

コンピュータ アーキテクチャ I 第9回

塩谷 亮太 (shioya@ci.i.u-tokyo.ac.jp)

東京大学大学院情報理工学系研究科 創造情報学専攻

課題の解説

考え方の指針

1. まず理想的にプログラムが実行された場合のサイクル数を考える
2. 何かイヤなイベント（ハザード）が起きる回数を計算する
 - 全体の命令数，命令の種類毎の出現率，イベントの発生確率などをかけていけば良い
3. イベントによる総延長時間を求める
 - イベントの発生回数と，イベント 1 回あたり何サイクル実行時間が伸びるかをかけて求める
4. 1. の理想的な実行サイクル数に総延長時間を加えて実際の実行サイクル数を求める
5. 実行命令数を 4. で求めたサイクル数で割る

分岐予測ミスによる実行サイクルの増加のモデル

■ 以下のようにおいた場合,

- ミスがない理想的な実行の際のサイクル数 : C_t
- 分岐予測ミスの発生回数 : $N_m = N_i \times P_b \times P_m$
 - 実行命令数 : N_i
 - プログラム中の分岐命令の出現率 : P_b
 - 分岐命令毎の予測ミス発生率 : P_m
- 分岐予測ミス・ペナルティ : C_p

■ 実行サイクル数 C_r は, 理想サイクル数 C_t に対して,

- $C_r = C_t + N_m \times C_p$
- 意味 : 予測ミスの発生回数×ペナルティ だけ実行時間が伸びる

具体的な値を入れてみる

■ 以下のように置いた場合,

- 理想的な実行の際のサイクル数 : $Ct = 1M$ (メガ)
- 分岐予測ミスの発生回数 : $Nm = Ni \times Pb \times Pm = 0.025M$
 - 実行命令数 : $Ni = 1M$
 - 分岐命令の出現率 : $Pb = 0.25$
(プログラムは大体 4 命令に 1 つぐらい分岐が出てくる)
 - 分岐命令毎の予測ミス発生率 : $Pm = 0.1$
- 分岐予測ミス・ペナルティ : $Cp = 4$

■ 実行サイクル数 Cr は

- $Cr = Ct + Nm \times Cp = 1M + 0.1M = 1.1M$
- 分岐予測ミスにより 10% 実行サイクル数が伸びている

IPC で考えると

- 最終的な性能を考える上で IPC の方が都合がよい

- 実行サイクル数 C_r を命令数 N_i で正規化すると,

- $$\frac{C_r}{N_i} = \frac{C_t}{N_i} + \frac{(N_m \times C_p)}{N_i} = \frac{C_t}{N_i} + \frac{(N_i \times P_b \times P_m \times C_p)}{N_i} = \frac{C_t}{N_i} + P_b \times P_m \times C_p$$

- IPC は命令数を実行サイクル数で割ったもの = つまり上記の逆数

- $$IPC_r = \frac{1}{\frac{C_t}{N_i} + P_b \times P_m \times C_p} = \frac{1}{\frac{1}{IPC_t} + P_b \times P_m \times C_p}$$

- ここで IPC_r は実際の IPC, IPC_t は理想 IPC

一般化できる

- 以下のようにおいた場合,
 - 理想的な実行の際のサイクル数 : C_t
 - 何らかのハザードの発生回数 : $N_h = N_i \times P_i \times P_h$
 - 実行命令数 : N_i
 - ハザードを起こす命令の出現率 : P_i
 - その命令毎のハザード発生率 : P_h
 - ハザード時のサイクル数の増加 : C_p
- 実行サイクル数 C_r は, 理想サイクル数 C_t に対して,
 - $C_r = C_t + N_h \times C_p$

一般化できる

- ハザード a, ハザード b, ハザード c ... とあった時
 - それぞれおきた回数が $Nh..$, ペナルティが $Cp ...$ とする
- 実行サイクル数 Cr は, 理想サイクル数 Ct に対して,
 - $Cr = Ct + Nha \times Cpa + Nhb \times Cpb + Nhc \times Cpc + \dots$
- 実行命令数を Ni とすると, $IPCr = Ni/Cr$

課題 8

■ 以下のような条件を考える

- 10段のパイプラインを持つ 2-way スーパースカラプロセッサであり, 理想的には $IPC=2$ で実行できる
- 全実行命令におけるなんらかのデータハザードの発生率は 0.2
- このデータハザード発生時は 1 サイクル実行時間が伸びるものとする
- 全実行命令における分岐命令の出現率は 0.2
- 分岐予測ミス率は 0.3

- ## ■ (1) この CPU を改良する際,
- 「3-way スーパースカラにする」
 - 「2-way のまま15段パイプラインにする」
 - 「2-way のまま分岐予測器を改良する」

のどれが最も性能が上がるかを性能を計算して検討せよ

- この CPU を 3-way スーパースカラにすると理想的には $IPC=3$ で実行できるがデータハザードの発生率は 0.3 に上昇するとする
- また, 分岐予測器を改良すると分岐予測ミス率が 0.2 にまで削減されるとする

課題 8 (1)

まず実行サイクル数を求めてから、正規化する

■ 以下のような条件を考える

- 10段のパイプラインを持つ 2-way スーパースカラプロセッサであり、理想的には IPC=2 で実行できる
- 全実行命令におけるなんらかのデータハザードの発生率は 0.2
- このデータハザード発生時は 1 サイクル実行時間が伸びるものとする
- 全実行命令における分岐命令の出現率は 0.2
- 分岐予測ミス率は 0.3

■ 上記のベースライン CPU の 実行サイクル数 C_r は、実行命令数を N_i とすると

- $C_r = C_t + N_{ma} \times C_{pa} + N_{mb} \times C_{pb}$
- $= \frac{N_i}{IPC_t} + (N_i \times 0.2) \times 1 + (N_i \times 0.2 \times 0.3) \times 9$
- $= \frac{N_i}{2} + N_i \times 0.2 + N_i \times 0.54 = 1.24 \times N_i$

■ IPC_r は N_i を C_r で正規化したものなので、

- $IPC_r = \frac{N_i}{C_r} = \frac{N_i}{1.24 \times N_i} \approx 0.81$

課題 8 (1)

- ベースライン CPU の動作周波数を 1 とすると、その性能は周波数 $1 \times \text{IPC } 0.81 = 0.81$
- 「3-way スーパスカラにする」
 - $$\text{IPC}r = \frac{1}{\frac{1}{3} + 0.3 \times 1 + 0.2 \times 0.3 \times 9} \approx \frac{1}{1.17} \approx 0.85$$
 - 先ほどは丁寧にサイクル数を 1 回計算したが、ここでは最初から「1 命令あたりのサイクル数」として考えて IPC をいきなり求めている（結果は同じ）
 - 周波数は変わらないなら性能も 0.85

課題 8 (1)

■ 「2-way のまま15段パイプラインにする」

- $IPC_r = \frac{1}{\frac{1}{2} + 0.2 \times 1 + 0.2 \times 0.3 \times 14} \approx \frac{1}{1.54} \approx 0.65$
 - 分岐予測ミスペナルティが 10 から 15 に
- 10 → 15 段になったことにより周波数が 1.5 倍に
- 性能は $1.5 \times 0.65 = 0.97$

■ 「2-way のまま分岐予測器を改良する」

- $IPC_r = \frac{1}{\frac{1}{2} + 0.2 \times 1 + 0.2 \times 0.2 \times 9} = \frac{1}{1.1} \approx 0.94$
- 周波数は変わらないので, 性能も 0.94

課題 8 (1)

- まとめると,
 - ベースライン : 0.81
 - 1. 3-way スーパスカラにする : 0.85
 - 2. 2-way のまま15段パイプラインにする : 0.97 最も速い
 - 3. 2-way のまま分岐予測器を改良する : 0.94

課題 8 (2)

- (2) ここで効率を表す「性能エネルギー比 (性能 / 消費エネルギー)」という指標を導入する。この数字が大きいほど小さなエネルギーで速く動くことを意味する。前述の3つの改良方針のうち、以下の追加の条件のもとで、最も性能エネルギー比が良いものはどれかを計算して検討せよ。
 - クロック周波数を N 倍にすると、消費エネルギーは N^2 倍増える
 - 3-way スーパースカラにすると消費エネルギーは 1.5 倍になる
 - 分岐予測器を改良すると消費エネルギーは 1.1 倍になる

課題 8 (2)

■ ベースラインの消費エネルギーを 1 とすると : $0.81/1=0.81$

1. 3-way スーパスカラにする : $0.85/1.5 \approx 0.57$

2. 2-way のまま15段パイプラインにする : $\frac{0.97}{1.5^2} \approx 0.43$

- 最も速いが, エネルギー効率がとても悪い

3. 2-way のまま分岐予測器を改良する : $\frac{0.94}{1.1} \approx 0.85$

- 2 より少し遅いがエネルギー効率はずっと良い

メモリと、キャッシュの基本

今日の内容

1. メモリ
2. キャッシュの基本

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

メモリ

データ： 07h 10h 30h ... 3Eh 01h 32h 20h 80h ...

アドレス： 0h 1h 2h ... 8000h 8001h 8002h 8003h 8004h ...

- メモリは命令列と、計算するデータを保持する
 - 単一の巨大な配列があると思えばよい
 - C 言語の配列は、これを切り出してユーザーに見せている
- 数字が入る箱がたくさん並んでいるイメージ
 - アドレス：箱の通し番号（住所）
 - データ ：箱の中身の数字

メモリ

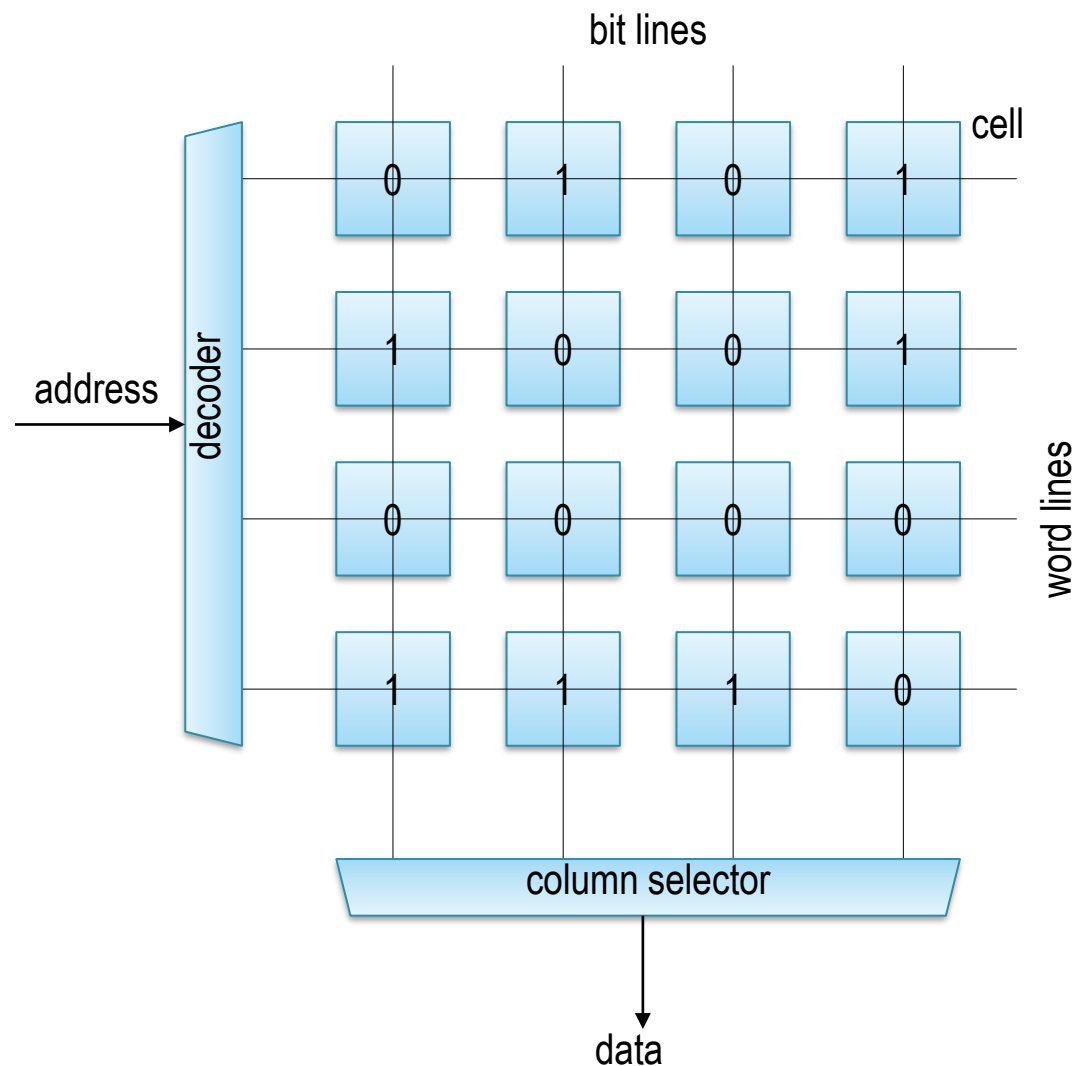
1. メモリの基本

1. 構造
2. 動作
3. 容量と速度

2. メモリの詳細

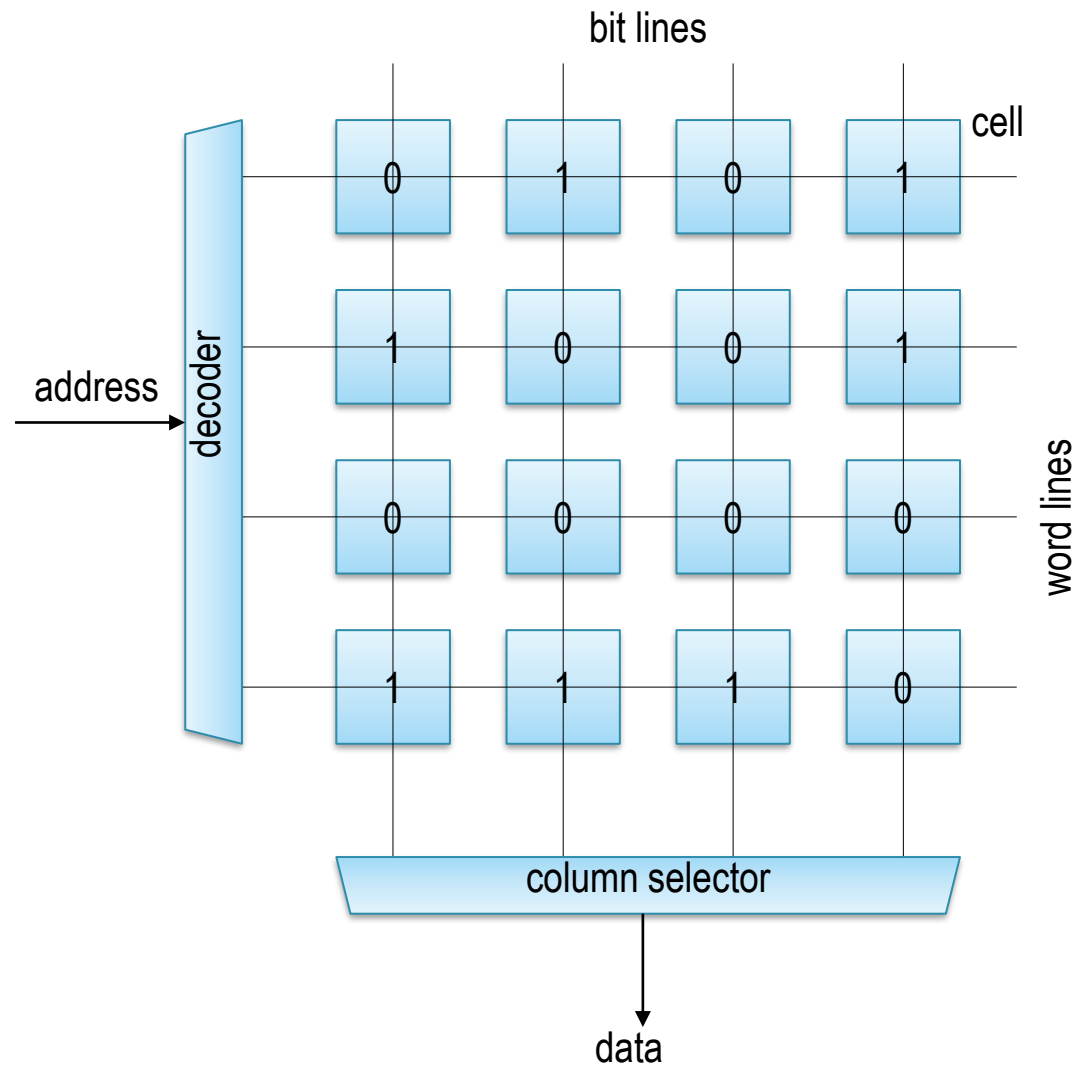
1. SRAM と DRAM
2. なぜメモリが存在するのか？

メモリの基本構造：セルを行列状に配置



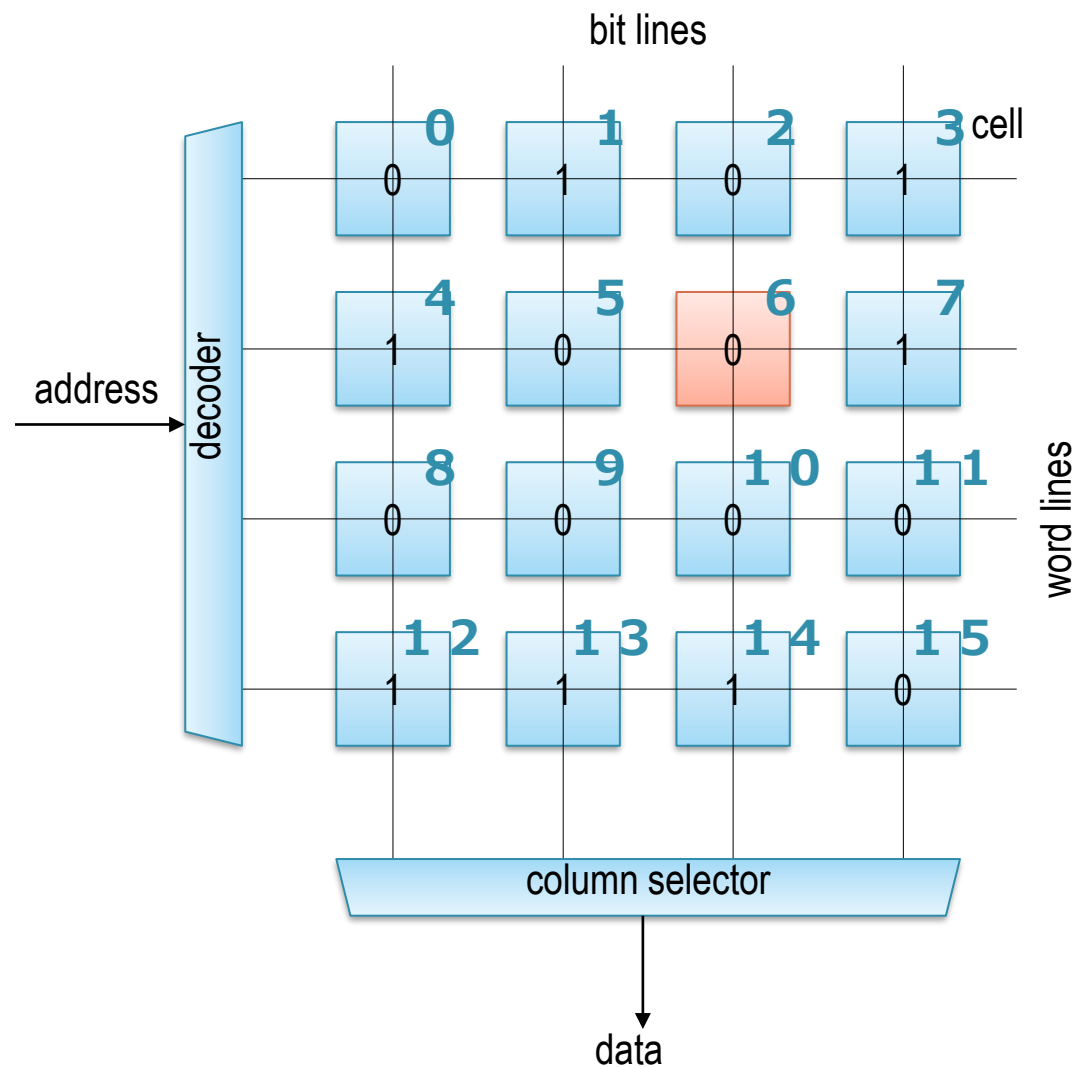
- セル：
 - 1ビットの情報を記憶
- ビットライン：
 - 列方向の配線
- ワードライン：
 - 行方向の配線
- デコーダ：
 - アドレスをデコードしてワードラインをアサート
- カラム・セレクトタ：
 - ビットライン出力から1つを選んで出力

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 $16 = 2^4$

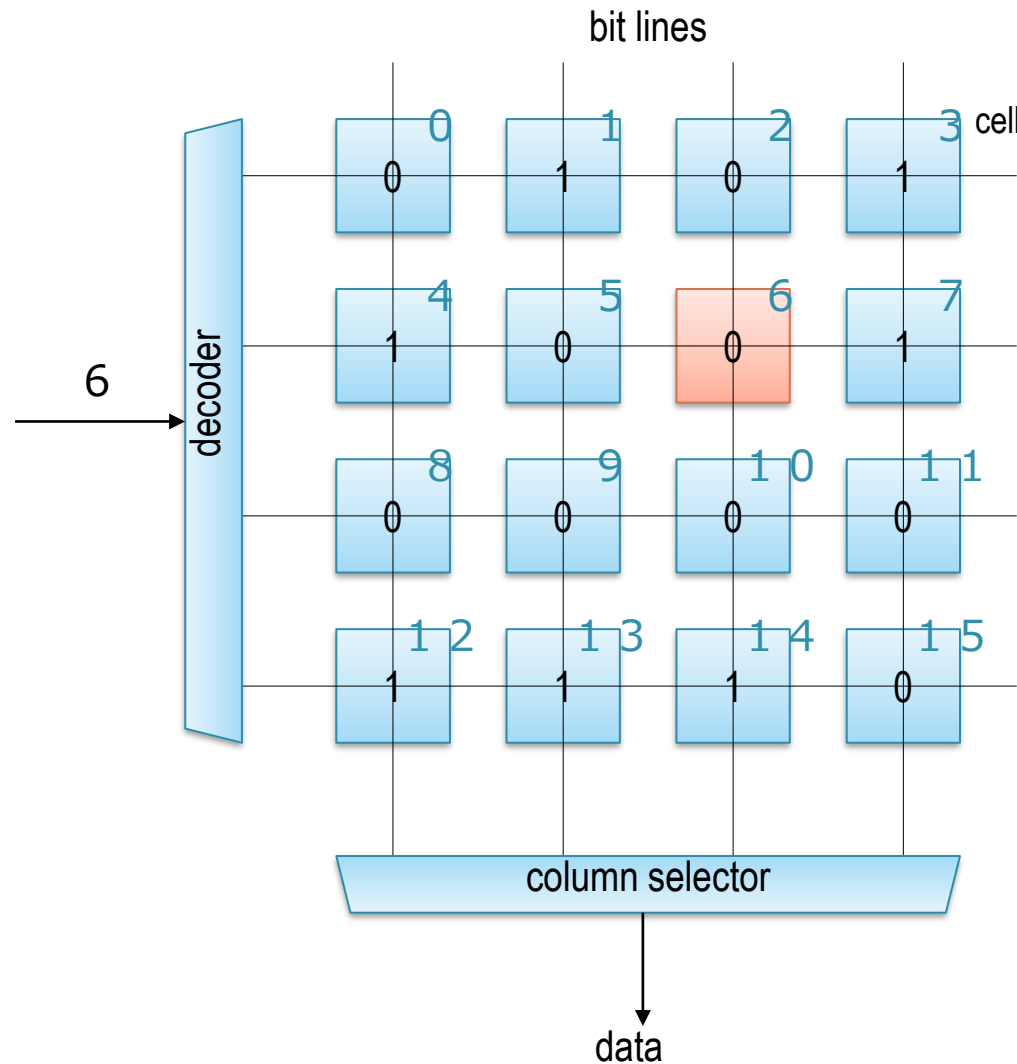
- アドレスは4ビット
- 各セルの右上の数字がアドレス

■ 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)

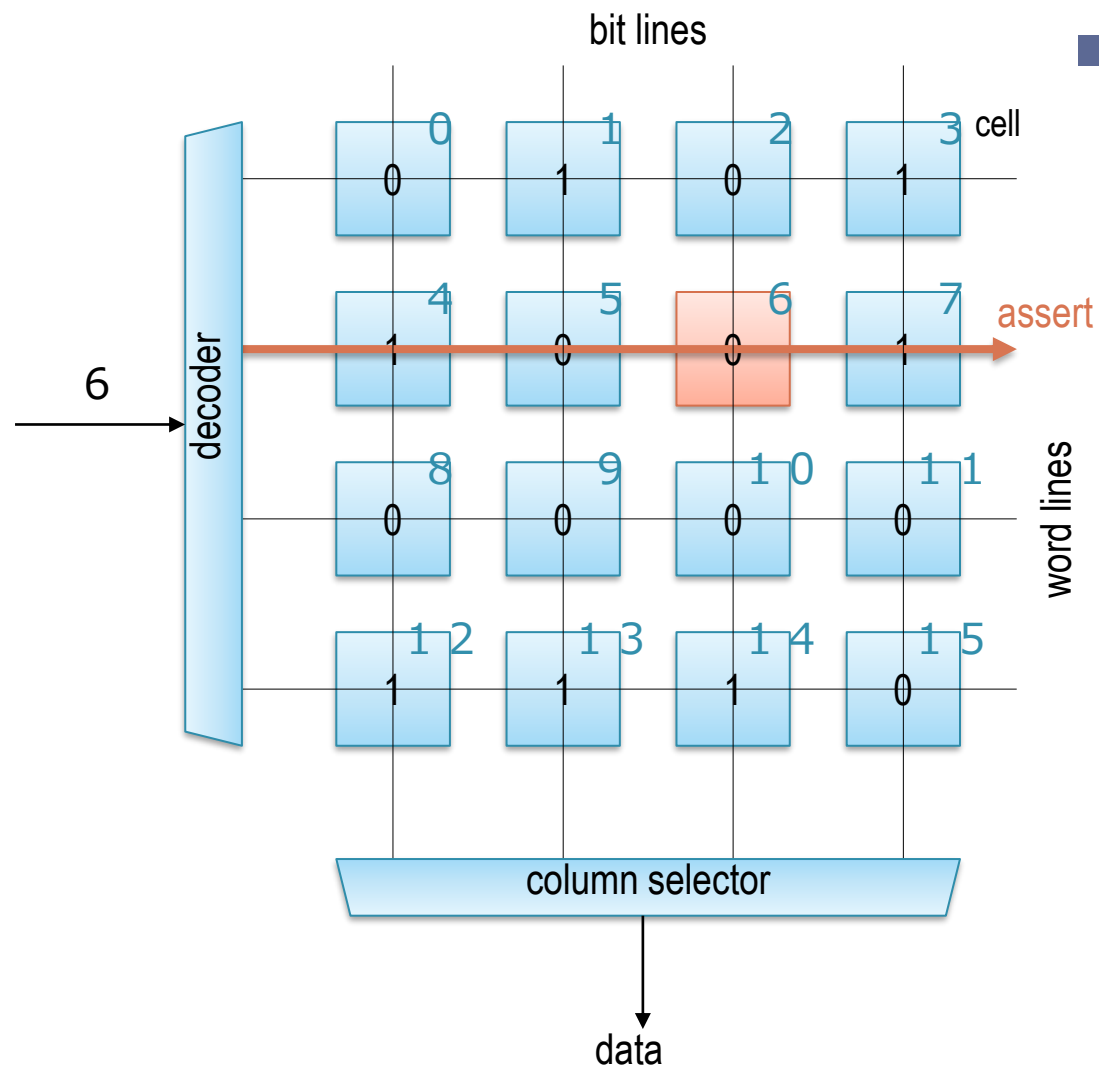
■ デコーダ

- 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)

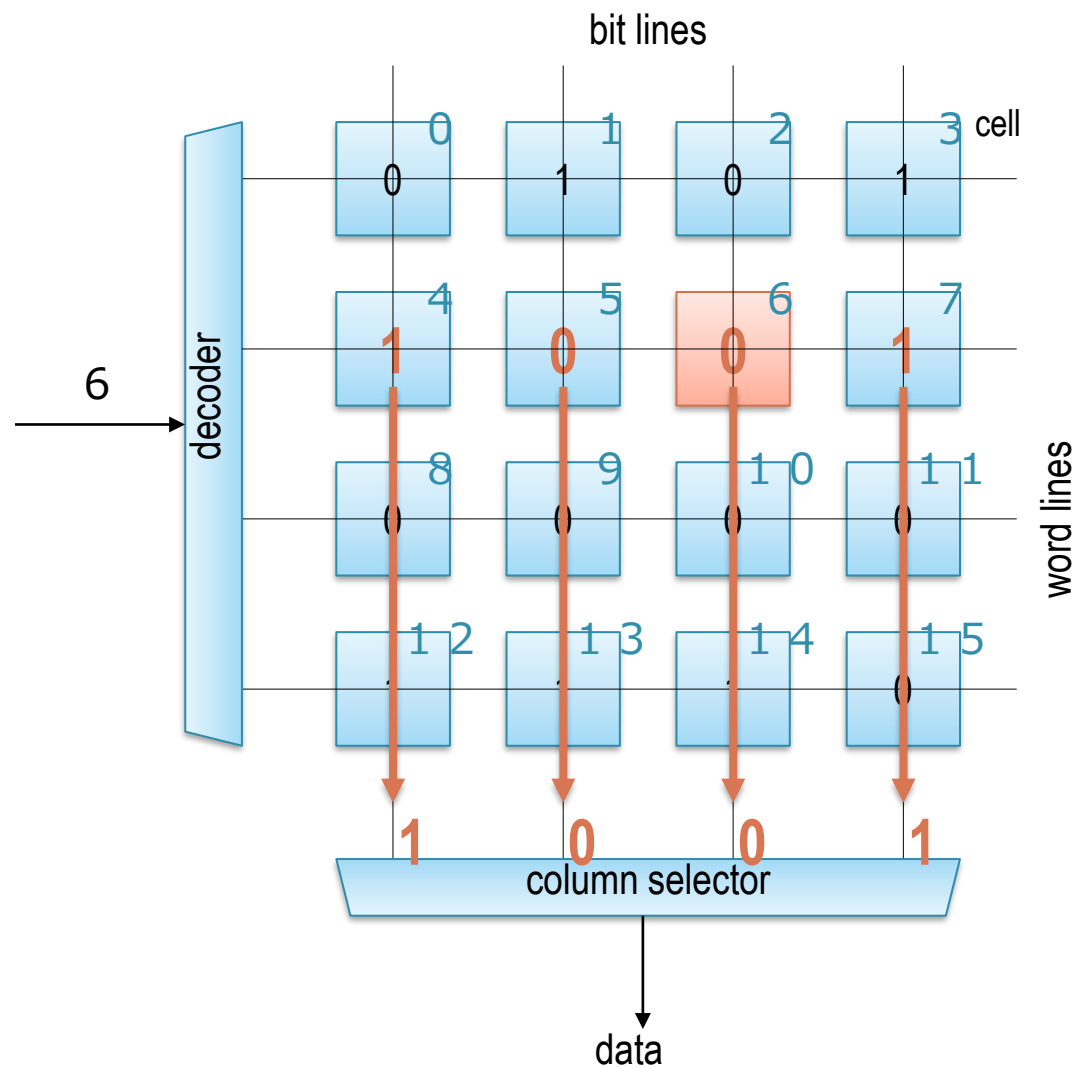
ワードラインのアサート

- 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)

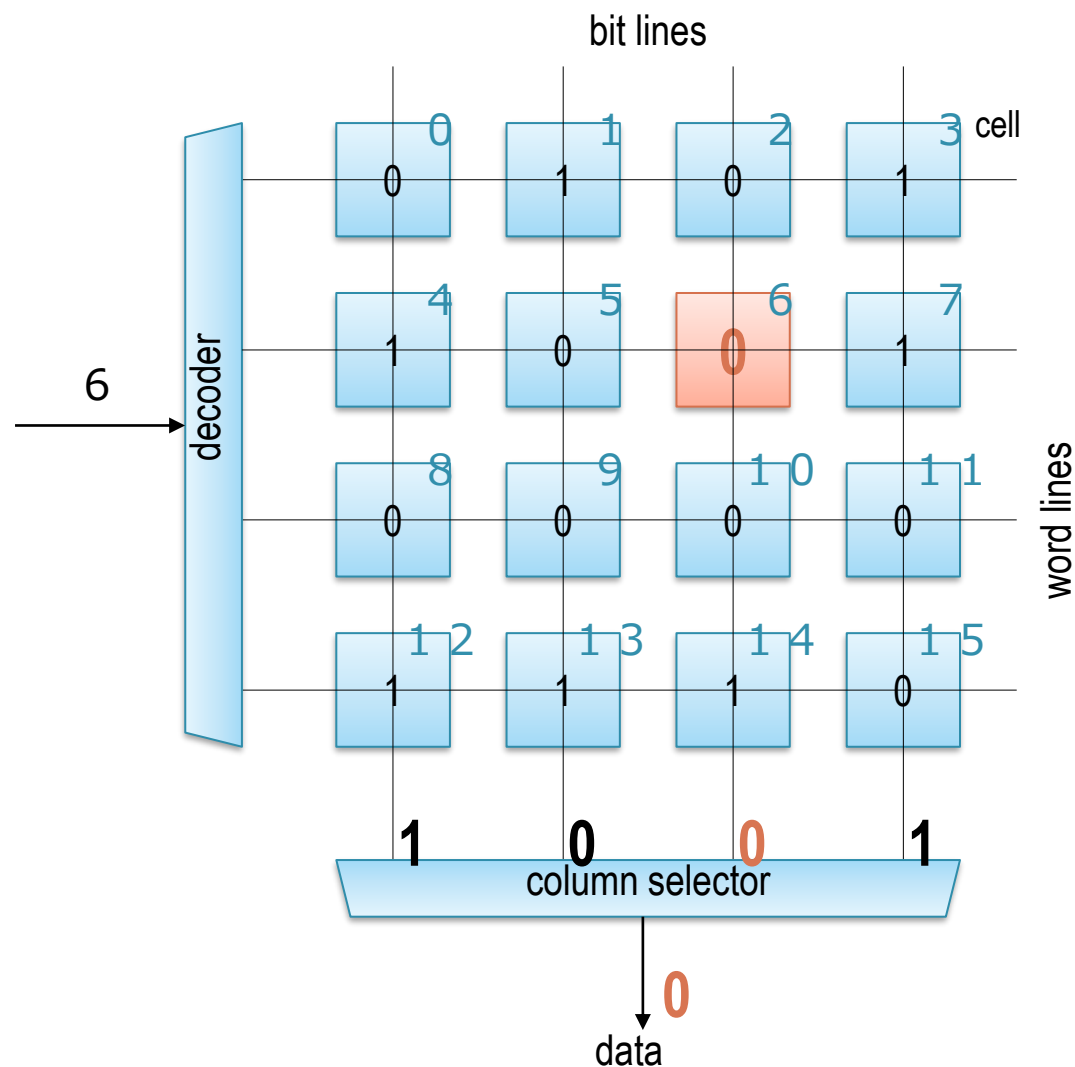
ワード (1行分のデータ) の読み出し



- 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)

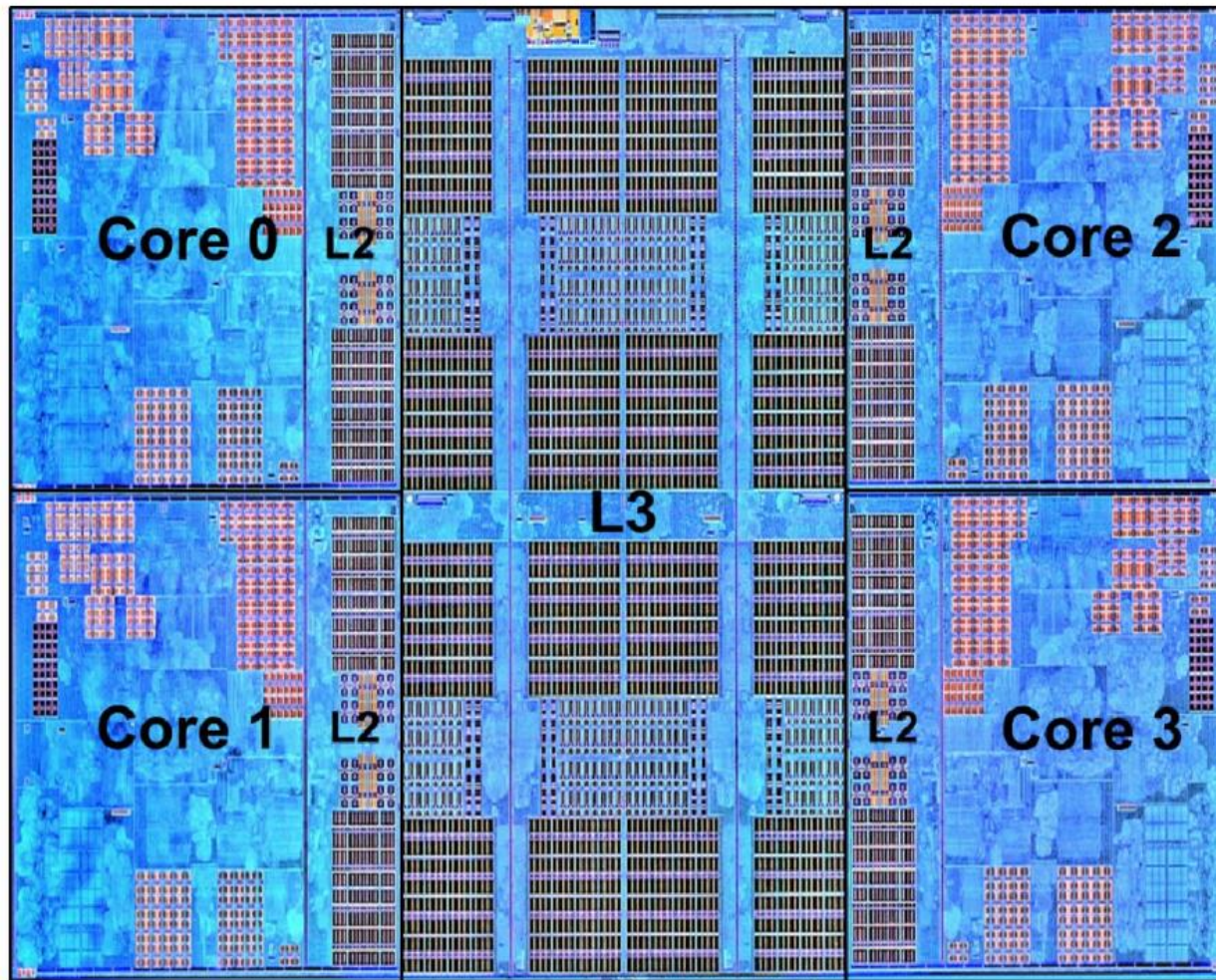
列の選択



- 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・セレクト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

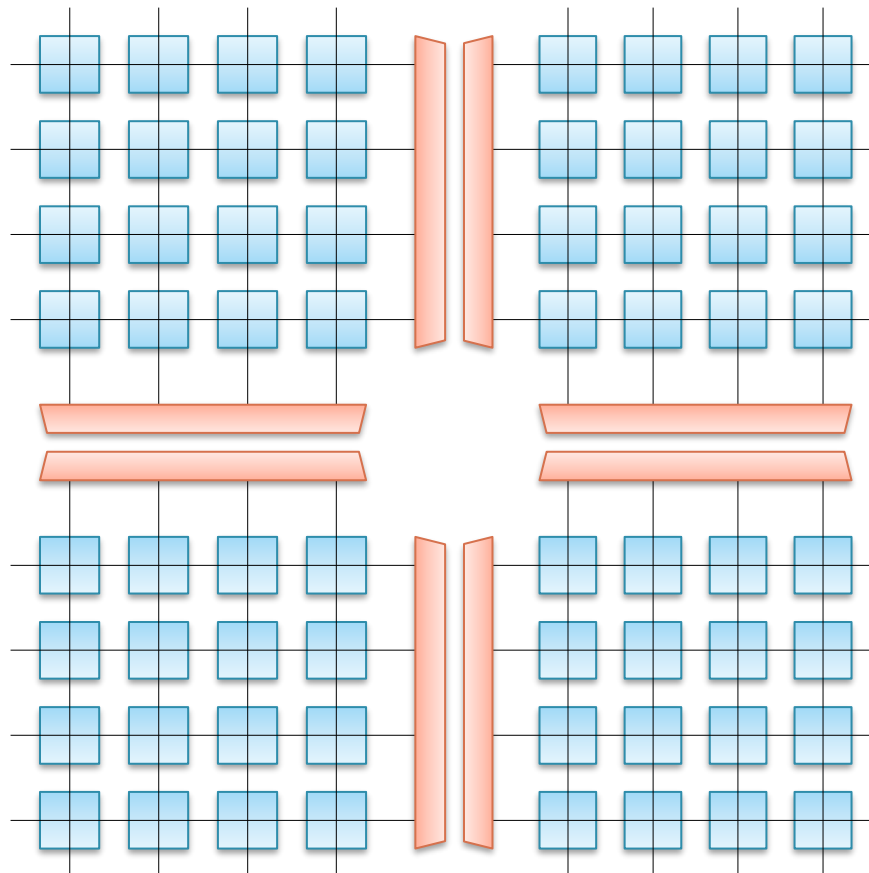
AMD Zen という CPU のチップ写真

Teja Singh et al., Zen: An Energy-Efficient High-Performance x86 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. 容量と速度**

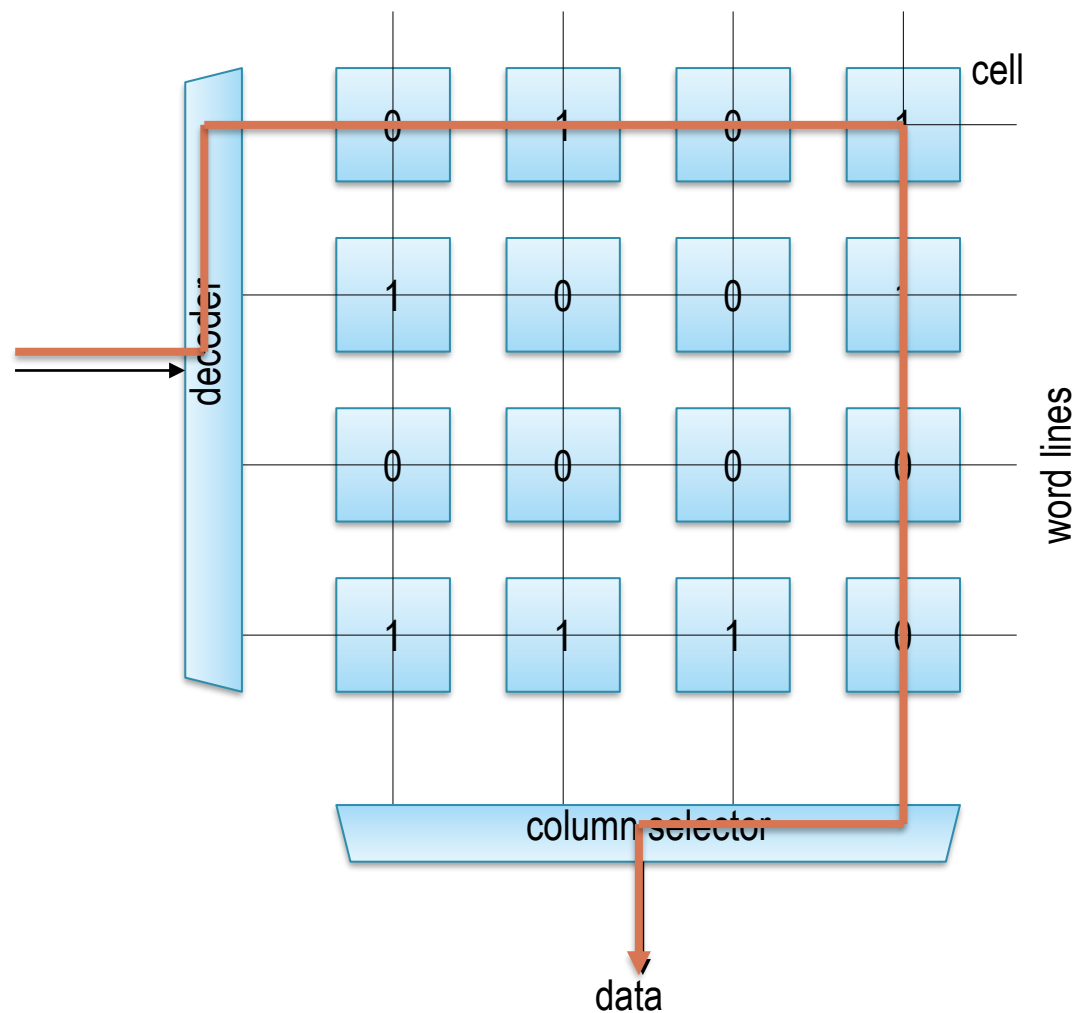
2. メモリの詳細

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

メモリの性質

- 容量（大きい or 小さい）
 - どのくらい数のデータを覚えることができるか
- 速度（速い or 遅い）
 - どのくらいの速さで読み書きできるか
- 容量と速度にはトレードオフがある
 - 「大きくて速くいメモリ」は作ることができない

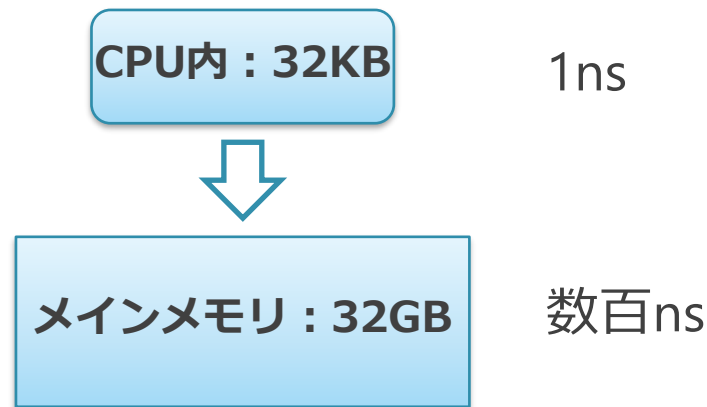
アクセス時間は容量の平方根ぐらいに比例



- 格子状に並んだセルの**外周部の信号線の長さ**がアクセス時間を決める
- 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
- **メモリ容量**
 $= (\text{外周部の長さ})^2$

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、容量に対して線形には伸びない

速度

- 信号線の長さがなぜ問題に？
 - 電気信号が伝わる速度はとても速いのでは？
- 計算してみる：
 - 光の速度： 秒速 30万Km = 3×10^8 m
 - CPU の動作周波数： 3GHz = 3×10^9 Hz
 - 1回の処理で光が進める長さ = $(3 \times 10^8) / (3 \times 10^9) = 0.1\text{m} = 10\text{cm}$
- 光の速度でも大して進めないぐらい、今のコンピュータは速い
 - 回路中の電気信号の伝達は光よりもだいぶ遅い

メモリの速度と容量のまとめ

- 速さと大きさにはトレードオフがある
 - 「大きくて遅いメモリ」か「小さくて速いメモリ」しか作れない
 - 容量を大きくすると、その分メモリ内の信号線が伸びる
- 実際には、セルをどのような方式で作るかでも大きく変わる
 - しかし、上記のトレードオフは常になりたつ

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？

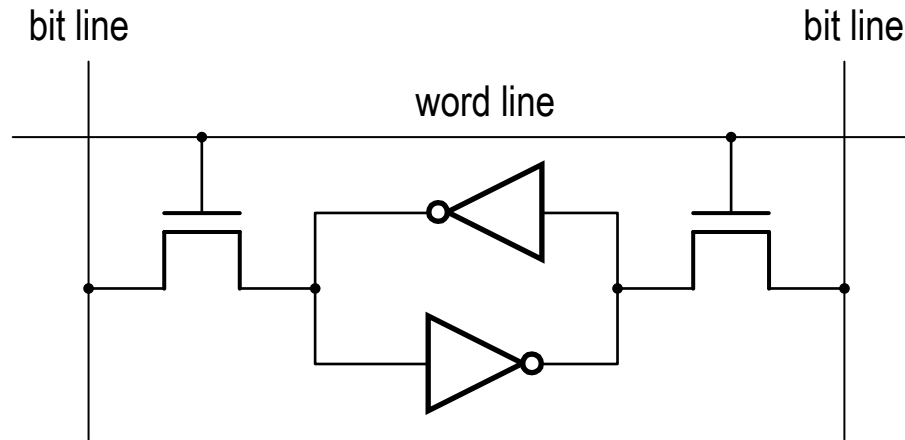
メモリのバリエーション

- メモリは基本的にはみな同じ構造を取る
 - 情報を記憶するセルの中身が方式ごとに異なる
- SRAM と DRAM を簡単に紹介
 - 現代のコンピュータはおおよそこの2種類で作られている
 - この講義的には性質の大ざっぱな違いがわかっているだけでよい
 - （細かい動作まではあまりわかってなくても良い

Static Random Access Memory (SRAM)

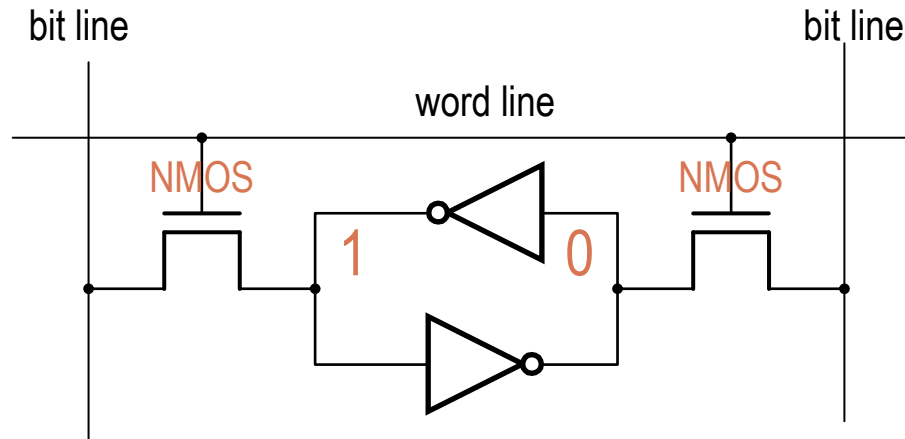
- セルにインバータ（NOT ゲート）のループを使った方式
- Static とはどのような意味か？
 - 動作が静的（何もしなければ電流をほとんど消費しない）
 - 電源を入れている限りは内容が消えない
- 一般的にはメモリの中で最も高速
 - CPU 内の論理回路と同じトランジスタを使って作るから
 - レジスタなどは SRAM で作られている

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
- D-FF とかと同じ原理
 - ただし, クロックに同期させるための仕組みがない分単純

SRAM の読み出し

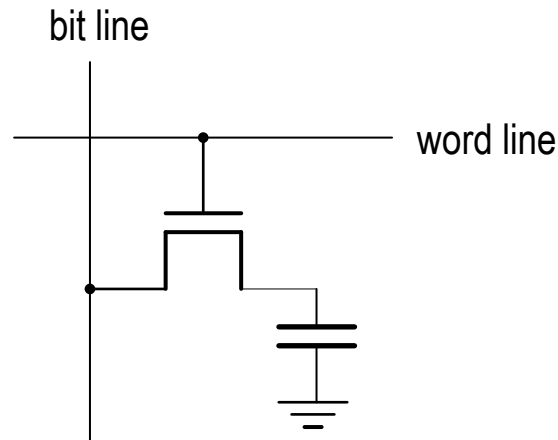


1. 読み出す行のワードラインが高電位（1）に
2. その行の NMOS が ON になりビットラインとインバータが接続
3. ビットラインを通じてセルの内容が伝えられる

Dynamic Random Access Memory (DRAM)

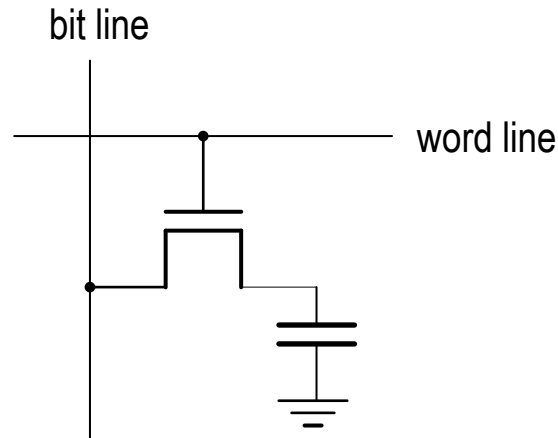
- セルにコンデンサを使った方式
- Dynamic とはどのような意味か？
 - 動作が動的：電荷がもれて記憶が消えていく
- 速度は遅いが、セルを小さく作れるので容量が稼げる
 - メイン・メモリは DRAM で出来ているのが普通

DRAM のセル



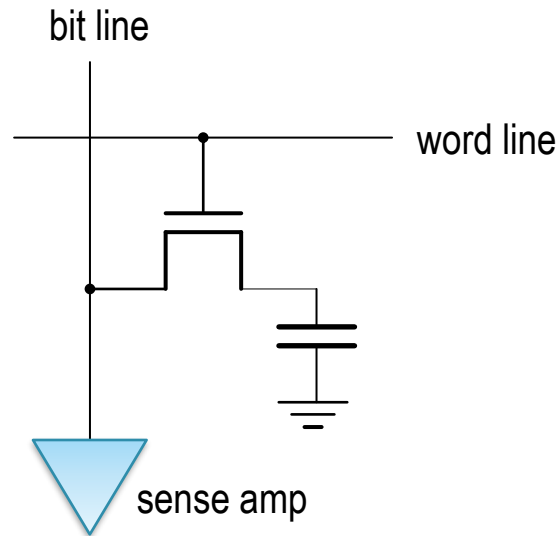
- セルをコンデンサによって構成
 - 電荷がたまっていれば 1, そうでなければ 0
 - 時間が経つと自然に電荷が抜けてデータが消える
- 電荷を維持するため, 定期的にリフレッシュと呼ばれる操作を行う
 - 具体的には, 一回読んで同じデータを書き戻す

DRAM の読み出し



1. ビットラインに電荷をプリチャージ
 - ビットライン全体を一種のコンデンサだと思う
 - 高電位にして電荷を信号線にためる
2. 読み出す行のワードラインを高電位に
 - NMOS が ON になりビットラインと接続
3. セルの状態に応じてディスチャージが起きる
 - セルに電荷がある (1) → ビットラインの状態は変わらない
 - セルに電荷がない (0) → ビットラインの電位が下がる

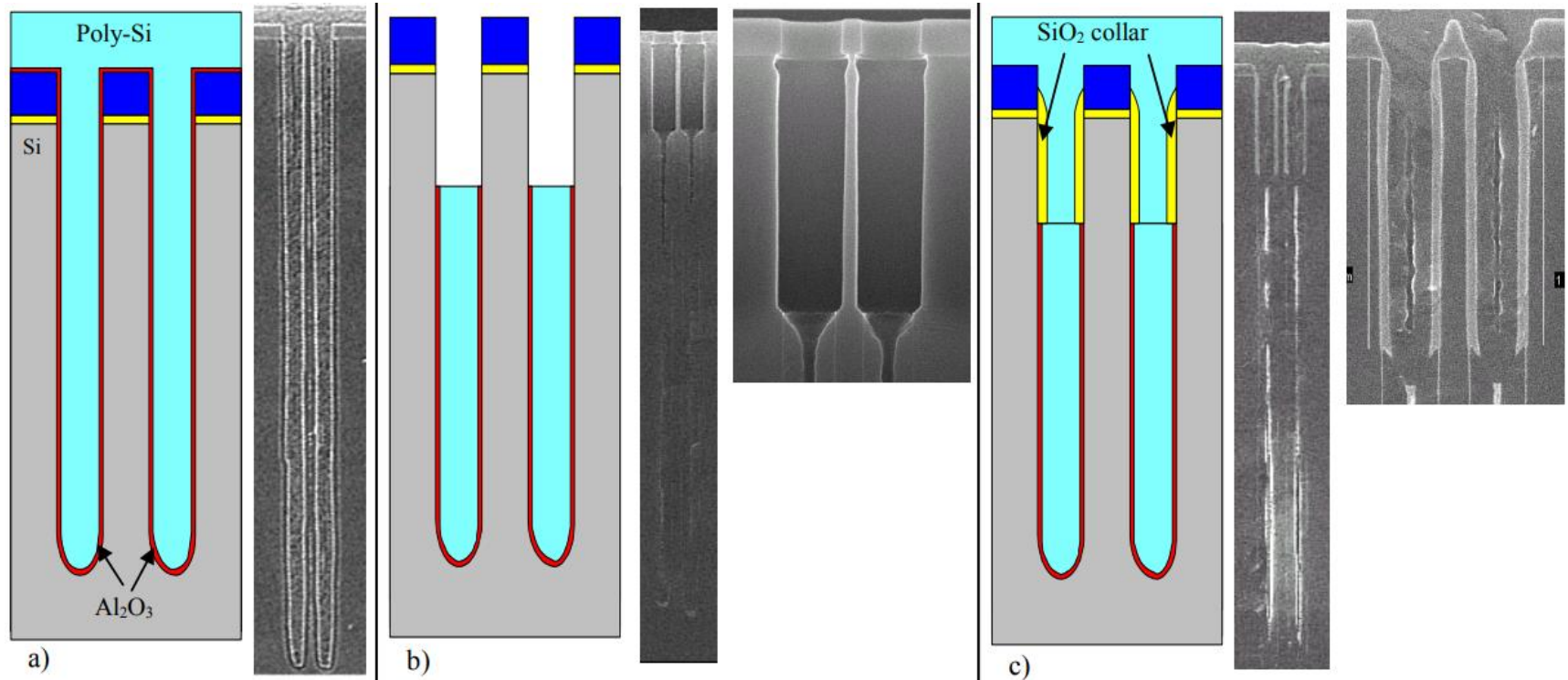
SRAM の場合との違い



- 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 セルのコンデンサの作り方

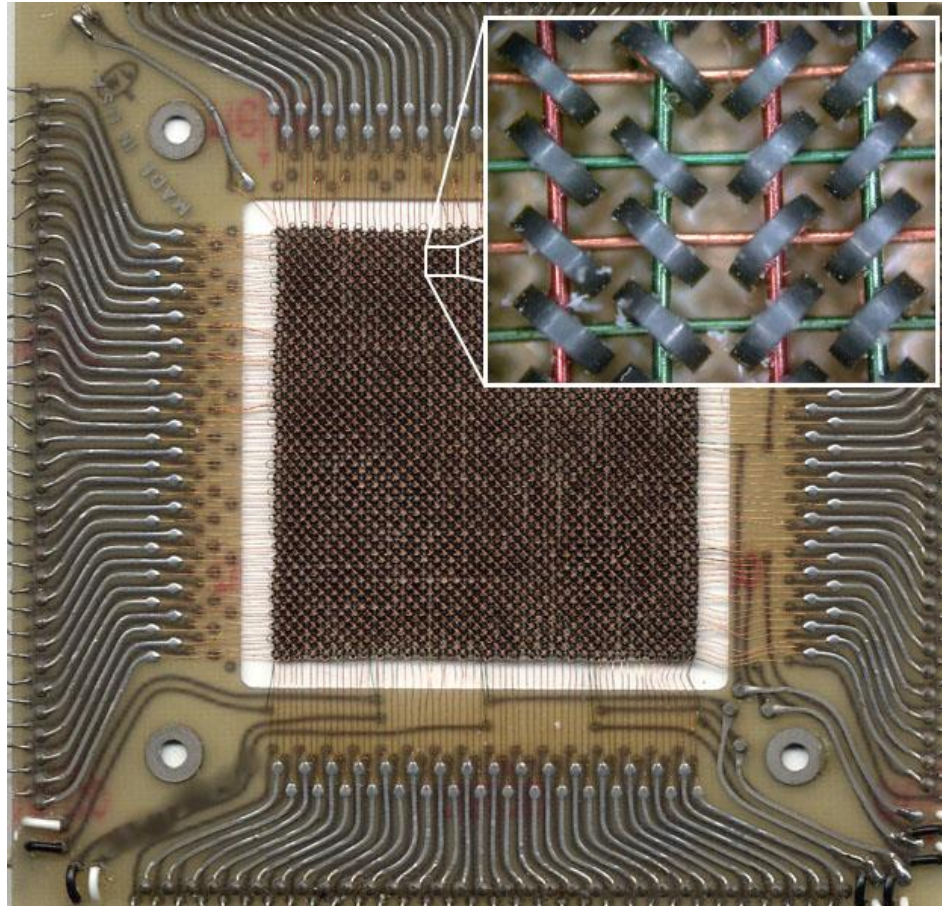
H. Seidl et al, A Fully Integrated Al₂O₃ Trench Capacitor DRAM for Sub-100nm Technology より



- 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
 - DRAM は CPU とは違う専用のチップで製造される

過去に使われていたもの：磁気コアメモリ

<https://ja.wikipedia.org/wiki/%E7%A3%81%E6%B0%97%E3%82%B3%E3%82%A2%E3%83%A1%E3%83%A2%E3%83%AA> より



- 1950年代から使用されていた
- いわゆる「コアダンブ」のコアとは元々この磁気コアのこと

メモリ

1. メモリの基本

1. 構造
2. 動作
3. アクセス時間

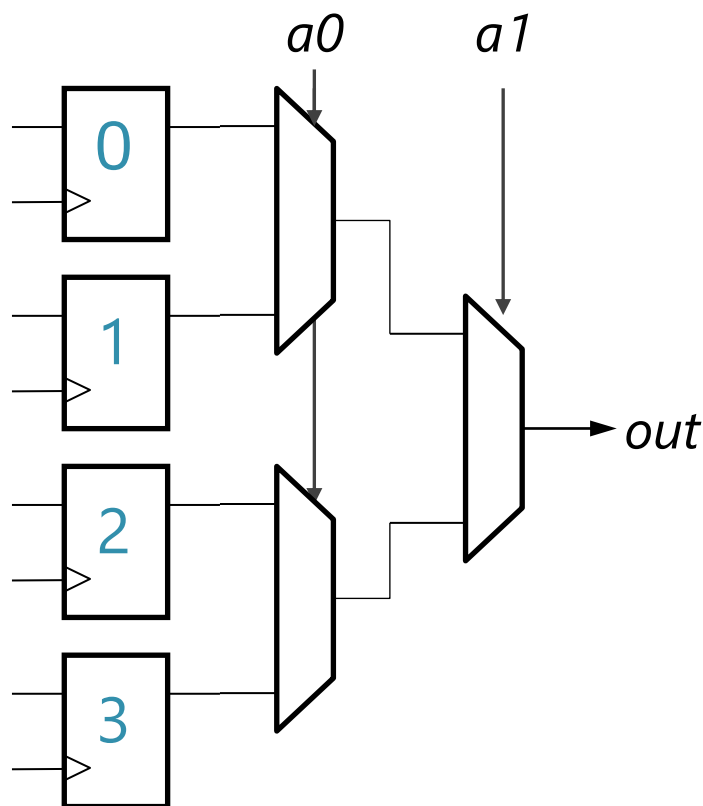
2. メモリの詳細

1. SRAM と DRAM
2. **なぜメモリが存在するのか？**

メモリの存在する理由

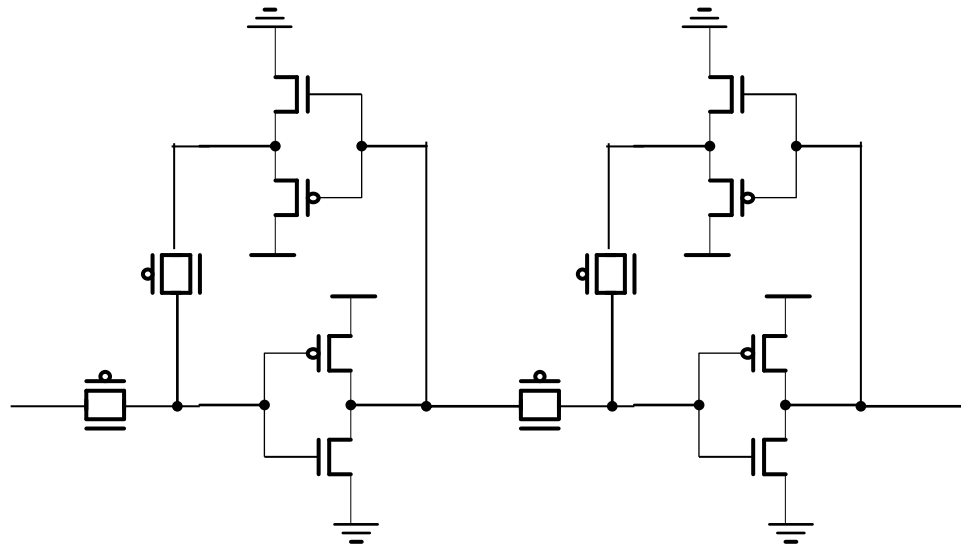
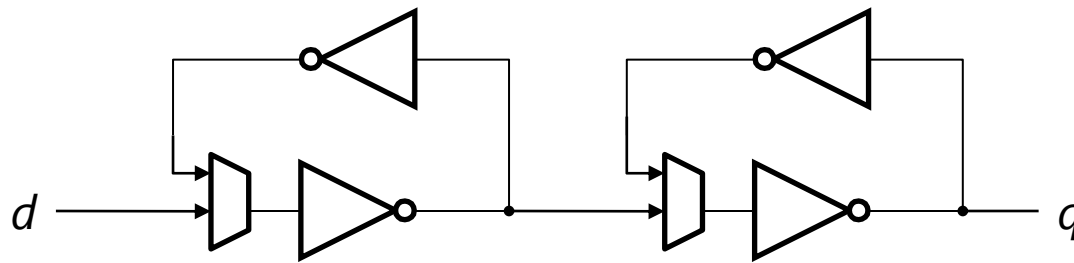
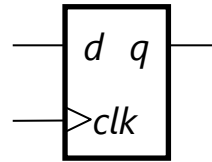
- D-FF とマルチプレクサがあれば、等価な機能は実現できる
 - しかし実際には、メモリ専用の回路が用意される
- メモリ専用の回路が用意される理由：
 - D-FF とマルチプレクサよりも、高密度に実装したいから
 - 同じチップ面積で、より多くの情報を記憶したい

D-FF とマルチプレクサによる 4bit の RAM

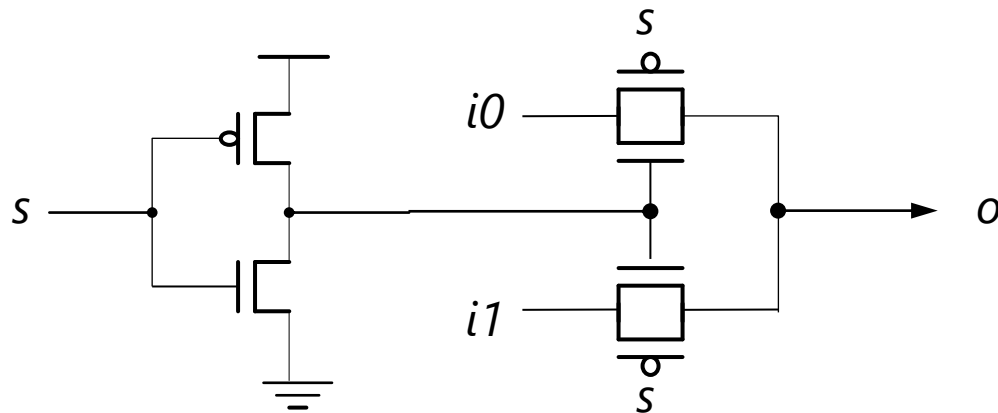
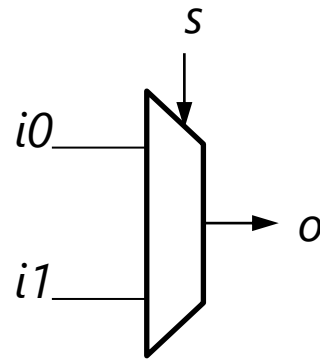


- 4 エントリの LUT を D-FF で構成してみる
 - 中身を憶える 4つの D-FF
 - 場所を指定して選択する2段のマルチプレクサ

D-FF : トランジスタ 16個

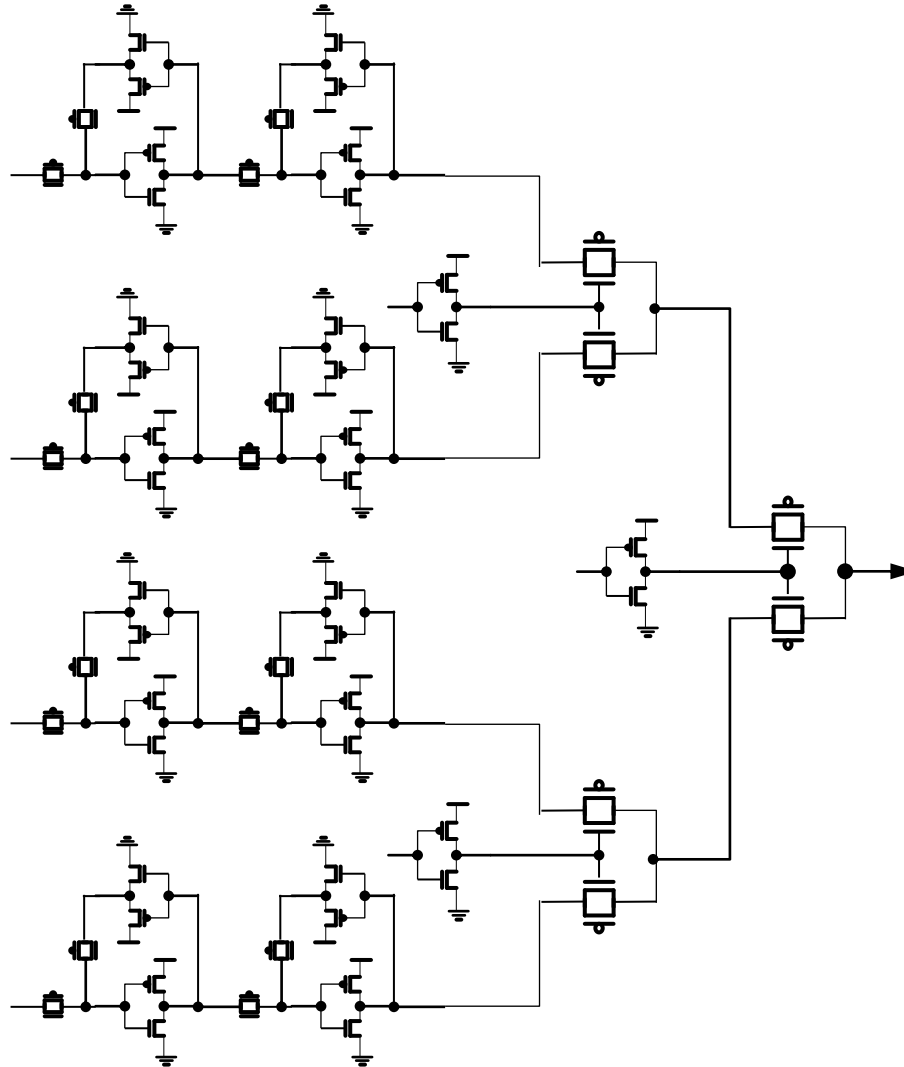


マルチプレクサ：トランジスタ 6個



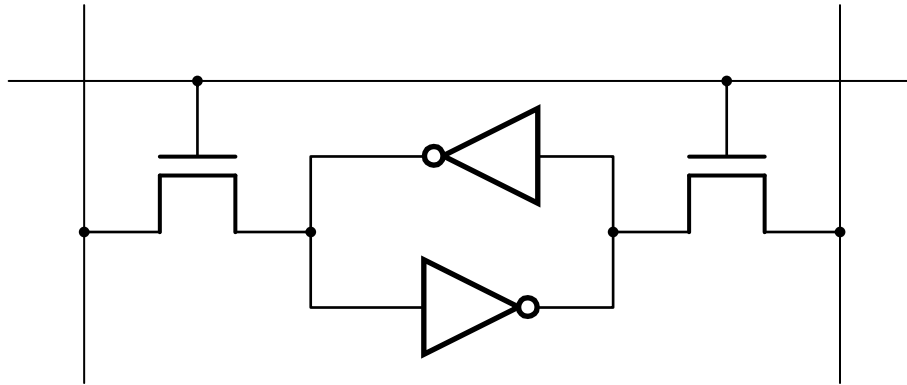
4bit メモリに必要なトランジスタ数 : 82

$$16 \times 4 + 6 \times 3 = 82$$

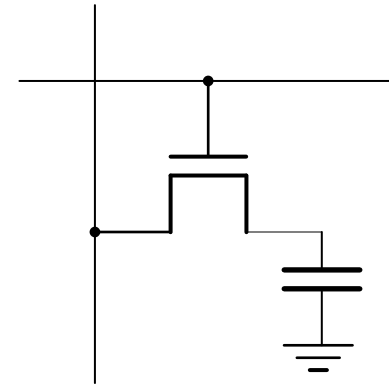


SRAM と DRAM のセル (1bit)

SRAM : $2 + 2 \times 2 = 6$



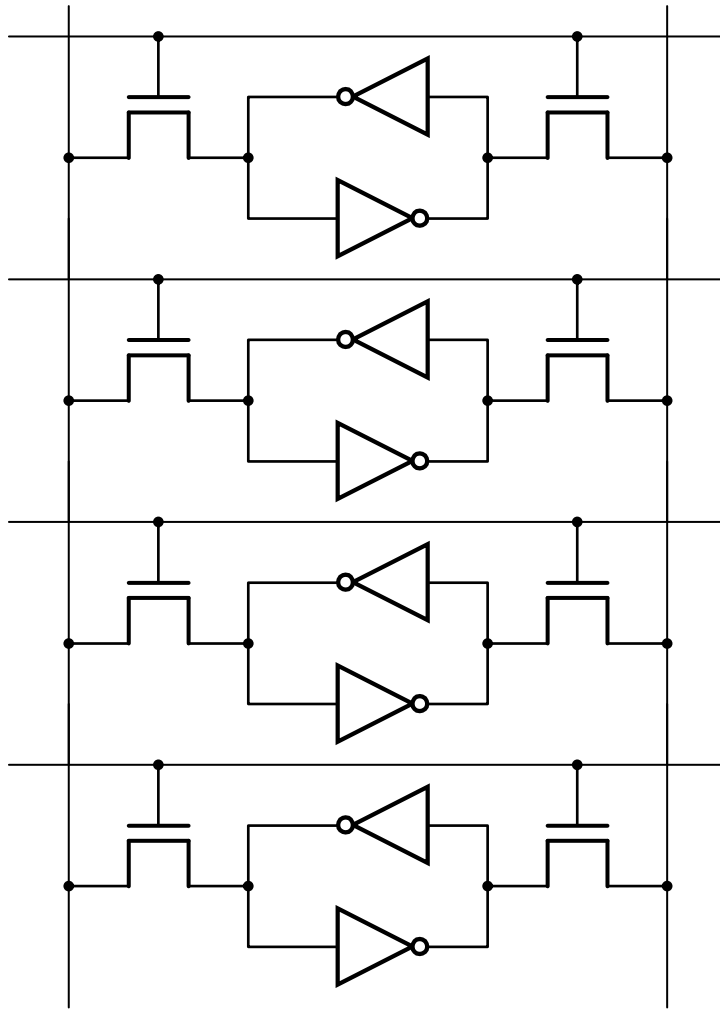
DRAM : 1 + コンデンサ 1



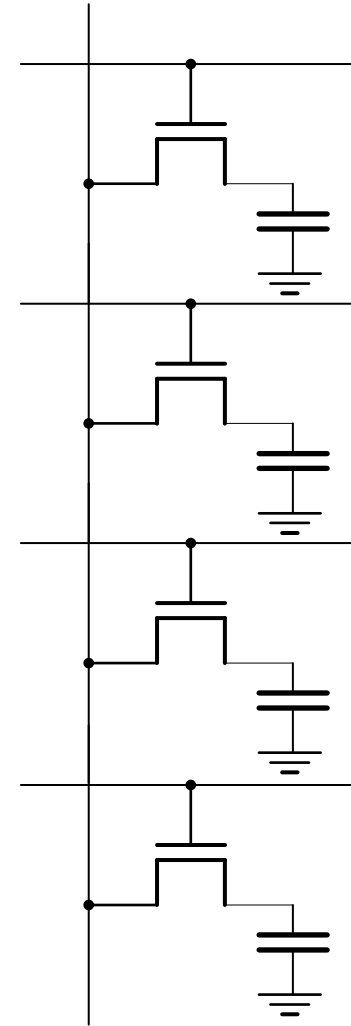
- D-FF よりも圧倒的に単純

SRAM と DRAM (4bit)

SRAM : $6 \times 4 = 24$



DRAM : 4 + コンデンサ 4



メモリのまとめ

(おおよそこのぐらいの事がフワッと分かってれば良い)

■ メモリ (RAM: Random Access Memory)

- 複数のデータを記憶し、配列のように位置を指定して読み書きする
- なるべく少ない回路で なるべく多くの記憶を行うために存在

■ 構造とアクセス時間：

- 1bit の記憶を行うセルを格子状にならべた構造を持つ
- 読み書きにかかる時間：
 - 記憶容量の平方根程度に比例
 - 信号線の長さに比例するため、速度と容量にはトレードオフがある

メモリのまとめ

(おおよそこのぐらいの事がフワッと分かってれば良い)

■ SRAM :

- セルにインバータのループを使ったもの
- 高速だが低密度
- 電源を入れている限りは記憶が消えない

■ DRAM

- セルにコンデンサを使ったもの
- 低速だが高密度
- 電源を入れていても記憶が消えていく
 - リフレッシュが必要

キャッシュ

キャッシュとは？

- 「キャッシュ」って、ブラウザのあれ？
- ちょっと違うけど、原理は同じもの

原理は同じ

■ ブラウザのキャッシュ

- WEB サーバーからページを取ってくるのは遅い
- 1回見たページを PC やスマホの「キャッシュ」に置いておく
- 2回目からは表示が速い

■ メモリのキャッシュ

- メイン・メモリからデータを取ってくるのは遅い
- 1回読んだデータを CPU の「キャッシュ」に置いておく
- 2回目からは読み込みが速い

性能へ大きく影響するし、影響範囲も広い

- キャッシュは現代のコンピュータにおいてとても重要
 - ほとんどのプログラムで性能に大きな影響を与えている

例：行列積の実装と性能

■ 三重ループとして実装できる

- ループの順番は任意に入れ替え可能

```
for (int k = 0; k < SIZE; k++)  
    for (int j = 0; j < SIZE; j++)  
        for (int i = 0; i < SIZE; i++)  
            a[k][j] += b[k][i] * c[i][j];
```

■ 単にループの順番を入れ替えるだけで処理時間が大きく変化する

- 外側から k j i の順 → 178秒
- 外側から k i j の順 → 20秒
- 外側から j i k の順 → 1100秒

■ この変化は、キャッシュを有効に働かせているかどうかによって由来

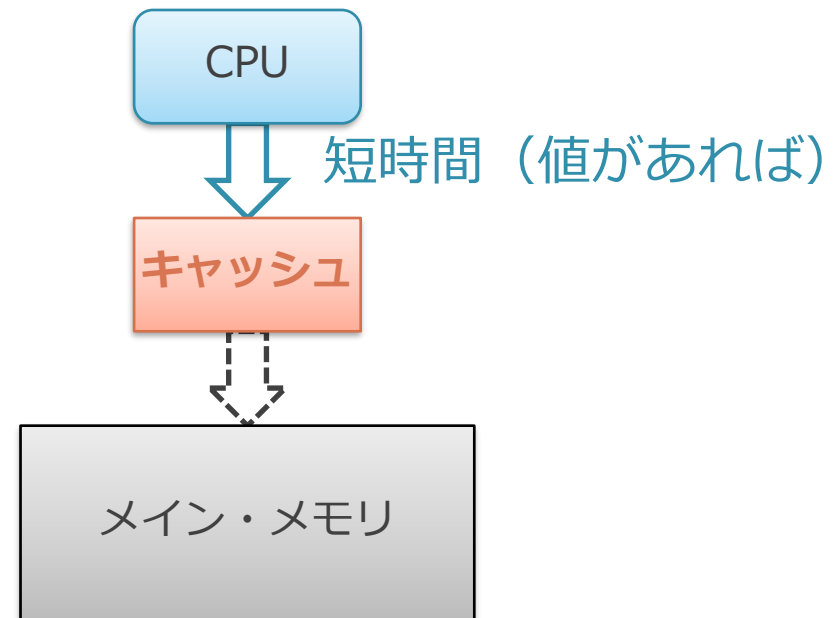
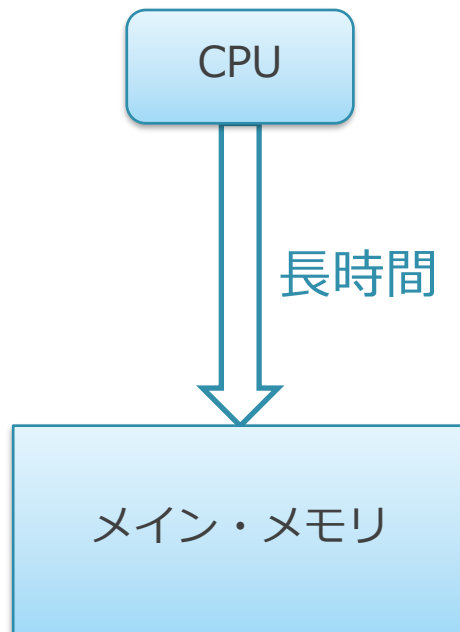
- 計算量や全体としてのメモリ使用量は全く同じなのに

1. キャッシュの基本的な考え方

1. 基本的な原理と構造
2. 容量の性能への影響

記憶階層

- 以下を階層的に組み合わせる
 - 小さいけど速いメモリ：キャッシュ
 - 大きいけど遅いメモリ：メイン・メモリ
- 背景：大きくて速いメモリは作れない

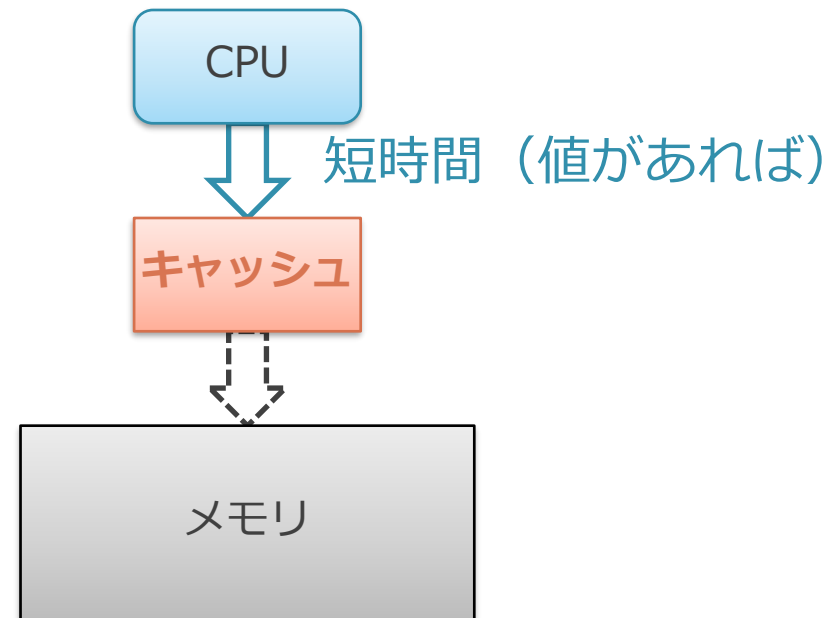
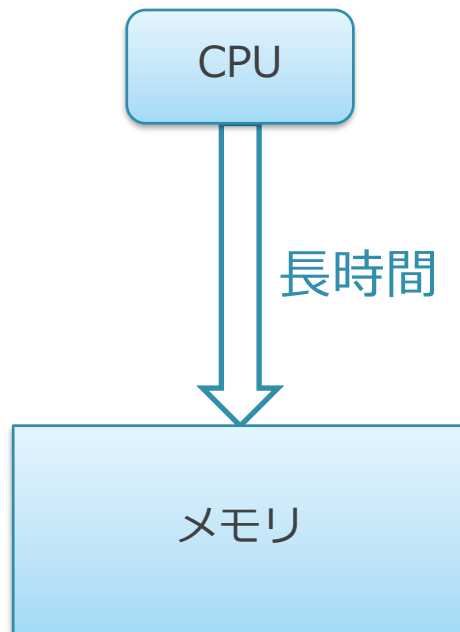


キャッシュの動作

■ ソフトの性質：

- 一度使用した値は，すぐにまた使う可能性が高い
- ブラウザのキャッシュと同じ

■ 一度利用した値を入れておくことで，2回目からは高速に

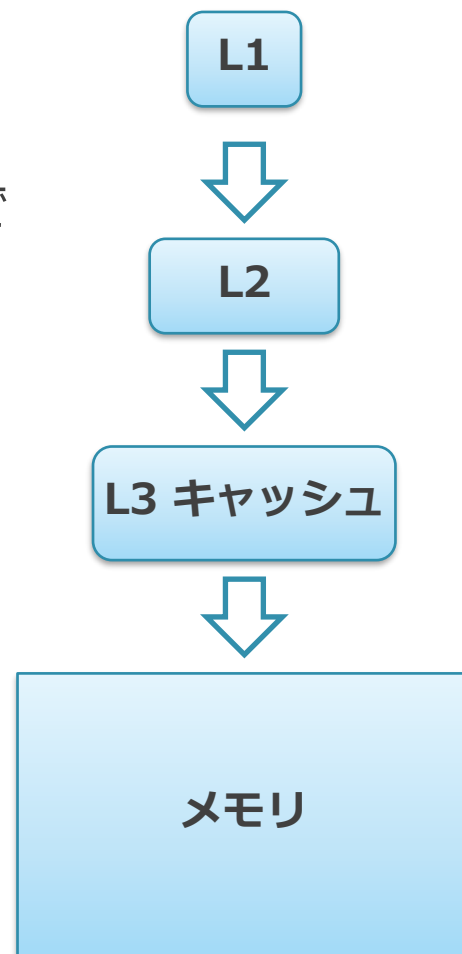


時間的局所性

- 「一度使用した値は、すぐにまた使う可能性が高い」という性質
- なぜか？
 - 人間は、「普通は」関連のある処理同士を近くを書く
 - プログラムの各行ごとに全く関係無い処理を混ぜたりは「普通は」しない
- 関連する処理は、当然同じデータを使う可能性が高い
 - 同じ入力データの使いまわし
 - 前の行の結果を次の行の入力で使う
 - for ループなどはこれらの典型

実際には多層の構造になっている

- 中間の速さと大きさのキャッシュが複数組み合わされている
 - PC に使われている CPU だと L1 ～ L3 まであるのが一般的
 - L は Level の略で、大きいほど（高次）CPU から遠い



キャッシュの基本的な考え方のまとめ

■ 記憶階層

- 小さいけど速いメモリ：キャッシュ
- 大きいけど遅いメモリ：メイン・メモリ

■ 時間的局所性

- 一度使用した値は、すぐにまた使う可能性が高い性質のこと

■ 実際には多層の構造になっている

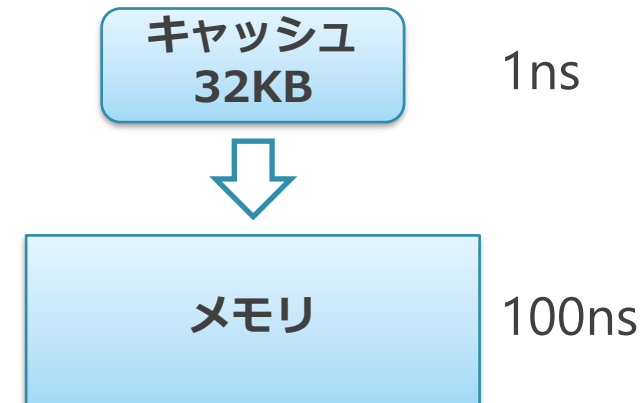
- 中間の速さと大きさのキャッシュが複数組み合わせられている

1. キャッシュの基本的な考え方
 1. 基本的な原理と構造
 2. **容量の性能への影響**

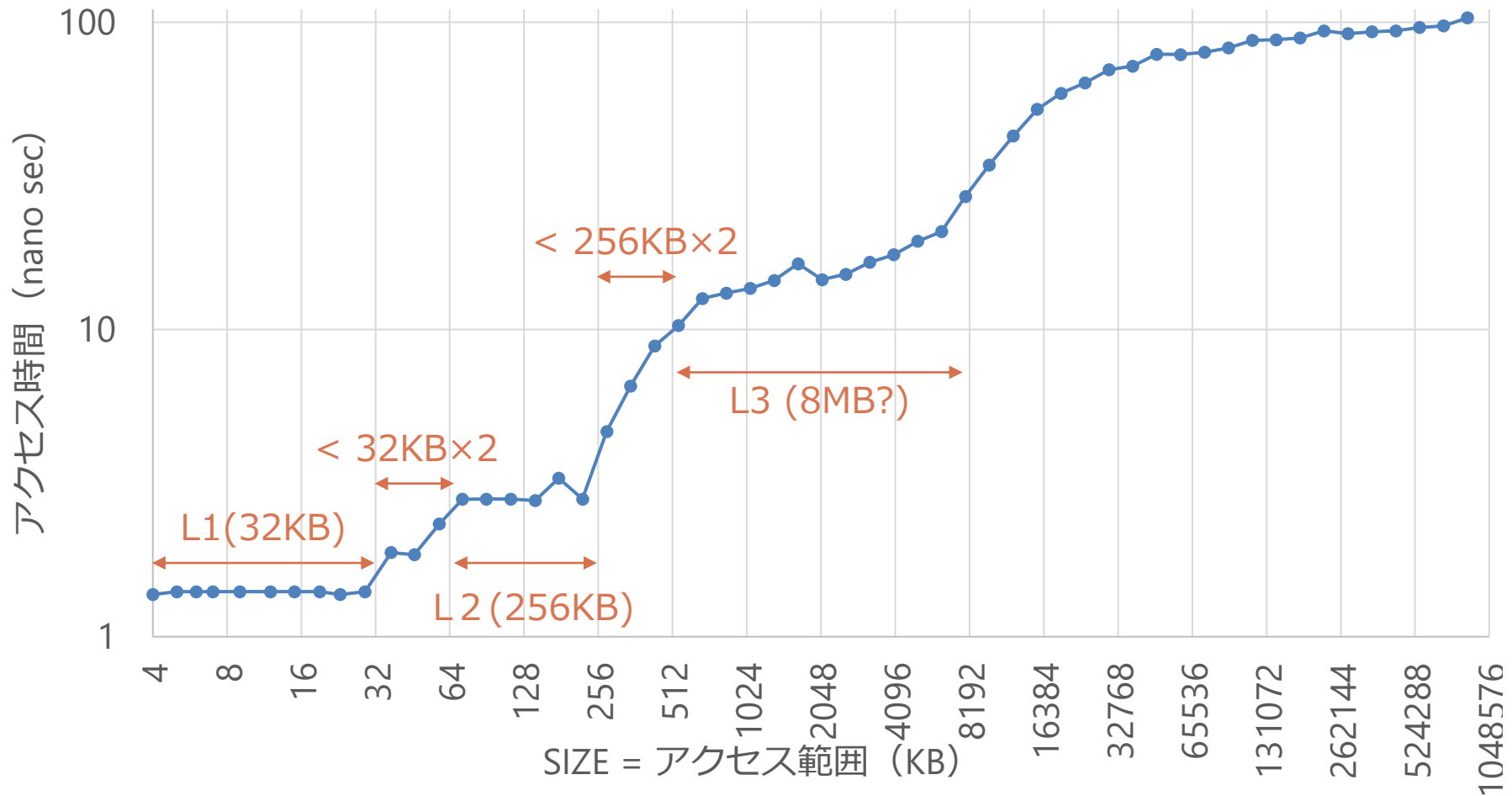
キャッシュへの性能への影響

- 下記のような二重ループを考える
 - SIZE がキャッシュの容量に収まっていれば、内側ループ終了後に table の全データがキャッシュに乗る
 - 次の内側の周回では全データがキャッシュに乗っているので速い！

```
for (int i = 0; i < NUM_TEST; i++) {  
    uint32_t p = 0;  
    for (int j = 0; j < SIZE; j++) {  
        p += table[j];  
    }  
}
```

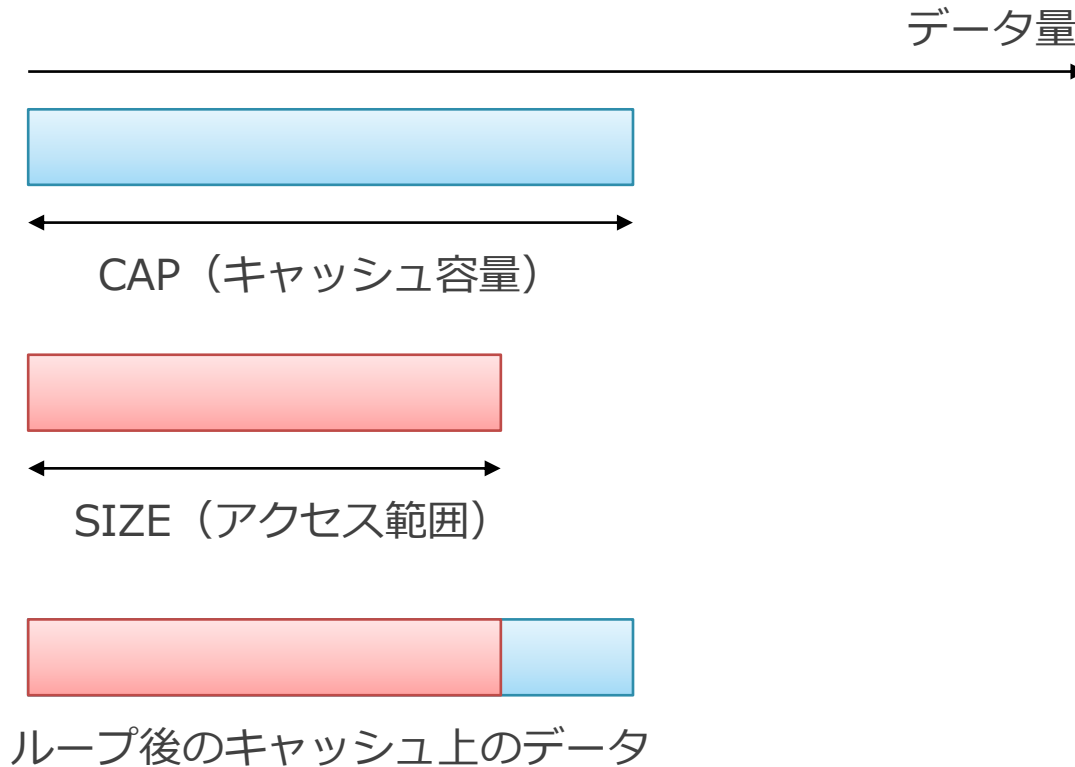


実際の測定データ



- 縦軸を基数10の対数軸, 横軸を基数2の対数軸に
- 低次キャッシュの内容は必ず高次に含まれる仕様
 - (そうなるかはメーカーや世代に依存. 含まれないこともある)
 - 256KB+32KB ではなく 256KB で変化しはじめる

プログラム最適化の上で、重要なポイント



■ 上記の状態を保つこと

- ワーキング・セット：
 - 処理のまとまりごとに、アクセスするデータの範囲（使用量）
- ワーキング・セットがキャッシュ容量に収まることが重要

まとめ

- メモリ
 - 構造, 動作, 容量と速度
 - SRAM と DRAM
 - メモリの存在理由
- キャッシュ
 - 基本的な原理と構造
 - 容量の性能への影響

課題 9

■ 以下のような条件を考える

- 10段のパイプラインを持つ 2-way スーパースcalarプロセッサであり, 理想的には $IPC=2$ で実行できる
- ロード命令の出現率は 0.15
- CPU は L1 キャッシュをもつ
- ロード命令のみが L1 キャッシュやメモリにアクセスするものとする
- L1 キャッシュのヒット時には一切のペナルティなしで実行できる

■ (1) 以下の場合の IPC を求めよ

- ロード命令による L1 キャッシュのアクセス 1 回あたりのミス率が 0.02
- ミスの発生時は 100 サイクル追加で時間がかかる

課題 9

- (2) 以下の場合の IPC を求めよ
 - 1. L1 キャッシュの容量を倍にしたもの
 - L1 キャッシュに良く当たるようになったため, ミス率が 0.01 に
 - 2. L2 キャッシュを追加したもの
 - L2 へのアクセス 1 回あたりのミス率は 0.1
 - ◇ L2 は L1 にミスしたときのみアクセスするものとする
 - L2 ヒット時は L1 ヒット時からの実行時間の追加が 15 サイクル
 - L2 ミス時は L1 ヒット時からの実行時間の追加が 150 サイクル
- (3) これまでに出てきた 3 つのモデルの性能を求め比較せよ
 - ただし L1 キャッシュ容量を倍にした場合,
キャッシュのアクセス時間が伸びてしまうため,
周波数が 0.8 倍になるものとする

期末試験について

- いまのところ 7/30 の予定
- A4 裏表 1 枚 手書きのみの持ち込み可の予定

提出方法

■ 以下を提出：

1. 課題9：

- 提出は Moodle の「課題9」のところからお願いします
- 紙に書いた場合は写真を撮ってアップロードしてください

2. 感想や質問：

- 「感想や質問」のところに投稿してください
- わからない場所がある場合、具体的に書いてもらえると良いです

■ 提出締め切り

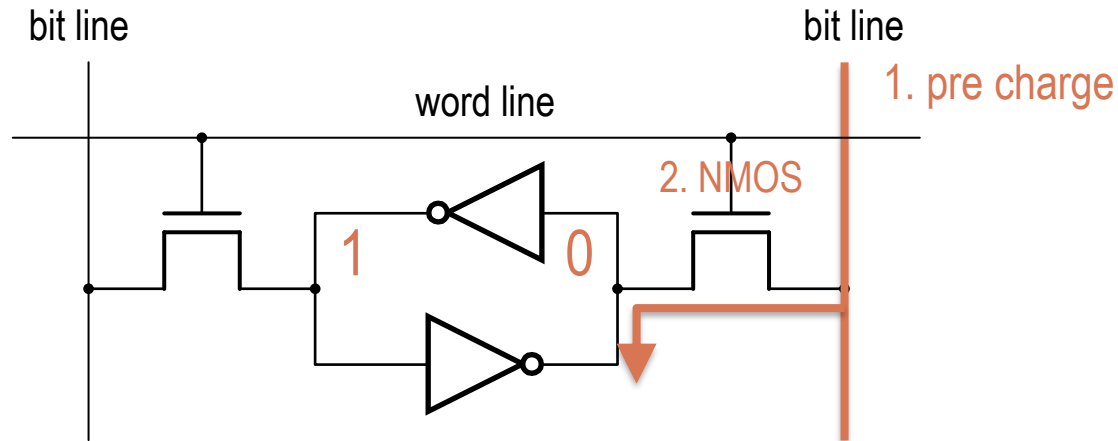
- 次の講義の開始まで

■ 注意：

- 課題の出来は、ある程度努力したあとがあれば良しです
- 必ずしも正解していなくても良いです

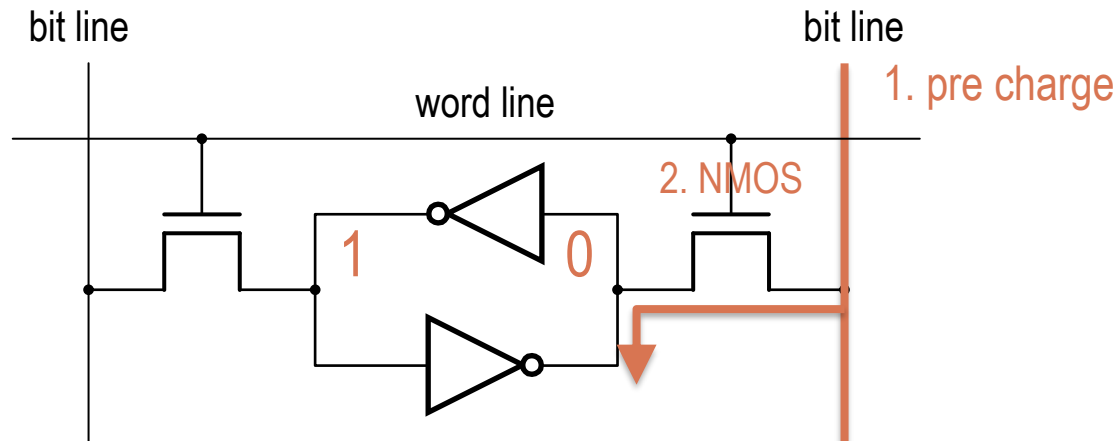
付録

SRAM の読み出し



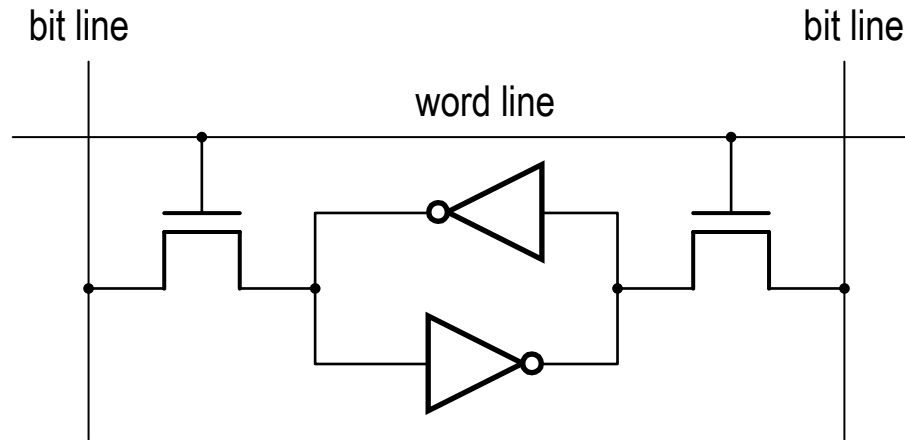
1. ビットラインのプリチャージ
 - ビットラインをあらかじめ高電位（1）にチャージする
2. ワードラインをアサート
 - ループの右にある NMOS が ON に
 - ループとビットラインが接続される
3. ビットラインのディスチャージ
 - ループの右が 0 なら, ビットラインが 0 に
 - ループの右が 1 ならそのまま

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

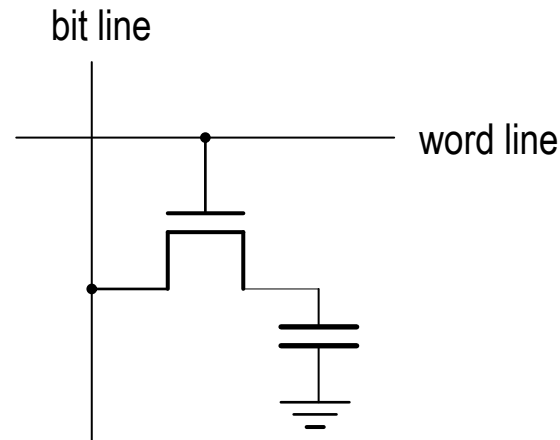


■ 書き込み手順

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

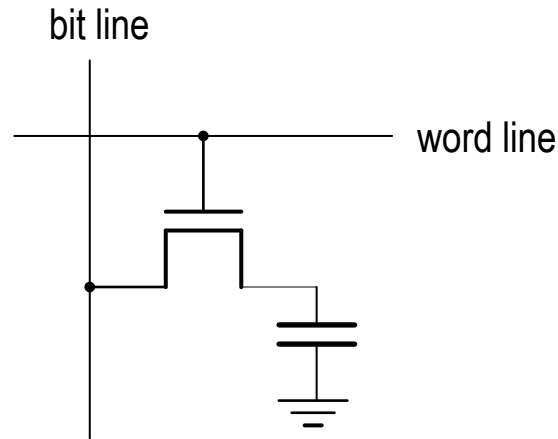
■ NMOS が低電位しか通せないなので、書き込みには 2 本いる

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

質問とか感想

- 課題 7 (7) の解説でやり直した後に結局 LABEL の下の 命令も実行する、がどういうことなのかわかりませんでした。 7 (7) のパイプラインの下 2 行の理解ができていないので、もう一度解説して頂けると嬉しいです。

課題 7

- (7) 以下の命令列を実行するのに必要な時間を計算せよ
ここで beq はオペランドが等しい時に分岐する分岐命令である
プロセッサは分岐予測を行うものとし, beq が分岐予測ミスを起こして LABEL に
飛んだあとにフラッシュされてやり直した場合を想定せよ

li x1 ← 1

li x2 ← 2

beq x1 == x2, LABEL

add x1 ← x2 + x3

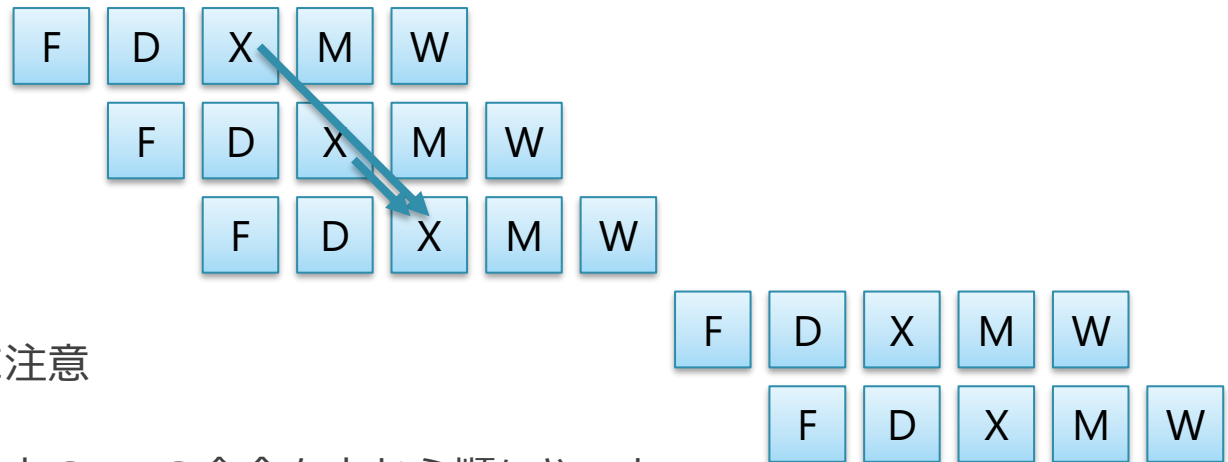
LABEL:

add x2 ← x3 + x4

- 13ns
フォワーディング
ありの場合

やり直した後に
結局 LABEL の下の
命令も実行することに注意

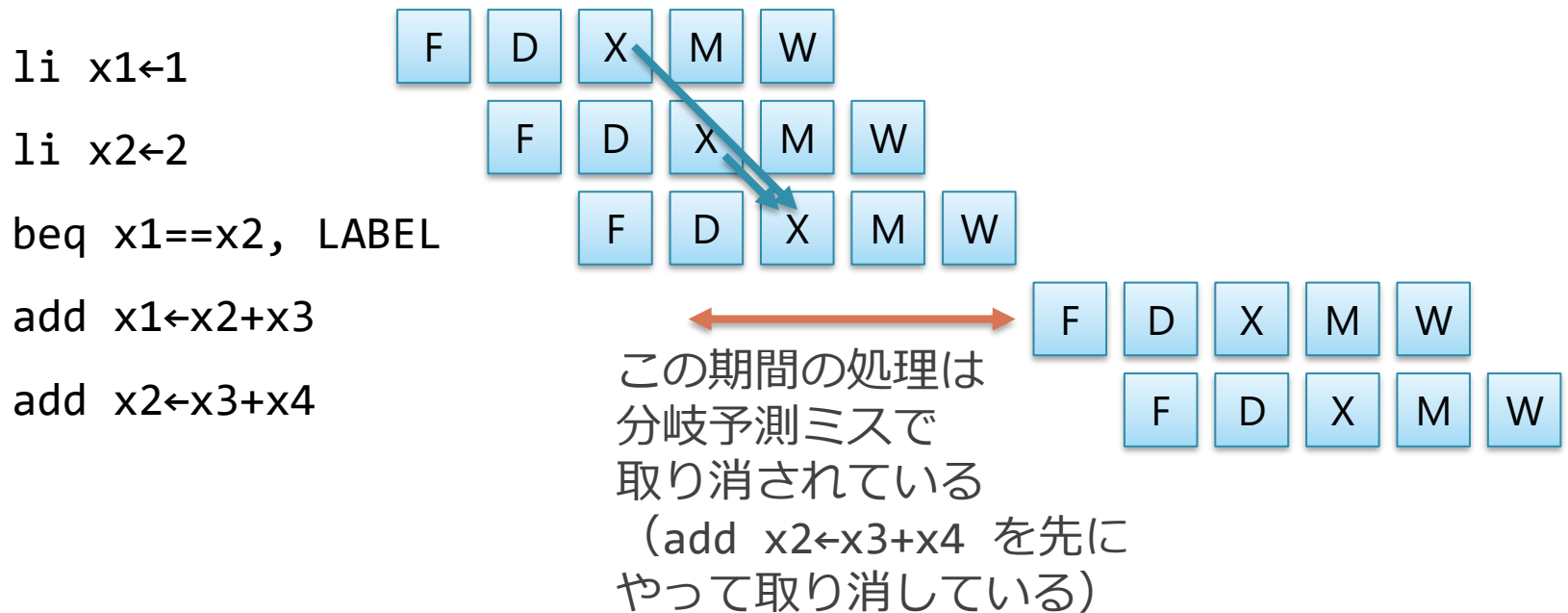
パイプラインの各行は上の5つの命令を上から順にやった
場合に対応



課題 7

■ 13ns フォワーディングありの場合

- やり直した後に結局 LABEL の下の命令 (add x2←x3+x4) も実行することに注意



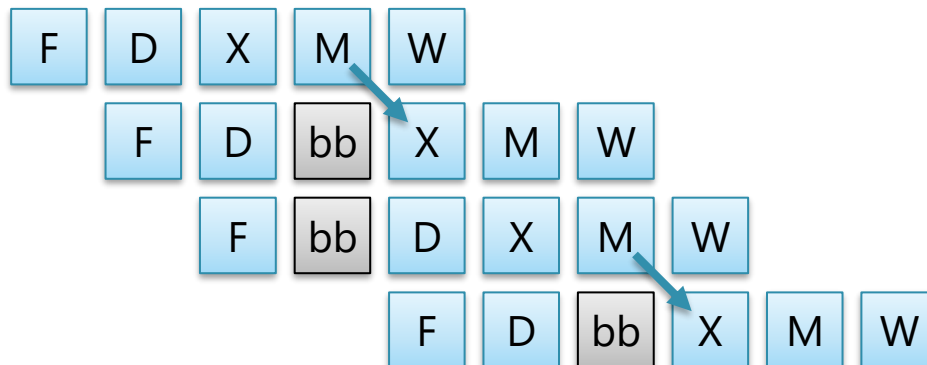
- 課題7で、Dの後にバブルが発生するのと、Fの後にバブルが発生するのの違いが、まだ理解できません。

課題 7

- (5) 以下の命令列を実行するのに必要な時間を計算せよ
依存関係のために必要な場合のみパイプラインを適宜ストールして実行するものとせよ

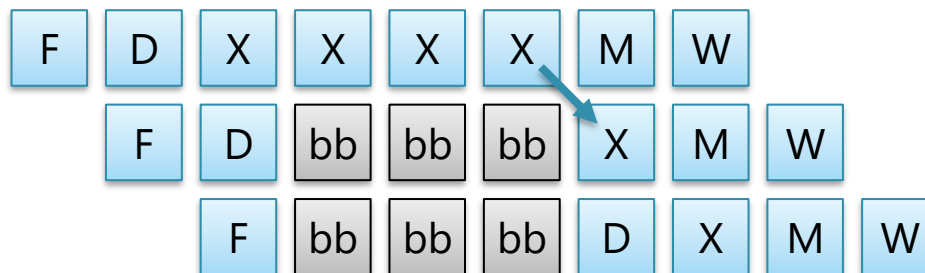
```
ld  x2 ← (x1)
add x1 ← x2 + x3
ld  x2 ← (x3)
add x5 ← x2 + x7
```

- 10ns (フォワーディングありの場合)
1,3 命令目の ld では M が終わるまで値が取れないため,
それに依存した後ろにある命令ではバブルが生じる (ストールが発生する)



課題 7

- (6) 以下の命令列を実行するのに必要な時間を計算せよ
ここで mul は乗算命令であり X に 4 サイクルが必要である。
依存関係を満たすために必要な場合のみ
パイプラインを適宜ストールして実行するものとせよ
mul $x2 \leftarrow x1 + x4$
add $x1 \leftarrow x2 + x3$
add $x5 \leftarrow x2 + x7$
- 10ns (フォワーディングありの場合)



- 課題の中の C_p はどのように求めますか？

- 授業を聞いているときは内容を理解できているか怪しかったのですが、実際に課題を解いて見るとIPCの計算の意味だったりを理解できたような気がします。

- クロック周波数を上げて消費電力が上がらない方法はありませんか

- データハザードが発生しやすいプログラムにはどのようなものがありますか

- どの方法が一番早くなる、みたいな計算は実際の現場だとどのよう
に求められているのか気になりました。

質問とか感想

- がっつり数式が出てきてビビりましたが、そこまで複雑ではなかったなので助かりました。
- 最初言葉と数式でIPCの式を説明された時は混乱しましたが、実際手を動かしながらスライドを見返すと意味がわかってきて理解することができました。同じ1部分変えるという条件でも性能の変化は様々で面白かったです。

質問とか感想

- どんどん複雑になってきて作業と化してるけど、実際には授業で扱った内容をきちんと理解して発展させられるようにならないと意味がないように感じるし、なんだかゾッとしてきます。
- あと、この間、昔エンジニアだったおじいちゃん与会ったときに、一方的にコンピュータの難しい構造について語られたのですが、この授業で扱った内容と被ってて話の意味が少し理解できて盛り上がりました。

- p20の、パイプライン段数を倍にした場合、1 サイクルあたり 2 命令の仕事をしているかと思った。
 - 同じ「時間」あたりで 2 命令の仕事をしています

- 分岐予測のミス率が思っていたよりも小さく設定されていて少し驚きました。if文なら2択だから確率も半々くらいなのではと思っていました。
当たり前ではありますが、実際に計算してみて性能が上がっているのを見るとなんとなく嬉しい感じがします。

- アスキーねこの出番が減るということで少し悲しいです……それはさておき、IPCの式に出てくる用語が代入するうちにこんがらがりそうで怖いです。しっかり復習しようと思います。