先進計算機構成論 07

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

前回の内容

1. 分岐予測器全体の構造
2. 動的分岐方向予測

補足：分岐予測の効果

- 簡単な見積もり
  - ◇ 実行命令数： N
  - ◇ 全ての分岐予測があたる場合の実行サイクル数： N
    - □ この場合に1サイクルあたり1命令実行できると仮定
  - ◇ 分岐予測ミスの発生確率： P
  - ◇ 分岐命令の出現率： 0.2 (5命令に１回出現)
  - ◇ 予測ミスペナルティ： 20 サイクル
- N命令を実行するのにかかる実行サイクル数 C を考える
  - ◇ $C = N + N \times P \times 0.2 \times 20 = N \times (1 + P * 10)$
  - ◇ 実行時間は元の $(1 + P * 10)$ 倍になる

補足：分岐予測の効果

- 実行時間は元の $(1 + P * 10)$ 倍になる
- 場合ごとの実行時間の増加率
  - ◇ 分岐予測を全くしない場合：　　P=1 → $1 + 1 * 10 = 11$ 倍
  - ◇ ミス率が0.5の場合：　　P=0.5 → $1 + 0.5 * 10 = 6$ 倍
  - ◇ ミス率が0.05の場合：　　P=0.05 → $1 + 1 * 0.05 = 1.05$ 倍
- 実際には１サイクル３命令程度は実行できる
  - ◇ $N \times (1 + P * 10)$ではなく，$N \times (1/3 + P * 10)$
  - ◇ N/3 を基準にすると，悪化率はもっとヒドくなる

今日の内容

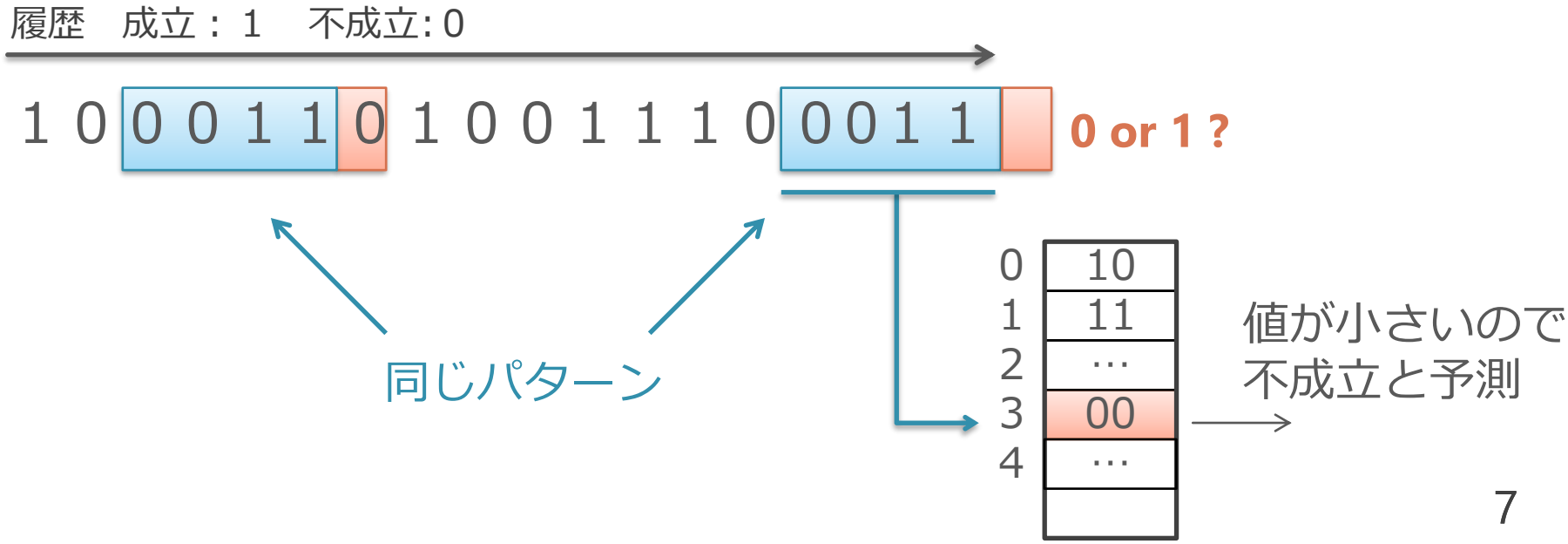
1. 高度な分岐予測
    1. 動的分岐方向予測の続き
    2. 間接分岐予測
2. メモリ

動的分岐方向予測の続き

- パーセプトロン予測器
- TAGE 予測器
  - ◇ これらのより詳細はこの講義では扱いませんが,
    高度な分岐予測器の詳細に関する付録の資料を用意しています
  - ◇ github のリポジトリの末尾の方にあります

前回の復習：「履歴（history）」を用いた予測器

- 基本的なアイデア：分岐方向の履歴をビット列で表す
  - ◇ 履歴のビット列をインデクスとしてテーブルにアクセス
    - □ テーブルのエントリは飽和型カウンタ
    - □ 成立時にインクリメント，不成立時にデクリメント
  - ◇ 直前の履歴でテーブルをひく
    - □ 直前に同じパターンがくると，同じエントリにアクセス
    - □ 二進数で 0011 = 表の3番目のエントリ



履歴長と予測精度

- 一般に，グローバル履歴長を長くするほど精度はあがる
  - より遠い分岐の相関が拾えるようになる
  - 履歴長が1000以上のところに相関がある場合もある
    - ある関数で分岐した後，色んな所にいってまた来るとか

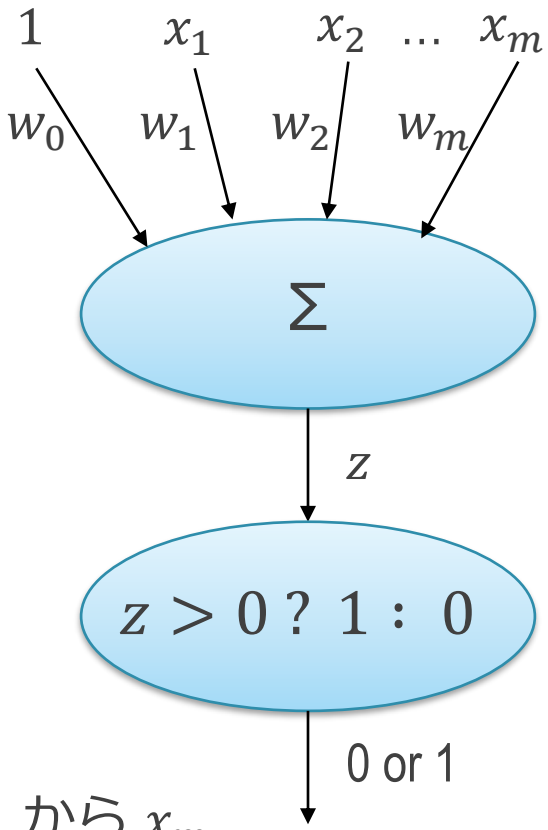
履歴長と予測精度

- 実際にはハードウェア（特に PHT の大きさ）の制約がある
  - １サイクル内にアクセス可能な大きさに限られる
    - 最大数K エントリ程度
    - （最近はもうちょっと大きいかも）
  - 履歴長に対し，２の累乗のオーダーでエントリ数が増加
    - インデクスの最大値 = 2^履歴長 – 1
    - ハッシュ関数で折りたためば解決するが，出現パターン数が増えることには違いがない

パーセプトロン予測器

- モチベーション：
  - グローバル履歴のうち，本当に相関があるのは一部のビットのみ
  - ある特定の if 文同士で相関がある場合，間の履歴は無駄
- パーセプトロン予測器：
  - １層パーセプトロンを使って予測
  - 高速に予測を行う必要があるため，ややこしいことは無理
    - １層限定で，重みは８ビット固定小数点とか
- 塩谷が学生の頃は半分ネタだと思われていたが，今は実用化されている
  - 富士通 A64fx や AMD Zen はこれを使っている

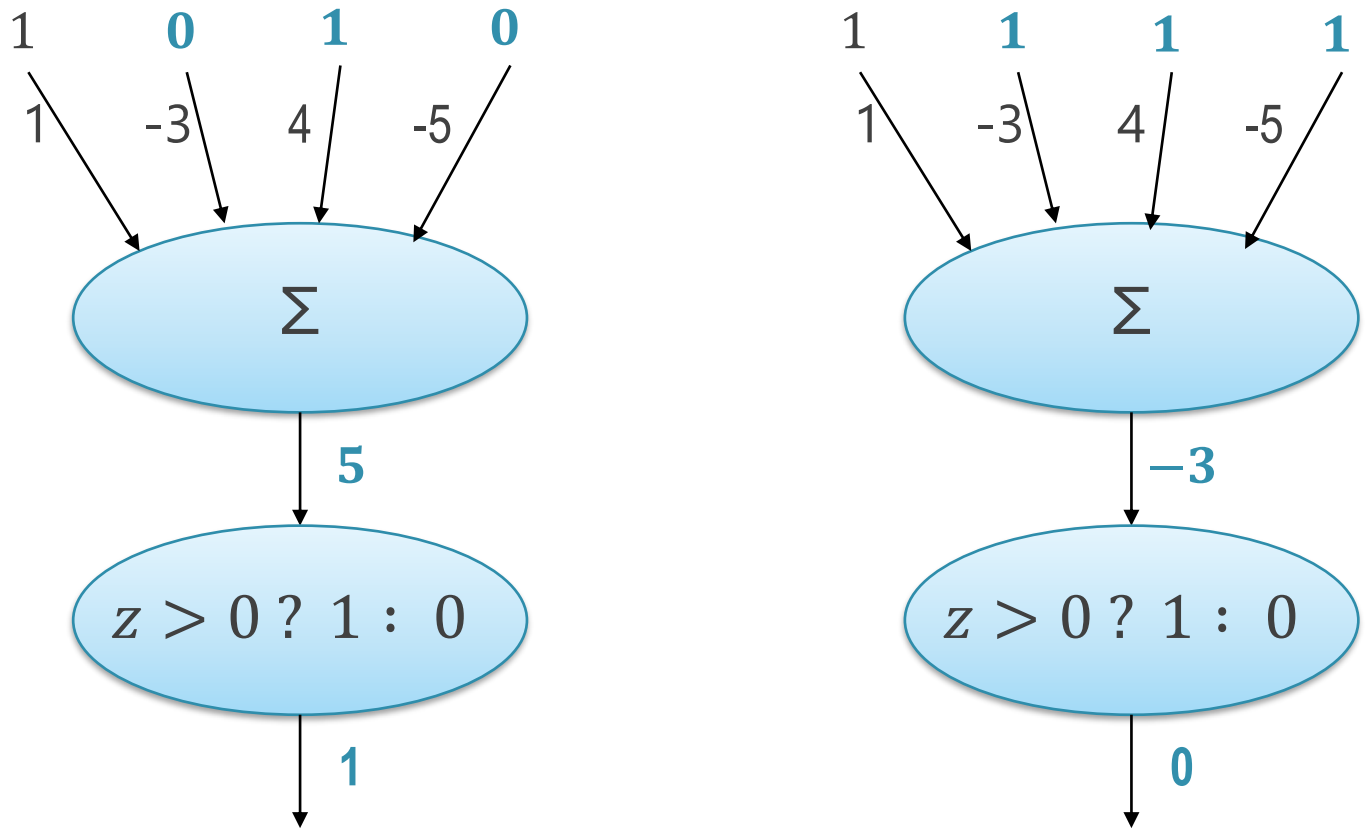
パーセプトロン



- $z = w_0 + \sum_{i=1}^{m} w_i x_i$

- ◇ 入力（0 or 1）：x_1 から x_m
- ◇ 重み（アナログ値）：w_0 から w_m（w_0 はバイアス）
- ◇ 学習：
  - □ z が 1 の時，x_i が 1 だったなら w_i を大きく
    0 だったなら w_i を小さくする

「010 なら 1 」を学習させた場合

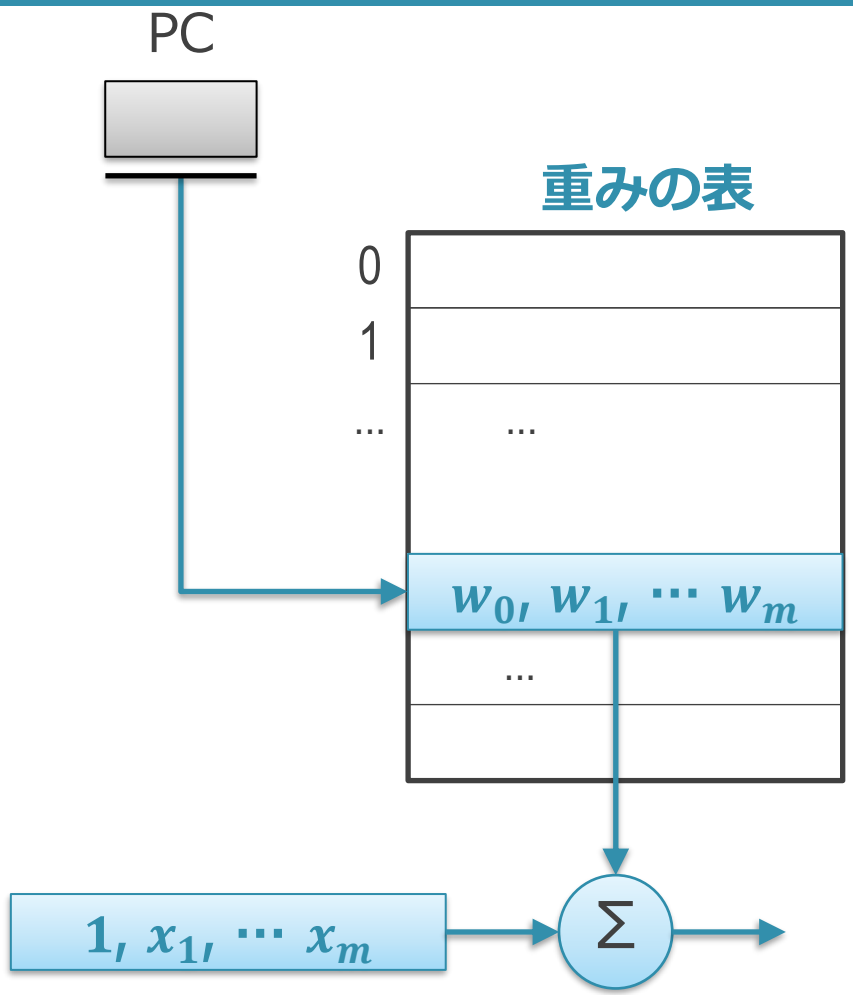


- 010 以外のパターンが来た場合は，z は負の方向に
- 左端の係数はバイアス
  - ◇ 真ん中の「$z > 0$?」 は，「$z < w_0$?」と等価の意味になる

パーセプトロン予測器

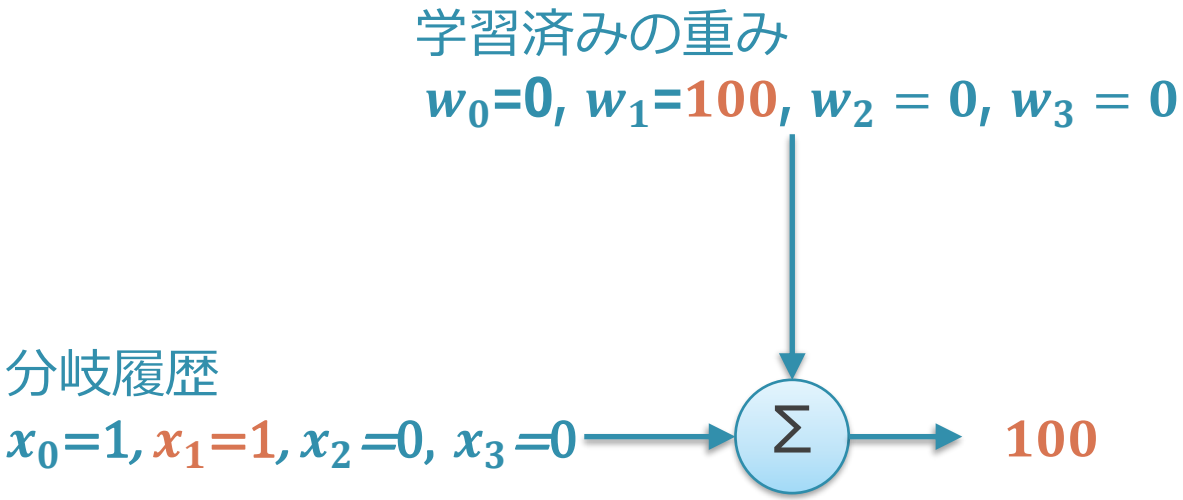
- 結構バリエーションがある
  - グローバル履歴をそのまま重みにかけるもの
  - g-share 的なテーブルから重みを出すもの
    - Hash perceptron
    - O-GHEL

グローバル履歴をそのまま重みにかけるもの



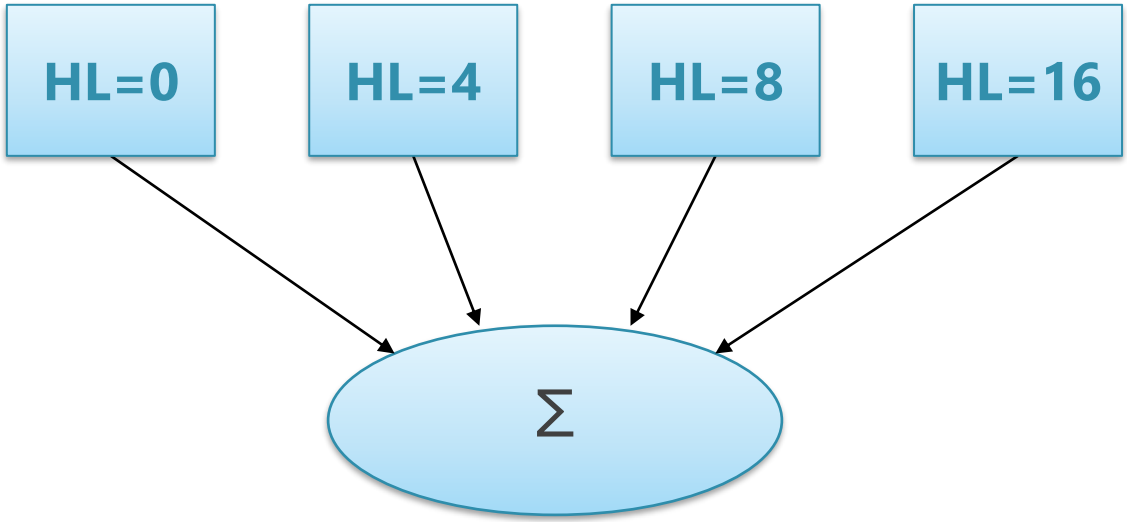
- PC の一部をインデクスとして重み表をひく
  - その PC に対応した重みのセットがとれる
- あとはグローバル履歴を x_i としてパーセプトロンの処理を行う

実際の学習の様子



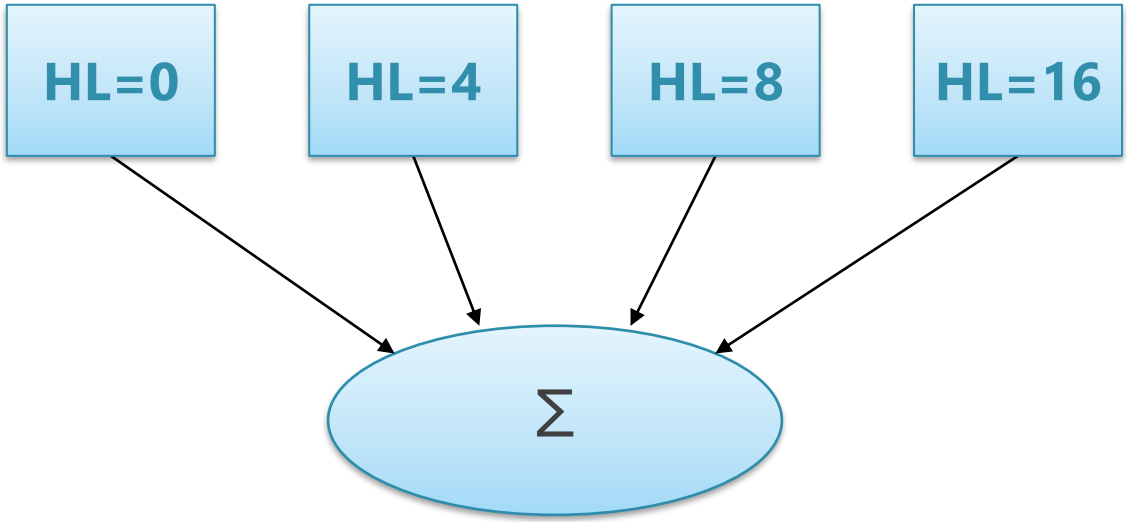
- x_4 の方向を予測する
  - x_4 は x_2 x_3 の方向とは関係なく，x_1 の方向のみと相関する場合を考える
  - x_1 に対応する w_1 の重みを大きく，それ以外の絶対値を小さくなるよう学習すればよい
    - 分岐方向に応じて毎回重みを加算/減算すれば，成立に偏る場合は大きくなり，無相関な場合は 0 に近づく
  - 内積をとると，x_1 が 0 か 1 かにのみ応じて結果が変わる

Hash perceptron
（パーセプトロン予測器に分類されるが，前ページまでのものとはかなり違う）



- 履歴長（HL）が異なるグローバル予測器のようなテーブルを複数用意
  - ◇ 履歴 と PC の組み合わせでテーブルを引く（グローバル予測器と同じ）
  - ◇ 中身は 2 ビットカウンタではなく，重み（8 ビットなど）にする
- History length (HL)=4 のテーブルの場合
  - ◇ 4ビットの履歴と PC を XOR してそれをインデックスにしてアクセス
  - ◇ 8 ビットの重み（符号付き）が取れてくる
    - □ 符号なし 2 ビットにすれば本当にただのグローバル分岐予測器と等価

Hash perceptron



- 全員の出力を加算してパーセプトロンの処理
  - ◇ 通常のパーセプトロン予測器と違い，各テーブルから出てきた重みはそのまま加算される
  - ◇ 履歴と重みの内積はとらない
    - □ 履歴は各テーブルをアクセスするときに使用されている

ハッシュ・パーセプトロンの効果

- g-share やオリジナル・パーセプトロン予測器の問題
  - ◇ 長い履歴を捉えようとすると履歴長に比例した記憶容量が必要
  - ◇ 長い履歴を憶えようとすると，短い履歴で済む場合も大きな記憶容量が必要
- ハッシュ・パーセプトロンの効果：
  - ◇ 長い履歴をきちんと憶えつつ，履歴が短い大半の場合は少ないエントリ数で記憶できる
    - □ 相関がある履歴長のテーブルの重みの絶対値が増えるように学習される
    - □ 相関がない履歴長のテーブルは平均的には変化しない（ランダムに汚される）

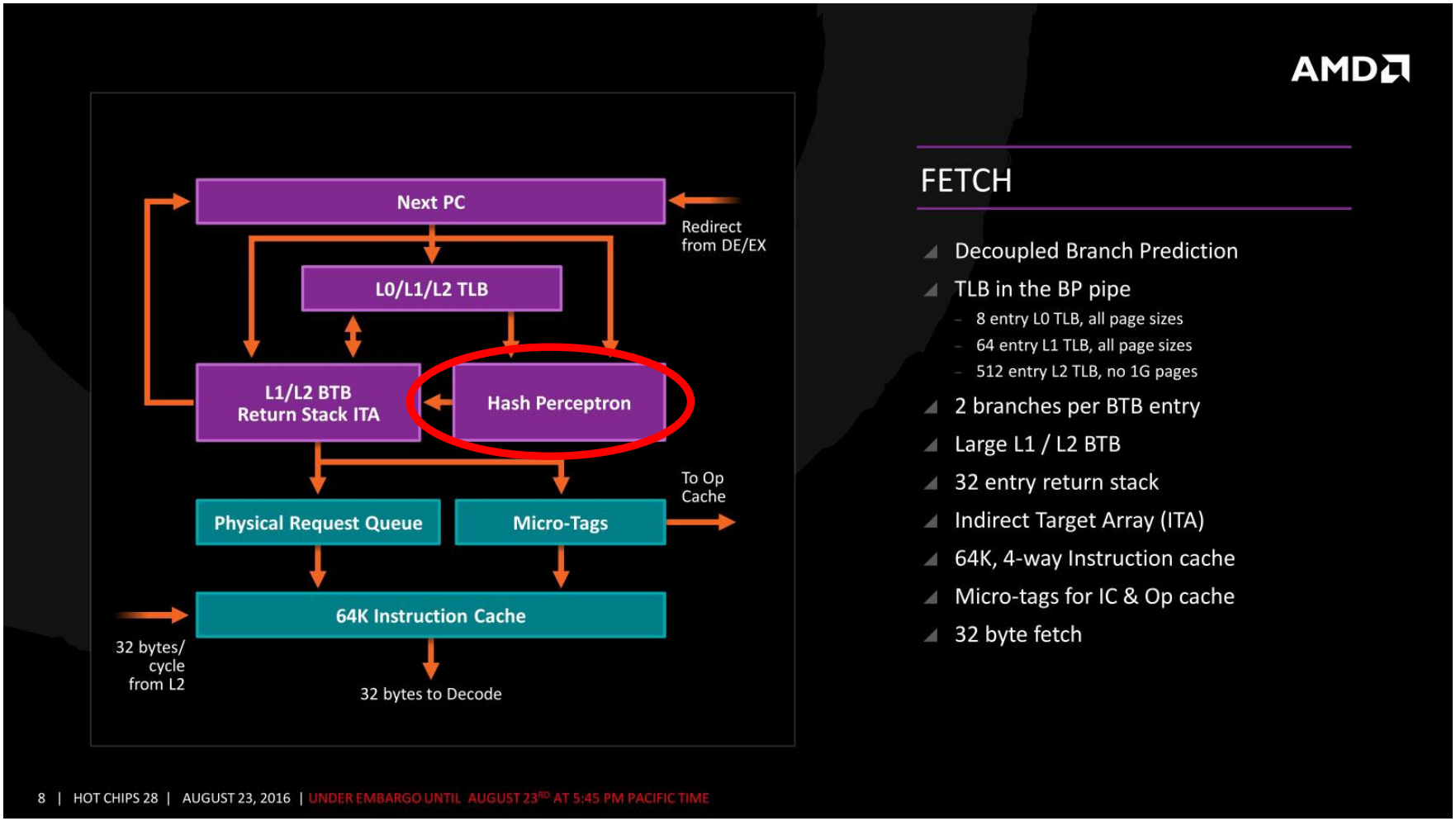
ハッシュ・パーセプトロンの効果の例

- たとえば 101 と 11111 の２パターンがあった場合
  （左から右に taken:1, untaken:0）

- 固定長グローバル履歴予測 (HL=5)
  - ５カ所を更新：00101, 01101, 10101, 11101, 11111

- ハッシュ・パーセプトロン (HL=3,5) ：
  - 101：主に１カ所を更新
    - HL=3のテーブルでは，101 の重みを大きく
    - HL=5のテーブルでは，00101, 01101, 10101, 11101 に散らされるので薄く汚す（平均的には変化しない）
  - 11111：主に１ or 2 カ所を更新
    - HL=3のテーブルでは，111 の重みを大きく？
    - HL=5のテーブルでは，11111 の重みを大きく

AMD Zen では Hash Perceptron を使ってるらしい



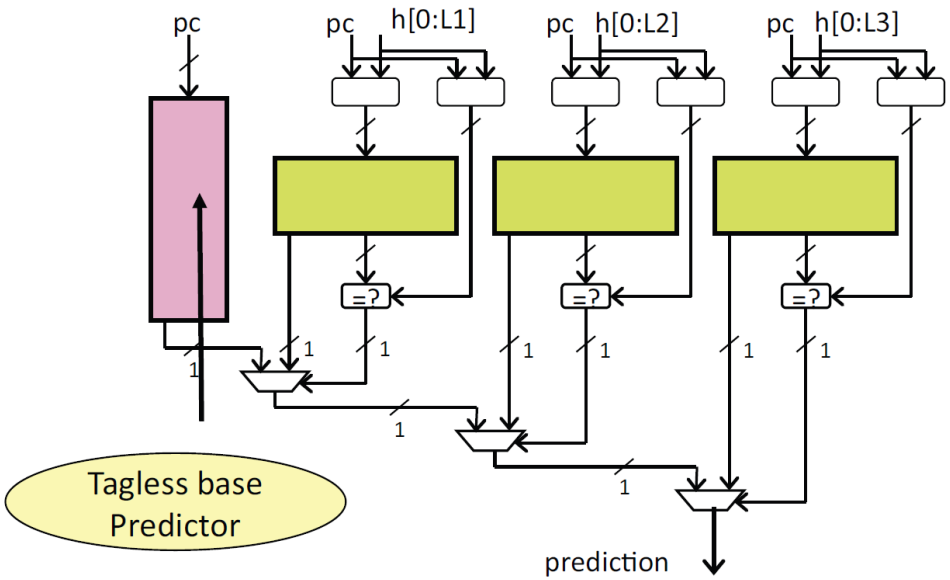
- HotChips 2016 の発表より

TAGE 予測器

- TAGE 予測器
  - A. Seznec and P. Michaud. A case for (partially)-tagged geometric history length predictors, Journal of Instruction Level Parallelism (http://www.jilp.org/vol8), 2006
- 現在最も予測精度が高いと言われている予測器
  - 最近のインテルの CPU に乗っている... らしい
  - E. Rohou, B. Narasimha Swamy, A. Seznec
    Branch prediction and the performance of interpreters — Don't trust folklore, 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)
- モチベーション
  - 必要となるテーブルのエントリ数は 2 の履歴長乗に比例するので，なるべく履歴が短いテーブルだけで学習を済ましたい

TAGE 予測器

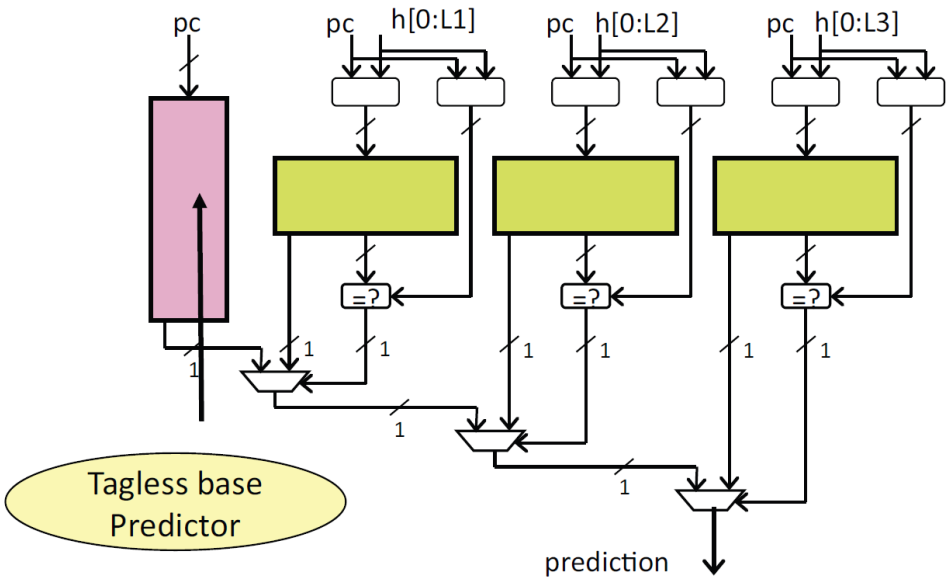
図は A. Seznec and P. Michaud. A case for (partially)-tagged geometric history length predictors より



- トーナメント式の構造：
  - 左端：　　単純な２ビット・カウンタ
  - それ以外：
    - グローバル履歴+PC でアクセスし，BTB の時のようにヒット/ミス判定を行う
    - ヒットした場合，そのテーブルの n ビット・カウンタの内容で予測

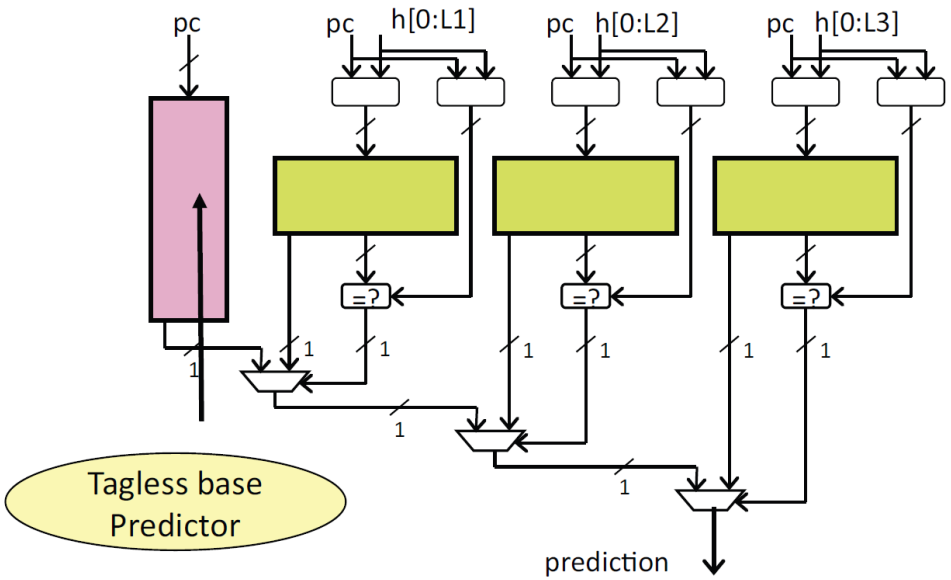
TAGE 予測器

図は A. Seznec and P. Michaud. A case for (partially)-tagged geometric history length predictors より



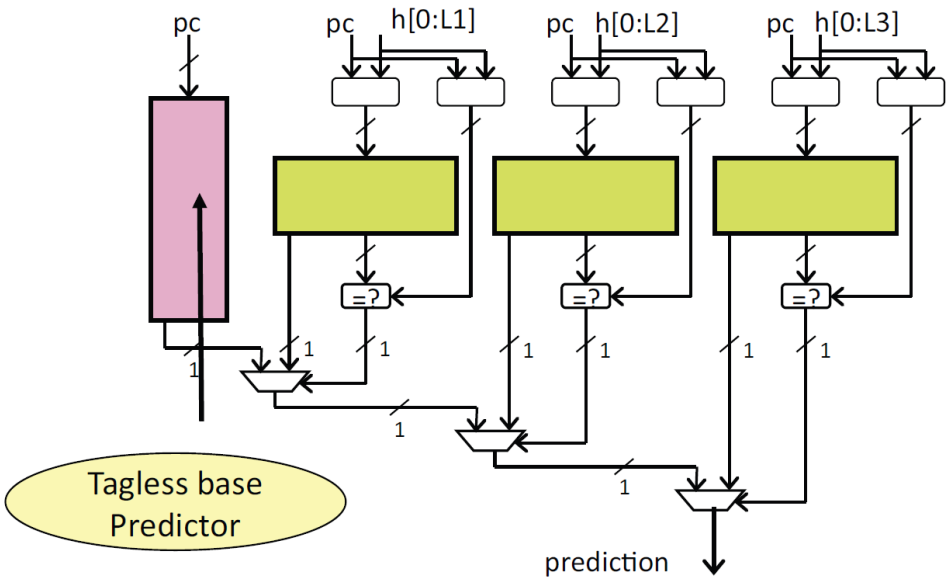
- パターン長が長いテーブルの結果を優先して使う
  - ◇ 右側でヒットするほど，その結果を優先する
  - ◇ 右に行くほど指数的に履歴長が長くなっている
- ヒット/ミスの判定ができるため，トーナメント状に優先度が決定できる
  - ◇ ただの PHT だと，だれの結果を使えばいいかわからない

TAGE 予測器のメリット（１）



- 利点１：パターン長ごとに，最適なテーブルに学習できる
  - ◇ 記憶すべきパターン数が減る
  - ◇ たとえば 101 と 11111 の２パターンがあった場合
    - □ 固定長(5)　　：00101, 01101, 10101, 11101, 11111
    - □ TAGE(3+5)　：101, 11111

TAGE 予測器のメリット（２）



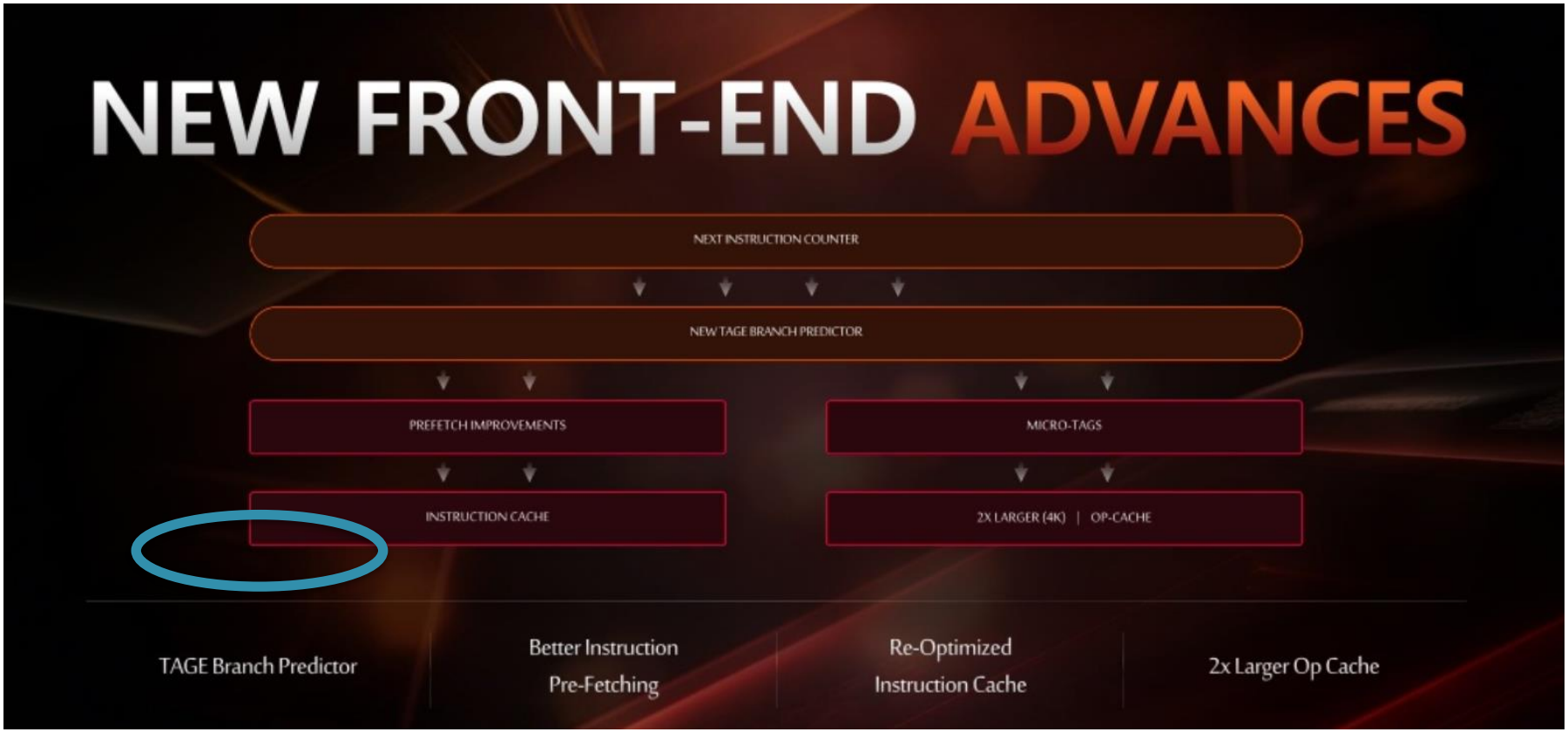
- 利点２：学習が早い
  - ◇ 大ざっぱに成立/不成立な傾向があるような場合，
  - ◇ 難しいパターンを学習する前に，とりあえず左端で大ざっぱな予測ができる
    - □ 左端は履歴を使わないただのカウンタだから

TAGE 予測器 と パーセプトロン予測器

- それぞれ，今一番予測精度が高いと思われている
  - インテルや AMD の CPU でも実際に採用
- TAGE の方が，基本的には良い精度を示す
  - しかし，かなりのチューニングがいる
    - 特に各テーブルのサイズのバランスが難しい
  - ちょうどテーブルに履歴長が収まらず，はずみでガクッと精度が落ちたりする
- パーセプトロン予測器は，結構適当でも大丈夫だし安定している
  - 企業の人的には，そこがありがたいらしい

AMD Zen2 (Ryzen 9)

- 方向に分岐予測器に TAGE を導入したみたい
  - ◇ 実際は TAGE （低速高精度）とパーセプトロン（高速低精度）のハイブリッドらしい
  - ◇ とりあえずパーセプトロンが言った方向でフェッチしておいて，遅れて TAGE が言った方向が違ったらそっちでやり直す



最近の分岐予測器の研究

- TAGE をベースに，補助予測器（ループ専用，ローカル予測器，統計的補正など）をつけたもの
  - L-TAGE [Seznec07]
  - ISL-TAGE [Seznec11]
  - Wormhole [Albericio14]
- TAGE の構造をより簡単にし，補正も入れずに同等の性能を出すもの
  - BATAGE [Michaud18]
- 近年でもさらに発展を続けている
  - 今日話した話の大半は A. Seznec と P. Michaud の 2 人の成果

予測器の精度

Pierre Michaud, An alternative TAGE-like conditional branch predictor, TACO 2018 より

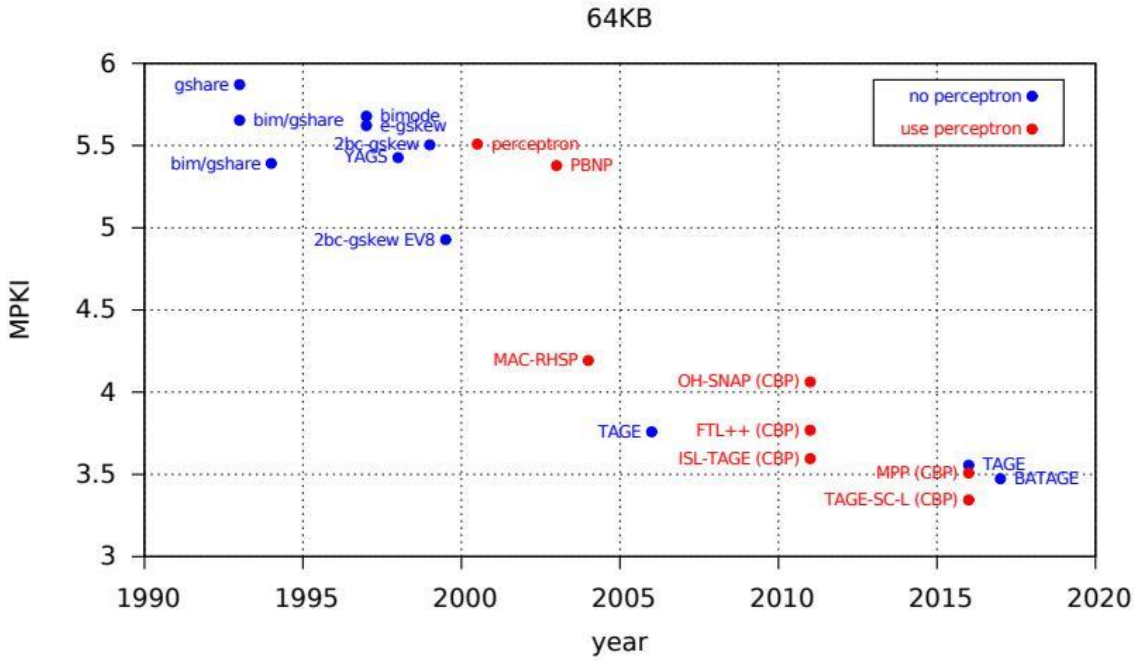


Figure 1: Average number of mispredictions per 1000 instructions (MPKI) for various conditional branch predictors on the CBP 2016 traces for 8KB, 32KB and 64KB storage budgets (see Appendix A).

実際の例

- Adam Collura, Anthony Saporito, James Bonanno, Brian R. Prasky, Narasimha Adiga, Matthias Heizmann (IBM), The IBM z15 High Frequency Mainframe Branch Predictor, ISCA 2020

  - 汎用機むけの分岐予測器の実装
    - 通常の CPU とは少し前提が異なるのでちょっと注意

  - 汎用機（Main Frame）：
    - 昔からある IBM の大型コンピュータのこと
    - 尋常じゃなく信頼性を重視して作られている
    - 銀行とかで今でも使われている

  - 実際の製品の中身について書かれたものとしては，かなり詳しい
    - TAGE やパーセプトロンが実際に使われているようだ

今日の内容

1. 分岐予測（最終回）
    1. 動的分岐方向予測の続き
    2. **間接分岐予測**

間接分岐命令

- 方向分岐命令
  - 成立 or 不成立で行き先が分岐
  - bne x1, x2, TARGET
  - if 文に相当
- 間接分岐命令
  - レジスタに格納されているアドレスに飛ぶ
  - 「`jr x1`」は，x1 レジスタに格納されている値のアドレスにジャンプする間接分岐命令

間接分岐命令の使いどころ

- 関数ポインタ呼び出し，仮想関数呼び出し
  - ◇ 関数ポインタ変数やオブジェクト内の仮想関数ポインタをロード
  - ◇ ロードしたアドレスに間接分岐で飛ぶ
- switch-case 文
  - ◇ 例：0 ～ 255 の 256 のパターンの case がある場合
  - ◇ 各 case の処理の先頭の命令アドレスを要素数 256 の配列に格納
  - ◇ 配列の要素をロードして，そこに間接分岐で飛ぶ
- return 文
  - ◇ 関数呼び出し時に，呼び出し元をメモリ上に保存
  - ◇ 呼び出し元アドレスをメモリからロードして，間接分岐で戻る

間接分岐予測

1. 間接分岐一般の予測
2. 関数からの return の予測

間接分岐一般の予測

- 基本的に間接分岐の予測は難しい
- 方向分岐：
  - ◇ 成立 or 不成立の 2 択を予測
  - ◇ 分岐先ターゲット・アドレスは不変
- 間接分岐：
  - ◇ 動的に変化しうる，無数にある分岐先アドレスを予測
  - ◇ 間接分岐の飛び先は，ロードした結果など = 64 bit などの値

簡単なもの：BTB を使った間接分岐の予測

- 基本的には，「前回飛んだ先に今回も飛ぶ」と予測
- BTB に「分岐の種類」を表す１ビットを格納して予測する
  - ０：方向分岐 →
    - これまで説明した方法で予測
  - １：間接分岐 →
    - 方向分岐予測の結果は無視
    - BTB の読み出し結果をそのまま使う
- BTB には前回に間接分岐を実行した際の飛び先が入っている
  - これを読み出してそのまま飛び先として予測

もう少し凝ったもの

- BTB を引くときのアドレスに履歴をまぜる
  - ◇ 方向分岐の，グローバル分岐履歴を使うものと同じ
  - ◇ 過去に辿ってきた分岐方向によって飛び先を変えることができる
  - ◇ ARM Cortex A15 などで搭載
- より発展させたものとしては，TAGE 予測器を応用したものなどが提案されている

間接分岐予測

1. 間接分岐一般の予測
2. **関数からの return の予測**

関数からの return の予測

- 関数の call
  - ◇ 関数の先頭アドレスは固定
  - ◇ したがって，飛び先も固定
- 関数からの return
  - ◇ 同じ関数でも，呼び出し元は無数に存在
  - ◇ 直前に実行された call に戻ると期待できる
- 上記の性質を使って，return は高精度に予測できる

Return Address Stack (RAS) を使った予測

- 使用する構造：
  - RAS：
    - 関数の呼び出し元を格納するスタック
  - BTB：
    - BTB の「分岐の種類」を表すビットを拡張
      - 関数コール or リターンであることを記録
      - デコードするまで，関数呼び出しであることはわからない
- 動作：
  - BTB 引きの結果が関数コール：
    - スタックにその時の PC + 4（戻り先）を積む
  - BTB 引きの結果がリターン：
    - スタックのトップを読んで，そこに戻ると予測

分岐予測のまとめ

- 今日の内容：
  - 高度な予測器
    - パーセプトロン予測器
    - TAGE 予測器
  - 間接分岐予測

- 「付録1:分岐予測の詳細 」（github 上にある）の内容：
  - パーセプトロン予測器や TAGE 予測器の詳細
    - Piece-wise パーセプトロン，O-GEHL
    - BATAGE（Bayesian TAGE）
  - 予測ミスからの回復の実装方法
  - 複数命令を同時にフェッチする場合のやりかた

興味がある人向け

- Branch Prediction Championship
  - ◇ https://jilp.org/cbp2016/framework.html
- 各種アルゴリズムを評価するシミュレータ
  - ◇ レポート課題の選択の１つに，解析結果を出したい

今日の内容

1. 分岐予測（最終回）
    1. 動的分岐方向予測の続き
    2. 間接分岐予測
2. **メモリ**

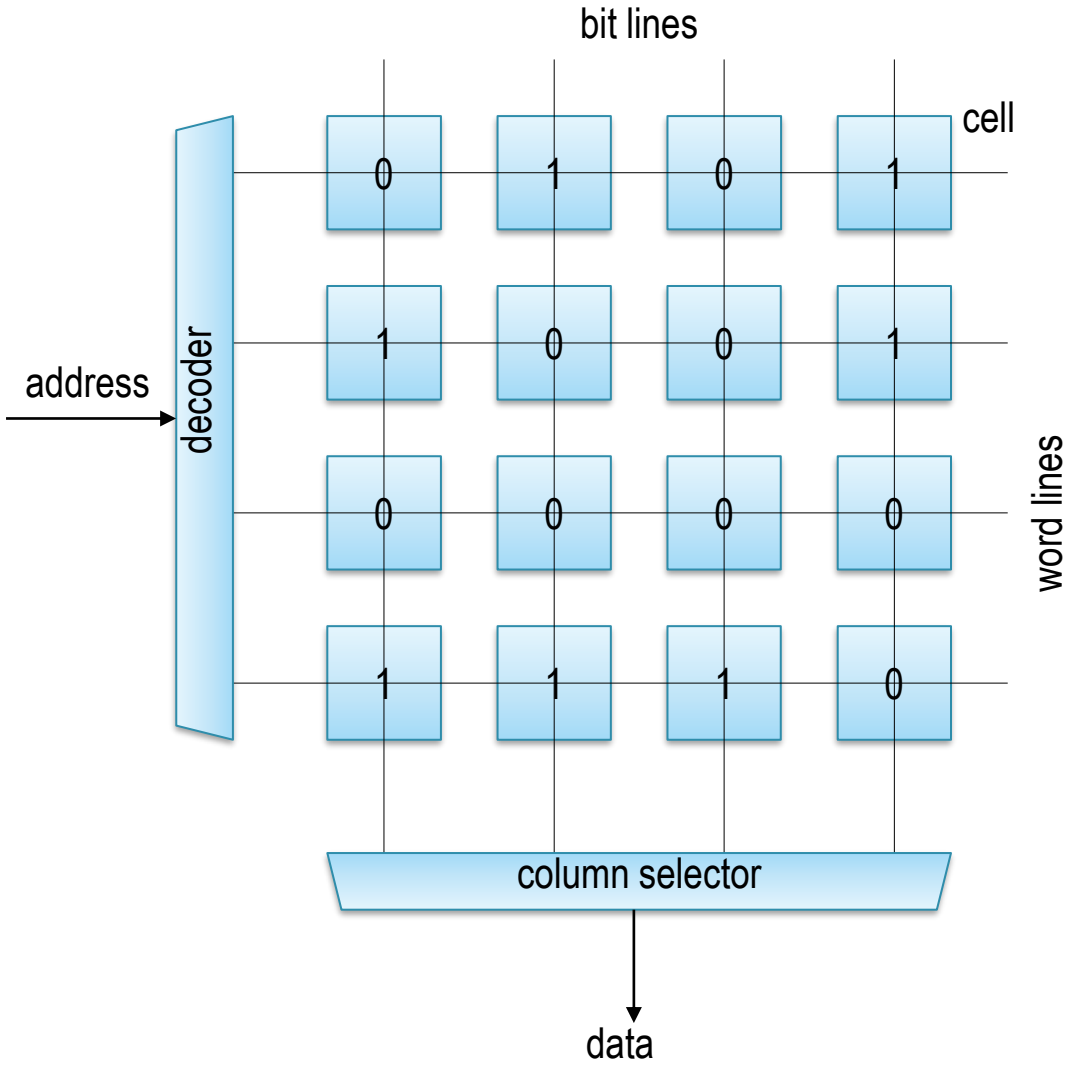
メモリ

- メモリ：RAM（Random Access Memory）
  - ◇ 複数のデータを記憶する回路
  - ◇ 配列のように，位置を指定して読み書きする
- なぜデータを記憶するための専用の構造が使われているのか？
  - ◇ D-FF とマルチプレクサでも同じ機能のものは作れる
  - ◇ メモリの方が圧倒的に速度や面積効率が高い
    - □ 後半で説明
- なぜメモリの構造を勉強するのか？：その性質の理解が重要だから
  1. 高度な CPU はメモリで作られた「表」を多用する
  2. 主記憶やキャッシュなどは直接メモリで作られる

メモリ

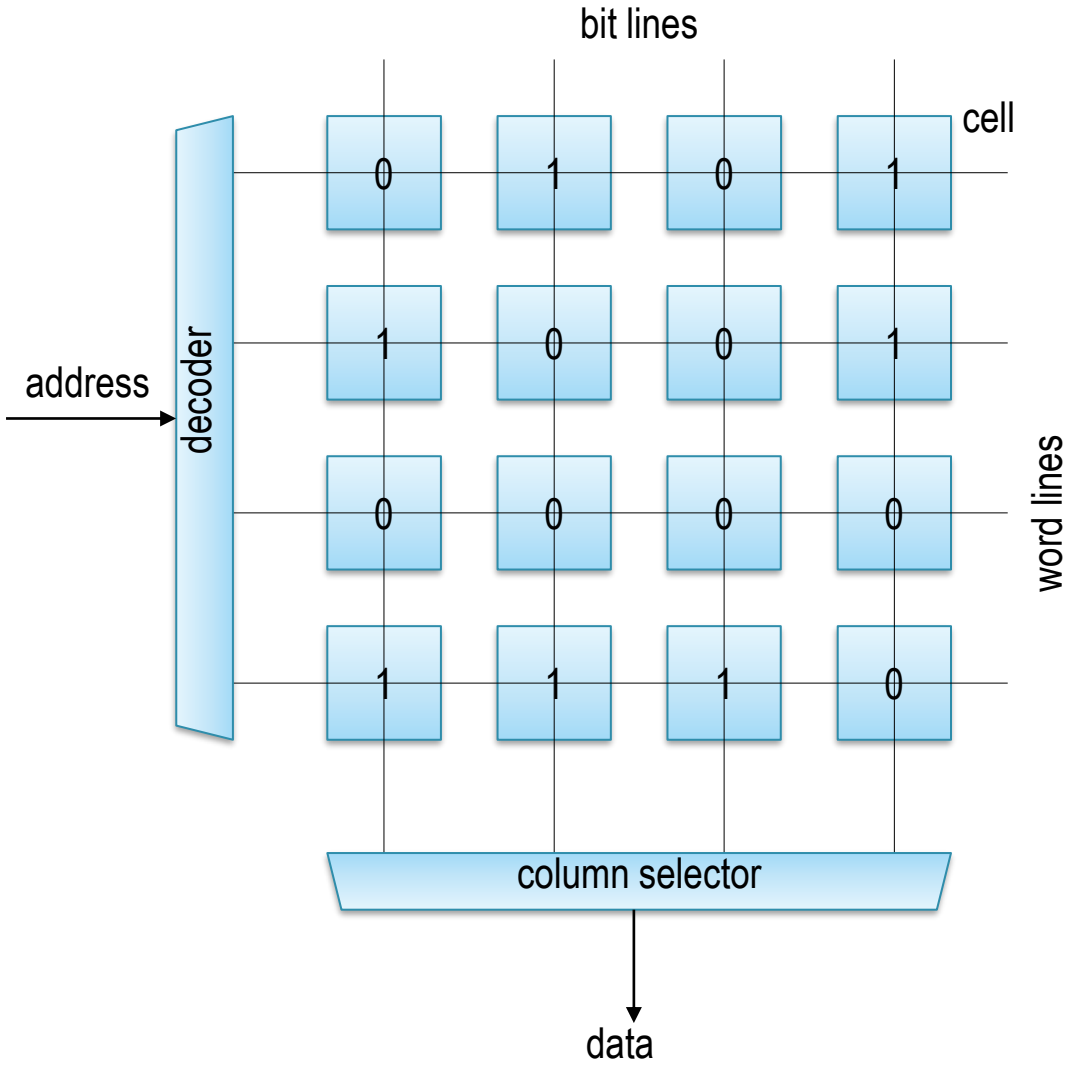
1. **メモリの基本**
    1. 構造
    2. 動作
    3. アクセス時間
2. メモリの詳細
    1. SRAM と DRAM
    2. なぜメモリが存在するのか？
    3. マルチポート・メモリ
    4. メモリの脆弱性を対象にしたアタック
        - Raw Hammer, Cold Boot Attack

メモリの基本構造：セルを行列状に配置



- セル：
  - 1ビットの情報を記憶
- ビットライン：
  - 列方向の配線
- ワードライン：
  - 行方向の配線
- デコーダ：
  - アドレスをデコードしてワードラインをアサート
- カラム・セレクタ：
  - ビットライン出力から１つを選んで出力

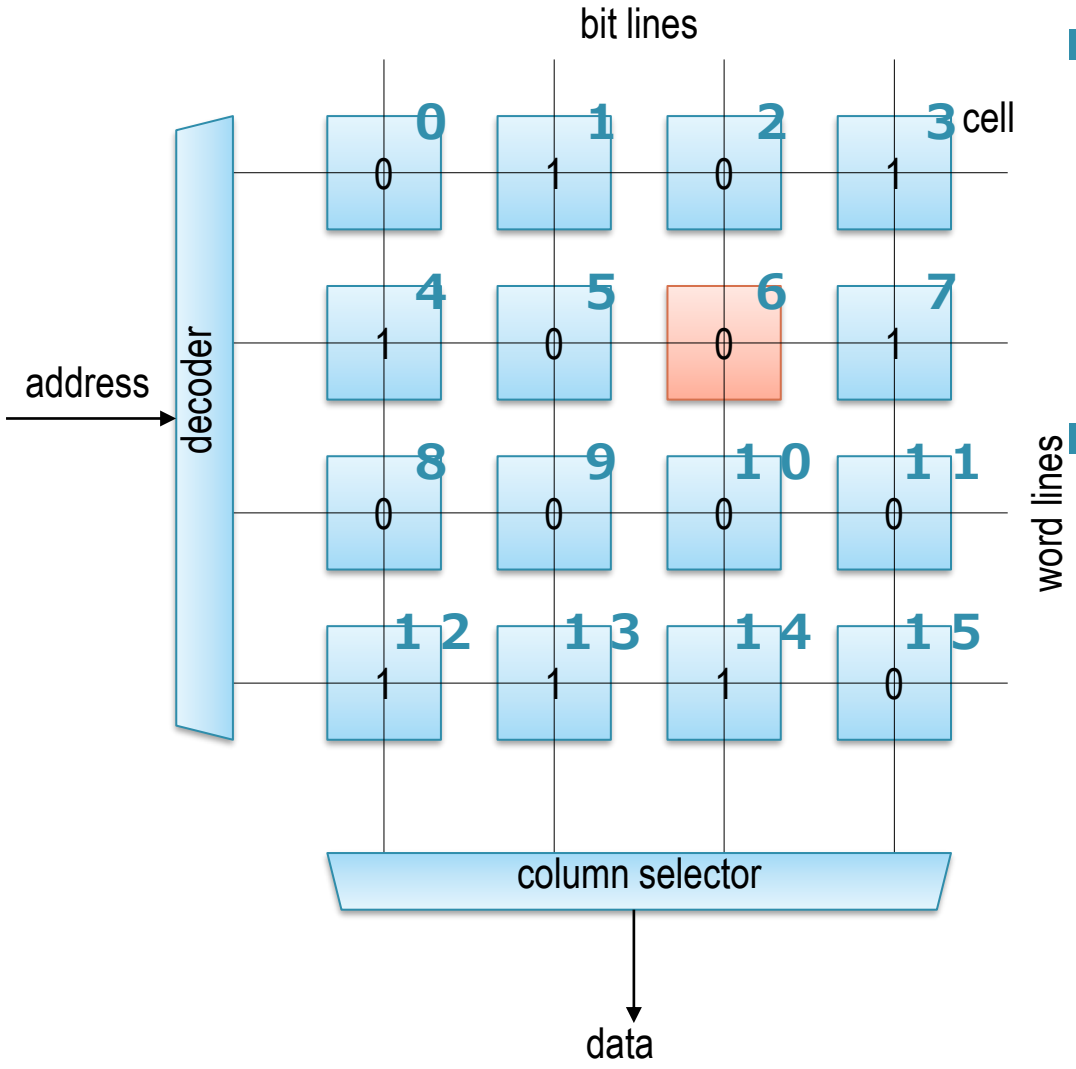
メモリの読み出し操作



1. アドレスのデコード
2. ワードラインのアサート（行の指定）
3. ビットラインへの１行分のデータの読み出し
4. カラムセレクタによる目的のビットの選択

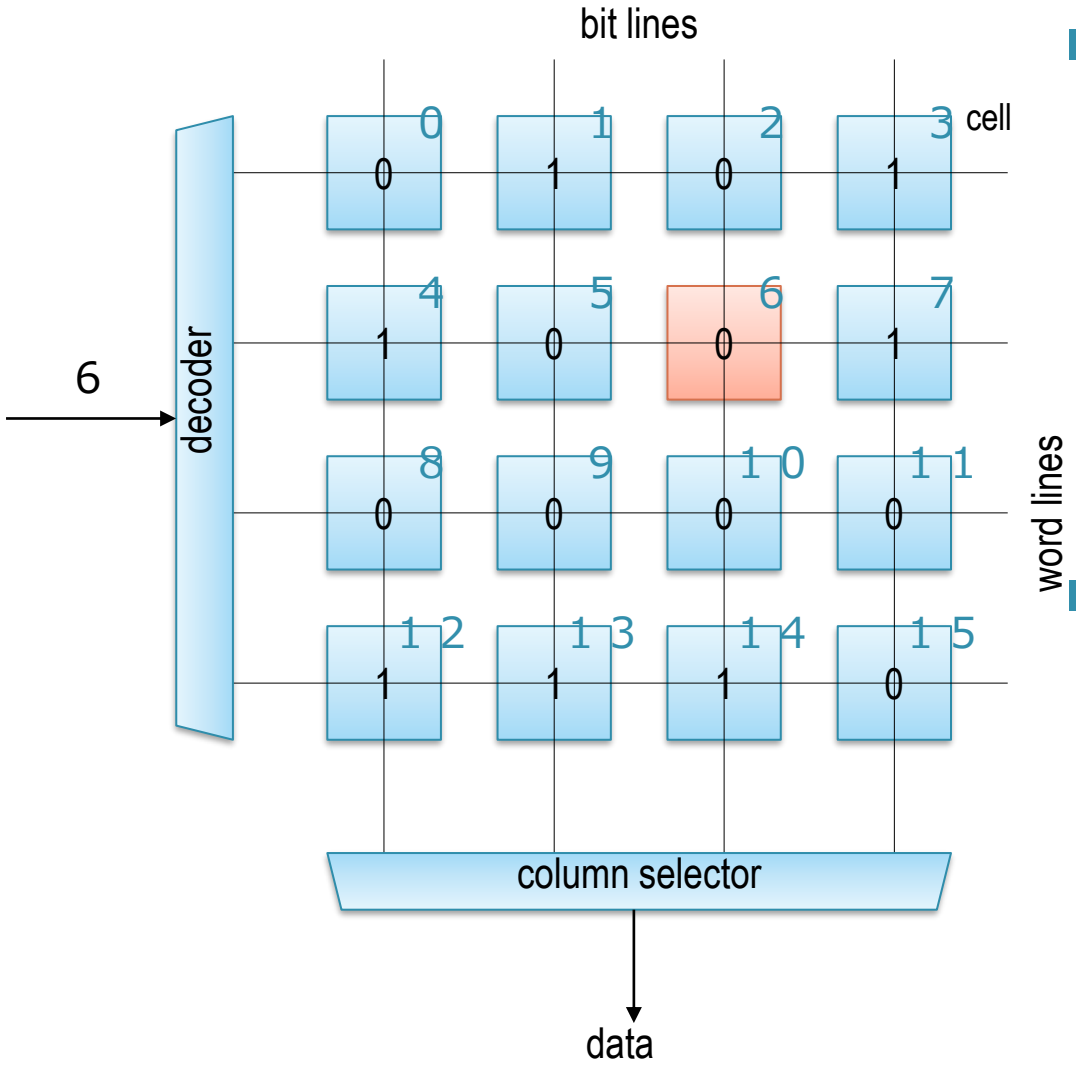
◇ （書き込みはこれの逆を行う

メモリの読み出し動作の例



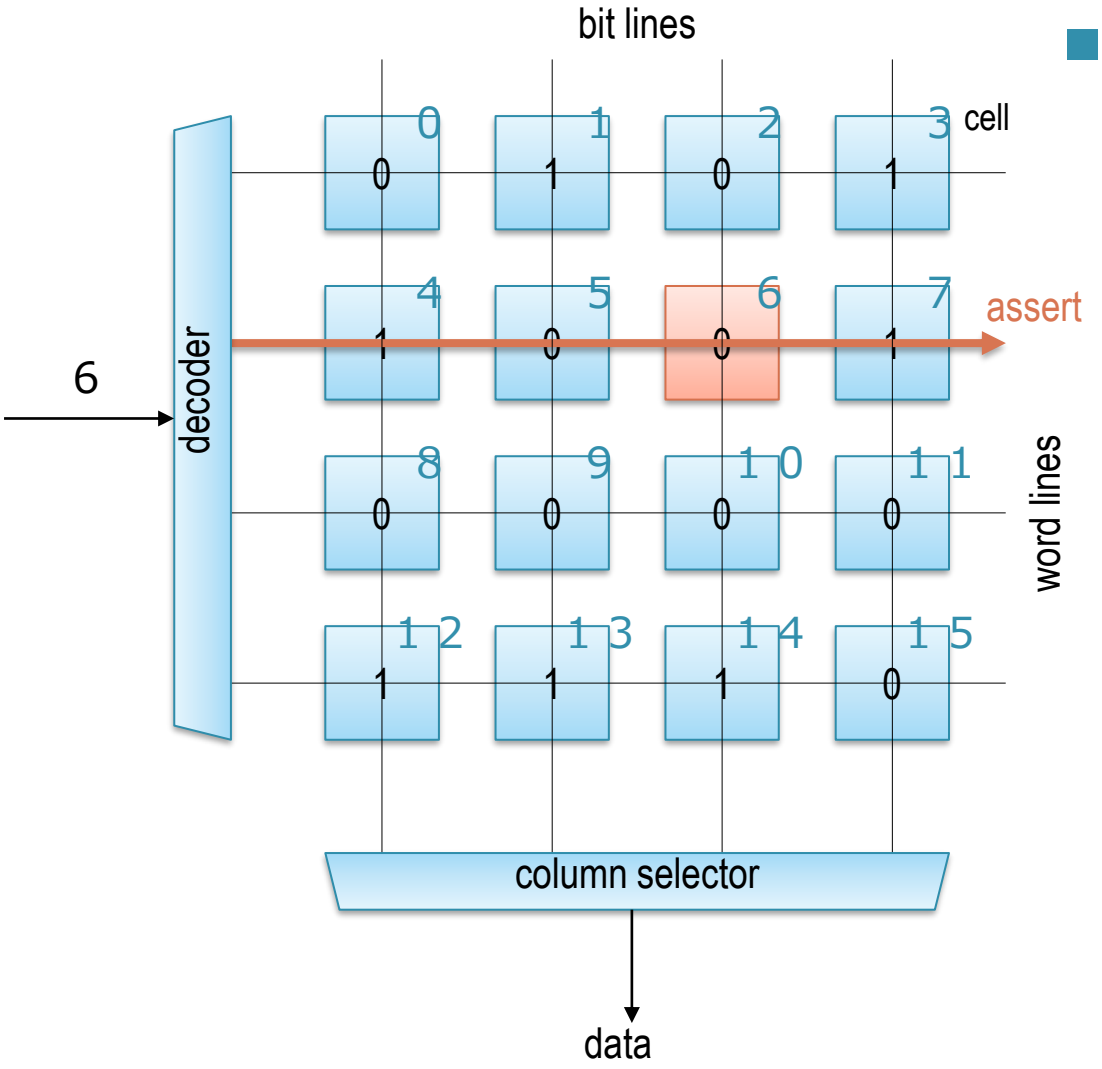
- 要素数 $16 = 2^4$
  - ◇ アドレスは4ビット
  - ◇ 各セルの右上の数字がアドレス
- このメモリの読み出し操作を例として説明
  - ◇ アドレス６のセルを読み出す

メモリの読み出し動作（１）
アドレスのデコード



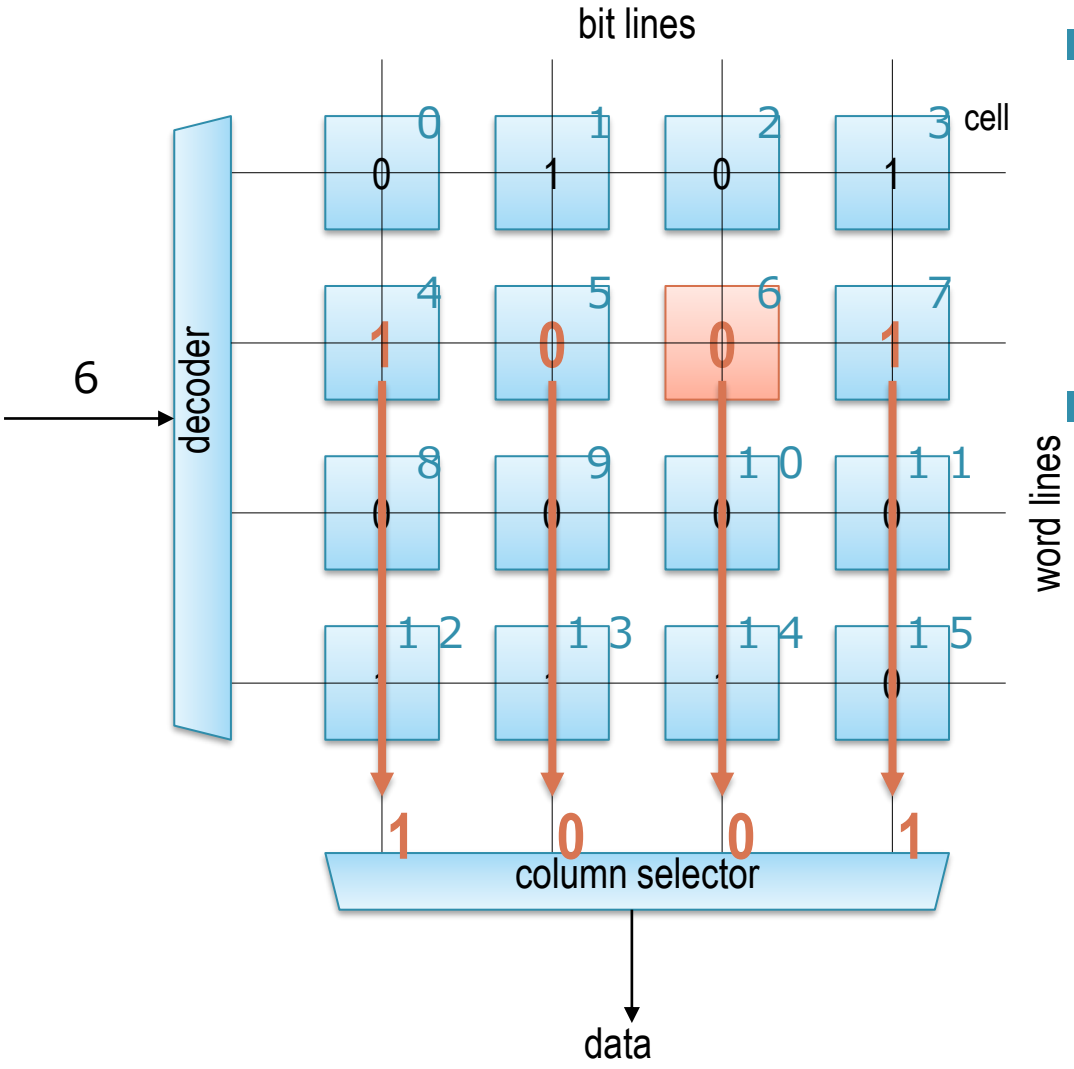
- どの行を読むか決める
  - ◇ 各行は４つセルがある
  - ◇ アドレスを４で割って切り捨てた行目が読むべき場所
  - ◇ 6 =0110（２進数）
- デコーダ
  - ◇ 数字を対応するワンホット信号に変換する回路
  - ◇ ワンホット信号：
    - □ n 本のうち，１つだけが１で他が０の信号
  - ◇ アドレス上位の２ビットをデコード

メモリの読み出し動作（２）
ワードラインのアサート



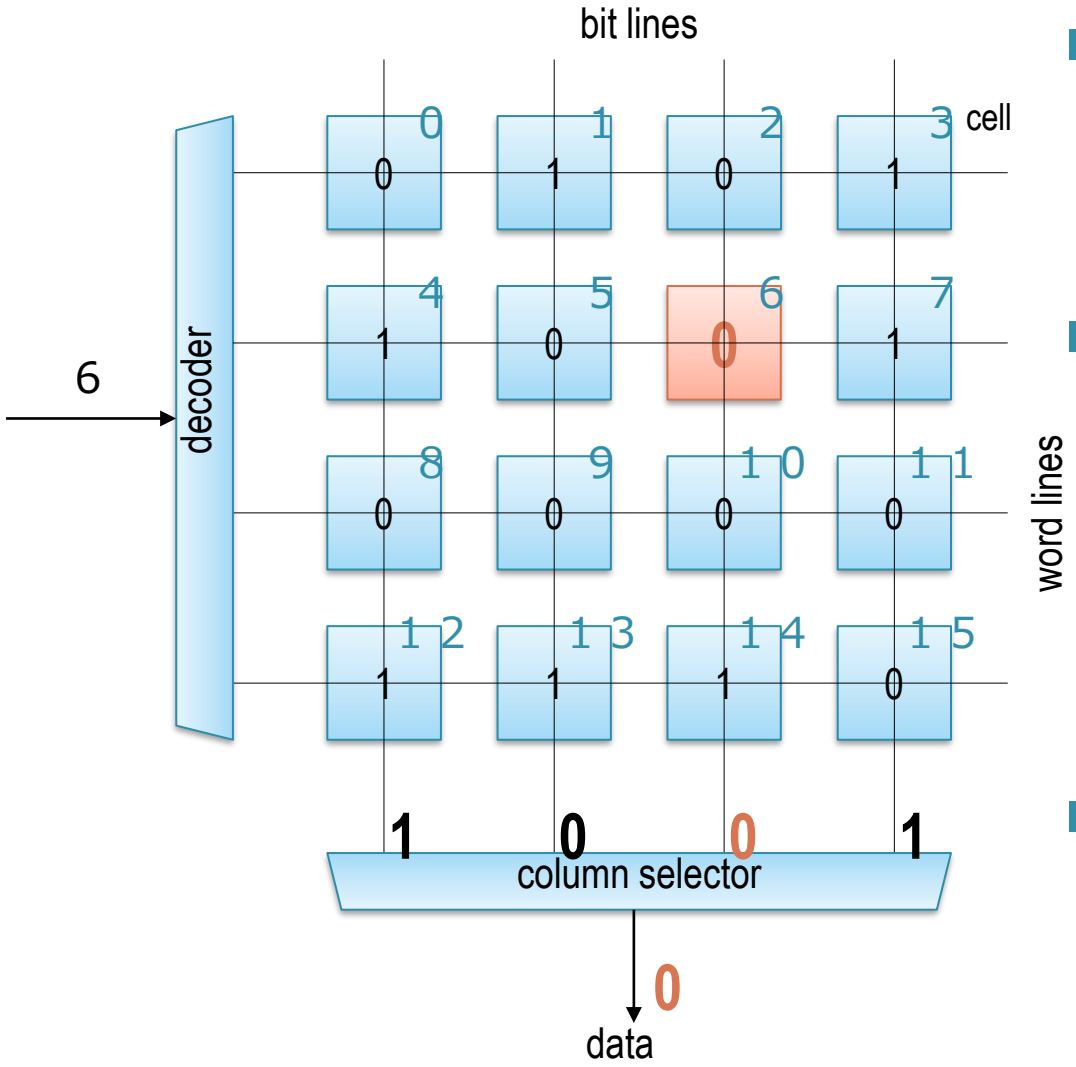
- 読み出す行に対応したワードラインをアサート
  - ◇ 先ほどのデコード結果を使う

メモリの読み出し動作（３）
ワード（１行分のデータ）の読み出し



- ワードラインがアサートされると，
  そこに接続されたセルがビットラインに自身の中身を流す
- これにより，１行分のデータが読み出される

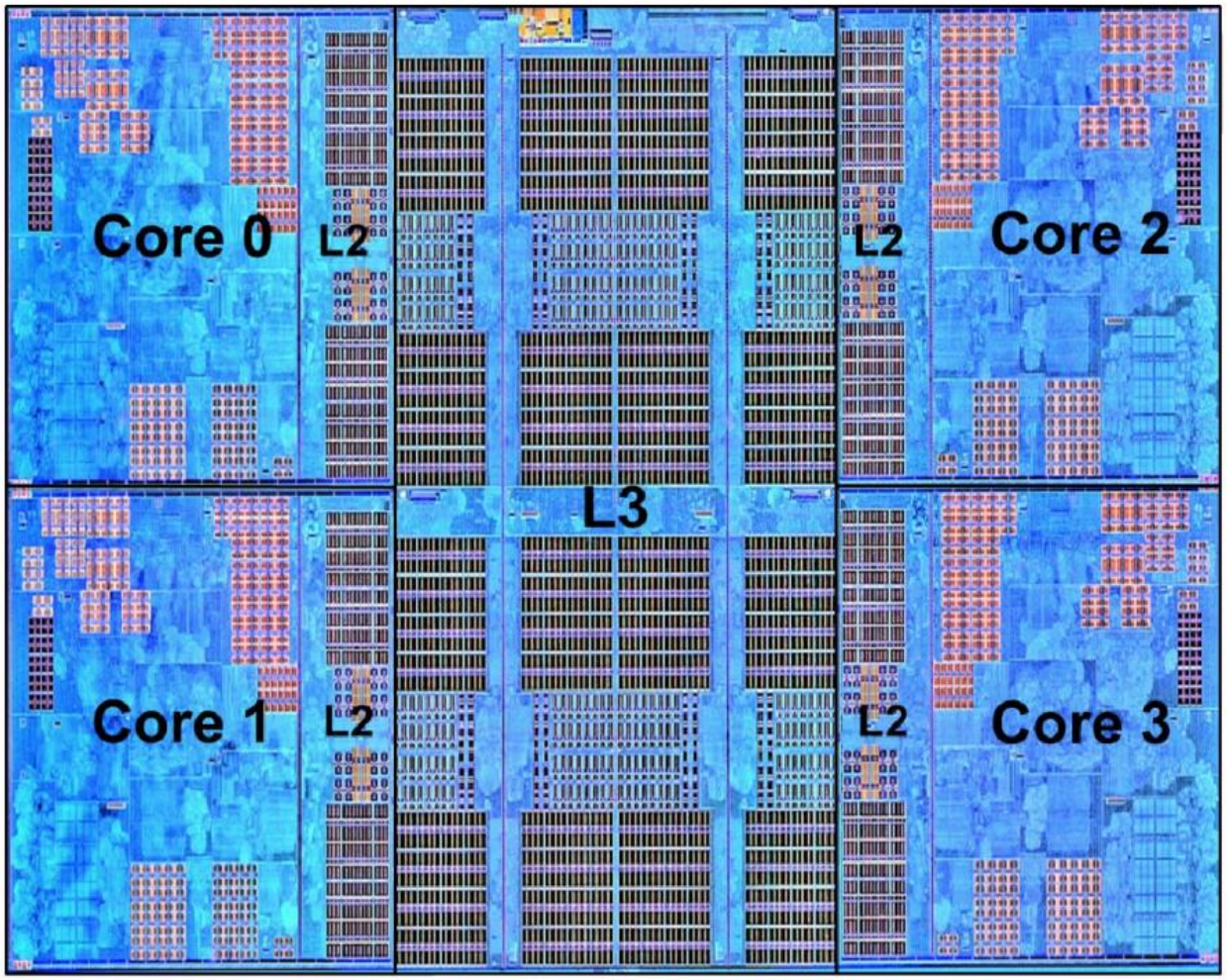
メモリの読み出し動作（４）
列の選択



- 読み出された１行分のデータから，最終的に必要な１ビットを選択
- どの列を読むか決める
  - ◇ 各列は４つセルがある
  - ◇ アドレスを４で割った余りの列が読むべき場所
  - ◇ 6 =0110（２進数）
- カラム・セレクタによりビットを選択する
  - ◇ 読み出し幅によってはこの部分がないこともある

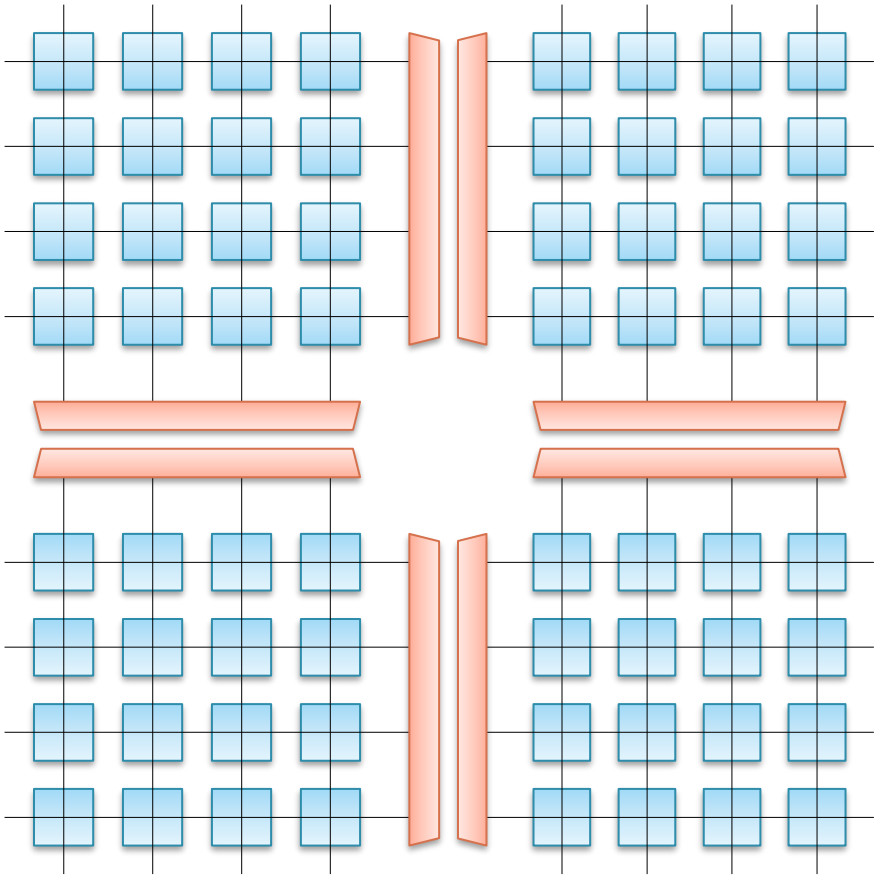
AMD Zen コアの場合

Teja Singh et al., Zen: An Energy-Efficient High-Performance ×86 Core より



- 赤や紫の「田」の字の構造の部分が全部メモリ（SRAM）
  - レジスタやキャッシュ，各種テーブルなど

「田」の字の構造

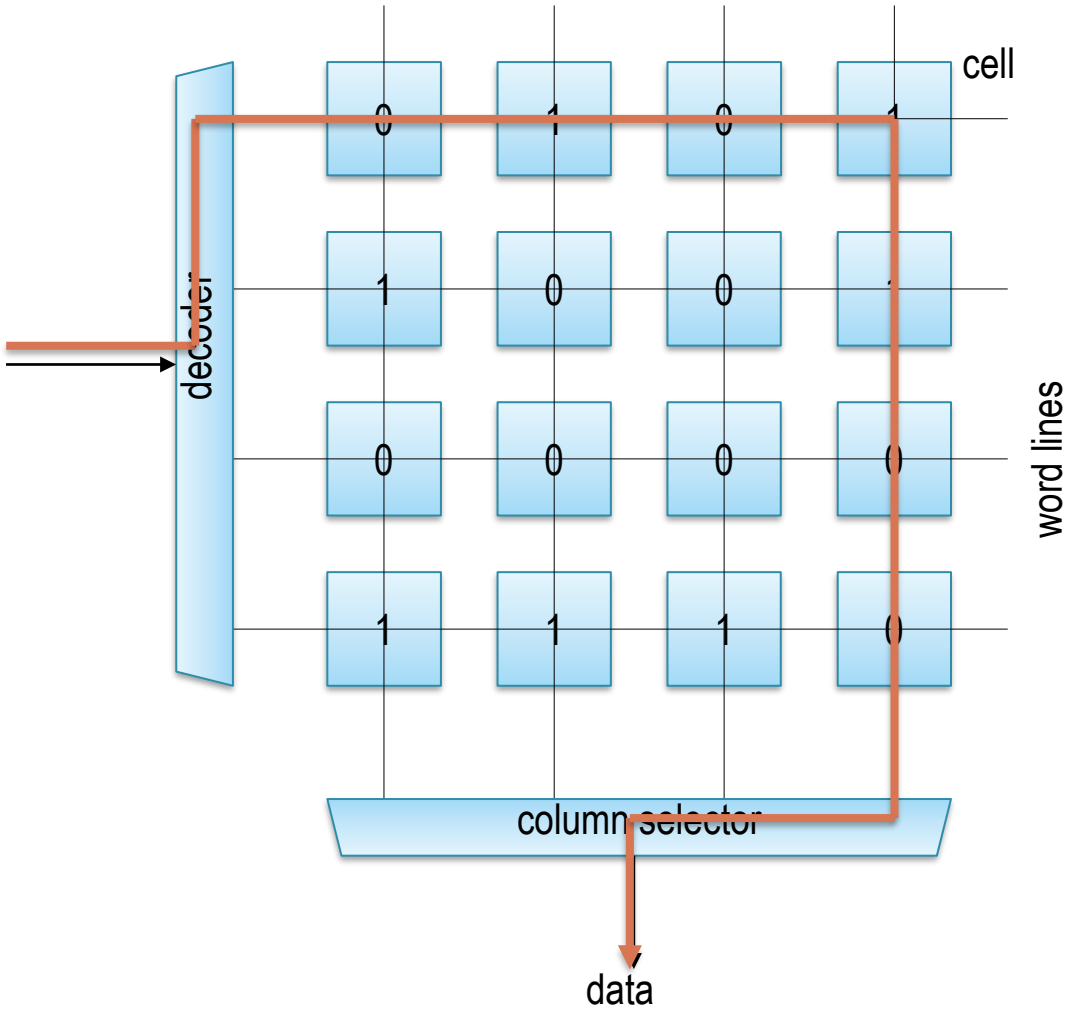


- 「田」の字の構造になっているのは,
  - ◇ 縦線がデコーダで左右に対してワードラインをアサート
  - ◇ 横線でビットラインのデータを拾う

メモリ

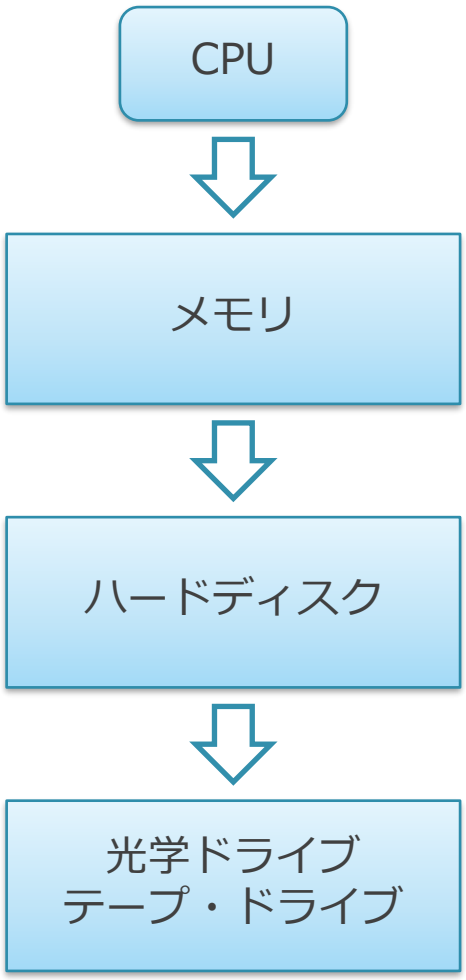
1. メモリの基本
    1. 構造
    2. 動作
    3. **アクセス時間**
2. メモリの詳細
    1. SRAM と DRAM
    2. なぜメモリが存在するのか？
    3. マルチポート・メモリ
    4. メモリを対象にしたアタック

メモリの読み出し時間



- メモリの読み出し時間：
  - ◇ 一番遠い経路を信号が通るときの時間
- 容量（セルの数）が一定の場合，正方形に近くするのが最も経路が短くなる

データをとってくるのに，どのぐらいかかるか？



1ns

マイ本棚：
1分（目的の本を探す）

100ns

名古屋：
95分（新幹線）

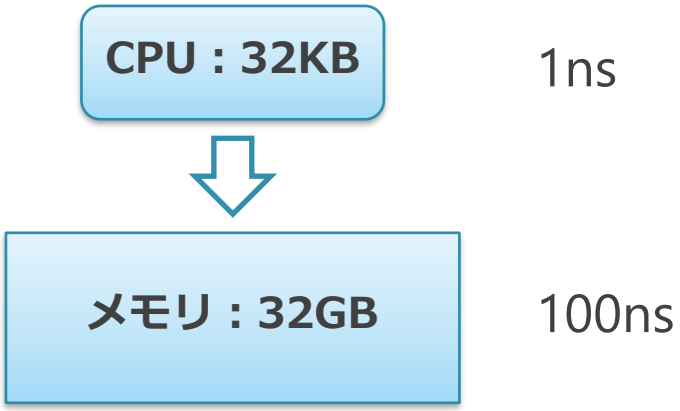
10^6ns

木星：
2年　（ロケット）

10^9ns

白鳥座デネブ：
2000年（光速）

アクセス時間は容量に直接は比例しない
（容量の平方根ぐらいに比例

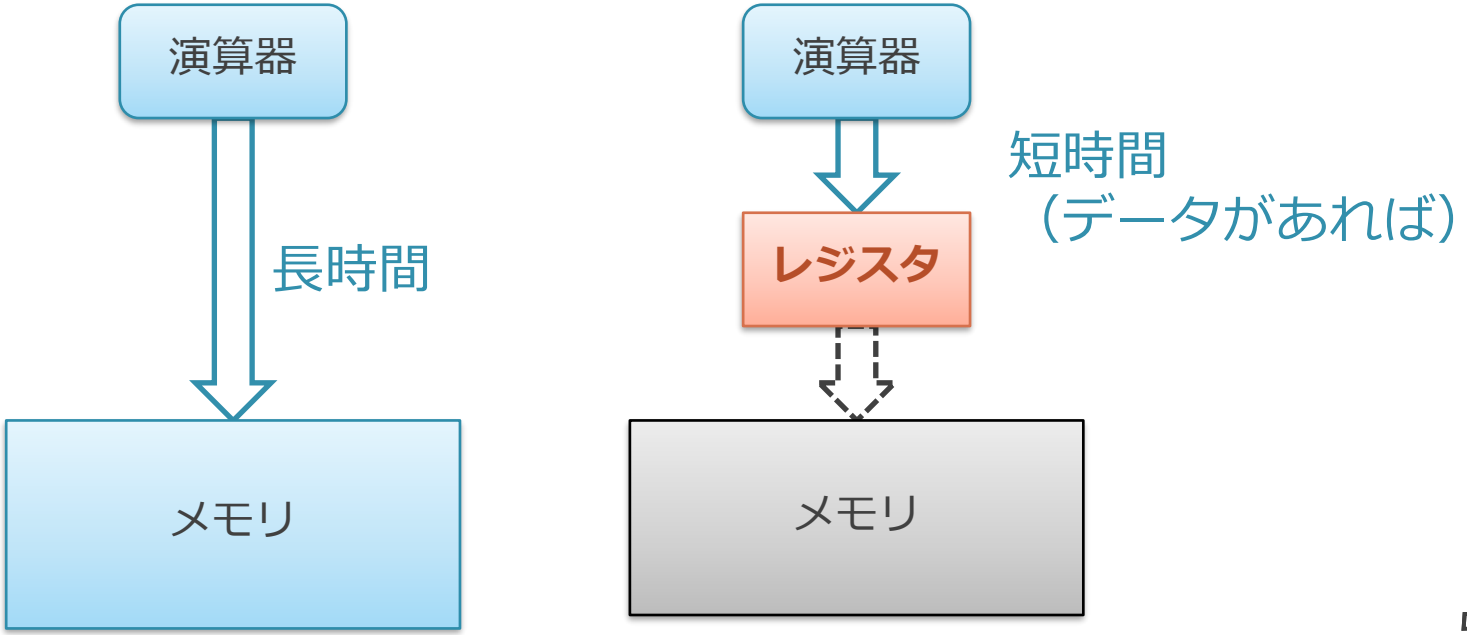


- 格子状に並んだセルの外周部の長さがアクセス時間を決めるから
  - ◇ メモリ容量 = (外周部の長さ)^2
  - ◇ アクセス時間は，容量の 0.5 乗でしか伸びない

なぜレジスタやキャッシュがあるのか？

- 小容量だけど，高速なレジスタやキャッシュを用意
  - ◇ 一度利用した値を入れておくことで，2度目からは高速に

- **大容量かつ高速なレジスタややキャッシュは作れない**
  - ◇ 記憶容量に応じたアクセス時間が必要だから
  - ◇ 最終的には信号が伝わる長さに帰着する



メモリ

1. メモリの基本
    1. 構造
    2. 動作
    3. アクセス時間

2. **メモリの詳細**
    1. SRAM と DRAM
    2. なぜメモリが存在するのか？
    3. マルチポート・メモリ
    4. メモリを対象にしたアタック

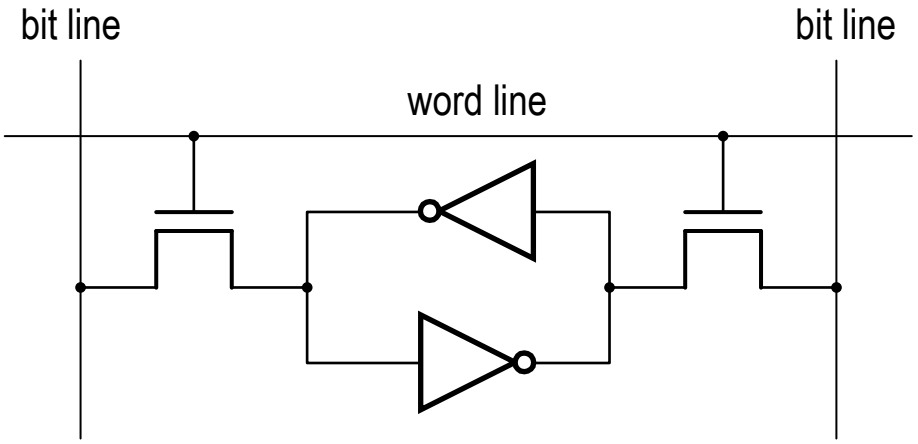
メモリのバリエーション

- メモリは基本的にはみな同じ構造を取る
  - ◇ セルの中身が方式ごとに異なる
- SRAM と DRAM を紹介

Static Random Access Memory (SRAM)

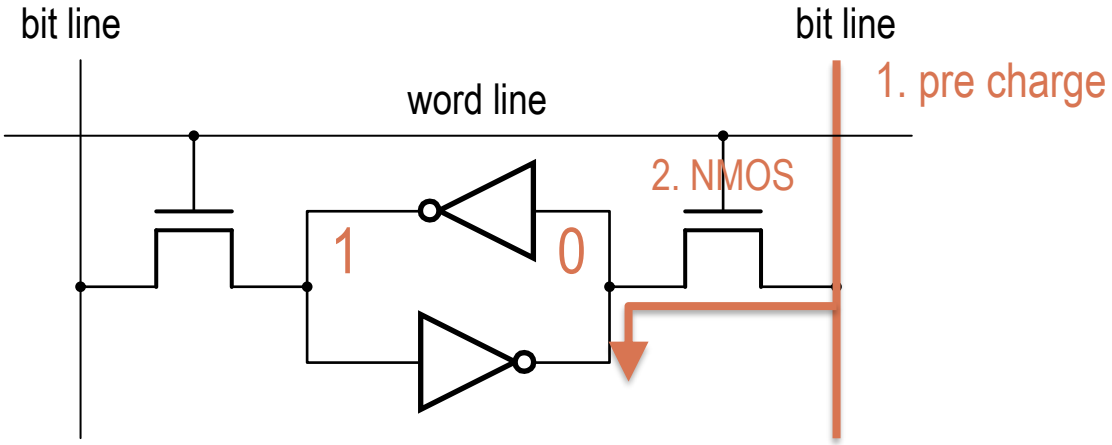
- Static とは：
  - ◇ 動作が静的（何もしなければ電流をほとんど消費しない）
  - ◇ 電源を入れている限りは内容が消えない
- 一般的にはメモリの中で最も高速
  - ◇ CPU 内の論理回路と同じトランジスタを使って作るから
  - ◇ レジスタやキャッシュ，各種テーブルは SRAM で作られている

SRAM のセル（1bit）



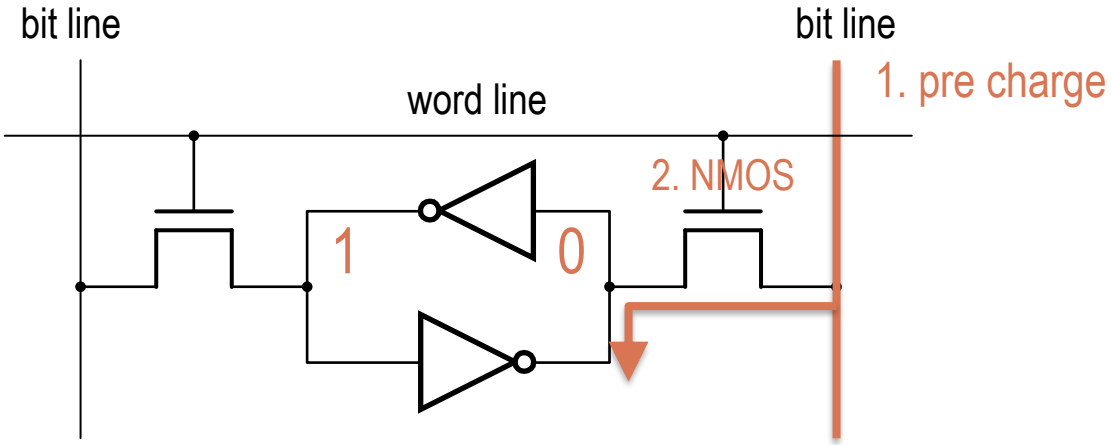
- インバータのループにより，1 bit を記録
  - ◇ ループの左右が [1:0] あるいは [0:1] の 2 つの定常状態を持つ

SRAM の読み出し



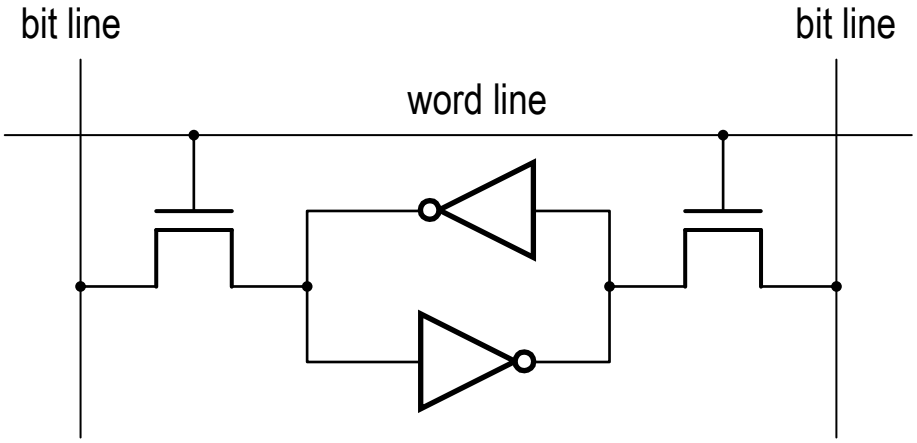
1. ビットラインのプリチャージ
   - ビットラインをあらかじめ高電位（１）にチャージする
2. ワードラインをアサート
   - ループの右にある NMOS が ON に
   - ループとビットラインが接続される
3. ビットラインのディスチャージ
   - ループの右が 0 なら，ビットラインが 0 に
   - ループの右が 1 ならそのまま

なぜプリチャージが必要なのか？



- 単にループとビットラインが接続されるだけではダメ
  - ◇ NMOS が高電位をうまく伝えられないから
    - □ もしループの右側が 1 で NMOS が ON になっても，ビットラインを 1 に引き上げることはできない
  - ◇ PMOS を追加すればできるが，なるべく小さく作りたいから嫌だ
- NMOS を介してビットラインの電位を下げることだけができる
  - ◇ = あらかじめ電位を上げておいて，下がったかどうかで判定

SRAM の書き込み



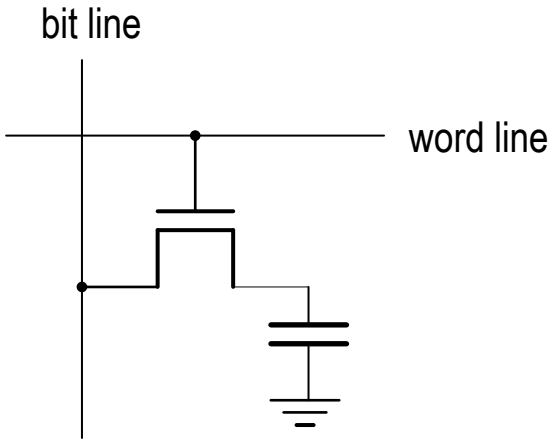
- 書き込み手順
  1. ループ左右のどちらか 0 にしたい方のビットラインの電位を下げる
  2. ワードラインをアサートして NMOS を ON に
  3. インバータの状態を強制的にビットライン側から低電位にする

- NMOS が低電位しか通せないので，書き込みには 2 本いる

Dynamic Random Access Memory (DRAM)

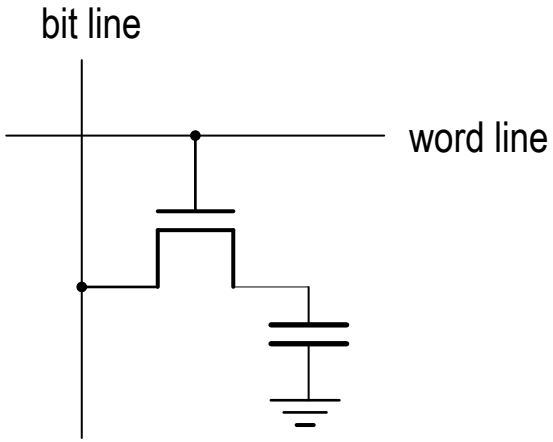
- 速度は遅いが，セルを小さく作れるので容量が稼げる
  - ◇ メイン・メモリは DRAM で出来ているのが普通

DRAM のセル



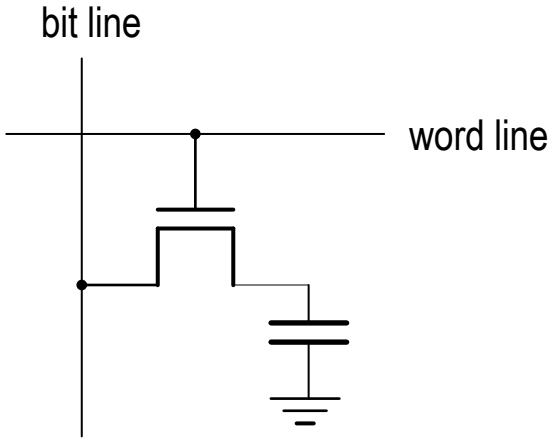
- セルをコンデンサによって構成
  - ◇ 電荷がたまっていれば１，そうでなければ０
  - ◇ 時間が経つと自然に電荷が抜けてデータが消える
- 電荷を維持するため，定期的にリフレッシュと呼ばれる操作を行う
  - ◇ 具体的には，一回読んで同じデータを書き戻す

DRAM の読み出し



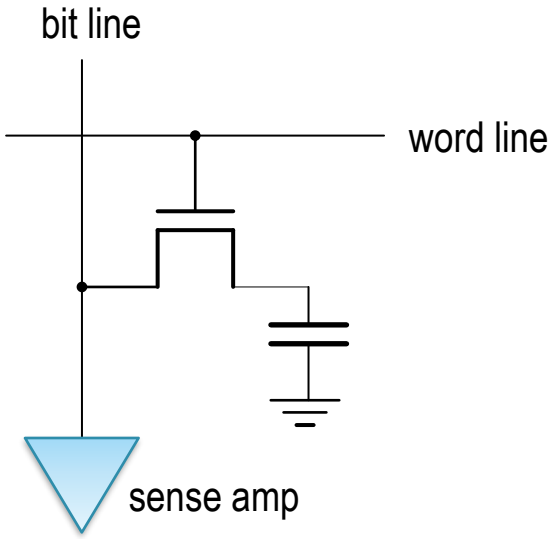
- 読み出しは，基本的には SRAM と同じ
  - ◇ ビットラインをプリチャージ
  - ◇ ワードラインをアサート
  - ◇ セルの状態に応じてディスチャージが起きる

DRAM の書き込み



- 書き込みは，NMOS でも 0.5 までは上げられるのでそれで行う（多分）
  - ◇ SRAM ではインバータをひっくり返す必要があるので，0.5 ではダメだった
- DRAM では 低電位（0）or 中間電位（0.5）を記録する
  - ◇ 読み出し時のプリチャージの電圧を 0.5 にして，下がったかどうかで判定

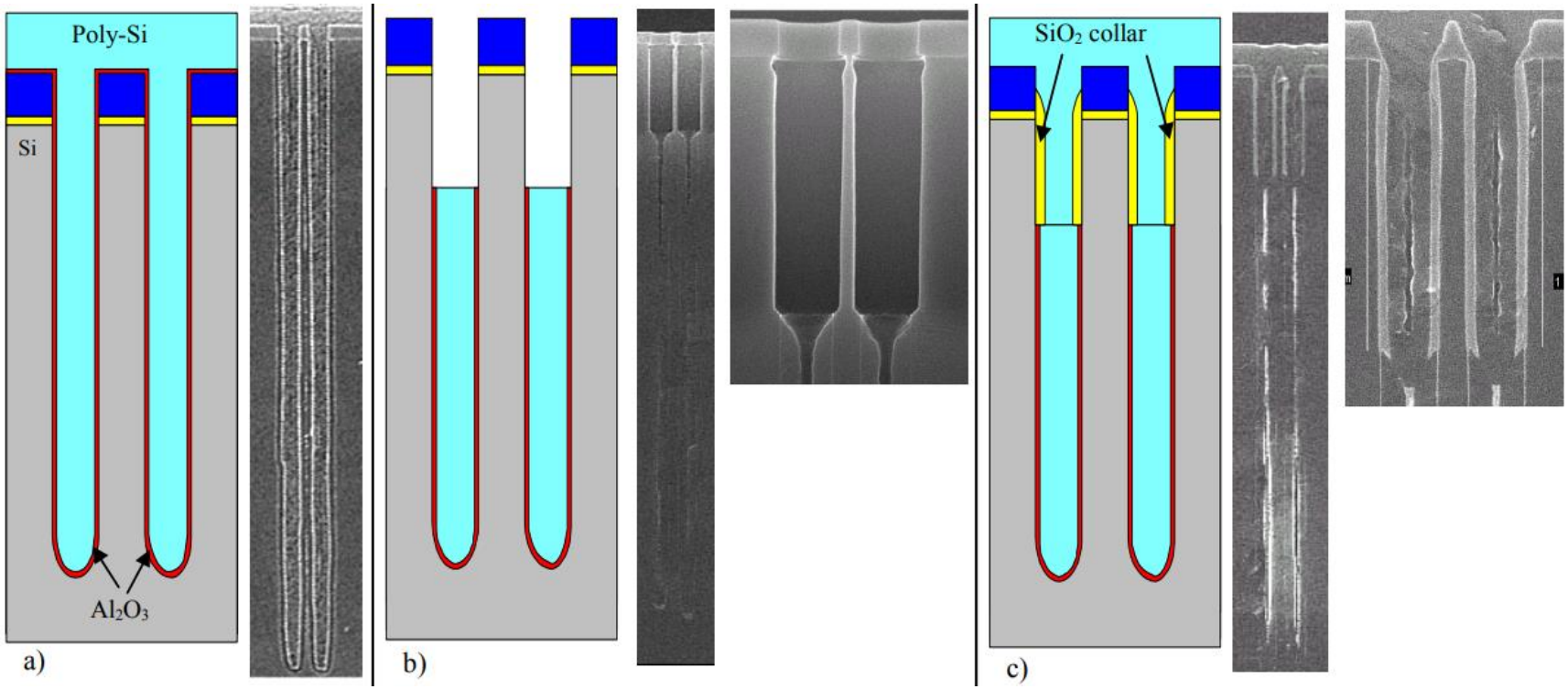
SRAM の場合との違い



- 読み出し時に内容が破壊される
  - ◇ ビットラインとコンデンサが接続されると，電荷が流れ出す
- コンデンサがビットラインをディスチャージする速度は遅い
  - ◇ センスアンプと呼ばれる微小な電圧の変化を増幅する回路を使う

DRAM のコンデンサの作り方

H. Seidl et al, A Fully Integrated Al2O3 Trench Capacitor DRAM for Sub-100nm Technology より



- チップの表から裏に向けて深い穴を掘って表面積を稼ぐ
  - ◇ コンデンサの容量は表面積に比例
  - ◇ トレンチ（塹壕の意味）と呼ばれる
  - ◇ 特殊な工程が必要なので通常のトランジスタと混ぜて作るのが難しい

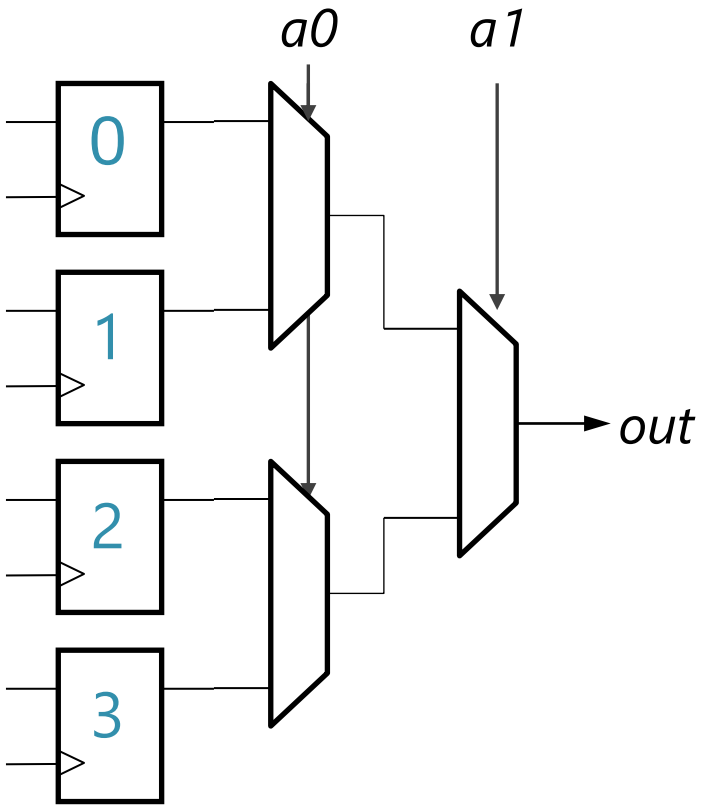
メモリ

1. メモリの基本
    1. 構造
    2. 動作
    3. アクセス時間
2. メモリの詳細
    1. SRAM と DRAM
    2. **なぜメモリが存在するのか？**
    3. マルチポート・メモリ
    4. メモリを対象にしたアタック

メモリの存在する理由

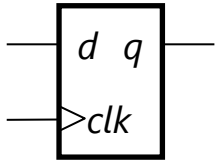
- D-FF とマルチプレクサがあれば，等価な機能は実現できる
  - ◇ 実際には，メモリ専用の回路が用意される
- メモリ専用の回路が用意される理由：
  - ◇ D-FF とマルチプレクサよりも，高密度に実装したいから
  - ◇ 同じチップ面積で，より多くの情報を記憶したい

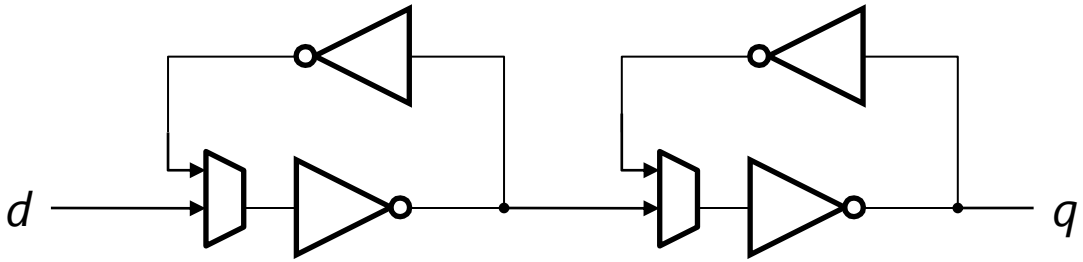
D-FF とマルチプレクサによる 4bit の RAM

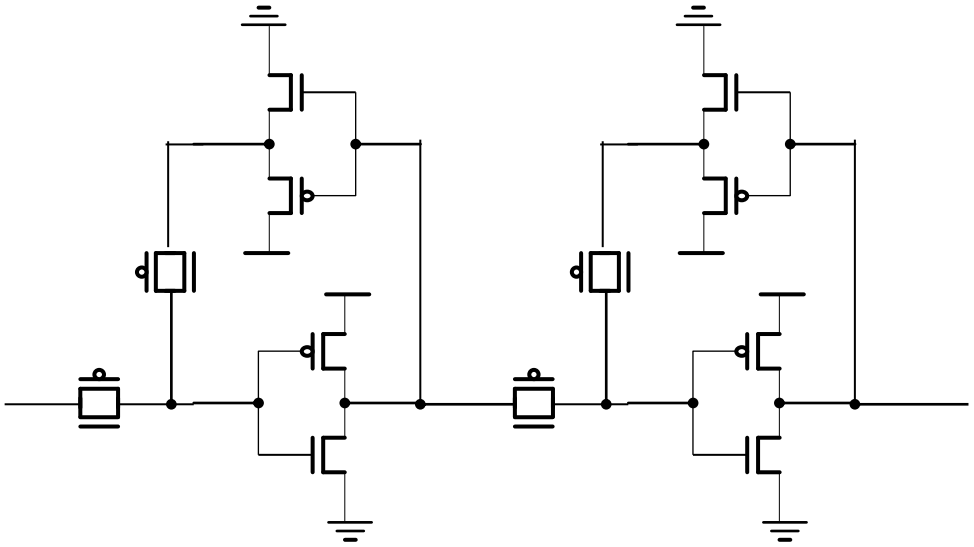


- ４エントリの LUT を D-FF で構成してみる
  - ◇ 中身を憶える 4つの D-FF
  - ◇ 場所を指定して選択する2段のマルチプレクサ

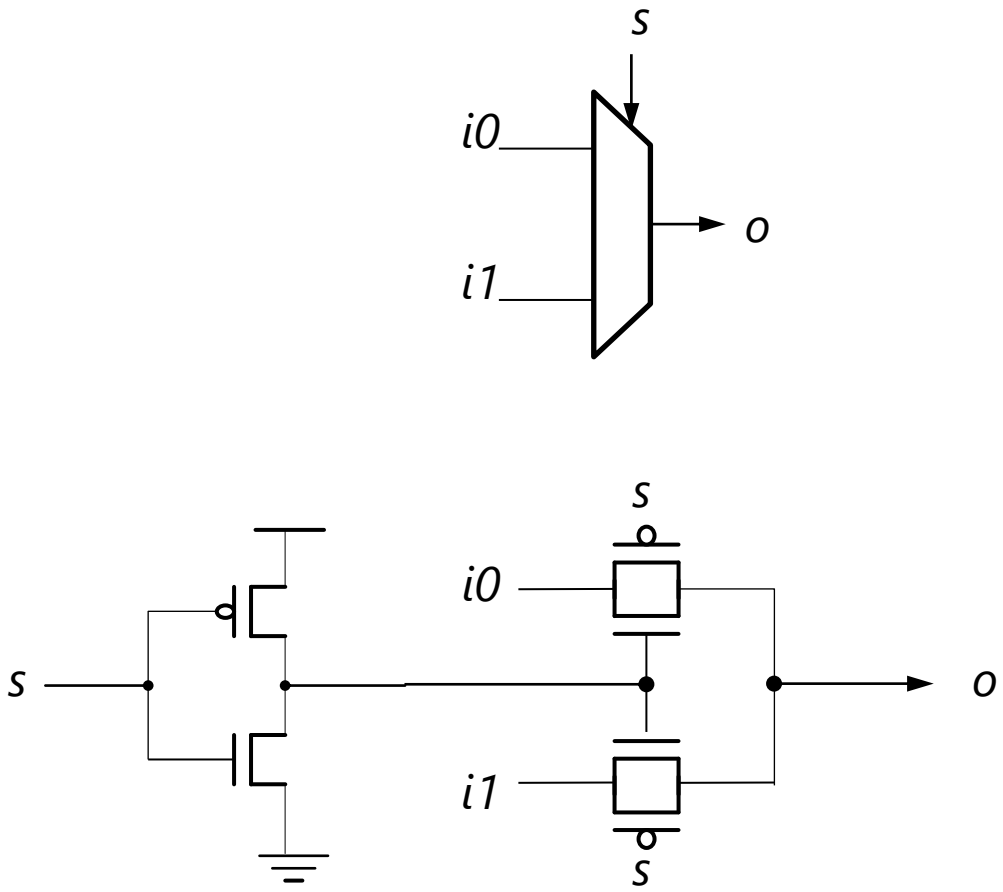
D-FF：トランジスタ 16個





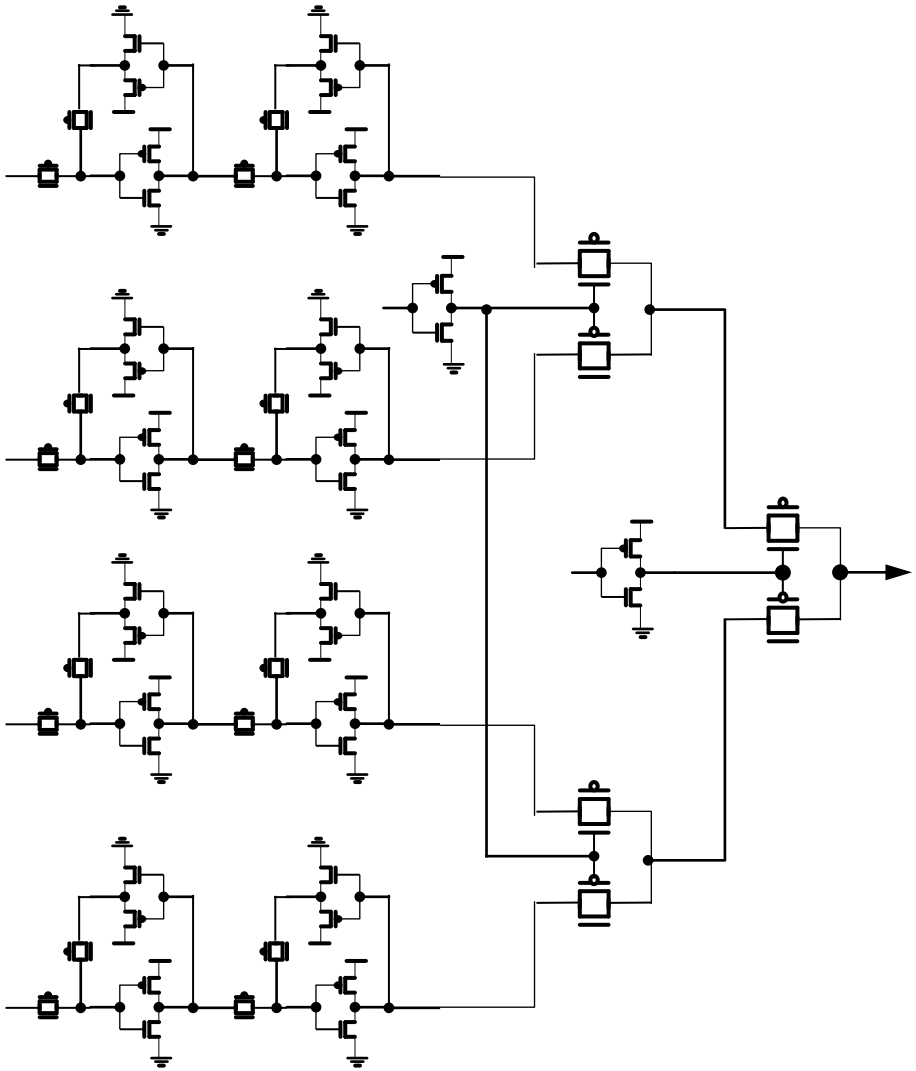


マルチプレクサ：トランジスタ 6個



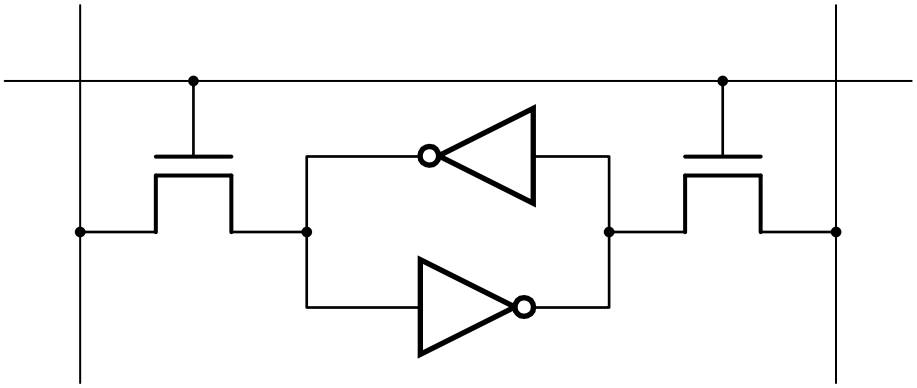
4bit メモリに必要なトランジスタ数：80

16*4+6*3=82

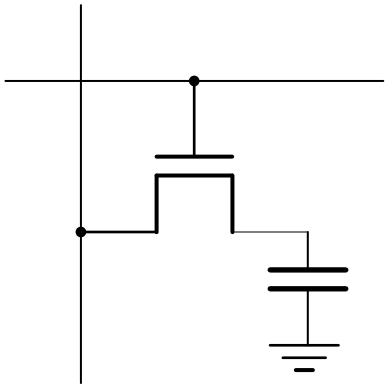


SRAM と DRAM のセル（1bit）

SRAM：2 + 2*2 = 6



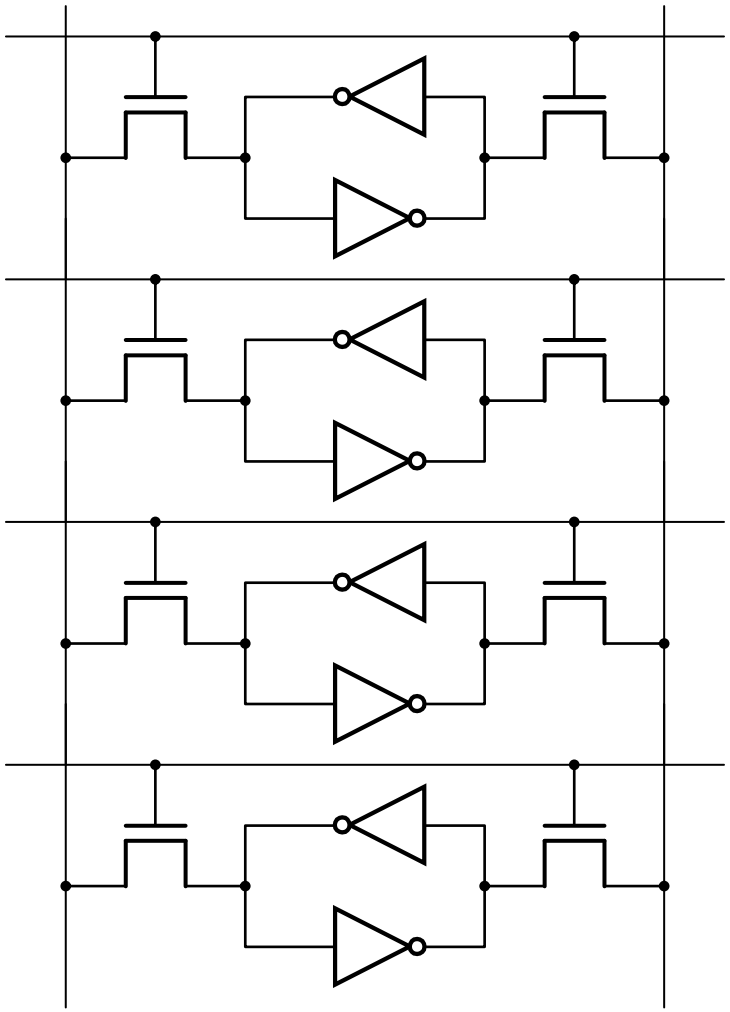
DRAM：1 +コンデンサ 1



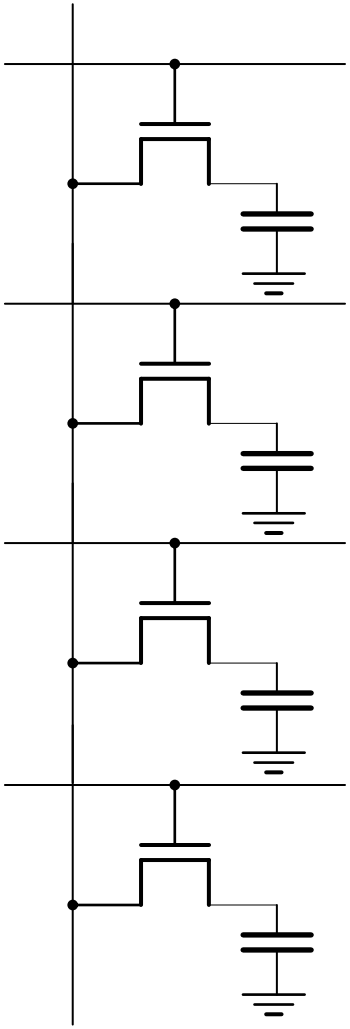
◇ D-FF よりも圧倒的に単純

SRAM と DRAM（4bit）

SRAM：6*4 = 24



DRAM：4 +コンデンサ 4



メモリ

1. メモリの基本
    1. 構造
    2. 動作
    3. アクセス時間
2. メモリの詳細
    1. SRAM と DRAM
    2. なぜメモリが存在するのか？
    3. **マルチポート・メモリ**
    4. メモリを対象にしたアタック

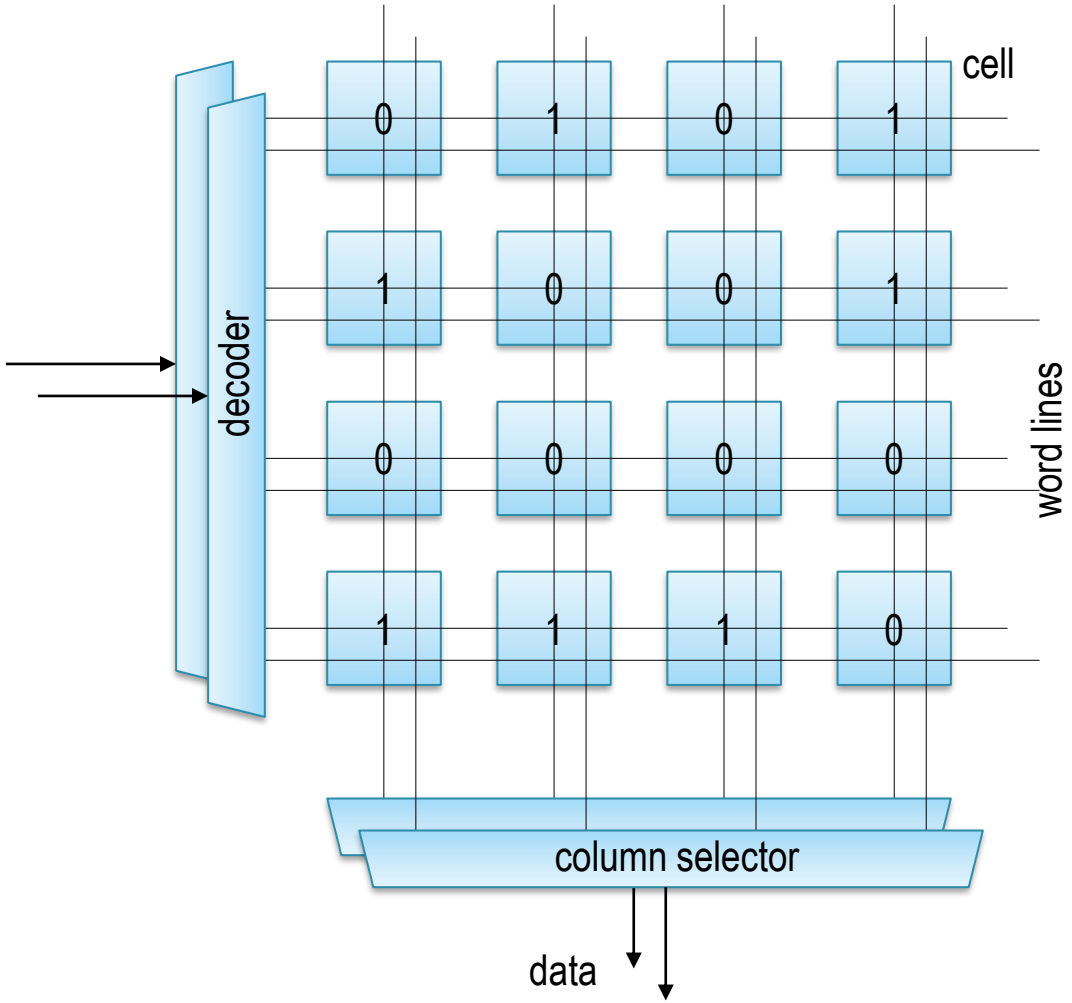
マルチポート・メモリ

- ランダムな複数の箇所を同時にアクセスできるメモリ
  - レジスタ・ファイルなどを作るときに必要
    - ２つのソースの読み出しと，ディスティネーションの書き込み
  - 同時にクセスできる数 = ポート数
- CPU に特有に良く必要になる
  - 一般的な回路ではあまり出てこない
- ポート数の２乗に比例して面積が大きくなる
  - 同時にアクセスできる数の２乗に比例する

マルチポート・メモリの回路面積

- ポート数の２乗に比例して面積が大きくなる
  - ◇ 同時にアクセスできる数の２乗に比例する
- レジスタ・ファイルなどは，同一記憶容量あたりの回路面積が大きい

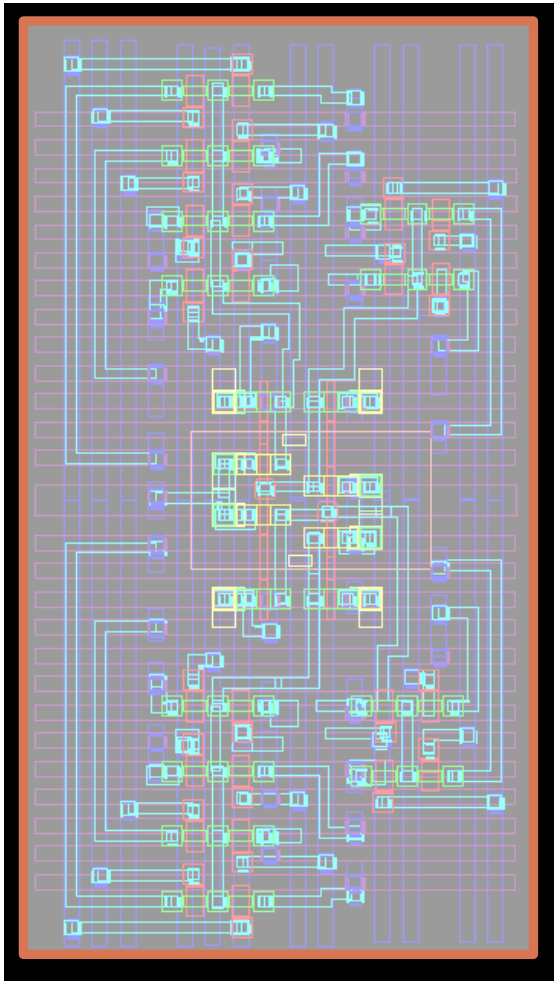
ポート数の２乗に比例して面積が大きくなる



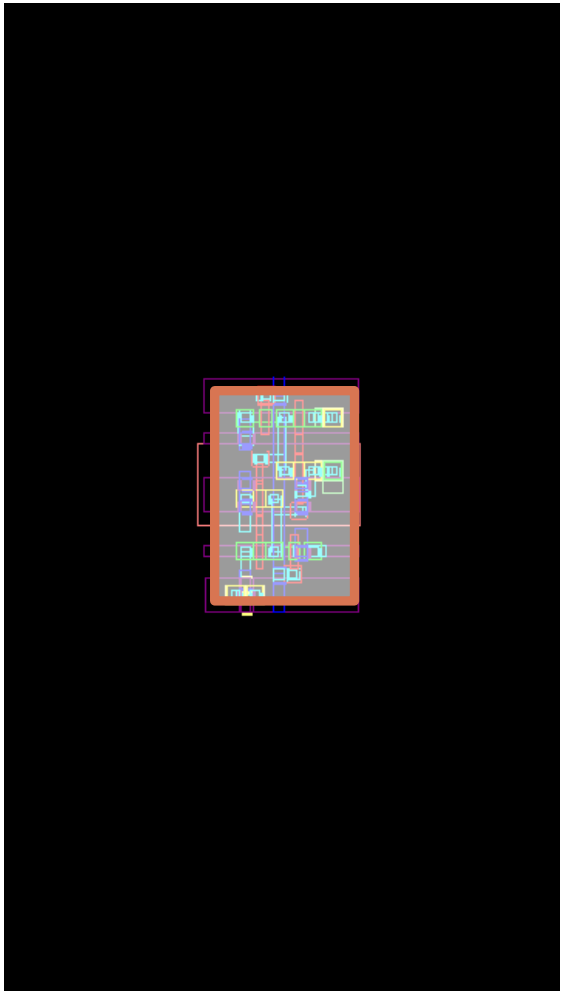
- 複数の異なる行をアクセス
  - ◇ ワードラインが複数必要
  - ◇ ビットラインも複数必要
- 縦と横がそれぞれポート数分増える
  - ◇ ポート数が増えると配線の面積が支配的に

SRAM（1bit）のレイアウト

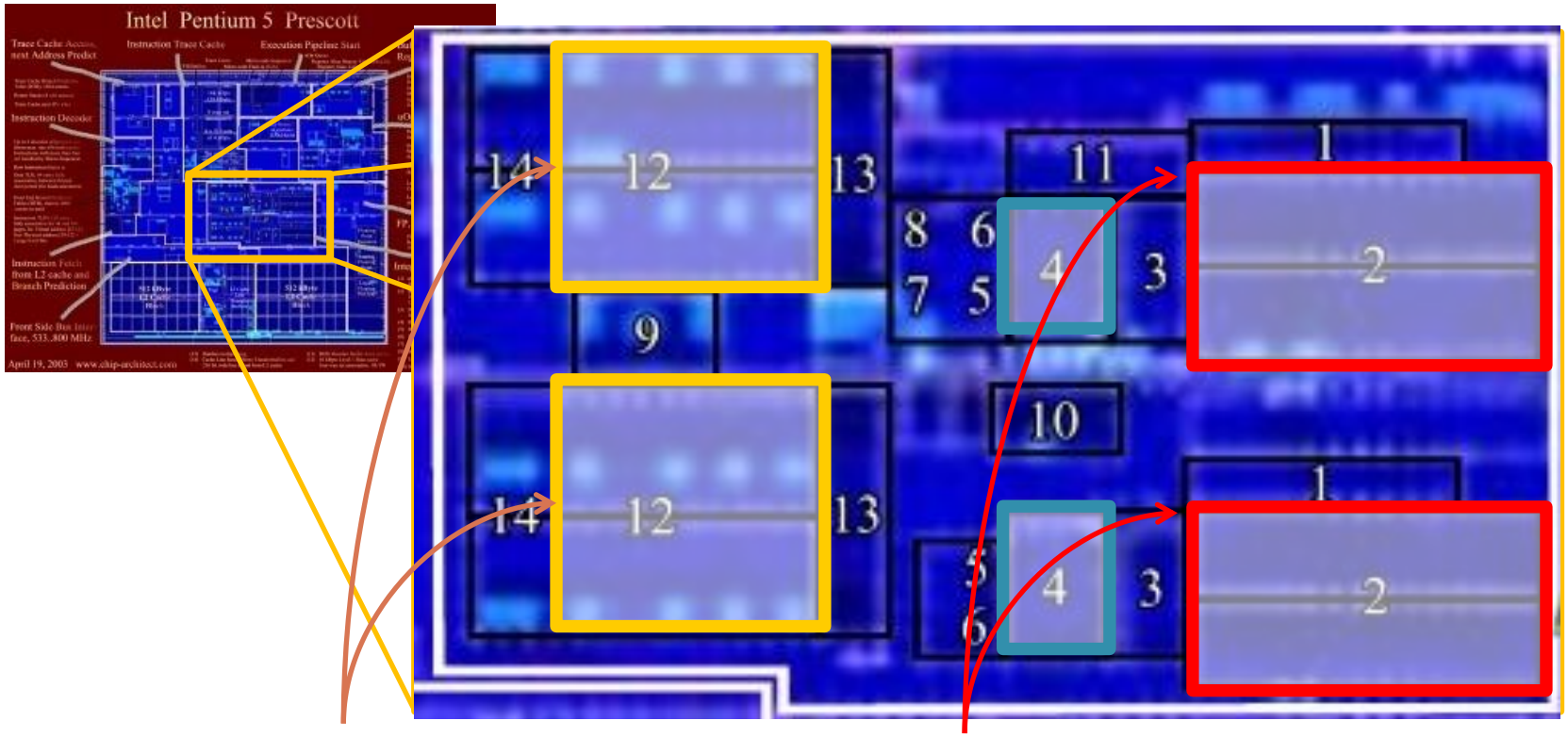
12 ports



2 ports



Intel Pentium 4 の整数演算ユニット



L1 Data cache

16KB 1 read/1 write

（from http://www.chip-architect.com）

Register file

64 bits 144 entries 6 read/3 write
(double pumped)

マルチポート・メモリと CPU の回路規模

- CPU の性能を上げようとすると，ポート数が増える事が多い
  - 例：２命令を同時に処理できるようにする
    - レジスタ・ファイルの読み出しと書き込みが倍に
    - ポート数が倍 → 回路面積４倍

マルチポート・メモリと CPU の回路規模

- CPU はそう言ったメモリのかたまり
  - ◇ 演算器よりもメモリの方がずっと大きい
  - ◇ 投機実行や並列実行をするために様々なテーブルが導入される
  - ◇ それらは同時アクセス可能な数に対して，急速に大きくなる
- 同時実行数を増やすことが難しい
  - ◇ 完全に独立したメモリなら問題ない
  - ◇ １つの大きなコアを作るより，お互いが独立しているマルチコア化の方が面積効率は良い

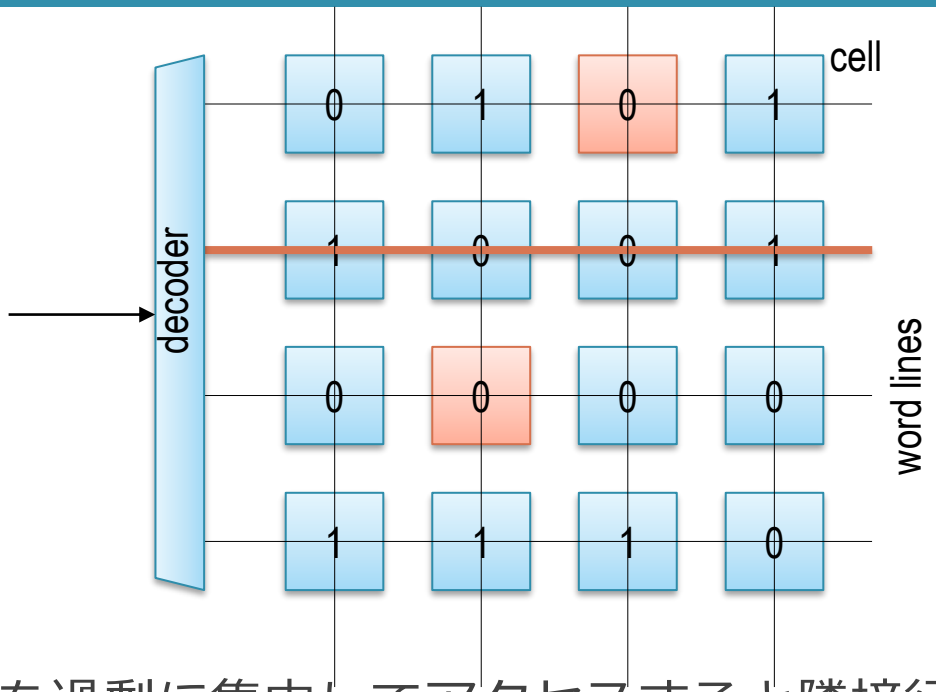
メモリ

1. メモリの基本
    1. 構造
    2. 動作
    3. アクセス時間
2. メモリの詳細
    1. SRAM と DRAM
    2. なぜメモリが存在するのか？
    3. マルチポート・メモリ
    4. メモリを対象にしたアタック

メモリを対象にしたアタック

- メモリの仕組みがわかると，原理がわかる
  1. Row Hammer
  2. Cold Boot Attack

Row Hammer



- 特定の行を過剰に集中してアクセスすると隣接行のデータが化ける
  - ◇ 通常よりも DRAM セルの放電が早まるため
  - ◇ 意図的にデータを化けさせて様々なアタックを行う
    - □ 特権昇格とかは実証コードがでている・・・らしい

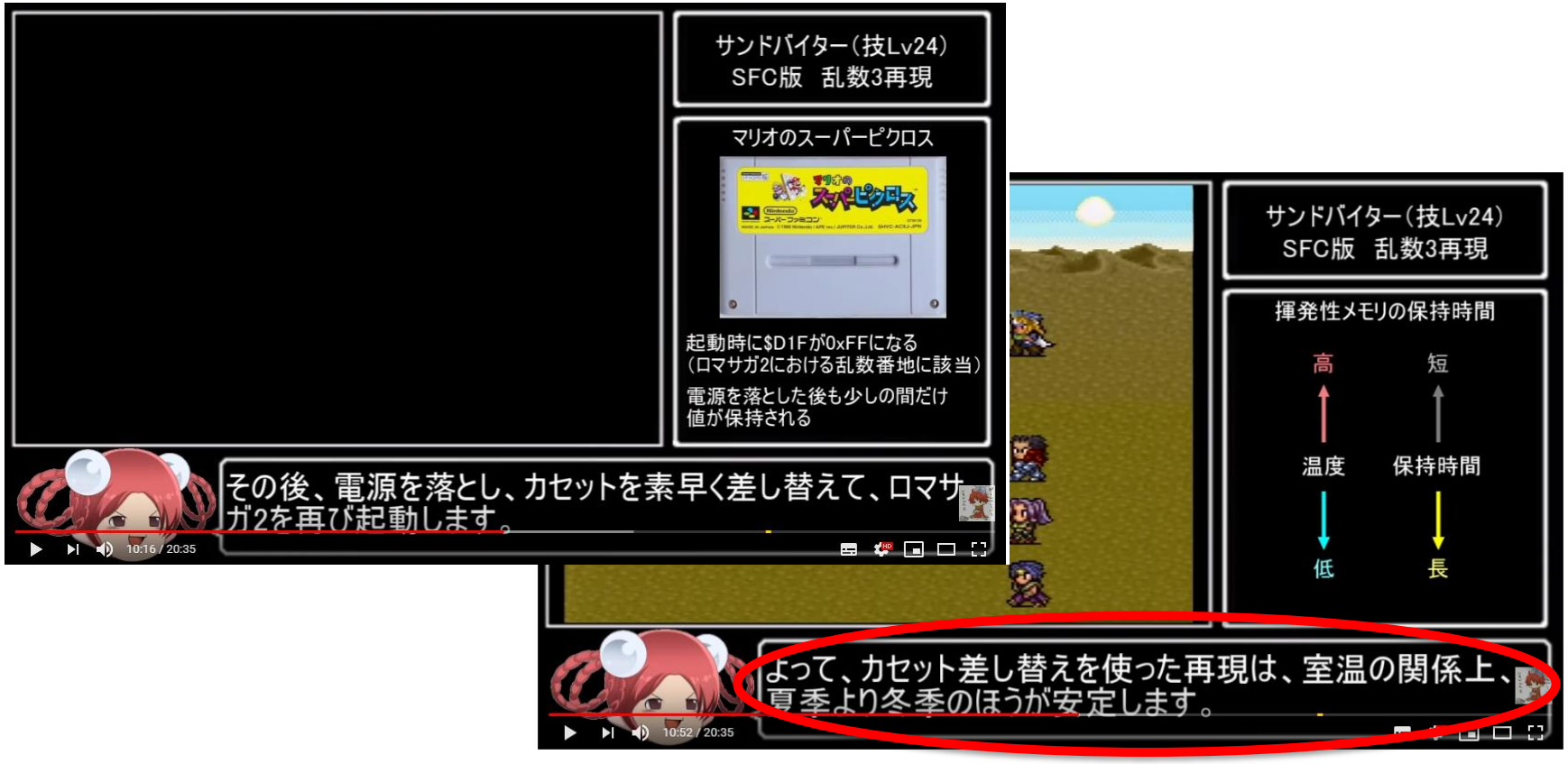
- Yoongu Kim et al., Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors など

Cold Boot Attack

- 稼働中のマシンの DRAM を物理的に引っこ抜いて，
  他のマシンにさして読み取る
  - ◇ DRAM セルの電荷の放電は，温度が低いほどゆっくりになる
  - ◇ メモリを極低温に冷やすことで，攻撃成功率が飛躍的に上がる
  - ◇ -50 度まで冷やすと，1時間ぐらい持つ・・・らしい
- J. A. Halderman et al., Lest we remember: Cold-boot attacks on encryption keys など

Cold Boot Attack の応用例

- 【ロマサガ2】ほぼ全ての技を100%閃く方法 (おやつの人，2019)
  - https://www.youtube.com/watch?v=gWfcZnAydQ4



ロマサガ２の技のひらめき判定（イメージ）
このゲームでは戦闘中に条件に応じて一定の確率で技をひらめく

```
// 乱数値で初期化されたテーブル
uint8_t RAND_TABLE[256] = …;
…
// 乱数テーブルのインデクス
// 初期化されていないので，
// 直前にさしたカセットで書かれていた値が使われる
uint8_t rand_index;
uint8_t rand() {
    return RAND_TABLE[rand_index++];
}

// 技のひらめき境界値より小さい乱数を引けば技をひらめく
bool waza_hirameki() {
    return rand() < waza_threshold() ? true : false;
}
```

今日のまとめ

- 分岐予測の続き
- メモリの基本と詳細

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

感想とか質問の回答

感想とか質問の回答

- CPUにはタグを使う部分が複数ありますね、ArmもMTEというものを打ち出しており、これはOSにも何らかの変化をもたらすかもしれません。 既存のCPUの性能を十分に引き出すために、OSは何をする必要があるのでしょうか？

感想とか質問の回答

- ヘネパタのコンピュータの設計以外でおすすめの参考書はありますか？ 尚、ある程度広く学びたいと考えています。

感想とか質問の回答

- hash perceptron, メモリアクセスがキャッシュに乗っているか否かの予測に使っているのを読んだことがあって面白いなと思っていました

感想とか質問の回答

- グローバル履歴長を長くすると精度が上がるという点がいまいち分からなくて、例として示されていた"関数呼び出しの前後でパターンが維持される"はローカル履歴で十分なのではないかと思った。

感想とか質問の回答

- 履歴の話がまだ完全には理解できていないです.

感想とか質問の回答

- nビットカウンタ予測器ってnをあまりに大きくしすぎても意味がなさそうなのは直感的にわかりますが，2ビットが一番良さそうなのは面白いなと思いました.

感想とか質問の回答

- 予測器ってどうやって実装されているのでしょうか？

感想とか質問の回答

- パイプラインの段数が少ないときにも分岐予測は大きな効果を発揮するのでしょうか？
- 72ページにある予測器の精度について、30年で0.6%->0.3%の向上のマシン全体の性能の向上への寄与が知りたいです。減少割合で言えば大きいですが、減少量は0.3%と小さいことのどちらが影響するのかが知りたいです。

感想とか質問の回答

- 予測器はハードウェア資源を多く使い時間をかけるほど正確な予測ができると思います．予測器に使われるリソースも予測の正確さの向上とともに増えていますか？ 72ページの図を横に見ていったときに予測器が使うハードウェア・時間的リソースは大きくなっているのか，あるいは既に頭打ちになっているでしょうか？

感想とか質問の回答

- 講義の中で、Nvidiaの用語が少し授業のものとずれているという話があったが、CPUでもIPとPCなどずれているものがあるので、統一してほしいなと感じた（慣れればなんてことないですが）。

感想とか質問の回答

- 企業で行われている分岐予測の研究結果が論文などの形で外部に公開されることは多いのでしょうか？（競争力のためにあえて公開しない場合もあるのではないかと思いました。）

感想とか質問の回答

- 分岐予測器を作成するにあたって、昔と今とではCPUで動くワークロードが変わってきているみたいなことを少し話されていた気がするんですが、具体的にどんな傾向の変化があるのかが気になります

感想とか質問の回答

- 今回の講義の内容とはあまり関わりがないのですが、自動車など工業製品は部品の数が増えれば増えるほど壊れやすいとされているのに、CPUはトランジスタ数の増加に対して故障しやすさが上がっていないような印象があり、なぜか気になります。

感想とか質問の回答

- 静的分岐についても予測が必要なのは、パイプライン上で命令が連続していないために生じるストールを回避するためで、命令をインライン化すれば予測は必要ないという認識はあってますか？

感想とか質問の回答

- 履歴エントリの履歴長を伸ばすほど、適当な値でテーブルが埋まるまで分岐予測が当たらない期間が伸びますし、相関がないエントリについても(出現しうる)すべてのエントリが埋まるまで予測が当たらないので直感的には性能は悪くなりそうなものですが、どうしてそうならないのでしょうか。

感想とか質問の回答

- SPECなどのベンチマークで計測した分岐予測の精度が項目毎に異なっているけど、授業で紹介したMPKIの図はどうやって計測したのですか