# マイクロアーキテクチャ攻撃 1

九州大学　大学院システム情報科学研究院

谷本 輝夫

# 自己紹介

- 谷本 輝夫（３５歳）
- 岐阜県出身
- 略歴
  - 東京大学（学部・修士）
  - 株式会社富士通研究所（研究員、３ヶ年）
  - 九州大学（博士後期→助教→准教授）
- 研究テーマ
  - アウトオブオーダプロセッサ性能解析
  - 汎用・専用ハイブリッドプロセッサ設計
  - セキュアプロセッサ設計
  - 量子コンピュータのシステムアーキテクチャ設計

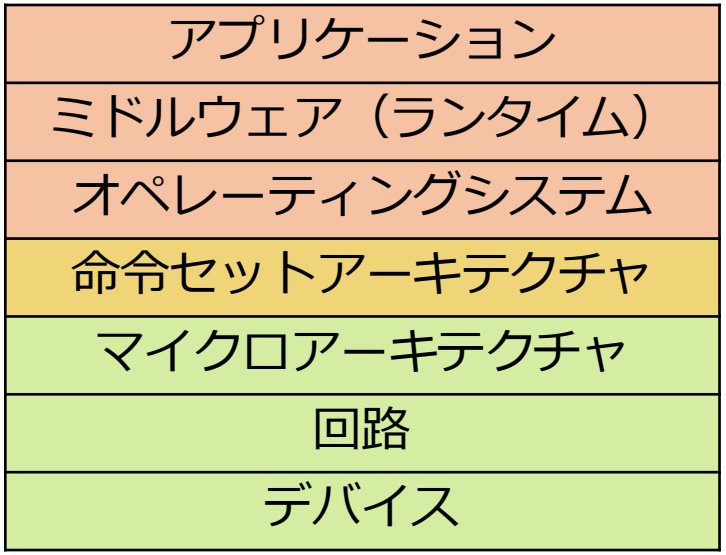
# マイクロアーキテクチャ攻撃 1

- マイクロアーキテクチャ攻撃とは
- CPU に対する攻撃
  - Meltdown 攻撃
  - Spectre 攻撃
- 実習（準備）
  - プロセッサシミュレータを使った Spectre の解析

# マイクロアーキテクチャ攻撃とは

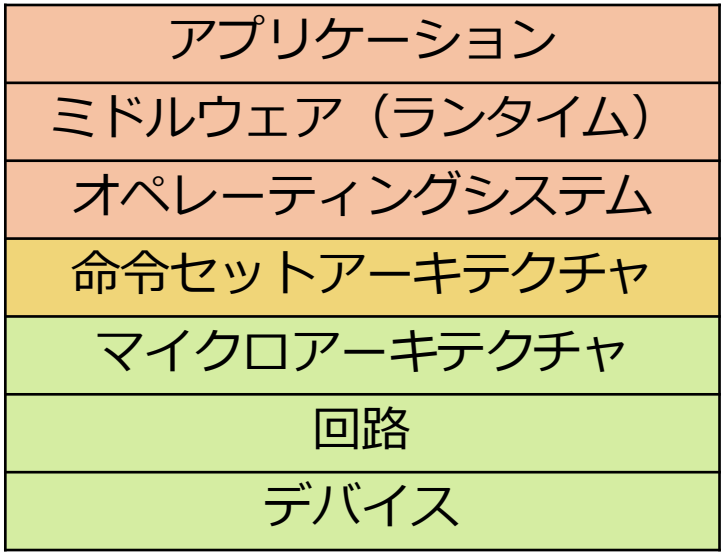
# マイクロアーキテクチャとは

- コンピュータ・システムの抽象化レイヤの１つ
  - 命令セットアーキテクチャ（ISA）の実装におけるアーキテクチャを指す
  - ソフトウェア／ハードウェアの境界に位置



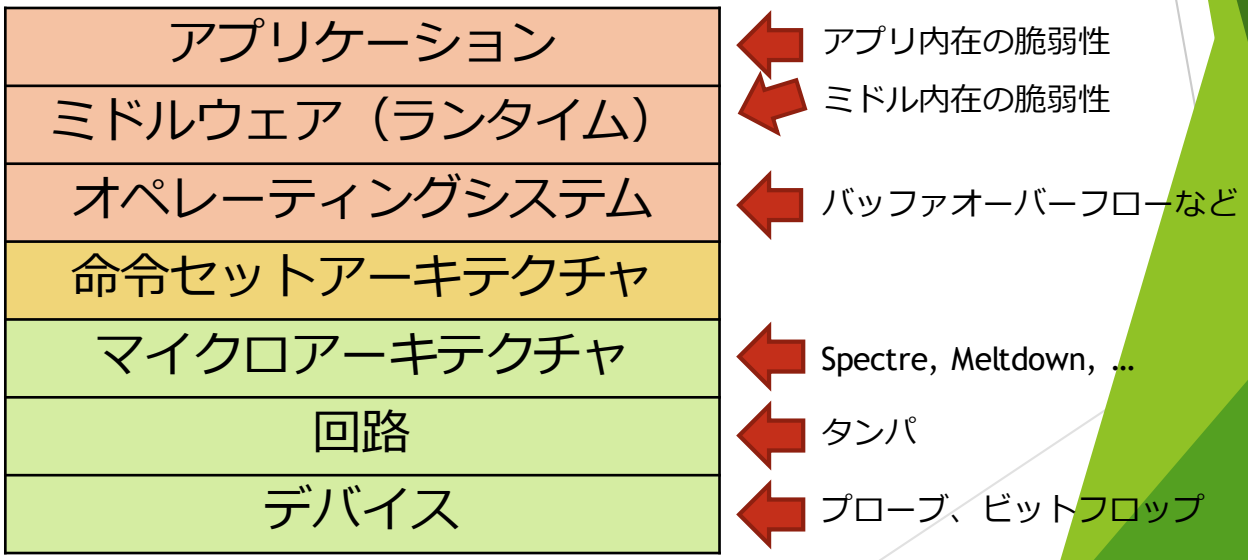
# 抽象化レイヤとセキュリティ

- セキュアなコンピュータシステムのためにはすべてのレイヤでセキュリティを考慮する必要がある
  - 抽象化されているため、別のレイヤが攻撃されていても分からない
  - 最も攻撃しやすいレイヤが狙われる



# 抽象化レイヤとセキュリティ

- セキュアなコンピュータシステムのためにはすべてのレイヤでセキュリティを考慮する必要がある
  - 抽象化されているため、別のレイヤが攻撃されていても分からない
  - 最も攻撃しやすいレイヤが狙われる



# マイクロアーキテクチャ攻撃とは

- マイクロアーキテクチャの実装に起因する脆弱性に対する攻撃
- ソフトウェアベース・マイクロアーキテクチャ攻撃
  - ハードウェア・ソフトウェア間で本来規定されている機能（入力→処理→出力）以外の副作用を起こすことによる攻撃が多数報告されている
  - ソフトウェアから見えるコンピュータシステムの状態（アーキテクチャ・ステート）とハードウェアの内部状態（マイクロアーキテクチャ・ステート）の違いに着目し、サイドチャネル攻撃により本来取得できない情報を取得・推測する手法
    - Meltdown [M. Lipp+ 2018]
    - Spectre [P. Kocher+ 2018]

    } CPUに対する攻撃
    - …
  - 命令実行によりアーキテクチャ・ステートを書き換える手法
    - Rowhammer [Y. Kim+ 2014]



[M. Lipp+ 2018] Meltdown: Reading Kernel Memory from User Space
[P. Kocher+ 2018] Spectre Attacks: Exploiting Speculative Execution
[Y. Kim+ 2014] Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors

# マイクロアーキテクチャ攻撃とは

- マイクロアーキテクチャの実装に起因する脆弱性に対する攻撃
- ソフトウェアベース・マイクロアーキテクチャ攻撃
  - ハードウェア・ソフトウェア間で本来規定されている機能（入力→処理→出力）以外の副作用を起こすことによる攻撃が多数報告されている

今回紹介

  - ソフトウェアから見えるコンピュータシステムの状態（アーキテクチャ・ステート）とハードウェアの内部状態（マイクロアーキテクチャ・ステート）の違いに着目し、サイドチャネル攻撃により本来取得できない情報を取得・推測する手法
    - Meltdown [M. Lipp+ 2018]
    - Spectre [P. Kocher+ 2018]
    - …

    } CPUに対する攻撃

  - 命令実行によりアーキテクチャ・ステートを書き換える手法
    - Rowhammer [Y. Kim+ 2014]



[M. Lipp+ 2018] Meltdown: Reading Kernel Memory from User Space
[P. Kocher+ 2018] Spectre Attacks: Exploiting Speculative Execution
[Y. Kim+ 2014] Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors

# マイクロアーキテクチャ攻撃とは

- マイクロアーキテクチャの実装に起因する脆弱性に対する攻撃
- ソフトウェアベース・マイクロアーキテクチャ攻撃
  - ハードウェア・ソフトウェア間で本来規定されている機能（入力→処理→出力）以外の副作用を起こすことによる攻撃が多数報告されている
  - ソフトウェアから見えるコンピュータシステムの状態（アーキテクチャ・ステート）とハードウェアの内部状態（マイクロアーキテクチャ・ステート）の違いに着目し、サイドチャネル攻撃により本来取得できない情報を取得・推測する手法
    - Meltdown [M. Lipp+ 2018]
    - Spectre [P. Kocher+ 2018]

    } CPUに対する攻撃
    - …

次回紹介

- 命令実行によりアーキテクチャ・ステートを書き換える手法
  - Rowhammer [Y. Kim+ 2014]



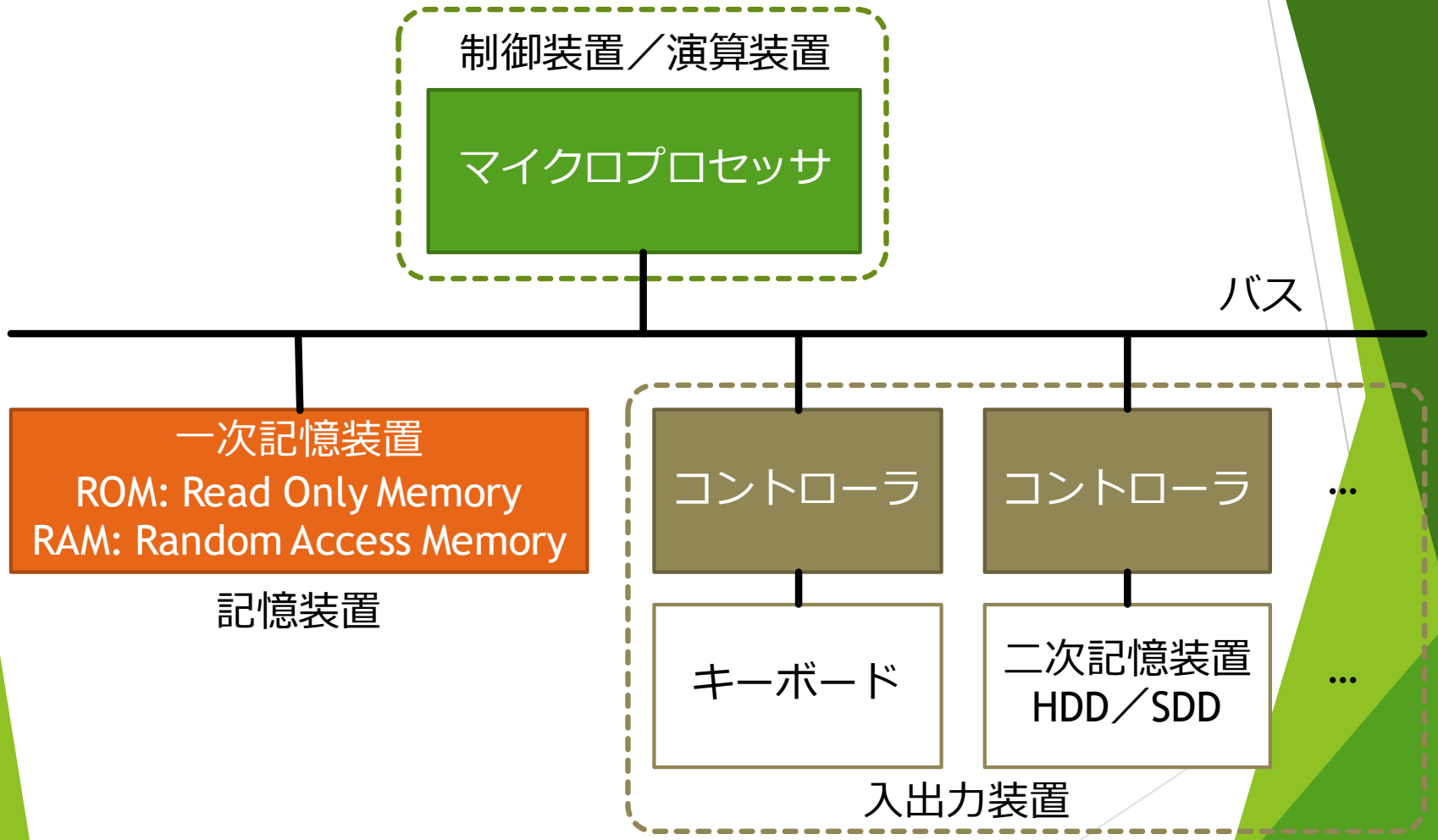
[M. Lipp+ 2018] Meltdown: Reading Kernel Memory from User Space
[P. Kocher+ 2018] Spectre Attacks: Exploiting Speculative Execution
[Y. Kim+ 2014] Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors

# CPUに対する攻撃

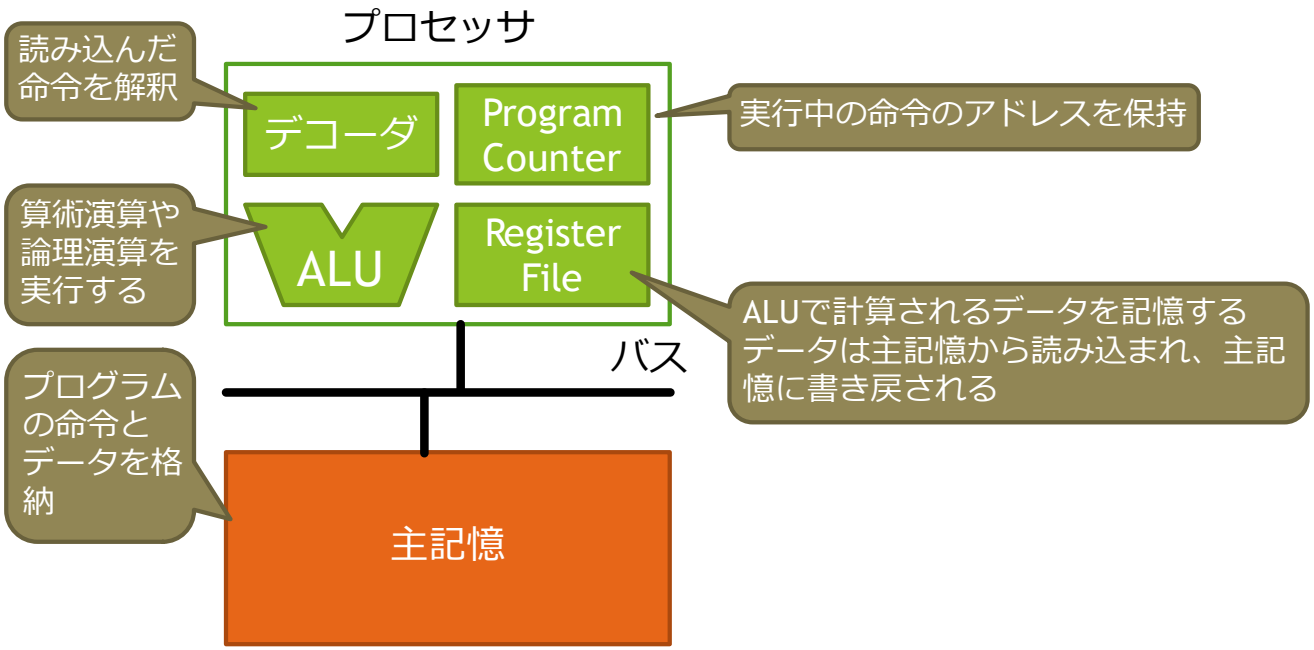
# 理解するために必要な知識

- コンピュータの構成
- プロセッサでの命令実行
- キャッシュメモリ
- アウトオブオーダ命令実行
- 分岐予測による投機実行
- アークテクチャステート
- マイクロアーキテクチャステート
- サイドチャネル攻撃

# コンピュータの構成


制御装置／演算装置
マイクロプロセッサ
バス
一次記憶装置
ROM: Read Only Memory
RAM: Random Access Memory
記憶装置
コントローラ
コントローラ
…
キーボード
二次記憶装置
HDD／SDD
…
入出力装置

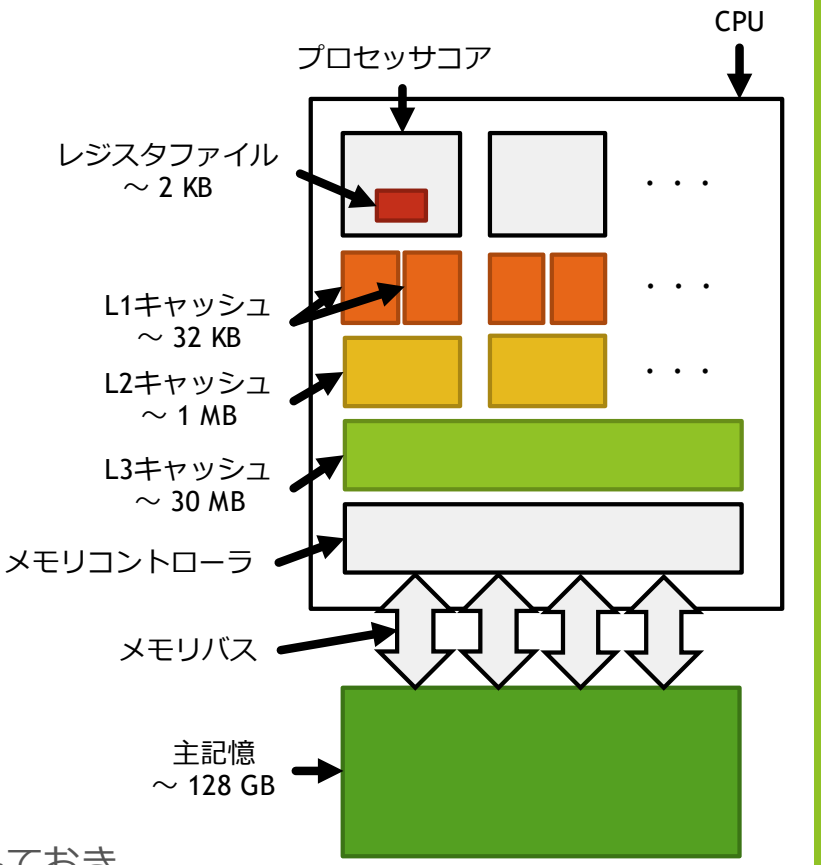
# プロセッサでの命令実行




- **ロード／ストア・アーキテクチャ**（基本的な考え方）
  - 主記憶へのアクセスは、ロード命令（読出し）とストア命令（書込み）だけが可能
  - すべての演算はレジスタに格納されているデータ（⇒オペランド）に対して行う

# キャッシュメモリ

- メモリ階層の構成要素
  - 小容量ほど高速アクセス
- ソフトウェアとメモリ階層
  - ソフトウェアから**見える**メモリ
    - レジスタ：レジスタ番号
    - 主記憶：アドレス
    - どちらも「論理（仮想）→物理」のマッピングがある
  - ソフトウェアから**見えない**メモリ
    - キャッシュ
    - 見えないことを「透過的」という
    - メモリアクセス時にキャッシュに保存しておき、次回アクセス時に使用



# アウトオブオーダ命令実行

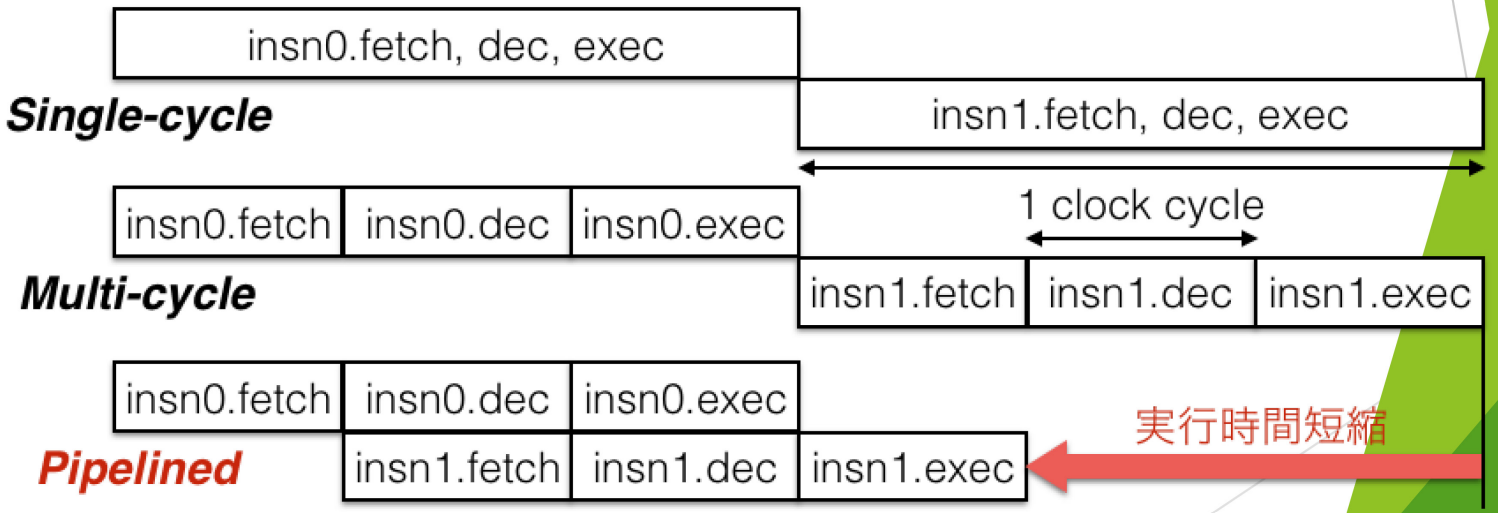
- プロセッサが命令を実行する方式の１つ
  - サーバ、PC、スマートフォンなどのCPUはほぼすべてこの方式を採用
- プログラムの命令順に関係なく命令を実行する方式
  - ⇔インオーダ命令実行（命令順通りに実行する方式）
- アウトオブオーダプロセッサ
  - プロセッサ内で命令間の依存関係を解析し、計算に必要なデータがそろった命令から先にどんどん実行していく
  - 利点：クロックサイクル当たりの命令実行数が高い
  - 欠点：回路が複雑化、消費電力の増加

# アウトオブオーダプロセッサ

- プロセッサ高性能化技術の結晶
  - パイプライン化
  - スーパスカラ
  - アウトオブオーダ命令実行

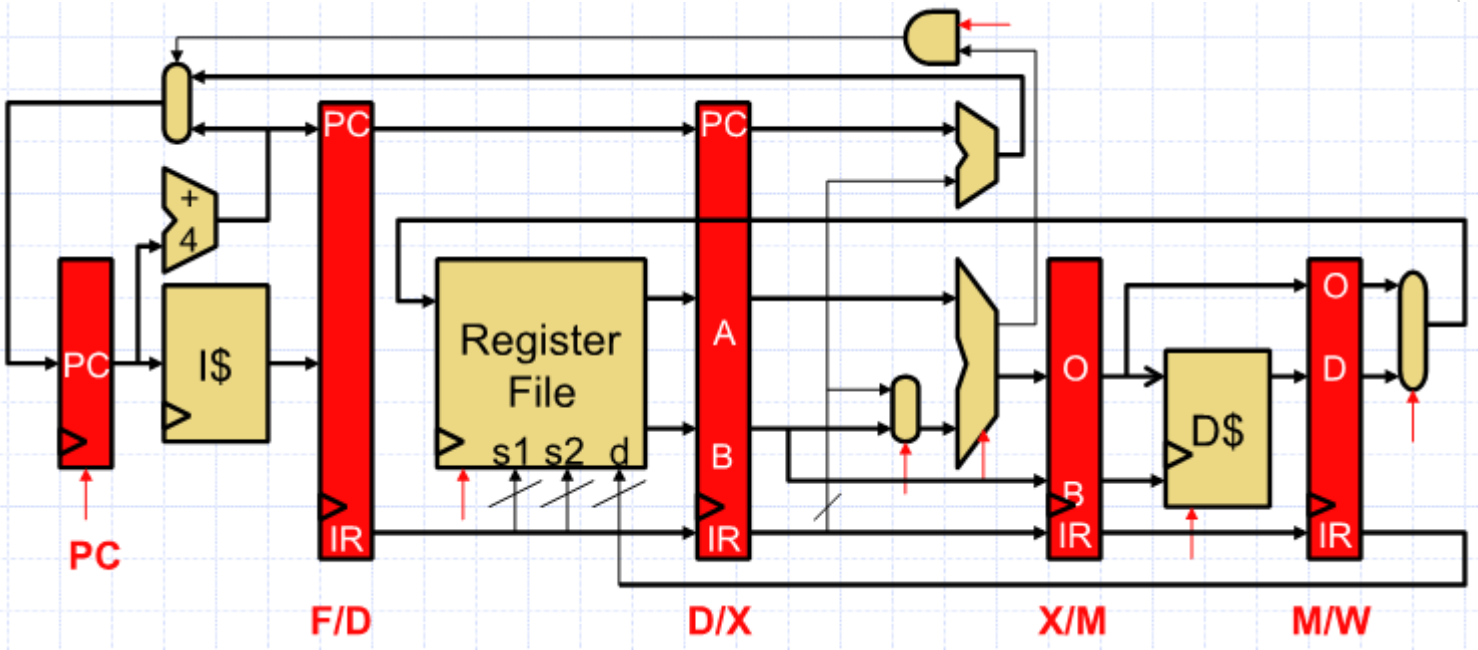
# プロセッサのパイプライン化

- プロセッサのパイプライン化
  - プロセッサの基本的な高速化手法
  - プロセッサをN段のパイプラインで構成
    →（理想的には）周波数N倍→スループットN倍
- パイプライン化の例



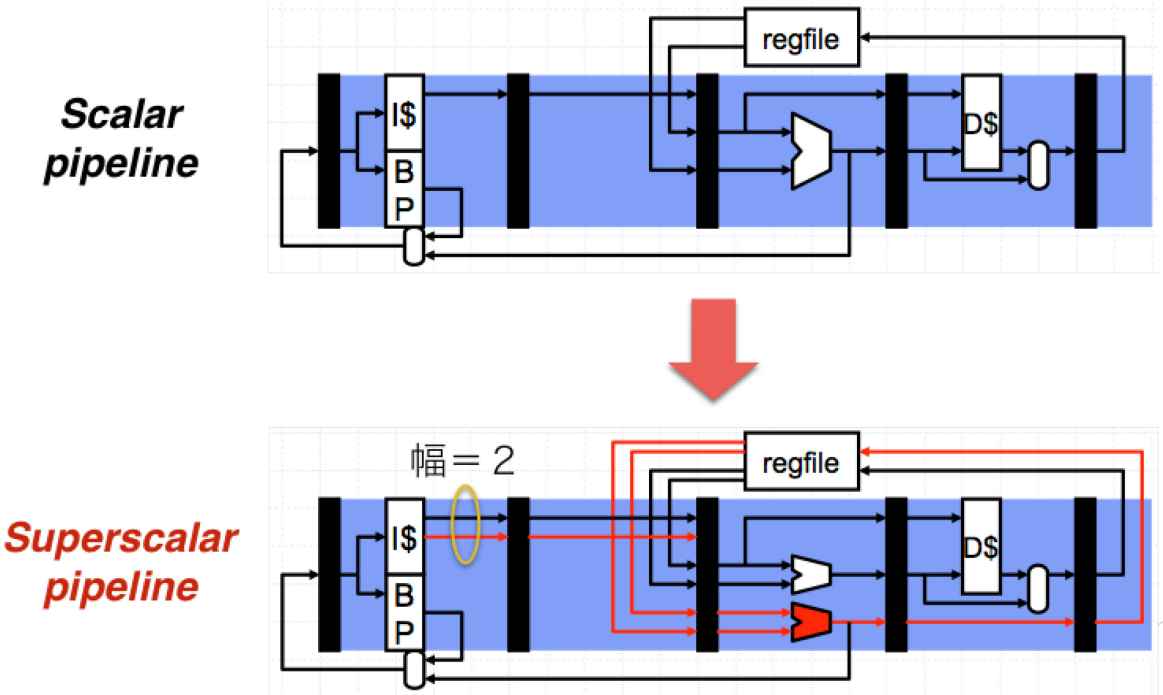
# （参考）
# 5段パイプラインプロセッサ

- ブロック図（インオーダプロセッサ）
  - 5段に分割→Fetch, Decode, eXecution, Memory, Writeback

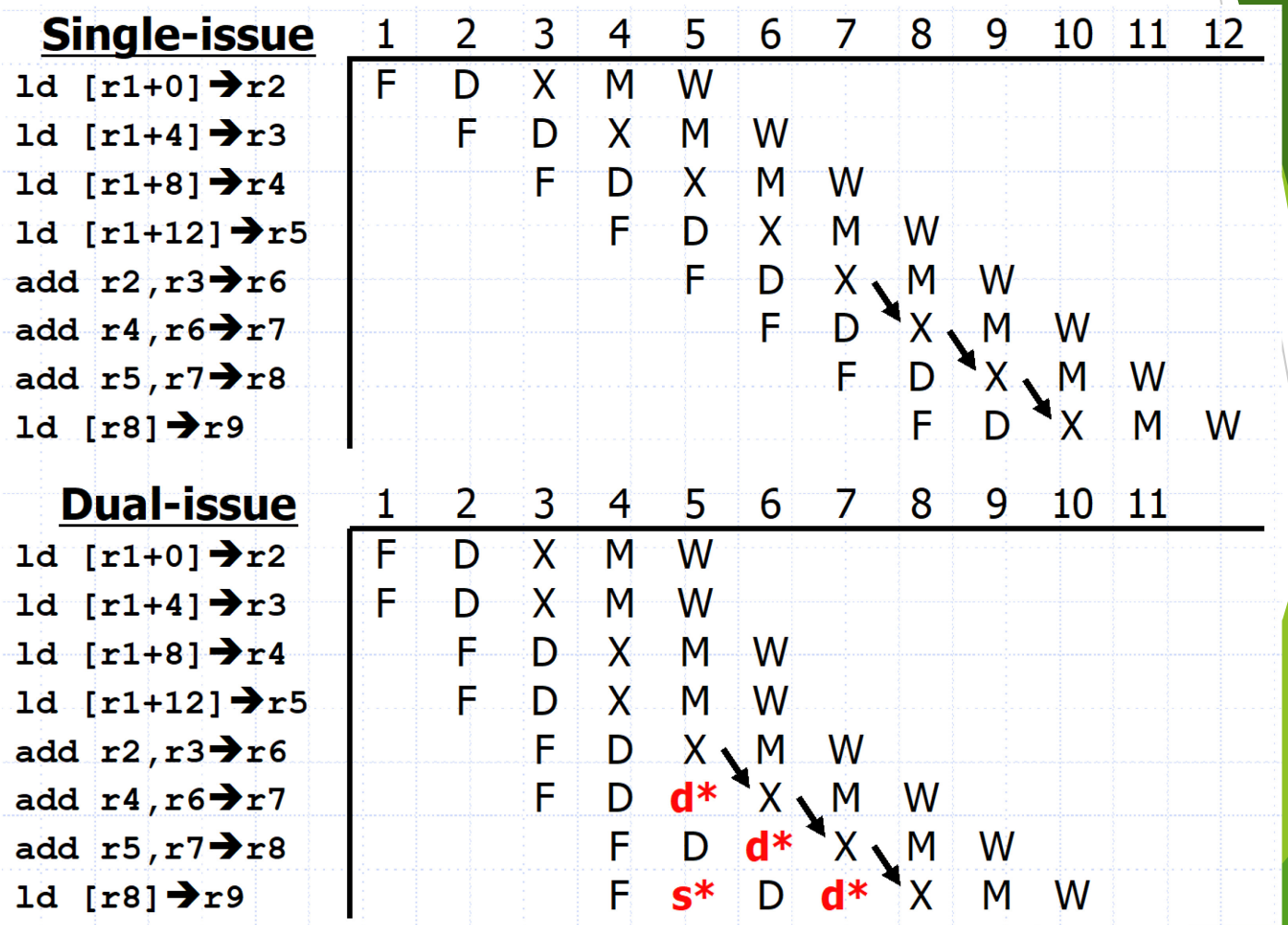


# スーパスカラプロセッサ

- パイプラインを複数持つプロセッサ
  - １つのパイプライン→IPC (Instruction per Cycle)の最大値１
  - パイプラインの「幅」を増やすことでIPC>1を実現



# スーパスカラにおける命令実行



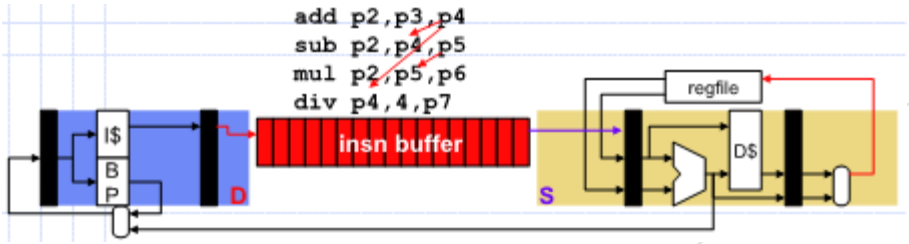
# スーパスカラプロセッサの限界

- ここまでで説明した命令実行方式
  - パイプライン化
  - スーパスカラ
- これらはプログラムの命令順通りに命令を実行
- プロセッサ資源を増やしても利用効率が頭打ち
  - データハザード（前の命令を実行しないと次が実行不能）
  - 構造ハザード（プロセッサの資源における競合）

- 実行順を入れ替えながら実行することでプロセッサ資源の利用効率向上を図るアウトオブオーダ方式が登場

# アウトオブオーダ（OoO）プロセッサ

- OoO 命令実行
  - 実行結果が変わらない範囲で命令順を入れ替えながら実行
    1. 命令を複数読み込み、その中から実行可能な命令を選択
    2. 実行可能な命令から順次実行
  - ハードウェアで動的スケジューリング
- OoO 実行の目的＝高性能
  - メモリアクセス中に他の実行可能命令を実行→遅延隠蔽
  - パイプラインストール（命令が次ステージへ進まないこと）削減
  - 複数の演算ユニット（整数、浮動小数点）の利用効率向上



# パイプラインプロセッサと制御ハザード

- 分岐命令は何をしているか？
  - プログラムカウンタの値を次の命令でない値にする
  - 条件分岐・・・条件判定に基づいて分岐するかどうか決定
  - 無条件分岐・・必ず分岐する
- プロセッサ上での条件分岐命令
  - 実行するまで次にどの命令を読みだせばよいかわからない
  - パイプラインを埋められない（制御ハザード）
    →性能が上がらない
  - パイプライン段数が深くなるほど、スーパスカラの幅が大きくなるほどインパクト大

# 分岐予測による投機実行

- 分岐命令の実行よりも早いステージで分岐するか否かを予測し、後続命令のフェッチを開始する手法
- 予測が外れた場合はプロセッサの状態を戻して正しい命令を実行し直す
- 分岐予測が当たればパイプラインを埋めることができて性能向上可能

# 分岐予測手法

- 静的予測
  - 常に分岐しないものと予測する
- 動的予測（履歴情報を利用）
  - 飽和カウンタ
    - 2bit のカウンタを使う方式で、ネストしたループ構造を持つプログラムに適合する
  - ２レベル適応型
    - 直近n回の分岐結果を分岐予測に利用する方式で、規則的な分岐に適合する
  - 局所分岐予測
    - 個々の条件分岐命令の履歴を用いて予測する方法
  - 広域分岐予測（gshare, gselectなど）
    - 個々の条件分岐履歴を使わず、すべての条件分岐命令で共有

# アーキテクチャステート

- プロセスの状態を保持している情報
  - プログラムカウンタ
  - 制御レジスタ
  - アーキテクチャレジスタ
    - 汎用レジスタ
    - 浮動小数点レジスタ

  CPU内部の情報

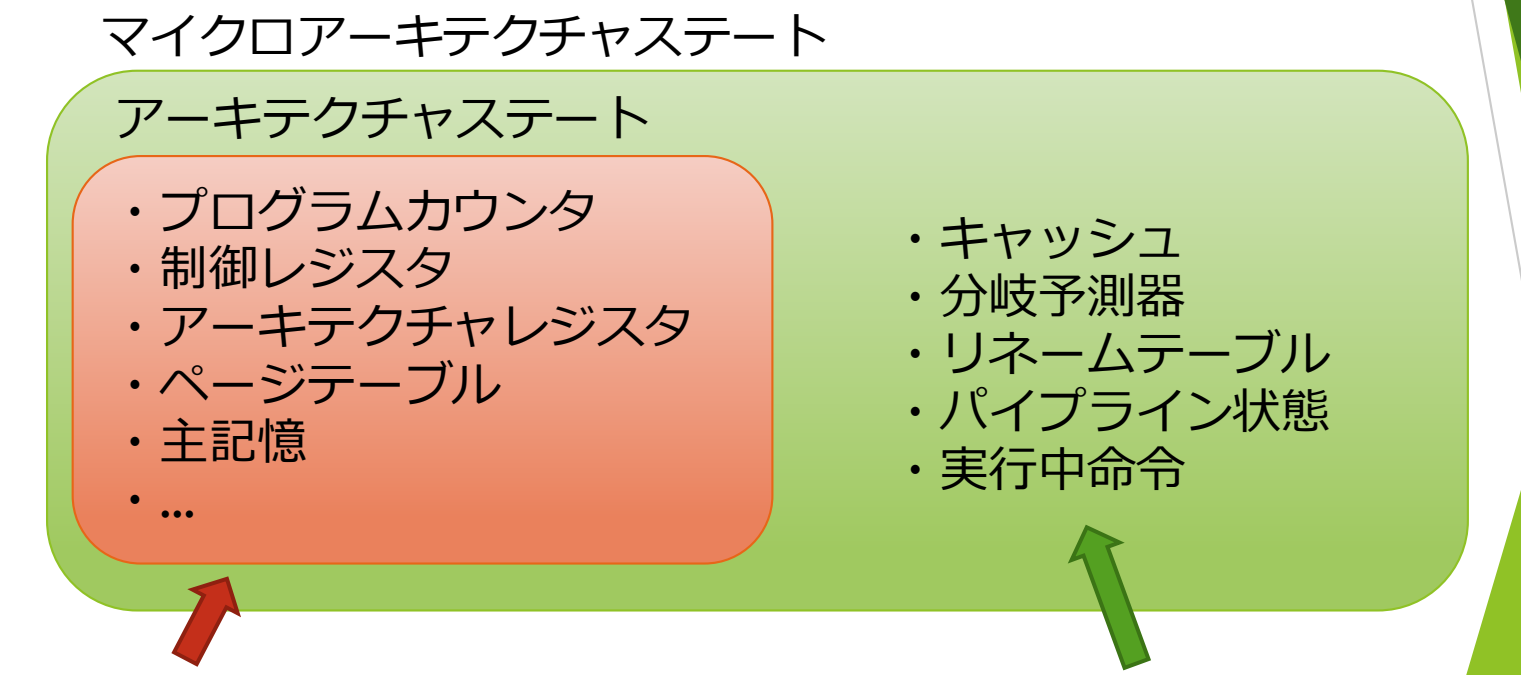
  - ページテーブル（仮想アドレス→物理アドレス）
  - 主記憶

- これらはソフトウェアが管理可能

# マイクロアーキテクチャステート

- 実際にはプロセッサの中の状態はより複雑
  - どの命令がどのパイプラインステージにいるか
  - 実行中の命令の経過
  - キャッシュにどのアドレスのデータが存在するか
  - 分岐予測器のテーブルにどんな値が入っているか
  - レジスタリネームテーブルの値がどうなっているか
    - 物理レジスタとアーキテクチャレジスタの対応表
  - マルチコアプロセッサの場合、キャッシュラインの状態がどうなっているか
  - …

# アーキテクチャステートとマイクロアーキテクチャステートの関係



# サイドチャネル攻撃

- 装置の動作状況を様々な物理的手段で観察することにより、装置内部のセンシティブな情報を取得しようとする攻撃方法の総称

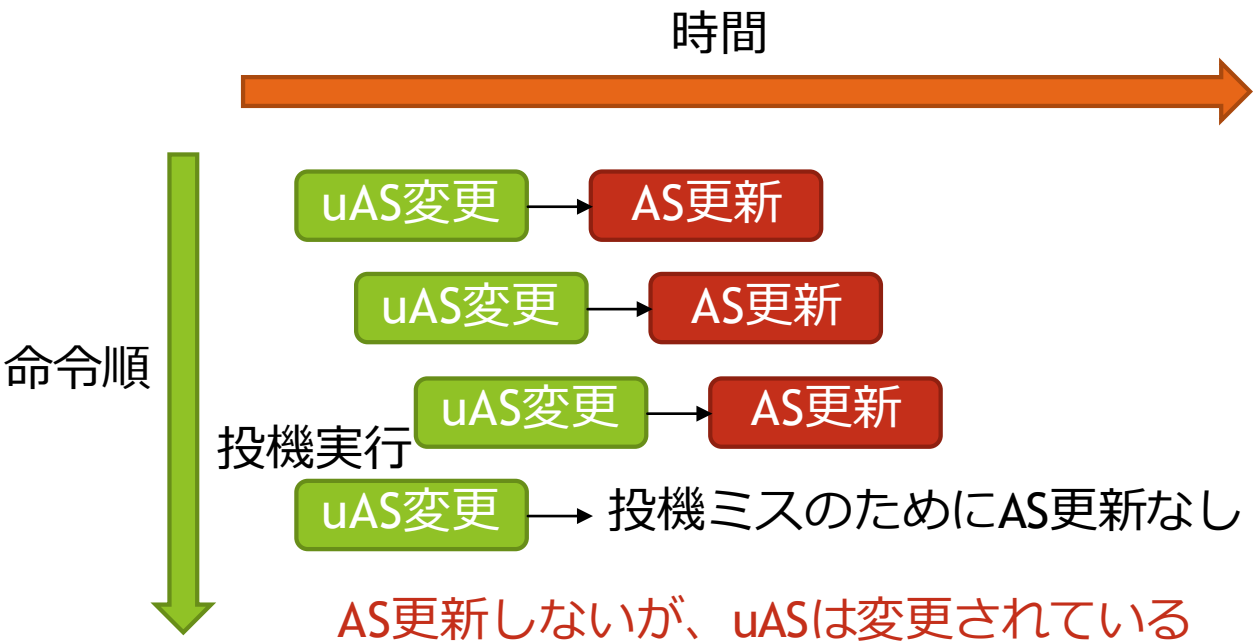
- 分類
  - タイミング攻撃
    - 処理時間の違いに着目
  - 電力解析攻撃
    - 消費電力の違いに着目
  - 電磁波解析攻撃
    - 装置から漏洩する電磁波に着目

# サイドチャネル攻撃

- 装置の動作状況を様々な物理的手段で観察することにより、装置内部のセンシティブな情報を取得しようとする攻撃方法の総称

- 分類
  - タイミング攻撃 → CPUはかなり高精度に取得可能（サイクル精度）
    - 処理時間の違いに着目
  - 電力解析攻撃 → CPUは最近取れるようになってきた
    - 消費電力の違いに着目
  - 電磁波解析攻撃 → 特殊環境が必要
    - 装置から漏洩する電磁波に着目

# 投機実行を使った攻撃




AS更新しないが、uASは変更されている

- 本来アクセスできない攻撃対象の情報をuASに残す
- uASの変更をサイドチャネル攻撃で検出

# 紹介する具体的な攻撃

- Meltdown, Spectre
  - 様々な亜種が報告されているが、原著論文に沿った手法について紹介
  - 攻撃の基本が理解できれば、亜種についても理解しやすくなる

# Meltdownについて

- CVE-2017-5754
- 概要
  - 権限を持たないユーザが PC のカーネルメモリと物理メモリに完全にアクセスすることが可能になる脆弱性
- 攻撃方法
  - 対象 PC 上で攻撃プログラムを実行
  - クラウドサービスのような共有システムへの攻撃を考えると、他のユーザの情報にアクセス可能
- 影響範囲
  - Intel CPU を採用したシステム *1
  - ARM Cortex-A57 を採用したシステム *2
  - AMD CPU では亜種による攻撃可能性が実証された

*1 https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00088.html
*2 https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability

# Meltdown攻撃の原理１：概要

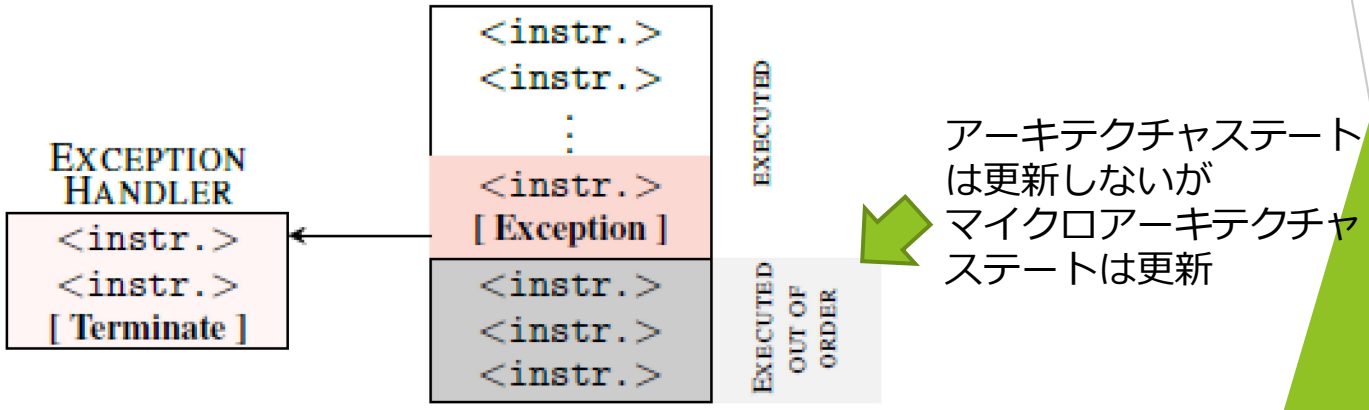
1. アクセスが禁止されている領域をアクセスするような処理を書いておく（→実行されると例外を起こす）
2. 例外が発生するであろう処理の後に、アクセス履歴をCPUのキャッシュ内に残すような処理を書いておく
3. アクセスが禁止されている領域にアクセスすると、例外が発生して処理が中断される
4. アウトオブオーダー実行により、例外発生前に2.の処理が実行される
5. 4.のCPUのキャッシュを読み取ることにより、物理メモリのどの場所にアクセスが禁止されている情報があるかを判断して、参照することが可能となる

# Meltdown攻撃の原理 2

- こんなプログラムを考えてみる

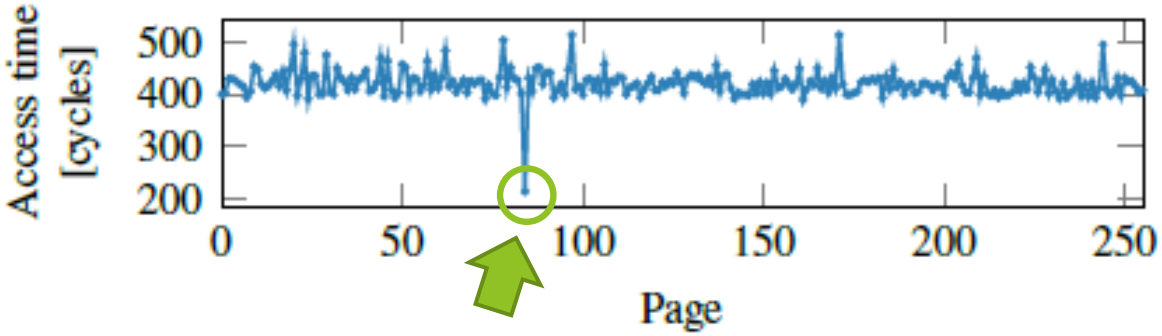
```
1 raise_exception();
2 // the line below is never reached
3 access(probe_array[data * 4096]);
```

- アウトオブオーダ実行では例外後の処理も一部実行



# Meltdown攻撃の原理３

- probe_arrayへのアクセス時間を測定すると・・・




明らかにアクセスが速いページが存在

- probe_array[data*4096] にアクセスしていたな・・・
  - data = アクセスの速いページ番号
    （probe_array は１要素１Ｂ、ページサイズ４ＫＢを想定）
  - マイクロアーキテクチャステートへの副作用からdataの値推定

# なぜページ？

- わかる方いますか？

# なぜページ？

- ちょっとマニアックな話になりますが
- 後でアクセス時間を測るときにhardware prefetchが効かないようにするため
  - 近いアドレスを連続してアクセスしていくとプリフェッチ機能でメモリからキャッシュにデータが運ばれる
  - 結果、アクセス時間が隠蔽される
  - Hardware prefectherは基本的には物理メモリアドレスしか知らないので、ページをまたいだプリフェッチはしない
  - なので、ページ単位でアクセスしていけばアクセス時間がプリフェッチの影響を受けない

# Meltdown攻撃の原理４

- 実際の攻撃コードを見てみましょう

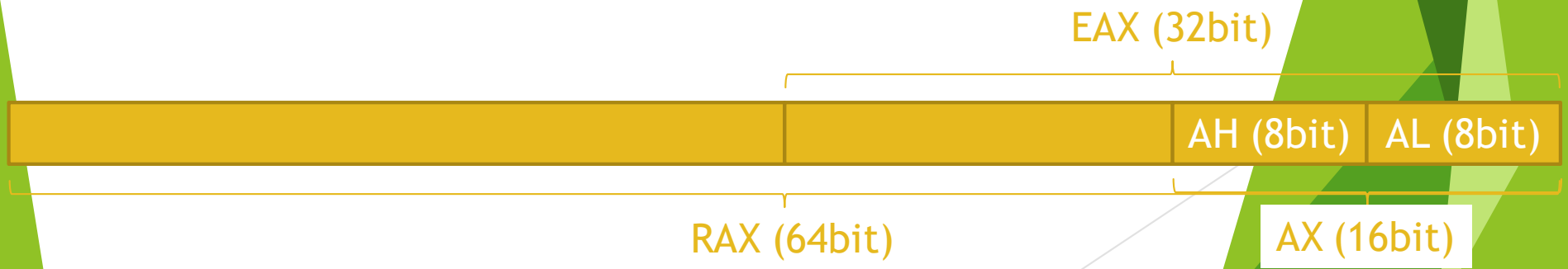
```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

# Meltdown攻撃の原理 5

- 実際の攻撃コードを見てみましょう

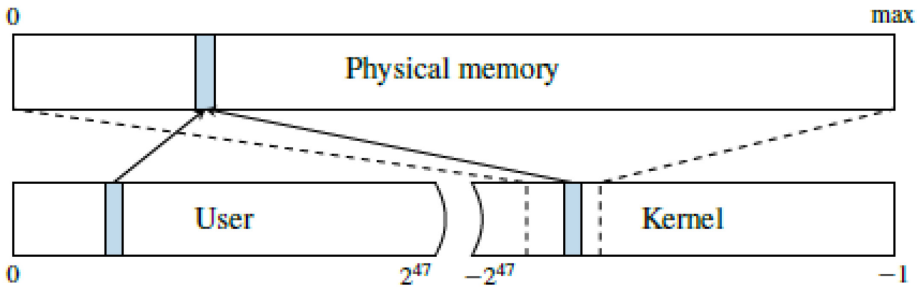
```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax                    ← rax = 0
3 retry:
4 mov al, byte [rcx]              ← rax に呼び出し→例外発生
5 shl rax, 0xc                    ← rax = rax * 4096
6 jz retry                        ← 攻撃に失敗した場合は繰り返す
7 mov rbx, qword [rbx + rax]      ← rbx+rax のデータをキャッシュに載せる
```

- X86汎用レジスタ（A～D）



# Meltdown攻撃の原理 6

- Kernel address ?
  - LinuxとMacOSでは、Meltdownの対策が本格化する前まで、プロセスのアドレス空間にUser空間もkernel空間もマッピングして単一のページテーブルで管理するのが一般だった
  - カーネル空間にはシステムの全メモリをダイレクトマップ




  - そのため、（例外は起こるが）ユーザプロセスからマシンの任意の物理メモリアドレスに対してアクセス可能

# Meltdown対策

- OSでの対策が可能
  - Linuxでは4.15でマージ、4.14.11、4.9.75、4.4.110でも利用可能
  - Windows, MacOS でも対策済み
- 方法：User-modeとKernel-modeでアドレス空間を分離すればそもそもアクセスできない
  - 理由：アドレス変換できないから
- Kernel page-table isolation (KPTI, PTI, KAISER)
  - 実はMeltdownが見つかるよりも前に実装されていた
  - Kernel Address Space Layout Randomization (KASLR) が持つ脆弱性対策

# Spectreについて

- CVE-2017-5753, CVE-2017-5715
- 概要
  - 投機的実行に関する欠陥に基づく
- 攻撃方法
  - 攻撃者と対象が同じコードを共有
  - ２つの亜種
    - 単一プロセス空間への攻撃
    - スーパーユーザのアクセス権を使った攻撃
- 影響範囲
  - Intel, AMD, ARM のプロセッサ
  - Meltdown よりも範囲大

# Spectre攻撃の原理１：概要

1. あるプロセスの投機的実行で、読まれてはいけない情報をCPUのキャッシュメモリに読み込むように仕向ける
2. 別プロセスでキャッシュの情報を参照することで、読まれてはいけない情報を読むことが可能となる

# Spectre攻撃の原理２

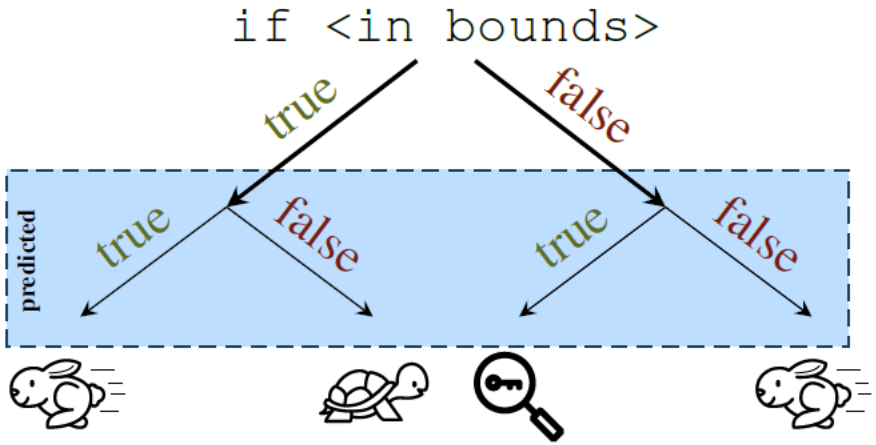
- 攻撃１：分岐予測ミスを利用した攻撃
  - 以下のコードを考えてみましょう

```
if (x < array1_size)
    y = array2[array1[x] * 4096];
```

    - array1の各要素は１Ｂを想定
  - まずは分岐がTakenに予測されるよう分岐予測器を学習させた状態を想定
  - その後、xの値を書き換えるとどうなるか？
  - 投機的にアクセスするアドレス値
    - array2_base + [array1_base + x] × 4096
  - どのアドレスのデータがキャッシュに載っているか調べることで[array1_base + x]の値が分かる

# Spectre攻撃の原理 3

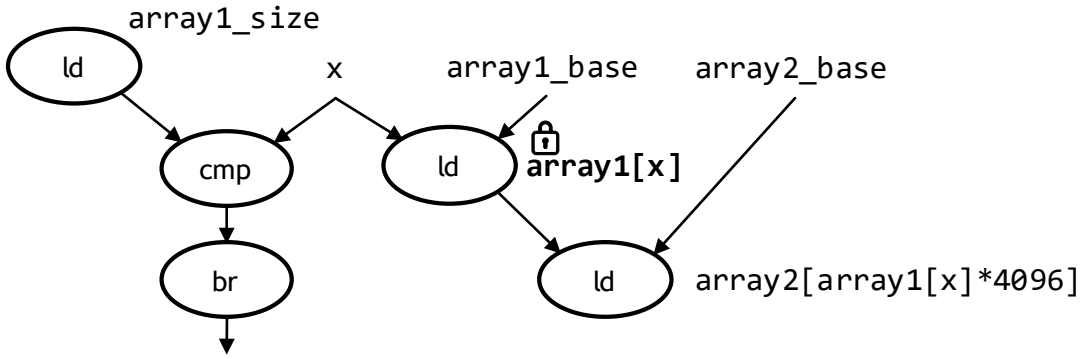
- 分岐予測と攻撃の関係




  - 分岐予測が当たった時に速くなるだけならよかったが、予測ミス時に特定の場合にキャッシュにデータ値に対応した痕跡が残る

# Spectre攻撃の原理４

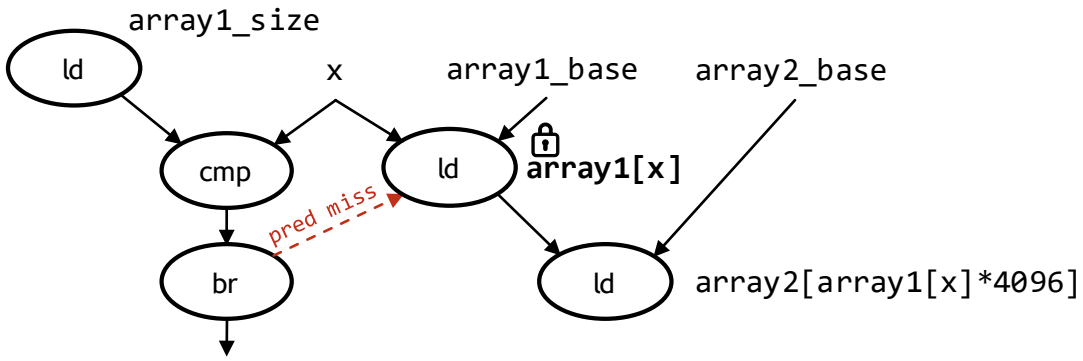
- 攻撃１成立の条件
  - xの値をarray1[x]が対象プログラムの秘密情報アドレスに対応するように設定可能なこと
  - array1_sizeとarray2がキャッシュされておらず、秘密情報がキャッシュされていること




  - xに関する条件分岐がtakenに予測なるよう分岐予測器を学習させていること

# Spectre攻撃の原理４

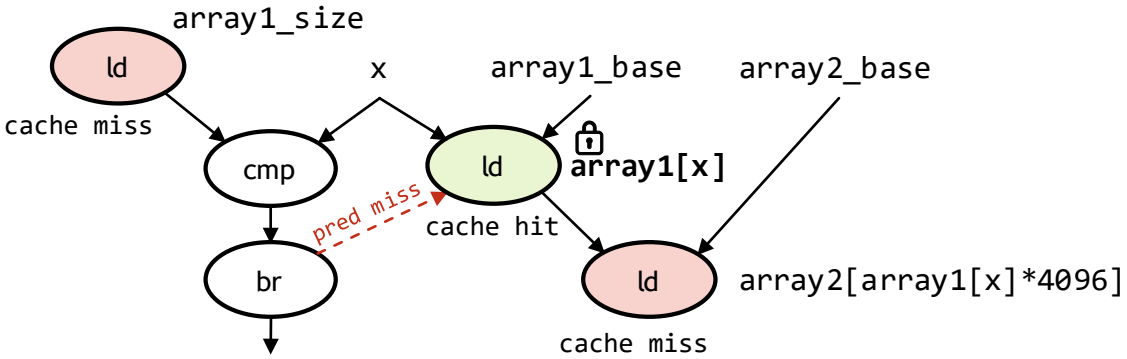
- 攻撃１成立の条件
  - xの値をarray1[x]が対象プログラムの秘密情報アドレスに対応するように設定可能なこと
  - array1_sizeとarray2がキャッシュされておらず、秘密情報がキャッシュされていること




  - xに関する条件分岐がtakenに予測なるよう分岐予測器を学習させていること

# Spectre攻撃の原理４

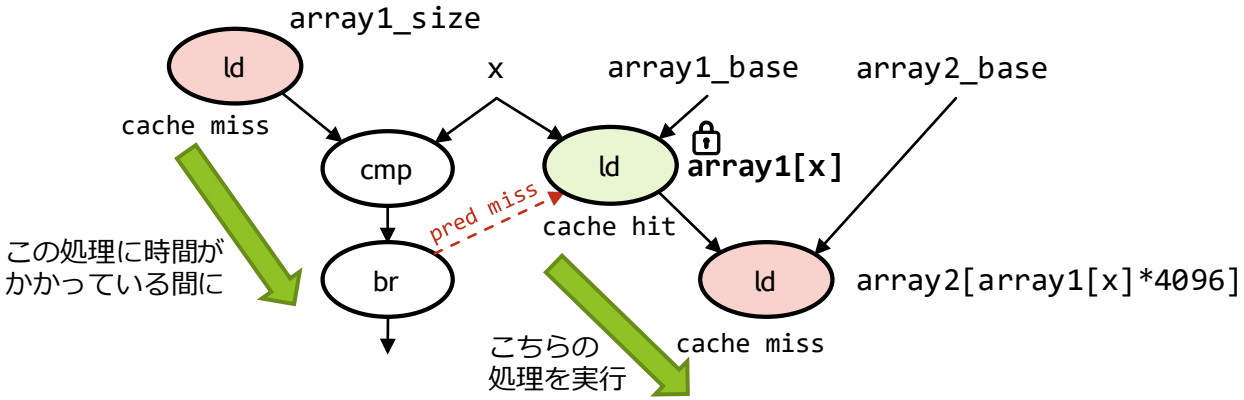
- 攻撃１成立の条件
  - xの値をarray1[x]が対象プログラムの秘密情報アドレスに対応するように設定可能なこと
  - array1_sizeとarray2がキャッシュされておらず、秘密情報がキャッシュされていること




  - xに関する条件分岐がtakenに予測なるよう分岐予測器を学習させていること

# Spectre攻撃の原理４

- 攻撃１成立の条件
  - xの値をarray1[x]が対象プログラムの秘密情報アドレスに対応するように設定可能なこと
  - array1_sizeとarray2がキャッシュされておらず、秘密情報がキャッシュされていること



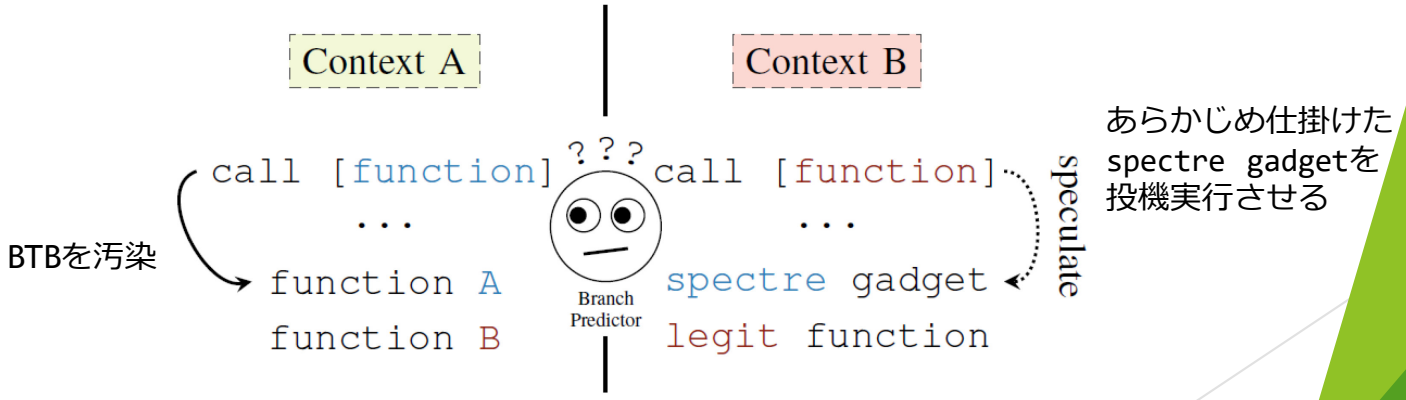

  - xに関する条件分岐がtakenに予測なるよう分岐予測器を学習させていること

# Spectre攻撃の原理 5

- 攻撃 1 のポイント
  - xの値さえ変更できれば、残りのデータアクセスするプログラム自体は既存のものを利用できる
  - 利用できる条件分岐の例
    - アクセス範囲チェック（既出）
    - 前にアクセスしたデータのチェック
    - 前にアクセスしたオブジェクトの型チェック
  - 利用できる投機実行処理の例
    - チェック結果をメモリに書く処理
    - 複雑な（たくさんの）命令を実行する処理

# Spectre攻撃の原理 6

- 攻撃 2 ：間接分岐汚染
  - 間接分岐：分岐先アドレスとしてレジスタ値を指定
    - 分岐予測→分岐するかどうかを予測
    - 分岐先予測→間接分岐の分岐先を予測
      Branch Target Buffer (BTB) に分岐先履歴を保持し予測に使用
      BTBはプロセス間で共有されている
  - 2 つのコンテキスト（プログラム）を使って攻撃



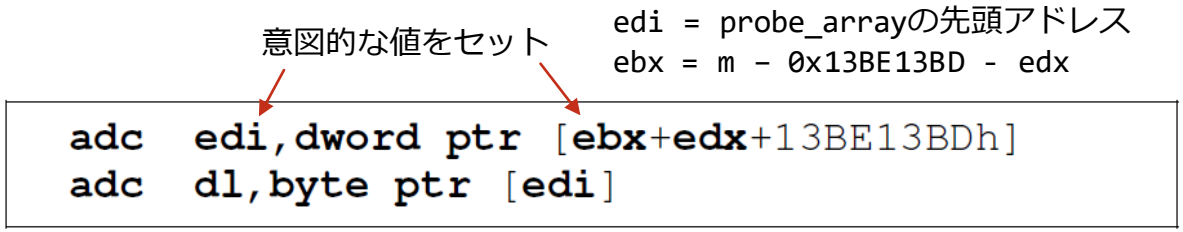
# Spectre攻撃の原理 7

- 攻撃２：spectre gadget
  - gadgetは対象プログラムから実行できないといけない
  - 数MBある共有ライブラリの一部がgadgetとして使える
    - Windows 8, 10 の共有ライブラリ (ntdll.dll) の一部

```
adc  edi,dword ptr [ebx+edx+13BE13BDh]
adc  dl,byte ptr [edi]
```

# Spectre攻撃の原理 7

- 攻撃２：spectre gadget
  - gadgetは対象プログラムから実行できないといけない
  - 数MBある共有ライブラリの一部がgadgetとして使える
    - Windows 8, 10 の共有ライブラリ (ntdll.dll) の一部



# Spectre攻撃の原理 7

- 攻撃２：spectre gadget
  - gadgetは対象プログラムから実行できないといけない
  - 数MBある共有ライブラリの一部がgadgetとして使える
    - Windows 8, 10 の共有ライブラリ (ntdll.dll) の一部

意図的な値をセット

```
edi = probe_array_base
ebx = m – 0x13BE13BD - edx
```

```
adc  edi,dword ptr [ebx+edx+13BE13BDh]
adc  dl,byte ptr [edi]
```

・アドレスmから32-bitのデータを読み出しediに加算
  edi = probe_array_base + [m]
・２命令目で[m]の値を使ったアドレスのデータをキャッシュに載せる

- probe_array を共有ライブラリ領域に取ればほかのプロセスからもキャッシュに載っているかどうか調べることが可能

# Spectre攻撃の原理 8

- 攻撃２成立の条件
  - 分岐予測器を狙った通りに学習させることが可能
    - 通常分岐予測器の細かい仕様は公開されない
      研究者たちはプロセッサの分岐予測器をリバースエンジニアリングして挙動を解析し、学習させる方法を開発
  - spectre gadgetの準備
    - 対象のプログラムか、プログラムがリンクしている共有ライブラリの中からgadgetとして使えるコード辺を見つける
    - つまり、コードが既知である必要がある
  - gadgetへの入力の作成
    - 投機実行中に知りたいデータのアドレスに対してロードが行われるようにレジスタに値をセット
    - 分岐先予測ミスを利用して作り出す

# Spectre攻撃の原理 8

- 攻撃 2 のポイント
  - BTBの値を狙った値に書き換えられてしまうと投機的にではあるが任意のコードを実行できるのに近い状態を作ることができる
  - 投機的に実行された結果を知る方法としてキャッシュに載っているデータのアドレスを使う
    - 必ずしもキャッシュを使わなくてもほかのマイクロアーキテクチャ部分を利用することも（原理的には）可能
  - 任意のコードというのは攻撃者が用意する必要はなく、すでに実行可能なコードから攻撃に使える断片を見つければよい

# Spectre攻撃の例 1

- 攻撃 1 の例：JavascriptによるWebブラウザに対する攻撃
  - Javascript：Webサイト構築に使うスクリプト言語
    - WebブラウザはWebサイトからJavascriptをダウンロードして実行できるように作られている
  - JavascriptコードからChromeプロセスのプライベートメモリの値を読み出すことができた

```
1 if (index < simpleByteArray.length) {
2   index = simpleByteArray[index | 0];
3   index = (((index * 4096)|0) & (32*1024*1024-1))|0;
4   localJunk ^= probeTable[index|0]|0;
5 }
```

  - このことはWebサイトがChromeプロセスのデータを読み出せることを意味する
  - Webブラウザにパスワードなど覚えさせていると・・・

# Spectre攻撃の例 2

- 攻撃 1 の例：eBPFを使った攻撃
  - extended Berkley Packet Filter (eBPF)
    - ユーザ空間で作成したプログラムをカーネルに送り込んで、独自の命令セットを持つカーネル内部の仮想マシン上で実行できる機能（仮想マシン＝この場合インタプリタ）
    - 様々な用途で利用できる
  - eBPFのコードはカーネル空間で実行されるため、OSのメモリが読める

# Spectre攻撃の例 2

- 攻撃 2 ：KVMに対する攻撃
  - Kernel-based Virtual Machine (KVM)
    - Linuxカーネルをハイパーバイザとして機能させるための仮想化モジュール
  - ホストOSのメモリをゲストOSから読み出すことに成功
  - eBPFを使った攻撃を利用
  - 攻撃の条件
    - 攻撃者がゲストのring 0特権レベルにアクセスできること
    - これは、ゲストOSをフルコントロールできることと同等
  - IaaS事業者へのインパクト大

# Spectre対策

- 分岐予測を利用するので分岐予測をしなければよいが
  - プロセッサの性能がかなり低下することになるので難しい
  - 現状、ソフトウェアから分岐予測をON/OFFできない
- 根本的な解決のためにはハードウェアの変更が必要
- 緩和策：x86命令セット（Intel, AMD）の場合
  - lfence命令を分岐の後に挿入することを推奨
    - lfance命令：lfence命令より後のロード命令が先に実行されることを防ぐ
  - ただし性能低下あり
  - コンパイラの工夫でgadgetとして使われそうな箇所に挿入
  - カーネル・プログラムをコンパイルし直して更新

# まとめ

- マイクロアーキテクチャ攻撃
  - マイクロアーキテクチャの実装の脆弱性を狙った攻撃
- アーキテクチャステートとマイクロアーキテクチャステートの違いを利用した攻撃
  - 悪意ある命令実行によりハードウェアの内部状態を変更し、それをサイドチャネルアタックにより検出することで本来アクセスできない情報を取得する攻撃について紹介
  - 具体的な攻撃手法
    - Meltdown
    - Spectre
- プロセッサを高速化しようとすると内部状態はより複雑に多くの情報を持つ必要がある
  - 性能とセキュリティのトレードオフ関係がある