Assembly Language and Microcomputer Interface

Introduction

Ming Cai

cm@zju.edu.cn

College of Computer Science and Technology
Zhejiang University

Course Information

- **Text book**
  - **Barry B. Brey 《INTEL Microprocessors》 8th Edition**
- **QQ group: 891509207 (rename to “ID + name”)**
  password: assembly
- **TA1: Qinran Wu wuqinran@zju.edu.cn**
- **TA2: Jingwen Pu 3180105872@zju.edu.cn**

Course Resources

References

- Intel® 64 and IA-32 Architectures Software Developer Manuals: https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html
- AMD64 Architecture Programmer's Manual Volumes 1-5: https://www.amd.com/system/files/TechDocs/40332.pdf
- Irvine K R. Assembly language for X86 processors[M]. 2015.
- Kusswurm D. Modern X86 Assembly Language Programming[M]. Apress, 2018.
- Jo Van Hoey. Beginning X64 Assembly Programming[M]. Apress, 2019.
- Intel Intrinsics Guide: https://software.intel.com/sites/landingpage/IntrinsicsGuide/

Recommended videos

- 汇编语言程序设计(华中)：www.bilibili.com/video/BV1Nt411V7fa?p=27&t=138
- 微机原理与接口技术 (西交大)：www.bilibili.com/video/BV1DA411t7NN?p=1
- 汇编语言程序设计(清华)：www.bilibili.com/video/BV1G7411Z7VP?p=1

Course Resources

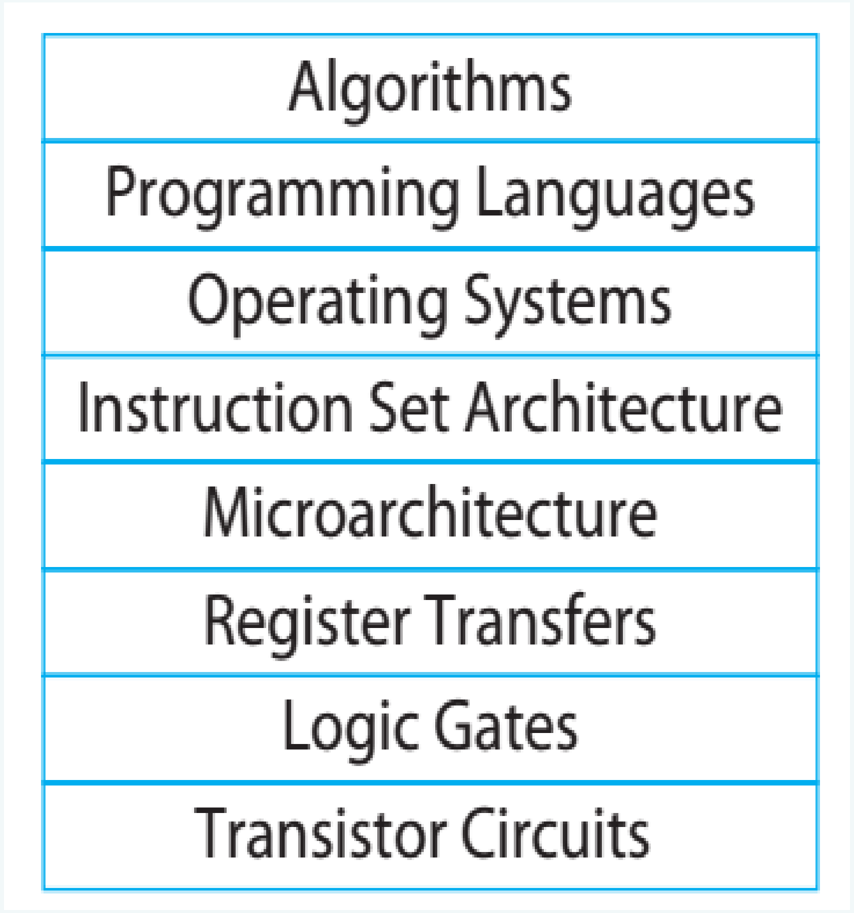
- **Microsoft Macro Assembler (MASM) references**
  - Directives Reference: https://docs.microsoft.com/en-us/cpp/assembler/masm/directives-reference?view=msvc-160
  - Symbols reference:

    https://www.amd.com/system/files/TechDocs/40332.pdf
  - Operators reference: https://docs.microsoft.com/en-us/cpp/assembler/masm/operators-reference?view=msvc-160
- **Useful software links**
  - EMU8086 - Microprocessor Emulator:

    https://emu8086-microprocessor-emulator.en.softonic.com
  - Godbolt compiler explorer: https://www.godbolt.org

    https://www.bilibili.com/video/BV1Uv4y1Z7pe?from=search&seid=8220486653503703739
  - Online x86 / x64 Assembler and Disassembler: https://defuse.ca/online-x86-assembler.htm#disassembly

Abstraction Layers in Computer Systems



Algorithms
Programming Languages
Operating Systems
Instruction Set Architecture
Microarchitecture
Register Transfers
Logic Gates
Transistor Circuits

- **greedy, heuristic, LP, DP**
- **C/C++, Java, Python**
- **Linux, Windows, Android**
- **X86, ARM, RISC-V**
- **Pipeline, OOE, Multiprocessing**
- **Register, Datapath, Control Unit**
- **AND, OR, NOT, NAND, NOR**
- **BJT, JFET, IGFET**

Reasons to learn assembly language (1/6)

- Whether you should learn assembly language depends on what your goals are. For most developers, the answer is "no".
- Only a relative handful of the world's engineers and computer scientists actually use assembly language. Some specific areas where assembly language gets used are:
  - Operating systems
  - Firmware
  - Device drivers
  - Compiler design
  - Embedded systems
  - Hardware design
  - Advanced cryptography
  - Theoretical computer science

Reasons to learn assembly language (2/6)

- Lacking knowledge of assembly prevents us from understanding valuable information on how a program runs, and limits understanding of what the code is actually doing.
- Consider the following example from "Dive Into Systems" (Chap 12 Code Optimization) with a = INT_MAX:

```
int  silly (int a)  {
     return (a + 1) > a;
}
```

inconsistent output

- using x86-64 clang with optimization (-O0)

return 0

```
......
mov   eax,  DWORD PTR [rbp-0x4]
add   eax,  0x1
cmp   eax,  DWORD PTR [rbp-0x4]
......
```

- using x86-64 clang with optimization (-O1)

return 1

```
mov   eax,  0x1

ret
```

Reasons to learn assembly language (3/6)

- To gain a deeper insight of how optimization works.

```
int Sum1ToN(int n) {
        int sum = 0;
        for (int i = 0; i < n; i++) {
                sum += i;
        }
        return sum;
}
```

Adding the numbers from 1 to n-1

$$sum = \frac{(n-1)\times(n-2)}{2} + n - 1$$
$$= \frac{(n-1)\times(n-2+2)}{2} = \frac{(n-1)\times n}{2}$$

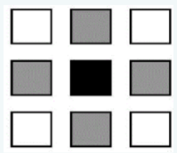
- using compiler explorer: https://www.godbolt.org/z/xTYPhr14n
- using x86-64 gcc with different optimization options (-O1, -O3)
- using x86-64 icc with optimization option (-O3)
- using x86-64 clang with optimization option (-O3)

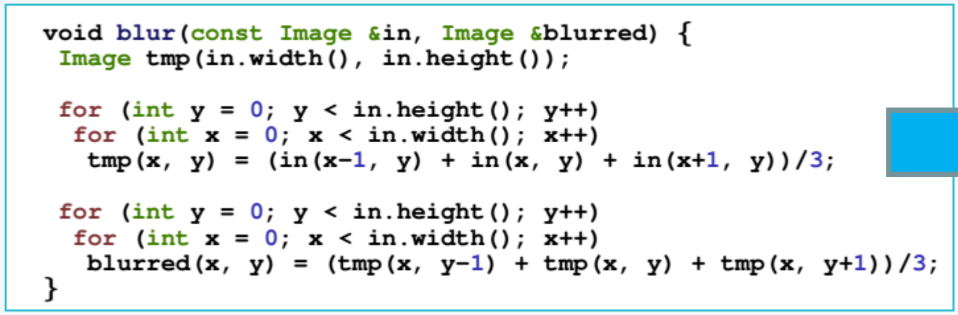
Reasons to learn assembly language (4/6)

- To develop highly efficient and lightweight applications.
  - **NCNN** (Tencent) is a high-performance neural network inference computing framework for mobile platforms featured by ARM NEON assembly level of careful optimization.
  - **MNN** (Alibaba) is a highly efficient and lightweight deep learning framework. It supports inference and training of deep learning models on-device. It implements core operations by relying on large amounts of handwritten assembly code to make full use of the ARM CPU.

Reasons to learn assembly language (5/6)

- Assembly language enables us to exploit the single-instruction multiple-data (SIMD) capabilities of a processor.





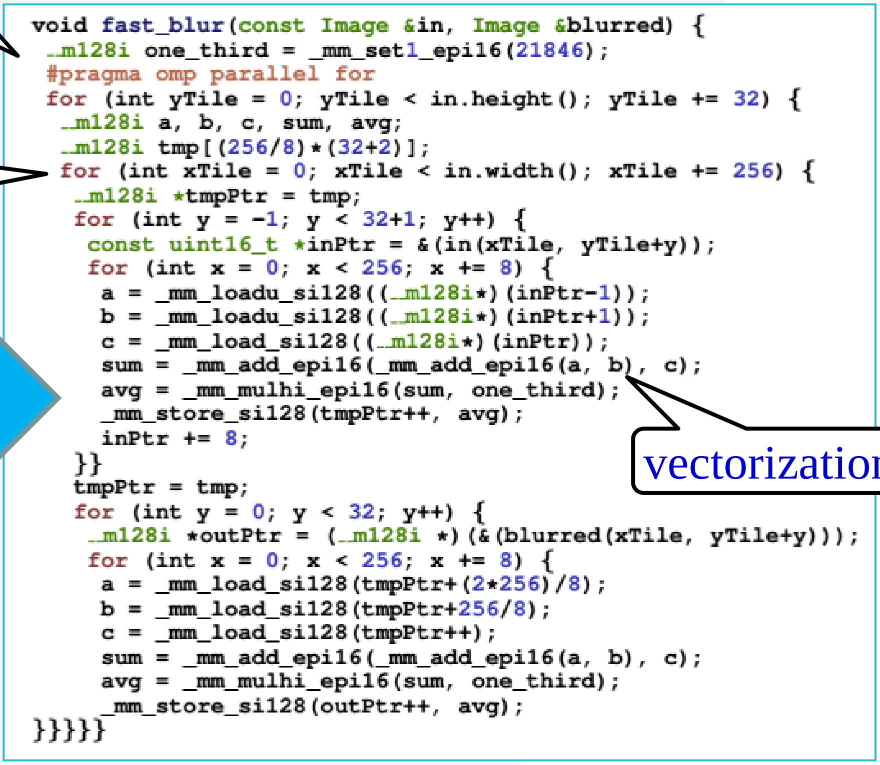
```
void blur(const Image &in, Image &blurred) {
 Image tmp(in.width(), in.height());

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;
}
```

Clean C++ : 9.94 ms per megapixel

multithreading

tiling

vectorization



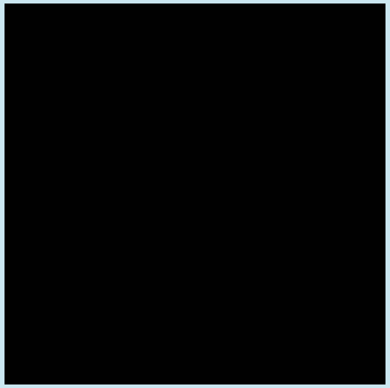
```
void fast_blur(const Image &in, Image &blurred) {
 _m128i one_third = _mm_set1_epi16(21846);
 #pragma omp parallel for
 for (int yTile = 0; yTile < in.height(); yTile += 32) {
  _m128i a, b, c, sum, avg;
  _m128i tmp[(256/8)*(32+2)];
  for (int xTile = 0; xTile < in.width(); xTile += 256) {
   _m128i *tmpPtr = tmp;
   for (int y = -1; y < 32+1; y++) {
    const uint16_t *inPtr = &(in(xTile, yTile+y));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_loadu_si128((_m128i*)(inPtr-1));
     b = _mm_loadu_si128((_m128i*)(inPtr+1));
     c = _mm_load_si128((_m128i*)(inPtr));
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(tmpPtr++, avg);
     inPtr += 8;
   }}
   tmpPtr = tmp;
   for (int y = 0; y < 32; y++) {
    _m128i *outPtr = (_m128i *)(&(blurred(xTile, yTile+y)));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_load_si128(tmpPtr+(2*256)/8);
     b = _mm_load_si128(tmpPtr+256/8);
     c = _mm_load_si128(tmpPtr++);
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(outPtr++, avg);
}}}}}
```

Fast C++ (for x86) : 0.90 ms per megapixel

Reasons to learn assembly language (6/6)

- Assembly language enables us to exploit the single-instruction multiple-data (SIMD) capabilities of a processor.

Clean C++

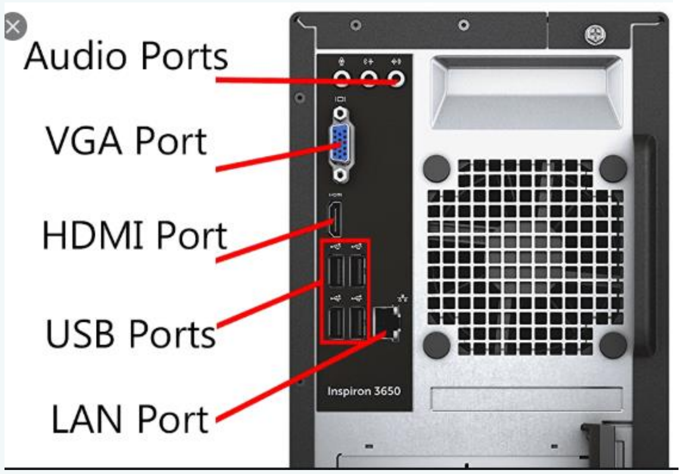


multithreading & tiling

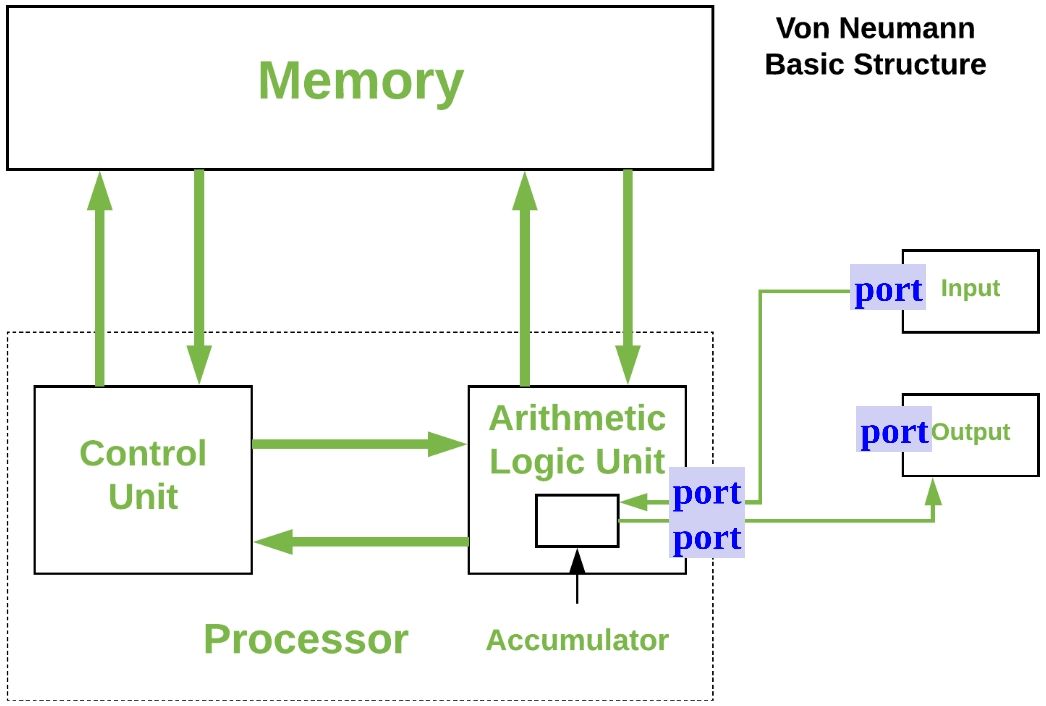
vectorization

Fast C++ (for x86)

Computer interface and their functions (1/3)

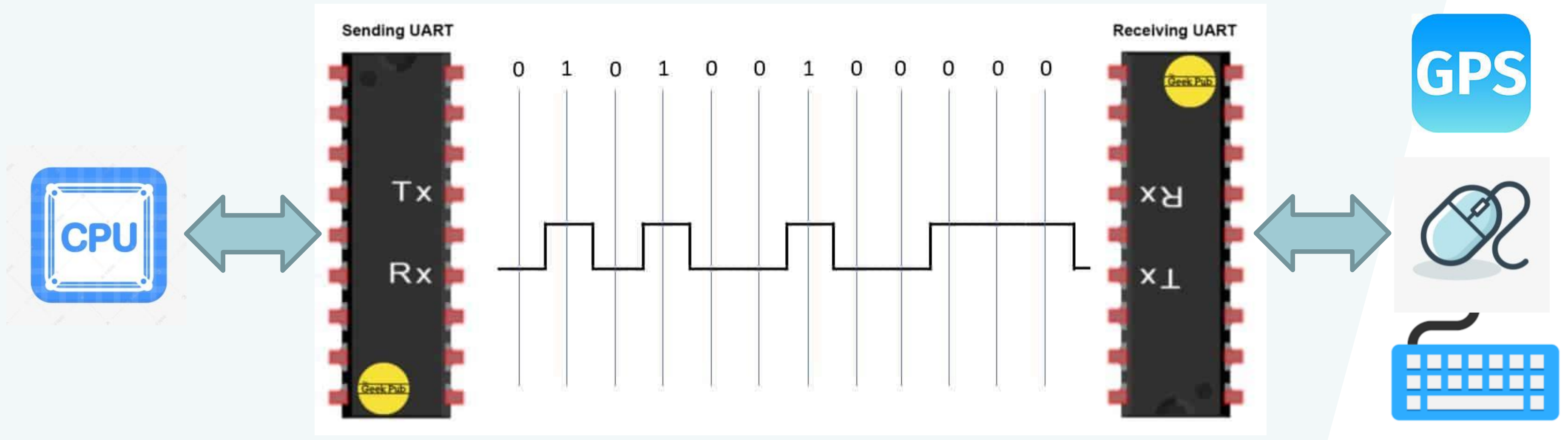


Computer ports



Computer interface and their functions (2/3)

- Interface or port is a point of connection that links together computer and peripheral devices, so that they can work together.



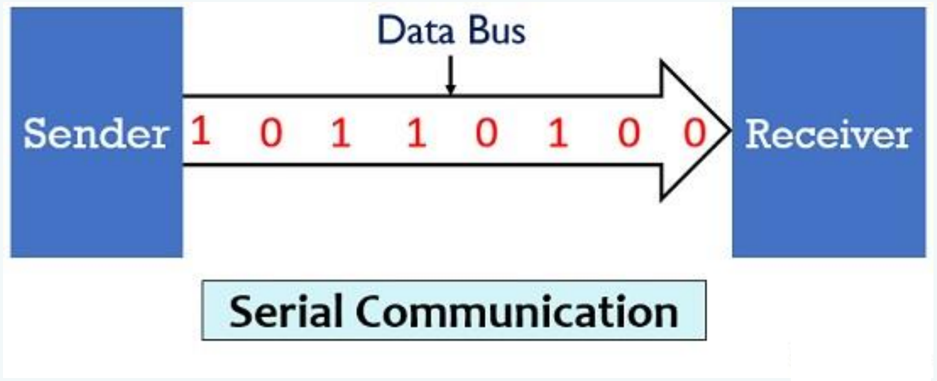
Computer

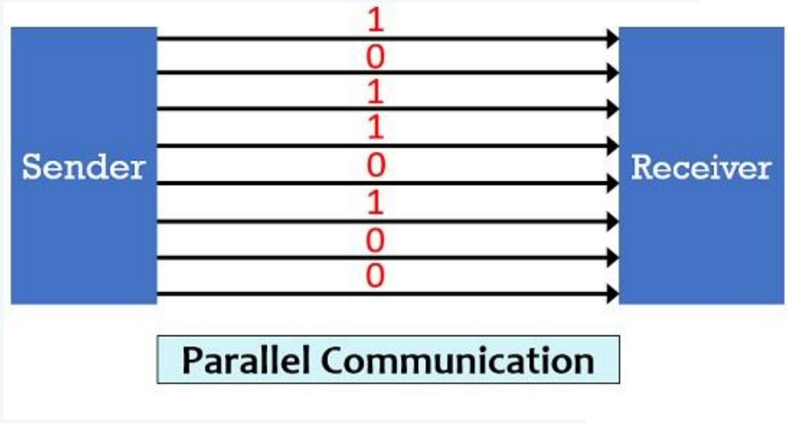
Serial interface

Peripheral

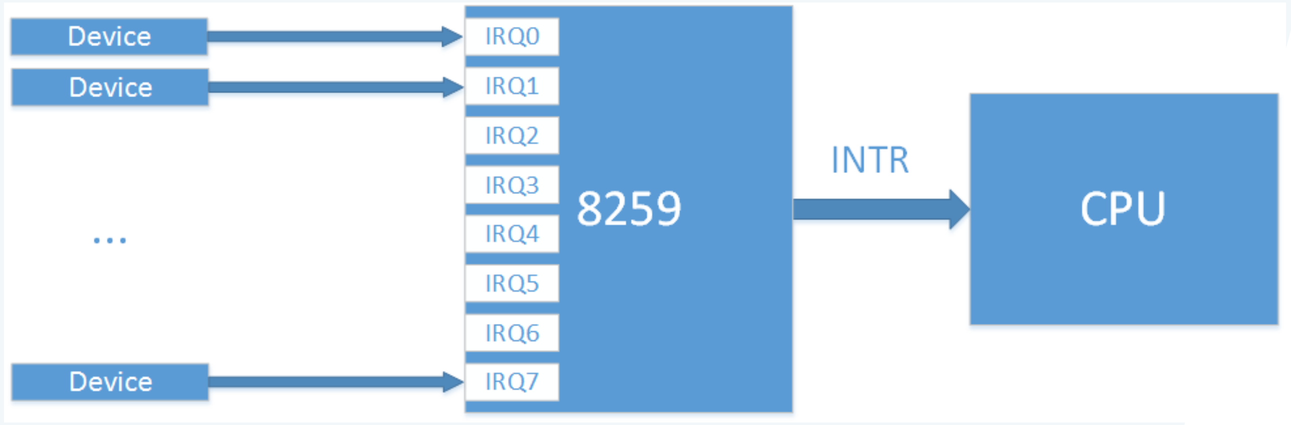
Computer interface and their functions (3/3)

- The function of interface:
  - Communication: serial/parallel communication





  - Control: interrupt controller, timer



Interrupt controller

Topics covered

- **Topics**
  - **The microprocessor and its architecture (Chap. 1-2)**
  - **Addressing modes (Chap. 3)**
  - **Data movement instructions (Chap. 4)**
  - **Arithmetic and logic instructions (Chap. 5)**
  - **Program control instructions (Chap. 6)**
  - **Using assembly language with c/c++ (Chap. 7)**
  - **Basic I/O interface (Chap. 11)**
  - **Interrupts (Chap. 12)**
  - **Arithmetic coprocessor and SIMD (chap. 14/++)**

Experiment (1/5)

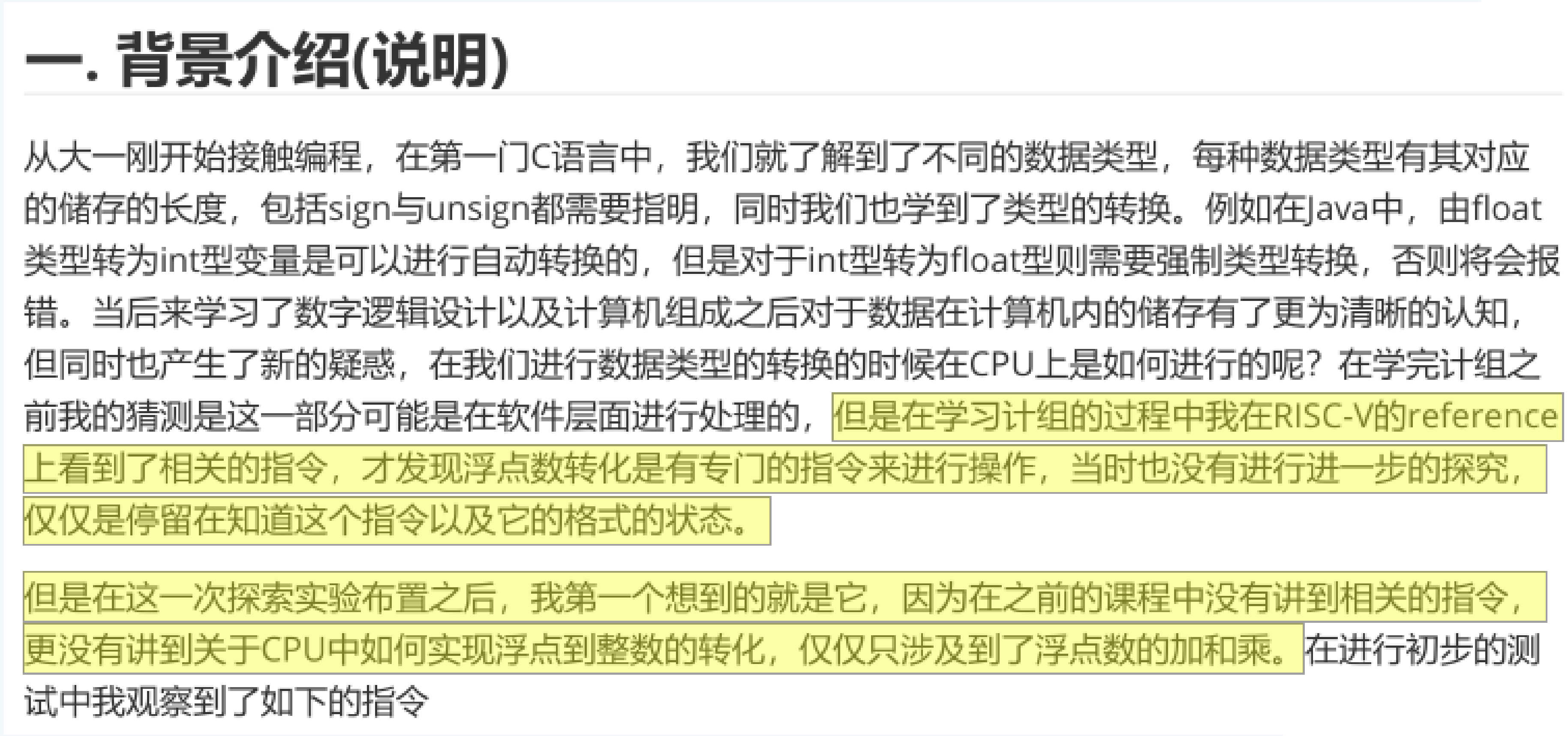
- Preliminary experiment (optionally)
  - Programming——Environment setup
  - Programming——Branching
  - Programming——Loops
  - Programming——Mixing assembly and C/C++
  - Programming——x64 assembly programming
  - Programming——Basics of SIMD programming
  - I/O interface

Experiment (2/5)

- Exploratory research (mandatory)
  - Programming language exploration
  - SIMD exploration
- Why exploration?
  - understanding low-level implementation details of programming language
  - using low-level techniques to optimize your programming

Language exploration: sample 1

- RISC-V下浮点数转换的round问题



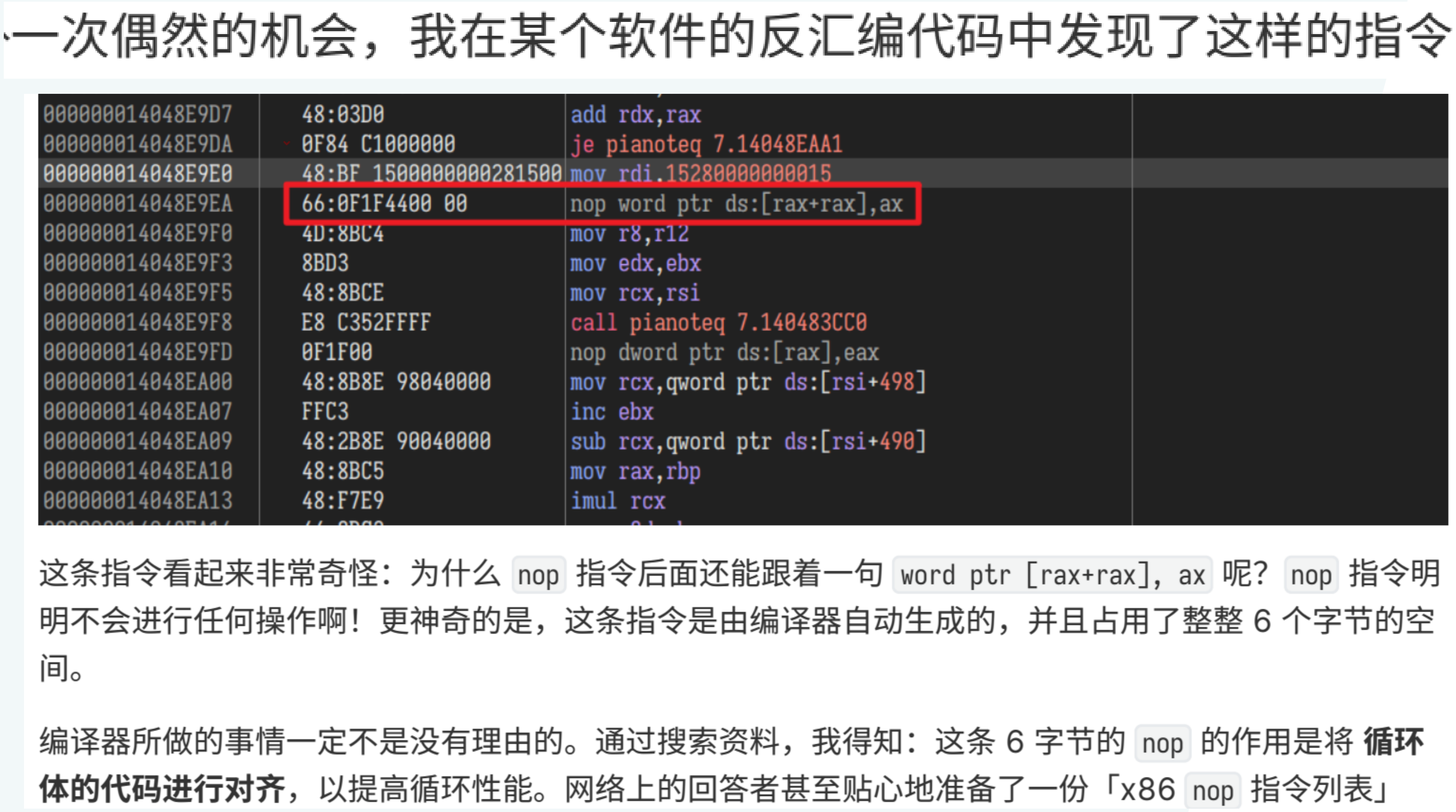
一. 背景介绍(说明)

从大一刚开始接触编程，在第一门C语言中，我们就了解到了不同的数据类型，每种数据类型有其对应的储存的长度，包括sign与unsign都需要指明，同时我们也学到了类型的转换。例如在Java中，由float类型转为int型变量是可以进行自动转换的，但是对于int型转为float型则需要强制类型转换，否则将会报错。当后来学习了数字逻辑设计以及计算机组成之后对于数据在计算机内的储存有了更为清晰的认知，但同时也产生了新的疑惑，在我们进行数据类型的转换的时候在CPU上是如何进行的呢？在学完计组之前我的猜测是这一部分可能是在软件层面进行处理的，但是在学习计组的过程中我在RISC-V的reference上看到了相关的指令，才发现浮点数转化是有专门的指令来进行操作，当时也没有进行进一步的探究，仅仅是停留在知道这个指令以及它的格式的状态。

但是在这一次探索实验布置之后，我第一个想到的就是它，因为在之前的课程中没有讲到相关的指令，更没有讲到关于CPU中如何实现浮点到整数的转化，仅仅只涉及到了浮点数的加和乘。在进行初步的测试中我观察到了如下的指令

Language exploration: sample 2

- The Art of “nop” in x86



一次偶然的机会，我在某个软件的反汇编代码中发现了这样的指令：

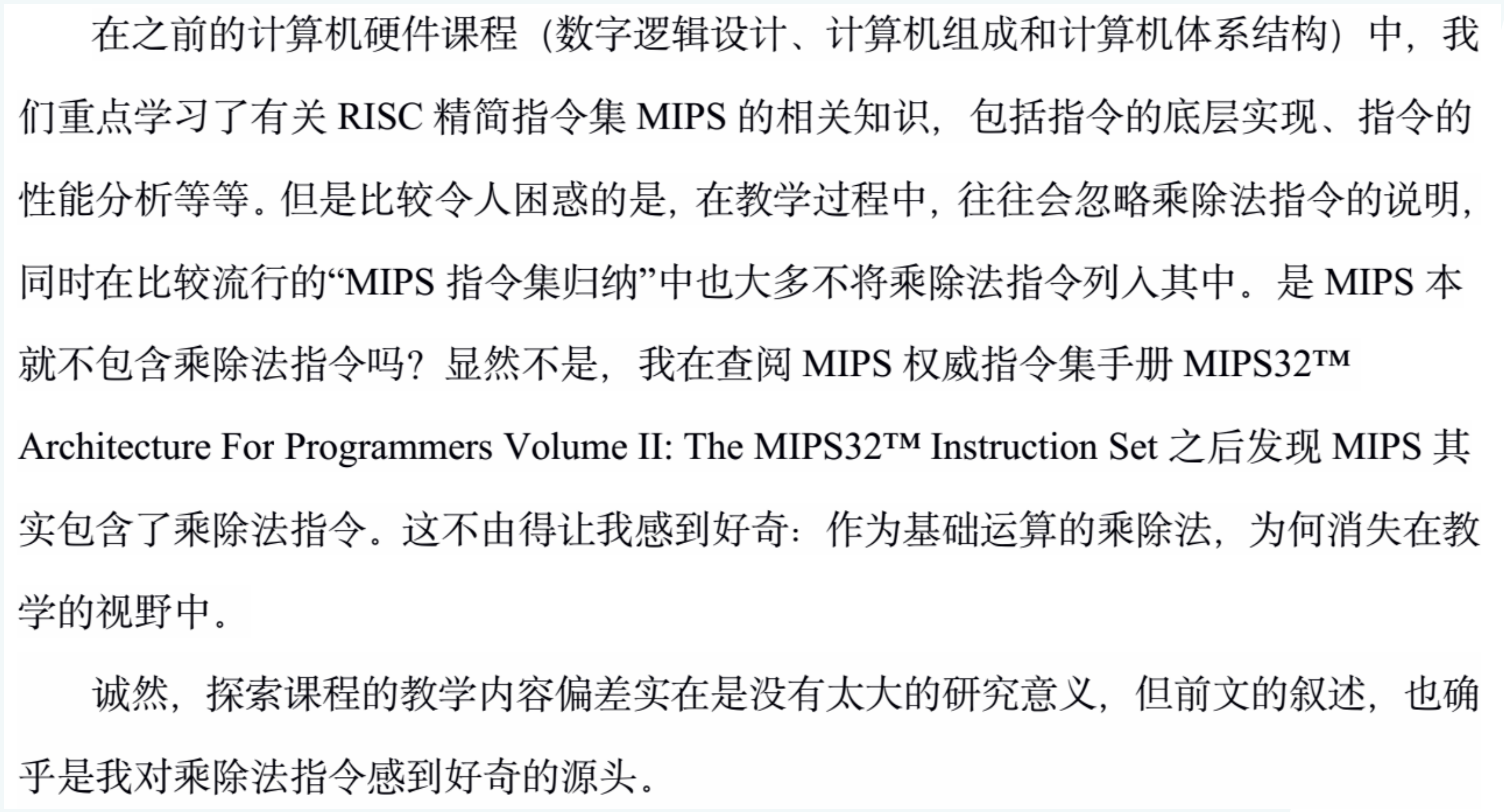
```
000000014048E9D7    48:03D0                  add rdx,rax
000000014048E9DA    0F84 C1000000            je pianoteq 7.14048EAA1
000000014048E9E0    48:BF 1500000000281500   mov rdi,15280000000015
000000014048E9EA    66:0F1F4400 00           nop word ptr ds:[rax+rax],ax
000000014048E9F0    4D:8BC4                  mov r8,r12
000000014048E9F3    8BD3                     mov edx,ebx
000000014048E9F5    48:8BCE                  mov rcx,rsi
000000014048E9F8    E8 C352FFFF              call pianoteq 7.140483CC0
000000014048E9FD    0F1F00                   nop dword ptr ds:[rax],eax
000000014048EA00    48:8B8E 98040000         mov rcx,qword ptr ds:[rsi+498]
000000014048EA07    FFC3                     inc ebx
000000014048EA09    48:2B8E 90040000         sub rcx,qword ptr ds:[rsi+490]
000000014048EA10    48:8BC5                  mov rax,rbp
000000014048EA13    48:F7E9                  imul rcx
```

这条指令看起来非常奇怪：为什么 `nop` 指令后面还能跟着一句 `word ptr [rax+rax], ax` 呢？ `nop` 指令明明不会进行任何操作啊！更神奇的是，这条指令是由编译器自动生成的，并且占用了整整 6 个字节的空间。

编译器所做的事情一定不是没有理由的。通过搜索资料，我得知：这条 6 字节的 `nop` 的作用是将 **循环体的代码进行对齐**，以提高循环性能。网络上的回答者甚至贴心地准备了一份「x86 `nop` 指令列表」

Language exploration: sample 3

- RISC-V 乘除法指令的探索和性能影响分析



在之前的计算机硬件课程（数字逻辑设计、计算机组成和计算机体系结构）中，我们重点学习了有关 RISC 精简指令集 MIPS 的相关知识，包括指令的底层实现、指令的性能分析等等。但是比较令人困惑的是，在教学过程中，往往会忽略乘除法指令的说明，同时在比较流行的“MIPS 指令集归纳”中也大多不将乘除法指令列入其中。是 MIPS 本就不包含乘除法指令吗？显然不是，我在查阅 MIPS 权威指令集手册 MIPS32™ Architecture For Programmers Volume II: The MIPS32™ Instruction Set 之后发现 MIPS 其实包含了乘除法指令。这不由得让我感到好奇：作为基础运算的乘除法，为何消失在教学的视野中。

诚然，探索课程的教学内容偏差实在是没有太大的研究意义，但前文的叙述，也确乎是我对乘除法指令感到好奇的源头。

Language exploration: more samples

- 汇编视角下的静态局部变量保持状态的秘密

  https://www.bilibili.com/video/BV1ms4y1C7xa/?spm_id_from=333.880.my_history.page.click&vd_source=dd3293dbce7e18681664249d45630c84

- 如何从汇编角度理解const char* s = “hello”

  https://www.bilibili.com/video/BV1SP411i7Cq/?spm_id_from=333.788.recommend_more_video.1&vd_source=dd3293dbce7e18681664249d45630c84

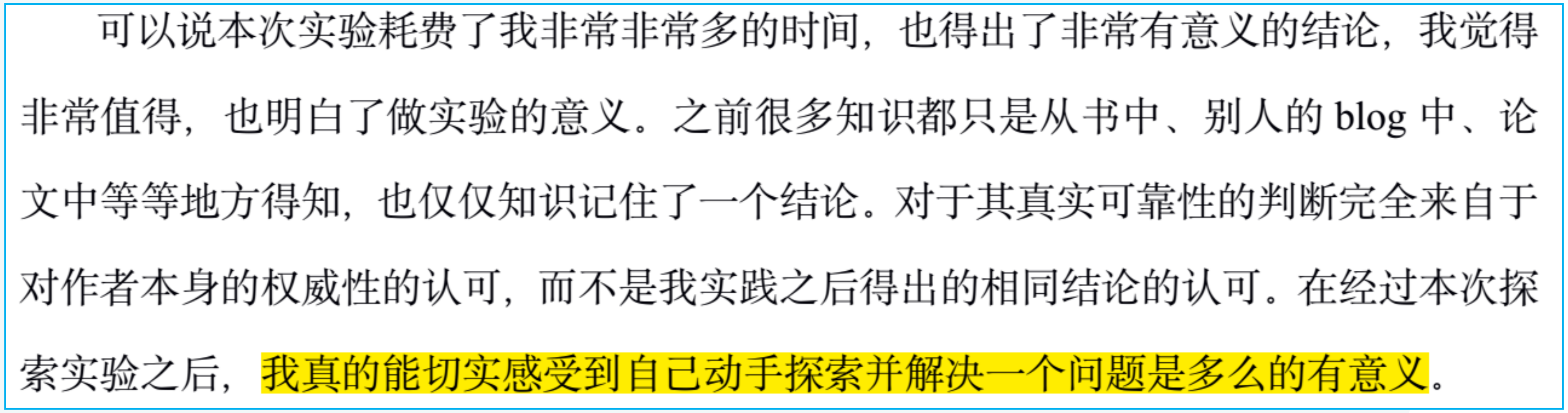
- 静态链接中符号重定位的奥妙

  https://www.bilibili.com/video/BV1az4y1x76v/?spm_id_from=333.999.0.0&vd_source=dd3293dbce7e18681664249d45630c84

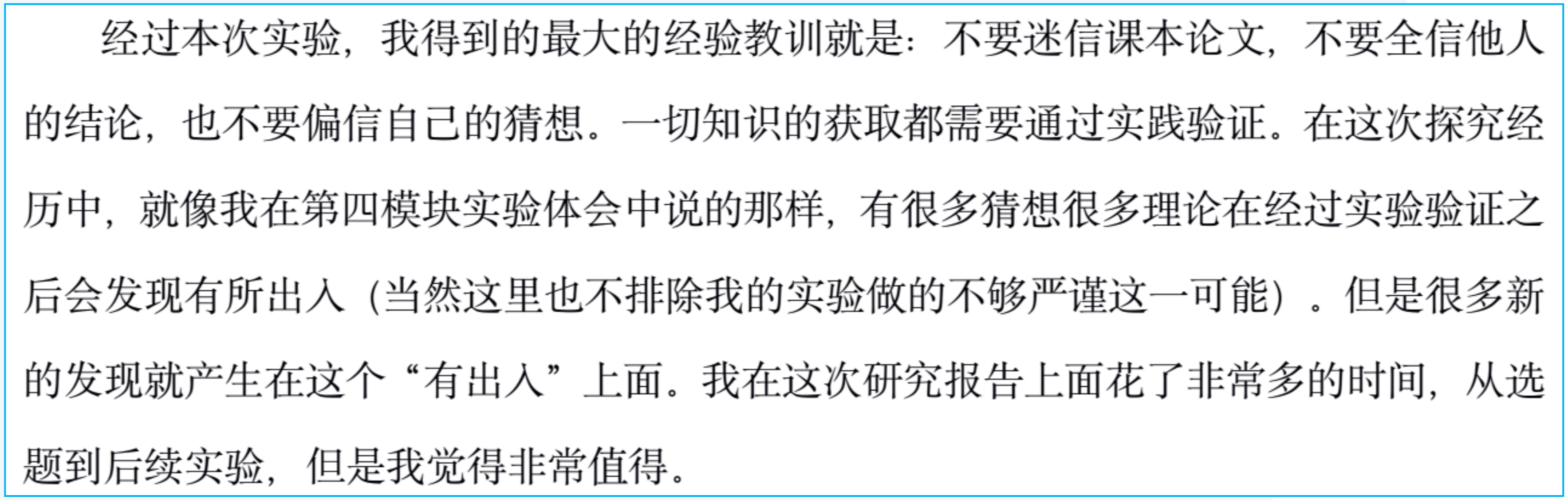
- sizeof是操作符还是函数?

  https://www.bilibili.com/video/BV1Hm4y1e7ju/?spm_id_from=333.1007.top_right_bar_window_history.content.click&vd_source=dd3293dbce7e18681664249d45630c84

Comments on language exploration (1/3)

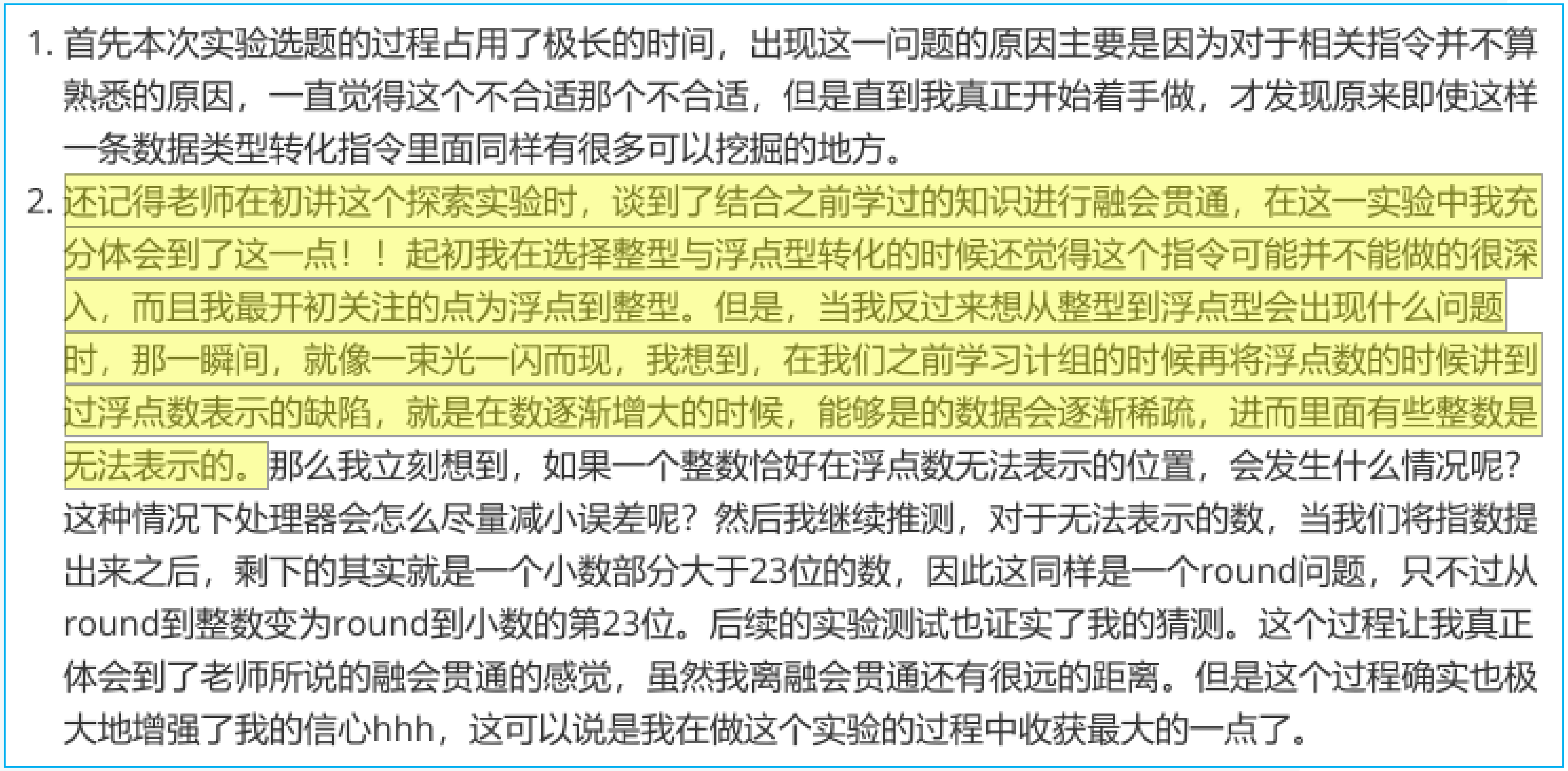


可以说本次实验耗费了我非常非常多的时间，也得出了非常有意义的结论，我觉得非常值得，也明白了做实验的意义。之前很多知识都只是从书中、别人的 blog 中、论文中等等地方得知，也仅仅知识记住了一个结论。对于其真实可靠性的判断完全来自于对作者本身的权威性的认可，而不是我实践之后得出的相同结论的认可。在经过本次探索实验之后，我真的能切实感受到自己动手探索并解决一个问题是多么的有意义。



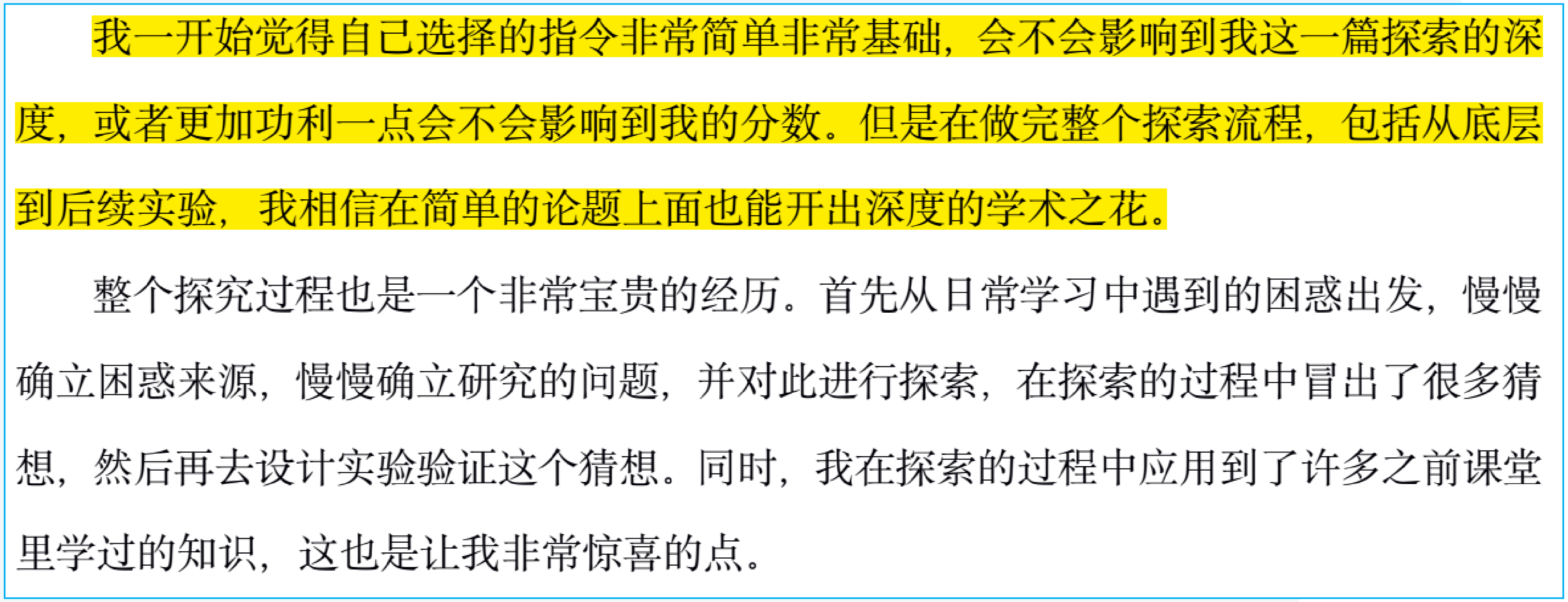
经过本次实验，我得到的最大的经验教训就是：不要迷信课本论文，不要全信他人的结论，也不要偏信自己的猜想。一切知识的获取都需要通过实践验证。在这次探究经历中，就像我在第四模块实验体会中说的那样，有很多猜想很多理论在经过实验验证之后会发现有所出入（当然这里也不排除我的实验做的不够严谨这一可能）。但是很多新的发现就产生在这个“有出入”上面。我在这次研究报告上面花了非常多的时间，从选题到后续实验，但是我觉得非常值得。

Comments on language exploration (2/3)



1. 首先本次实验选题的过程占用了极长的时间，出现这一问题的原因主要是因为对于相关指令并不算熟悉的原因，一直觉得这个不合适那个不合适，但是直到我真正开始着手做，才发现原来即使这样一条数据类型转化指令里面同样有很多可以挖掘的地方。
2. 还记得老师在初讲这个探索实验时，谈到了结合之前学过的知识进行融会贯通，在这一实验中我充分体会到了这一点！！起初我在选择整型与浮点型转化的时候还觉得这个指令可能并不能做的很深入，而且我最开初关注的点为浮点到整型。但是，当我反过来想从整型到浮点型会出现什么问题时，那一瞬间，就像一束光一闪而现，我想到，在我们之前学习计组的时候再将浮点数的时候讲到过浮点数表示的缺陷，就是在数逐渐增大的时候，能够是的数据会逐渐稀疏，进而里面有些整数是无法表示的。那么我立刻想到，如果一个整数恰好在浮点数无法表示的位置，会发生什么情况呢？这种情况下处理器会怎么尽量减小误差呢？然后我继续推测，对于无法表示的数，当我们将指数提出来之后，剩下的其实就是一个小数部分大于23位的数，因此这同样是一个round问题，只不过从round到整数变为round到小数的第23位。后续的实验测试也证实了我的猜测。这个过程让我真正体会到了老师所说的融会贯通的感觉，虽然我离融会贯通还有很远的距离。但是这个过程确实也极大地增强了我的信心hhh，这可以说是我在做这个实验的过程中收获最大的一点了。

Comments on language exploration (3/3)



我一开始觉得自己选择的指令非常简单非常基础，会不会影响到我这一篇探索的深度，或者更加功利一点会不会影响到我的分数。但是在做完整个探索流程，包括从底层到后续实验，我相信在简单的论题上面也能开出深度的学术之花。

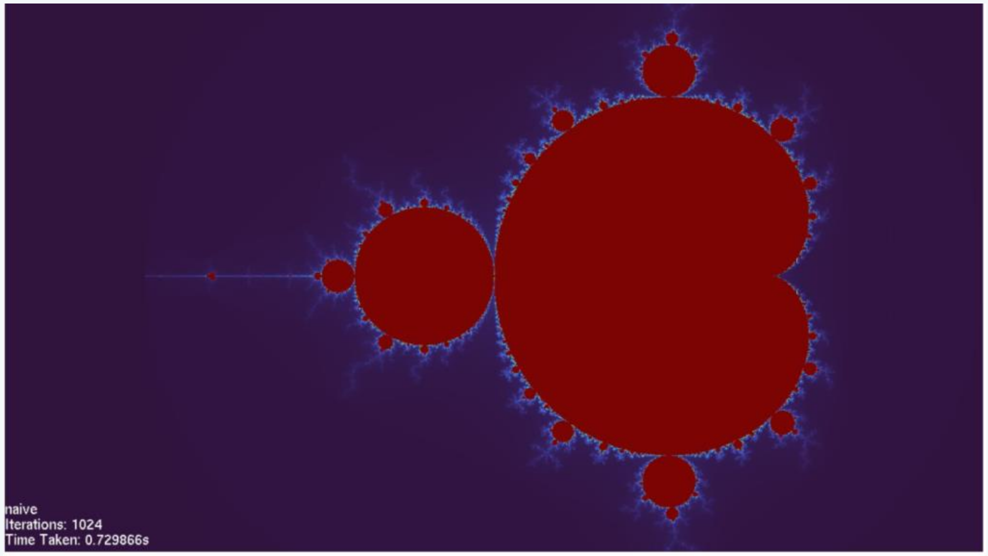
整个探究过程也是一个非常宝贵的经历。首先从日常学习中遇到的困惑出发，慢慢确立困惑来源，慢慢确立研究的问题，并对此进行探索，在探索的过程中冒出了很多猜想，然后再去设计实验验证这个猜想。同时，我在探索的过程中应用到了许多之前课堂里学过的知识，这也是让我非常惊喜的点。

Exploratory research2: SIMD exploration

- From your standpoint (e.g., SRTP, project)
  - Practicing SIMD to maximize performance (>= 10X speedup)
- From a compiler (GCC/LLVM/…) standpoint
  - Automatic vectorization in compiler (GCC, LLVM, ...)
  - Vectorization in programming language
- From a CPU standpoint
  - X86: SSE and AVX
  - RISC-V: “V” and “P” extension
  - ARM: NEON and SVE
- From a framework/library standpoint
  - NumPy, Eigen, OpenBLAS
  - OpenCV, Tencent/NCNN, Alibaba/MNN

SIMD exploration: sample 1

- Mandelbrot Set Visualization



```
naive
Iterations: 1024
Time Taken: 0.729866s
```

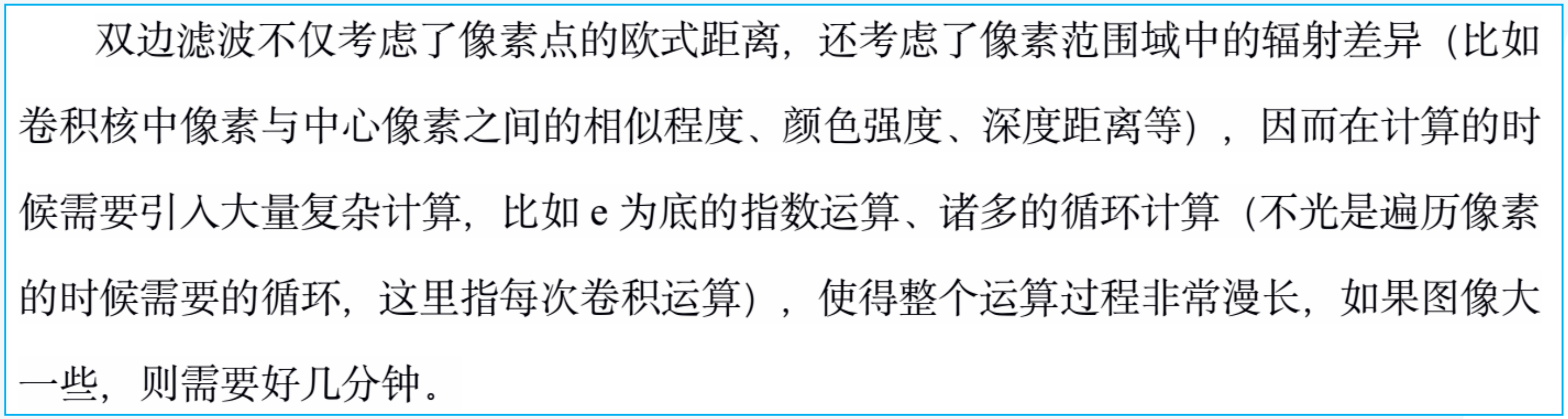
虽然效果还行，但是速度差强人意。
尤其是后期迭代加入缩放和拖移功能
后发现，程序响应速度极慢，宛
如崩溃前的系统UI界面一样。

然后剩下的就交给 MSVC 强大的 /O2 优化了。（至少上这门课之前我是这么想的）
结合使用 SIMD 和 multi-threading，应用延时基本满足了与用户实时交互的需求
渲染时间从最初的 730 ms 缩短到最终 23 ms，加速比达到了 32 倍。

SIMD exploration: sample 2

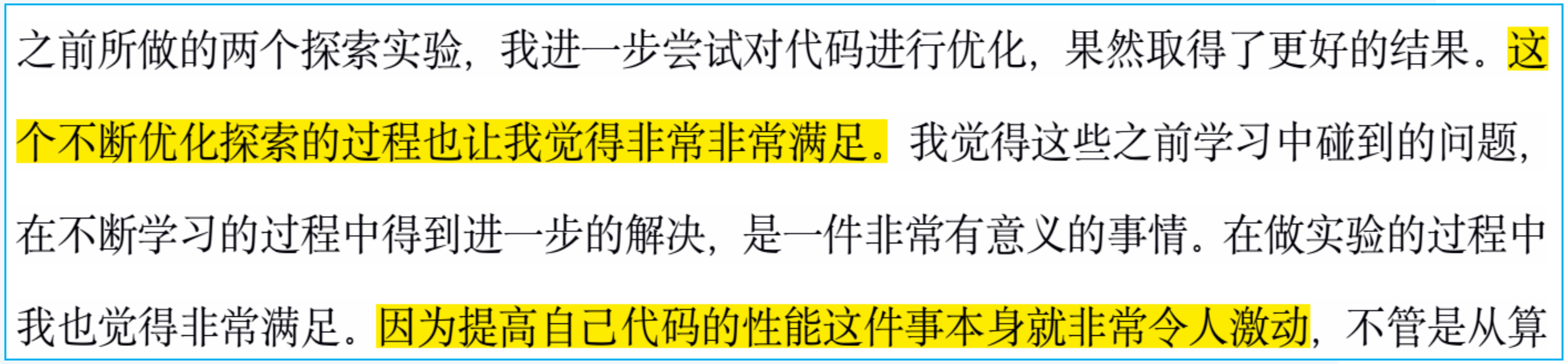
- 双边滤波算法在向量化编程下的运算速率优化

Question:



双边滤波不仅考虑了像素点的欧式距离，还考虑了像素范围域中的辐射差异（比如卷积核中像素与中心像素之间的相似程度、颜色强度、深度距离等），因而在计算的时候需要引入大量复杂计算，比如 e 为底的指数运算、诸多的循环计算（不光是遍历像素的时候需要的循环，这里指每次卷积运算），使得整个运算过程非常漫长，如果图像大一些，则需要好几分钟。

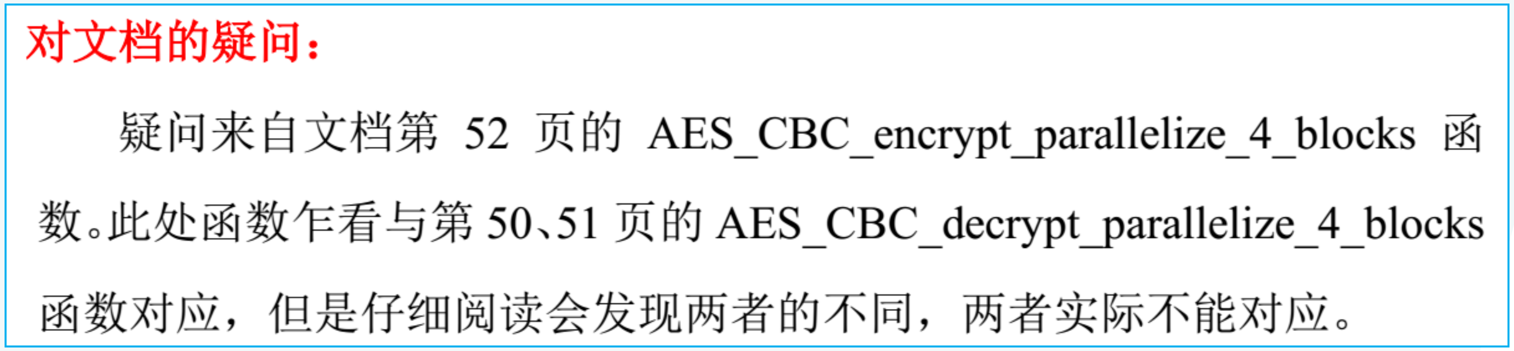
Conclusion:



之前所做的两个探索实验，我进一步尝试对代码进行优化，果然取得了更好的结果。这个不断优化探索的过程也让我觉得非常非常满足。我觉得这些之前学习中碰到的问题，在不断学习的过程中得到进一步的解决，是一件非常有意义的事情。在做实验的过程中我也觉得非常满足。因为提高自己代码的性能这件事本身就非常令人激动，不管是从算

SIMD exploration: sample 3

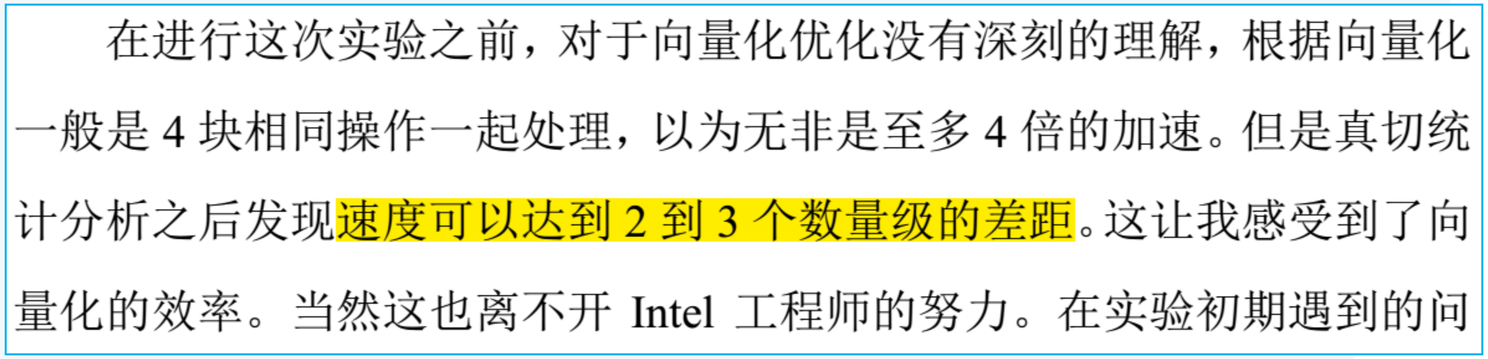
- AES加、解密算法的向量化加速



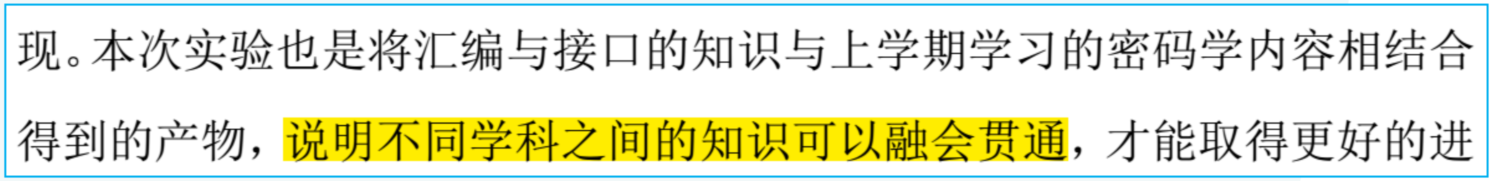
对文档的疑问：

疑问来自文档第 52 页的 AES_CBC_encrypt_parallelize_4_blocks 函数。此处函数乍看与第 50、51 页的 AES_CBC_decrypt_parallelize_4_blocks 函数对应，但是仔细阅读会发现两者的不同，两者实际不能对应。

实验体会:



在进行这次实验之前，对于向量化优化没有深刻的理解，根据向量化一般是 4 块相同操作一起处理，以为无非是至多 4 倍的加速。但是真切统计分析之后发现速度可以达到 2 到 3 个数量级的差距。这让我感受到了向量化的效率。当然这也离不开 Intel 工程师的努力。在实验初期遇到的问



现。本次实验也是将汇编与接口的知识与上学期学习的密码学内容相结合得到的产物，说明不同学科之间的知识可以融会贯通，才能取得更好的进

Steps for an exploratory research

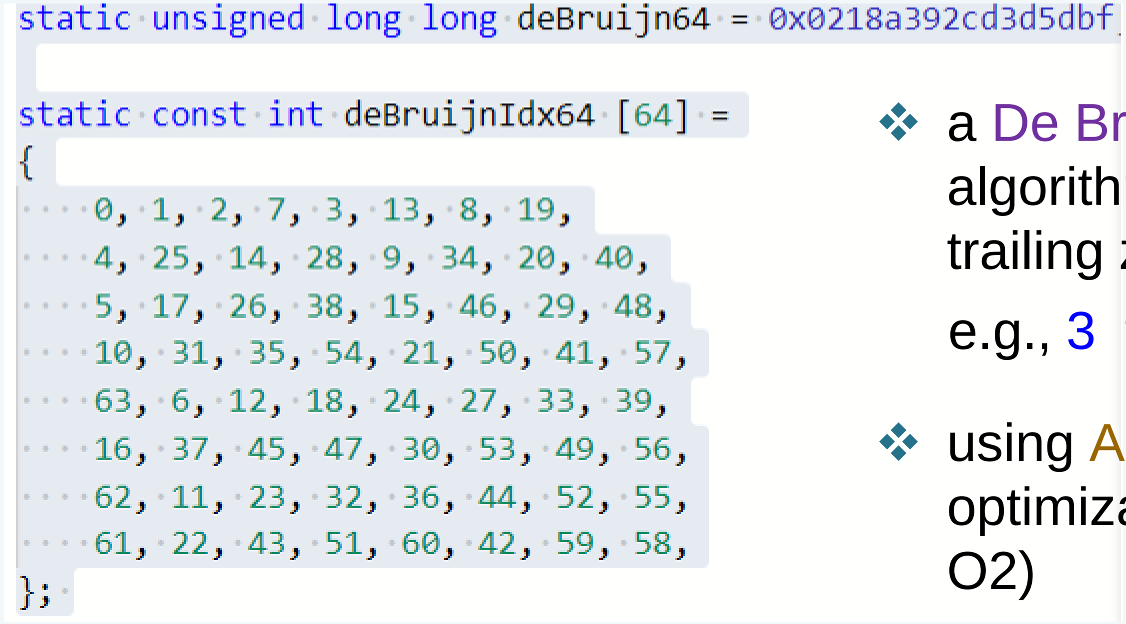
- How to conduct an exploratory research?
  - Narrow down your research question
  - Observation/motivation → assumption → method → evaluation
  - Integrate your previously learned knowledge for a comprehensive understanding
  - Use tools for analyzing effectively
  - Get your hands dirty with code
- Requirements
  - Work independent
  - Report
  - Deadline, before Nov.15
- Your research will be the source of our final examination!

Course Assessment

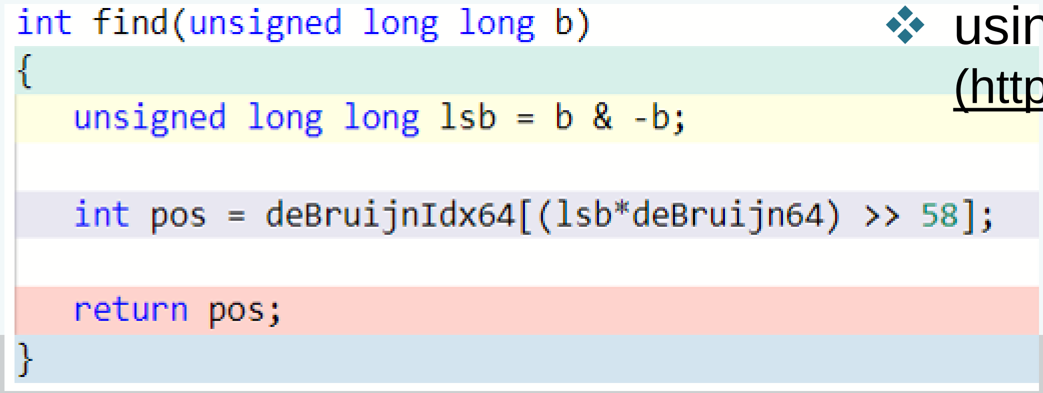
- **Exploratory research: 30%**
- **Final: 70%**
  - Score of Final Examination: ≥ 50

Appendix A: Peek into compiler's code

- To gain a better understanding of how a compiler works.



```c
static unsigned long long deBruijn64 = 0x0218a392cd3d5dbf

static const int deBruijnIdx64 [64] =
{
    0, 1, 2, 7, 3, 13, 8, 19,
    4, 25, 14, 28, 9, 34, 20, 40,
    5, 17, 26, 38, 15, 46, 29, 48,
    10, 31, 35, 54, 21, 50, 41, 57,
    63, 6, 12, 18, 24, 27, 33, 39,
    16, 37, 45, 47, 30, 53, 49, 56,
    62, 11, 23, 32, 36, 44, 52, 55,
    61, 22, 43, 51, 60, 42, 59, 58,
};
```



```c
int find(unsigned long long b)
{
   unsigned long long lsb = b & -b;

   int pos = deBruijnIdx64[(lsb*deBruijn64) >> 58];

   return pos;
}
```

- a De Bruijn sequence based algorithm to count number of trailing zeros,

  e.g., 3 for 01111000

- using ARM64 gcc with different optimization options (-O0, -O1, -O2)

- using compiler explorer (https://www.godbolt.org)

Reasons to learn assembly language (3/5)

- You will be better equipped to analyze bugs in programs.

```
struct __attribute__((__packed__)) packed_struct
{ char c; short i; int j; };
```

variable
assignment

```
struct  unpacked_struct { char c; short i; int j; };
```

- Debug edition (-O0)

LDNW: Load nonaligned word from memory

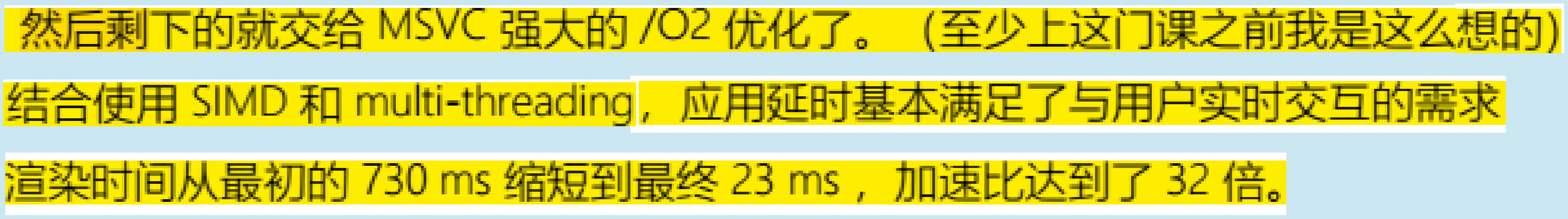
- Release edition (-O3)

LDW: Load word from memory being aligned

Assembler Directives

- Assembler directives are special instructions that provide information to the assembler but do not generate any code. Examples include the LABEL, EQU, ASSUME, and PROC. These mnemonics are not valid 80x86 instructions. They are messages to the assembler, nothing else.
- Three types of assembler directives in MASM:
  - Directives: .CODE, .DATA, .386, .586, LABEL, ORG, BYTE(DB), WORD(DW), PROC, STRUCT, IF, ELSE, etc.
  - Symbols: ?, @data, @code, @stack, $, @FileName, etc.
  - Operators: DUP, ‘’, ;, SEG, OFFSET, ==, >, !=, &&, ZERO?, SIGN?, CARRY?, OVERFLOW?, etc.

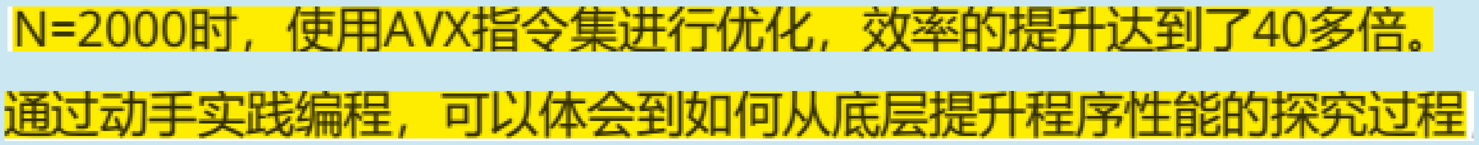
Comments on exploratory research



然后剩下的就交给 MSVC 强大的 /O2 优化了。 (至少上这门课之前我是这么想的)

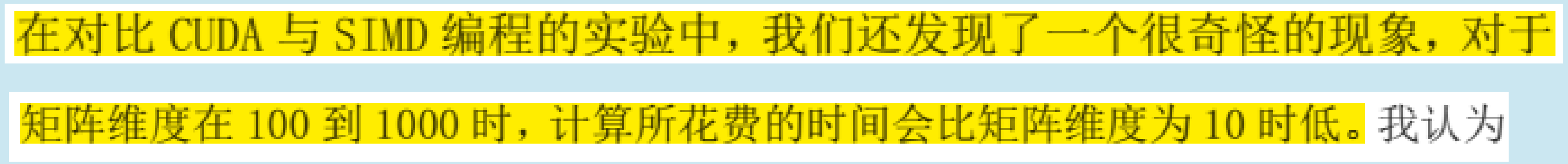
结合使用 SIMD 和 multi-threading，应用延时基本满足了与用户实时交互的需求

渲染时间从最初的 730 ms 缩短到最终 23 ms，加速比达到了 32 倍。



N=2000时，使用AVX指令集进行优化，效率的提升达到了40多倍。

通过动手实践编程，可以体会到如何从底层提升程序性能的探究过程



在对比 CUDA 与 SIMD 编程的实验中，我们还发现了一个很奇怪的现象，对于

矩阵维度在 100 到 1000 时，计算所花费的时间会比矩阵维度为 10 时低。我认为

Experiment (2/5)

- Exploratory research (mandatory)
  - Exploring Low-Level Programming With Assembly
  - Instruction exploration
  - SIMD exploration
  - I/O interface exploration (tentative)

- Why instruction exploration?
  - LLVM compiler: 1,115 contributors, 2.5 million lines (2013.12)
  - Only generates < 1000 out of 3,600 x86 instructions
  - Too many instructions and problems are worth to be explored!

Exploratory research1: instruction exploration

- From an xPU (Intel/ARM/RISC-V/GPU/TPU…) standpoint
  - x86: NOP, LEA, XCHG, CMOV……
  - GPU: FMA, DP4, HMMA, IMMA
  - Atomic instructions, Floating Point instructions
- From a compiler or virtual machine standpoint
  - Directive: .code, .align, org, assume, struct, ptr……
  - Code generation: switch, while, volatile, ……
  - Different optimization level: loop, ……
  - Python Virtual Machine (PVM), Java Virtual Machine (JVM), ...
- From your standpoint
  - Bug Reporting
  - Based on your experience