

コンピュータ アーキテクチャ I 第10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

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 0.2
- 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 0.01
- ミスの発生時は 100 サイクル追加で時間がかかる

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,
 - $C_r = C_t + N_h \times C_p$

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,

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

- 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

- $$IPC_r = \frac{1}{\frac{C_t}{N_i} + P_i \times P_h \times C_p} = \frac{1}{\frac{1}{IPC_t} + P_i \times P_h \times C_p}$$

- ここで IPC_r は実際の IPC, IPC_t は理想 IPC

課題 9 (1)

■ (1) の解

- $$\frac{1}{\frac{1}{IPCt} + Pi \times Ph \times Cp} = \frac{1}{\frac{1}{2} + 0.2 \times 0.01 \times 100} = \frac{1}{0.7} \approx 1.43$$

■ ちなみに, キャッシュが全く無い場合 (補足) :

- $$\frac{1}{\frac{1}{IPCt} + Pi \times Ph \times Cp} = \frac{1}{\frac{1}{2} + 0.2 \times 1 \times 100} \approx 0.049$$

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 0.006 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 10 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 100 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に時間がかかるため、周波数が 0.8 倍になるものとする

課題 9 (2)

■ 容量が倍

- $$\frac{1}{\frac{1}{IPCt} + Pi \times Ph \times Cp} = \frac{1}{\frac{1}{2} + 0.2 \times 0.006 \times 100} = \frac{1}{0.62} \approx 1.61$$

■ L2 の追加

- L1 ヒット : ペナルティなし
- L1 ミス & L2 ヒット : $Ph1 = 0.01 \times (1 - 0.1) = 0.009, Cp1 = 10$
- L1 ミス & L2 ミス : $Ph2 = 0.01 \times 0.1 = 0.001, Cp2 = 100$
- $$\frac{1}{\frac{1}{IPCt} + Pi \times Ph1 \times Cp1 + Pi \times Ph2 \times Cp2} = \frac{1}{\frac{1}{2} + 0.2 \times 0.009 \times 10 + 0.2 \times 0.001 \times 100} \approx 1.85$$

課題 9 (3)

- すいません, これは問題が良くなかったです
 - CPU の周波数が変わってもメモリの速度は変わらない
 - メモリのアクセスのサイクル数が変わってしまうので, IPC が変化する
- 例 : 周波数 1G Hz の場合
 - 1 サイクル=1ns
 - 100サイクルかかるメモリアクセスは100nsになる
 - 周波数が 0.8 倍になると, 1サイクルは $1/0.8\text{GHz}=1.25\text{ns}$
 - $100\text{ns}/1.25=80$ サイクル
 - メモリアクセス時間は変わらない

課題 9 (3)

■ 容量が倍の際の IPC を再計算

- $$\frac{1}{\frac{1}{IPC_t} + P_i \times P_h \times C_p} = \frac{1}{\frac{1}{2} + 0.2 \times 0.006 \times 100 \times 0.8} = \frac{1}{0.62} \approx 1.68$$

■ 性能：

- もともと： $1.43 \times 1 = 1.43$
- L1容量倍： $1.68 \times 0.8 \approx 1.34$
- L2追加： $1.85 \times 1 \approx 1.85$

キャッシュの詳細

内容

1. キャッシュの構成方法
2. 行列積での動作例

キャッシュの構成方法

キャッシュの構成方法

1. 3つの方式：
 1. 基本的な構造（フルアソシアティブ方式）
 2. ダイレクトマップ方式
 3. セット・アソシアティブ方式
2. ライン単位での管理
3. アドレスとキャッシュ構造の具体的な対応関係

キャッシュの作り方の方針

■ キャッシュ：

- 小容量で高速なメモリ
- メイン・メモリの一部をコピーして保持
 - こっちを略してメモリということも

■ 目的のデータがコピーされているかどうかを確認したい

- コピー時に、どここのデータをコピーしたかの情報も一緒に記録
 - つまり、コピー元のアドレスもキャッシュに記録する
- キャッシュの読み書き時は、記録されているアドレスとの突き合わせをして確認する

キャッシュの基本的な構造

アドレスやデータは 16 進数

アドレス データ

0000 84

0001 ff

0002 12

⋮

8000 33

8001 55

メモリ

タグ データ

0002 12

8001 55

エントリ

容量が 2 エントリのキャッシュ

■ キャッシュのエントリの内容

- タグ : コピーしてきたデータが、メモリのどこのアドレスにあったかを表す
(後で詳しく話すように本当はアドレスの一部が入る)

- データ : その内容

■ コピー時に元のアドレスと一緒に格納する

- 上記の例 :
0002 にあった 12 と、8001 にあった 55 を保持

読み出し時の動作

アドレスやデータは 16 進数

アドレス	データ
0000	84
0001	ff
0002	12
⋮	⋮
8000	33
8001	55
	メモリ

タグ データ

0002	12
8001	55

容量 2 のキャッシュ

1. まず全てのタグを読み出す（この場合 2 つ）
 2. アドレスと一致するタグがあるかをチェック
 1. ヒット：もしあれば、そのデータを読む
 2. ミス： なければ、メモリにアクセス
- たとえば CPU がアドレス 8001 を読むと、タグに 8001 があるのでヒット

フルアソシアティブ方式とその問題

タグ データ

0002	12
8000	33

容量2のキャッシュ
2つのタグをチェック

タグ データ

0002	12
8000	33
0102	00
5511	78

容量4のキャッシュ
4つのタグをチェック

- 先ほどの方式をフルアソシアティブ方式と呼ぶ
 - キャッシュ内の全てのタグをチェックする方式
- 問題：
 - 格納データ数を増やすと、比例して比較するタグの数が増える
 - 比較のための回路は複雑で遅いし、電気もバカ食いする

ダイレクトマップ方式

アドレスやデータは 16 進数

	タグ	データ
0	800 0	12
1	100 1	33
2	010 2	00
3	550 3	78

- 全てのエントリではなく、アドレス毎に特定の1つのエントリのみを使う
 - 比較が1つのみでよくなる
- 「アドレス mod サイズ」の番号のエントリにアクセス
(mod は剰余, 数字は16進数表記)
 - アドレス 8000 : $8000 \bmod 4 = 0$ 番にアクセス
 - アドレス 5513 : $5513 \bmod 4 = 3$ 番にアクセス

ダイレクトマップ方式

アドレスやデータは 16 進数

	タグ	データ
0	800 0	12
1	100 1	33
2	010 2	00
3	550 3	78

■ フルアソシアティブとの違い：

- 利点：チェックするタグは常に 1 つですむ
- 問題：アドレス下位がかぶると（競合とよぶ），上書きされる
 - 800**0**, 700**0**, 010**0** の順にアクセスがあると，0 番しか使えない

セットアソシアティブ方式

アドレスやデータは 16 進数



	タグ	データ	タグ	データ
0	8000	12	0100	53
1	1001	33	7701	44
2	0102	00	5102	22
3	5513	78	0503	87

- 「アドレス mod サイズ」のセットにアクセス
 - 上の例の場合, 1つのセット内に2つのタグ+データがある
- 連想度 :
 - セットの中にいくつ要素を入れるかのこと
 - 上記の場合連想度は2 (2-way と呼ぶ)
- 利点 : 競合するデータを複数持てる
 - キャッシュに必要なデータが在る率 (ヒット率) 上がる

セットアソシアティブ方式の動作

		タグ	データ	タグ	データ
	0	800 0	12	010 0	53
	1	100 1	33	770 1	44
	2	010 2	00	510 2	22
	3	551 3	78	050 3	87

■ アドレス 0100 にアクセスがあった場合：

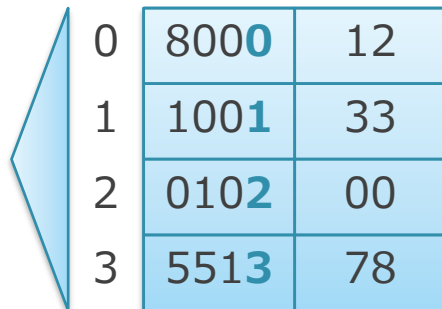
- 「 $0100 \bmod 4 = 0$ 」よりセット 0 のタグを全て読んで、これと比較
- 右側のタグ 0100 がヒットしたので、ここを読み出す

■ どこにもヒットしなかった場合

- メモリからデータを取ってきて、キャッシュに書き込む
- 同一セット内で最も長時間アクセスされていないものに書き込むことが一般的

容量一定 (= 4) にして連想度を変えた場合

連想度 1
(=ダイレクトマップ)




0	8000	12
1	1001	33
2	0102	00
3	5513	78

連想度2



0	8000	12	0100	53
1	1001	33	7701	44

連想度4
(=フルアソシアティブ)



0	8000	12	0100	53	7000	12	0500	53
---	------	----	------	----	------	----	------	----


■ 容量 = 連想度 × セット数

■ 各方式との関係：

- ダイレクトマップ： 連想度= 1 のとき
- フルアソシアティブ： 連想度=容量のとき

競合と複雑さのトレードオフ

連想度 1
(=ダイレクトマップ)



0	8000	12
1	1001	33
2	0102	00
3	5513	78

連想度 2



0	8000	12	0100	53
1	1001	33	7701	44

連想度 4
(=フルアソシアティブ)



0	8000	12	0100	53	7000	12	0500	53
---	------	----	------	----	------	----	------	----

■ 容量一定の場合のトレードオフ

- 連想度大：競合の影響が小さいが、回路が複雑
- 連想度小：競合の影響が大きいが、回路が簡単

■ 現実的には、連想度 2 から 32 ぐらいまでが良く使われる

各方式のまとめ

■ キャッシュ

- 小容量で高速
- メモリの一部をアドレス（タグ）と共にコピー

■ 方式

- ダイレクトマップ
- セットアソシアティブ
- フルアソシアティブ

■ 性質

- 連想度によって分類可能
- ヒット率と複雑さにトレードオフ

キャッシュの詳細

1. 方式：

- 基本的な構造
- ダイレクトマップ方式
- セット・アソシアティブ方式

2. ライン単位での管理

3. アドレスとキャッシュ構造の対応

ライン

- キャッシュ上のデータはラインと呼ばれる単位で管理される
 - ライン：複数バイトからなる塊
 - 実際には 16 から 128バイトぐらい
 - ブロックと呼ばれることもある
- 理由：
 1. 容量の効率をあげるため
 2. 空間局所性を利用するため

容量の効率

タグ データ
(32bit=4バイト) (1バイト)

f3568000	12
----------	----

■ タグが大きくて無駄

- これまでの説明では、アドレスごとに1バイトのデータを仮定
- 一方、アドレスは 32 から 64 ビット
- このままではデータよりもタグを覚えているようなもの

容量効率の向上

■ ライン

- タグが指すアドレスから始まるデータのまとまりのこと
- キャッシュの各エントリでは、このライン単位でデータを持つ

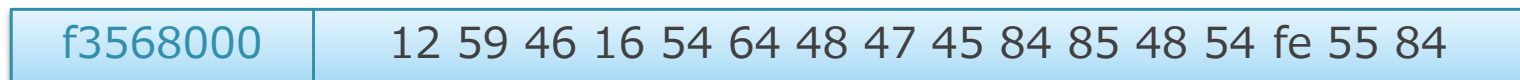
■ 利点：ラインサイズが増えると、データが占める割合が増える

- 1 バイト： $1 \times 4 = 4$ バイト
- 16 バイト： 16 バイト
- (双方、タグとデータ合計で20バイト)

タグ データ
(32bit=4バイト) (1バイト)



タグ ライン (16バイト)
(32bit=4バイト)



空間的局所性

■ 2種類の局所性

1. 時間的局所性：

- 「一度使ったデータは、**すぐに**また使われる」

2. **空間的局所性**：

- 「あるデータが使われると、**その近くにある**データも使われる」

■ 空間的局所性の例、

- 以下では i 番目がアクセスされると $i+1$ にもアクセスされる

```
for(i = 0; i < SIZE; i++)
```

```
    v += buf[i]
```

- ある構造体内の要素にアクセスがあると、その構造体の別の要素にもアクセスがある

ライン単位の管理と空間局所性

- データはライン単位でやりとりされる
 - あるデータがアクセスされると、周囲のデータも一緒にキャッシュに格納される
- たとえば、ラインが 16 バイトだった場合
 - 各要素は1バイトで16要素の配列 `buf[16]` を考える
 - `buf[0]` のアクセス時に、`buf[1] ~ buf[15]` までをまとめて読む
 - まとめてメモリから取ってきてキャッシュにおく
 - `buf[1]` から `buf[15]` アクセス時は、キャッシュにヒット

キャッシュの詳細

1. 方式：
 - 基本的な構造
 - ダイレクトマップ方式
 - セット・アソシアティブ方式
2. ライン単位での管理
3. アドレスとキャッシュ構造の対応

キャッシュ内のデータの配置

- 以下に要素に関連して変化
 - 連想度
 - 容量
 - ラインのサイズ
- プログラムの高速化のためには、以下を知る必要がある
 - アドレスとキャッシュ内のラインの位置の対応
 - 結果、どのようにアクセスするとキャッシュにヒットするのか
- さらに後半ではいくつかの実例をつかって説明

セットアソシアティブ・キャッシュの例



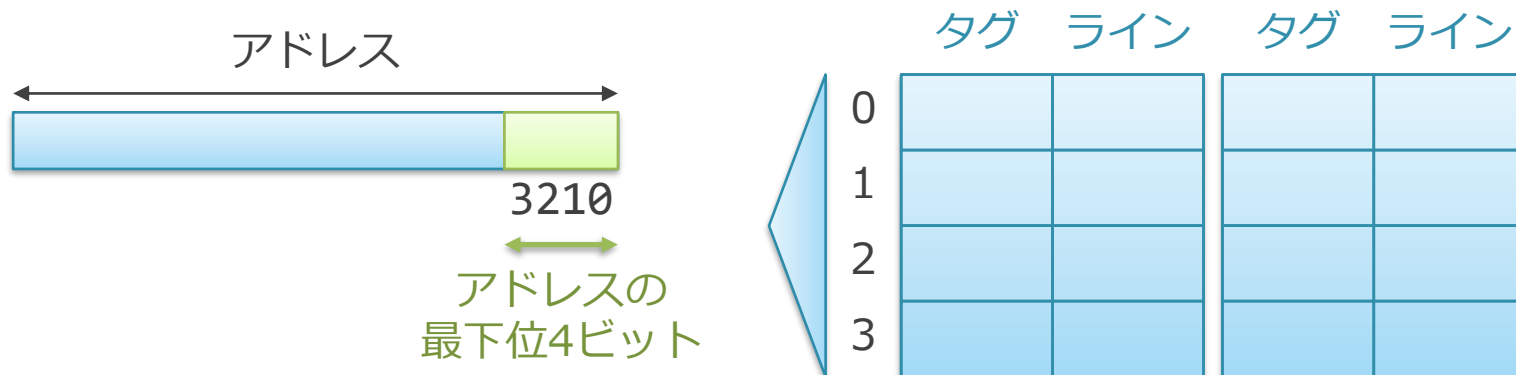
■ 構成

- 連想度 : 2
- セット数 : 4
- ラインサイズ : 16バイト

■ 総記憶容量

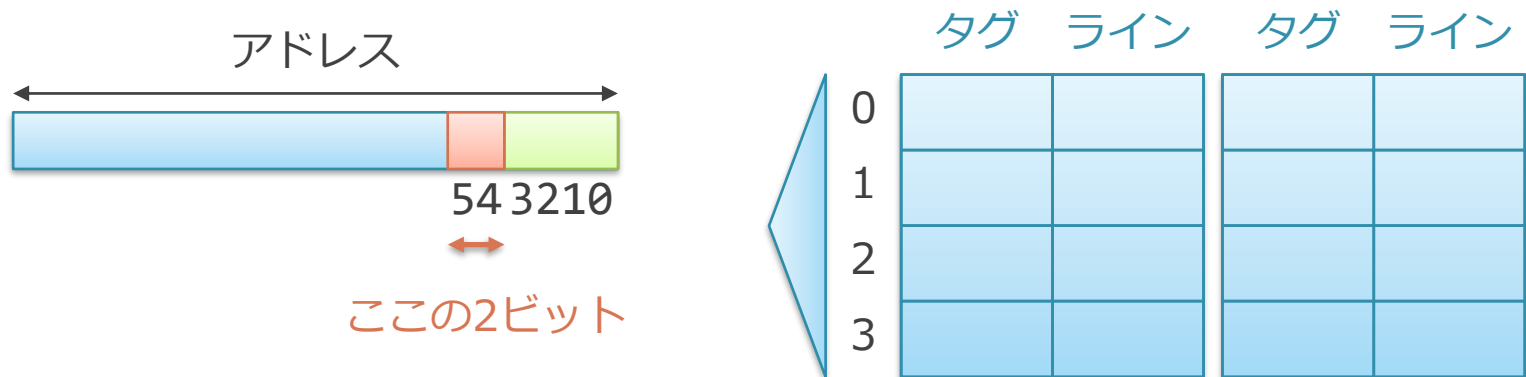
- 連想度 2 × セット数 4 × ライン 16 バイト = 128 バイト

アドレスとラインの対応



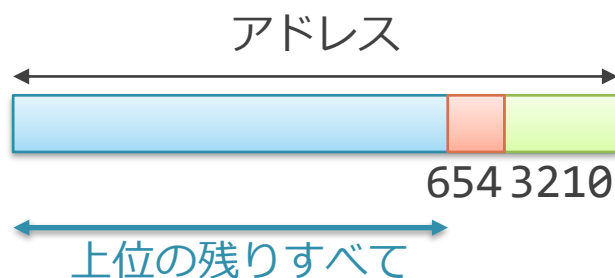
- アドレスは1バイト単位でメモリの位置を表すものとする
- 最下位ビット 0 ~ 3 （計 4 ビット）
 - 最下位部分がライン内の位置に対応
 - 空間局所性を利用するために連続した 16 バイトが 1 ラインに
 - 4ビットなのは, ラインサイズが16バイトだから
 - $2^4 = 16$
 - (ラインサイズは必ず 2 の累乗になる)

アドレスとセットの対応



- ライン部分の上位にあるビット 4 ~ 5 （計2ビット）
 - この部分を使って、どのセットにアクセスするか決める
 - 2ビットなのは、セット数が4だから
 - $2^2 = 4$
 - セット数も必ず2の累乗になる
- アドレスのこの部分はなるべくばらけた方がよい
 - 同じセットにアクセスがいかず、競合がおきにくくなる

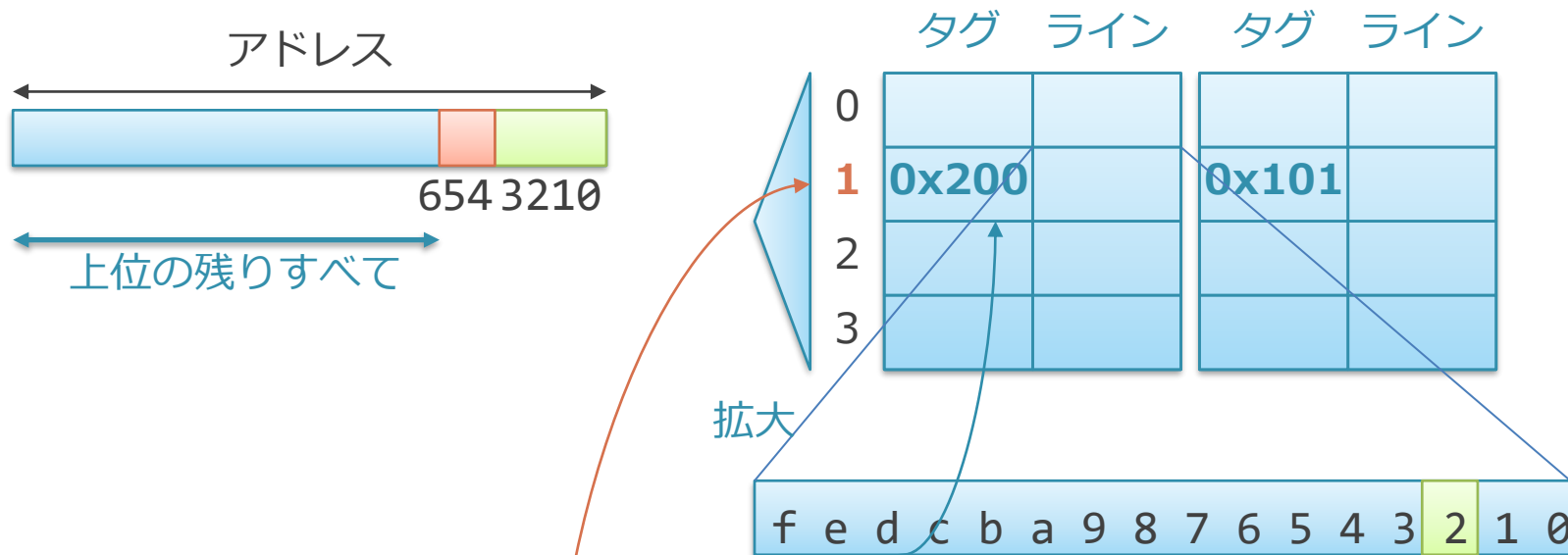
アドレスとタグの対応



	タグ	ライン	タグ	ライン
0				
1				
2				
3				

- 残りの上位のビットがタグとなる
- タグにはセット（赤）やライン（緑）の部分は入れないでよい
 - あるセットにアクセスするアドレスは、赤部分は常に一定だから
 - セット 1 にアクセスする場合、赤部分は絶対 01
 - 緑部分はラインの中の位置を表すので、関係ない

アクセス時の動作の例



- アドレス0x8014 (1000 0000 0001 0010) へのアクセスがあった場合
 - ライン内位置 : 2 (0010)
 - セット位置 : 1 (01)
 - タグ : 0x200 (1000 0000 00)
- セット1の左側のエントリにタグ 0x200 があるのでヒット
 - ライン内の2バイト目にアクセス

キャッシュの詳細のまとめ

- 基本的な構造と各方式について
 - セット・アソシアティブ方式
 - ライン単位での管理
- アドレスとキャッシュ構造の具体的な対応関係

行列積での動作例

内容

1. キャッシュの構成方法
- 2. 行列積での動作例**

キャッシュによる性能変化の例：密行列積

■ 密行列積

- ディープ・ラーニングも、実際の計算はひたすら行列積をやっている事が多い
- google の TPU は行列積超特化計算機ともいえる
 - TPU: Tensor Processing Unit
 - 機械学習に特化したハードウェア

■ 行列積はものすごい時間がかかる

- 行列のサイズの三乗に比例して演算が必要
- なんも考えないとキャッシュにもうまく乗らない

目次

1. 背景：
 1. 行列の二次元配列による表現
 2. 二次元配列のメモリ配置
2. 行列同士の乗算

行列の2次元配列による表現

```
uint32_t A[2][2];
```

$$\begin{bmatrix} A[0][0], A[0][1] \\ A[1][0], A[1][1] \end{bmatrix}$$

■ $A[y][x]$ の場合 :

- 1次元目 (x) : 何列目か
 - x が増えると参照位置が右に移動
- 2次元目 (y) : 何行目か
 - y が増えると参照位置が下に移動

2次元配列のメモリ上の配置

アドレス (uint32_t は 32bit=4バイトなので, 4飛ばしになる)

0	A[0][0]
4	A[0][1]
8	A[0][2]
	⋮
124	A[0][31]
128	A[1][0]
132	A[1][1]
	⋮
254	A[1][31]
256	A[2][0]
	⋮

```
uint32_t A[32][32];
```

- 実際のメモリは1次元の構造
 - ずっと連続して箱が並んでる
- 低次元（添え字の右側）が連続するように展開されて配置される

キャッシュ上の配置 (ラインサイズ64バイトの場合)

アドレス

0	A[0][0]
4	A[0][1]
8	A[0][2]
	⋮
124	A[0][31]
128	A[1][0]
132	A[1][1]
	⋮
254	A[1][31]
256	A[2][0]
	⋮

uint32_t A[32][32];

キャッシュ

タグ ライン

0	A[0][0], A[0][1], ... A[0][15]
64	A[0][16], A[0][17], ... A[0][31]
128	A[1][0], A[1][1], ... A[1][15]
160	A[1][16], A[1][17], ... A[1][31]
192	A[2][0], A[2][1], ... A[2][15]
...	

- 1次元目の添え字が連続した部分がライン上に
 - ラインは64Bなので, 16要素格納できる
 - 1次元目を連続にして参照すると効率が良い

配列のアクセス

- 2次元目を連続させた場合の問題
 1. ラインの利用効率が悪い
 2. コンフリクトが起きる

2次元目を連続させた場合の動作

アドレス

0	A[0][0]
4	A[0][1]
8	A[0][2]
	⋮
124	A[0][31]
128	A[1][0]
132	A[1][1]
	⋮
254	A[1][31]
256	A[2][0]
	⋮

タグ ライン

0	A[0][0] , A[0][1], ... A[0][15]
64	A[0][16], A[0][17], ... A[0][31]
128	A[1][0] , A[1][1], ... A[1][15]
192	A[1][16], A[1][17], ... A[1][31]
256	A[2][0] , A[2][1], ... A[2][15]
...	

```
for (int j = 0; j < SIZE; j++)  
    A[j][0]++;
```

- 2次元目を連続にしてアクセスした場合
 - 赤字の部分がアクセスされる

2次元目を連続させた場合の問題（1）

タグ ライン

0	A[0][0] , A[0][1], ... A[0][15]
64	A[0][16], A[0][17], ... A[0][31]
128	A[1][0] , A[1][1], ... A[1][15]
192	A[1][16], A[1][17], ... A[1][31]
256	A[2][0] , A[2][1], ... A[2][15]
...	

```
for (int j = 0; j < SIZE; j++)  
    A[j][0]++;
```

■ 問題 1 :

- **ラインの先頭しか使われない**
- A[0][1] から A[0][15] もキャッシュに勝手に乗るが使われない

2次元目を連続させた場合の問題（1）

タグ ライン

0	A[0][0] , A[0][1], ... A[0][15]
64	A[0][16], A[0][17], ... A[0][31]
128	A[1][0] , A[1][1], ... A[1][15]
192	A[1][16], A[1][17], ... A[1][31]
256	A[2][0] , A[2][1], ... A[2][15]
...	

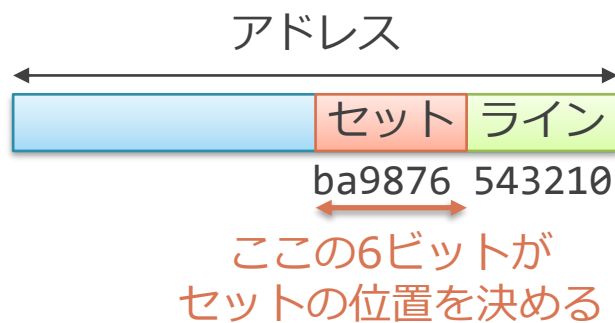
```
for (int j = 0; j < SIZE; j++)
```

```
    A[j][0]++;
```

■ 問題 2

- **アドレスが等間隔になる**
 - 0, 128, 256 ...
- 間隔は、配列の1次元目のサイズに比例
 - 今回は32要素×4 = 128 が間隔に

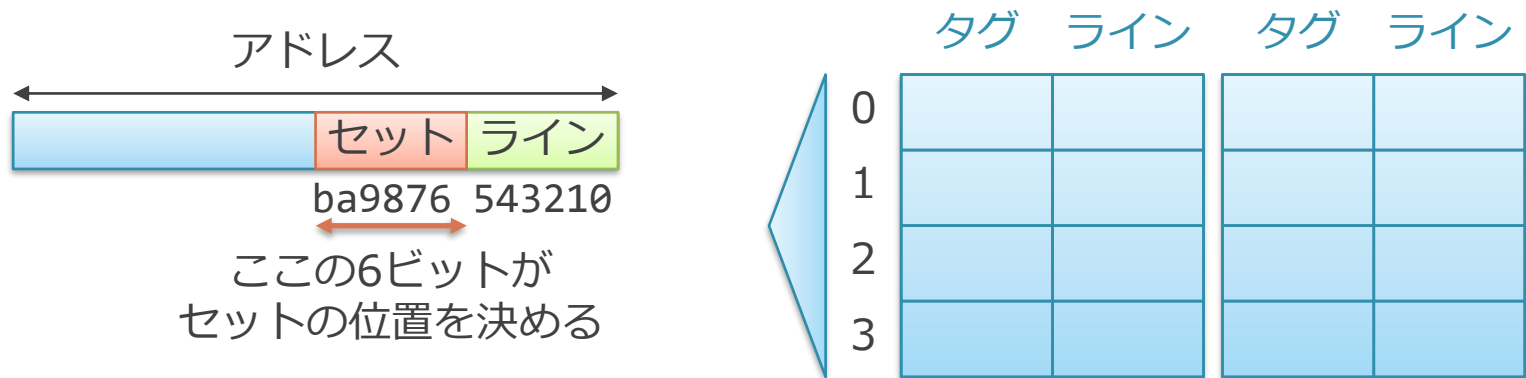
アドレスとセットの対応の復習



	タグ	ライン	タグ	ライン
0				
1				
2				
3				

- ライン部分の上位にあるビット 6 ~ b（計6ビット）
 - この部分を使って，どのセットにアクセスするか決める
- L1キャッシュのセット数部分は6ビットある
 - 32KB, 64バイトライン, 8-way
 - $32768 / 64 / 8 = 64 = 2^6$

大きな二次元配列で、2次元目を連続にすると

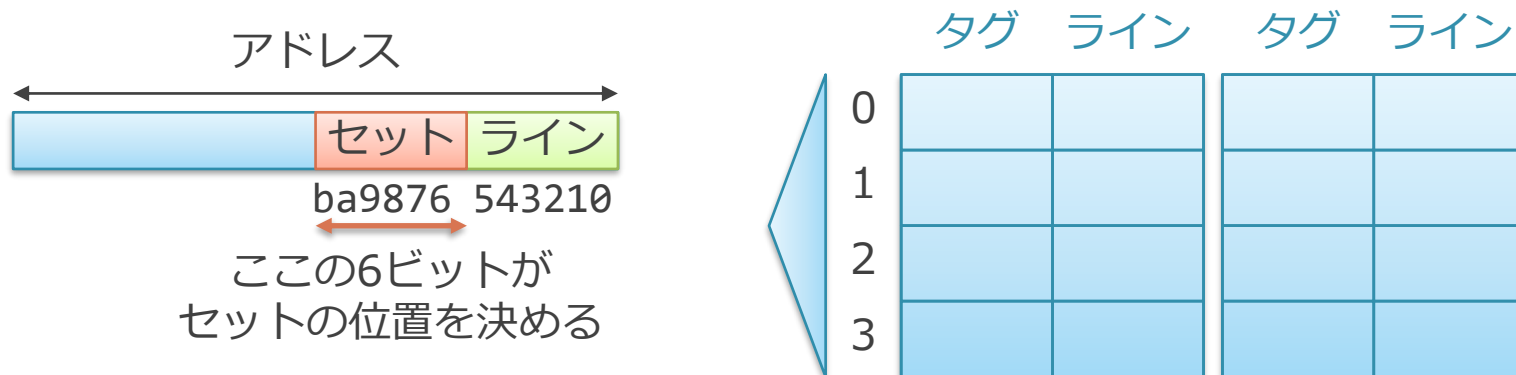


```
uint32_t A[1024][1024];  
for (int j = 0; j < SIZE; j++)  
    A[j][0]++;
```

■ アドレス：

- $A[0][0]$: 0,
- $A[1][0]$: 4096
- $A[2][0]$: 8192
- $1024\text{要素} \times 4\text{B} = 4096 = 2^{12}$ ごとにアクセス

アドレスが等間隔になるとどうなるか



```
uint32_t A[1024][1024];  
for (int j = 0; j < SIZE; j++)  
    A[j][0]++;
```

- 何がまずいのか：セット位置を決める部分が全部一定に
 - 0: 0000000000000000
 - 4096: 0100000000000000
 - 8192: 1000000000000000
- 大きな二次元配列で二次元目を連続にすると，連想度分ぐらいしかキャッシュできない

行列と二次元配列のまとめ

■ 構造

- 行列は二次元配列として表限
- 二次元配列は、1次元目が連続するよう展開される

■ 二次元目を連続させるとやばい

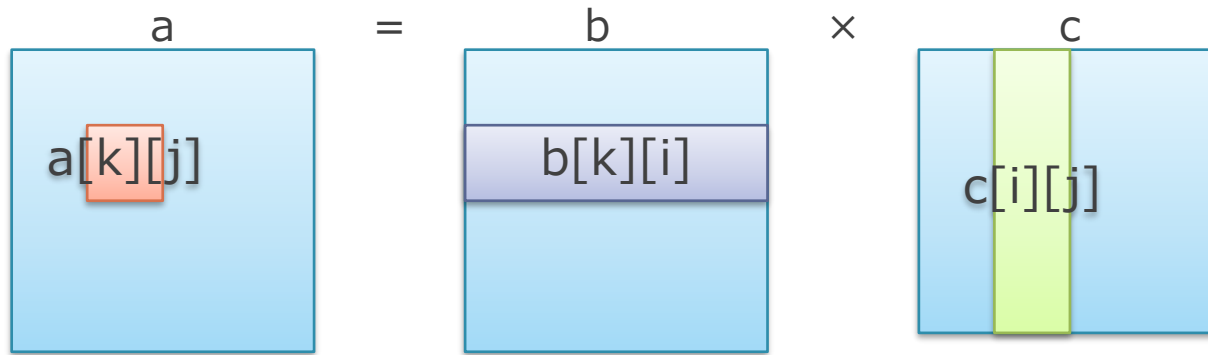
- ラインの利用効率が悪い
- 大きな二次元配列ではアドレスが等間隔に
 - コンフリクトが起きてキャッシュがほとんど利用できない

基本的な行列積の実装

```
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];  
        }  
    }  
}
```

- 三重ループとして実現できる

行列積の動作イメージ



```
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];  
        }  
    }  
}
```

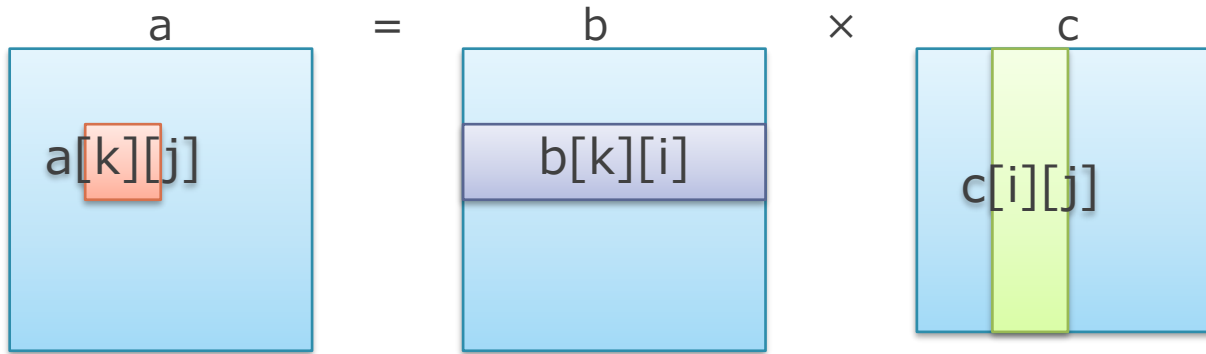
- $a[k][j]$ は, b の k 行目 (紫) と, c の j 列目 (緑) の各要素を乗算して累積することにより求まる
 - 一番内側の i はこの各要素を参照するために回る

重要なポイント

```
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];  
        }  
    }  
}
```

- このプログラムをよく見ると,
 - $a[k][j] +=$ の部分の計算の順序は自由に入れ替え可能
 - 足し算はどのような順序でやってもよい
 - たとえば, ループの外側と内側を入れ替えても, 結果は同じ

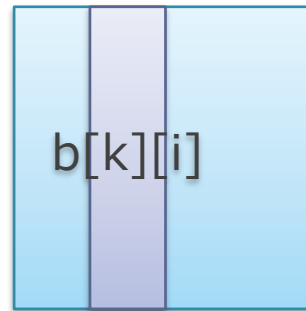
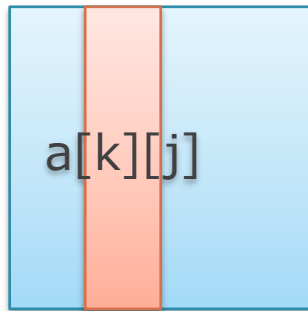
i, j, k をひっくり返した時の、 最内周ループのアクセス範囲



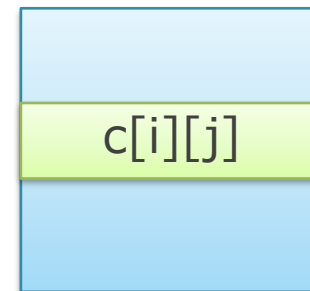
```
for (int k = 0; k < SIZE; k++) {  
    for (int j = 0; j < SIZE; j++) {  
        for (int i = 0; i < SIZE; i++) { // i が変化
```

- 最内周ループのアクセス範囲が横向きになっているのが重要

最悪の場合（1100秒）と最良の場合（20秒） 上側はキャッシュを全く利用できていない



```
for (int i = 0; i < SIZE; i++) {  
    for (int j = 0; j < SIZE; j++) {  
        for (int k = 0; k < SIZE; k++) { // k が変化
```



```
    for (int k = 0; k < SIZE; k++) {  
        for (int j = 0; j < SIZE; j++) { // j が変化
```

まとめ

■ キャッシュの構成方法

- 3つの方式
 - 基本的な構造（フルアソシアティブ方式）
 - ダイレクトマップ方式
 - セット・アソシアティブ方式
- 性質
 - 連想度によって分類可能
 - ヒット率と複雑さにトレードオフ
- ライン単位での管理
- アドレスとキャッシュ構造の具体的な対応関係

■ 行列積での動作例

課題 10

- アドレスの幅が 16 bit, ラインサイズ8B, 4 エントリのキャッシュについて考える
- 連想度を以下の様に変えた場合に,
 - 1 (ダイレクトマップ)
 - 2
 - 4 (フルアソシアティブ)
- 以下のようなアドレスによる 1B のアクセスがあった場合を考える
 1. 0x8000, 0x8001, 0x8002, 0x8003, 0x8000, 0x8001, 0x8002, 0x8003
 2. 0x8000, 0x9000, 0xA000, 0xB000, 0x8000, 0x9000, 0xA000, 0xB000
 3. 0x8000, 0x9001, 0x8002, 0x9003, 0x9004, 0xA005, 0x9006, 0x8007

課題 10

- (1) 上記それぞれの場合で、アクセスが全て終わった後のキャッシュの状態（タグの中身）を示せ
 - 4 エントリのタグにそれぞれ何が残っているかを、
連想度3パターン×アクセス系列3パターン= 9 パターン分答える
- (2) 上記それぞれの場合のヒット率を計算せよ
- (3) 各アクセスにおけるヒット時に、それが空間的局所性と時間的局所性のいずれによるのかを分類して答えよ
- 多少多いかもですが、
 - 途中までしか出来なくても良いです
 - 試験までには1回解いておくの良いです
 - 実は (1) がちゃんとできれば (2) と (3) はおまけみたいなものです

提出方法

■ 以下を提出：

1. 課題 1 0：

- 提出は Moodle の「課題 1 0」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

- Moodle に設定した締め切りまで（7/9 日曜日の 23:59 頃、要確認）

■ 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

休講について

- 7/10 は出張のため休講です
 - 修士の学生さんの発表で海外に行ってきます
- 7/17 は祝日

期末試験について

- 8/7 予定
 - 7/31 は本務校の業務のため休講になる可能性あり
 - ただ、業務が入るかどうかが直前まで確定せず
 - 期末試験の日程が直前まで未定なのはさすがにまずいので
- A4 裏表 1 枚 手書きのみの持ち込み可
- 基本的に課題で出した部分を中心に出题する予定

練習問題について

- すいません、用意したいと思っておりますが、まだ時間がとれてないです
- 基本的には課題で出した問題の、パラメータが違うものを用意したいと思います
- もしもですが、問題作って投げてくれる人がいれば、解答の確認をこちらでやって公開したいです

質問とか感想

質問とか感想

- キャッシュの考え方がシステムプログラミングで習ったバッファリングに似てるなと思いました
- また、キャッシュやバッファリングに限らず、大きなメモリからプログラムを実行する際に動作を速くするために何か別の場所にデータをコピーしてすぐに何度も使えるようにするという考え方は重要だと感じました。

質問とか感想

- 期末試験は持ち込み可だとありがたいです。毎授業内容が新しく重いですが復習しつつ頑張ります。
- 試験は持ち込みありがたいです！

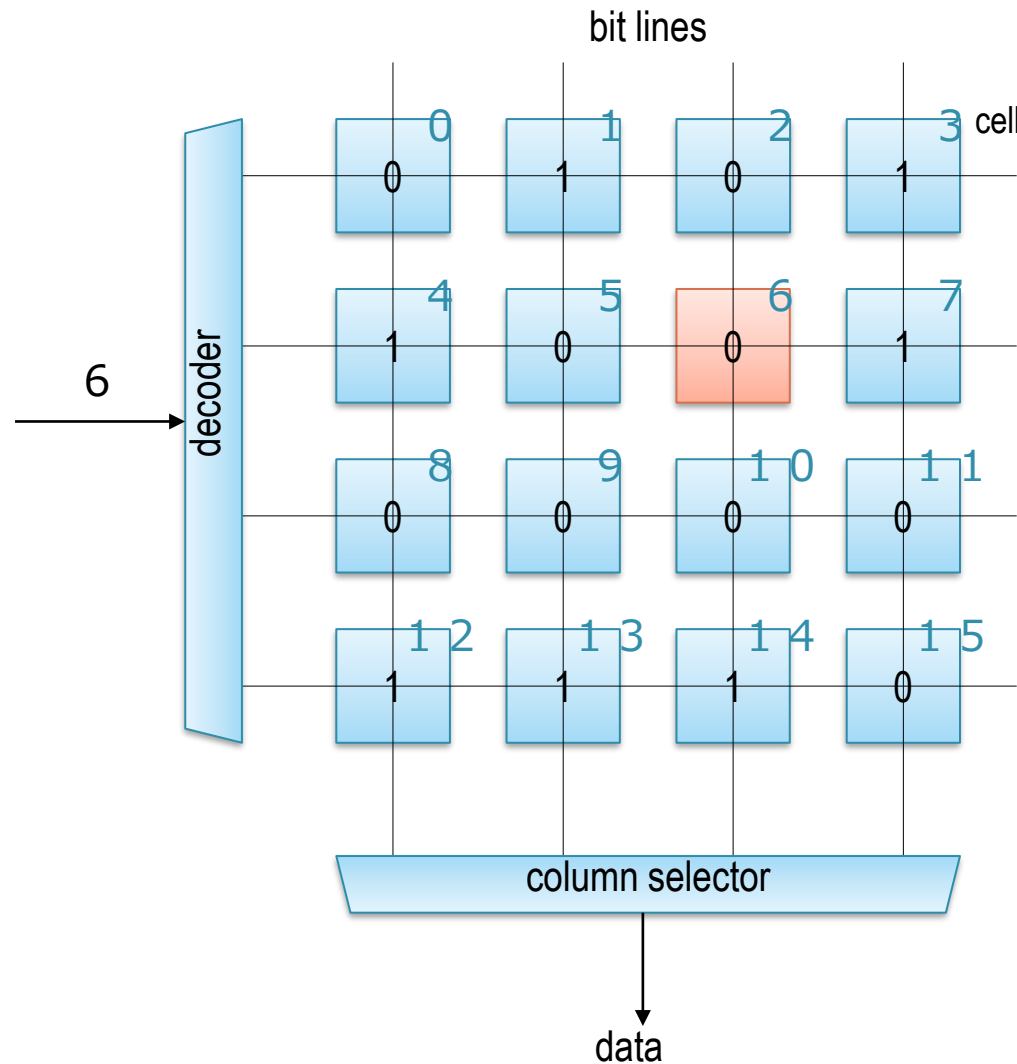
- スライド26でアドレスを4で割って切り捨てた行目を読むと書いてありましたが、アドレス6は2行目にあるように見えました。0行目から数えるものなのでしょうか。
- メモリの読み出し動作の部分で各列のうちアドレスを4で割ったあまりが読みだすべき場所になるというのが分かりませんでした。行の場合はアドレス6が4で割ったあまりが2だから、上から2行目を取り出してきたと思うのですが、列の場合は数が小さい左からではなく右から2番目ということですか？
 - 行はあまりではなく、割った結果の切り捨てです

質問とか感想

- メモリの読み出しの際、アドレスを4で割って切り捨てた行目が読むべき場所、とありましたがよくわからなかったです。各行を数えるときに0行目から始めるということでしょうか？
- メモリの読み出し動作の部分で、アドレス6のセルを読み出すためにアドレスを4で割って切り捨てた行目が読むべき場所という部分が分からなかったのもう一度説明していただきたいです。また、メモリのアドレスは必ず左上から0 1 2 3、、、という順番になっているのでしょうか？
 - マス目の右上に書いてあるのがアドレスです
 - // を割って切り捨てだとして、
 - $0 // 4 = 0, 1 // 4 = 0, 2 // 4 = 0, 3 // 4 = 0$
 - $5 // 4 = 1, 6 // 4 = 1, 7 // 4 = 1, 3 // 4 = 1$

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

- 10進数で4以上の数字は2進数で3桁以上になる性質を利用して、2進数の最初の2桁が行、後ろの2桁が列、としていることに気がついて感動しました。

質問とか感想

- 質問や感想に答えるスライドに関して、その質問に対する回答も軽くで良いので一緒に載せて欲しいと思いました。後から試験勉強などで資料を見返した時にも理解できるようにしたいです。
- 結構準備がつらいので、ここに回答まで書くのは特に重要というものにしぼってますが、努力します

- 試験勉強のためにも、期末テストを想定した小テスト（練習問題）の様なものが欲しいです。

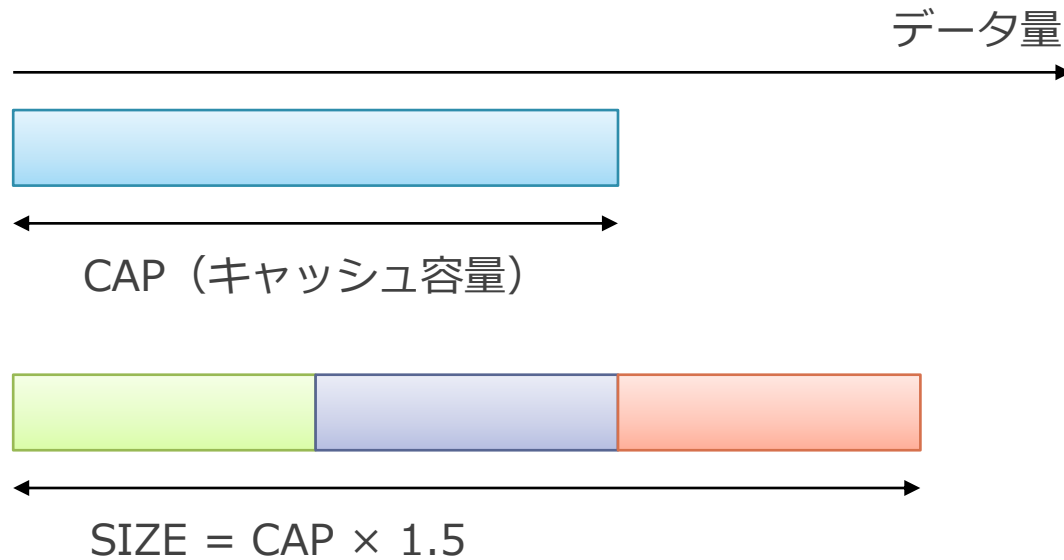
■ キャッシュの性能について

$$\text{CAP} < \text{SIZE} \leq \text{CAP} \times 2$$

の時、緑の分は上書きされて残らないということはわかりますが、次回は紫の部分のみヒットするという部分が理解できませんでした。赤の部分はヒットしないのでしょうか。

- すいません、ここの説明は間違いがありました
- 「 $\text{SIZE} \leq \text{CAP}$: 定速 (速い)」は正しいですが、それ以降はちょっと説明が間違っており (説明に書いてない仮定が加わっている) , 訂正を準備しています

CAP < SIZE ≤ CAP×2 : 徐々に遅くなる



- 左から順に（緑，紫，赤）アクセスすると...



ループ後のキャッシュ上のデータ

- 終了後には緑の分は上書きされて残らない

質問とか感想

- テストできれば7月31日だとありがたいです(;_;)
- テスト日程は変更の予定はありませんか？テストが3週間に渡ることになり(週に1つずつ)、なかなか厳しいというのが本音です。前倒しの方向で検討していただけたら、とても嬉しいです。
- テストについてなのですが、微積のテストが7/28と決定しており、一週間以上間が空いてしまい、3週間もテストになってしまうので、前にずらしてほしいというのが本音です。
- 試験の日程が 8/7 かつ A4 両面持ち込み可、とてもありがたいです！！ぜひその方向でよろしくお願いいたします。

質問とか感想

- 課題の解き方をしっかり理解していれば試験は問題ないでしょうか？
- どんな感じの問題が出るかわからなくて不安すぎます🌀課題とおなじくらいのレベルですか？
 - とりあえず課題は解けるようになっててもらえればと思います

質問とか感想

- テストは練習問題と似たような問題だけが出るなら持ち込みはなくても大丈夫だと思います。
- 言葉を問う問題があるなら暗記が多い（と思う）ので持ち込みがあると助かります
- 課題の解き方をしっかり理解していれば試験は問題ないでしょうか？

- 解説を聞いて、なぜ先週解けなかったのか不思議に思うほどすんなり入ってきました。でもテストは凄く不安です。
SRAM, DRAMは昨年度習ったことを思い出しました。

質問とか感想

- スマホアプリの設定にある、「キャッシュを削除」の意味を理解することができました。メモリのキャッシュを削除するわけではなく、アプリ内で一度読み込んだものを二度目からは早く読み込むためのキャッシュを削除していたということだとわかりました。キャッシュという言葉はよく目にするものの、きちんと理解ができていなかったため知ることができて嬉しいです。
- たまにコピーしてきたものと大元の一貫性が（バグとかで）取れなくなることがある
- そう言うときにキャッシュをクリアすると直ったりする

- キャッシュとレジスタがどちらも一時的な値の置き場として同じような役割を持つメモリという認識で、両者の違いがよくわかりません。

- キャッシュの説明のところで図がたくさんあってとても分かりやすかったです。実際の測定データについて、SIZEが大きくなってもアクセス時間が増えていない（減少している）ところがあるのはなぜでしょうか？これは誤差で、基本は逆転することはないのでしょうか。
 - 誤差であり，基本逆転はないです
 - パソコン上で測ると，他のアプリも裏でちょっとだけ同時に動いているのでどうしてもノイズがのります

質問とか感想

- 課題 8 の(2)で質問です。
ハザード時のサイクル数の増加は1ではなく1.5で計算しました。
段数が15に増えたのでそう予想したのですが、10段で1サイクル増える時と15段ではどうして同じで良いのですか。
- 前提：サイクル数は基本的に整数しか取れない
 - 全てはクロックを基準として動いているので、1.5 サイクル止める、とかはできない
- データハザードは、ある命令の結果がまだ使えない時に、ベルトコンベア全体を止めて結果が出るのをまつ

課題 8

■ 以下のような条件を考える

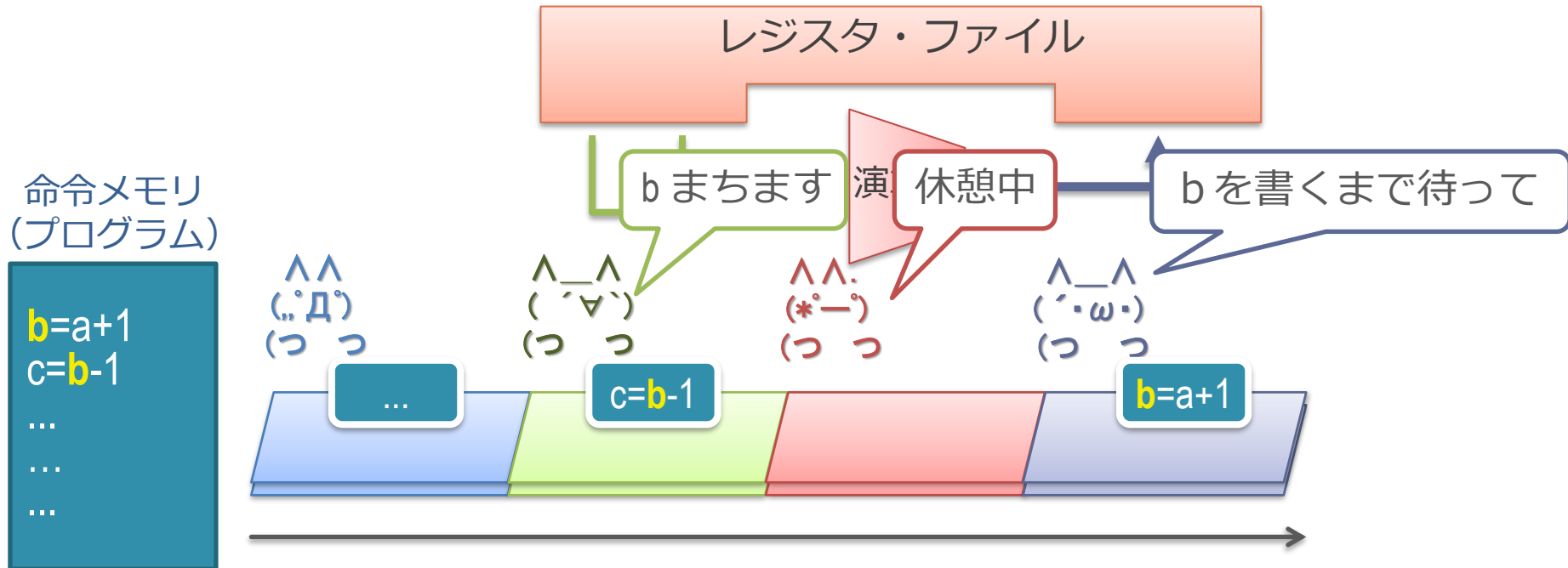
- 10段のパイプラインを持つ 2-way スーパースカラプロセッサであり, 理想的には $IPC=2$ で実行できる
- 全実行命令におけるなんらかのデータハザードの発生率は 0.2
- このデータハザード発生時は 1 サイクル実行時間が伸びるものとする
- 全実行命令における分岐命令の出現率は 0.2
- 分岐予測ミス率は 0.3

- ## ■ (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

1. ストール



■ 直前の命令の結果を使う命令が現れた場合：

- $(\cdot \omega \cdot)$ の人が計算結果をレジスタに書くまで,
 $(\cdot \nabla \cdot)$ の人と上流のラインを止める
- 間にバブル (何もしないステージ) が入る

- キャッシュとメインメモリはどのように使い分けるのですか？

- 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に時間がかかるため、周波数が 0.8 倍になるものとする、というのは、キャッシュに入っている全情報の中からほしいものを見つけるのに時間がかかるというようなことですか？

質問とか感想

- これまでの課題では、先週の計算や回路図、アセンブリ言語といった考えて解く問題！というようなものでしたが、テストでもこのような形式の問題で単語丸暗記穴埋めみたいな形式はなしですか？資料のページ数も多いですし、調べたらすぐに出てくることを丸暗記するメリットも思いつかないので。。
- 穴埋めは多分やらないと思いますが、「～という条件のもとで～の例を書け」とかはあるかもしれません

- DRAMについて、リフレッシュする前に全ての電荷が抜けてしまっ
てデータがなくなってしまうたら、どうなるのか気になった。
 - 本当に動かなくなるのでなんとかさけます

質問とか感想

- 私はキャッシュが溜まっているといつでも操作が低速になってしまっていた（自分のスマホなどを触っていて感じる）ので、場合によって高速になることもあったと知って、キャッシュは必要な機能であったんだなと思った。
- キャッシュの作り方が良くないのであって、本来ははやくなるはず

- 前回の講義資料p61の例とキャッシュの話が頭の中で結びつきません。キャッシュがどう関係した結果処理時間が変化したのかももう少し詳しく知りたいです。

質問とか感想

- DRAMは定期的にはリフレッシュを行わなければならないため、速度ではSRAMに劣るという認識で良いのでしょうか。
- コンデンサの微妙な電荷の溜まり具合を検出するのも時間がかかるので、そこでも遅くなってます

質問とか感想

- ・ P28の最後に書かれているようなカラムセレクトがない場合には、どのようにビットを選択しているのでしょうか？全てのビットを直接データとして送っているということでしょうか。
また、カラムセレクトがない場合＝ビット数が少なくビット選択の処理をしなくてもデータが膨大にならない場合 ということでしょうか？
- レジスタファイルなどで 32エントリ × 32 bit でちょうど1行が1回の読み出し単位になっている時などはカラムセレクトなしになることもあります
- なるべくメモリが正方形に近づくようにして、行の長さが読み出し単位より長いとカラムセレクトが入ります

質問とか感想

- 課題7(7)の解答の図で、それぞれの段(FDXMW)がどの命令を表しているのかがよく分からなかったのですが、
- `li x1←1`
- `li x2←2`
- `beq x1==x2, LABEL`
- `add x2←x3+x4`
- `add x1←x2+x3`
- で合っているでしょうか。また、分岐予測により飛んだ先で実行される命令は、分岐命令のFが終わってすぐに始まるのだと思っていたのですが、もし上図の通りだとすると、分岐予測をしても、LABELに飛ぶと実行される命令`add x2←x3+x4`は、`beq x1==x2`のWが終わってから始まるのでしょうか。

課題 7 (2023/7/3訂正)

- (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

$$\text{li } x_1 \leftarrow 1$$
$$\text{li } x^2 \leftarrow 2$$

```
beq x1==x2, LABEL
```

add $x_1 \leftarrow x_2 + x_3$

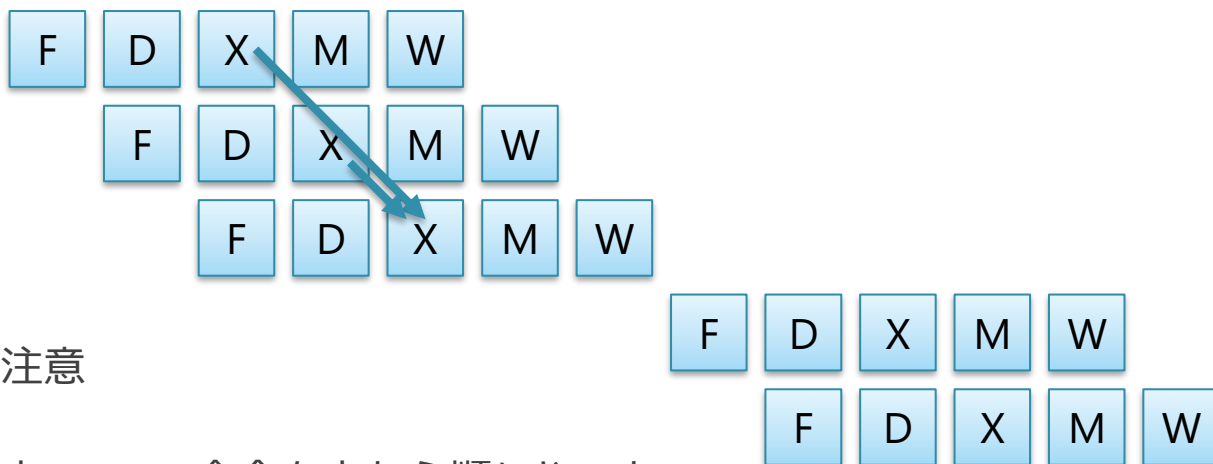
LABEL:

```
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



- あと赤ちゃんの写真可愛すぎました、美味しいお店何か教えてあげたいけど茗荷谷開発私もできてない、
 - 茗荷谷以外でもおすすめあればぜひ

質問とか感想

- 期末は範囲も広いので持ち込み可にしていきたいです。あと、手土産は和菓子ですが護国寺の方にある群林堂が有名です！
- この講義の担当になったときから楽しみにしてたんですが、月曜定休でした・・・

質問とか感想

- 自転車でお越しとのことで、東京ドームのラクーアという施設まで足を伸ばしていただくと（大学から自転車で10分程度の道のりのようです）DELI&DISHというコーナーがあり、美味しいケーキやお菓子、お惣菜のお店が集まって、小さいデパ地下のようなかたちになっています！
- 個人的にはboBというカヌレ屋さんがとても美味しかったのでおすすめです！他のお店もとても美味しそうです
- <https://www.laqua.jp/shops/list/bob/>

質問とか感想

- また、パティスリーレセンシエルというお店のシュークリームとケーキがとても美味しいのでぜひ、、、！
 - 先週行ってきました



ちなみに東大の近くだと

- TIES というお店がおすすめ
 - 本郷三丁目 or 湯島からが多分近い

