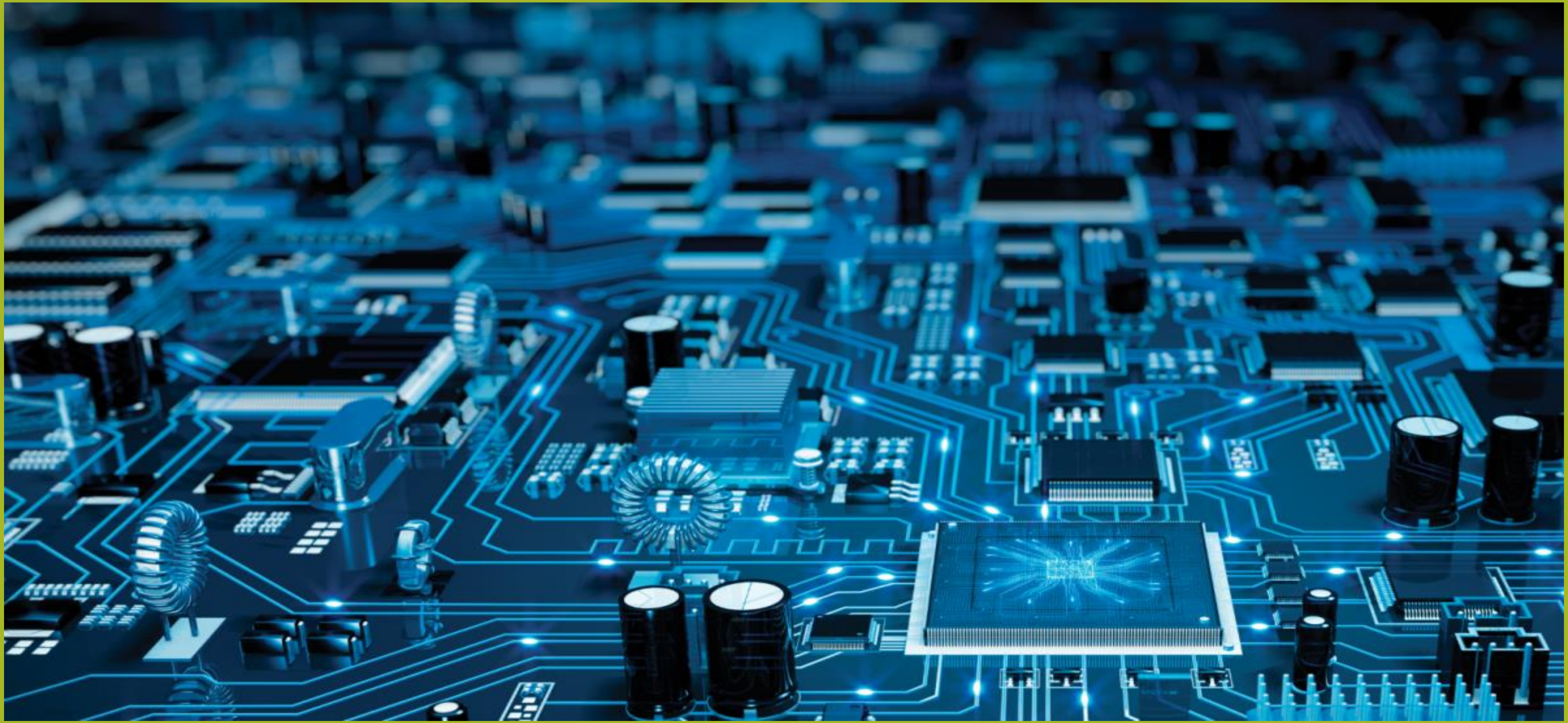


Vectorization

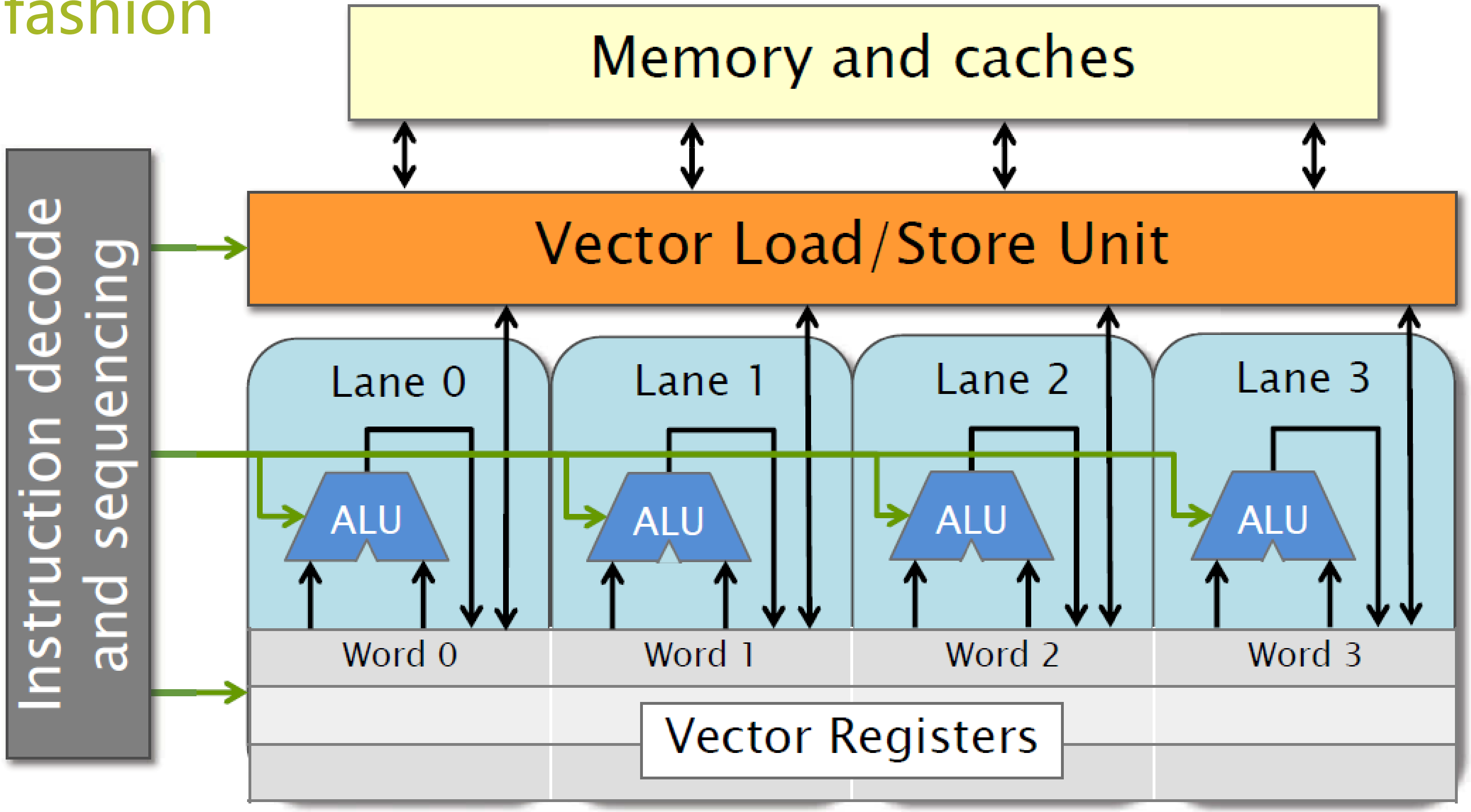
黄 波
bhuang@dase.ecnu.edu.cn

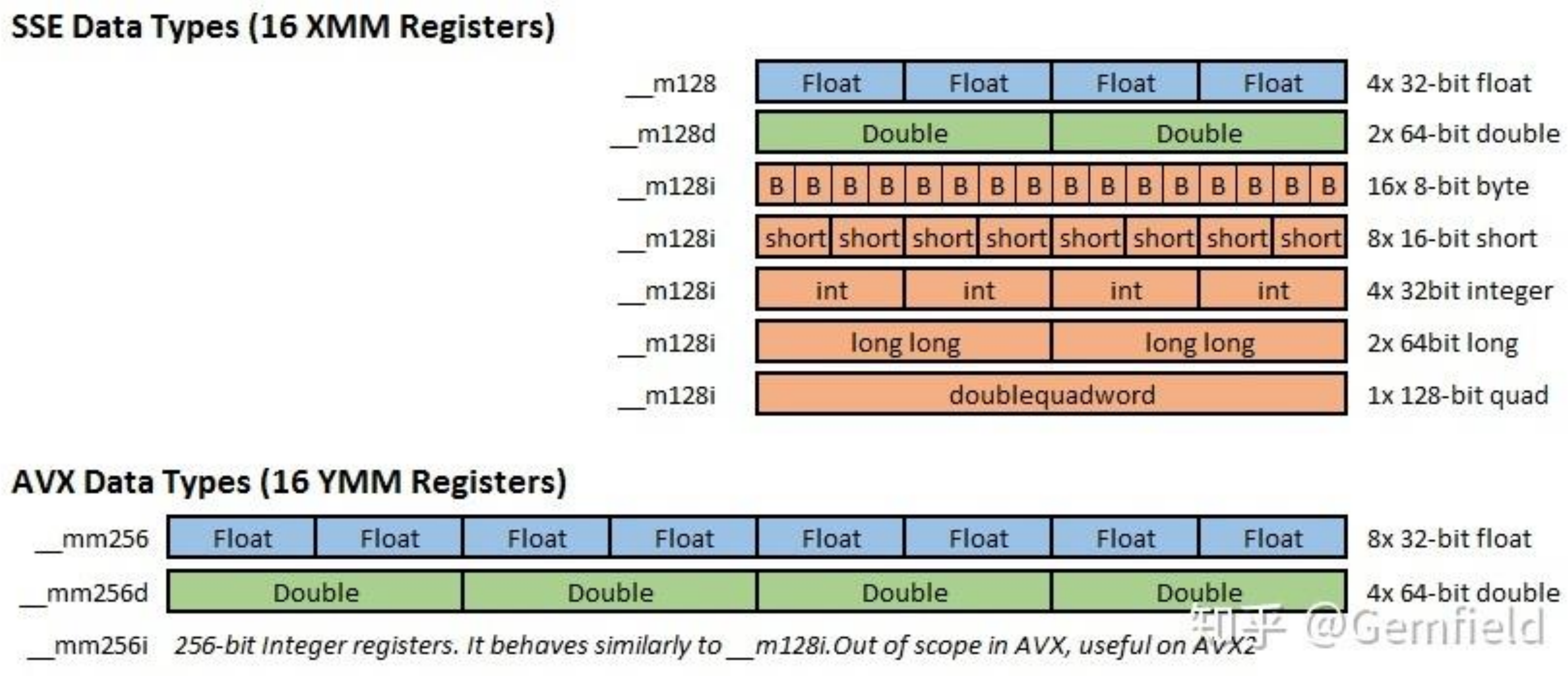
SOLE 系统优化实验室
华东师范大学

** Recompiled from MIT's OCW*

Vector Hardware

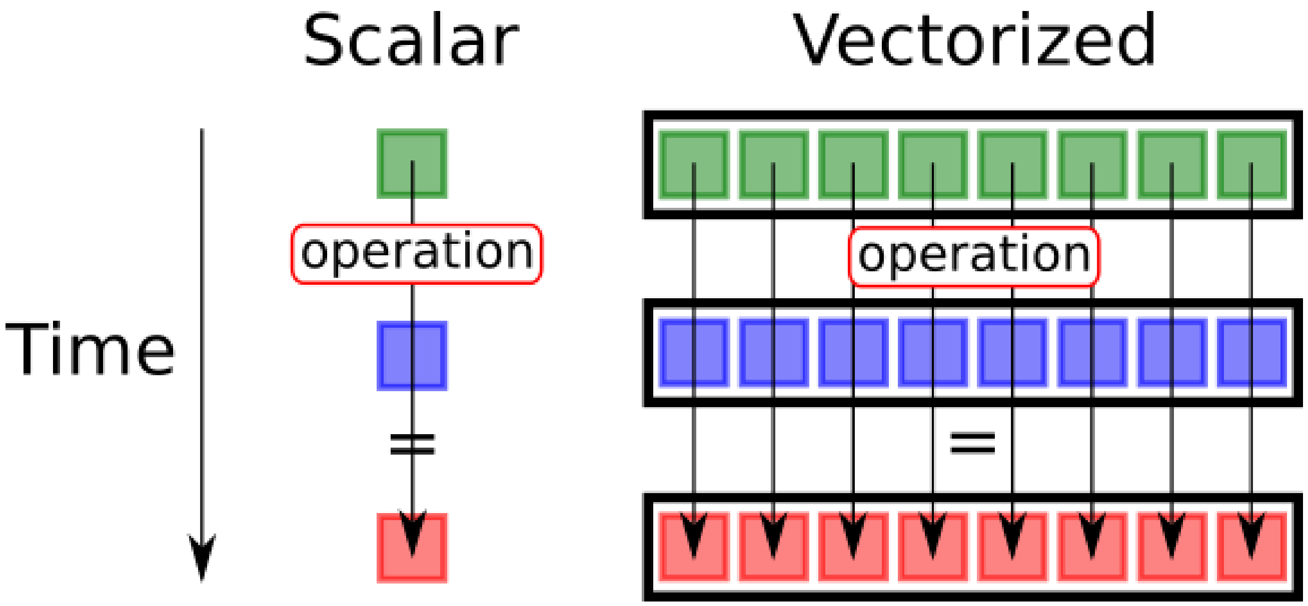
- Modern microprocessors often incorporate vector hardware to process data in a single-instruction stream, multiple-data stream (SIMD) fashion

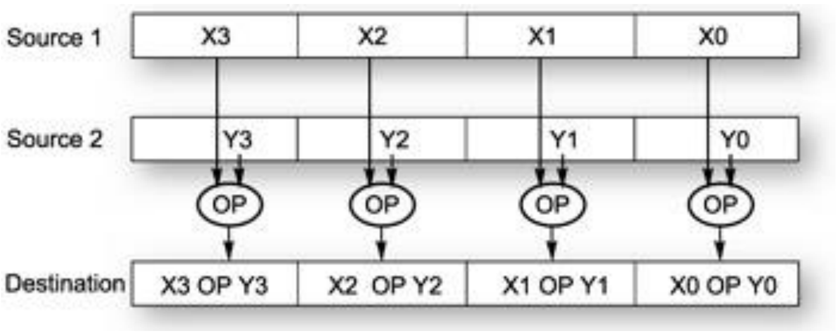


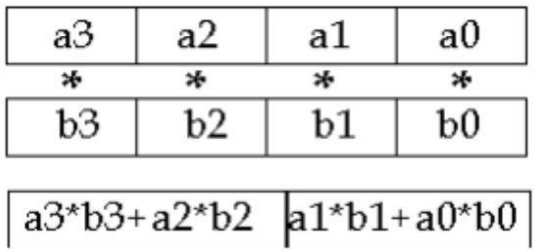


https://pic3.zhimg.com/v2-94f7b921e07e7240fdf19601a9cda45a_r.jpg

Vector Operations







PMADDWD: 16b x 16b -> 32b Multiply Add

Vectorization

QUESTION: Does the following loop vectorize?

C code

```
void daxpy(double *y, double a,
           double *x, int64_t n) {
  for (int64_t i = 0; i < n; ++i)
    y[i] += a * x[i];
}
```

What does the report say?

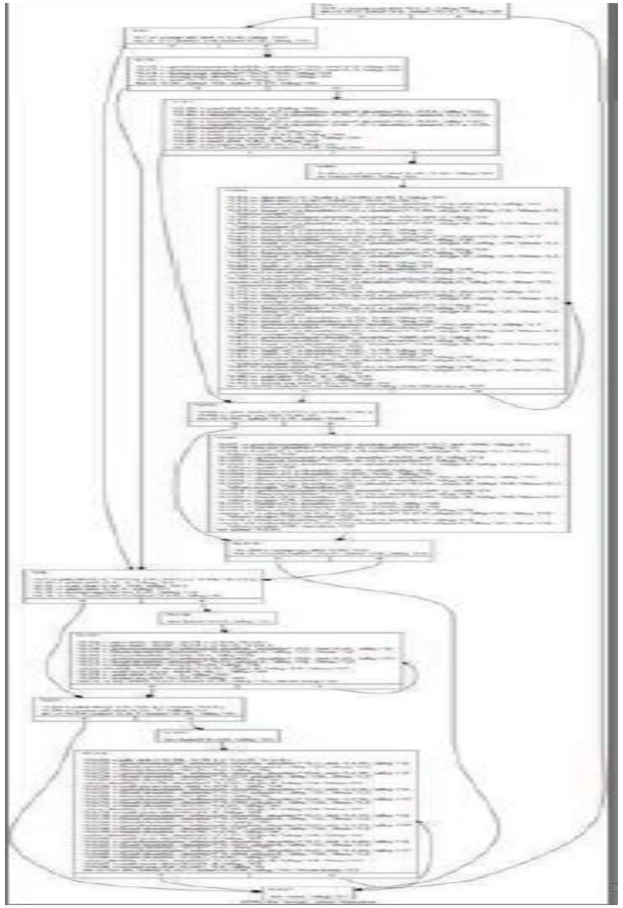
```
$ clang -O3 -c daxpy.c –Rpass=vector -Rpass-analysis=vector
daxpy.c:6:3: remark: vectorized loop (vectorization width: 2, interleaved count: 2) [-Rpass=loop-vectorize]
 for (int64_t i = 0; i < n; ++i)
 ^
```

Actual Compiled Code

- The code generated by –O2 optimization is complicated.

C code

```
void daxpy(double *y, double a,
           double *x, int64_t n) {
  for (int64_t i = 0; i < n; ++i)
    y[i] += a * x[i];
}
```



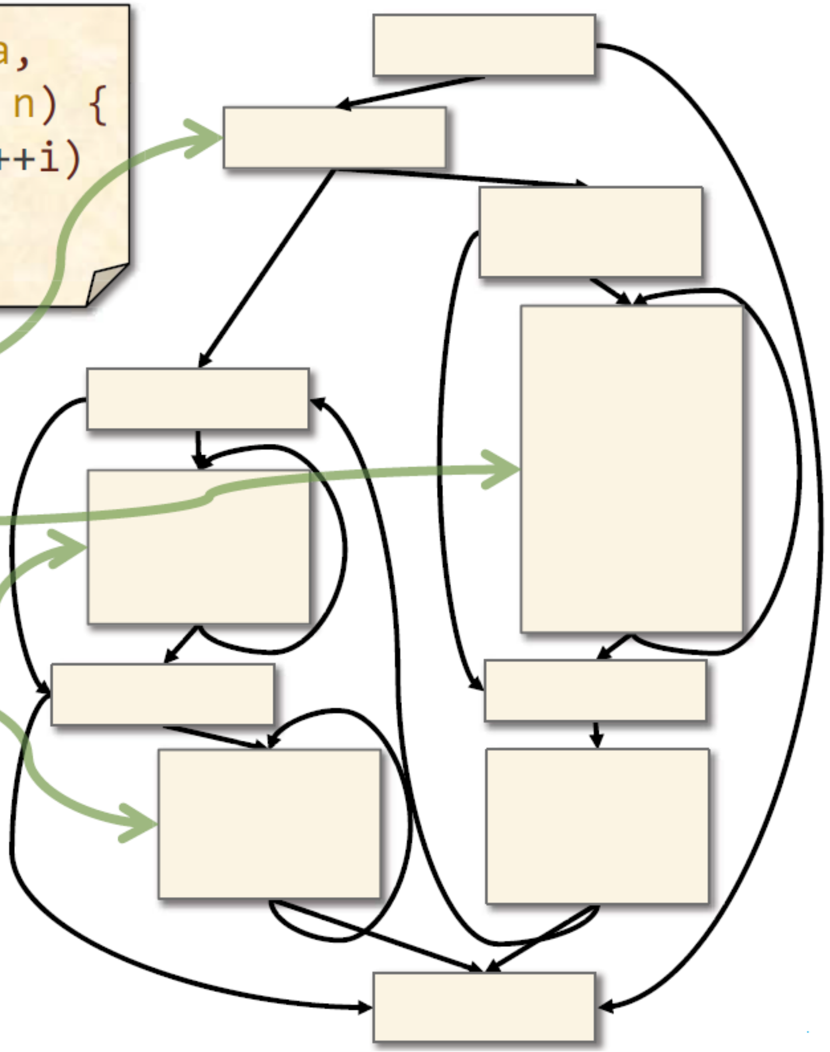
Multiple Loops

```c
void daxpy(double *y, double a,
           double *x, int64_t n) {
  for (int64_t i = 0; i < n; ++i)
    y[i] += a * x[i];
}
```

Conditional branch

Vectorized loop

Not vectorized loops

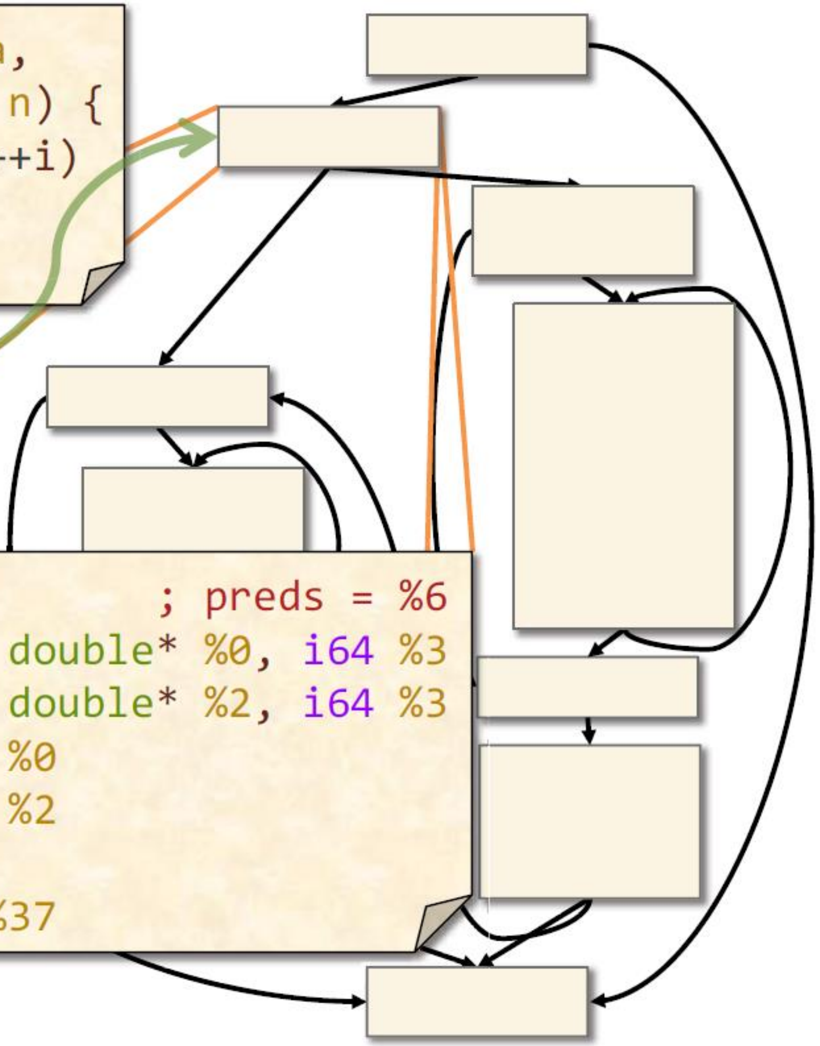


Simplified control-flow graph structure

Choosing Between Loops

```c
void daxpy(double *y, double a,
           double *x, int64_t n) {
  for (int64_t i = 0; i < n; ++i)
    y[i] += a * x[i];
}
```

Conditional branch



```llvm
; <label>:31:                                    ; preds = %6
  %32 = getelementptr double, double* %0, i64 %3
  %33 = getelementptr double, double* %2, i64 %3
  %34 = icmp ugt double* %33, %0
  %35 = icmp ugt double* %32, %2
  %36 = and i1 %34, %35
  br i1 %36, label %8, label %37
```

What's The Condition?

```
void daxpy(double *y, double a,
           double *x, int64_t n) {
  for (int64_t i = 0; i < n; ++i)
    y[i] += a * x[i];
}
```

Computes `y + n`

Register `%0` stores `y`, register `%2` stores `x`, and register `%3` stores `n`.

```
; <label>:31:                              ; preds = %6
  %32 = getelementptr double, double* %0, i64 %3
  %33 = getelementptr double, double* %2, i64 %3
  %34 = icmp ugt double* %33, %0
  %35 = icmp ugt double* %32, %2
  %36 = and i1 %34, %35
  br i1 %36, label %8, label %37
```

What's The Condition?

```c
void daxpy(double *y, double a,
           double *x, int64_t n) {
  for (int64_t i = 0; i < n; ++i)
    y[i] += a * x[i];
}
```

Computes `x + n`

Register `%0` stores `y`, register `%2` stores `x`, and register `%3` stores `n`.

```llvm
; <label>:31:                                  ; preds = %6
  %32 = getelementptr double, double* %0, i64 %3
  %33 = getelementptr double, double* %2, i64 %3
  %34 = icmp ugt double* %33, %0
  %35 = icmp ugt double* %32, %2
  %36 = and i1 %34, %35
  br i1 %36, label %8, label %37
```

What's The Condition?

```
void daxpy(double *y, double a,
           double *x, int64_t n) {
  for (int64_t i = 0; i < n; ++i)
    y[i] += a * x[i];
}
```

Compares `x + n > y`

Register `%0` stores `y`, register `%2` stores `x`, and register `%3` stores `n`.

```
; <label>:31:                           ; preds = %6
  %32 = getelementptr double, double* %0, i64 %3
  %33 = getelementptr double, double* %2, i64 %3
  %34 = icmp ugt double* %33, %0
  %35 = icmp ugt double* %32, %2
  %36 = and i1 %34, %35
  br i1 %36, label %8, label %37
```

What's The Condition?

```
void daxpy(double *y, double a,
           double *x, int64_t n) {
  for (int64_t i = 0; i < n; ++i)
    y[i] += a * x[i];
}
```

Compares y + n > x

Register %0 stores y, register %2 stores x, and register %3 stores n.

```
; <label>:31:                               ; preds = %6
  %32 = getelementptr double, double* %0, i64 %3
  %33 = getelementptr double, double* %2, i64 %3
  %34 = icmp ugt double* %33, %0
  %35 = icmp ugt double* %32, %2
  %36 = and i1 %34, %35
  br i1 %36, label %8, label %37
```

What’s The Condition?

```
void daxpy(double *y, double a,
           double *x, int64_t n) {
  for (int64_t i = 0; i < n; +
    y[i] += a * x[i];
}
```

Combines the comparisons to compute
$(y + n > x)$ & $(x + n > y)$

Register %0 stores y, register %2 stores x, and register %3 stores n.

```
; <label>:31:                                ; preds = %6
  %32 = getelementptr double, double* %0, i64 %3
  %33 = getelementptr double, double* %2, i64 %3
  %34 = icmp ugt double* %33, %0
  %35 = icmp ugt double* %32, %2
  %36 = and i1 %34, %35
  br i1 %36, label %8, label %37
```

QUESTION: What does the result of this condition mean?

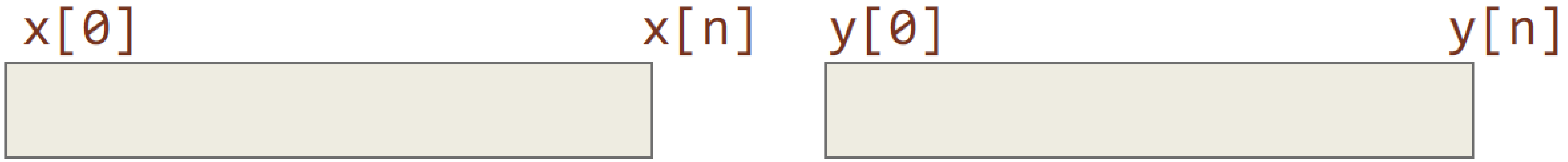
What’s The Condition?

```c
void daxpy(double *y, double a,
           double *x, int64_t n) {
  for (int64_t i = 0; i < n; ++i)
    y[i] += a * x[i];
}
```

Condition:

`(y + n > x) & (x + n > y)`

Arrays x and y in memory:



The condition is false if x appears before y in memory or vice versa.

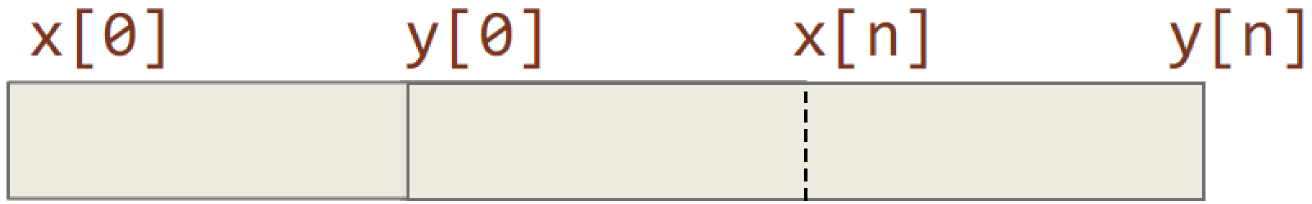
What’s The Condition?

```
void daxpy(double *y, double a,
           double *x, int64_t n) {
  for (int64_t i = 0; i < n; ++i)
    y[i] += a * x[i];
}
```

Condition:

(y + n > x) & (x + n > y)

Arrays x and y in memory:



The condition is true if arrays x and y *alias*, meaning that they overlap in memory.

Condition In Context

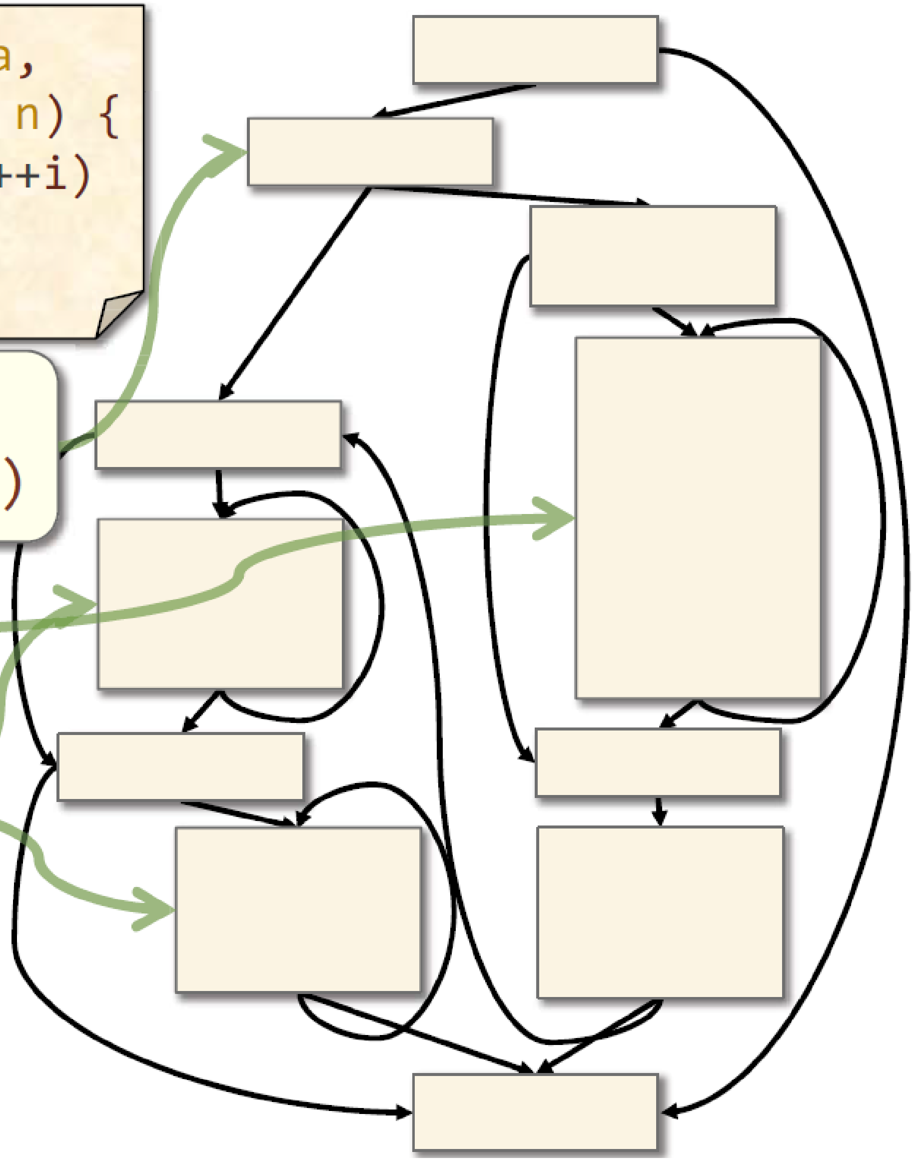
```
void daxpy(double *y, double a,
           double *x, int64_t n) {
  for (int64_t i = 0; i < n; ++i)
    y[i] += a * x[i];
}
```

Branch based on
(y + n > x) & (x + n > y)

Vectorized loop

Not vectorized loops

Simplified control-flow graph structure

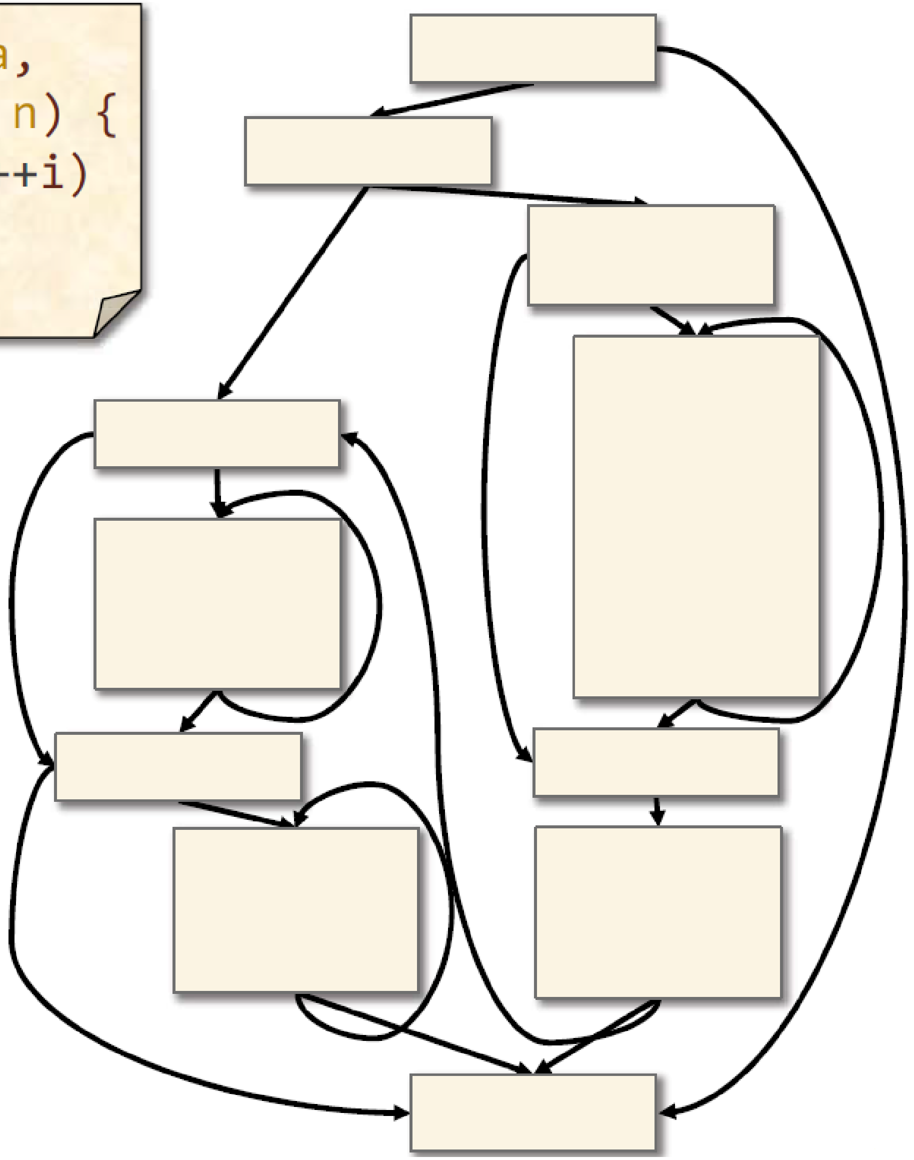


Multiple Versions

```
void daxpy(double *y, double a,
           double *x, int64_t n) {
  for (int64_t i = 0; i < n; ++i)
    y[i] += a * x[i];
}
```

Question: Does the following loop vectorize?

Answer: Yes and no. The compiler generated multiple versions of the loop, due to uncertainty about memory aliasing.



Memory Aliasing

Many compiler optimizations act conservatively when memory aliasing is possible.

```
void mm_base(double *C, int n_C, double *A, int n_A,
             double *B, int n_B, int n) {
  for (int i = 0; i < n; ++i)
    for (int k = 0; k < n; ++k)
      for (int j = 0; j < n; ++j)
        C[i*n_C+j] += A[i*n_A+k] * B[k*n_B+j];
}
```

```
$ clang -O3 -c mm_base.c -Rpass-missed=.*
mm_base.c:20:23: remark: failed to move load with loop-invariant
address because the loop may invalidate its value [-Rpass-
missed=licm]
        C[i*n_C+j] += A[i*n_A+k] * B[k*n_B+j];
                      ^
```

Dealing with Memory Aliasing

Compilers perform alias analysis to determine which addresses computed off of different pointers might refer to the same location.

- In general, alias analysis is undecidable [HU79, R94].
- Compilers use a variety of tricks to get useful alias-analysis results in practice.
- EXAMPLE: Clang uses metadata to track alias information derived from various sources, such as type information in the source code.

```
%34 = load double, double* %33, align 8, !tbaa !3,
!alias.scope !12, !noalias !9
```

What You Can Do About Aliasing

SOLUTION: Annotate your pointers.

- The `restrict` keyword allows the compiler to assume that address calculations based on a pointer will not alias those based on other pointers.
- The `const` keyword indicates that addresses based on a particular pointer will only be read.

```
void daxpy(double *restrict y, double a,
           const double *restrict x, int64_t n) {
  for (int64_t i = 0; i < n; ++i)
    y[i] += a * x[i];
}
```

Example Code: Normalize

Sometimes it’s not enough to just annotate pointers.

EXAMPLE: Normalizing a vector

```
double norm(const double *X, int n);

void normalize(double *restrict Y,
               const double *restrict X,
               int n) {
  for (int i = 0; i < n; ++i)
    Y[i] = X[i] / norm(X, n);
}
```

Compute the norm of the input vector.

Divide each input element by the norm.

IDEA: The `norm` call always returns the same value, so the compiler can move it out of the loop.

Normalize in LLVM IR

C code

```
for (int i = 0; i < n; ++i)
  Y[i] = X[i] / norm(X, n);
```

LLVM IR of loop body
with -O2 optimization

```
; <label>:8:                                          %5
  %9 = phi i64 [ 0, %5 ], [ %15, %8 ]
  %10 = getelementptr inbounds double, double* %1, i64 %9
  %11 = load double, double* %10, align 8, !dbg !12
  %12 = tail call double @norm(double* %1, i32 %2) #2
  %13 = fdiv double %11, %12
  %14 = getelementptr inbounds doubl                  i64 %9
  store double %13, double* %14, ali
  %15 = add nuw nsw i64 %9, 1
  %16 = icmp eq i64 %15, %6
  br i1 %16, label %7, label %8
```

LLVM didn't move the call to norm.

What went wrong?

Fixing Normalize

PROBLEM: The compiler does not know that the norm function does not modify memory.

EXAMPLE: Normalizing a vector

```
__attribute__((const))
double norm(const double *X, int n);

void normalize(double *restrict Y,
               const double *restrict X,
               int n) {
  for                                )
    Y[                               ;
}
```

For instance, norm might modify a global variable.

Allows LLVM to assume that norm does not modify any memory.

SOLUTION: Annotate the norm function.

QUESTION: Does the following loop vectorize?

```
void daxpy4(double *restrict z, double a,
            const double *restrict x,
            const double *restrict y,
            size_t n) {
  for (size_t i = 0; i < n; i+=4) {
    z[i] = a * x[i] + y[i];
    z[i+1] = a * x[i+1] + y[i+1];
    z[i+2] = a * x[i+2] + y[i+2];
    z[i+3] = a * x[i+3] + y[i+3];
  }
}
```

What went wrong?

```
$ clang -O3 -c daxpy.c –Rpass=vector -Rpass-analysis=vector
daxpy.c:21:3: remark: loop not vectorized: could not determine number of loop
iterations [-Rpass-analysis=loop-vectorize]
 for (size_t i = 0; i < n; i+=4) {
 ^
```

PROBLEM: In C, the behavior of unsigned-integer overflow is to wrap to 0.

```
void daxpy4(double *restrict z, double a,
            const double *restrict x,
            const double *restrict
            size_t n) {
  for (size_t i = 0; i < n; i+=4) {
    z[i] = a * x[i] + y[i];
    z[i+1] = a * x[i+1] + y[i+1];
    z[i+2] = a * x[i+2] + y[i+2];
    z[i+3] = a *
  }
}
```

Implemented as an unsigned 64-bit integer.

QUESTION: How many iterations are in this loop?

ANSWER: Either $\lfloor n/4 \rfloor$ or infinity.

SOLUTION: Use signed integer types, unless you absolutely need an unsigned type, e.g., for bit-hacking.

```c
void daxpy4(double *restrict z, double a,
            const double *restrict x,
            const double *restrict y,
            int64_t n) {
  for (int64_t i = 0; i < n; i+=4) {
    z[i] = a * x[i] + y[i];
    z[i+1] = a * x[i+1] + y[i+1];
    z[i+2] = a * x[i+2] + y[i+2];
    z[i+3] = a * x[i+3] + y[i+3];
```

```
$ clang -O3 -c daxpy.c –Rpass=vector
daxpy.c:21:3: remark: vectorized loop (vectorization width: 2, interleaved count: 1) [-Rpass=loop-vectorize]
 for (int64_t i = 0; i < n; i+=4) {
 ^
```

WHY IT WORKS: In C, signed-integer overflow has **undefined behavior**.

- As a result, when analyzing code, the compiler is allowed to assume that signed-integer arithmetic never overflows.

Why is signed-integer overflow undefined behavior?

- Not all architectures implement signed overflow the same way.
- Programmers generally don't write code that explicitly accommodates signed overflow.
- So the compiler and language compromise.

上机作业 A4：循环向量化

布置周：第 11 周 11 月 16 日
提交周：第 13 周 11 月 30 日

目标：通过本次作业，希望同学们进一步掌握 Intel 的向量化指令以及编译器是如何对循环进行向量化优化的，并通过理解和分析编译器生成的汇编代码，确认编译器进行向量化优化的效果以及如何对源程序或者编译选项进行一些小修改从而让编译器能够生成更加高质量的向量化代码。

作业要求：用 clang 开源编译器对 https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2018/assignments/MIT6_172F18_hw3.zip 中的多个程序示例进行循环向量化相关的实验，理解影响循环向量化的一些关键因素，掌握相关的循环向量化的优化实践，并且通过对向量化后程序的运行性能测量与分析，从而对循环向量化带来的具体性能提升获得一个感性的认识。

***备注**：此作业从MIT 6.172 Performance Engineering of Software Systems "Homework 3: Vectorization" 改编而来（源文档地址：https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2018/assignments/MIT6_172F18hw3.pdf），MIT6_172F18hw3.pdf可以作为参考资料，但**请同学们按照本改编后的作业要求完成作业**。*

对于使用Mac（M1/M2）机器的同学，Writeup 1 – Writeup 7可以在Mac（M1/M2）上通过交叉编译（安装libc6-dev-amd64-cross并修改Makefile）生成相应的汇编文件后做分析，Writeup 8可以有三个选择

1. *在Mac（M1/M2）上用Qemu仿真器运行X86-64的执行文件并记录性能*
2. *在Mac（M1/M2）上借助于Rosetta 2来运行生成的X86-64可执行文件并记录性能*
3. *把编译出来的binary拿到同学的X86-64机器或者云实例上去测性能数据*


Q
&
A