

第13回 HPC-Phys 勉強会

Dockerで体験する富岳のアーキテクチャ 「AArch64」ハンズオン

ハンズオン資料

https://github.com/kaityo256/xbyak_aarch64_handson

慶應義塾大学理工学部物理情報工学科
渡辺宙志

本ハンズオンの構成

- 事前準備編
 - Dockerイメージのビルド
 - SVEとXbyakの説明
- ハンズオン編
 - 動作確認
 - 組み込み関数編
 - Xbyak編

Dockerイメージのビルド

ハンズオン資料「ハンズオン編」 「Dockerイメージのビルド」

適当な場所でリポジトリをクローン

```
git clone https://github.com/kaityo256/xbyak_aarch64_handson.git  
cd xbyak_aarch64_handson
```

Dockerイメージをビルド(3～5分くらい)

```
cd docker  
make
```

富岳でやりたい人

ハンズオン資料「富岳実機での動作」

適当な場所(~/.github)でリポジトリをクローン

```
cd github  
git clone --recursive https://github.com/kaityo256/xbyak_aarch64_handson.git
```

インタラクティブキューに入ってXbyakのビルド

```
cd xbyak_aarch64_handson  
# ここでインタラクティブキューに入る  
cd xbyak_aarch64/  
make
```

環境変数の設定

```
export XBYAK_PATH=~/.github/xbyak_aarch64_handson/xbyak_aarch64  
export CPLUS_INCLUDE_PATH=$XBYAK_PATH  
export LIBRARY_PATH=$XBYAK_PATH/lib
```

組み込み関数はFCC、Xbyakは `g++ filename.cpp -lxbyak_aarch64` でビルドできる

富岳概要

ノード数：158976

ネットワーク：Tofu (24,23,24,2,3,2)

1CPU/1ノード

4CMG + 2アシスタントコア/1CPU

12 core/ 1CMG

ISA: Armv8.2-A + SVE

富岳概要

プログラマから見た

この辺はMPI

ノード数：158976

ネットワーク：Tofu (24,23,24,2,3,2)

1CPU/1ノード

4CMG + 2アシスタントコア/1CPU

12 core/ 1CMG

この辺はOpenMP

ISA: Armv8.2-A + SVE

ここをどうするか？

CPUコアの性能

$$\text{性能} = \text{動作周波数} \times \text{コア数} \times \text{同時命令発行数} \times \text{SIMD}$$

富岳の場合(倍精度)

$$3072\text{GF} = 2\text{GHz} \times 48 \times 4 \times 8$$

(2 x 積和)

※ 演算にはレイテンシがあるが、パイプライン処理により「理想的には」1サイクルに1演算できる(スループット)

CPUコアの性能

ここを上げたい

ここはもう無理

ここも多分無理

性能

=

動作
周波数

x

コア数

x

同時命令
発行数

x

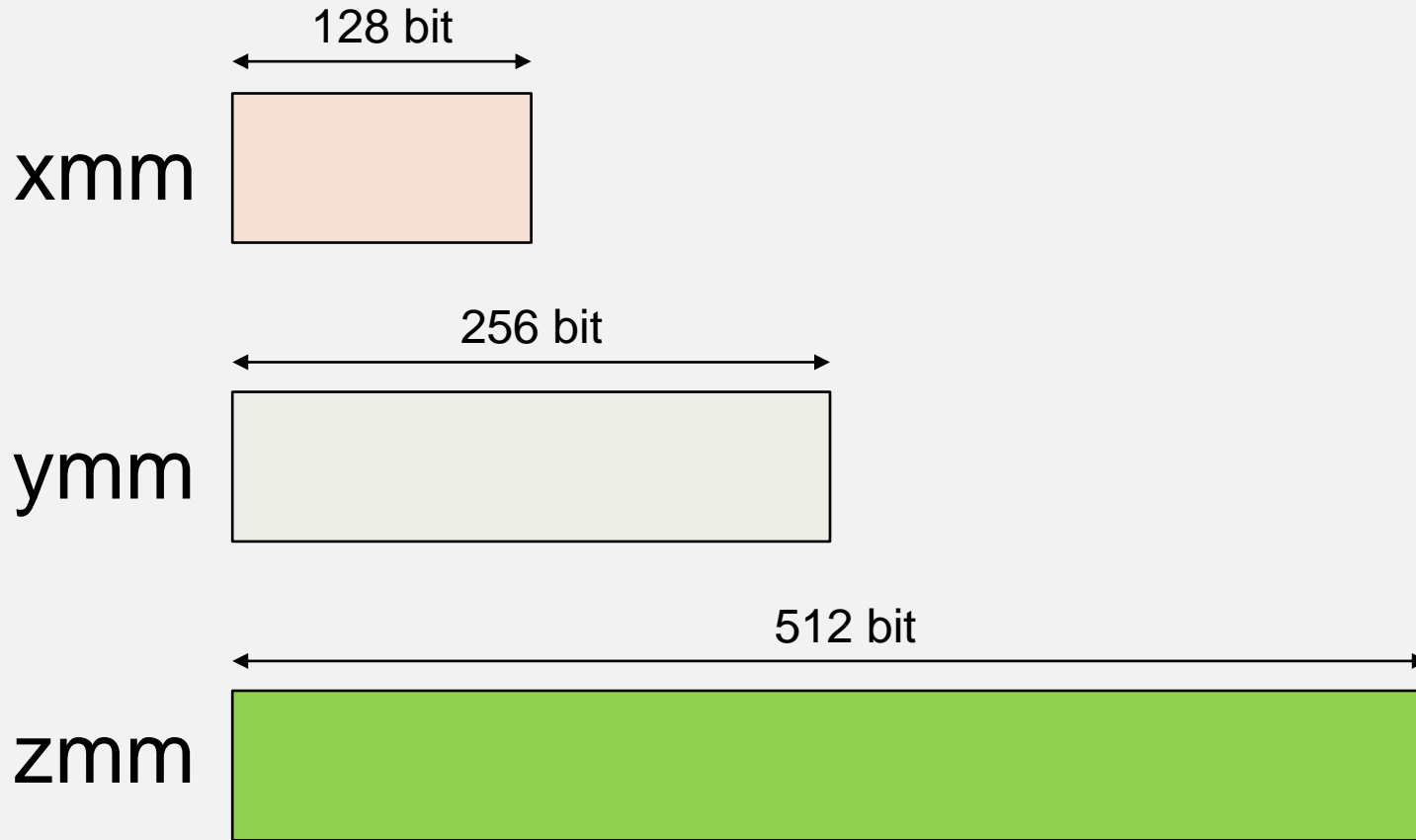
SIMD

ここを増やす
(メニーコア)

ここを増やす
(幅広SIMD)

SIMD幅を伸ばす

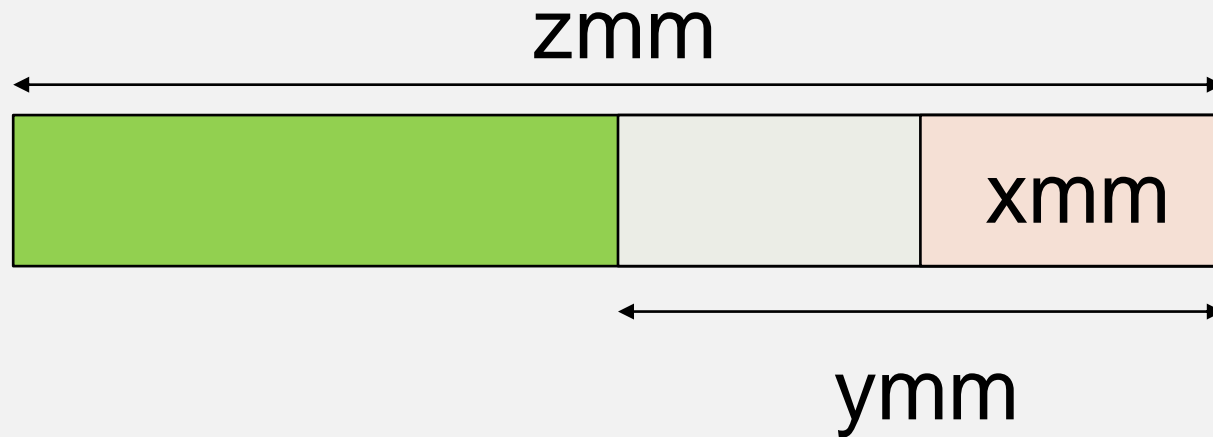
x86の場合



順調に倍々ゲームで増えてきた

SIMD幅を伸ばす

SIMD幅が伸びても下位を同じ名前でアクセスできるようにする



後方互換性を保つ



古いコードは、広くなったSIMD幅を活かせない

SIMD幅を伸ばす



SIMD幅が伸びるたびにコードを書き直し。なんとかならないかな...



SIMD幅を固定しない命令セットにすればよいのでは？

SVEとは何か？

Scalable

Vector

Extension

幅を固定しない

SIMDの

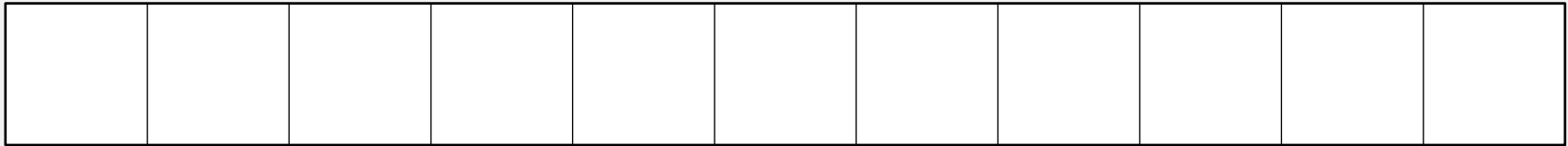
追加命令セット

特徴： Predicate-centric Approach

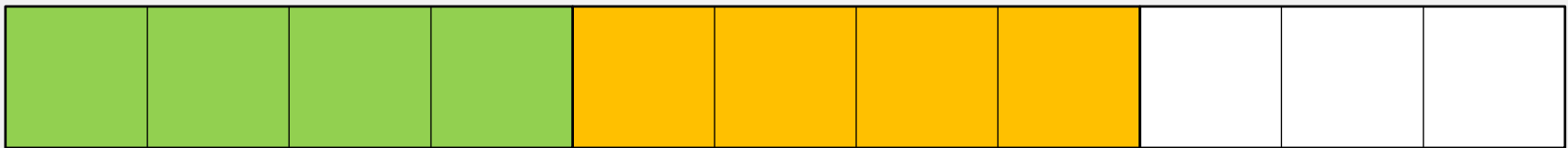
命令ごとにどの要素を使うかをマスク処理できる

マスク処理

11個のデータを4つずつ処理したい



普通にやると3個余る
余りをスカラーで回す
→ベクトル2回転+スカラー3回転



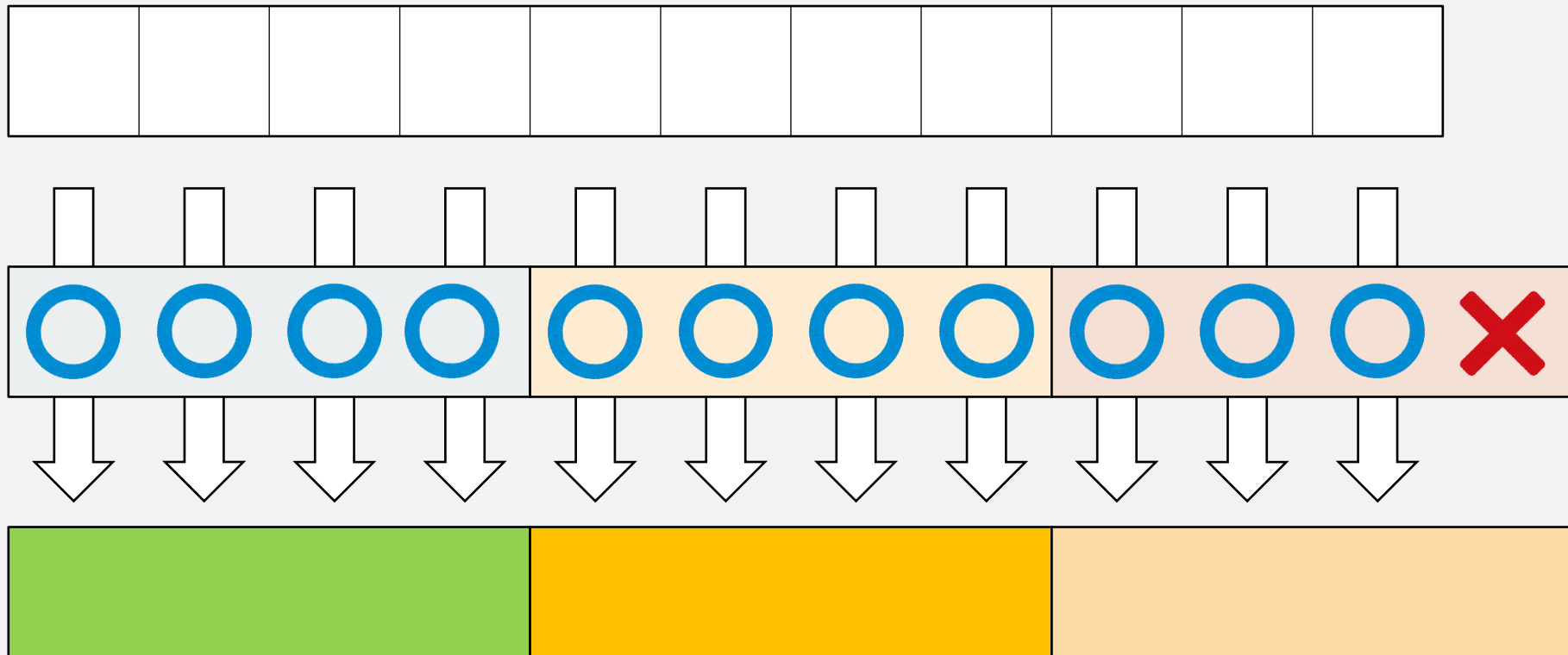
ベクトル処理

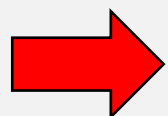
ベクトル処理

スカラー処理

マスク処理

プレディケートレジスタにより、どの要素をロードするか指定



 ベクトル3回転で済む

SVEの概要

スケーラブルなSIMD幅

スケーラブルなコードを書いておけば、将来SIMD幅が増えたハードウェアで実行した時に、その恩恵を受けることができる

Predicate-centric Approach

ほぼ全ての命令にプレディケートレジスタを指定でき、どの要素にどんな命令を実行するか細かく指定できる

SVEをどう使うか？

高レイヤ (楽だが細かい調整が難しい)

コンパイラに任せる

ディレクティブを指定する

組み込み関数で書く

Xbyakで書く

フルアセンブリで組む

低レイヤ (細かく調整できるが大変)

組み込み関数

アセンブリと一体一対対応した関数を使う

組み込み関数

アセンブリ

svcntb

cntb

svptrue_b8

ptrue p0.b, ALL

svld1_f64

ld1d

概ね「sv + アセンブリ名 + 型」という命名規則

組み込み型

svfloat64_t

レジスタにfloat64_tが詰まっているとして扱う
コンパイル時に要素数が確定しない
512ビットレジスタなら8要素

svbool_t

プレディケートレジスタを表す型
コンパイル時にビット長が確定しない

こんな感じに使う

```
std::vector<float64_t> a;  
svbool_t tp = svptrue_b64();  
svfloat64_t va = svld1_f64(tp, a.data());
```

組み込み関数を使う

Pros

- 関数の呼び出し規約を気にしなくて良い
- アドレッシングを気にしなくて良い
- レジスタ割り当てを気にしなくて良い
- コンパイラによる最適化が期待できる

Cons

- 組み込み関数以外の場所は制御できない
- コンパイラが余計なことをする場合がある

Xbyakとは

Xbyak (カイバック)はJITアセンブラ
関数単位でアセンブリで書く

作者は光成(herumi)さん

実行する命令を関数単位で実行時に作る

Intelの機械学習ライブラリoneDNNなどが利用

x86向け

<https://github.com/herumi/xbyak>

Aarch64向け



https://github.com/fujitsu/xbyak_aarch64

Xbyakの使い方

Xbyak_aarch64::CodeGeneratorを継承し、コンストラクタにアセンブリに対応したコードを並べておく

```
struct Code :  
Xbyak_aarch64::CodeGenerator {  
    Code() {  
        mov(w0, 1);  
        ret();  
    }  
};
```

テンプレートに関数のシグネチャを指定して関数へのポインタを取得
その関数を呼び出す

```
int main() {  
    Code c;  
    auto f = c.getCode<int (*)(*)>();  
    c.ready();  
    printf("%d¥n", f());  
}
```

Xbyakの動作原理

実行時にメモリを確保して、そこに実行時に命令を並べる

```
struct Code :  
Xbyak_aarch64::CodeGenerator {  
    Code() {  
        mov(w0, 1);  
        ret();  
    }  
};
```

f:

mov w0, 1
ret



その領域に実行権限をつけ、先頭アドレスを呼び出すことで関数として使う

```
int main() {  
    Code c;  
    auto f = c.getCode<int (*)()>();  
    c.ready();  
    printf("%d\n", f());  
}
```

※x86では不要だが、ARMでは実行前にready()を呼ぶ必要がある

Xbyakを使う

Pros

- 実行時の情報を使ったコード生成ができる
 - キャッシュサイズやCPUの種類
 - コンパイル時に決まらない実行時定数
- 書いた通りに動く
- 生アセンブリより書きやすい

Cons

- 関数の呼び出し規約やアドレッシング等のアセンブリの知識必須
- ローカル変数を自分で管理する必要がある
- レジスタ割り当てをする必要がある

ハンズオン編

先ほどmakeしたディレクトリでmake runすれば
Dockerの中に入ることができる

```
$ make run  
[user@291e9d9cad93 ~]$
```

xbyak_aarch64_handson/sampleにサンプルコードがある

以下でそれぞれ動作テストをする

- intrinsic/01_sve_length
- xbyak/01_test

組み込み関数編 (1)

プレディケートレジスタ (PR)

SVEのレジスタは128ビット x N

プレディケートレジスタは最低8ビット単位

→ レジスタ長は16ビット x N

512ビットならN=4なので、PRは64ビット

確認すること

1. どの型に使うかにより、立てるビットが異なる
2. 立てるパターンを指定できる
3. レジスタ長を変えて実行してみる

組み込み関数編 (1)

組み込み関数

```
svptrue_b8()  
svptrue_pat_b8(SV_ALL)
```



アセンブリ

ptrue $p_{\theta}.$ **b**, **ALL**

64bit

[illegible]

組み込み関数

```
svptrue_b16()  
svptrue_pat_b16(SV_ALL)
```



アセンブリ

ptrue p0.h, ALL

01

svpttrue_ **b32**



ptrue p0.s, ALL

svptrue_ **b32**



ptrue p0.d, ALL

組み込み関数編 (2)

レジスタへのロード

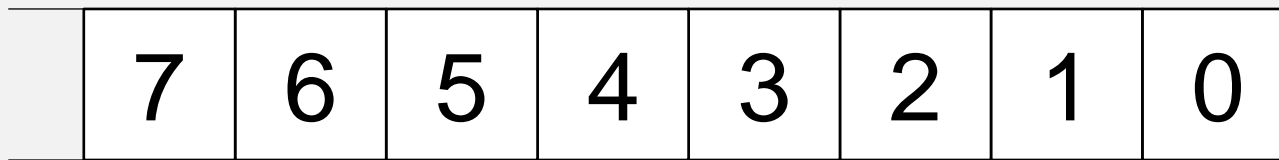
svfloat64_t型へのロードや加算を試してみる

確認すること

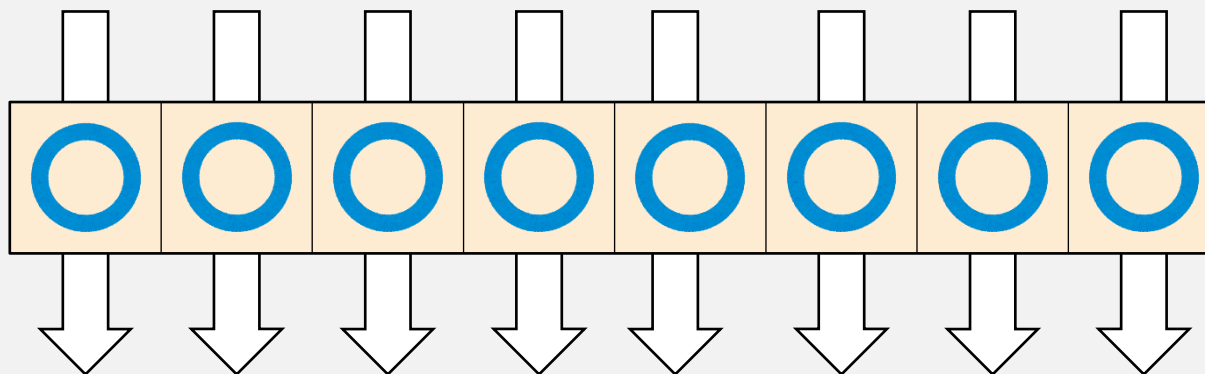
1. 指定の先頭アドレスからまとめてレジスタにロードできる
2. 一回の命令で複数要素まとめて演算できる
3. 演算にマスク処理ができる
4. inactiveな要素に対して
 1. ゼロクリアする (zeroing predication)
 2. 第一引数透過 (merging predication)

組み込み関数編 (2)

```
double a[] = {0, 1, 2, 3, 4, 5, 6, 7};  
svfloat64_t va = svld1_f64(svptrue_b64(), a);
```



メモリ



プレディケート
レジスタ



SVレジスタ

組み込み関数編 (2)

そのまま全部足す

7	6	5	4	3	2	1	0	va
---	---	---	---	---	---	---	---	----

○	○	○	○	○	○	○	○
---	---	---	---	---	---	---	---

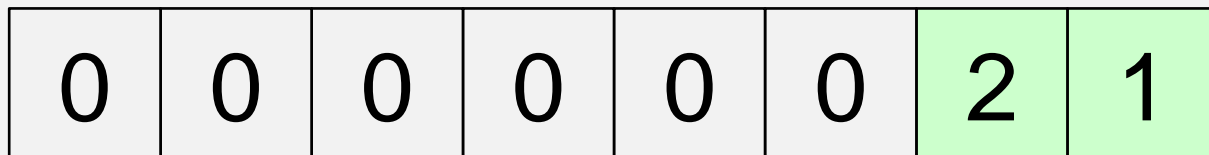
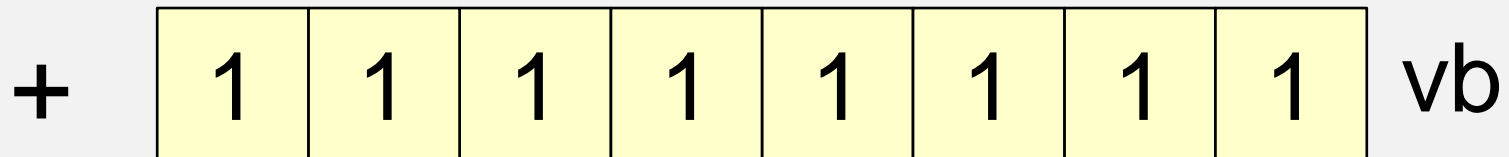
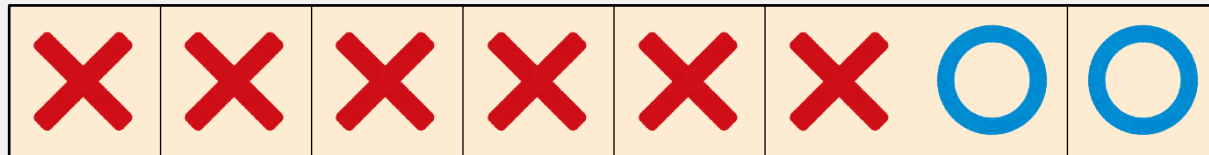
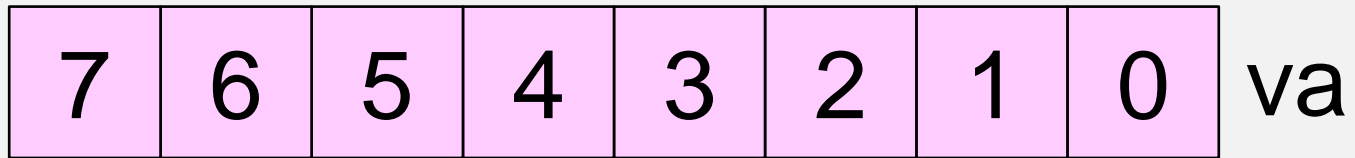
+	1	1	1	1	1	1	1	vb
---	---	---	---	---	---	---	---	----

8	7	6	5	4	3	2	1
---	---	---	---	---	---	---	---

組み込み関数編 (2)

Inactiveな場所はゼロクリア (zeroing predication)

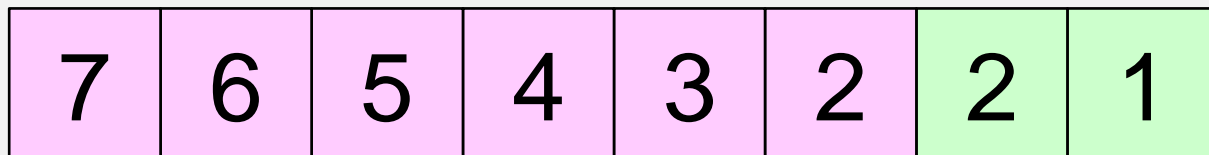
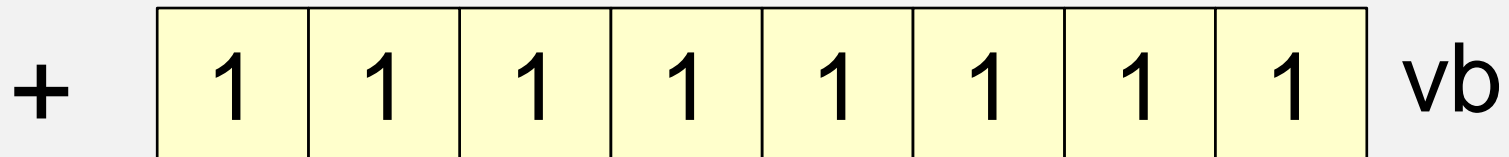
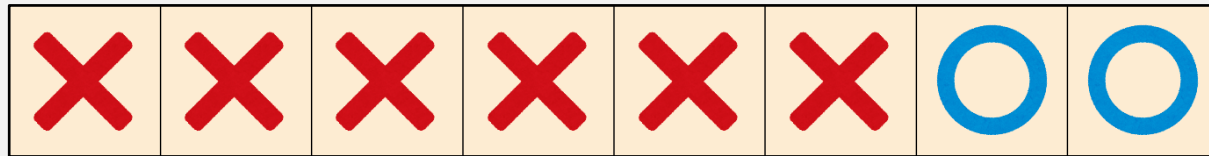
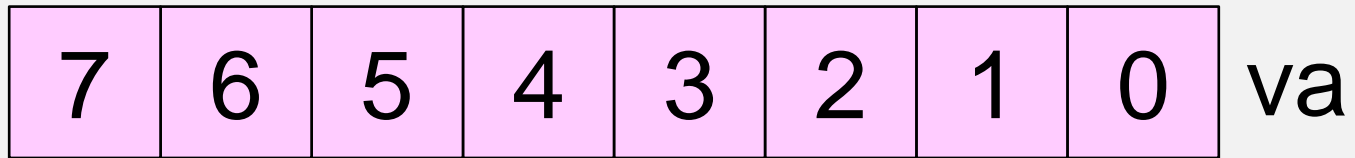
svadd_f64_**z**(svptrue_pat_b64(SV_VL2), va, vb)



組み込み関数編 (2)

Inactiveな場所は透過 (merging predication)

svadd_f64_m(svptrue_pat_b64(SV_VL2), va, vb)



Xbyak編(1)：呼び出し規約

ABI (Application Binary Interface)が定めるものの一つ
関数を呼び出す時、引数をどうやって渡すか、戻り値を
どう返すかを定める

こんな関数を作りたい

```
int f(int i){  
    return i + 1;  
}
```

Xbyakではこう書く

```
struct Code :  
Xbyak_aarch64::CodeGenerator {  
    Code() {  
        add(w0, w0, 1);  
        ret();  
    }  
};
```

整数の第一引数がレジスタw0に渡され、戻り値を
w0に入れてretすることを知っている必要がある

Xbyak編(2)：ダンプの確認

Xbyakのコードは動的に作られるため、実行時までアセンブリがわからない
→ 実行時のコードをダンプし、逆アセンブルすることでデバッグする

```
#include <cstdio>
#include <xbyak_aarch64/xbyak_aarch64.h>

struct Code : Xbyak_aarch64::CodeGenerator {
    Code() {
        mov(w0, 1);
        ret();
    }
    void dump(const char *filename) {
        FILE *fp = fopen(filename, "wb");
        fwrite(getCode(), 1, getSize(), fp);
    }
};
```

ファイル名を受け取り
機械語バイナリを保存

Xbyak編(2)：ダンプの確認

```
int main() {  
    Code c;  
    auto f = c.getCode<int (*) (int)>();  
    c.ready();  
    c.dump("xbyak.dump");  
    printf("%d¥n", f(10));  
}
```

ここでダンプをxbyak.dump
という名前で保存

実行するとxbyak.dumpができるので、objdumpで逆アセンブル

```
$ aarch64-linux-gnu-objdump -D -maarch64 -b binary -d xbyak.dump
```

長いので xdump にaliasをはってある

```
0000000000000000 <.data>:  
0: 52800020      mov     w0, #0x1      // #1  
4: d65f03c0      ret
```

Xbyak編(3) : FizzBuzz

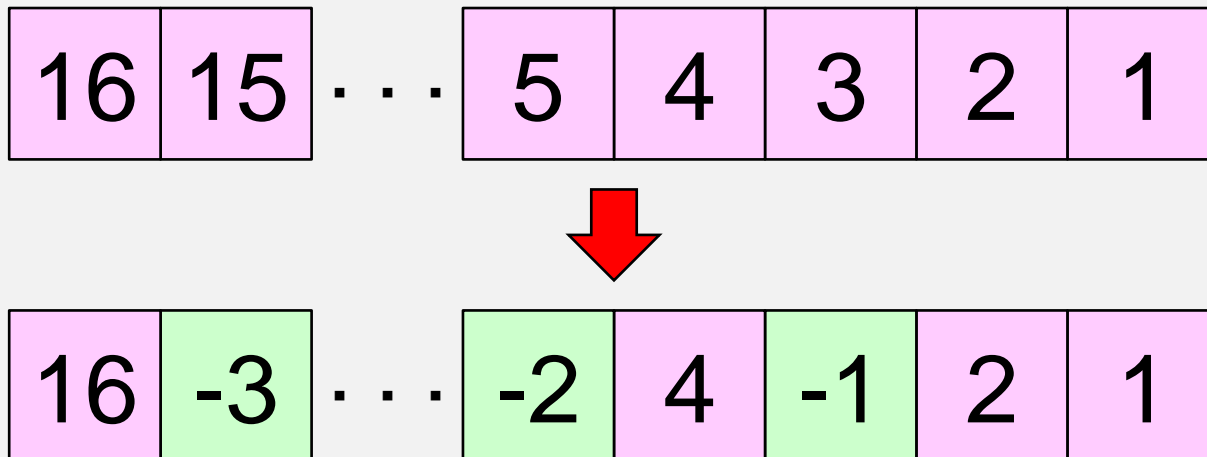
与えられた配列の要素が

3の倍数なら-1

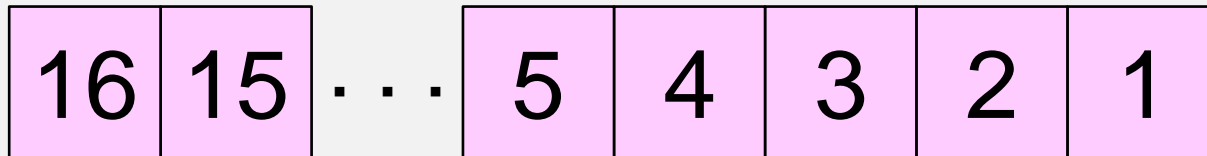
5の倍数なら-2

15の倍数なら-3

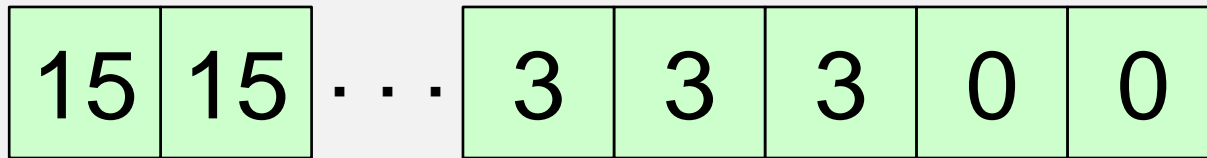
で上書きする



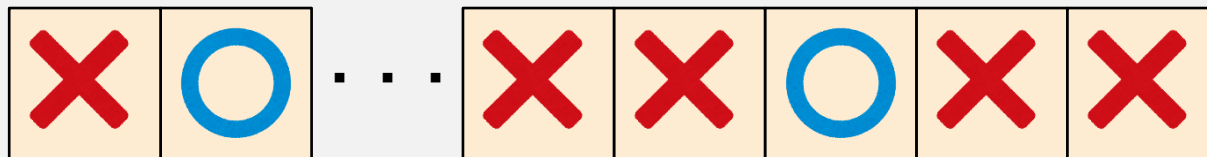
Xbyak編(3) : FizzBuzz



3で割って3をかける

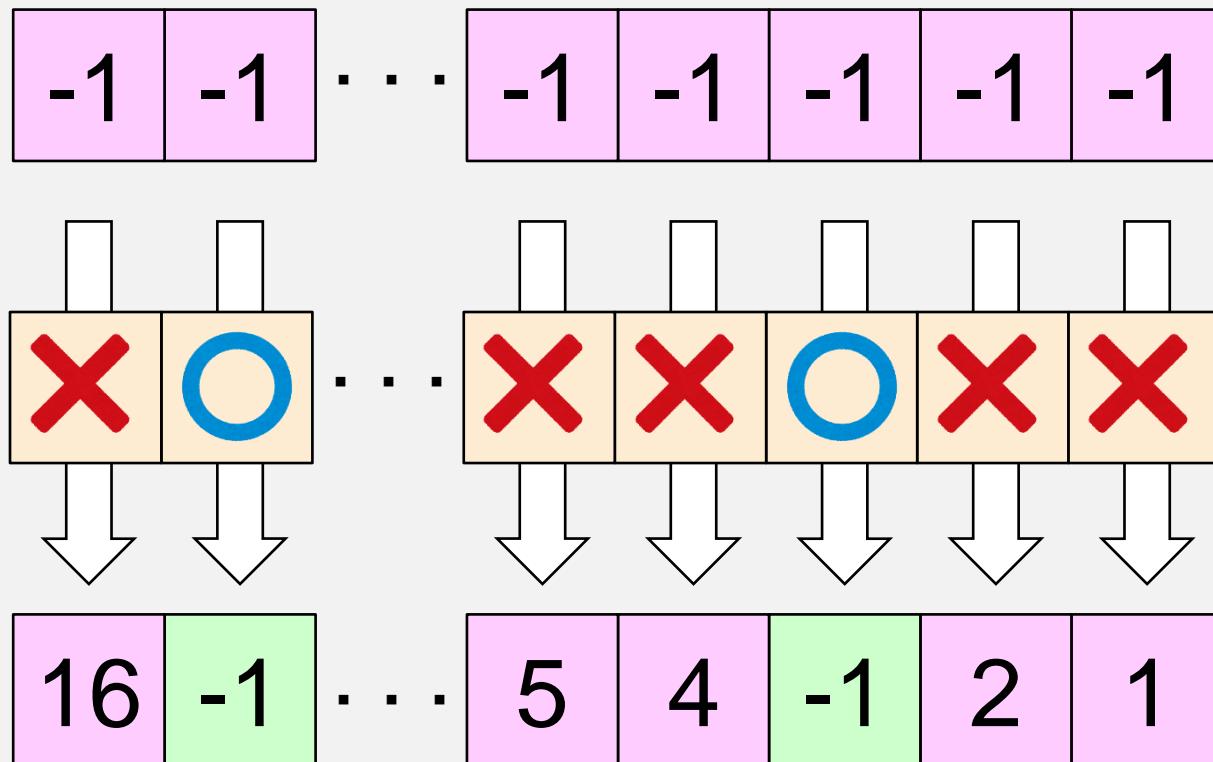


等しい場所にフラグを立てる



Xbyak編(3) : FizzBuzz

作成したマスクを使って書き戻し(store)



5の倍数も同様

15の倍数は、3の倍数マスクと5の倍数マスクのANDをとる