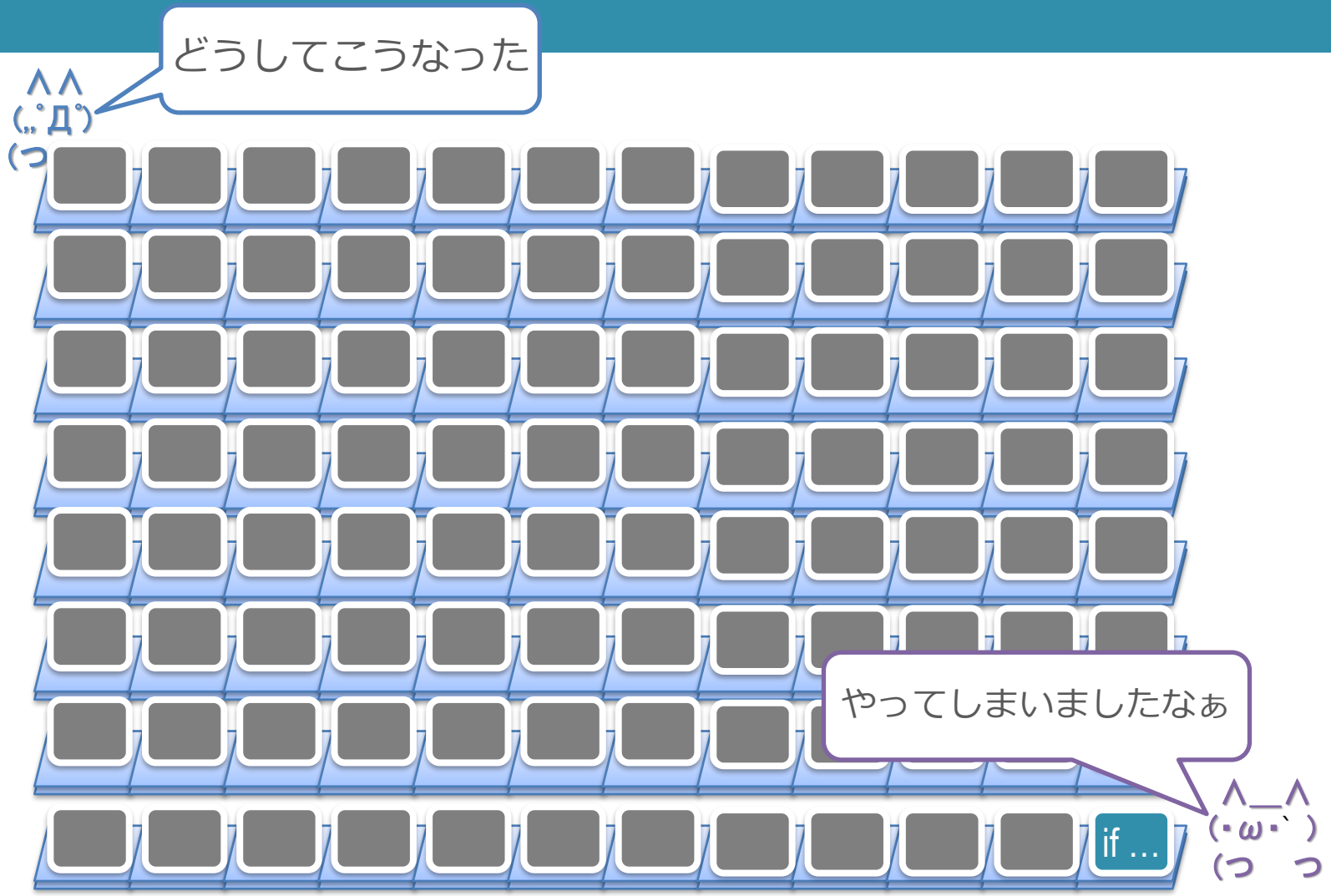
プログラム

```
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

- パイプラインの詳細
- 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，わからなかったところ，質問，感想などを書いてください（なんか一言書いてね）
  - ◇ 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：hazard
- 意見や内容へのリクエストもあったら書いてください