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

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

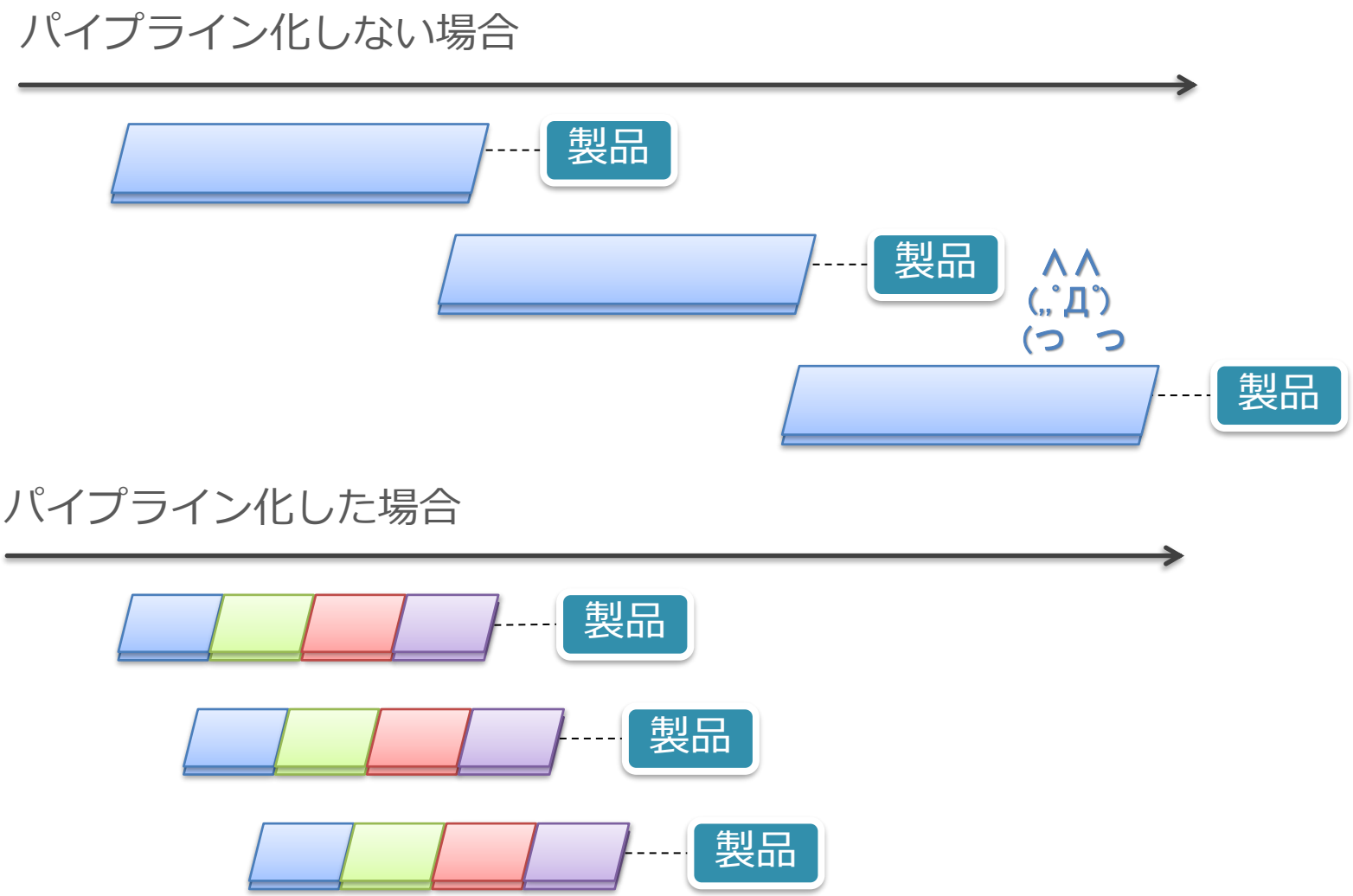
導入：工場のラインを考える

- 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

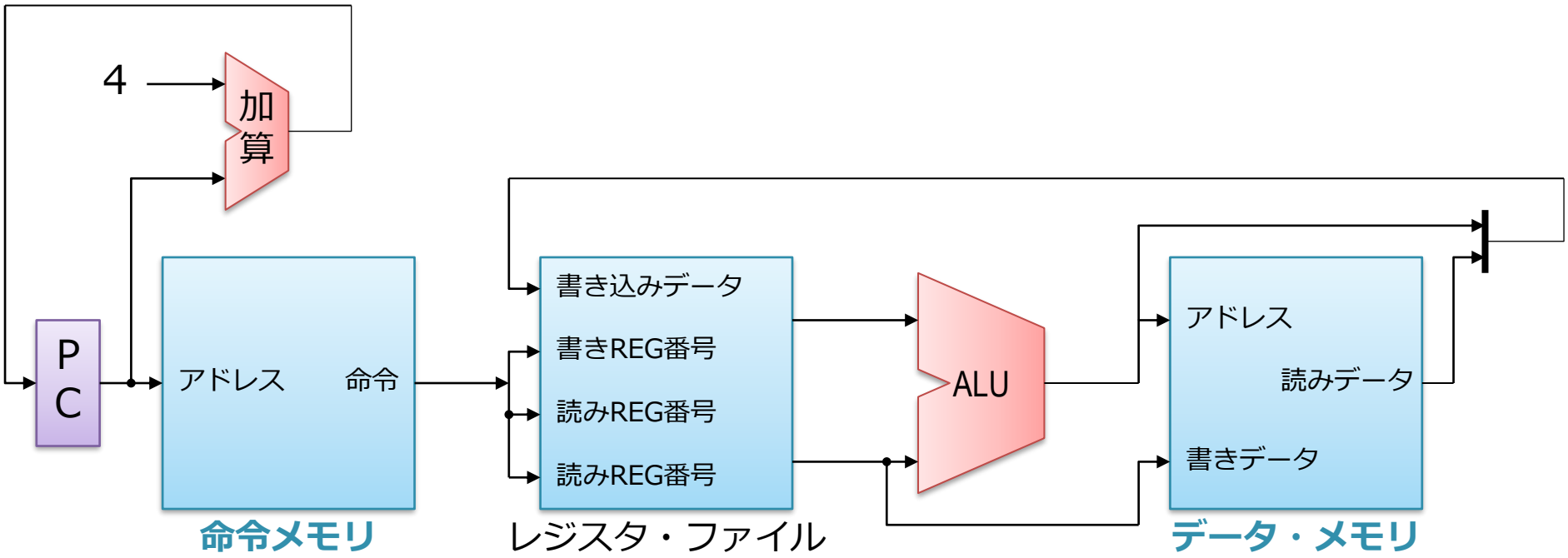


シングル・サイクル・プロセッサの動作

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



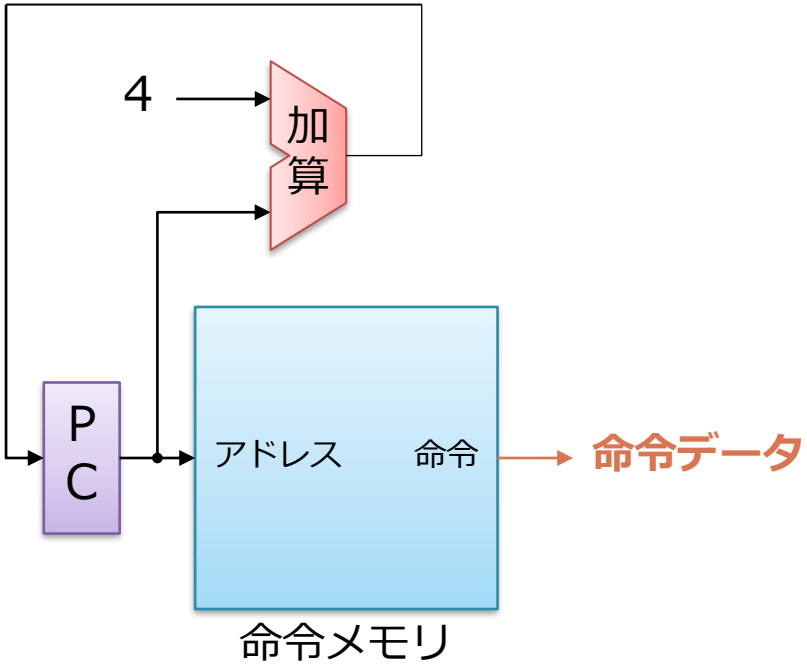
- 以前説明したものとの違い：
  - ◇ メモリが命令メモリとデータメモリに別れている
  - ◇ 算術 & 論理演算，ロード，ストアのみを実行可能
    - □ 分岐とジャンプは，簡単のために今は考えない

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

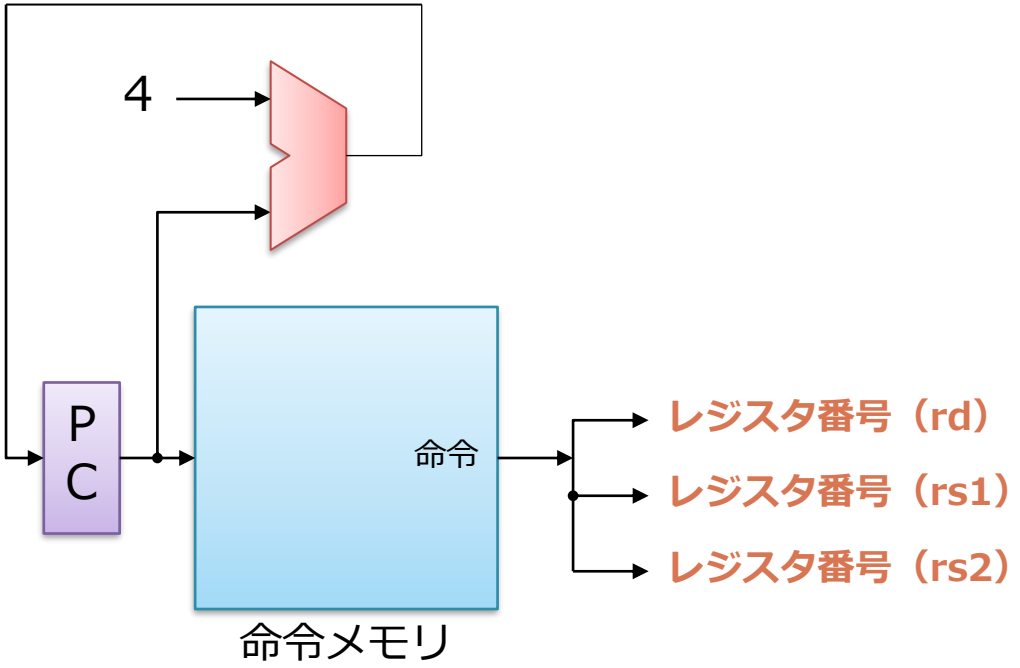
- RISC-V の加算命令を実行する流れをざっとみる

命令フェッチ

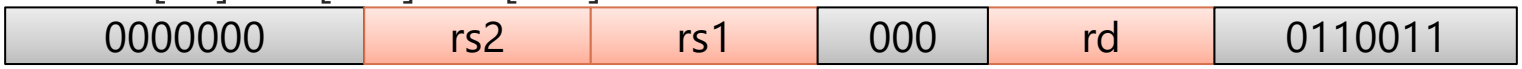


- 命令メモリから命令を読み出す
  - 命令メモリを順に読んでいくため，PC は毎サイクル加算される
  - 足している４は，RSIC-V では命令の幅が４バイトだから
  - 基本的に，この部分はどの命令でも変わらない

命令デコード

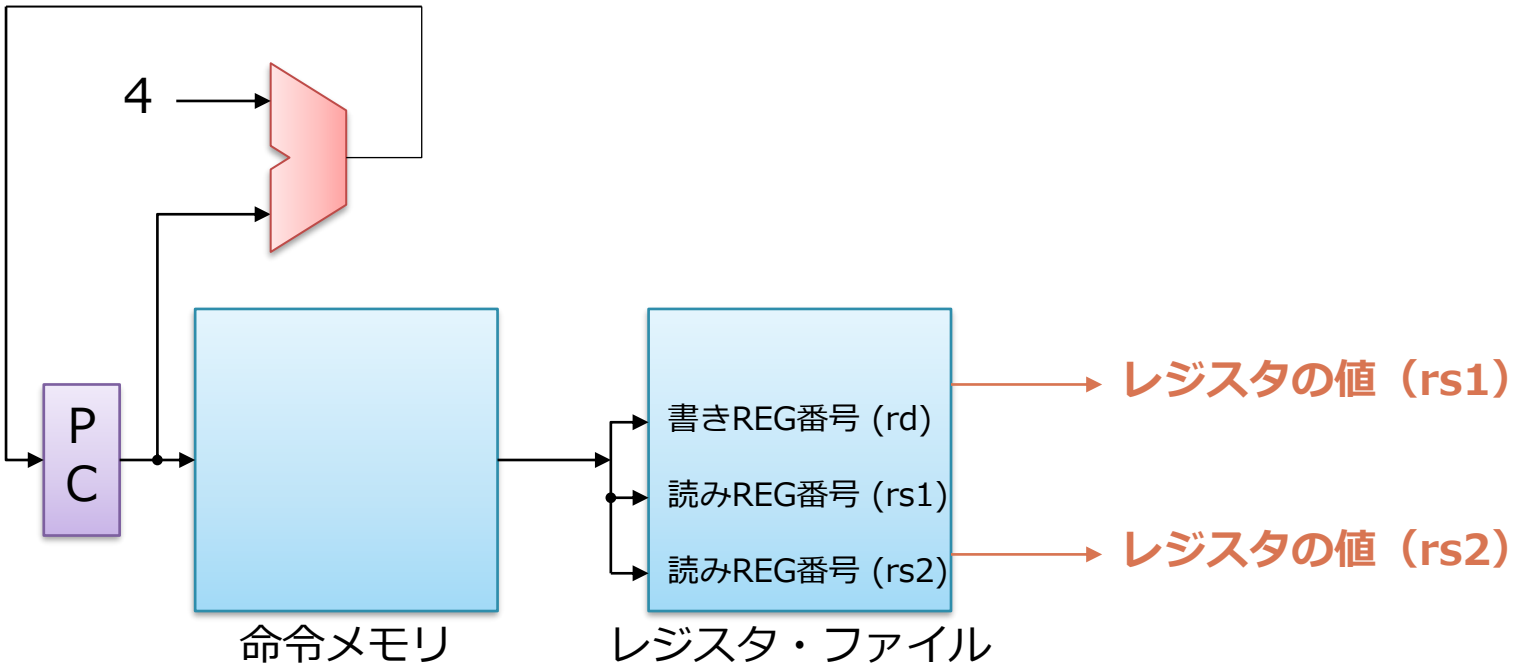


ADD : x[rd] ← x[rs1] + x[rs2]



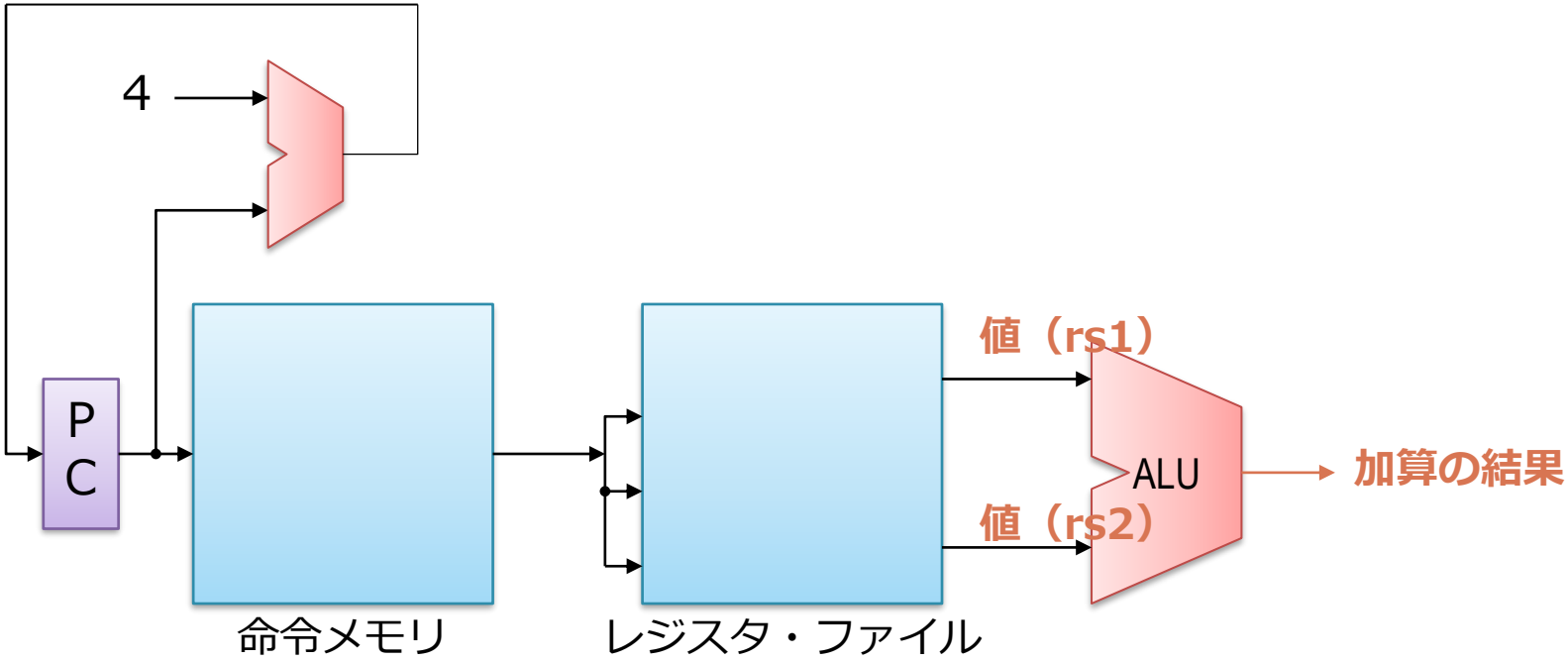
- 取り出した命令からレジスタ番号を表す部分のビットを取り出す
  - ◇ ソース（rs1, rs2）とディスティネーション（rd）

レジスタ読み出し



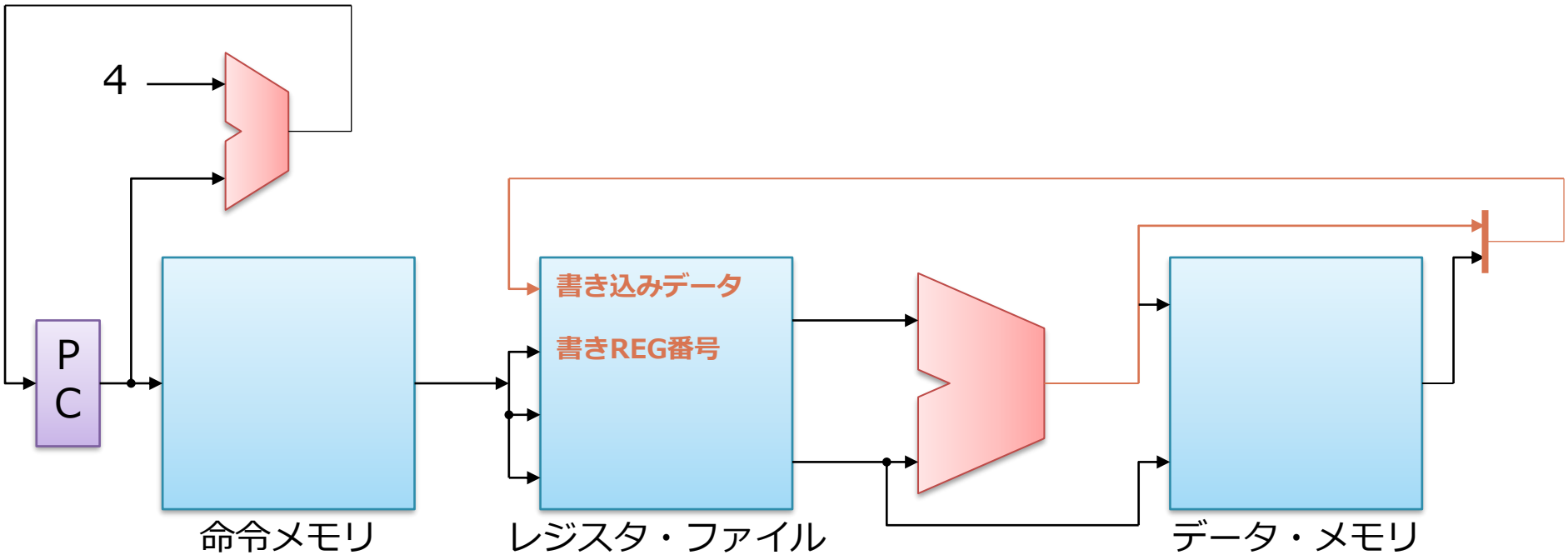
- デコードで得られたレジスタ番号を使って RF にアクセス
  - ソース・オペランドの値を読み出す

実行



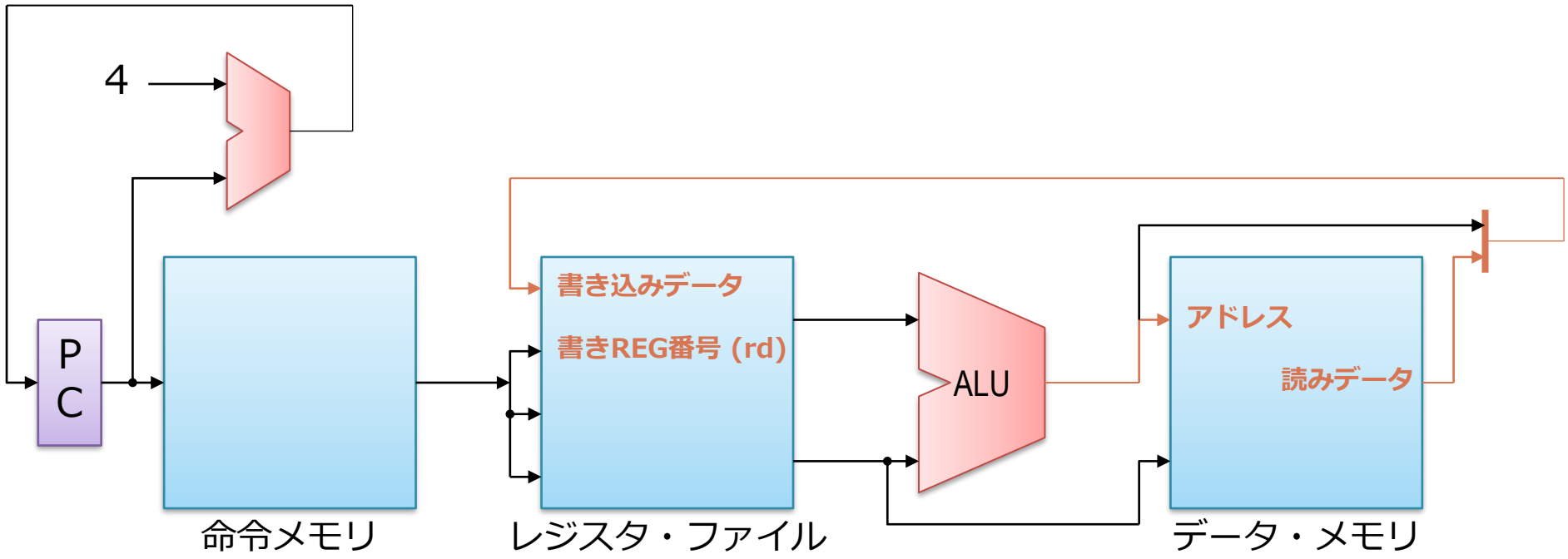
- RF から読みだした２つの値を加算

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

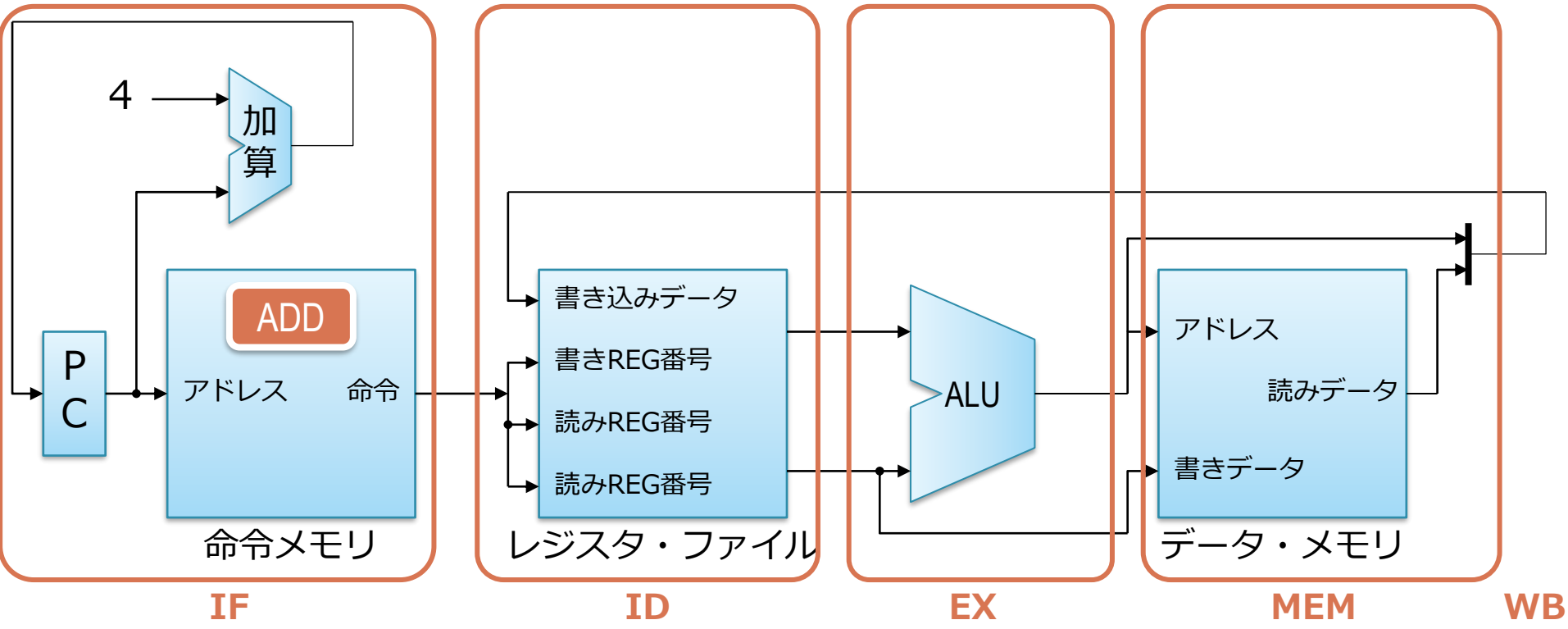


LW : x[rd] ← (x[rs1] + immediate)

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

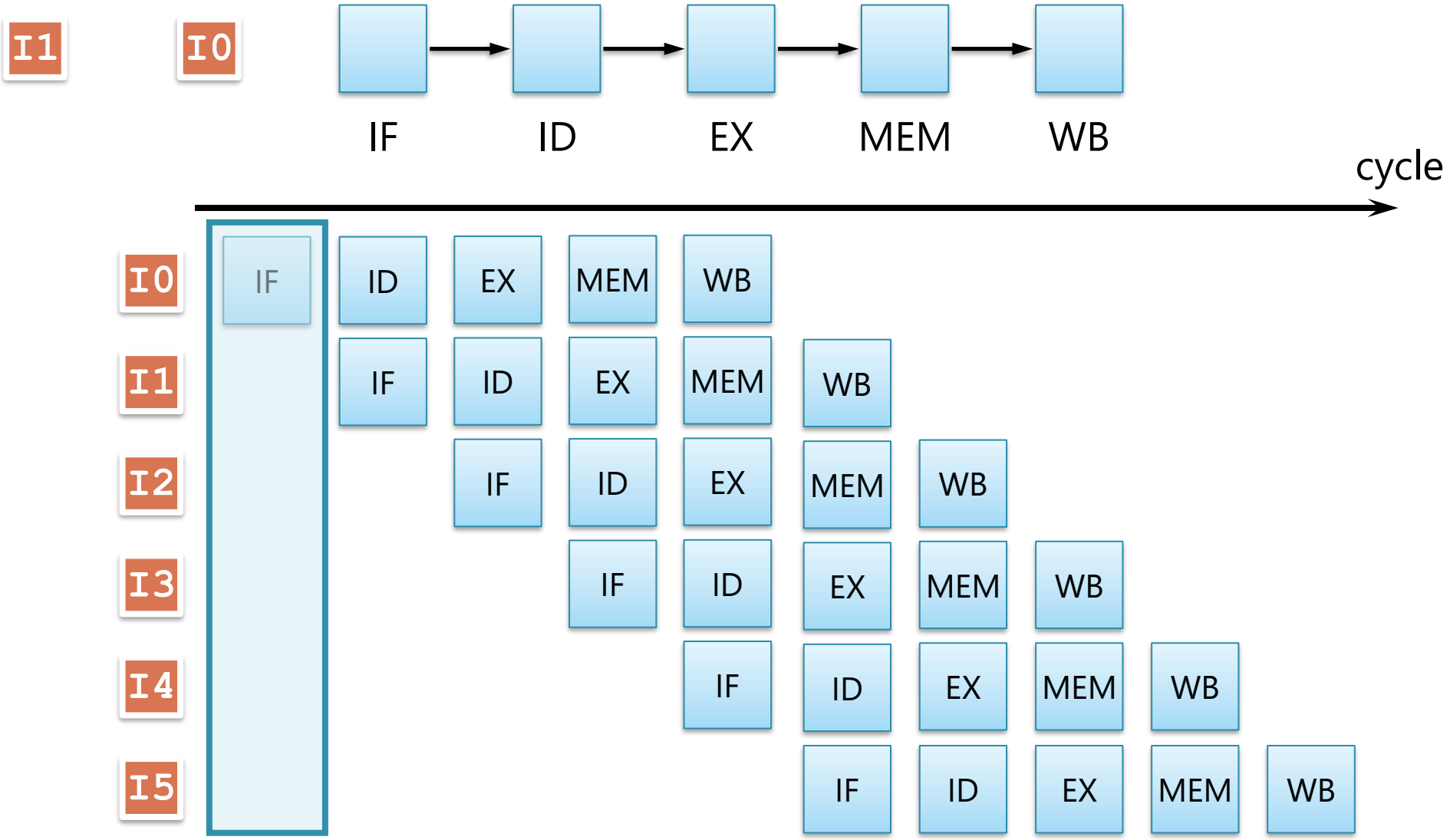
もくじ

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

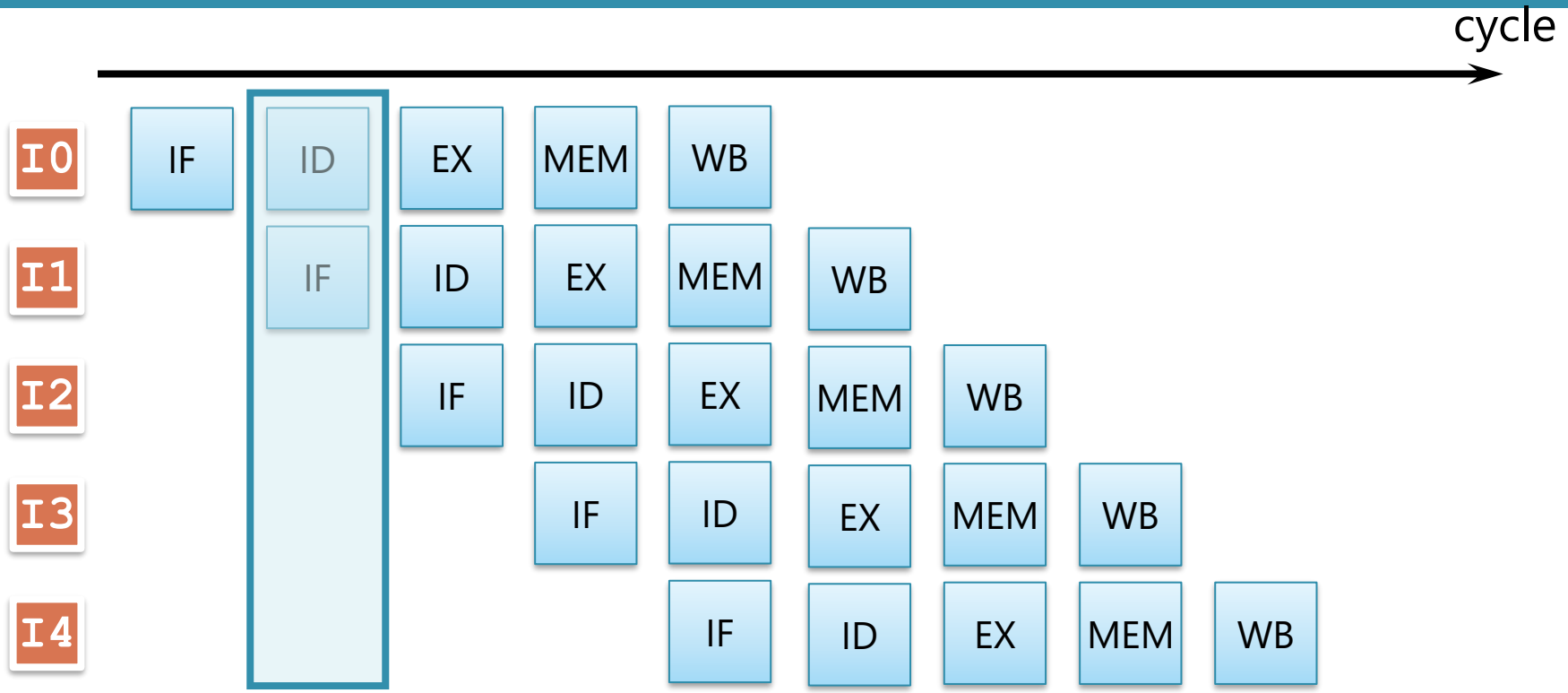
パイプライン化

- 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

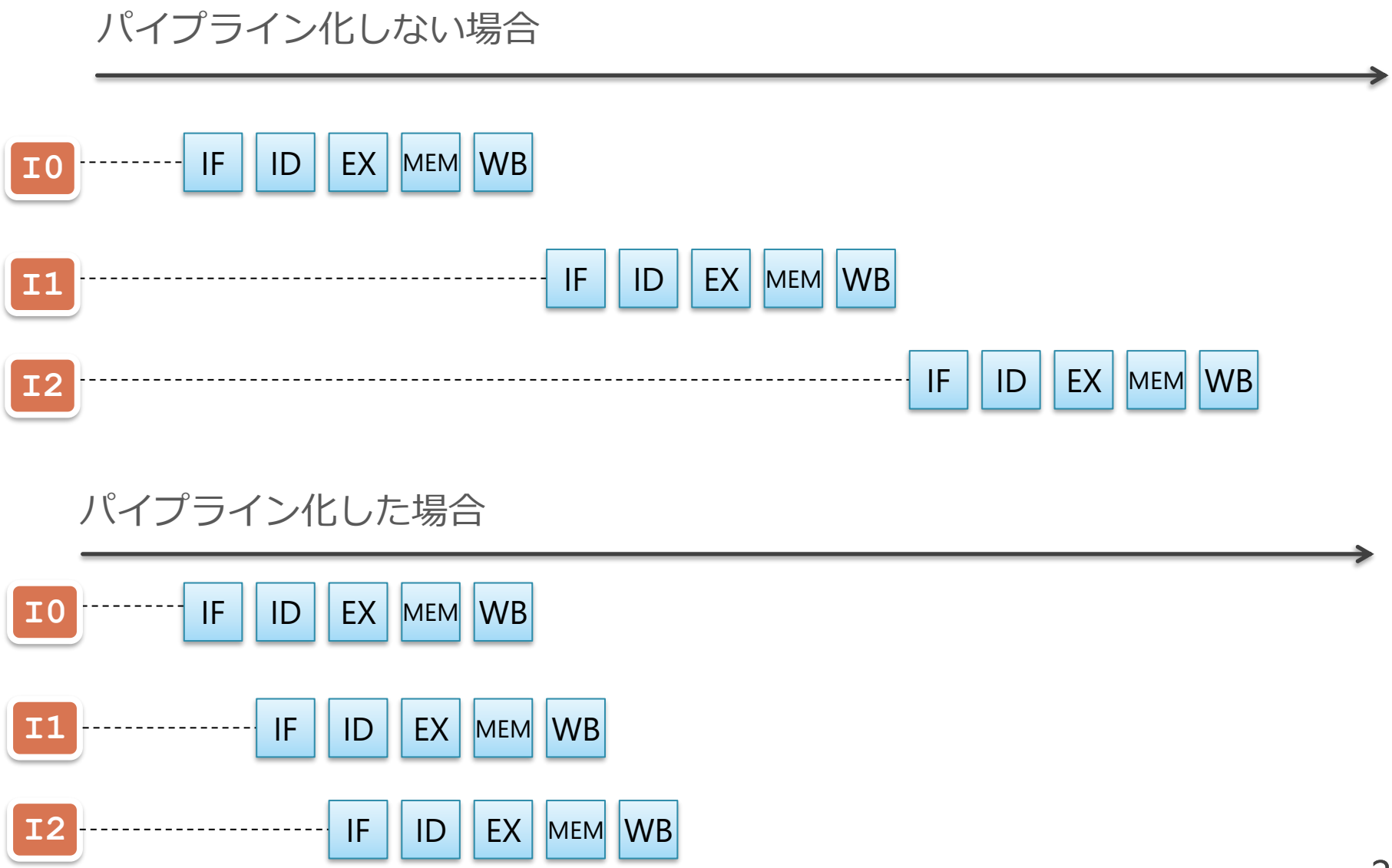


パイプライン・チャートの見方
ここから先で多用されるので重要

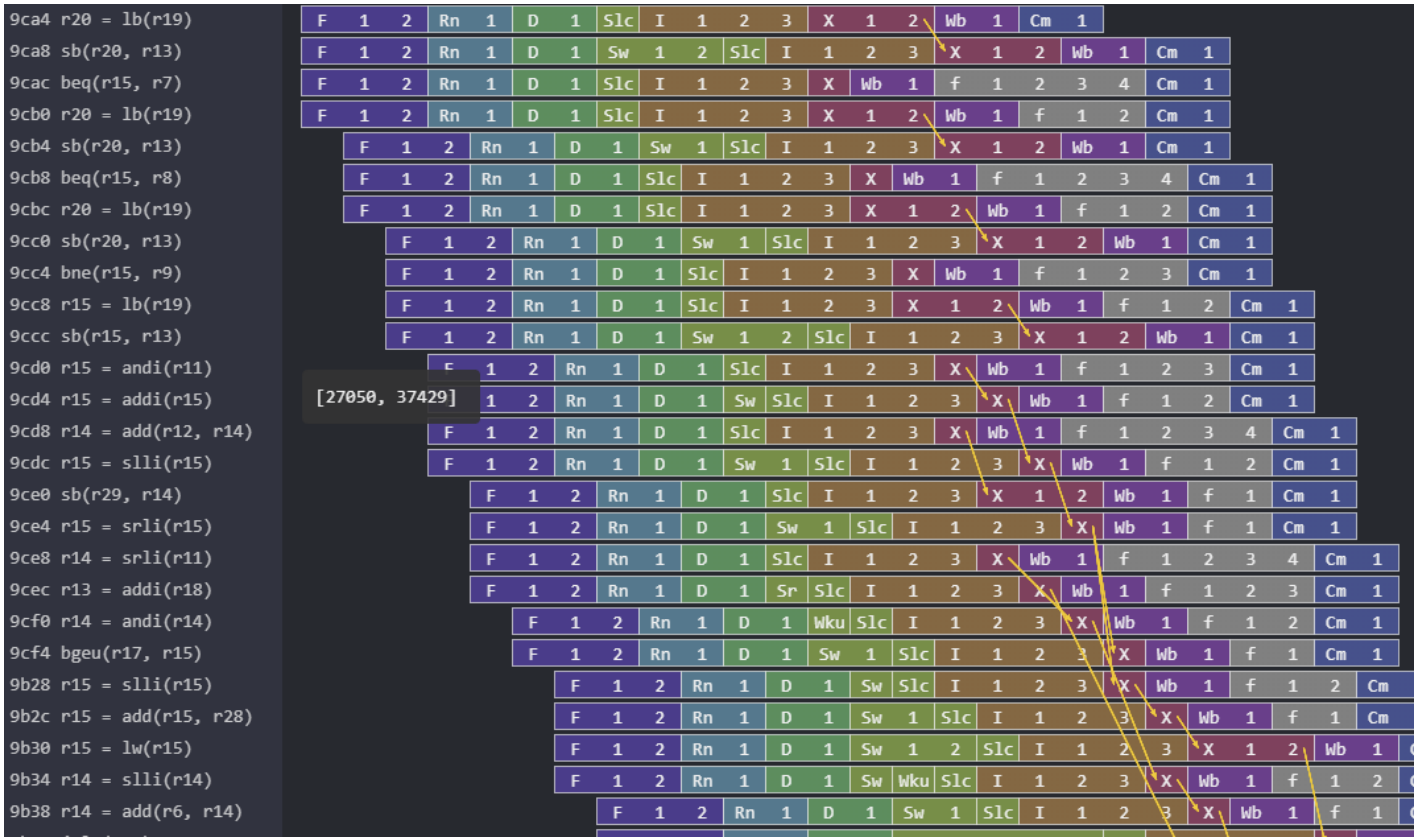


- 左から右にむかって時間は進む
- 上から下にむかって命令が実行順に置かれる
- 各ステージを表す四角は左側にある命令がその時そこにいることを示す
  - 上記では２サイクル目に，I0 が ID に，I1 が IF で処理されている

パイプライン化による性能（スループット）向上



余談：実際の CPU を実行した場合のパイプライン



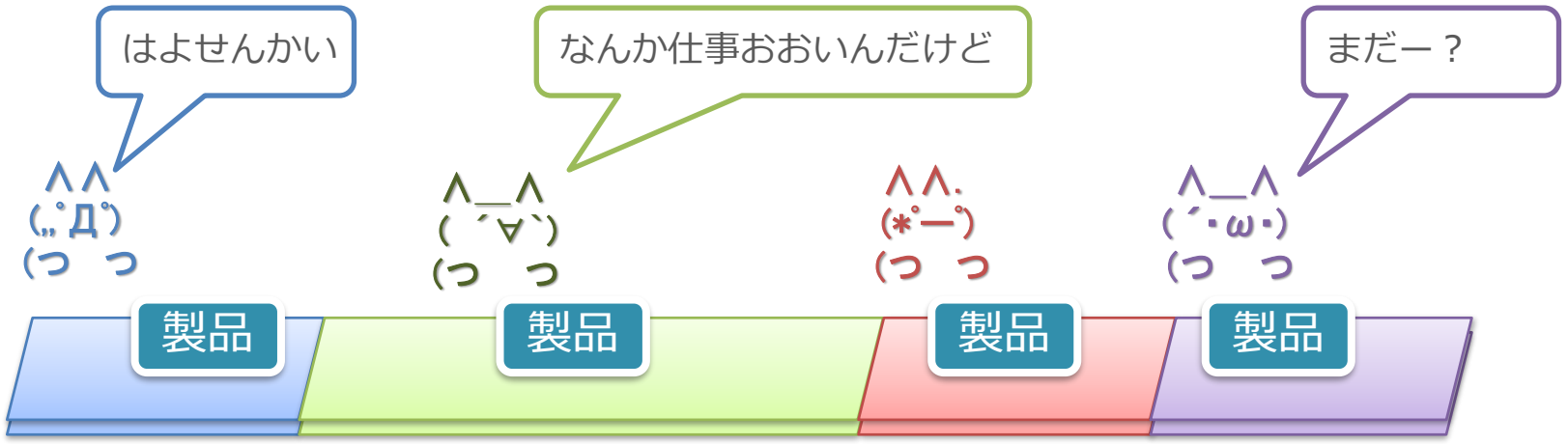
- 塩谷が開発している RISC-V CPU（RSD）の実行を可視化したもの
  - ◇ https://github.com/rsd-devel/rsd
  - ◇ out-of-order 実行をしているので，途中からプログラム順とは異なるタイミングで実行が進んでいる

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

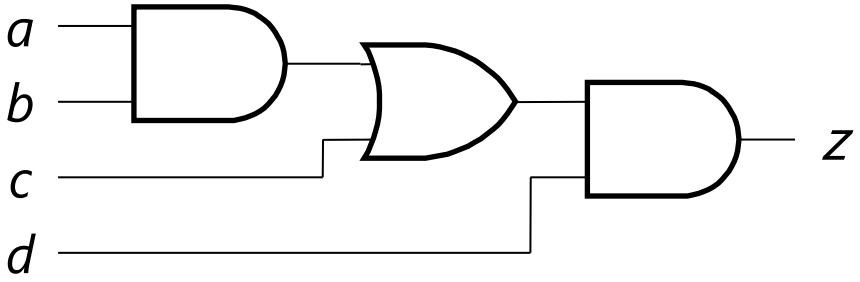
- 上記では，デコードとレジスタ読み出しが ID ステージにまとめられている
  - ◇ デコードにかかる遅延はほとんどない
  - ◇ 読み出した命令からオペランドを取り出すのは，単に信号線を繋ぐだけで良い

ステージを「どうやって」切るか

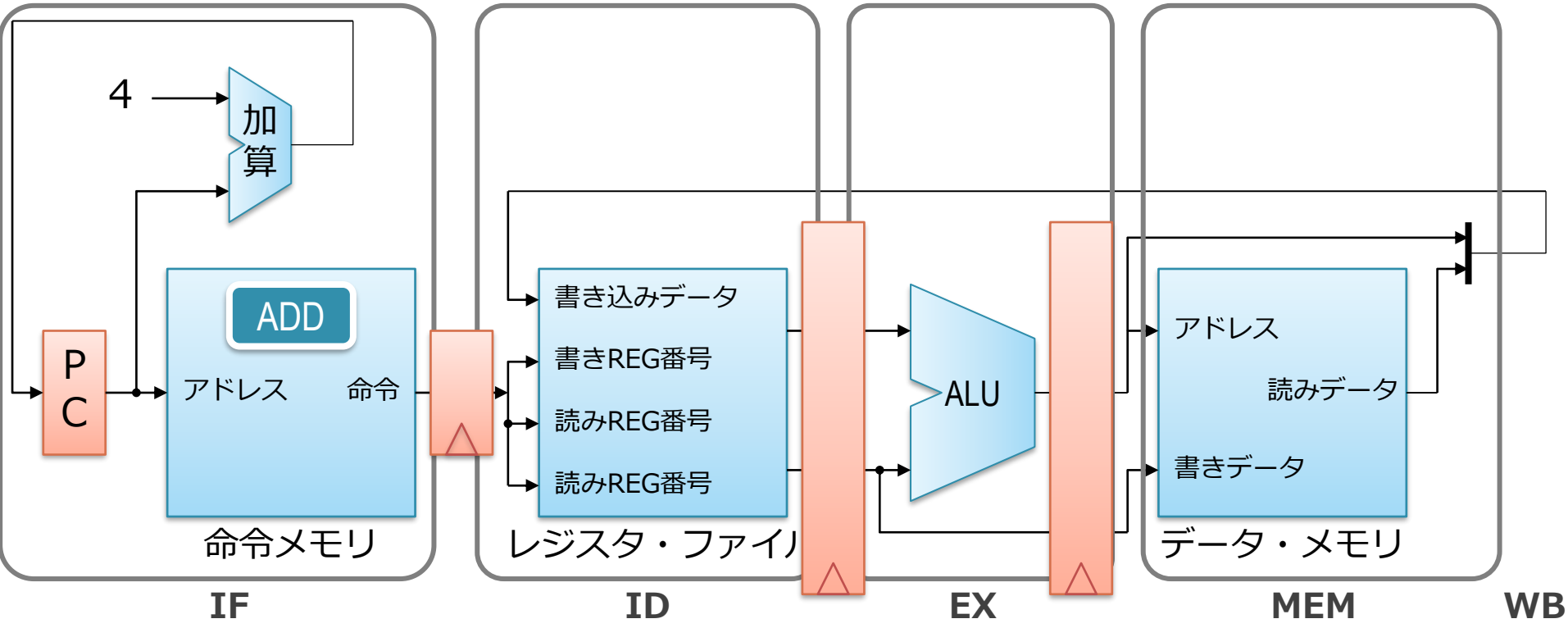
- シングル・サイクル・プロセッサの回路に適当に間隔をあけて命令を流せばよいというものもない

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

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

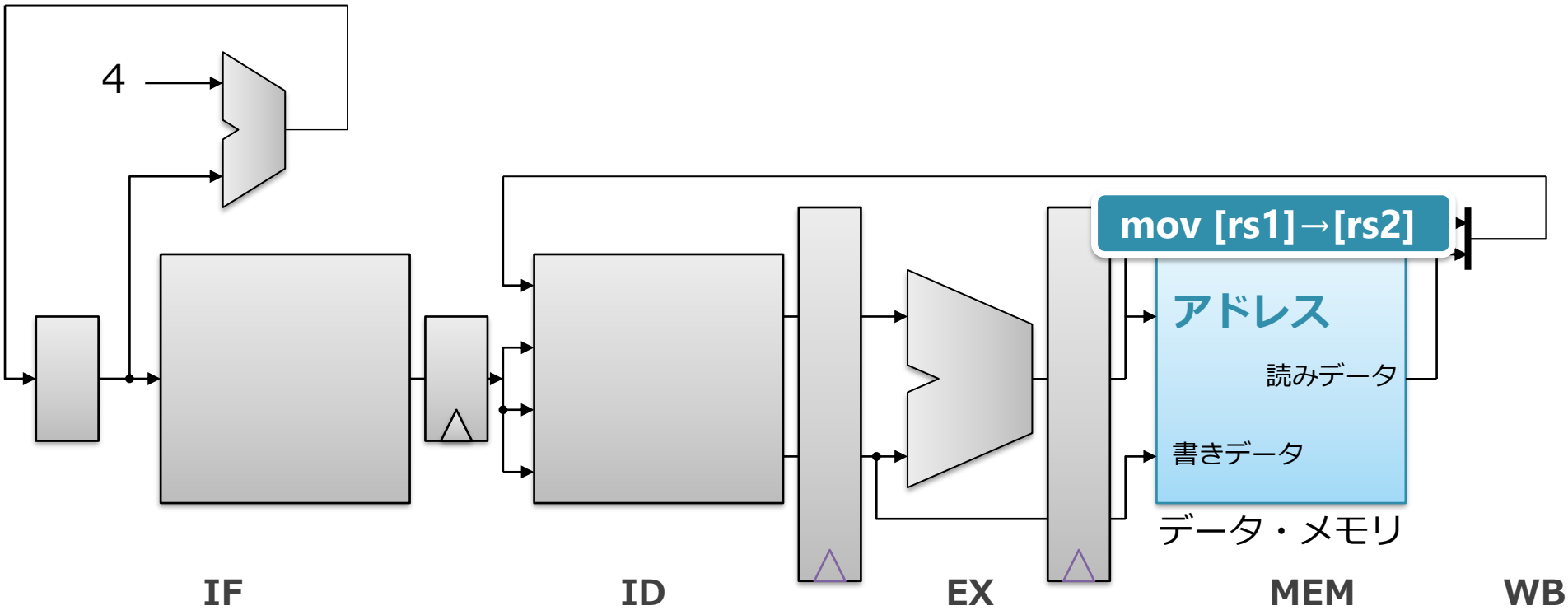
構造ハザード

- 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 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

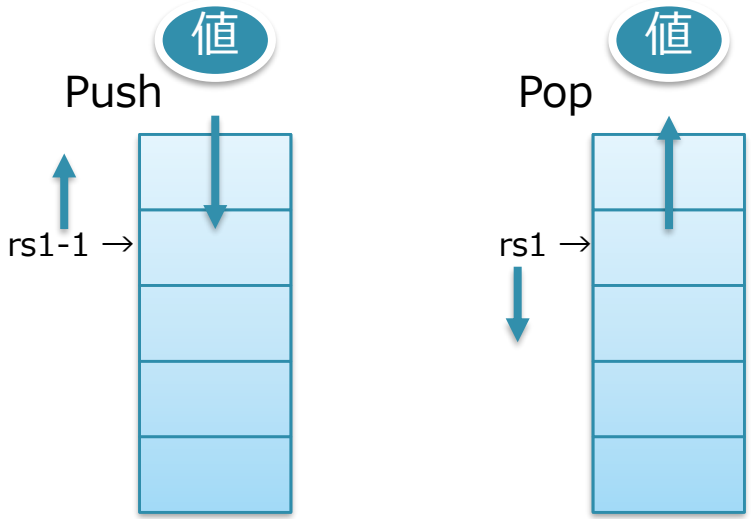
mov [rs1]→[rs2] // [rs1]→[rs2] へのコピー



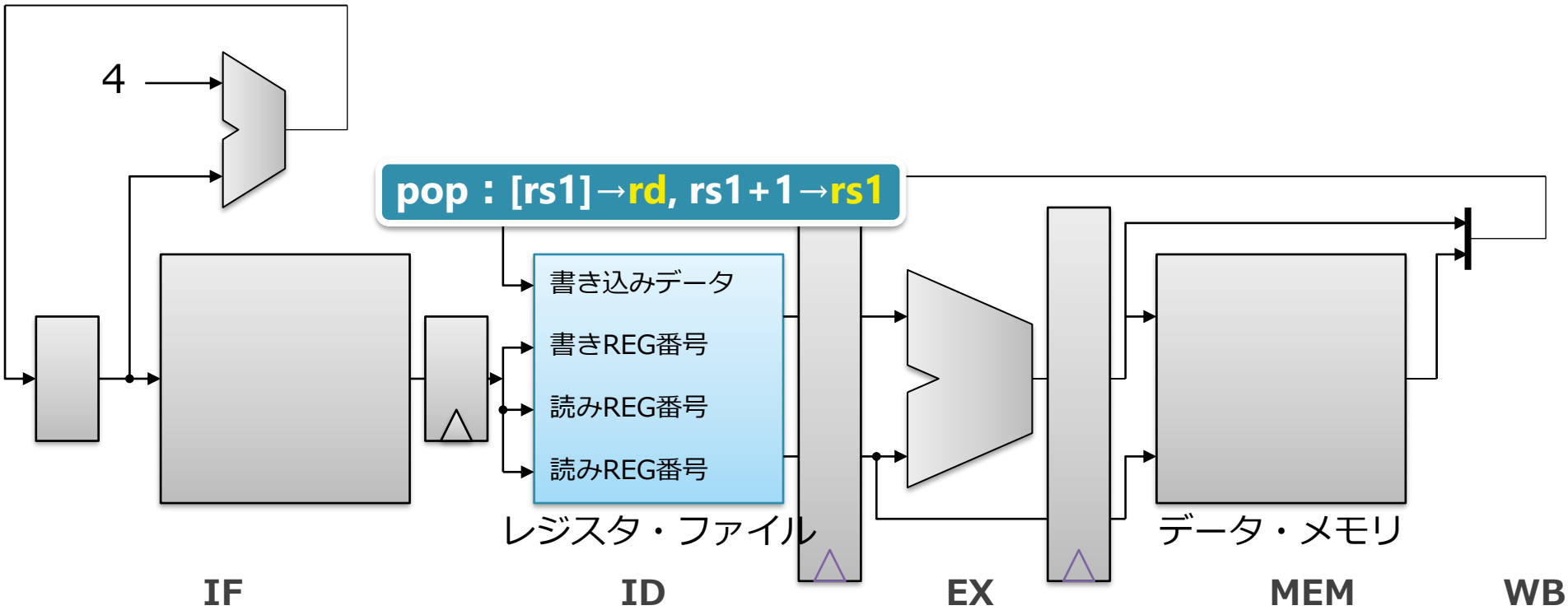
- メモリをあるサイクルに同時に読んで書く必要がある
  - ◇ しかし，データ・メモリのアドレスの口は1つしかない
  - ◇ MEM ステージでデータ・メモリの読みと書きが同時にできない

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



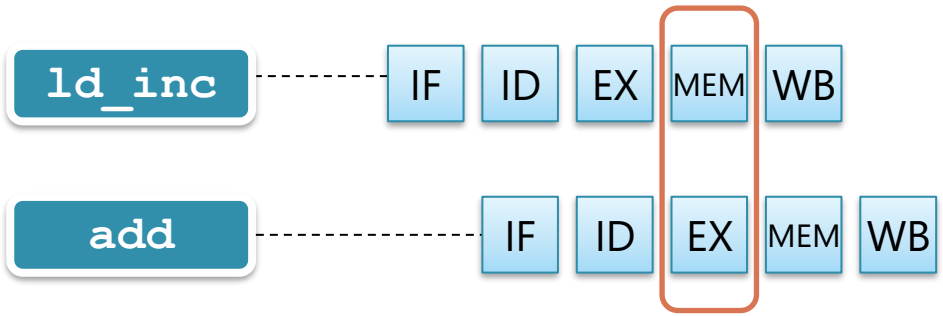
- WB ステージでレジスタに rd と rs1 の2つを書き込む必要がある
  - ◇ レジスタ・ファイルへの書き込みは，同時に２つはできない

構造ハザードの例３

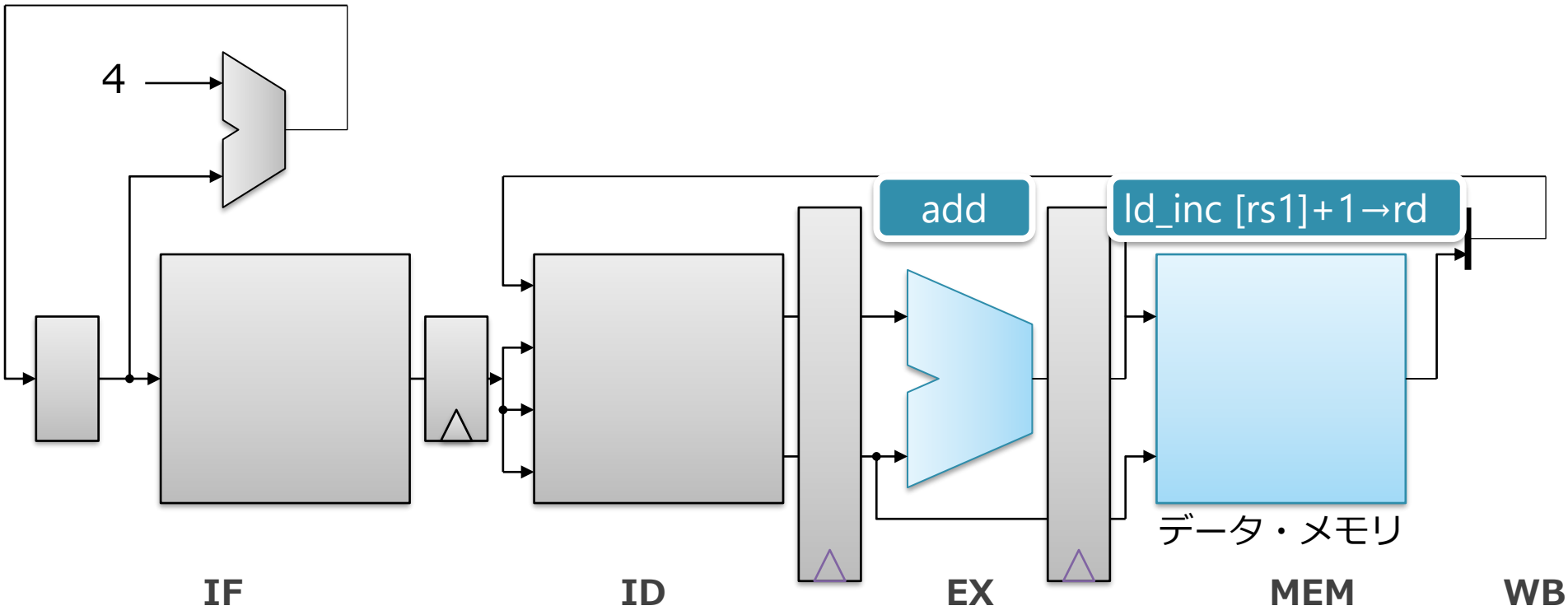
- 使用資源の異なるステージ間のぶつかりでも起きる
  - ◇ これまでの例は，同じステージ内で資源が足りない例
- **`ld_inc [rs1]+1→rd`** のような命令があったとする
  1. rs1 の指すアドレスからメモリを読む
  2. 読んだ値にさらに+１してから rd に書く
- 一見，資源は足りているようだが･･･
  - ◇ レジスタの読み書きは１つずつしかない
  - ◇ メモリも１カ所を読むだけ
  - ◇ 加算も１回行うだけ

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）

ARM（32ビット）の Load/Store Multiple (LDM/STM) 命令
ビットマスクで指定した最大16個のレジスタへのロードストアを行う
関数呼び出し/復帰の際の，レジスタの保存や復帰でよく使われる

A3.12 Load and Store Multiple instructions

Load Multiple instructions load a subset, or possibly all, of the general-purpose registers from memory.

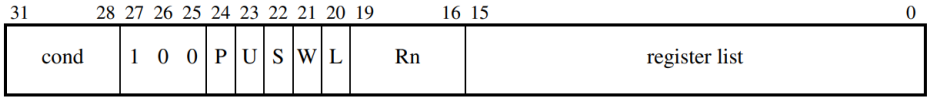
Store Multiple instructions store a subset, or possibly all, of the general-purpose registers to memory.

Load and Store Multiple instructions have a single instruction format:

```
LDM{<cond>}<addressing_mode>  Rn{!}, <registers>{^}
STM{<cond>}<addressing_mode>  Rn{!}, <registers>{^}
```

where:

```
<addressing_mode> = IA | IB | DA | DB | FD | FA | ED | EA
```



register list The list of <registers> has one bit for each general-purpose register. Bit 0 is for R0, and bit 15 is for R15 (the PC).

The register syntax list is an opening bracket, followed by a comma-separated list of registers, followed by a closing bracket. A sequence of consecutive registers can be specified by separating the first and last registers in the range with a minus sign.

P, U, and W bits These distinguish between the different types of addressing mode (see *Addressing Mode 4 - Load and Store Multiple* on page A5-41).

S bit For LDMs that load the PC, the S bit indicates that the CPSR is loaded from the SPSR after all the registers have been loaded. For all STMs, and LDMs that do not load the PC, it indicates that when the processor is in a privileged mode, the User mode banked registers are transferred and not the registers of the current mode.

L bit This distinguishes between a Load (L==1) and a Store (L==0) instruction.

Rn This specifies the base register used by the addressing mode.

A3.12.1 Examples

```
STMFD    R13!, {R0 - R12, LR}
LDMFD    R13!, {R0 - R12, PC}
LDMIA    R0, {R5 - R8}
STMDA    R1!, {R2, R5, R7 - R9, R11}
```

- ARM Architecture Reference Manual より

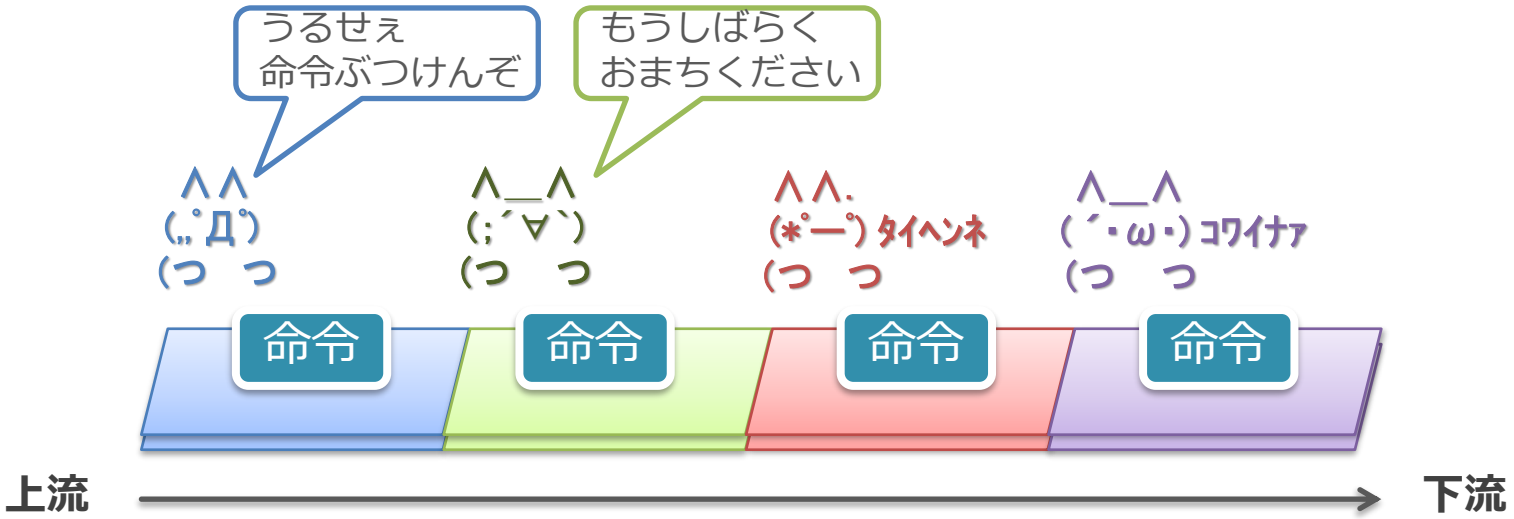
構造ハザードの解決方法

- 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

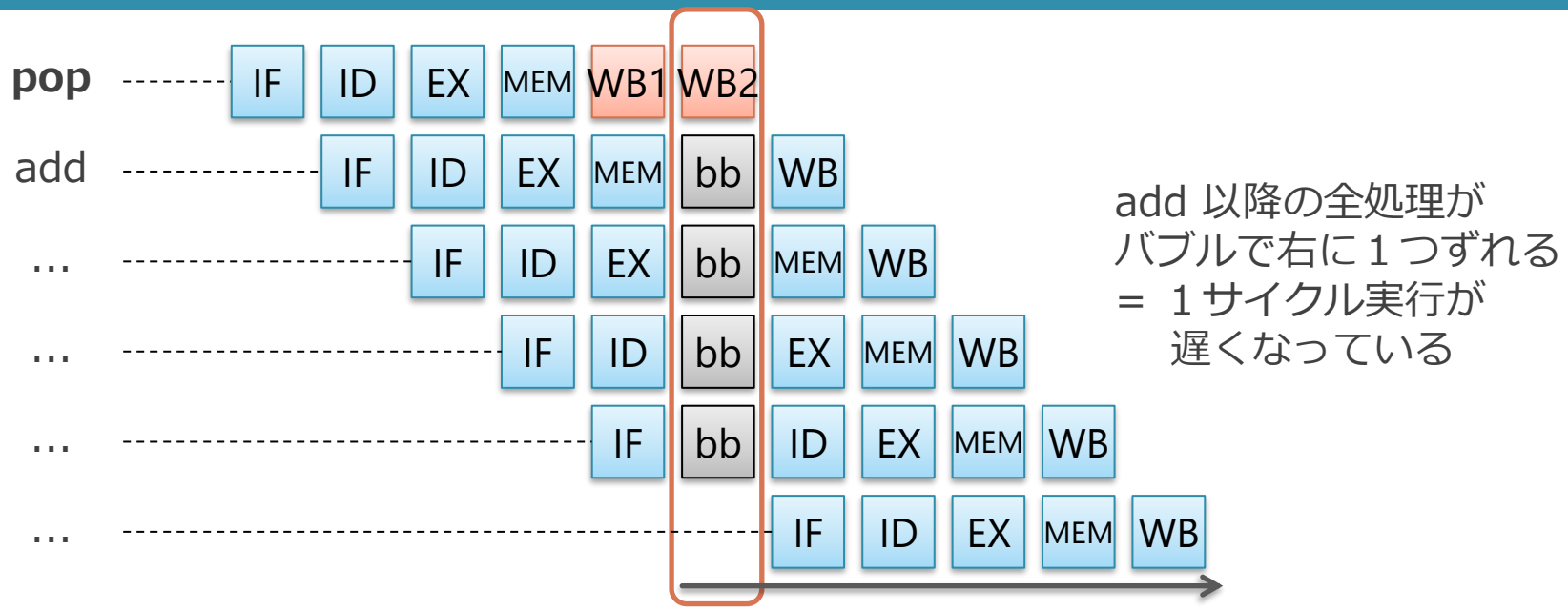


- 上流を止めないと破綻する
  - ◇ (;´∀`) が複数サイクルをかけて仕事をしている場合，命令はそこにとどまり続ける
  - ◇ その間は上流をとめないと命令をおく場所がないし，依存関係がまもられない
- (*ﾟー) より下流は流れていっても，この場合は問題ない

パイプラインを止めること

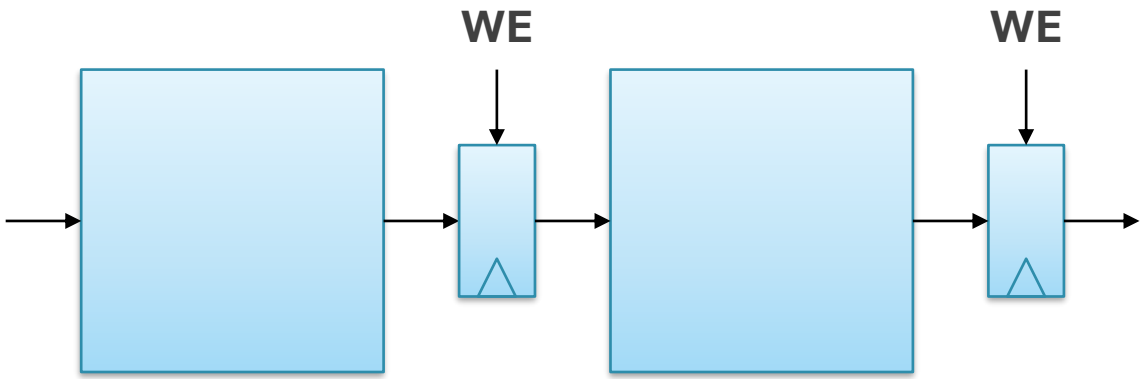
- パイプラインを止めるとことを「ストール」や「インターロック」という
  - ◇ 本や人によって，意味や使い方が微妙に統一されていない
  - ◇ この講義では，以降はストールで統一

ストールの動作



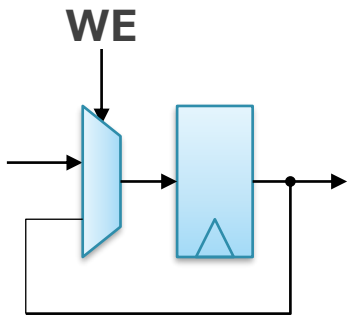
- pop： 1 つレジスタに書いたあと，次のサイクルで書き込む
  - ◇ WB1 と WB2 の 2 サイクルで書き込む
  - ◇ WB2 の間は上流を全て止める
- パイプライン・チャート上では上記のようになる
  - ◇ 止める原因の命令の下が全部右にずれる
  - ◇ ずれた部分の空き（bb）を「バブル」とよぶ

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

マイクロ命令の分解の例

- 解析結果をまとめているサイトも
  https://uops.info/table.html
  下の例は，x86-64 の CMOVBE 命令の Alder Lake 大きいコアでの分解の説明

CMOVBE (R64, M64)

Summary:	"Conditional Move"
Reference:	https://www.felixcloutier.com/x86/CMOVcc.html
Extension:	BASE
Category:	CMOV
ISA-Set:	CMOV
CPL:	3
iform:	CMOVBE_GPRv_MEMv
iclass:	CMOVBE
ASM:	CMOVBE

Operands

- Operand 1 (w): Register (RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, R8, R9, R10, R11, R12, R13, R14, R15)
- Operand 2 (r): Memory
- Operand 3 (r, suppressed): Flags (CF: r, ZF: r)

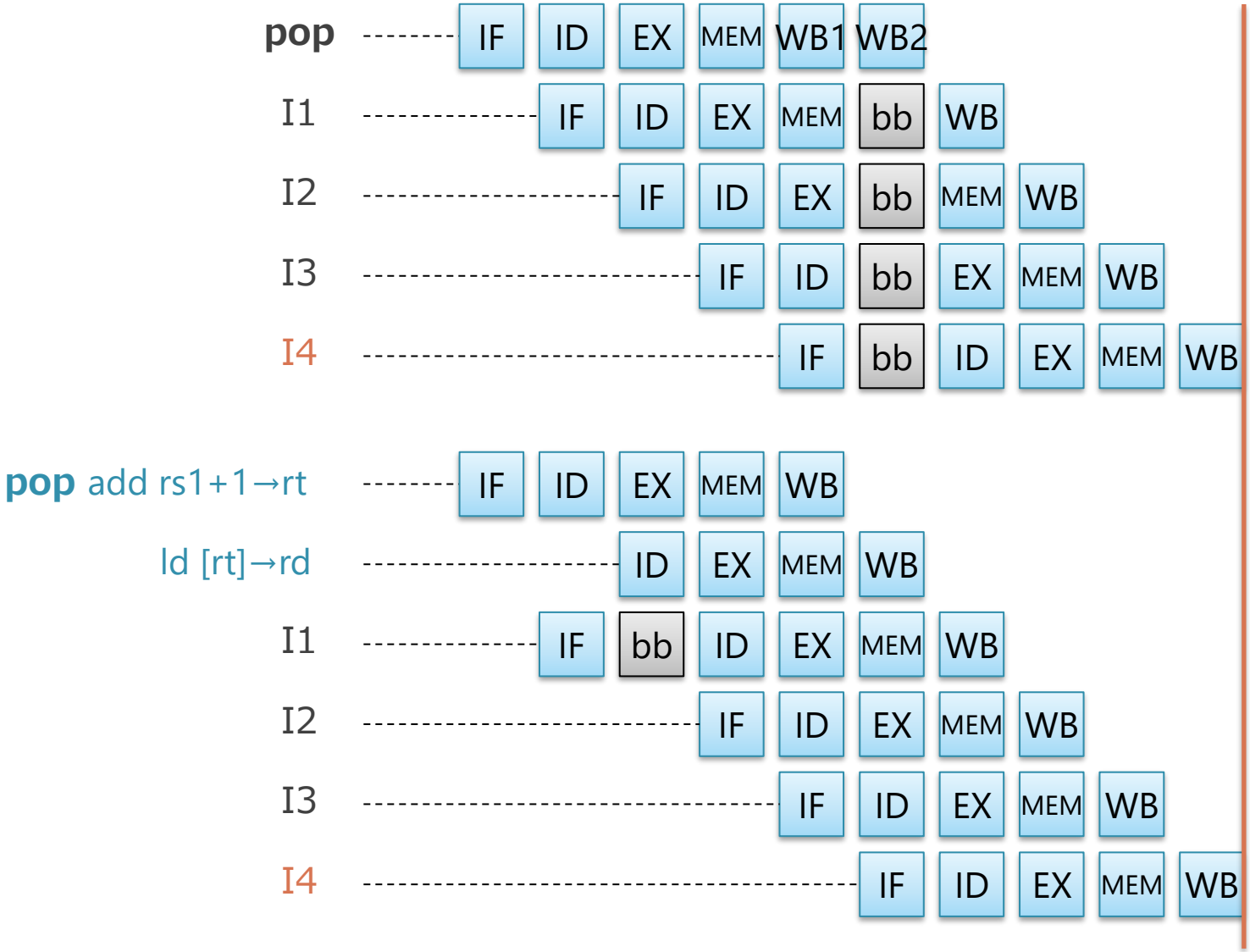
Available performance data

- Alder Lake-P
- Alder Lake-E
- Rocket Lake
- Tiger Lake

Alder Lake-P

- Measurements
  - Latencies
    - Latency operand 1 → 1: 1
    - Latency operand 2 → 1 (address, base register): 6
    - Latency operand 2 → 1 (address, index register): 6
    - Latency operand 2 → 1 (memory): ≤6
    - Latency operand 3 → 1: 2
  - Throughput
    - Computed from the port usage: 1.00
    - Measured (loop): 1.02 (if an indexed addressing mode is used: 1.00)
    - Measured (unrolled): 1.00
  - Number of µops
    - Executed: 3
    - Retire slots: 2 (if an indexed addressing mode is used: 3)
    - Decoded (MITE): 2
    - Microcode Sequencer (MS): 0
    - Requires the complex decoder (4 other instructions can be decoded with simple decoders in the same cycle)
  - Port usage: 2*p06+1*p23A

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

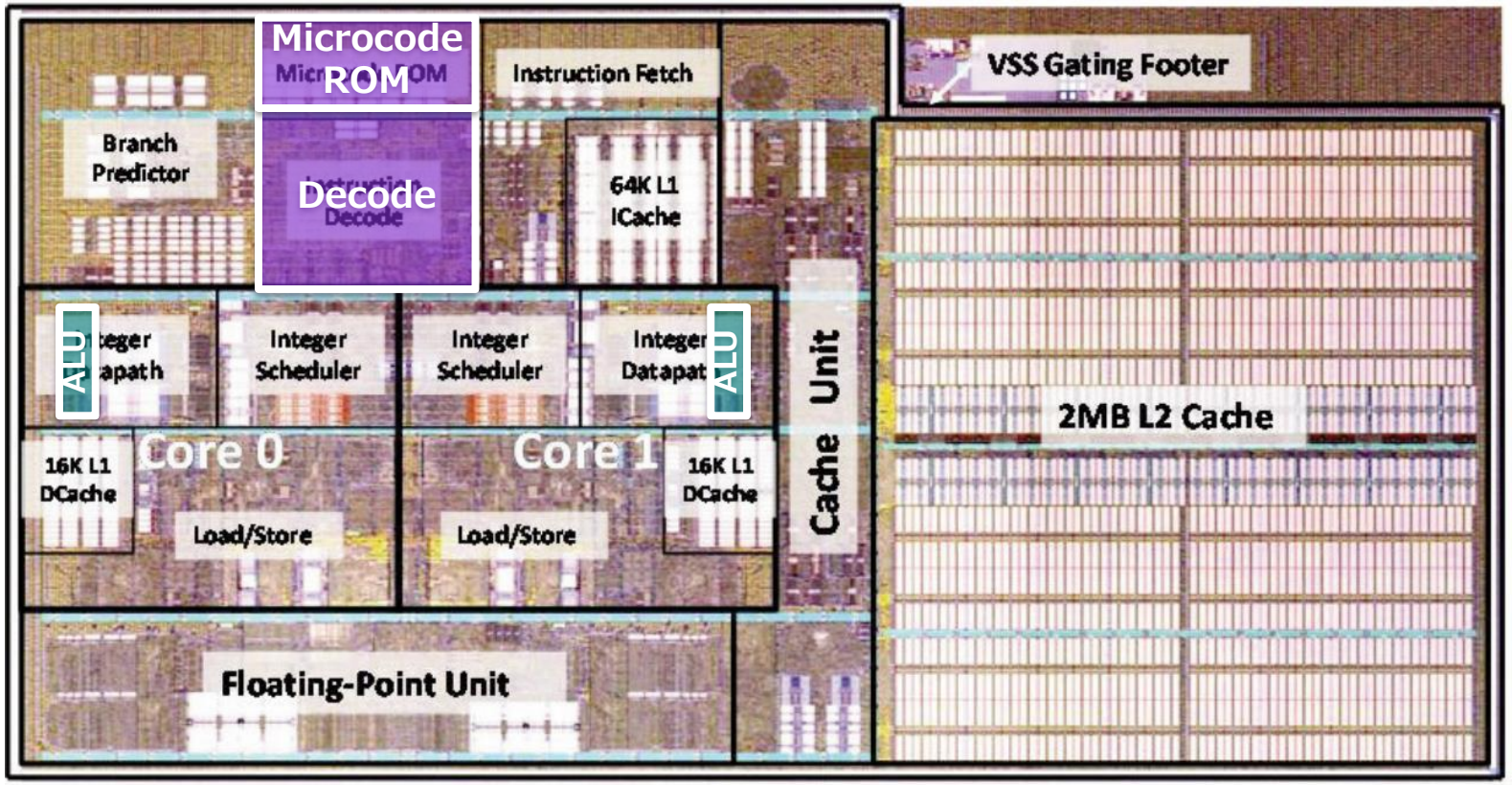
マイクロ命令への分解の利点

- 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

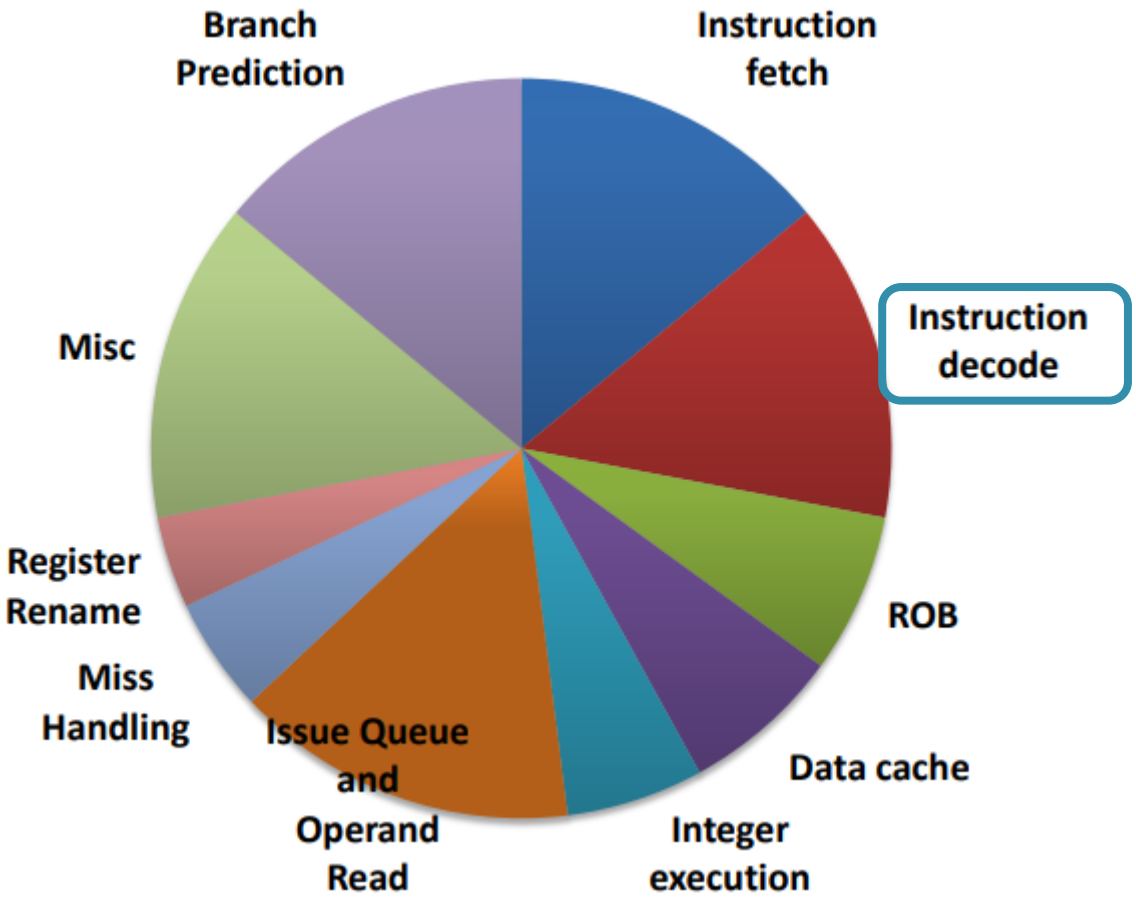
- 分解（デコード）がマジで大変
- 基本的には，ひたすらパターン・マッチング
  - ◇ でかい真理値表がいる
  - ◇ 本当に複雑なものは，メモリで出来たテーブルも使う

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，違う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
  - ◇ ただし，元の命令の動作を他の複数の命令で再現したりするので，かなり遅くなる事もある

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 Summary Table

Erratum ID	Processor Line/Stepping							Title
	UP3	IOT UP3	UP4	H35	H81	UP3-Refresh	H35-Refresh	
TGL001	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	X87 FDP Value May be Saved Incorrectly In Real-Address Mode or Virtual-8086 Mode
TGL002	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	Debug Exceptions May be Lost or Misreported When MOV SS or POP SS Instruction is Not Followed by a Write to SP
TGL003	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	CPUID L2 Cache Information May be Inaccurate
TGL004	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	Placing Posted-Interrupt Descriptors Within the PRMRR May Result In a Processor Hang
TGL005	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	Intel® PT CBR Packet May be Delayed or Dropped
TGL006	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	Intel® PT TIP or FUP Packets May be Dropped Without OVF Packet
TGL007	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	Overflow Flag in IA32_MC0_STATUS MSR May be Incorrectly Set
TGL008	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	An Exception During a 32-bit Mode Task Switch With CET Enabled May Lead to an Incorrect TSS Busy Flag Value
TGL009	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	Exit Qualification For EPT Violations Incorrectly Indicate On Instruction Fetches That the Guest-Physical Address Was Writeable
TGL010	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	Processor May Generate Spurious Page Faults On Shadow Stack Pages
TGL011	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	HDMI 1.4 Inter-Pair Skew Test May Fail
TGL012	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	Intel® PT ToPA Tables Read From Non-Cacheable Memory During an Intel® TSX Transaction May Lead to Processor Hang
TGL013	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	Performing an XACQUIRE to an Intel® PT ToPA Table May Lead to Processor Hang
TGL014	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	PECI Frequency Limited to 3.2Kbps-1Mbps
TGL015	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	No Fix	PCIe Gen4 jTOL – Jitter Tolerance Compliance Test May Fail
TGL016	Fixed	Fixed	N/A	N/A	N/A	N/A	N/A	System May Fail To Exit Warm Reset or S3

TGL063	Branch Predictor May Produce Incorrect Instruction Pointer
Problem	Under complex microarchitectural conditions, the branch predictor may produce an incorrect instruction pointer leading to unpredictable system behavior.
Implication	Due to this erratum, the system may exhibit unpredictable behavior.
Workaround	It may be possible for BIOS to contain a workaround for this erratum.
Status	For the steppings affected, refer to the Summary Table of Changes.

TGL064	Processor May Encrypt TME Exclude Range if Mapped to Remap Range
Problem	The processor accesses to TME exclude range may be encrypted but not decrypted if mapped to remap range.
Implication	Due to this erratum, the processor exclude range it may be encrypted but may but not decrypted if mapped to remap range.
Workaround	It may be possible for BIOS to workaround this erratum.
Status	For the steppings affected, refer to the Summary Table of Changes.

TGL065	xHCI Force Header Command Incorrect Return Code
Problem	The xHCI controller does not return the correct completion code for the Force Header Command as defined in the Section 4.6.16 of the eXtensible Host Controller Interface for Universal Serial Bus (xHCI) Requirements Specification Rev 1.2.
Implication	xHCI CV TD4.12 - Force Header Command Test may report an error. Intel® has obtained a waiver for TD 4.12. The Force Header Command is only used by the USB-IF Command Verifier (xHCI CV) tool for device testing. There are no known functional failures due to this erratum.
Workaround	None identified.
Status	For the steppings affected, refer to the Summary Table of Changes.

TGL066	USB Type-C Monitor Removal May Result In System Hang

- https://edc.intel.com/content/www/us/en/secure/design/confidential/products-and-solutions/processors-and-chipsets/tiger-lake/11th-generation-intel-core-processor-family-specification-update/summary-tables-of-changes/#_Summary_Tables_of 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
  - ◇ 富岳の A64FX 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

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（ここは次回
    - データ・ハザード
    - 制御ハザード
- 来週は非構造ハザードと分岐予測など

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
  - ◇ 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っ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

質問や感想

質問とか回答など

- クロックジッターのによる誤差がクロックサイクルに占める割合が大きくなるため、クロック周波数が増加し続けることができない原因でしょうか。
  - ◇ 電力や熱の制約の方が支配的だと思います

質問とか回答など

- 現代のCPUはより大きな面積ではなく、より小さなプロセスを優先している、これは消費電力やクロック遅延のためでしょうか。
  - ◇ 物理的なチップの面積は歩留まりやコストの問題が大きいです
  - ◇ 論理的な回路量は，プロセスの改良により増えています

質問とか回答など

- 自分は研究室でカーボンナノチューブの物性に関する研究を行っています。塩谷先生は「超微細ナノカーボン・プロセッサのアーキテクチャに関する研究」と題してCNFETの応用に向けた研究をされていたようですが、授業で紹介されたCMOSFETと比べてどのような違いがあったのでしょうか？

  - 電流の

質問とか回答など

- ノイマン型コンピュータの弱点であるノイマンボトルネックは、メモリとCPUをバスで繋ぐという構造上にあると言われていますが、これはバス上を通じて読み書きをしているということが何らかの弱点を生み出しているということでしょうか？

  - ◇ キャッシュとか SIMD により既に克服されている

- また、この弱点を解消する非ノイマン型コンピュータを用いることで消費電力が１万分の１程度になると説明する論文があると以前耳にしたのですが、どのような原理でしょうか？

  - ◇ 加算や乗算みたいな本質的に必要な演算だけで数割以上はくってるので，非ノイマン型にしても１万分の１には絶対ならない

質問とか回答など

- top500の上位10個あたりを見るとGPUを載せたシステムが多い理由が気になっています。ゴリゴリに作り込んだASICの方が、制御命令やデータ転送の無駄が少なくなり性能が出るのかなと思っていたのですが、何か別の要素があるのでしょうか？
  - ◇ Top500 のような倍精度 FP 演算がメインの場合，SIMD 構造にすることで，それらの無駄は無視できるレベルまで削減できます

質問とか回答など

- トランジスタのサイズを1/Kにスケーリングすると電圧が1/Kになるのはよく分からなかった.
  - ◇ 電圧が 1/K に勝手になるのではなく，1/K に下げても速度が保てていたのでそうしていたと言う事になります

質問とか回答など

- 近年自動運転車等でコンピュータの消費電力が重要視されている理由はなぜでしょうか

質問とか回答など

- FPGAでのハードウェア開発について、FPGAに適した回路を作るときもかなり開発コスト(時間など)がかかると聞きますが、実際どのくらい大変なのでしょうか。

質問とか回答など

- FPGAはリアルタイム性が求められるところで使われる印象を抱いていたのですが，直接回路作るより遅延大きいけど，結局並列処理能力（で低遅延になる）と回路の規模感のトレードオフってことですかね？論文読んでみます！

質問とか回答など

- 消費電力の話が面白かったです。ヒーターのように電力を消費して熱に変えているだけなのに情報処理ができるというのはとても興味深いです。

質問とか回答など

- 高速化をしようとすると結局LUTを使うことになるのだなぁと思っていました．別のアルゴリズムの授業で，完全なLUTではなく不完全なものでもそれなりに高速化できることがあると見たので，そのような発想がハードウェアの層にもあるのか気になりました．

質問とか回答など

- 半導体業界ではTSMCの工場建設などが話題です。みてみるとより半導体のサイズを小さくしようとする流れのように思うのですが、ダークシリコンへは到達しないのでしょうか？（半導体業界の進歩とアーキテクチャの進歩の関連があまりよくわかっていないです）

質問とか回答など

- FPGAは金融分野でよく使われていると聞きますが、具体的にどういう計算をするのが得意なのか気になりました。

質問とか回答など

- FPGAとASICの速度差をトランジスタ数で考えると大体の差がわかるというのが面白かった（自分はむしろ20倍程度の差で済むことに驚いた）

質問とか回答など

- また、MIPSの制御部分と浮動小数点演算器でこの程度の差しかないのなら、深層学習のためにGPUの代わりに専用回路を作るみたいな試みは結構難しいのかなと思った。

- 深層学習では推論時には結構低い精度でも行けるみたいな話もあるので、(演算精度が低く、回路制御の負担が大きい)推論専用チップがGPUよりは様々なところで作られているのはそういう理由もあるのかもしれないと考えた。

質問とか回答など

- HDLで回路を記述するとき、入力の数は自由に設定できるため必要な真理値表のサイズは固定ではないと思うのですが、書き出しのたびにLUTの回路を動的に構成しているのでしょうか。あるいは、FPGAの内部に（例えば）サイズ 4のLUTのモジュールがたくさんあり、書き出される回路が複雑な場合、複数のモジュールが協調して所望の真理値表を構成するのでしょうか。

質問とか回答など

- 消費電力がマシンを設計する上で、重要な考慮点であることはわかるが、具体的にはどのような値の消費電力に抑えるのか、またスパコンではどのぐらいの消費電力になるように設計するのかが気になった。

質問とか回答など

- FPGAの回路のオーバーヘッドについて、基本となる回路素子(AND/OR等)についても組み込みのものを配線することで利用すればいいと考えてしまったが、むしろマルチプレクサを必要としてしまうので、なかなか単純にはいかないものだと考え直した。

質問とか回答など

- Snapdragon 888 SoCのより多く発熱の原因は製造プロセスのせいで、リーク電流が多く発生すると言われていて、授業資料の話とやや異なるけど、実際そうでしょうか？

質問とか回答など

- AVX512の消費電力が多いのは、主の原因は該当する部分の回路の規模が大きいですか？