Enhancing SIMD Assembly Language Development with Visualization Techniques

A Thesis Submitted to the Department of Computer Science and Communications Engineering,
the Graduate School of Fundamental Science and Engineering of Waseda University
in Partial Fulfillment of the Requirements for the Degree of Master of Engineering

Submission Date: January 22nd, 2024

Zhong ZHONG
(5121F099–7)
Advisor: Prof. Kazunori Ueda
Research guidance: Research on Parallel Knowledge Information Processing

Abstract

As the demand for high-performance computing in today's society continues to increase, most common processors provide advanced parallel processing capabilities, such as Single Instruction Multiple Data (SIMD). Thanks to these new instruction sets, we can parallelize some computationally intensive tasks to improve the efficiency of program execution. However, the concept of vector computing by SIMD instruction set may be more difficult for developers to understand because it is not exactly the same as traditional scalar computation; Moreover, using SIMD instruction sets requires developers to have a deeper understanding of the underlying hardware, which adds difficulty to the use of SIMD instruction sets. Although modern compilers can automatically convert some codes into SIMD instructions to implement implicit automatic vectorization, their optimization capabilities are limited by multiple factors, and writing explicit vector code is still an important work in code optimization.
To address these challenges, this thesis presents a dynamic visualization method for SIMD assembly language code and its implementation: PixelAssemblySIMD. PixelAssemblySIMD provides a user-friendly interface. Developers can use this tool to represent SIMD instructions in the form of animation, which helps them understand how these instructions are executed in parallel in the processor.
Compared with other SIMD assembly language visualization tools, PixelAssemblySIMD has the following advantages:

- A universal visualization method, which can use the same animation logic to represent different SIMD instruction sets and SIMD instructions;
- Based on data flow rather than control flow, which is more in line with SIMD features;
- Supports a variety of different SIMD instruction sets, including SSE, AVX and AVX-512, etc.;
- Uses an independent CPU emulator named cpulib to ensure the correctness of CPU visualization data;
- Visualization method used in PixelAssemblySIMD is more appropriate for teaching and learning than compared to debugger type visualization.
- PixelAssemblySIMD supports cross-platform compatibility; it can be compiled into native binary files for multiple platforms, and it can even be compiled into WebAssembly to run directly in browsers.

In the user-based research evaluation, PixelAssemblySIMD and its visualization method have been proven to be a useful tool. The majority of developers believe that PixelAssemblySIMD is helpful in assisting them to understand the execution process of SIMD instructions, and it has advantages in representing complex SIMD instructions.

Keywords: Software visualization; SIMD; CPU emulation

Contents

Chapter 1 Introduction 1

1.1 Background and Purpose of the Research . . . 1
1.2 Contributions of the Research . . . 13
1.3 Structure of the Thesis . . . 14

Chapter 2 Research Background 16

2.1 CPU Emulator . . . 16
2.2 Rust Programming Language . . . 19
2.3 GUI . . . 20

Chapter 3 Design of Visualization Method and PixelAssemblySIMD 24

3.1 Evolution of Design . . . 25
3.2 Visualization Approach for SIMD Instruction . . . 37
3.3 Design of PixelAssemblySIMD . . . 41

Chapter 4 Implementation of PixelAssemblySIMD 51

4.1 CPU Emulator: cpulib . . . 51
4.2 GUI . . . 60
4.3 Animation Executor . . . 71
4.4 Instruction Executor . . . 78

Chapter 5 Evaluation 82

5.1 Evaluation Methods . . . 82
5.2 Evaluation Result . . . 85
5.3 Supplementary Evaluation . . . 101
5.4 Examples . . . 103

Chapter 6 Comparison with Related Research 111

6.1 SIMDGiraffe . . . 111
6.2 SIMD-Visualiser . . . 112

6.3 NEVADA 116

Chapter 7 Conclusions and Further Work 118

7.1 Conclusions 118

7.2 Further Work 119

Acknowledgements 121

References 122

Appendix A Source Code Structure 125

A.1 cpulib 125

A.2 PixelAssemblySIMD 125

List of Figures

1.1 Comparison between SISD and SIMD 4
1.2 Visual representation of the `vperm2f128` instruction 11

3.1 Legend used in the visualization description 25
3.2 Visual representation of GPR or element in vector registers 26
3.3 Visualization of the registers in Method1 27
3.4 Method1 instruction visualization animation: move operation 28
3.5 Method1 instruction visualization animation: assignment operation 29
3.6 Method1 instruction visualization animation: exchange operation 30
3.7 Design of Prototype1 32
3.8 Legend explaining graphical primitives for GUI design 33
3.9 GUI design of Prototype1 34
3.10 State diagram of Prototype1 GUI 34
3.11 Component Design of Prototype1 36
3.12 Different data widths in Method2 38
3.13 Visual representation of registers in Method2 38
3.14 Animation design of instruction visualization in Method2 - Start stage 39
3.15 Animation design of instruction visualization in Method2 - Execute stage 40
3.16 Animation design of instruction visualization in Method2 - End stage 41
3.17 Animation design of calculation instruction visualization in Method2 - Execute stage 42
3.18 Design of Prototype2 43
3.19 GUI Design of Prototype2 45
3.20 Design of cpulib 47
3.21 Animation executor based on egui in Prototype2 49
3.22 Instruction Executor in Prototype2 50

5.1 Implementation of the Survey Questionnaire 85
5.2 Basic Information of Survey Respondents 86

5.3 Survey Respondents' Understanding of Advanced Programming Languages 88
5.4 Influence of Programming Difficulty on Respondents' Understanding of Advanced Programming Languages 89
5.5 Understanding of Advanced Programming Languages by Professional Programmers 89
5.6 Influence of Programming Difficulty on the Understanding of Advanced Programming Languages by Surveyed Professional Programmers 90
5.7 Understanding of Advanced Programming Languages by Non-Professional Programmers 91
5.8 Influence of Programming Difficulty on the Understanding of Advanced Programming Languages by Surveyed Non-Professional Programmers 92
5.9 Understanding of Advanced Programming Languages by People Who Have Not Studied Programming 93
5.10 Influence of Programming Difficulty on the Understanding of Advanced Programming Languages by People Who Have Not Studied Programming 94
5.11 Survey Respondents' Understanding of Assembly Language 94
5.12 Understanding of Assembly Language by Professional Programmers 95
5.13 Understanding of Assembly Language by Non-Professional Programmers 95
5.14 Understanding of Assembly Language by People Who Have Not Studied Programming 95
5.15 Proportion of Respondents Who Can Understand Static Visualization of SIMD Computations 96
5.16 Help of Static Visualization in Understanding SIMD Computations 96
5.17 Proportion of Respondents Who Can Understand Static Visualization of SIMD Permuting 96
5.18 Help of Static Visualization in Understanding SIMD Permuting 97
5.19 Proportion of Respondents Who Can Understand Dynamic Visualization of SIMD Computations 98
5.20 Help of Dynamic Visualization in Understanding SIMD Computations 98
5.21 Proportion of Respondents Who Can Understand Dynamic Visualization of SIMD Permuting 99
5.22 Help of Dynamic Visualization in Understanding SIMD Permuting 99
5.23 The Degree of Assistance Provided by Static and Dynamic Visualization in Understanding Different Levels of Code Complexity 100

5.24 Preferences of All Respondents for Visualization Methods 100
5.25 Preferences of Professional Programmers for Visualization Methods 101
5.26 Supplementary Evaluation: Overall 102
5.27 Supplementary Evaluation: GUI 103
5.28 Visualization process of `valignd zmm1, zmm0, zmm2, 15` 104
5.29 Visualization results of the Prefix Sum algorithm 105
5.30 Visualization results of the Matrix Transpose algorithm 108
5.31 Visualization results of the Matrix Multiplication algorithm 109
5.32 Visualization process of `vfmadd213pd ymm6, ymm2, ymm7` 110

6.1 Visualization of the Prefix Sum Algorithm in SIMD-Visualiser 113
6.2 AST generated by SIMD-Visualiser 114
6.3 NEVADA 117

Source Code Listings

1.1 Code for executing `vperm2f128` instructions using assembly language 7
1.2 Code for executing `vperm2f128` instruction with C language and intrinsic functions 8
4.1 External Interface of cpulib (Structure) 51
4.2 Implementation of the external interface of cpulib 51
4.3 Structural Definitions of Registers in cpulib 52
4.4 Implementation of Bit Read and Write in `SIMDRegister` of cpulib 53
4.5 Implementation of Register Read and Write in `SIMDRegister` of cpulib 53
4.6 Implementation of Register Read and Write in `GPR` of cpulib 54
4.7 Implementation of Reading Vector Register Bits in `Registers` of cpulib 54
4.8 Implementation of Reading Vector Registers in `Registers` of cpulib 55
4.9 Implementation of Reading GPRs in `Registers` of cpulib 56
4.10 Macro for Reading GPRs in `Registers` of cpulib 56
4.11 Implementation of Reading `RIP` and `RFLAGS` in `Registers` of cpulib 57
4.12 Structural Definitions of Memory in cpulib 58
4.13 Implementation of Byte Read and Write in `Memory` of cpulib 58
4.14 Implementation of Data Read and Write in `Memory` of cpulib 59
4.15 `main` function of PixelAssemblySIMD 60
4.16 Definition of the `APP` structure in PixelAssemblySIMD 61
4.17 Implementation of `update` in `APP` structure 62
4.18 Windows in PixelAssemblySIMD 63
4.19 Definition of the `VisualizerSetting` structure in PixelAssemblySIMD 64
4.20 Definition of the `Element` structure in PixelAssemblySIMD 65
4.21 Implementation of `show` in `Element` structure 66
4.22 Definition of the `RegVisualizer` structure in PixelAssemblySIMD 67
4.23 Implementation of `show` in `RegVisualizer` structure - Create Layout 67
4.24 Implementation of `create_layout` in `RegVisualizer` structure 69
4.25 Implementation of `create_elements` in `RegVisualizer` structure 69
4.26 Implementation of `show` in `RegVisualizer` structure - Show Elements 70

4.27 Implementation of `show_element!` in PixelAssemblySIMD 70
4.28 Implementation of `update` in `Element` structure 72
4.29 Implementation of `update` in `RegVisualizer` structure 72
4.30 Creation of Animation Sequence 73
4.31 Execution of Animation Sequence 74
4.32 Execution of Animation Sequence - **ExecuteAnimation** Signal 74
4.33 Execution of Animation Sequence - **Terminate** Signal 75
4.34 Execution of Animation Sequence - Animation Group 75
4.35 Execution of Animation Sequence - Individual Animation 76
4.36 End Detection of Animation Sequence 77
4.37 The status of FSM in Instruction Executor 78
4.38 Implementation of the FSM in Instruction Executor 79
4.39 Execution of the FSM in Instruction Executor 79
5.1 Prefix Sum algorithm (`_start` Part) 103
5.2 Prefix Sum Algorithm (Algorithm Part) 104
5.3 Matrix Transpose Algorithm (Data Declaration) 105
5.4 Matrix Transpose Algorithm (`_start` Part) 106
5.5 Matrix Transpose Algorithm (Algorithm Part) 106
5.6 The Matrix Multiplication algorithm 107
6.1 Example Code for SIMD-Visualiser: Prefix Sum Algorithm 112

List of Tables

1.1 common SIMD instruction sets in the x86_64 architecture 5

2.1 Common CPU Emulators 17
2.2 GPRs Supported by The x86_64 Architecture 18

1

Introduction

This chapter not only states the background and purpose of the research but also presents a comprehensive introduction to the research results. In the first section, the background and purpose of this study are introduced, and the problem is analyzed at both technical and philosophical levels. Then the achievements and innovations of this study are summarized. Finally, the structure of this thesis is introduced in the third section.

1.1 Background and Purpose of the Research

1.1.1 Background of the Research

Assembly Language

Utilization of Assembly Language in Programming With the development of high-level programming languages and modern compiler technology, the direct application of assembly language is decreasing. As a normal developer, it is difficult to encounter code that uses assembly language; even operating system developers rarely use assembly language for development. Most operating systems are developed in high-level languages like C, and assembly language is only introduced when there is a need to initialize hardware.

As analyzed in A. Fog. Optimizing subroutines in assembly language: an optimization guide for x86 platforms, 2008, there are many reasons why we are not inclined to use assembly language for development today, including:

- *Development level*: Using assembly language means it requires more development time and maintenance cost; meanwhile, debugging and validation of the program become more difficult.
- *Software level*: As assembly language closely related to machine language and being a low-level

programming language, the software written in assembly language is generally poor in portability. Also, because there is no systematic method for testing and validation, the reliability and safety of the software written in assembly language are often an issue.

- *Alternatives*: With the development of modern compilers, the performance of software developed in high-level languages is often sufficient to meet the requirements. And with the advent of intrinsic functions, system-level code can also avoid using assembly language directly through intrinsic functions.

However, as pointed out in this book, assembly language is still crucial and irreplaceable in certain fields, such as:

- *System-level code*: Embedded systems, operating system code, drivers, and other areas that need direct hardware manipulation. In these areas, assembly language is often indispensable, as high-level languages typically cannot provide the capability to manipulate hardware.
- *Software optimization*: If you want to optimize code in terms of time and space at a deeper level, assembly language is needed. Because although modern compilers can optimize code in many cases, their optimization capabilities are still limited in some extreme examples.
- *Special purposes*: Self-modifying code, compiler writing, and other special-purpose code often need to use assembly language.
- *Teaching and Research*: Assembly language is often used for education and research because it can help students and researchers better understand how computers work. Observing assembly language snippets generated by the compiler can also be used to learn principles of compiler operation and optimization methods.

In addition, assembly language is essential in the field of software security analysis, vulnerability mining, reverse analysis, etc. It is commonly used for software analysis because assembly language code is closer to machine language and easier to analyze.

Complexity of Assembly Language Programming Assembly language is known for its proximity to machine code, posing huge challenges even for experienced programmers. The complexity of assembly language can be reflected in the following points:

- *Low level and hardware dependency*: Assembly language provides direct control over hardware, including processor registers, instruction sets, etc. This low-level control means that programmers need to have a deep understanding of the specific hardware that runs the code. Different processor architectures have different assembly languages, so assembly language has strong plat-

form dependence.

- *Learning of mnemonics and instruction sets*: Assembly language uses mnemonics to represent machine instructions. Programmers need to learn and memorize these mnemonics and their usage methods, and their number is much more than the keywords in high-level languages. The instruction set of each processor architecture may be different, so a deep understanding of the specific instruction set is also needed.
- *Difficulty of debugging and maintenance*: Because assembly language is very close to machine code, debugging assembly programs is usually difficult. When an error occurs, the error information may not be as intuitive as in high-level languages. Also, assembly code maintenance is relatively difficult because the code is usually not as readable as in high-level languages.
- *Manual resource management*: In assembly language programming, programmers need to manually manage all resources, including memory allocation and register use. This increases the complexity of programming, especially when managing large programs.
- *Lack of high-level abstraction*: High-level programming languages provide abstractions such as functions, classes, and objects, making code easier to write and understand. Assembly language does not have these abstractions, making the implementation of complex logic more difficult.
- *Complexity of performance optimization*: While assembly language allows for fine performance optimization, this usually requires programmers to have a deep understanding of processor performance characteristics, such as how the cache works, the execution cycle of instructions, etc.
- *Portability issues*: Because assembly language is closely dependent on a specific hardware platform, porting assembly code between different platforms is very difficult.

It can be seen that writing or even understanding assembly language program has a certain complexity.

Single Instruction Multiple Data(SIMD)

Introduction of SIMD Single Instruction Multiple Data (abbreviated as SIMD) computing is a commonly used parallel processing technique in modern computer architectures that allows for basic operations to be performed on multiple data elements in parallel with fewer instructions to accelerate application performance[2]. In the case of calculating intensive sets, programs using SIMD instructions can run much faster than traditional ones using scalar calculations[3]. For instance, a program using SIMD instructions in the Prefix Sum algorithm can be about three times faster than an unoptimized one[4]. Currently, SIMD has a wide range of applications in performance-demanding programs such as multimedia processing, data security, database and scientific computing[2].

Figure 1.1 shows a comparison of addition calculations. On the left is an example of SISD (Single In-

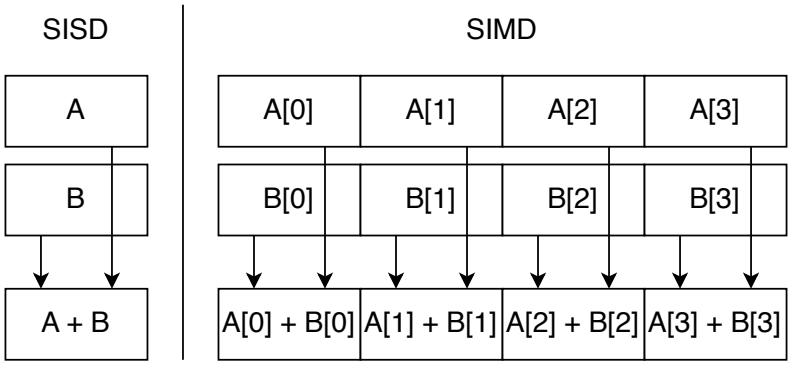


Figure 1.1: Comparison between SISD and SIMD

struction, Single Data), while on the right is an example of SIMD. SISD can only perform one addition at a time, but SIMD can perform four additions simultaneously. Such data parallelism can significantly speed up program performance to a certain extent.

At present, there are generally three methods to apply SIMD computation in programs:

1. Writing programs in high-level languages, then using the compiler for automatic SIMD optimization;
2. Writing programs in high-level languages, then using intrinsic functions for explicit optimization;
3. Writing assembly language programs directly using SIMD instruction sets.

As mentioned earlier, with the rapid development of compiler technology, automatic SIMD implemented by the compiler is sufficient for general optimization; however, for performance-chasing applications, explicit SIMD instructions are sometimes needed, using intrinsic functions or even assembly language for optimization.

SIMD Instruction Sets in x86_64 Architecture Table 1.1 lists the common SIMD instruction sets in the x86_64 architecture and their descriptions.

Table 1.1: common SIMD instruction sets in the x86_64 architecture

Instruction Set	Time	Register	Width	Description
MMX	1996	MM0-MM7	64bits	MMX(MultiMedia Extensions) is one of the earliest SIMD instruction sets, mainly used to accelerate multimedia tasks, such as video processing, image processing, and audio processing, etc. MMX uses the registers of the existing floating-point unit.
SSE	1999	XMM0-XMM15	128bits	SSE(Streaming SIMD Extensions) is an extension to MMX, adding support for single-precision floating-point numbers. It has gone through multiple versions, from SSE1 to SSE4.2, and the XMM registers have expanded from the initial 8 to 16.
AVX	2011	YMM0-YMM15	256bits	AVX(Advanced Vector Extensions) extends the register width of SSE, supporting more advanced floating-point operations. Includes AVX, AVX2. The lower 128 bits of YMM registers are XMM registers.
AVX-512	2013	ZMM0-ZMM31	512bits	Extended the register width of AVX, expanded the number of vector registers to 32, supporting more advanced floating-point operations. The lower 256 bits of the ZMM register are YMM registers.

Table continued on next page 1.1 common SIMD instruction sets in the x86_64 architecture

Continued on next table 1.1 common SIMD instruction sets in the x86_64 architecture

Instruction Set	Time	Register	Width	Description
FMA	2013	Vector registers	-	FMA(Fused Multiply-Add) provides compound multiply-add instructions, able to complete multiplication and addition in one operation, improving computational performance.

End of table 1.1 common SIMD instruction sets in the x86_64 architecture

Challenges of Implementing SIMD in Programs Apparently, SIMD computing offers significant performance advantages, especially when dealing with large amounts of data, but this advantage comes at the expense of increased programming complexity and also makes understanding and debugging programs more difficult[5][6]. Specifically, using SIMD computing encounters challenges in the following aspects:

- *Data alignment*: Accessing memory blocks at locations that are not aligned on the natural vectorsize boundary is usually forbidden or results in severe performance penalty. This alignment constraint significantly affects the effectiveness of SIMD vectorization and is also one of the challenges encountered in writing compilers that automatically use SIMD extensions[2].
- *Hardware dependencies*: Different hardware platforms provide different SIMD instruction sets, making it difficult to port programs using SIMD instructions across different hardware platforms.
- *SIMD implementation limitations*: To make SIMD instructions perform at their best, programmers must assess the program segments to be implemented in SIMD computations in at least the following four aspects: (1) The program segment must be computationally intensive; (2) The execution frequency of program segments is enough to affect performance; (3) The program segment cannot depend on the data's control flow, if it does, it needs to be fragmented; (4) The program segment needs help to effectively use the cache hierarchy. These assessments require a deep understanding of the program's structure and the hardware's characteristics by the programmers, which makes implementing SIMD computations a challenging task[2].
- *Programming education*: Teaching SIMD programming skills is also challenging because most textbooks don't cover practical knowledge of the SIMD execution model[7].

Utilization of SIMD in Assembly Language

It's not hard to see that using assembly language and SIMD computing are both great ways to optimize the time and space performance of a program, and coding SIMD in assembly language is still the most effective method. Although using SIMD instructions in assembly language is the most difficult programming method, it offers the best performance[2]. Hence, situations that require dealing with the complexity of assembly language and SIMD computation are common, and easily encountered in software security analysis, vulnerability exploration, reverse analysis, and other fields.

Programming in assembly language using SIMD instructions is a relatively challenging task. I will show you this challenge with an example. First, let's consider the assembly language code shown in Listing 1.1.

Listing 1.1: Code for executing `vperm2f128` instructions using assembly language

```
1  section .data
2  array1 dq 1.0, 2.0, 3.0, 4.0
3  array2 dq 5.0, 6.0, 7.0, 8.0
4  array3 dq 0.0, 0.0, 0.0, 0.0
5
6  section .text
7  global _start
8  _start:
9      vmovapd ymm0, [array1]
10     vmovapd ymm1, [array2]
11     vperm2f128 ymm2, ymm0, ymm1, 0x21
12     vmovapd [array3], ymm2
13     ; exit system call
14     xor rdi, rdi
15     mov rax, 60
16     syscall
```

The purpose of this code is to arrange the data in two arrays according to certain rules, and then store the result in the `YMM2` register. The `vperm2f128` instruction is used in the code. This is a SIMD instruction. It arranges the data in the vector registers shown by the second and third operands according to the immediate number in the fourth operand as control bits, and then stores the result in the vector register indicated by the first operand.

Now the question is: What is the result of the `vperm2f128` instruction (i.e., the value in the `YMM2` register)? It is not difficult to find that if we only refer to this code, we can hardly understand its function. Because we can't clearly know the function of the `vperm2f128`instruction through this code: we only know that the operating register is `YMM0` and `YMM1`, and the result is placed in `YMM2`, but we know nothing about the role of control bits. If we want to understand the function of this code, we

must have a certain understanding of the assembly language, the architecture of the computer, and the working principle of SIMD instructions and the function of each instruction (and what their control bits will control).

So, can we understand this code and its SIMD instructions more easily in high-level language? In Listing 1.2, I give the code with the same function as in Listing 1.1 using C language and Intel's intrinsic function.

Listing 1.2: Code for executing `vperm2f128` instruction with C language and intrinsic functions

```
#include <stdio.h>
#include <immintrin.h>
int main() {
    __m256d array1 = _mm256_setr_pd(1.0, 2.0, 3.0, 4.0);
    __m256d array2 = _mm256_setr_pd(5.0, 6.0, 7.0, 8.0);
    __m256d array3 = _mm256_permute2f128_pd(array1, array2, 0x21);
    ; print the values in array3
    double* res = (double*)&array3;
    for (int i = 0; i < 4; ++i) {
      printf("%lf\n", res[i]);
    }
    return 0;
}
```

It can be seen that this code is more concise and easier to understand than the code in Listing 1.1. However, for the SIMD operation in this code, we still can't directly know what the `vperm2f128` instruction does, because we still don't know the role of the control bits. Therefore, it can be said that the intrinsic functions provided by high-level languages also do not solve the difficulty we have in understanding SIMD instructions.

One can see that assembly language has brought great challenges to programmers' cognition. Adding the need to deal with SIMD computations doubtlessly further complicates this situation. Although SIMD allows multiple data points to be processed in parallel with a single instruction and that's efficient, this is somewhat at odds with humans' cognitive model that tends to focus on single data elements. This discrepancy poses a conceptual hurdle to programmers; this complexity is not just a technical obstacle, but it also represents the cognitive gap between human intuitive processing and machine logical operations. Now I will delve into this and attempt to propose a new solution.

1.1.2 Cognitive Dissonance Between Humans and Computers

Understanding the Cognitive Challenges Posed by SIMD Computation

Here we discuss centering around SIMD, in an attempt to reveal the fundamental difference between humans and computers in the way they process information. In the use of SIMD, humans tend to focus on individual elements and their changes, while computers focus on efficiently processing the entire datasets. This disparity indeed can lead to cognitive conflicts, especially when designing and understanding complex computational systems.

Cognitive Dissonance in Programming The challenge to understand SIMD programming, decipher assembly language, and even grasp the application of SIMD instructions in assembly language all stem from the cognitive dissonance between human and machine processing. Humans tend to view data and tasks scalars and step-by-step, focusing on single elements and their transformations. Machines, specifically under the context of SIMD operations, can process data in bulk, without distinguishing their constituent parts. This difference sparks philosophical questions on the nature of computation, as well as the interface between human thinking and machine processing. Programmers must reconcile the human-centric data view with the machine-oriented computational methods (referring here to SIMD computation methods here).

Analogy of Thinking Processes If we regard the process the computer uses to execute commands as a sort of thinking process, then the cognitive contradiction between humans and machines arises: humans care more about how individual scalars within a vector change, while computers are more eager to know how they should accurately compute a set of vector – they do not care about the function of individual scalars, because they do not need to ”understand” the program. However, one question arises: can we consider the process computers use to execute commands as a ”thinking process”? This is a philosophical question concerning the definition of ”thinking.” Computer processes are strictly preset and programmed, while human thinking is more complex and flexible. Computers may not have real ”thinking,” but they indeed exhibit a ”decision-making process” when executing tasks. Moreover, the root cause of this contradiction is not the difference between human thinking and the computer’s ”decision-making process,” but the discrepancy of the human cognitive role.

Inner Contradictions of Cognition and Dual Nature of Programmer Cognition The contradiction mentioned above actually originates from the conflict between different cognitive roles within humans themselves, rather than a direct opposition between humans and computers.

This concept implies that in understanding and operating computers, we need to simultaneously acknowledge and deal with these different cognitive patterns. Programmers often oscillate between these two realms: they need to explore the function of programs from a human-centric perspective, and need to switch to a machine-centered perspective when writing programs and considering how computers will solve problems. This binary nature is not only a technical skill but also a cognitive exercise, requiring programmers to internalize and switch between different thought patterns.

"Programming Thinking" and its Impact on Humans A competent programmer should be able to seamlessly switch between these two states, which we can call "Programming Thinking". However, for beginners, smoothly transitioning between these cognitive modes is challenging because the "Programming Thinking" differs from cognition in everyday human life. Studies found that people with high programming experience might develop a computational form of thought that can be used for complex problem-solving tasks. In certain deductive tasks, experienced programmers exhibit more efficient neural processing and improved performance. This suggests that programming experience may be related to neural efficiency in programming-related psychological processes, such as pattern recognition and algorithmic thinking[8]. The "computational way of thought" mentioned here and the resulting "human psychological processes and thinking" are referred to as "Programming Thinking". These experiments already show some differences exist between Programming Thinking and regular human cognition, and these differences can impact how humans understand computers—even how they think and process thoughts in general.

Visualization: An Auxiliary Method for Bridging the Cognitive Gap Among Programmers

Assisting Human Cognition: A Necessary Approach Given beginners' difficulty in quickly transitioning between cognitive modes, tools helping human cognition should be available. These tools not only lower the entry barrier for novices but also enable advanced programmers proficient in coding to complete tasks more leisurely without having to frequently switch cognitive modes during the thought process. These tools can be programming languages, programming environments, and also programming auxiliary tools, like the visualization tool proposed in this thesis. While visualization has far lower information-delivering efficiency than writing code, it is indisputable that visualization methods are more intuitive to humans and can aid programmers in better understanding programs and algorithms[9].

Exploring the Concept of Visualization Visualization refers to a method using graphics or images to represent and display information, data, or programs[9]. Its aim is to make complex data easier to understand and interpret by visual representation. This technology is widely applied in numerous fields, including science, engineering, medicine, education, and business. Visualization can also be used in the programming realm, including visual programming and program visualization. The latter uses graphs or animations to show program execution, data structures, or algorithms to help programmers understand and debug programs. By leveraging the advantages of the human visual system, more intuitive, easier-to-grasp, and operable programming environments are provided[9]. In computer science education, visualization has been proven an effective means to understand and learn abstract and complicated concepts[10]. For programs using SIMD computation, graphical visualization reduces the difficulties in understanding and debugging[5].

At present, the application of visualization in programming is increasing. Scratch[11] and Blockly[12] are frequently-used visual programming tools while Alice[13] and others serve as program visualization tools, all designed to guide beginners in understanding programming terms and simplifying programming processes.

Using Visualization to Decode and Explain Code To prove that visualization does make up for the cognitive gap of programmers to some extent, and serve as an auxiliary cognitive tool for beginners, educators, and even expert programmers, let's return to the code in Listing 1.1. This time I will use a rudimentary method of visualization to illustrate the SIMD operations in the code. We can observe whether the visualizing method used in Figure 1.2 can reveal the specific function of the `vperm2f128` instruction in the code more simply.

From the static visualization of the given `vperm2f128` instruction, we can easily see the specific op-

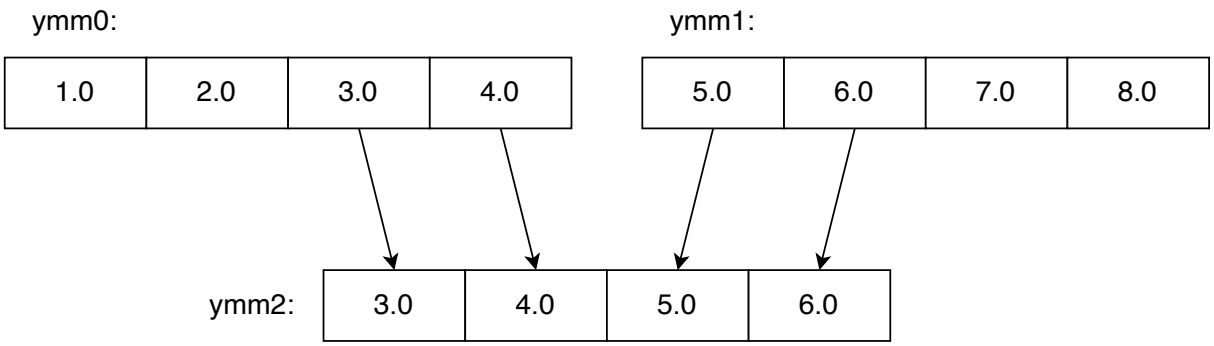


Figure 1.2: Visual representation of the `vperm2f128` instruction

eration this instruction will perform when the control bit is `0x21`, and easily derive the result stored in the `YMM2` register. Through static visualization, we can learn that when the control bit is `0x21`, the

lower 128 bits of the result come from the upper 128 bits of the second operand, and the upper 128 bits of the result come from the lower 128 bits of the third operand. Such understanding is clearly impossible to perform and comprehend through the operand "`0x21`". It is obvious that the intuitiveness of visualization methods helps humans better understand the functions of programs and SIMD instructions. However, there are still some limitations:

- This visualization method is only suitable for specific instructions. For other instructions, we also need to redesign the visualization method;
- This visualization method can only show the function of specific control bits of specific instructions. For control bits with other values, we need to redesign the visualization method;

These limitations make this visualization method unable to be applied to more complex code and more complex instructions. However, the emergence of this visualization method still proves that visualization methods can indeed bridge the gap in programmer cognition to a certain extent, and proves that visualization methods can indeed serve as an auxiliary method to help humans better understand programs and algorithms.

1.1.3 Purpose of the Research

The main goal of this research is to propose a universal data flow-based visualization method and develop its prototype tool, PixelAssemblySIMD, to unveil the mystery of SIMD computation in assembly languages, and to avoid the drawbacks mentioned in static visualization. PixelAssemblySIMD aims to bridge the aforementioned cognitive gap by transforming machine-centered SIMD operations into a format that is more intuitive for human programmers. This visualization tool can assist programmers in the following areas:

- *Enhancing programming education and practice*: This tool is expected to contribute to programming education by more closely integrating with human cognitive processes. Such consistency could make learning more efficient and intuitive, especially for beginners who have just been introduced to concepts such as SIMD and assembly language; for those who want to understand the workings of the computer, this visualization method is a good way to understand assembly code; for those who want to understand SIMD computations, this visualization method can help them understand the workings of SIMD computations better.
- *Diagnostic tool*: Debugging SIMD programs and assembly language programs is a difficult task, but this tool can also be used as a diagnostic tool for programs, thereby identifying and resolving

issues in SIMD code more quickly through visualization methods.

- *Applications in security and code analysis*: This visualization method also impacts security professionals, especially in the analysis of disassembled code. By providing a clearer understanding of the SIMD computation process and assembly code flow, it can help with disassembly vulnerability assessment and debugging; or when analyzing code, it can better and more quickly understand the data control flow of the code through visualization methods.

1.2 Contributions of the Research

This research mainly has the following contributions:

1. This study introduces a dynamic visualization method that can translate SIMD instructions in assembly language into a format that human programmers can understand more intuitively, thereby bridging the cognitive gap between humans and computers. This visualization method has a certain universality in describing instruction behavior and can use the same operation logic to display the behavior of different types of instructions. Meanwhile, this method is based on data flow rather than control flow to fit the characteristics of SIMD.
2. This research implements a prototype tool PixelAssemblySIMD. This tool, after several iterations, implements the aforementioned visualization method and also implements a CPU emulator based on the Rust language, and proposes a method for implementing asynchronous behavior animation in synchronous systems based on the immediate mode graphical user interface. Furthermore, PixelAssemblySIMD supports cross-platform compatibility; it can be compiled into native binary files for multiple platforms, and it can even be compiled into WebAssembly to run directly in browsers.
3. This study conducted a user-based evaluation of the proposed visualization method. The evaluation results show that the method is effective in assisting humans (including programming learners and those who do not understand programming at all) in understanding assembly instructions, with 34.3% (for simple instructions) and 42.2% (for complex instructions) respondents acknowledging its effectiveness; whereas when assisting programmers in understanding assembly instructions, 88.2% (for simple instructions) and 95.0% (for complex instructions) of programmers believe it to be effective.

Compared to other SIMD instruction visualization methods, the visualization method and prototype tool proposed in this study have several advantages:

1. The visualization method proposed in this study has a certain degree of universality, it can use the same operation logic to display behaviors of different types of instructions, while other SIMD instruction visualization methods may need to design different visualization methods for different types of instructions.
2. The visualization method proposed in this study is based on data flow, which is more in line with the characteristics of SIMD; while other SIMD instruction visualization methods are mostly based on control flow.
3. Compared to other tools created using frontend technology, PixelAssemblySIMD provides better data accuracy. PixelAssemblySIMD uses a standalone CPU emulator to ensure data accuracy, while other tools based on frontend technology might produce inaccurate data due to the characteristics of JavaScript.

1.3 Structure of the Thesis

The structure of this thesis is as follows:

- Chapter 1*Research Introduction*. This section first states the background and purpose of this study, and outlines the contributions and innovations of the research in the second section. In the last section of this chapter, the structure of this thesis and the role of each chapter are detailed.
- Chapter 2*Research Background*. This section introduces the technologies and professional terminology used in this thesis, as well as the technologies and code libraries used in Chapter 4.
- Chapter 3*Design and Proposal*. This section proposes a visualization scheme and the design of a prototype implementation of the scheme, PixelAssemblySIMD, detailing the attempts made in other directions before reaching the final method, and analyzing the shortcomings of these abandoned methods, thus highlighting the advantages of the current visualization method. It then describes the design of the current visualization scheme and prototype tool PixelAssemblySIMD in detail: including the design of a CPU emulator introduced to ensure data accuracy, the design of an animation executor capable of executing asynchronous behavior animations in synchronous systems, and the design of the instruction executor in PixelAssemblySIMD.
- Chapter 4*Implementation of the Prototype of the Visualization Method*. This section describes the implementation of the visualization method mentioned in Chapter 3 with Rust, mainly including the CPU emulation library *cpulib* independent of the visualization tool, the anima-

tion executor used for visual representation, and the instruction executor used when executing instructions.

- Chapter 5*Experiment and Evaluation*. This chapter first introduces the difficulty of evaluating the visualization scheme and proposes a method to evaluate the visualization scheme in this thesis. Then it introduces the results of the evaluation using this method and analyzes the evaluation results. Finally, it provides some SIMD algorithm examples based on assembly language, which can be used to detect the effectiveness of the implemented functionalities in Chapter 4.
- Chapter 6*Related Research*. This section introduces other research works related to this thesis, and compares these methods with this method in detail, highlighting the innovative points and advantages of the visualization method mentioned in this thesis.
- Chapter 7*Summary and Prospect*. This section summarizes the research results of this thesis and looks forward to the future research direction.

2

Research Background

This chapter mainly states the background knowledge of the research. This chapter will introduce other related background knowledge involved in this thesis besides assembly language, SIMD, and visualization, including CPU simulators, the Rust programming language, and GUI design. The related background knowledge of assembly language, SIMD, and visualization has been introduced in Chapter 1.

2.1 CPU Emulator

2.1.1 An Overview of CPU Emulator

A CPU emulator is a software tool that can emulate the operations of a physical CPU, allowing code to be executed without the need for a real CPU. This enables it to run and test programs designed for specific CPUs in different hardware or software environments. Generally, CPU emulators can be used for various purposes, such as analyzing malicious software code, verifying code semantics, and emulating cross-architecture console software[14]. For example:

- *Software development and testing*: Developers can test software on CPUs of different architectures without actually owning the hardware.
- *Hardware design verification*: Hardware engineers can emulate a newly designed CPU to verify its performance and functionality.
- *Education and learning*: Students and beginners can learn about the workings of CPUs of different architectures through emulators.
- *System debugging*: Debugging operating systems and other low-level software in a emulated environment is convenient for identifying and fixing errors.
- *Reverse engineering*: Security researchers and hackers may use emulators to analyze malicious software or conduct vulnerability research.

The most basic CPU emulators usually have the function of *instruction set emulation*, which can emulate the instruction set of a specific CPU and allow programs designed for that CPU to run. More advanced CPU emulators may also support *dynamic translation* and *system emulation* functions. The former can combine instruction set emulation with JIT compilation, dynamically translate the instructions of the target CPU into the instructions of the host machine's CPU, and improve the efficiency of the emulation. The latter can emulate the whole computer system, including other hardware components such as memory, I/O devices, etc.

So, according to the functions that the CPU emulator can achieve, we can divide the emulator into two basic types: *instruction set emulators* and *full system emulators*. The instruction set emulator can only emulate the instruction set of a specific CPU and is used for testing and analysis of specific applications; whereas the full system emulator can emulate the entire computer system, including CPU, memory, I/O, etc., and is generally used for more advanced operating system debugging or implementing non-kernel level virtualization environments.

2.1.2 Comparative Analysis of Common CPU Emulators

Table 2.1is a list of some commonly used CPU simulators.

Below, I will introduce QEMU and Unicorn, two of the most commonly used CPU emulators, within the context of full-system emulators and instruction set emulators.

Table 2.1: Common CPU Emulators

Name	Type	Architecture	Open Source
QEMU	Full System	ARM, MIPS, PowerPC, RISC-V, x86	Yes
Bochs	Full System	x86	Yes
Simics	Full System	ARM, MIPS, PowerPC, RISC-V, x86	No
Intel SDE	Full System	x86	Yes
Unicorn	Instruction	ARM, MIPS, PowerPC, x86	Yes

QEMU (Quick EMUlator) is a multifunctional open source software, mainly used for creating virtual machines and emulation environments. It is designed to emulate different hardware platforms, and facilitate the development and testing of complex systems. QEMU is capable of emulating different hardware architectures. In hardware/software co-development, QEMU's emulation ability allows testing and verification of hardware concepts and kernel drivers and applications before the actual hardware is available[15]. It allows running multiple operating systems on a single physical machine, making it

a valuable tool for cloud service providers and large organizations seeking to optimize their server infrastructure[16]. Currently, QEMU is being used by various user groups including developers, system administrators, researchers, and students. In academia and research fields, QEMU is often used as an educational tool to teach computer architecture, operating system, and virtualization concepts[16].

Unicorn is a standalone CPU emulator framework which supports multiple architectures, providing a clean, simple, lightweight and intuitive cross-architecture API. Unicorn is built based on QEMU, thus it supports QEMU's high performance and JIT technology. But it has modified QEMU code to fix issues such as memory leaks, while only retaining CPU emulation code. In this way, Unicorn not only builds a more lightweight and compact CPU emulation framework while retaining QEMU's advantages, but also improves the maintainability and security of Unicorn[14].

2.1.3 The Common Registers Supported by the x86_64 Architecture

The GPRs[*1] supported by the x86_64 architecture vary from 8-bit to 64-bit width, as shown in Table 2.2.

Table 2.2: GPRs Supported by The x86_64 Architecture

Width	Registers
8	AH, BH, CH, DH, AL, BL, CL, DL, SIL, DIL, BPL, SPL, R8B, R9B, R10B, R11B, R12B, R13B, R14B, R15B
16	AX, BX, CX, DX, SI, DI, BP, SP, R8W, R9W, R10W, R11W, R12W, R13W, R14W, R15W
32	EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, R8D, R9D, R10D, R11D, R12D, R13D, R14D, R15D
64	RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14, R15

End of table 2.2 GPRs Supported by The x86_64 Architecture

The vector registers supported by the x86_64 architecture that supports AVX-512 include XMM

[*1] GPR means General Purpose Registers

registers, YMM registers, and ZMM registers, which are used to store vector data of 128 bits, 256 bits, and 512 bits, respectively. Each type of vector register has 32 indices from 0 to 31, i.e., XMM0-XMM31, YMM0-YMM31, and ZMM0-ZMM31.

2.2 Rust Programming Language

Rust is a multi-paradigm system programming language developed with support from Mozilla, intending to provide superior memory safety, concurrency, and performance. Since the first pre-release version of the Rust compiler was released in January 2012, it has quickly gained widespread attention from the developer community. It has also often been voted the "most loved" programming language in Stack Overflow's developer surveys. To a certain extent, Rust deserves more attention and learning from programmers because it outperforms other mature languages in performance, security, and confidentiality[17]. The Rust language has the following main advantages:

- *Memory Safety*: Rust, based on a strong type system of ownership and borrowing, resolved the long-standing contradiction in programming language design between *control* and *safety*. It also provides the resource management control of low-level system programming languages and the safety of high-level languages[18].
- *Performance*: In comparisons among C, C++, Go, Java, Python and Rust, Rust surpasses Go, Java and Python in terms of time and space performance. Compared with C and C++, Rust is slightly behind in time comparison, but excels in memory usage[19][17].
- *Zero-cost Abstractions*: Rust offers high-level abstractions without sacrificing performance[17].
- *Concurrency*: Rust's design supports concurrent programming without data races, making it simpler to write efficient, multithreaded programs. Many data race issues related to concurrency can be avoided through Rust's type system design[20].
- *Modern Language Features*: Rust supports modern language features such as pattern matching, closures, which makes programming more efficient and enjoyable.

However, Rust also has the following disadvantages:

- *Unavoidable Concurrent Vulnerabilities*: Programmers generally do not understand Rust's rules for concurrent checks well, and excessive reliance on the compiler's checks may still lead to the occurrence of concurrent vulnerabilities[21].
- *Unproven Security Claims*: Rust uses an ownership-based type system to ensure safety and expands the expressiveness of the system by internally using libraries with unsafe features[22]. This

can potentially pose a security risk for libraries that use unsafe declarations that do not comply with certain features.

- *Learning Curve*: The ownership and borrowing system of Rust might be difficult for new users to understand, especially those who are not familiar with low-level memory management concepts; the syntax of Rust might also be more complex than some other programming languages[23].
- *Fewer Libraries and Frameworks*: Rust is still in a stage of rapid development, with a smaller community, and may lack some libraries and frameworks[23].

Despite the above-mentioned shortcomings, Rust is still a programming language worth learning. Rust's memory safety and concurrency make it a language very suitable for system programming, and Rust's performance is also outstanding, which can be used to write high-performance programs. Rust's syntax is also very modern, supporting many modern language features, making programming more efficient and enjoyable. Rust's learning curve may be steep, but once the ownership and borrowing system of Rust is mastered, it's possible to write safer and more efficient programs.

2.3 GUI

2.3.1 GUI Design

Declarative UI programming and imperative UI programming are two different methods for designing user interfaces in software development. As the pace of software updates and iterations speeds up, traditional imperative UI programming is gradually being replaced by declarative UI programming, which is widely used in mobile and web applications. At present, the mainstream declarative UI programming frameworks for web applications include React, Vue, Flutter, etc., and the new UI frameworks for mobile, like iOS's Swift UI and Android's Jetpack Compose, are declarative UI programming frameworks.

Declarative UI Programming

Declarative programming involves writing code to specify what you want, not explicitly defining the control flow. This style doesn't describe the steps to achieve a certain state, but describes the expected state itself[24]. For example, the simplest example is using HTML markup language to represent the expected view tree. The programmer writes code that directly represents the expected results, and automatic algorithms are responsible for converting the specifications into runtime structures or behav-

iors[24].

Using declarative UI programming has many advantages and is becoming more and more popular in UI programming:

- The code of declarative UI is usually more compact because the description of UI is usually at a higher level of abstraction.
- Declarative UI focuses on the goals, not the process of achieving the goals.
- Declarative UI promotes the creation of authoring tools, as declarative specifications can be easily loaded and saved by tools[24].

However, there are also some disadvantages of declarative UI programming:

- The declarative UI doesn't directly describe the control flow, which makes debugging more challenging.
- Specialized debugging tools are required for debugging, such as Vue.js framework usually uses Vue Devtools for debugging. Because the tools must comply with declarative specifications.
- Small changes in declarative code can lead to significant changes in output, which might bring some difficulty to code maintenance and iteration[24].

Imperative UI Programming

Imperative programming requires developers to explicitly define the steps to reach the desired state or perform operations. In UI creation, this involves writing detailed code instructions to tell the device how to display the components[25][26]. Imperative UI programming is more focused on describing which aspects of the state need to be reached, so it generally involves more specific and detailed step-by-step instructions.

Imperative UI programming also has some advantages, such as it can provide more control over the execution process, and due to the explicitness of the code, code written with imperative UI programming is generally easier to understand and debug. However, since imperative UI programming requires more code to describe the control flow when the UI changes, its code is usually more verbose, and programmers need to put more effort into managing the state and flow of the application.

Comparison and Analysis

Overall, declarative UI programming and imperative UI programming have obvious differences in the following aspects:

1. *Approach*: Declarative programming is more about stating *what* you want, while imperative programming is about specifying *how* to do something[26].
2. *Level of abstraction*: Declarative programming operates at a higher level of abstraction and usually requires imperative implementation on lower-level to convert declarative commands into executable instructions[26].
3. *State management*: In declarative programming, state changes are abstracted, making the code more readable and reducing the focus on the state management mechanism. By contrast, imperative programming often involves explicit state operations[26].
4. *Context dependency*: Declarative code can be more independent of context, which means that the same code can be used in different programs without modification. Imperative code usually depends on the context of the current state, making it less flexible in different environments[26].

In summary, declarative UI programming focuses on the final result, with a higher level of abstraction and less explicit control flow; whereas imperative UI programming spells out the exact steps to achieve the desired state, providing more control, but may result in code that is more complex and verbose. With the demand for faster software iteration speed now, declarative UI programming has gained widespread application on the Web and mobile, and imperative UI programming has been gradually replaced by declarative UI programming.

2.3.2 Exploring the egui Library in Rust

When implementing the project of this thesis, the GUI is based on the egui library. Egui is an Immediate Mode GUI(IMGUI) library written in Rust, aiming to provide a fast and easy-to-use experience for building graphical user interfaces. In IMGUI, each part of the user interface is recalculated and rendered in each frame. UI elements like buttons and sliders are dynamically created and rendered each time the application draws its interface. This is contrary to traditional Retained Mode, where UI elements are created once and retained until explicitly modified.
egui has the following main advantages:

- *Good performance*: Egui differs from traditional GUI toolkits in that it is not dependent on the event-driven model and adopts the immediate mode. This design is based on performance considerations, as the immediate mode usually uses less memory and has faster rendering times[27].
- *Multi-platform support*: Egui provides multi-platform support and can be compiled into web (WASM) and native binaries under multiple architectures, and it can even be embedded into

game engines.

- *Automatic layout*: Egui supports automatic layout, making it easier to create cross-platform responsive designs.
- *Simple and Efficient API Design*: egui's API design is simple and easy to understand and it provides an abundance of controls such as buttons, text boxes, and sliders, etc. A full-featured GUI program can be quickly implemented through natively provided APIs[27].
- *High-level Features*: egui also provides high-level features such as layout management[27].
- *Excellent Documentation and Active Community*: egui's official website provides detailed tutorials and examples and many third-party libraries and tools are available for development with egui[27].

Safety, responsiveness, friendliness, and portability are the main goals of egui, which claims to be one of the simplest libraries amongst Rust web applications[28].
However, due to the characteristics of the immediate mode GUI, egui does not support native animation and requires developers to implement animation logic themselves, such as by changing the state or properties of components in successive frames to create animated effects. Since egui recalculates the interface state in each frame, this makes the implementation of dynamic and responsive animations feasible, though it may require more programming effort to manually handle animation logic. However, current animation libraries are usually event-driven because we typically want to execute certain operations at the end of an animation rather than executing animations in every frame. Thus, due to this characteristic of egui, implementing asynchronous behavior animations in such a synchronous environment can be challenging. Therefore, to achieve dynamic visual effects, we must expand upon the foundation of egui and write a similar event-driven animator to display asynchronous action animations.

3

Design of Visualization Method and PixelAssemblySIMD

This chapter proposes a visualization method for representing SIMD calculations in assembly language and the design of a tool that implements this method named PixelAssemblySIMD. The first section of this chapter introduces the evolution of this visualization method, elaborates on the limitations of the first version of this visualization method and the deficiencies of the first prototype of PixelAssemblySIMD, leading to subsequent improvements. Starting from the second section of this chapter, it introduces the improved visualization method and its implementation. Firstly, the second section provides a detailed introduction to the design thought of the improved visualization method, including GUI design and animation design; The third section introduces cpulib, a CPU simulation tool designed by PixelAssemblySIMD to ensure data reliability, to address the serious defect mentioned in the first section; The fourth section introduces the design of an animation executor that can be used in egui framework, which supports animations with asynchronous behaviors under a synchronous environment; Lastly, this chapter presents the design of the instruction executor in the tool.

In this chapter, we use *element* to represent a scalar in a vector register.

There will be a lot of use of charts for illustration in this chapter and the legend used in these charts are as shown in Figure 3.1.

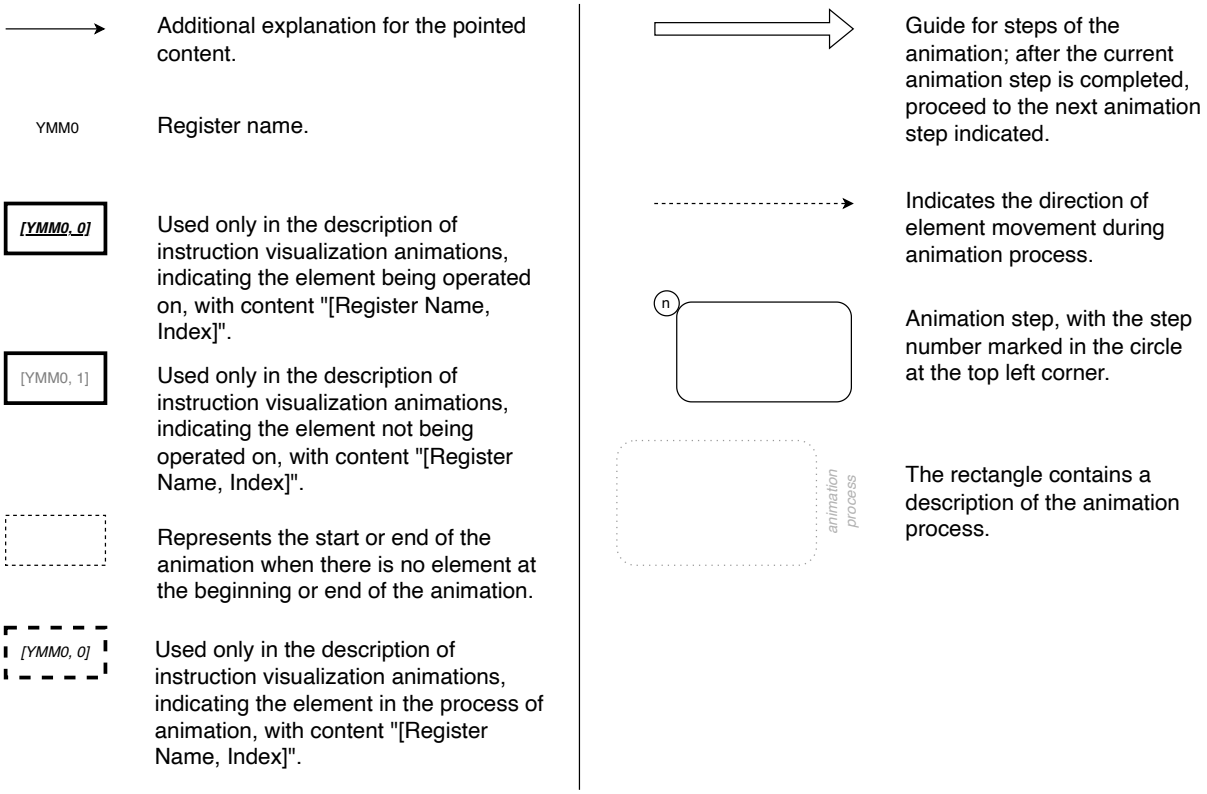


Figure 3.1: Legend used in the visualization description

3.1 Evolution of Design

3.1.1 Design and Limitation of the Visualization Method in the Initial Prototype

This section introduces the design of the visualization method in the initial prototype (hereinafter referred to as "Method1") and its limitations. The design of Method1 only considered the visualization of registers because most SIMD instructions interact with registers. The following first introduces the visual representation of the registers in Method1.

Visual Representation of Registers

In Method1, registers are divided into two types: GPRs and vector registers. Because vector registers usually save a set of scalars, while a single general-purpose register generally saves a scalar, we make the vector registers look like a collection of general-purpose registers in the graphic design of register visualization. The illustration shown in Figure 3.2 is a visual representation of the elements in the general-purpose register or vector register.

This representation uses rectangles to represent the elements of general-purpose registers or vector reg-

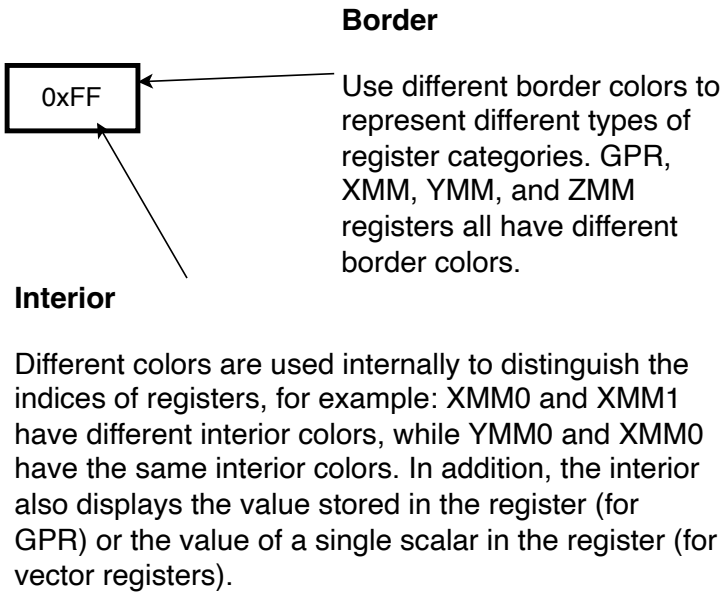


Figure 3.2: Visual representation of GPR or element in vector registers

isters. However, the rectangle can be divided into three parts: the outer frame, the internal fill color, and the internal value.

- The color of the outer border represents different types of registers. Thus, different border colors can distinguish 8-bit general-purpose registers, 16-bit general-purpose registers, 32-bit general-purpose registers, 64-bit general-purpose registers, XMM vector registers, YMM vector registers, and ZMM vector registers in the x86_64 architecture.
- The internal fill color represents the registers of different indices in each type of register. In the x86_64 architecture, except for the 8-bit general registers, each type of general register has 16 indices, so we use 16 different fill colors to distinguish registers of different indices; For 8-bit general registers, since the high and low 8 bits of the 16-bit registers AX, BX, CX, DX have been divided into two independent 8-bit registers, and the chance to use these registers separately is very rare, so we use the same color to represent the high and low 8 bits of these registers. For vector registers, each type of vector register has 32 indices, so we use 32 different fill colors to distinguish registers of different indices.
- The internal number represents the value in the GPR or element.

The complete representation of the register is shown inFigure 3.3.

The name of the register is displayed on the far left, followed by the visual representation of the register.

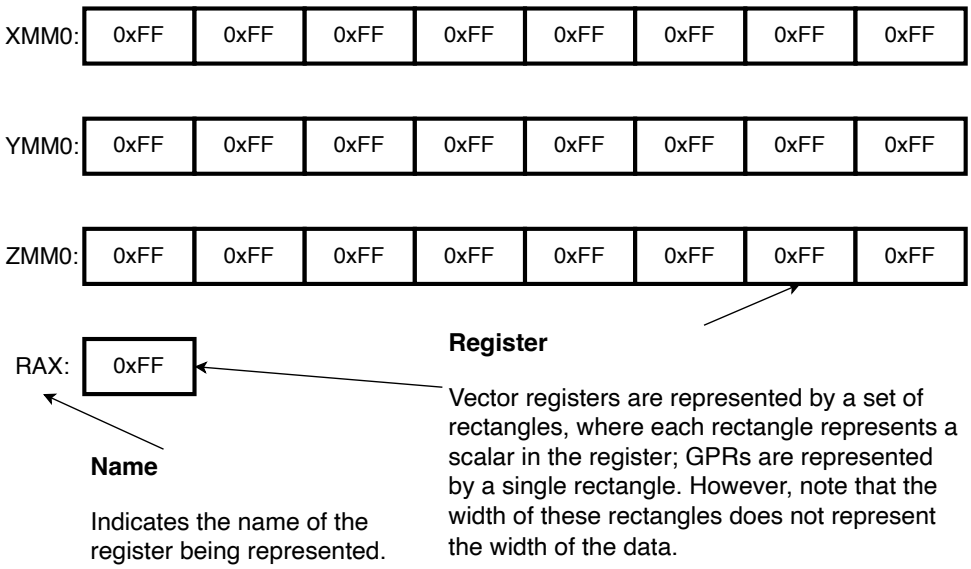


Figure 3.3: Visualization of the registers in Method1

Here, as introduced above, the GPR uses the same representation method as Figure 3.2; while the vector register is represented as a collection of elements, i.e., a group of rectangles in Figure 3.2. According to this visualization scheme, we can easily distinguish different types of registers, as well as registers with different indices, and know the values saved in them.

Visualization of SIMD Instrcutions

In Method1, for easier visualization, common SIMD instructions have been abstracted into three specific operations: *move*, *assignment* and *exchange*. Through the combination of these three types of operations, we can visualize most of the SIMD instructions, and this operation is also suitable for GPRs.

move operation means moving a value from one register to another, or from one element to another. It is worth noting that *move* in assembly language means copy rather than cut.

Figure 3.4 shows the animation design of the move operation. The animation design of the move operation contains 5 steps:

1. Animation starts, determining the element to be moved and the target position;
2. Move the element to be moved to the target position. Note that here the border and internal color of the moved element are not changed;
3. After completing step 2, the moved element will cover the target position, and there is still no need to change the border and internal color of the moved element. But at this time it is necessary to move the real data, i.e., to change the value of the target position to be the same as the

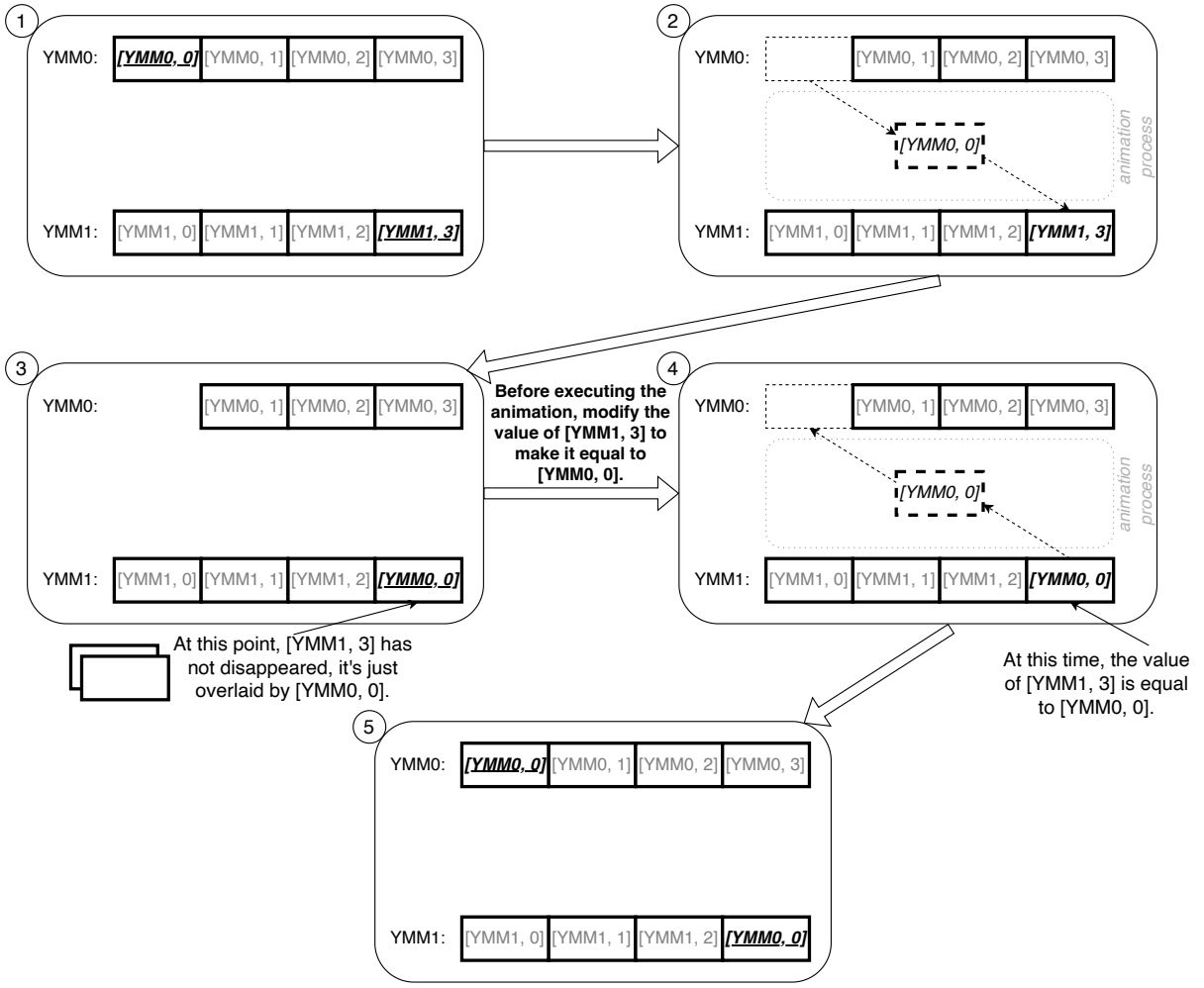


Figure 3.4: Method1 instruction visualization animation: move operation

moved element;

4. Move the moved element back to the original position;
5. Animation ends, the element at the target position has been assigned the same value as the moving element.

The animation of the move operation moves the moved element to the target position and changes the data of the target position after the coverage, so from the user's perspective, it is as if the value of the moved element has been moved to the target position.

assignment operation means assigning a new value to a register or an element.

The animation design of the assignment operation is shown in Figure 3.5. The animation of the assignment operation involves 6 steps in total. Despite the relatively high number of steps, the operations performed are quite simple:

1. The animation begins by determining the target element (i.e., the element to be assigned) and

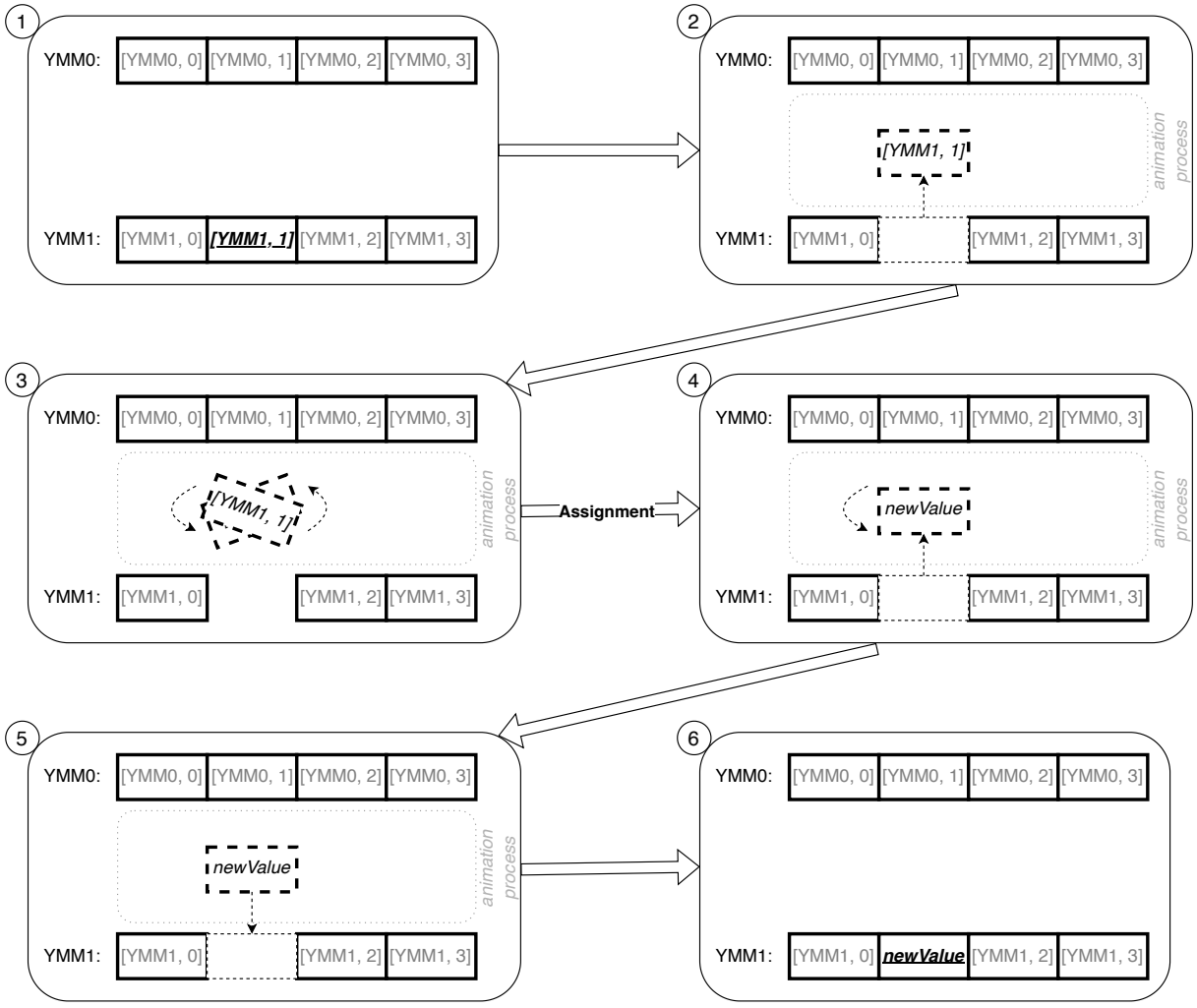


Figure 3.5: Method1 instruction visualization animation: assignment operation

the target value;

2. In order to highlight the assignment operation, the target element is slightly moved upwards;
3. The target element is made to sway left and right to highlight the assignment operation. At this point, the new value is assigned to the target element;
4. The swaying motion is ceased, and the value of the target element has been changed;
5. The target element is moved back to its original position;
6. The animation is complete, and the value of the target element has been changed.

The animation of the assignment operation has only completed the assignment operation on the target element. However, because the representation of elements uses different border colors and internal colors to distinguish different categories, it is difficult to highlight the target element of the assignment operation just by changing the border color, so a more exaggerated animation is used to highlight the target of the assignment operation.

exchange operation means to exchange the values in two registers or two elements.

The animation design of the exchange operation is shown in Figure 3.6. The animation of the exchange

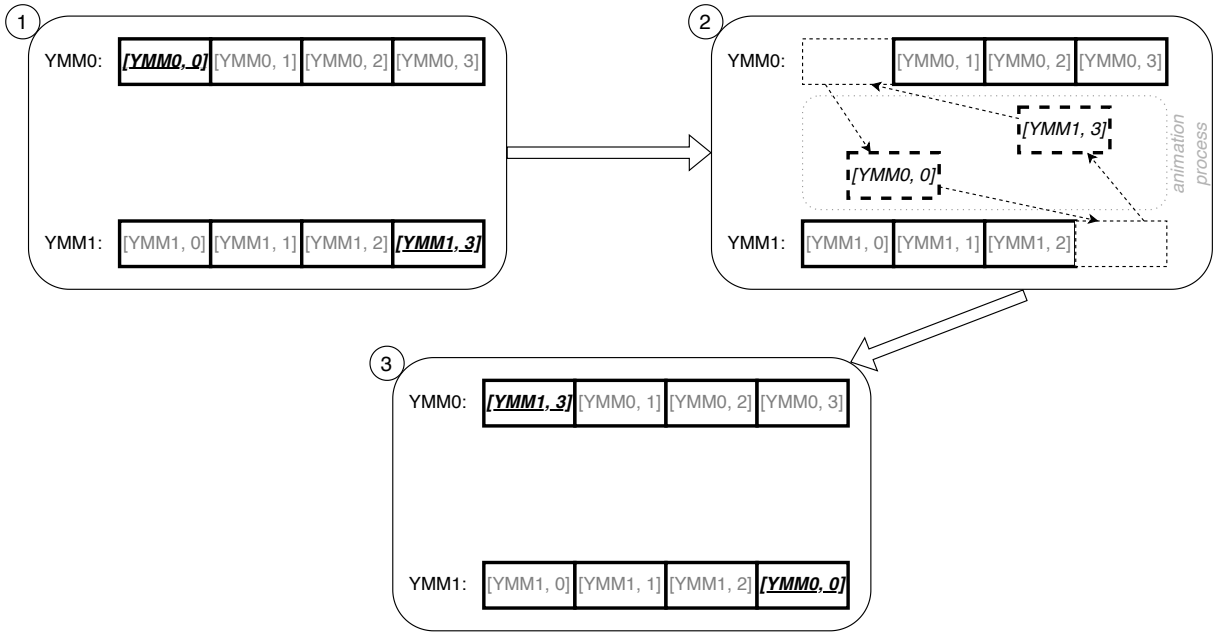


Figure 3.6: Method1 instruction visualization animation: exchange operation

operation only includes 3 steps, which is very simple:

1. The animation begins by determining the two elements to be exchanged;
2. The elements to be exchanged are moved to each other's positions, and their borders and internal colors are changed to match the type at the target location;
3. The animation is complete, and the positions of the two elements have been successfully swapped.

The exchange operation directly swaps the positions of two elements and changes their colors to match the type of the other, such a simple operation is also intuitive to the user. It should be noted that there is no direct exchange operation in SIMD instructions. We usually achieve the exchange operation by using other more general movement instructions. However, here we visualize the exchange operation as an independent operation, because the existing moving operation is not clear enough in the animation representation when exchanging two elements.

Limitations of Visualization Method

Visualization Method1 abstracts common SIMD instructions into three operations that are more conducive to visualization, and can perform visualization of most common SIMD instructions. How-

ever, the design of this visualization method still has limitations in the following aspects:

- *Inability to represent data width in the visualization of registers*: Method1 does not differentiate the visualization of registers based on data width, which can make it difficult for users to intuitively distinguish different width data stored in registers. For example, the visualization of an `XMM` register that holds four 32-bit floating point numbers and a `YMM` register that holds four 64-bit floating point numbers is the same in Method1 and has exactly the same width. This kind of visualization method can confuse users.
- *Unable to clearly represent the operational process*: In Method1, the instruction involving operation can be represented as "assigning the result after operation to the register or element." However, such an abstraction cannot clearly represent the process of operation, for example: when executing the `vaddpd` instruction, we cannot clearly represent the process of adding two elements, but can only be represented as assigning the result of adding the two elements to a new element. Such a representation can make complex computation instructions difficult to understand, for example, the `vfmadd132pd` instruction involves addition and multiplication, and the current visualization method cannot clearly represent which elements are involved in the operation.
- *visualization method is not universal*: Depending on Method1, the operation abstraction of commonly used SIMD instructions can successfully model most SIMD instructions. However, this visualization animation scheme will operate on the visualization of the original registers, which can cause a lack of clear guidance for executing complex instructions or when some operands of the instructions are the same, and cannot be used as a universal visualization method. For example, for the instruction `vmulpd xmm0, xmm0, xmm0`, its three operands are all the same, but the visualization of the register only has one, so such instructions in the visualization animation of Method1 can cause difficulty for users to understand.
- *visualization method binds data operations*: The visualization animation of move and assignment operations must carry out data operations in the middle to make the whole visualization process understandable to users. But such a design makes the module of visualization display bound with the module of data operation, which can lead to poor scalability of the visualization method.

3.1.2 Design and Shortcomings of the Initial Prototype of PixelAssemblySIMD

The initial prototype of PixelAssemblySIMD uses Svelte, Vite, and Anime.js frontend technologies based on JavaScript to implement the visualization method of Method1. The initial prototype of PixelAssemblySIMD is hereinafter referred to as "Prototype1".

Software Architecture

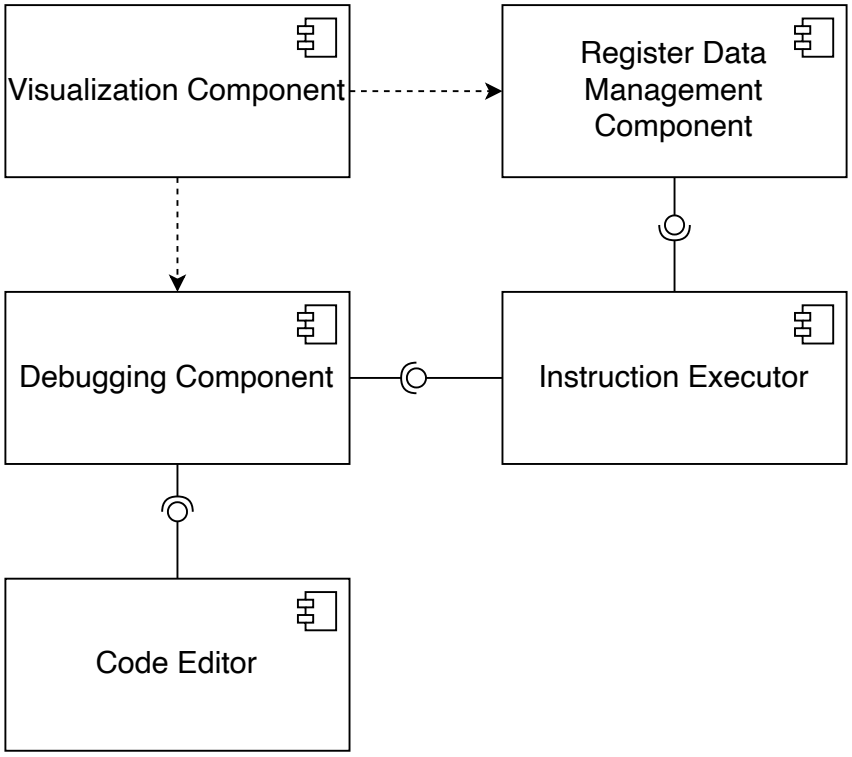


Figure 3.7: Design of Prototype1

The software architecture of Prototype1 is shown in Figure 3.7 with a UML component diagram. From this component diagram, we can see that the visualization component depends on the debugging component and the register data management component; the debugging component uses interfaces from the code editor and instruction executor to update register data to achieve the function of assembly code debugging; the instruction executor uses interfaces from the register data management component when updating register data.

Here, the visualization component does not directly use interfaces from the register data management component, but depends on it. This is because most frontend technologies are based on responsive models when rendering graphics. When the instruction executor changes the register data, the regis-

ter data management component will automatically notify the visualization component to update and re-render the visualization graphics.

GUI Design

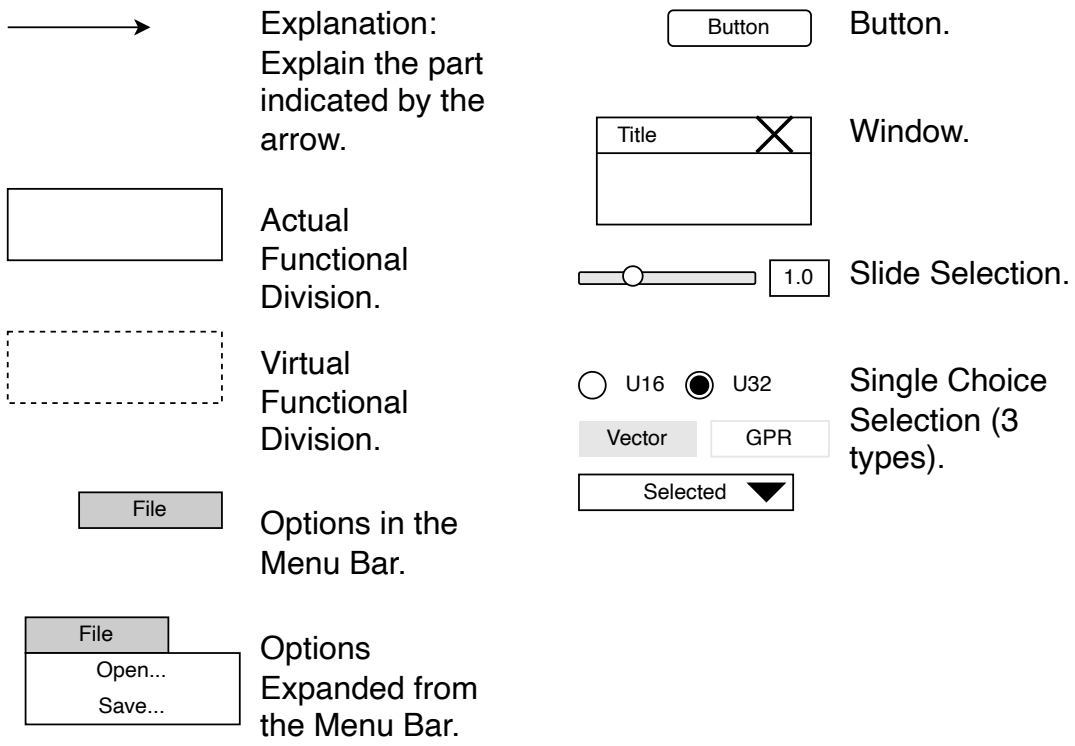


Figure 3.8: Legend explaining graphical primitives for GUI design

I used a set of graphical primitives to show the functionality and nesting relationship of elements on the user interface to display the software's GUI design. The figure for this set of graphical primitives is shown in Figure 3.8.

Figure 3.9 shows the GUI design of Prototype1. The top of the page is the software's title, which is in line with the design thoughts of most web APPs based on frontend production. The lower part of the page is divided into left and right parts. The left half has an embedded code editor for entering assembly code. The right half includes sections for visualization and debugging functions, with the debugging function located at the bottom of the right half page. This GUI design presents the three main functional areas in the simplest way on the web page, arranging them from large to small according to their importance. Users can understand how to use the software without learning when they use it for the first time.

Figure 3.10 uses UML state diagram is used to describe the software's behavior after user operations, which can help better understand the behavior of software based on the reactive model. As shown in the

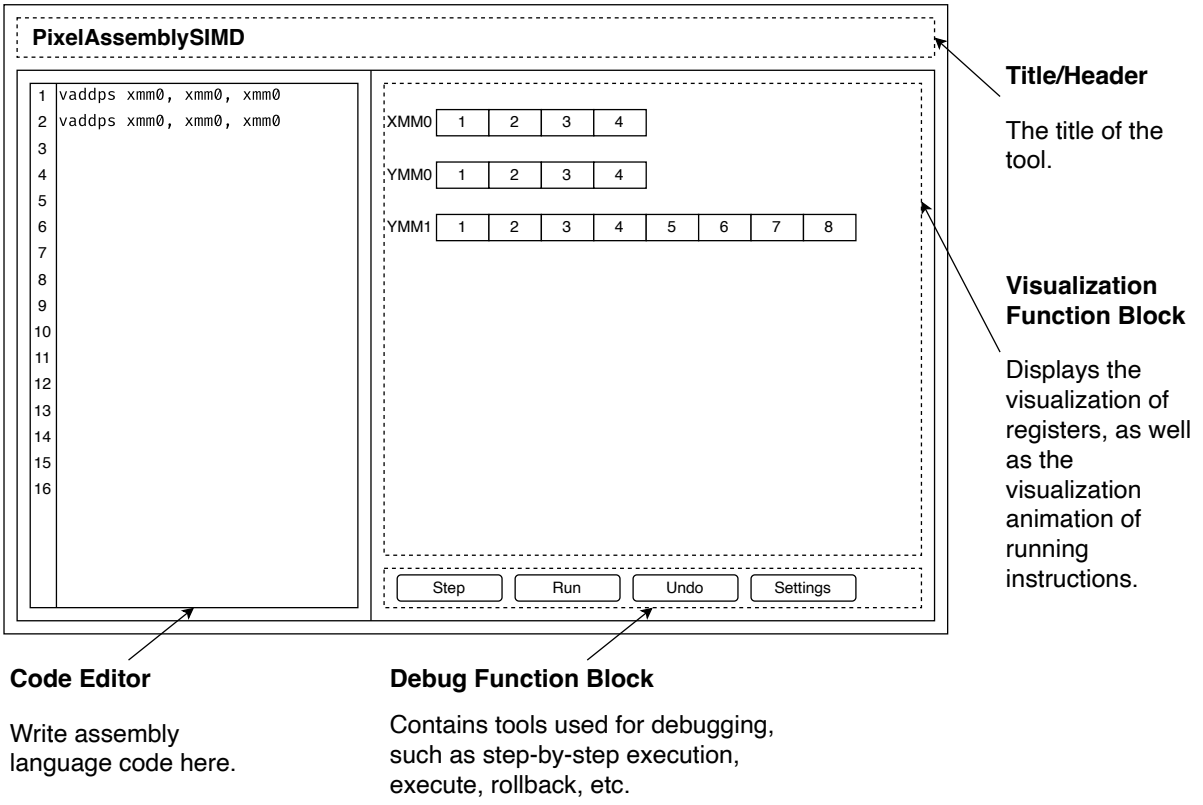


Figure 3.9: GUI design of Prototype1

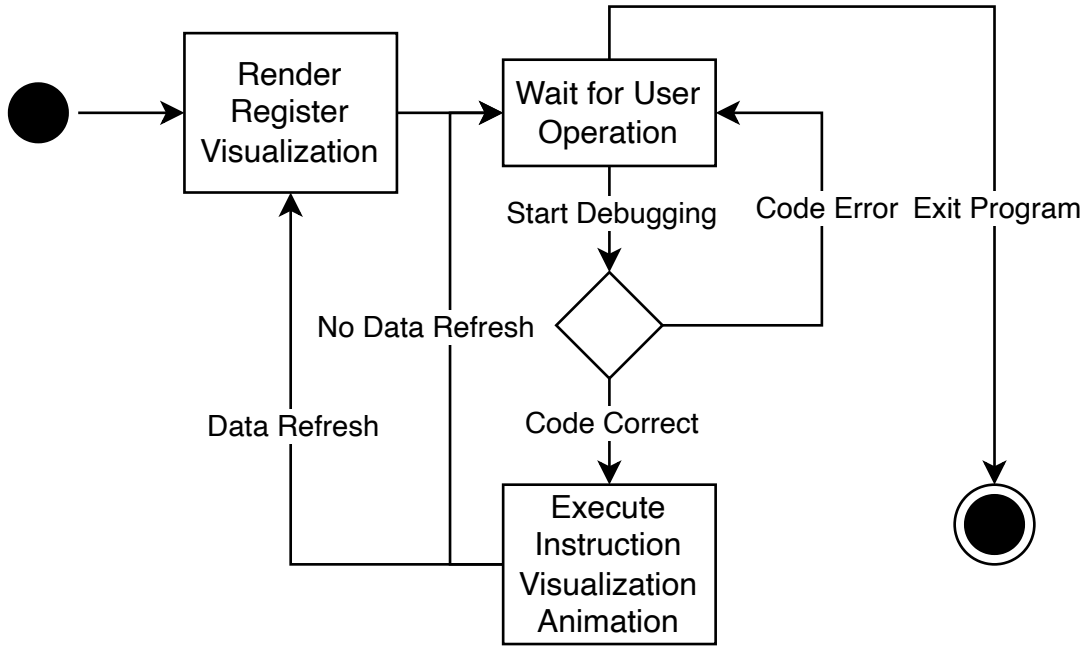


Figure 3.10: State diagram of Prototype1 GUI

figure, after opening the software, it will first automatically render a visual representation of the registers, and then wait for user input. If the user exits the program, it enters a termination state; otherwise, when the user starts debugging, the software will enter different states based on the correctness of the debugging assembly code: if the code is correct, the software will execute the visual representation of the code, and then optionally re-render the visual representation of the register based on whether the instruction has updated the register data; if the code is wrong, it will return to the waiting state.

Register Data Model and Visualization Components

JavaScript is not a traditional Object-Oriented Programming (OOP) language, but UML class diagrams are used here to explain the design of software components, data structures, and relationships, mainly for the following reasons:

- Wide readability and standardization: UML class diagram as a standard modeling language is widely understood and accepted. Even in non-pure OOP environments, many developers and analysts are familiar with class diagrams and can quickly comprehend the structures and relationships they represent.
- Abstraction and conceptualization: UML class diagrams offer an abstract way of describing entities in a system and their interrelationships without relying solely on specific programming paradigms. This abstraction is useful even in non-OOP languages.
- Data modeling: Class diagrams are not just for describing OOP class structures; they can also represent data models and relationships between data, which is essential for any programming language.

Figure 3.11 uses UML class diagram describes the data structure and its relationship with the register data model and visualization components in Prototype1. In the class diagram, **Main** is the main program with an aggregation relationship with **Visualization** (visualization component) and **Instructions** (instruction execution component). That is, **Main** has **Visualization** and **Instructions**, but does not affect their lifetimes. **Registers** is a collection of registers and forms part of the **Visualization** component (i.e., a composition relationship). In addition, **Instructions** also depends on **Registers** and **Visualization** because the instruction execution stage needs to read existing values from the registers for crucial information and GUI operations are defined in the interface of **Visualization**. Finally, **Registers** is a combination of **Register**, which is a data structure representing a single register.

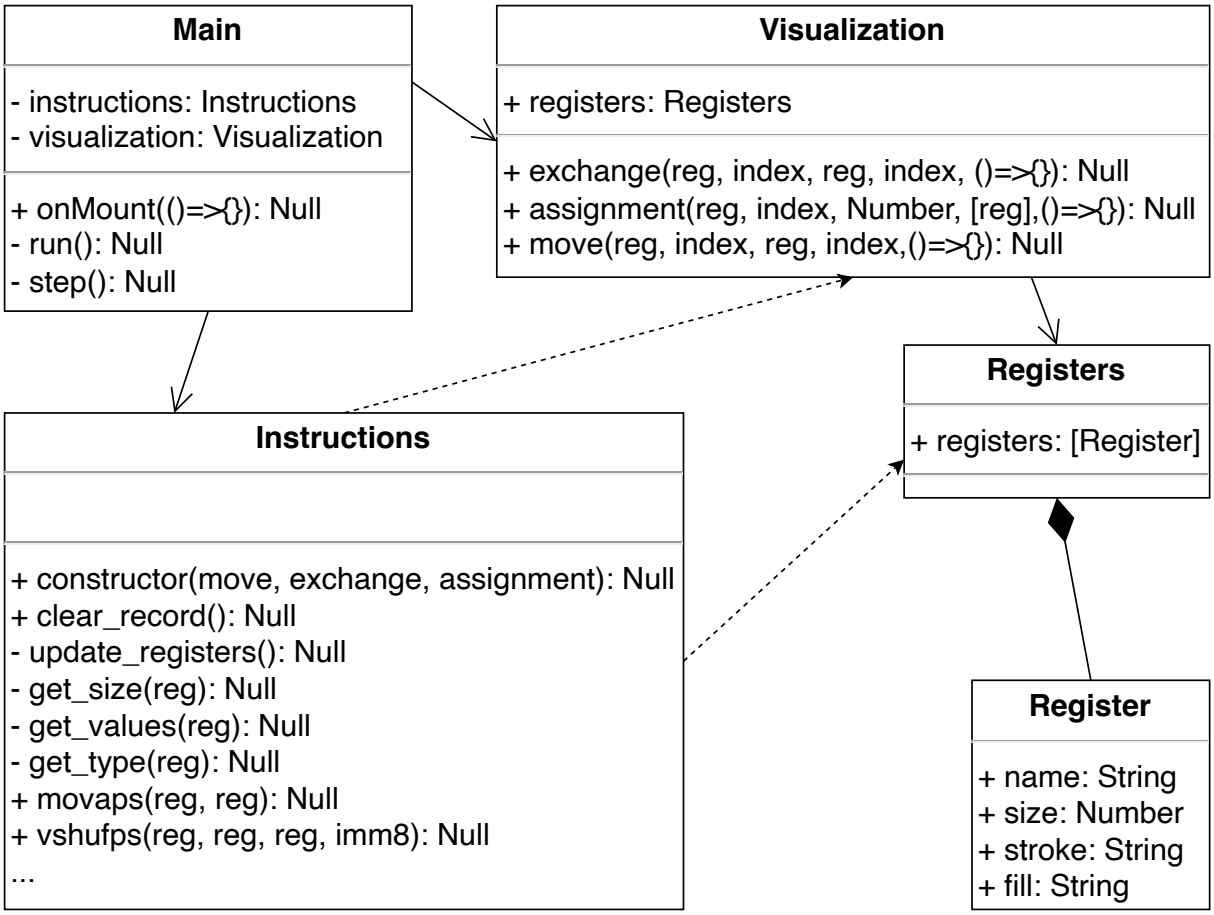


Figure 3.11: Component Design of Prototype1

Deficiencies

Through analyze Figure 3.11, an obvious problem can be detected: the design of the software architecture has tied the visualization component and the register data management component together. Also, the instruction execution component relies on the visualization component. This design has resulted in poor scalability of the visualization solution, as improvements to the visualization solution must simultaneously change the main program, register data management component, and instruction execution component. This is not a well-designed software system.

Moreover, another issue implicitly impacts the reliability of the software. JavaScript's way of handling types and data makes it unsuitable for handling low-level data. However, when modeling the register files, it's essential to handle the low-level data to recreate what the computer does when it interacts with registers. We might use a lot of bitwise operations or even operate directly on bits. Therefore, while the current version of the software does not have data accuracy issues when executing simple SIMD instructions, it cannot accurately simulate the real interactions with the registers when handling more complex SIMD instructions, especially those with different bit widths for input data and operation

data, which could easily lead to data errors.

3.1.3 Improvement Plan

The preceding section has analyzed the problems with the visualization method and software implementation of the first prototype. We proposed an improvement plan for these problems. For the visualization method, we will explore a more generic animation effect to represent changes on the register files by assembly language instructions and should more clearly represent the calculation process of instruction execution. As for the software implementation, the design idea of the improvement plan is to decouple the visualization method from the register data management component. This way, the visualization method can be improved independently of the register data management component. To solve the data reliability problem, we introduced a CPU emulator that can emulate the CPU's execution process, thus ensuring data reliability.

3.2 Visualization Approach for SIMD Instruction

3.2.1 Animation Design

The improved visualization method (hereinafter referred to as "Method2") uses a generic animation effect to represent changes on the register files by assembly language instructions. It can be used to represent instructions with any number of operands, and it won't confuse users even when executing instructions with a single operand. Also, this animation can more clearly illustrate the computation process of instruction execution.

Visualization Representation of Registers

The register representation in Method2 is roughly the same as that in Solution One, but the register representation in Method2 can be distinguished according to the data width, Figure 3.12 shows different data widths. As shown in Figure 3.13, the register representation in Method2 uses different rectangle widths to distinguish different data widths, such a design can allow users to intuitively distinguish registers of different data widths. In addition to this, Method2 removed the design of distinguishing different types and indexes of registers according to color. Using too many colors on the same interface can confuse users, and such a design is not necessary because users can already distinguish different registers through the names of the registers.

Among the different registers shown in Figure 3.13, GPRs are easier to understand because they have

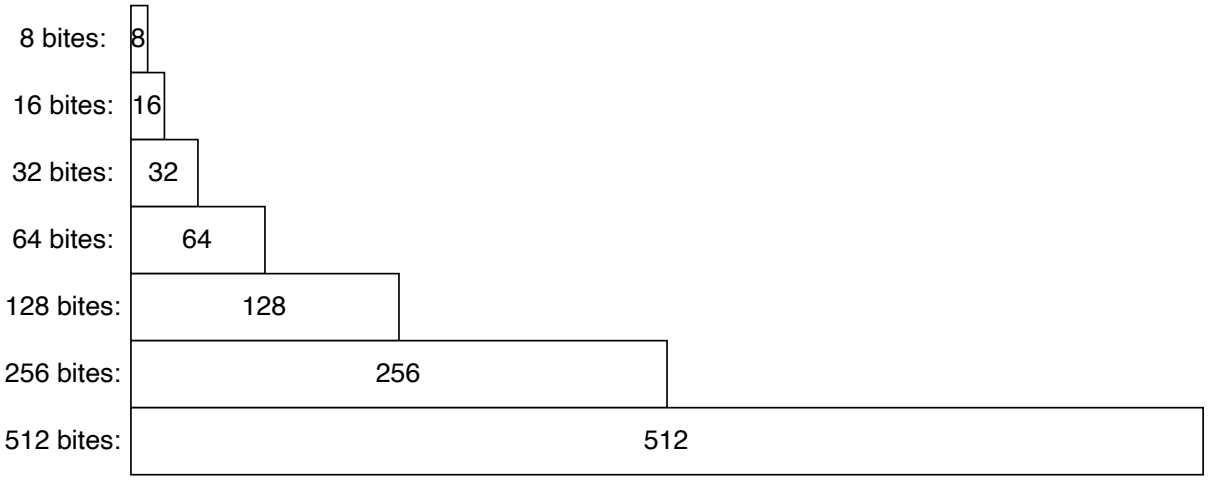


Figure 3.12: Different data widths in Method2

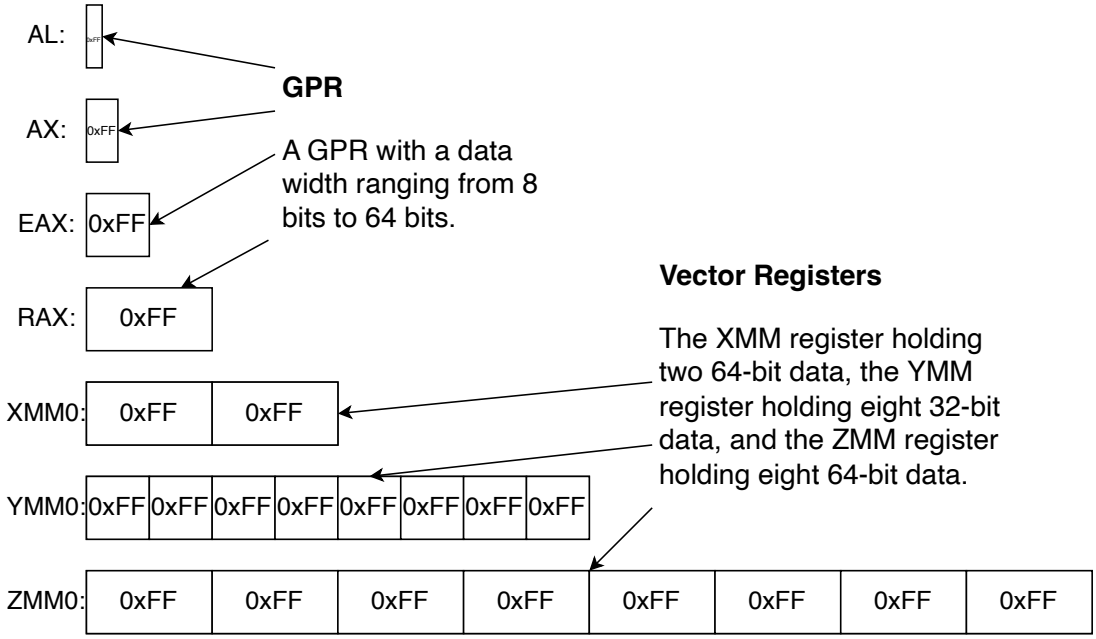


Figure 3.13: Visual representation of registers in Method2

a fixed data width. For vector registers, for example, the displayed `XMM0` register, it holds two 64-bit data. In the graphical representation, the width of these two 64-bit data is the same as the width of the 64-bit GPR (such as `RAX`), which allows the user to intuitively understand the data width stored in the current register without having to calculate based on the total register width and the number of data stored.

Visualization of SIMD instructions

When representing SIMD instructions, Method2 divides the animation into three stages: *Start*, *Execute*, and *End*. The following is a detailed description of the animation design of these three stages

based on the assembly code `vperm2f128 ymm1, ymm0, ymm0, 0×21`.

Figure 3.14 details the *Start* stage of the animation in Method2. The main task of this stage is to create

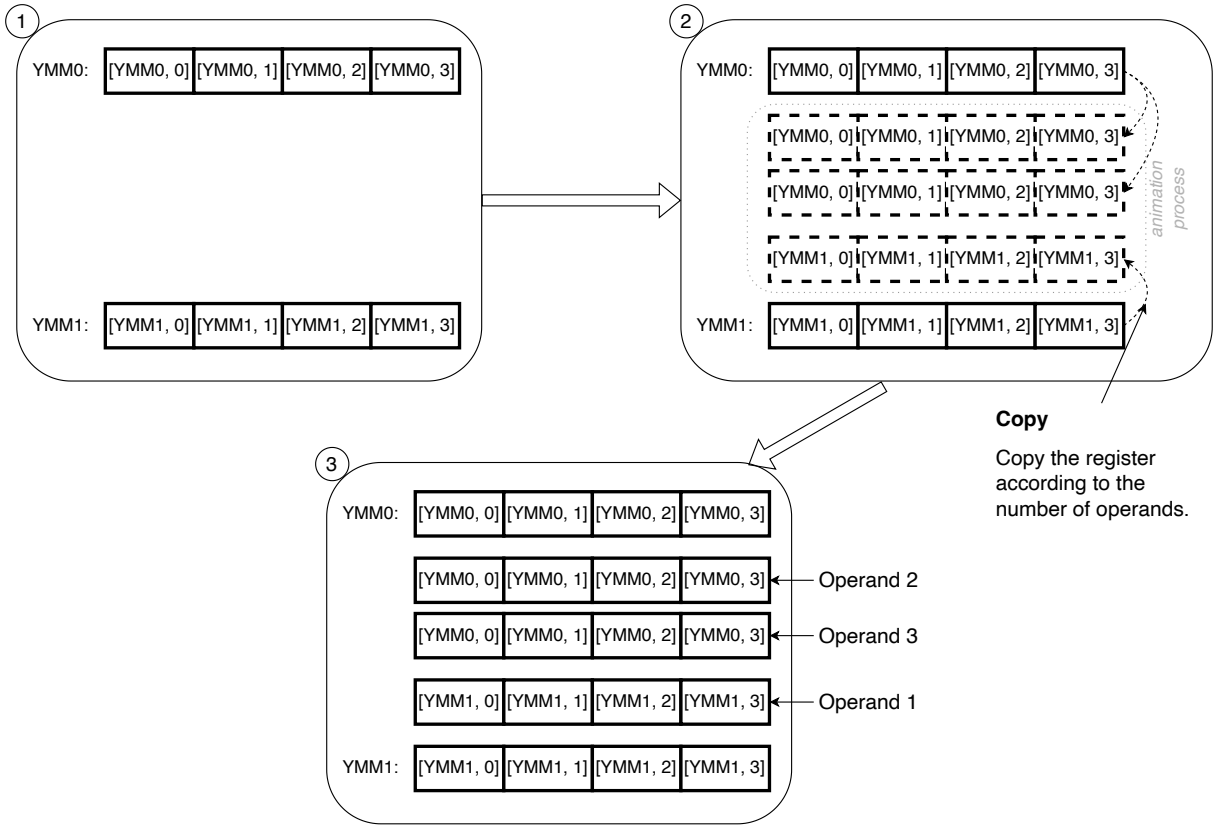


Figure 3.14: Animation design of instruction visualization in Method2 - Start stage

duplicates of the registers in the animation area according to the number of operands: if there are two YMM0 registers in the operands, duplicate the YMM0 register in the animation area twice, representing the two operands. In the *Execute* phase, all instruction execution animations will be carried out in the duplicate of the register, and the original register value will not be modified yet.

Figure 3.15 details the *Execute* stage of the animation in Method2. The main task of this stage is to operate on the duplicate of the register according to the instruction. The advantage of doing this is that most of the common SIMD instructions can be represented using a unique shift animation. In the example of Figure 3.15, the second step executes the operation of the `vperm2f128` instruction with the control bit is `0×21`, permuting the elements of the second and third operands into the corresponding positions of the first operand.

Figure 3.16 details the *End* stage of the animation in Method2. The main task of this stage is to copy the data in the duplicate of the register back to the original register. The advantage of doing this is that in the *Execute* stage, any operation can be performed on the duplicate of the register without worrying about affecting the value of the original register. In the example of Figure 3.16, the second move moves

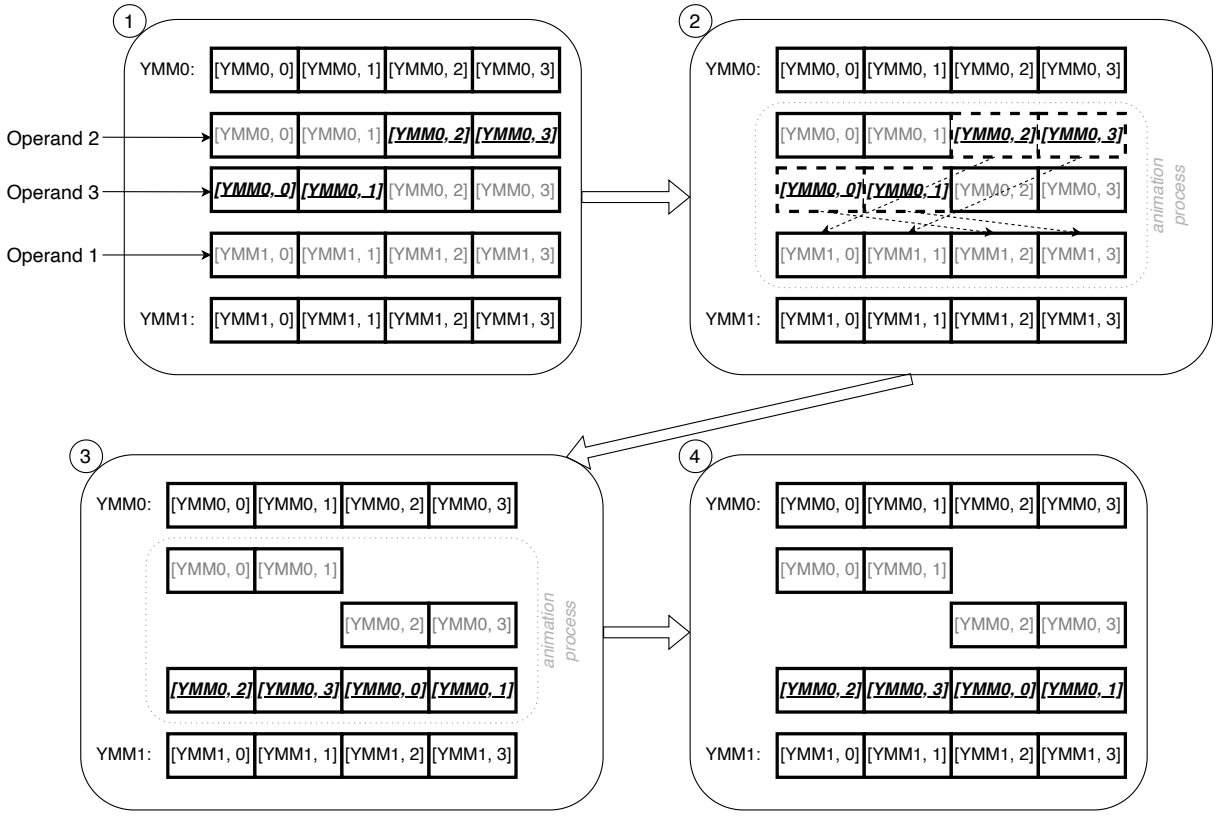


Figure 3.15: Animation design of instruction visualization in Method2 - Execute stage

the new value of the first operand that has been modified to the real register; then in the third step, the duplicate of the register that has not been used is discarded.

Since the animation design in Method2 is relatively complex, the just example did not demonstrate the animation design of instructions containing calculation operations. Next, the animation design of instructions containing calculation operations will be explained based on the assembly code `vaddpd ymm1, ymm0, ymm0`.

What all instructions do in the *Start* and *end* stages is the same. Figure 3.17 is a detailed description of the *Execute* stage of the `vaddpd ymm1, ymm0, ymm0` instruction. In the second step, we first move the second operand to the first operand; then in the third step, we move the third operand to the first operand and perform addition. It is worth noting that the execution of the addition in the third step is not directly concluded, but the calculation formula is given. Afterwards, in the last step, we calculate the result of the addition. Such a design can show the calculation process to the users before giving out the calculation result, allowing users to better understand the instruction execution process.

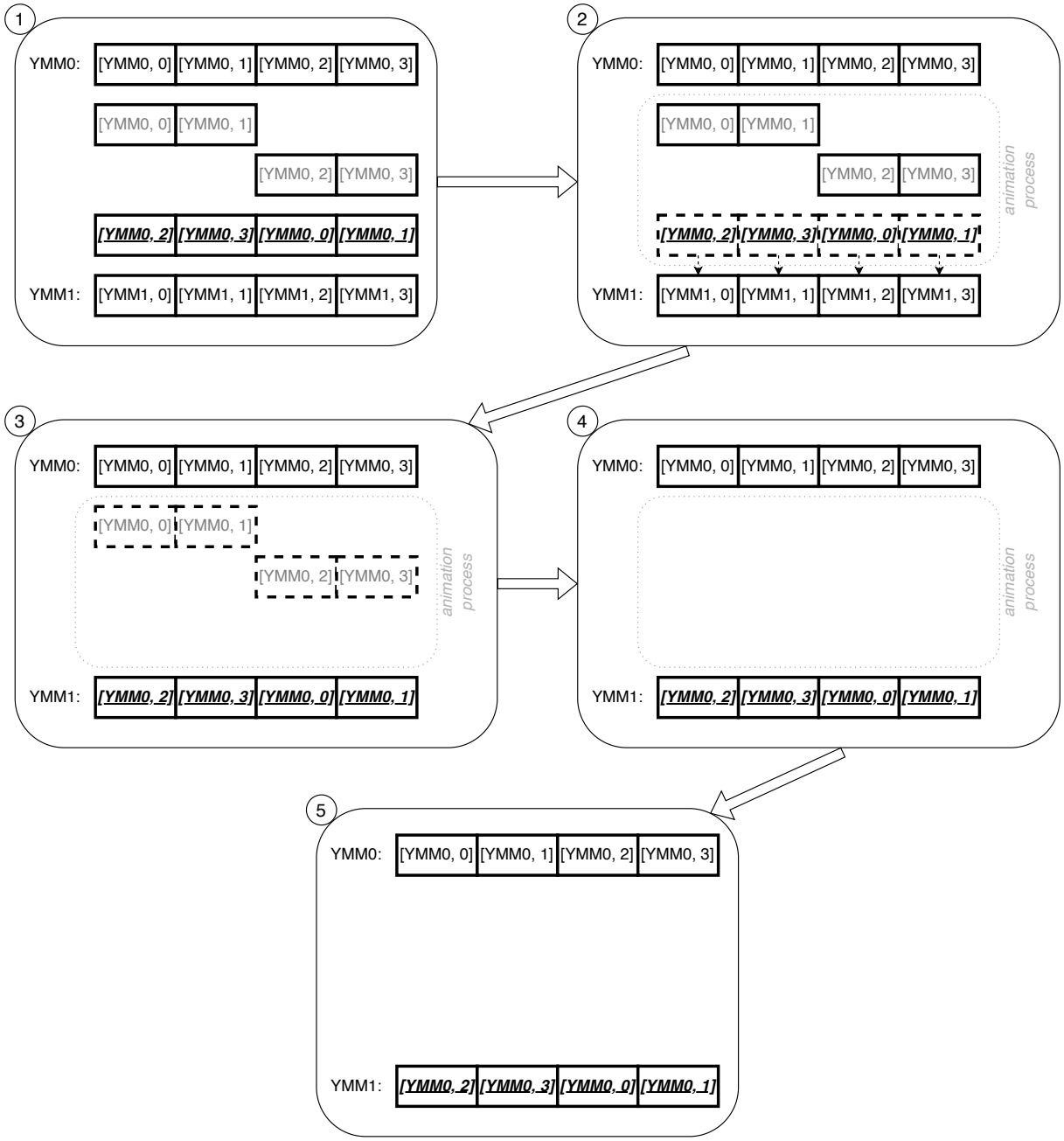


Figure 3.16: Animation design of instruction visualization in Method2 - End stage

3.3 Design of PixelAssemblySIMD

The improved version of PixelAssemblySIMD is written in Rust language, and the GUI part is based on the egui library. The following will refer to this version of PixelAssemblySIMD as "Prototype2".

3.3.1 Reasons for Using Rust and egui

"Prototype2" chose to use the Rust language primarily for the following reasons:

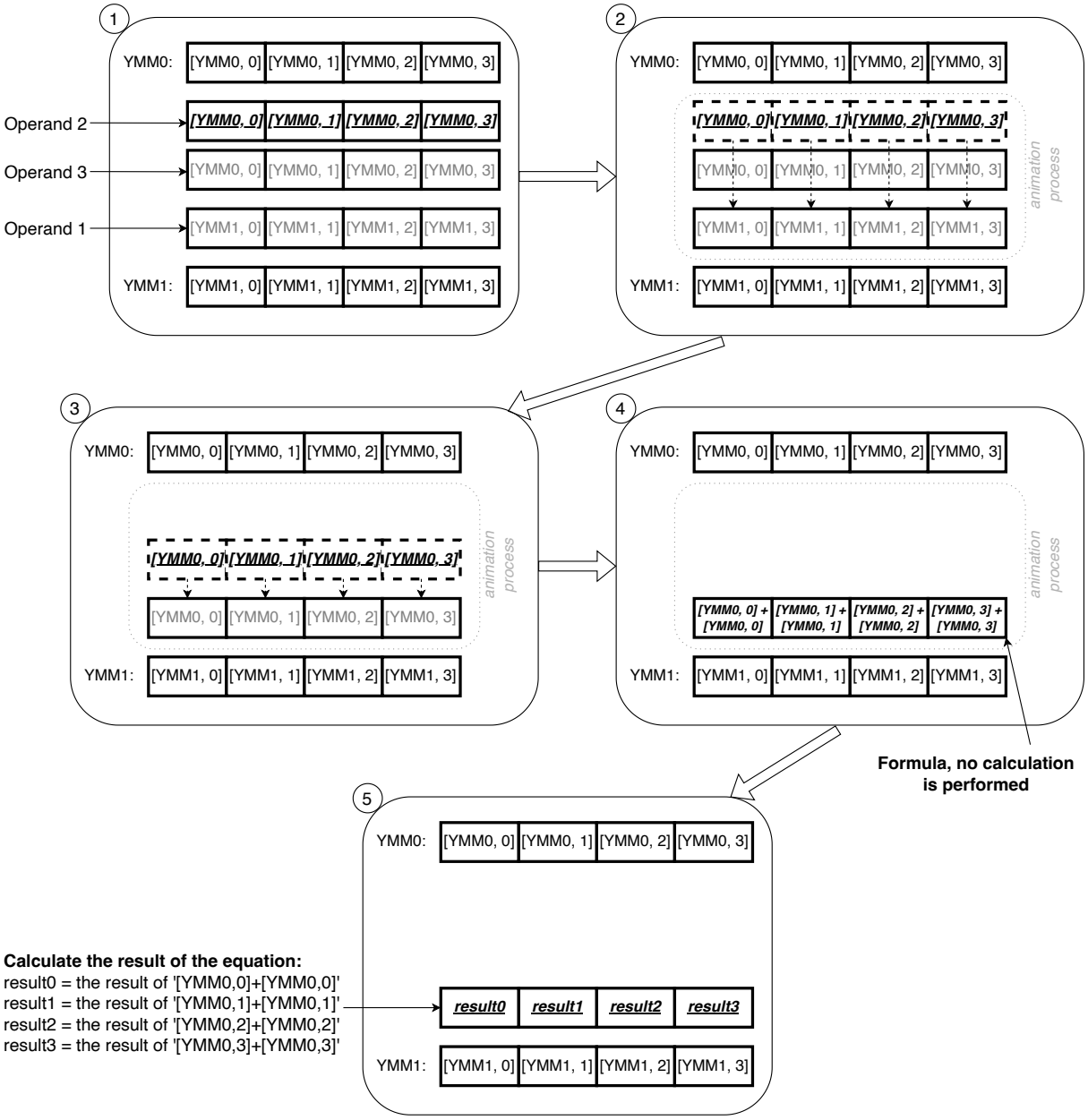


Figure 3.17: Animation design of calculation instruction visualization in Method2 - Execute stage

- *Support for Low-Level Operations*: Rust provides good support for low-level data and operations, exemplified by the development of cpulib in Rust.
- *Modern Language Features*: Rust supports modern language features, such as closures and iterators, which can make the code more concise and efficient.
- *Safety*: Rust's borrow checker ensures memory safety, significantly reducing bugs in the application.
- *High Performance*: Rust offers performance comparable to C/C++ while ensuring safety.
- *Cross-Platform*: Rust can run on a variety of platforms and supports common operating systems

as well as WebAssembly[29].

The reasons for "Prototype2" choosing the egui library are:

- *Immediate Mode GUI*: egui eliminates the need for the complex state and lifecycle management of traditional retained mode GUIs.
- *Declarative UI Programming*: Provides a higher level of abstraction, accelerating application development.
- *Cross-Platform*: egui can be compiled on a variety of platforms, including WebAssembly.

3.3.2 Software Architecture

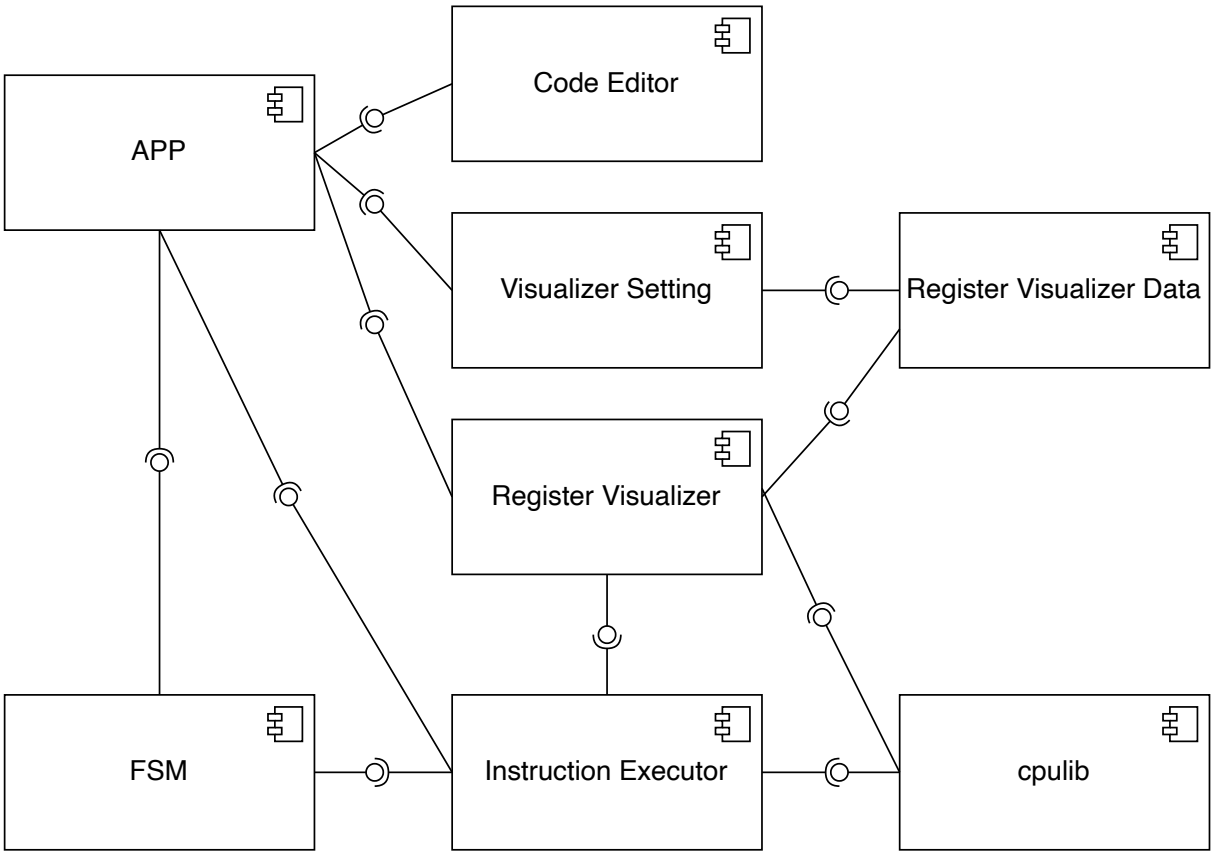


Figure 3.18: Design of Prototype2

The software architecture of Prototype2 is shown in Figure 3.18 with a UML component diagram. Prototype2 has significantly improved the architecture compared to Prototype1 by decomposing most of the functional modules, reducing the coupling between each module, and making the functions of each module clearer. According to the description in Figure 3.18, the **APP** component (i.e., the main class of the software) uses interfaces from most of the components because it needs to render all graph-

ical interfaces from these components. In fact, the operation of the **APP** component depends only on the **Instruction Executor** component (providing instruction execution functions) and the **FSM** component (providing a finite state machine to aid instruction execution) because of its integrated code debugging functionality. The **Register Visualizer Data** component provides management of visualization data, so the **Register Visualizer** component (providing visualization functionality) and **Visualizer Setting** component (providing visualization setting functionality) depend on it. In addition, the **cpulib** component provides management of CPU data, so both the **Register Visualizer** component and **Instruction Executor** component depend on it.

3.3.3 GUI Design

Prototype1 is designed with a single GUI interface, while Prototype2 utilizes the egui framework to implement a multi-interface GUI design. Users can open or close function windows as needed. To explain the GUI design of Prototype2, I still use the graphical primitives introduced in Figure 3.8.

As shown in Figure 3.19, the GUI design of Prototype2 is composed of main windows including main three windows. The overall GUI design of Prototype2 resembles a standard software made up of a top menu bar, left sidebar, and main window on the right. The menu bar provides basic functionalities such as opening and saving files, displaying, or closing the sidebar, etc. The sidebar presents the main functionality like visualization window, debugging window, etc. The main window is the main function window, offering a code editor. This design makes the software functions clearer, users can open or close function windows as needed, without being forced to use all function windows.

The visualization setting provides a series of setting functionalities for visualization, including animation play speed, adding or removing registers, sorting registers, etc. With these features, users can customize the visual effects as needed. The visualization window presents according to the settings in the visualization setting window.

3.3.4 Code Editor

The code editor in Prototype2 uses a third-party library, egui_code_editor, since the egui framework itself does not offer a professional code editor widget supporting line number display. The egui_code_editor is a text editor widget based on egui, with line number displays and simple syntax highlighting based on keyword sets[30]. However, egui_code_editor lacks line number highlighting functionality needed for debugging and highlighting functionality for assembly code syntax. Therefore, I improved upon egui_code_editor by adding line number highlighting and syntax highlighting

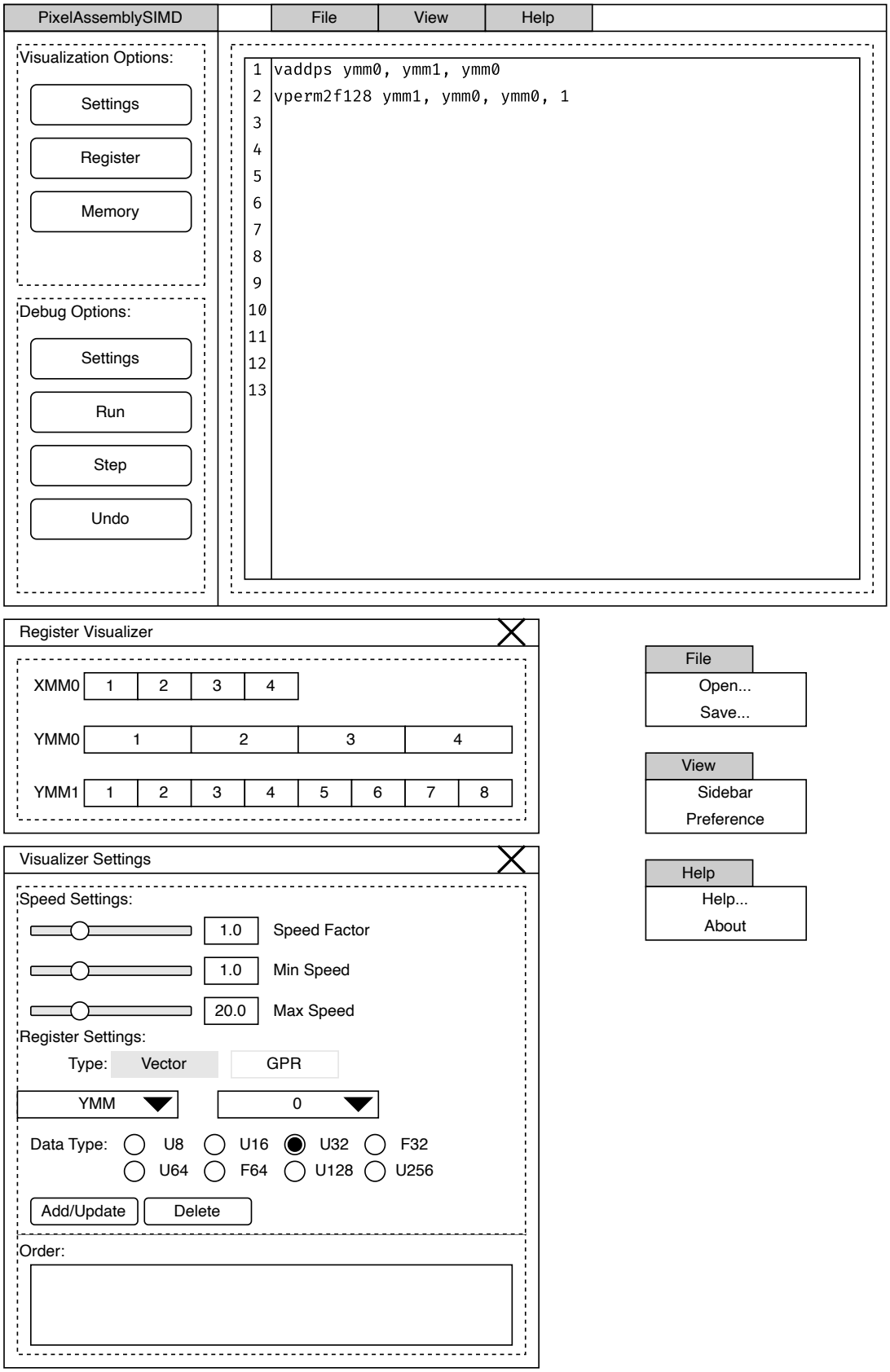


Figure 3.19: GUI Design of Prototype2

functionality for assembly code. Currently, the modification of assembly language syntax highlighting has been submitted as a Pull Request on GitHub and merged into the main branch of the project.

3.3.5 CPU Emulator: cpulib

To address data reliability and restore the operation method of CPU on registers, Prototype2 uses an independent CPU emulation library "cpulib" developed based on the Rust language to emulate the execution process of CPU.

Figure 3.20 uses UML class diagram describes the data structure of cpulib and its relationships. cpulib provides a CPU structure, which includes components of CPU such as register files, and memory, and provides operation methods for register files, and memory, etc.

- *Register Operations*：cpulib categorizes registers into four major types and provides respective methods for operating each type of register. In the four types of registers, in addition to GPR and vector registers, it also provides support for status registers and instruction counters. Furthermore, for vector registers, cpulib provides a variety of methods for reading or writing register values: full reads or writes, reading or writing vectors according to fixed data widths, and reading or writing based on selectors (selectors serve as selection means just like pseudocodes in the Intel official manual, `XMM0[63:0]` signifies selecting the lower 64 bits of the `XMM0` register). The implementation of these methods aspires for similar actual operations with the CPU, for example, cpulib showcases the physical relationships between registers such as the `XMM` register being the lower 128 bits of the `YMM` register, `EAX` being the lower 32 bits of the `RAX` register, etc.
- *Memory Operations*：cpulib supports not only register operations but also memory emulations. It provides methods for reading and writing memory while supporting different data widths for these operations. Moreover, cpulib supports batch reads and writes for memory, and such a design can better emulate CPU operations on the memory. To improve performance, operations on cpulib's memory are lazy, meaning that the actual data is written into the virtual memory page only when necessary. When reading blank pages, it directly returns 0. This design enhances cpulib's performance in emulating programs with a high number of memory accesses.

While designed, cpulib was referred to pseudocodes in the Intel official manual, aiming to mimic the actual CPU operations as much as possible. In addition, as a CPU emulator mainly focused on emulating instruction sets, cpulib can bypass issues found in emulators like QEMU. For instance, the large

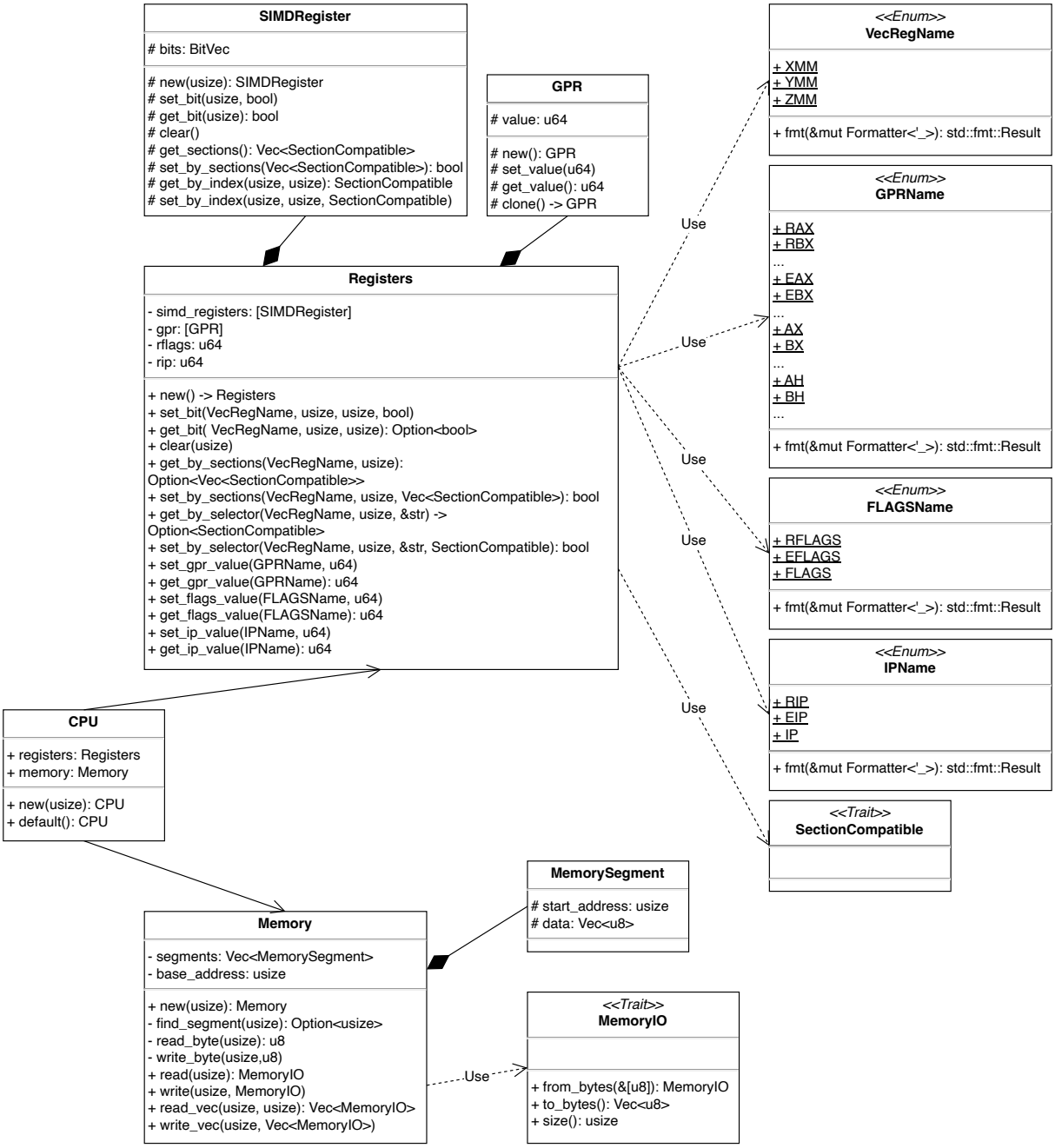


Figure 3.20: Design of cpulib

amount of I/O hardware emulation in QEMU is redundant for visualized instruction tools. Moreover, it considered AVX-512 expansion support during design, allowing cpulib to emulate AVX-512 architectural instructions. Meanwhile, a similar instruction set emulator, Unicorn, does not yet support AVX-512 extensions.

3.3.6 Animation Executor Based on egui

The egui library doesn't offer native animation functionality; thus, I have implemented an animation executor based on the egui library that can produce simple animation sequences. There's a challenge here: egui is a synchronous library that doesn't handle asynchronous operations directly nor offers any inbuilt asynchronous or concurrent features. Most animations tend to use event-driven asynchronous operations, such as the Anime.js library, which bases animation on JavaScript's asynchronous operations. To solve this problem, I used Rust's signals and closures to construct a simple animation library featuring asynchronous operations. It's non-blocking and event-driven and can seamlessly integrate with egui's main render loop to generate smooth animation sequences.

- *Non-blocking*：The animation library is non-blocking, meaning the animation execution won't block other parts of the program. Even with an ongoing animation, the program can still respond to user inputs, manage other events, or execute other tasks.
- *Event-driven*：The animation library is event-driven, controlling the animation flow through callback functions and signal mechanisms. After an animation segment ends, it can automatically trigger the next segment or another operation, indicating that animation control is event-based rather than sequential execution or synchronous waiting.

The design of the animation executor, described in Figure 3.21 based on the UML state diagram, primarily functions to execute animation sequences. Animation sequences composed of multiple animation groups, they're sequentially executed when the animation sequence runs. The execution of animation is signal-based; each complete animation sends a signal caught by the animation executor, triggering the execution of the next segment. This design makes the execution of animation sequences non-blocking and event-driven. The animations executed by this executor demonstrate asynchronous behavior in a synchronous system since the animation execution process doesn't impede the program from running other tasks, and the animation control is manipulated through callback functions and signal mechanisms. The key point is the decoupling of animation handling from other parts of the program.

According to the description in Figure 3.21, when the target position is the same as the UI element's current position (no animation) and no events (signals) are received, the software is in a rendering state, continuously rendering UI elements. If it is found that the current position of the UI element is different from the target position, animation execution begins. At this time, the position of the UI element will be updated to become closer to the target position. The specific increment is determined by the

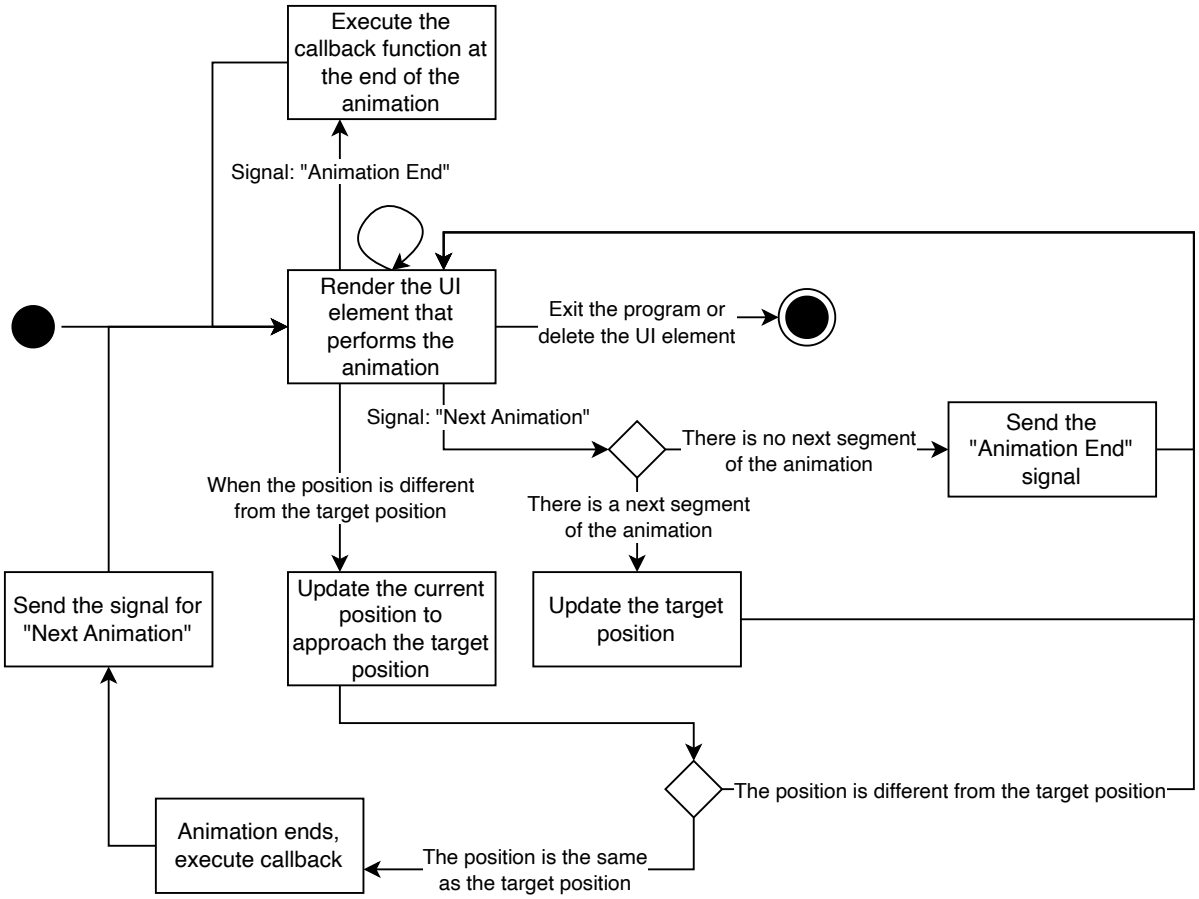


Figure 3.21: Animation executor based on egui in Prototype2

speed setting of the animation; after updating the current position, if it overlaps with the target position, the callback function after the end of the animation will be executed, and then it will return to the rendering state after sending the ”Next Animation” signal; if it does not overlap with the target position, it will directly return to the rendering state. When in the rendering state, two different signals can be processed:

- *Next Animation*：Start the next animation. This represents that the previous animation has ended and the next animation can be started. After receiving this signal, the target position of the UI element will be set according to the next animation set, and then it returns to the rendering state.
- *Animation End*：Animation ends. Receiving this signal means that the set animation sequence has been completed. At this time, the callback function after the end of the animation sequence will be executed, then it will return to the rendering state.

3.3.7 Instruction Executor

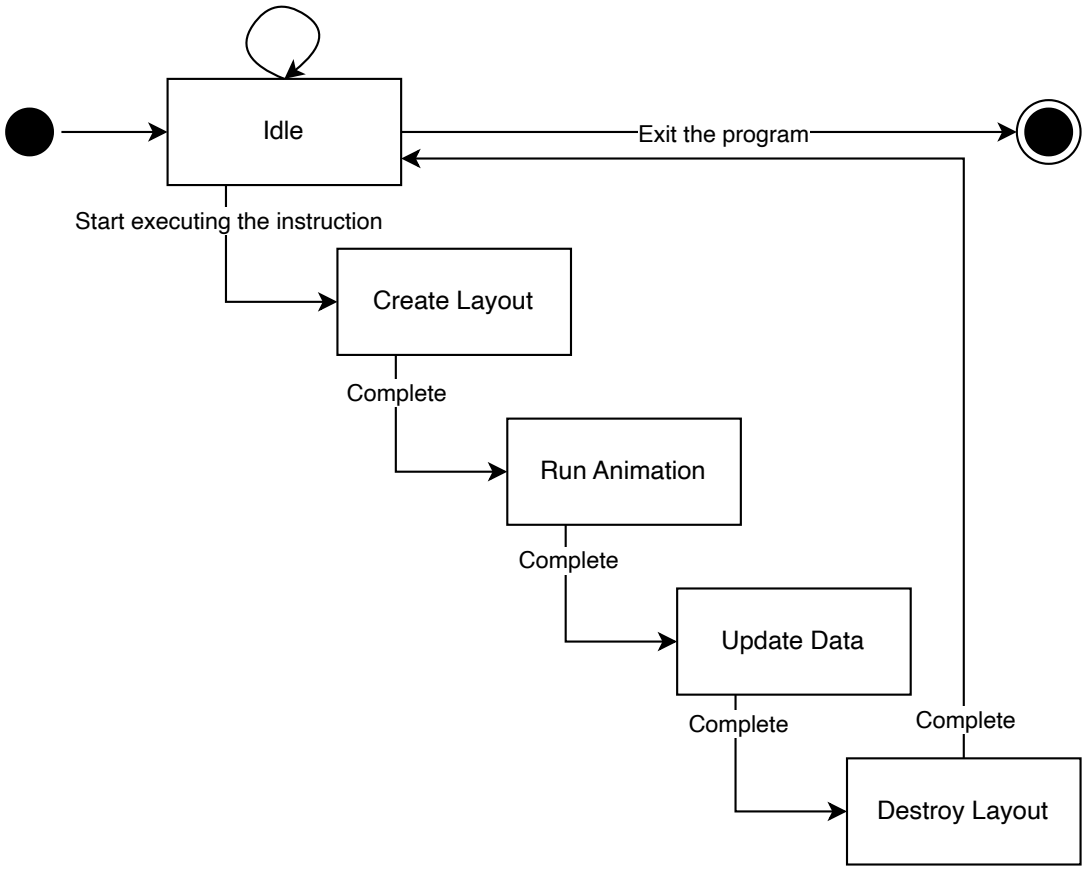


Figure 3.22: Instruction Executor in Prototype2

The Prototype2 implements command execution based on the state machine. In Prototype Implementation 2, if you want to perform command animation, you first need to create a copy of the target register, which requires additional space, so you need to create a Layout of the register copy during the layout. After creation, the animation can start. After the animation ends, the data in cpulib needs to be updated. Finally, delete the created Layout, and one animation process ends. Figure 3.22 describes the state transition during this process.

4

Implementation of PixelAssemblySIMD

This chapter will detail the implementation process of the design described in Chapter 3. This chapter first introduces the implementation of the CPU emulation library cpulib, and then introduces the specific implementation process of PixelAssemblySIMD from the aspects of GUI, animation executor, and instruction executor.

4.1 CPU Emulator: cpulib

4.1.1 External Interface

Listing 4.1: External Interface of cpulib (Structure)

```
pub struct CPU {
    pub registers: Registers,
    pub memory: Memory,
}
```

The external interface of cpulib is very simple. It exposes the internal register and memory-related structures directly. When it is called from the outside, the exposed interfaces of the register and memory structures can be used directly. The code is shown in Listing 4.1.

Listing 4.2: Implementation of the external interface of cpulib

```
impl CPU {
    pub fn new(base: usize) -> Self {
        CPU {
            registers: Registers::new(),
            memory: Memory::new(base)
        }
    }
}
```

```
impl Default for CPU {
    fn default() -> Self {
        CPU::new(0x00400000usize)
    }
}
```

Next, cpulib implements `new` and `default`, two methods for creating CPU instances, as shown in Listing 4.2. The `default` method calls the `new` method, creating a default CPU instance using the default memory base address value `0×00400000`.

4.1.2 Register

Structure Definitions

Listing 4.3: Structural Definitions of Registers in cpulib

```
struct SIMDRegister {
    bits: BitVec,
}
struct GPR {
    value: u64,
}
pub struct Registers {
    simd_registers: [SIMDRegister; 32],
    gpr: [GPR; 16],
    rflags: u64,
    rip: u64,
}
```

In cpulib, there are three structures related to registers: `SIMDRegister`, `GPR`, and `Registers`, as illustrated in Listing 4.3. The `SIMDRegister` structure represents a vector register and contains a BitVec structure to represent the data in the SIMD register. The `GPR` structure represents a general-purpose register and contains a u64 type value to represent the data in the general-purpose register. The `Registers` structure represents the entire set of registers, containing 32 vector registers, 16 general-purpose registers, an `RFLAGS` register, and a `RIP` register. Although only 32 512-bit vector registers and 16 64-bit general-purpose registers are represented here, based on the physical implementation of the CPU, other vector registers and general-purpose registers with lower data widths are essentially the lower bits of these registers. Thus, operations on these registers can also facilitate operations on registers with lower data widths.

Method Implementation

The `SIMDRegister` in cpulib includes a series of read and write methods, including reading and writing individual bits and reading and writing the entire register as an array. The bit-level read and write operations are the most fundamental, with other operations built upon them. The implementation of bit-level read and write operations is shown in Listing 4.4.

Listing 4.4: Implementation of Bit Read and Write in `SIMDRegister` of cpulib

```
fn set_bit(&mut self, position: usize, value: bool) {
    self.bits.set(position, value);
}
fn get_bit(&self, position: usize) -> bool {
    self.bits[position]
}
```

In these methods, the `set_bit` method sets the bit at the specified position to either `0` or `1`; the `get_bit` method returns the value of the bit at the specified position.

The implementation of methods for reading and writing the entire register is shown in Listing 4.5.

Listing 4.5: Implementation of Register Read and Write in `SIMDRegister` of cpulib

```
fn get_sections<T: SectionCompatible>(&self) -> Vec<T> {
    let mut sections = Vec::new();
    let type_bits = std::mem::size_of::<T>() * 8;
    for i in (0..self.bits.len()).step_by(type_bits) {
        let mut section_value: T = T::from(0u8);
        for j in 0..type_bits {
            if i + j ⩾ self.bits.len() {
                break;
            }
            if self.bits[i + j] {
                section_value = section_value | (T::from(1u8) << j);
            }
        }
        sections.push(section_value);
    }
    sections
}
fn set_by_sections<T: SectionCompatible>(&mut self, sections: Vec<T>) -> bool {
    self.clear();
    let type_bits = std::mem::size_of::<T>() * 8;
    if type_bits * sections.len() != self.bits.len() {
        return false;
    }
    let mut i = 0;
```

```
25      for section in &sections {
26          for j in 0..type_bits {
27              if i + j ≥ self.bits.len() {
28                  break;
29              }
30              if (*section >> j) & T::from(1u8) == T::from(1u8) {
31                  self.set_bit(i + j, true);
32              }
33          }
34          i += type_bits;
35      }
36      true
37  }
```

In these methods, the `get_sections` method divides the data in the register according to the specified data type and then stores the divided data in an array to be returned; the `set_by_sections` method writes the data from the specified array into the register according to the specified data type.

The read and write methods in the `GPR` structure of cpulib are relatively simple, consisting only of methods for reading and writing the entire register. The implementation of these methods is shown in Listing 4.6.

Listing 4.6: Implementation of Register Read and Write in `GPR` of cpulib

```
1  fn get_value(&self) -> u64 {
2      self.value
3  }
4  fn set_value(&mut self, val: u64) {
5      self.value = val;
6  }
```

In these methods, the `get_value` method returns the data in the register; the `set_value` method writes the specified data into the register.

The last to be introduced are the read and write methods in `Registers` of cpulib, which are wrappers around the read and write methods in `SIMDRegister` and `GPR`. Therefore, only the read operation wrappers in `Registers` are shown here, as the write operation wrappers are similar to the read operations. First is the operation for reading bits of vector registers, as shown in Listing 4.7.

Listing 4.7: Implementation of Reading Vector Register Bits in `Registers` of cpulib

```
1  pub fn get_bit(&self, reg_type: VecRegName, reg_index: usize, bit_position: usize)
     -> Option<bool> {
2      match reg_type {
3          VecRegName::XMM if bit_position < 128 => {
4              Some(self.simd_registers[reg_index].get_bit(bit_position))
```

```
        }
        VecRegName::YMM if bit_position < 256 => {
            Some(self.simd_registers[reg_index].get_bit(bit_position))
        }
        VecRegName::ZMM if bit_position < 512 => {
            Some(self.simd_registers[reg_index].get_bit(bit_position))
        }
        _ => None,
    }
}
```

In this method, the corresponding vector register is first located based on the register name and index, and then the specific bit is located based on its position, finally returning the value of the bit. When reading, it is necessary to differentiate based on the size of the vector register, for example, reading `XMM` registers must be within 128 bits to be valid. If the position of the bit to be read exceeds the data width of the vector register, `None` is returned.

Next is the operation for reading the entire vector register, as shown in Listing 4.8.

Listing 4.8: Implementation of Reading Vector Registers in `Registers` of cpulib

```
pub fn get_by_sections<T: SectionCompatible>(&self, reg_type: VecRegName, reg_index
  : usize) -> Option<Vec<T>> {
    let sections: Vec<T> = self.simd_registers[reg_index].get_sections();
    match reg_type {
        VecRegName::XMM => {
            let n = 128 / (std::mem::size_of::<T>() * 8);
            let slice = &sections[..n];
            Some(slice.to_vec())
        }
        VecRegName::YMM => {
            let n = 256 / (std::mem::size_of::<T>() * 8);
            let slice = &sections[..n];
            Some(slice.to_vec())
        }
        VecRegName::ZMM => {
            let n = 512 / (std::mem::size_of::<T>() * 8);
            let slice = &sections[..n];
            Some(slice.to_vec())
        }
    }
}
```

In this method, the corresponding vector register is first located based on the register name and index, and then the data in the vector register is divided according to the specified data type, finally returning

the divided data.

Following this is the operation for reading GPRs, as shown in Listing 4.9.

Listing 4.9: Implementation of Reading GPRs in `Registers` of cpulib

```
pub fn get_gpr_value(&self, reg_name: GPRName) -> u64 {
    register_get!(self; reg_name;
        RAX, EAX, AX, AL, AH,
        RBX, EBX, BX, BL, BH,
        RCX, ECX, CX, CL, CH,
        RDX, EDX, DX, DL, DH;
        R8, R8D, R8W, R8B,
        R9, R9D, R9W, R9B,
        R10, R10D, R10W, R10B,
        R11, R11D, R11W, R11B,
        R12, R12D, R12W, R12B,
        R13, R13D, R13W, R13B,
        R14, R14D, R14W, R14B,
        R15, R15D, R15W, R15B,
        RSP, ESP, SP, SPL,
        RBP, EBP, BP, BPL,
        RSI, ESI, SI, SIL,
        RDI, EDI, DI, DIL
    )
}
```

This method uses a macro to implement the reading operation for GPRs, with the definition of the macro shown in Listing 4.10.

Listing 4.10: Macro for Reading GPRs in `Registers` of cpulib

```
macro_rules! register_get {
    ($self:ident; $reg_name:expr; $( $r64:ident, $r32:ident, $r16:ident, $r8_l:ident, $r8_h:ident ),*; $( $r64_:ident, $r32_:ident, $r16_:ident, $r8_:ident ),*) => {
        match $reg_name {
            $(
                GPRName::$r32 => $self.gpr[GPRName::$r64 as usize].value & 0x00000000_FFFFFFFF,
                GPRName::$r16 => $self.gpr[GPRName::$r64 as usize].value & 0x00000000_0000FFFF,
                GPRName::$r8_l => $self.gpr[GPRName::$r64 as usize].value & 0x00000000_000000FF,
                GPRName::$r8_h => ($self.gpr[GPRName::$r64 as usize].value & 0x00000000_0000FF00) >> 8,
            )*
            $(
```

```
                GPRName::$r32_ => $self.gpr[GPRName::$r64_ as usize].value & 0
    x00000000_FFFFFFFF,
                GPRName::$r16_ => $self.gpr[GPRName::$r64_ as usize].value & 0
    x00000000_0000FFFF,
                GPRName::$r8_ => $self.gpr[GPRName::$r64_ as usize].value & 0
    x00000000_000000FF,
            )*
            _ => $self.gpr[$reg_name as usize].get_value(),
        }
    }
}
```

This macro first locates the corresponding GPR based on its name, and then returns the data according to the data width of the GPR. For example, for RAX, which has a 64-bit data width, the entire data in the GPR is returned; for AX, which has a 16-bit data width, the lower 16 bits of data in the GPR are returned.

Finally, `Registers` also provides methods for reading and writing RFLAGS and RIP, which are similar to the methods for reading and writing GPRs, both involving reading and writing a `u64` value. Only the read operations are shown here, as shown in Listing 4.11.

Listing 4.11: Implementation of Reading RIP and RFLAGS in `Registers` of cpulib

```
pub fn get_ip_value(&self, reg_name: IPName) -> u64 {
    match reg_name {
        IPName::RIP => {
            self.rip
        },
        IPName::EIP => {
            self.rip & 0x00000000_FFFFFFFF
        },
        IPName::IP => {
            self.rip & 0x00000000_0000FFFF
        }
    }
}
pub fn get_flags_value(&self, reg_name: FLAGSName) -> u64 {
    match reg_name {
        FLAGSName::RFLAGS => {
            self.rflags
        },
        FLAGSName::EFLAGS => {
            self.rflags & 0x00000000_FFFFFFFF
        },
        FLAGSName::FLAGS => {
```

```
23                 self.rflags & 0x00000000_0000FFFF
24             }
25         }
26     }
```

4.1.3 Memory

Structure Definitions

Listing 4.12: Structural Definitions of Memory in cpulib

```
1  struct MemorySegment {
2      start_address: usize,
3      data: Vec<u8>,
4  }
5  pub struct Memory {
6      segments: Vec<MemorySegment>,
7      base_address: usize,
8  }
```

In cpulib, there are two structures related to memory, namely `MemorySegment` and `Memory`, as shown in Listing 4.12. The `MemorySegment` structure is used to represent a continuous segment of memory, containing the start address and the data of that memory segment. The `Memory` structure represents the entire memory, containing all memory segments and the base address of the memory.

Method Implementation

The `MemorySegment` structure in cpulib does not include any methods, while the `Memory` structure includes a series of read and write methods. These methods cover operations like reading and writing individual bytes, single data items (integers of various data widths), and multiple data items (arrays of integers of various data widths). Byte reading and writing is the most fundamental operation, with other read and write operations built upon it. The implementation of byte reading and writing operations is shown in Listing 4.13.

Listing 4.13: Implementation of Byte Read and Write in `Memory` of cpulib

```
1  fn read_byte(&self, address: usize) -> u8 {
2      let real_address = address - self.base_address;
3      if let Some(index) = self.find_segment(real_address) {
4          self.segments[index].data[real_address - self.segments[index].start_address
       ]
5      } else {
```

```
6           0
7       }
8   }
9   fn write_byte(&mut self, address: usize, value: u8) {
10      let real_address = address - self.base_address;
11      if let Some(index) = self.find_segment(real_address) {
12          let start = self.segments[index].start_address;
13          self.segments[index].data[real_address - start] = value;
14      } else {
15          let adjusted_address = (real_address / DEFAULT_SIZE) * DEFAULT_SIZE;
16          let mut new_data = Vec::with_capacity(DEFAULT_SIZE);
17          new_data.resize(DEFAULT_SIZE, 0);
18          new_data[real_address - adjusted_address] = value;
19          let new_segment = MemorySegment {
20              start_address: adjusted_address,
21              data: new_data,
22          };
23          self.segments.push(new_segment);
24          self.segments.sort_by(|a, b| a.start_address.cmp(&b.start_address));
25      }
26      let mut i = 0;
27      while i + 1 < self.segments.len() {
28          if self.segments[i].start_address + self.segments[i].data.len() == self.
        segments[i + 1].start_address {
29              let next = self.segments.remove(i + 1);
30              self.segments[i].data.extend(next.data);
31          } else {
32              i += 1;
33          }
34      }
35  }
```

In these methods, the corresponding memory segment is first located based on the address. The offset in the memory segment is calculated using the address and the start address of the segment, and the data at that offset is then returned. If no corresponding memory segment exists, `0` is returned. When writing data, the corresponding memory segment is located and the offset is calculated in a similar manner, after which the data is written at that offset. If no corresponding memory segment is found, a new segment is created, and the data is written there. After writing data, the memory segments are merged, combining contiguous memory segments into a single segment.

The implementation of reading and writing data and data arrays is based on the byte read and write operations, as shown in Listing 4.14.

Listing 4.14: Implementation of Data Read and Write in `Memory` of cpulib

```rust
pub fn read<T: MemoryIO>(&self, address: usize) -> T {
    let mut bytes = Vec::new();
    for i in 0..T::size() {
        bytes.push(self.read_byte(address + i));
    }
    T::from_bytes(&bytes)
}
pub fn write<T: MemoryIO>(&mut self, address: usize, value: T) {
    let bytes = value.to_bytes();
    for (i, byte) in bytes.iter().enumerate() {
        self.write_byte(address + i, *byte);
    }
}
pub fn read_vec<T: MemoryIO>(&self, address: usize, number_of_value: usize) -> Vec<
  T> {
    let mut result: Vec<T> = vec![];
    for i in 0..number_of_value {
        result.push(self.read(address + i * T::size()));
    }
    result
}
pub fn write_vec<T: MemoryIO + Clone>(&mut self, address: usize, values: Vec<T>) {
    for (i, value) in values.iter().enumerate() {
        self.write(address + i * T::size(), value.clone());
    }
}
```

The `read` method reads a piece of data by reading the corresponding number of bytes according to the data size and then converting these bytes into the data to be returned. The `write` method writes a piece of data by converting the data into bytes and writing the corresponding number of bytes according to the data size. The `read_vec` method reads an array of data by calling the `read` method and storing the multiple pieces of data read in an array to be returned. The `write_vec` method writes an array of data by iterating through each piece of data in the array and using the `write` method to write each piece of data into memory.

4.2 GUI

4.2.1 Main Window

Listing 4.15: `main` function of PixelAssemblySIMD

```rust
struct APP {
    // Member ...
}
impl eframe::App for APP {
    // Methods ...
}
fn main() -> Result<(), eframe::Error> {
    let options = NativeOptions {
        viewport: egui::ViewportBuilder::default().with_inner_size([1200f32, 800f32
  ]),
        ..Default::default()
    };
    eframe::run_native(
        "PixelAssemblySIMD",
        options,
        Box::new(|_cc| Box::new(APP::default())),
    )
}
```

The crux of PixelAssemblySIMD is an egui-based GUI application. Therefore, the `main` function of the program displays the GUI part of the program by loading an instance of a structure with `App` traits from the eframe framework of egui, as shown in Listing 4.15. The `main` function creates a GUI application by executing the `run_native` method of eframe and returning it. This method specifies the title of the window as "PixelAssemblySIMD" and sets the window size to 1200*800 through the `options` property. Finally, it loads an instance of the `APP` structure that implements the `App` trait of eframe to execute the custom GUI interface.

According to the design of PixelAssemblySIMD, the `APP` structure will execute all the functions in PixelAssemblySIMD. Therefore, components including the animation executor and instruction executor exist as members of the `APP` structure. The specific implementation of the `APP` structure is shown in Listing 4.16.

Listing 4.16: Definition of the `APP` structure in PixelAssemblySIMD

```rust
struct APP {
    cpu: Arc<Mutex<CPU>>,
    reg_visualizer_data: RegVisualizerData,
    register_visualizer: Arc<Mutex<RegVisualizer>>,
    visualizer_setting: VisualizerSetting,
    animation_fsm: AnimationFSM,
    code: String,
    highlight: usize,
    // ...
```

```
10 }
```

The members of the APP structure shown in Listing 4.16 are, from top to bottom, as follows: an instance of CPU data (CPU emulator), an instance of visualization data, an instance of the window for register visualization, an instance of the window for visualization settings, an instance of the FSM of the instruction executor, the code input, and the row number highlighted in the editor. Some of the members are wrapped in mutex locks and reference counters, allowing them to pass multiple distinct references in a thread-safe manner.

Listing 4.17: Implementation of `update` in APP structure

```
1  impl eframe::App for APP {
2      fn update(&mut self, ctx: &eframe::egui::Context, frame: &mut eframe::Frame) {
3          eframe::egui::TopBottomPanel::top("top_panel").show(ctx, |ui| {
4              egui::menu::bar(ui, |ui| {
5                  egui::menu::menu_button(ui, "View", |ui| {
6                      if ui.selectable_label(self.show_sidebar, "Sidebar").clicked()
    {
7                          self.show_sidebar = !self.show_sidebar;
8                      }
9                  });
10             });
11         });
12         if self.show_sidebar {
13             eframe::egui::SidePanel::left("side_panel").show(ctx, |ui| {
14                 ui.vertical(|ui| {
15                     ui.label("Visualization␣Options:");
16                     if ui.button("Settings").clicked() {
17                         self.show_settings = true;
18                     }
19                     if ui.button("Visualizer").clicked() {
20                         self.show_visualizer = true;
21                     }
22                     ui.label("Debug␣Options:");
23                     if ui.button("Step").clicked() {
24                         // debug function ...
25                     }
26                 });
27             });
28         }
29         eframe::egui::CentralPanel::default().show(ctx, |ui| {
30             CodeEditor::default()
31                 .id_source("code_editor")
32                 .with_rows(24)
33                 .with_fontsize(14.0)
```

```
34                  .with_theme(ColorTheme::GRUVBOX)
35                  .with_syntax(Syntax::asm())
36                  .with_numlines(true)
37                  .show(ui, &mut self.code, &mut self.highlight);
38          });
39      }
40  }
```

The APP structure implements the App trait in eframe, which requires the structure to implement an `update` method. The `update` method is executed in every rendering loop of egui, rendering the user-defined GUI components. Since egui defines GUI based on declarative UI programming, we define the main GUI interface using different layouts and components based on the layout position provided by egui. For example, in Listing 4.17, we first defined a menu bar at the top, and in the menu bar, we defined a "View" menu that contains a "Sidebar" option. When the user clicks on this option, the left sidebar will be shown or hidden. In the sidebar, we defined a vertical layout that includes two parts: "Visualization Options" and "Debug Options," each containing a series of buttons for opening windows related to visualization and debugging. In the center of the main interface, we used egui_code_editor to define a code editor for users to input codes.

Listing 4.18: Windows in PixelAssemblySIMD

```
1   Window::new("Settings")
2       .open(&mut self.show_settings)
3       .show(ctx, |ui| {
4           self.visualizer_setting.show(ui, &mut self.reg_visualizer_data);
5       });
6   Window::new("Visualizer")
7       .open(&mut self.show_visualizer)
8       .show(ctx, |ui| {
9           let delta_time = ctx.input(|input|{
10              input.unstable_dt
11          });
12          let mut register_visualizer = self.register_visualizer.lock().unwrap();
13          register_visualizer.update(delta_time, self.reg_visualizer_data.factor,
      self.reg_visualizer_data.min_speed, self.reg_visualizer_data.max_speed);
14          let cpu = self.cpu.lock().unwrap();
15          register_visualizer.show(ui, &self.reg_visualizer_data, &cpu);
16          drop(cpu);
17          register_visualizer.move_animation_sequence(ctx);
18          register_visualizer.move_animation_finish(ctx);
19          if register_visualizer.is_animating() {
20              ctx.request_repaint();
21          }
```

```
22          drop(register_visualizer);
23          self.animation_fsm.run();
24      });
```

Important function windows also include: visualization settings window and register visualizer window. These windows are implemented by the `Window` component of egui, as shown in Listing 4.18. Among them, the visualization settings window is mainly implemented by an instance of the `VisualizerSetting` structure. The window only calls the `show` method of this instance to display the window; The register visualizer window is more complex, with the visualization part mainly implemented by an instance of the `RegVisualizer` structure. First, this window calls the `update` method of this instance, which is used to update visualized data, so it needs to pass in animation-related parameters, such as speed, time, etc.; After updating the data, the `show` method is called to display the window. In addition, the window calls the `move_animation_sequence` and `move_animation_finish` methods, which are mainly used to execute and detect animation sequences. After all GUI elements are output, the window immediately redraws the interface when an animation occurs, ensuring the smooth display of the animation. Finally, this window executes the instruction executor's FSM to track the state of the instruction execution.

4.2.2 Visualization Setting Window

Listing 4.19: Definition of the `VisualizerSetting` structure in PixelAssemblySIMD

```
1  pub struct VisualizerSetting {
2      factor: f32,
3      min_speed: f32,
4      max_speed: f32,
5      reg_type: RegType,
6      gpr_name: GPRName,
7      vec_name: VecRegName,
8      vec_index: usize,
9      data_type: ValueType,
10 }
```

The structure of `VisualizerSetting` is implemented as shown in Listing 4.19. This structure defines the settings related to visualization, from top to bottom, they are: the factor of animation execution speed, the minimum speed of animation execution, the maximum speed of animation execution, the selected register type, the name of the selected GPR and vector register, the index of the selected vector register, the data type in the register. The most important method implemented in this structure

is the `show` method. The implementation of this method is relatively simple, it uses the UI components provided by egui to implement the design of this window, so the code is not described in detail.

4.2.3 Register Visualization Window

Most of the content in the `RegVisualizer` structure instance is part of the animation executor, so here only the parts related to graphics display in the `RegVisualizer` structure are introduced.

Element Structure

The basic "element" used for displaying the register in `RegVisualizer` is the `Element` structure. This structure is used to define and display each register element, that is, a GPR or a element in vector registers. The definition of this structure is shown in Listing 4.20.

Listing 4.20: Definition of the `Element` structure in PixelAssemblySIMD

```
pub struct Element {
    // Data
    value: Value,
    string: Option<String>,
    // Animation
    display: bool,
    order: ElementOrder,
    color: Color32,
    border_color: Color32,
    is_highlight: bool,
    layout_position: Pos2,
    position: Pos2,
    target_position: Pos2,
    animating: bool,
    animation_finished_callback: Option<Box<dyn FnOnce(&mut Self) + Send + 'static
>>,
}
```

The `Element` structure includes data related to an element and animation-related attributes. The data attributes include: the value of the data, the string form of the data; the animation-related attributes include: whether to display, display order, color, border color, whether to highlight, layout position, current position, target position, a flag indicating whether an animation is in progress, and a one-time callback function that can be executed after the animation ends.

In addition, the `Element` structure has implemented `update` and `show` methods. The `update` method is called by the `update` method of `RegVisualizer`, used to update the information of each

element; the `show` method is called by the `show` method of `RegVisualizer`, used to display each element. The implementation of the `show` method is relatively simple, it uses the drawing feature of egui to draw a filled rectangle and an empty rectangle at the current position, which serve as the display of the color of the element and the color of the border; then it draws the string form of the data in the center of the rectangle; the basic logic of the code is as shown in Listing 4.21. The `update` method is used to update the information of the elements, which is the most basic feature of the animation implementation, and therefore will be described in the next section.

Listing 4.21: Implementation of `show` in `Element` structure

```
fn show(&self, ui: &mut Ui) {
    let rect_size = get_size_from_value(&self.value);
    // Display Rectangle
    ui.painter().rect_filled(
        Rect::from_min_size(self.position, rect_size),
        0.0,
        self.color,
    );
    // Display Border
    ui.painter().rect_stroke(
        Rect::from_min_size(self.position, rect_size),
        0.0,
        egui::Stroke::new(2.0, if self.is_highlight {Color32::RED} else if self.animating {Color32::KHAKI} else {self.border_color}),
    );
    // Display Text
    let text_pos = self.position + rect_size / 2.0 - Vec2::new(text_size / 2.0, font_size / 2.0);
    let galley = ui.painter().layout_no_wrap(
        if let Some(text) = &self.string {
            format!("{}", text)
        } else {
            format!("{}", self.value)
        },
        egui::FontId::new(font_size, egui::FontFamily::Monospace),
        Color32::BLACK,
    );
    ui.painter().galley(text_pos, galley);
}
```

The show Method of RegVisualizer

In addition, the `show` method of the previously mentioned `RegVisualizer` is also an important part related to graphics display. Before understanding the implementation of this method, we need to first understand the members of the `RegVisualizer` structure. The definition of the `RegVisualizer` structure is as shown in Listing 4.22.

Listing 4.22: Definition of the `RegVisualizer` structure in PixelAssemblySIMD

```
pub struct RegVisualizer {
    // Visualization Data
    layout_data: HashMap<Register, Vec<Vec<(Pos2, Vec2)>>>,
    elements: HashMap<Register, Vec<Vec<Element>>>,
    // Animation Data
    animation_config: HashMap<Register, RegAnimationConfig>,
    animation_layout_data: HashMap<(Register, LayoutLocation), Vec<Vec<(Pos2, Vec2
    )>>>,
    animation_elements: HashMap<(Register, LayoutLocation), Vec<Vec<Element>>>,
    // Animation Sequence
    sender: Sender<AnimationControlMsg>,
    receiver: Receiver<AnimationControlMsg>,
    sequence: Option<Vec<Arc<Mutex<Vec<(Vec<ElementAnimationData>, bool)>>>>>,
    finish_sender: Sender<ElementAnimationFinishMsg>,
    finish_receiver: Receiver<ElementAnimationFinishMsg>,
    sequence_finished_callback: Option<Box<dyn FnOnce() + Send + 'static>>,
}
```

The `RegVisualizer` struct contains the data for graphical display and animation execution. From top to bottom, the graphical display data includes: layout data, element data; the animation execution data includes: animation configuration, animation layout data, animation element data; and the animation execution data includes: the sending end of the animation control message, the receiving end of the animation control message, animation sequence, the sending end of the animation end message, the receiving end of the animation end message, and the callback function after the animation sequence is finished.

After understanding the members of the `RegVisualizer` struct, let's take a look at the implementation of the `show` method. The implementation of this method is complex and includes multiple parts, mainly including: creating the layout of the elements, displaying the elements, and displaying the animation elements. Wherein the animation elements refer to the duplicate of the register in the visualization method mentioned in Chapter 3. The code for creating the layout part is shown in Listing 4.23.

Listing 4.23: Implementation of `show` in `RegVisualizer` structure - Create Layout

```
ui.vertical(|ui| {
    data.registers[0].iter().for_each(|reg| {
        // Create Element's Layout through automatic layout
        ui.vertical(|ui| {
            ui.label(get_reg_name(reg).clone());
            ui.spacing_mut().item_spacing.x = 0.0;
            let location = if let Some(config) = self.animation_config.get(reg) {
                config.location
            } else {
                LayoutLocation::None
            };
            let size = get_size_from_value(&values[0]);
            let repeat_number = if let Some(config) = self.animation_config.get(reg) {
                config.repeat_numbers
            } else {
                (0, 0)
            };
            // Elements Layout
            RegVisualizer::create_layout(ui, size, reg, values.len(), 1, &mut self.layout_data);
            RegVisualizer::create_elements(&values, reg, reg, &self.layout_data, &mut self.elements, true);
            // Animation Layout
            RegVisualizer::create_layout(ui, size, &(reg.clone(), LayoutLocation::TOP), values.len(), repeat_number.0, &mut self.animation_layout_data);
                RegVisualizer::create_elements(&values, &(reg.clone(), LayoutLocation::TOP), reg, &self.animation_layout_data, &mut self.animation_elements, false);
        });
    });
});
```

In this section, we first used the `vertical` method provided by egui to create a vertical layout, then used the `for_each` method to traverse each register. For each register, we first created a vertical layout and created a label in the layout to display the name of the register. Then, we used the `create_layout` method to create a layout. This method will create a layout based on the number of values in the register and the number of repetitions in the animation configuration, which includes the position information of each element. Then, we used the `create_elements` method to create each element, which will create each element based on the position information in the layout. Finally, we use the same method to create layouts and elements for animation elements.

The implementations of the `create_layout` and `create_elements` methods cited here are

shown in Listing 4.24 and Listing 4.25.

Listing 4.24: Implementation of `create_layout` in `RegVisualizer` structure

```
fn create_layout<T: Hash + Clone + Eq + PartialEq>(ui: &mut Ui, size: Vec2, key: &T
  , data_size: usize, repeat_number: usize, layout_data: &mut HashMap<T, Vec<Vec<(
  Pos2, Vec2)>>>) {
    ui.vertical(|ui| {
        let mut layout_vecs = vec![];
        (0..repeat_number).for_each(|_| {
            ui.horizontal(|ui| {
                let mut layout_vec = vec![];
                (0..data_size).for_each(|_| {
                    let (layout_rect, _response) = ui.allocate_exact_size(size,
  Sense::hover());
                    layout_vec.push((layout_rect.min, size));
                });
                layout_vecs.push(layout_vec);
            });
        });
        layout_data.insert(key.clone(), layout_vecs);
    });
}
```

Listing 4.25: Implementation of `create_elements` in `RegVisualizer` structure

```
fn create_elements<T: Hash + Clone + Eq + PartialEq>(values: &Vec<Value>, key: &T,
  reg: &Register, layout_data: &HashMap<T, Vec<Vec<(Pos2, Vec2)>>>, elements: &mut
  HashMap<T, Vec<Vec<Element>>>, display: bool) {
    if !elements.contains_key(key) || values_changed {
        if let Some(vecs) = layout_data.get(key) {
            let mut element_vecs = vec![];
            vecs.iter().for_each(|vec| {
                if vec.len() == values.len() {
                    let mut element_vec = vec![];
                    vec.iter().enumerate().for_each(|(index, (position, size))| {
                        element_vec.push(Element::default()
                            .with_display(display)
                            .with_value(values[index].clone())
                            .with_position(position.clone()));
                    });
                    element_vecs.push(element_vec);
                }
            });
            elements.insert(key.clone(), element_vecs);
        }
    }
```

```
20 }
```

The `create_layout` method creates a vertical layout, then creates a horizontal layout within this vertical layout. The number of horizontal layouts is determined by the repeat count in the animation configuration. It then creates rectangles in the horizontal layout, the size of which is determined by the value width of the data, and saves the position information of the rectangle in the layout data.

The `create_elements` method then creates elements according to the position information in the layout data. This method traverses each position information in the layout data, creates elements according to the position information, and finally saves the elements in the element data.

The next part of the `show` method in the `RegVisualizer` structure is to display the Elements and Animation Elements. The code for these two parts is very similar, so only the code for displaying the Elements part is shown. The code is shown in Listing 4.26.

Listing 4.26: Implementation of `show` in `RegVisualizer` structure - Show Elements

```
1  let low_layer_id = LayerId::new(Order::Middle, Id::new("
     register_visualizer_animation_elements_low"));
2  let middle_layer_id = LayerId::new(Order::Foreground, Id::new("
     register_visualizer_animation_elements_middle"));
3  let high_layer_id = LayerId::new(Order::Tooltip, Id::new("
     register_visualizer_animation_elements_high"));
4  let top_layer_id = LayerId::new(Order::Debug, Id::new("
     register_visualizer_animation_elements_top"));
5  self.elements.iter().for_each(|(_, vec)| {
6      vec.iter().for_each(|elements| {
7          elements.iter().for_each(|element| {
8              show_element!(ui, element, low_layer_id, middle_layer_id, high_layer_id
     , top_layer_id);
9          });
10     });
11 });
```

In these two sections, we first use the `LayerId` provided by egui to create four layers of different depths to simulate the different hierarchical relations in the animation. Then, we used the `show_element!` macro to display each element. The implementation of the `show_element!` macro is shown in Listing 4.27.

Listing 4.27: Implementation of `show_element!` in PixelAssemblySIMD

```
1  macro_rules! show_element {
2      ($ui:expr, $element:expr, $low_layer_id:expr, $middle_layer_id:expr,
     $high_layer_id:expr, $top_layer_id:expr) => {
3          if $element.display {
```

```
4               match $element.order {
5                   ElementOrder::Normal => {
6                       $element.show($ui);
7                   }
8                   ElementOrder::Low => {
9                       $ui.with_layer_id($low_layer_id, |ui| {
10                          $element.show(ui);
11                      });
12                  }
13                  ElementOrder::Middle => {
14                      $ui.with_layer_id($middle_layer_id, |ui| {
15                          $element.show(ui);
16                      });
17                  }
18                  ElementOrder::High => {
19                      $ui.with_layer_id($high_layer_id, |ui| {
20                          $element.show(ui);
21                      });
22                  }
23                  ElementOrder::Top => {
24                      $ui.with_layer_id($top_layer_id, |ui| {
25                          $element.show(ui);
26                      });
27                  }
28              }
29          }
30      };
31  }
```

This macro will select different layers according to the display order of the elements, and then call the element's `show` method to display the elements.

4.3 Animation Executor

4.3.1 The `update` Methods

The core of the animation executor includes the `update` method of `Element` and `RegVisualizer` and the control part of the animation sequence. Among them, the `update` method can set the position of the element according to the set speed and execution time every time it is executed, thus achieving the effect of animation, which is the main part of generating animation. The implementation of the `update` method of `Element` is shown in Listing 4.28.

Listing 4.28: Implementation of `update` in `Element` structure

```
fn update(&mut self, delta_time: f32, factor: f32, min_speed: f32, max_speed: f32)
  {
    let direction = self.target_position - self.position;
    let distance = direction.length();
    if distance > 1.0 {
        self.animating = true;
        let base_speed = distance * factor;
        let speed = base_speed.min(max_speed).max(min_speed);
        let normalized_direction = direction.normalized();
        self.position += normalized_direction * speed * delta_time;
    } else {
        self.position = self.target_position;
        self.animating = false;
        // Run callback
        if let Some(callback) = self.animation_finished_callback.take() {
            callback(self);
        }
    }
}
```

This method first calculates the distance between the current position and the target position. If the distance is greater than `1.0`, it will update the current position according to the set speed; otherwise, this means that the distance between the two is already close enough. At this time, the current position will be directly set to the target position, and the animation flag is set to `false`, indicating that the animation has been completed. After the animation is completed, the animation end callback function will be executed. The callback function is very important because it can be used to update some information of the `Element` or send an animation end signal after the animation is over. The callback function and the signal sending performed in it are important reasons why this animation executor can implement event-driven.

Listing 4.29: Implementation of `update` in `RegVisualizer` structure

```
pub fn update(&mut self, delta_time: f32, factor: f32, min_speed: f32, max_speed:
  f32) {
    self.elements.iter_mut().for_each(|(_, vec)| {
        vec.iter_mut().for_each(|elements| {
            elements.iter_mut().for_each(|element| {
                element.update(delta_time, factor, min_speed, max_speed);
            });
        });
    });
    self.animation_elements.iter_mut().for_each(|(_, vec)| {
```

```
10          vec.iter_mut().for_each(|elements| {
11              elements.iter_mut().for_each(|element| {
12                  element.update(delta_time, factor, min_speed, max_speed);
13              });
14          });
15      });
16  }
```

The implementation of the `update` method in the `RegVisualizer` structure is shown in Listing 4.29. The method is relatively simple, its basic purpose is to provide an external interface for the `update` method of `Element`. It traverses all elements and then calls each element's `update` method, thus updating the position of each element.

4.3.2 Animation Sequence Control

The control of the animation sequence is another important part of the animation executor. It is used to generate an animation sequence that can be played continuously, thus achieving continuous animation playback. The control part of the animation sequence mainly includes the creation, execution, and end detection of the animation sequence.

Creation of Animation Sequence

Listing 4.30: Creation of Animation Sequence

```
1  pub fn set_group_move_animation_sequence(&mut self, sequence: Arc<Mutex<Vec<(Vec<
     ElementAnimationData>, bool)>>>) {
2      self.sequence = Some(vec![sequence]);
3  }
4  pub fn start_move_animation_sequence(&self) {
5      if !self.sequence.is_none() {
6          self.sender.send(AnimationControlMsg::ExecuteAnimation(0)).unwrap();
7      }
8  }
```

The code for creating the animation sequence is shown in Listing 4.30. This code will be called before the animation sequence starts and is used to create the animation sequence. This code will save the animation sequence data in the animation sequence and then send a message to execute the animation sequence. The execution message of the animation sequence will be received in the execution part of the animation sequence, thereby starting the execution of the animation sequence.

Execution of Animation Sequence

Listing 4.31: Execution of Animation Sequence

```
pub fn move_animation_sequence(&mut self, ctx: &Context) {
    match self.receiver.try_recv() {
        // Receive Message
    }
}
```

The code for executing the animation sequence is shown in Listing 4.31. This code will be called in each render loop and it's used to execute the animation sequence, and it will receive the message from the animation control message receiver.

Listing 4.32: Execution of Animation Sequence - **ExecuteAnimation** Signal

```
Ok(AnimationControlMsg::ExecuteAnimation(index)) => {
    if self.sequence.is_none() {
        self.sender.send(AnimationControlMsg::Terminate).unwrap();
        return;
    }
    let sequence = self.sequence.as_ref().unwrap()[0].clone();
    let mut groups = sequence.lock().unwrap();
    let length = groups.len();
    let group = std::mem::take(&mut groups[index]);
    if group.0.is_empty() {
        if index + 1 < length {
            self.sender.send(AnimationControlMsg::ExecuteAnimation(index + 1)).
unwrap();
        } else {
            self.sender.send(AnimationControlMsg::Terminate).unwrap();
        }
    } else {
        let sender_clone = self.sender.clone();
        self.group_move_animation(group.0, group.1, move || {
            if index + 1 < length {
                sender_clone.send(AnimationControlMsg::ExecuteAnimation(index +
1)).unwrap();
            } else {
                sender_clone.send(AnimationControlMsg::Terminate).unwrap();
            }
        });
    }
}
```

When an **ExecuteAnimation** message is received as shown in Listing 4.32, the animation within the

animation sequence will be executed according to the information in the message, followed by a request to redraw the interface. **ExecuteAnimation** will receive the number of the animation sequence to be executed.

Listing 4.33: Execution of Animation Sequence - **Terminate** Signal

```
Ok(AnimationControlMsg::Terminate) => {
    if let Some(s) = self.sequence.as_deref_mut() {
        let mut s = s.to_vec();
        s.remove(0);
        self.sequence = Some(s.clone());
        if s.is_empty() {
            self.sequence = None;
            if let Some(callback) = self.sequence_finished_callback.take() {
                callback();
            }
        } else {
            self.sender.send(AnimationControlMsg::ExecuteAnimation(0)).unwrap();
            ctx.request_repaint();
        }
    } else {
        if let Some(callback) = self.sequence_finished_callback.take() {
            callback();
        }
    }
}
```

If a **Terminate** message is received, as shown in Listing 4.33, it will carry out the clean-up work according to the information in the message, and then request to redraw the interface. If no message is received, no command will be executed, as the animation may still be ongoing, and there is no need to switch the animation sequence. The `group_move_animation` method in the code is used to execute a group of simultaneous animations in the animation sequence. The implementation of this method is shown in Listing 4.34.

Listing 4.34: Execution of Animation Sequence - Animation Group

```
pub fn group_move_animation<F>(&mut self, data_vec: Vec<ElementAnimationData>,
  is_layout: bool, callback: F)
    where
        F: FnMut() + Send + 'static,
{
    let total_animations = data_vec.len();
    let completed_animations = Arc::new(Mutex::new(0));
    let shared_callback = Arc::new(Mutex::new(Some(callback)));
    for data in data_vec.into_iter() {
```

```
9           let completed_animations_clone = Arc::clone(&completed_animations);
10          let shared_callback_clone = Arc::clone(&shared_callback);
11          self.move_animation(data, is_layout, move || {
12              let mut callback = shared_callback_clone.lock().unwrap();
13              let mut completed_animations = completed_animations_clone.lock().unwrap
    ();
14              *completed_animations += 1;
15              if *completed_animations == total_animations {
16                  if let Some(mut callback) = callback.take() {
17                      callback();
18                  }
19              }
20          });
21      }
22  }
```

This method traverses each animation in the animation group and executes them. After each animation is completed, a callback function will be executed. The callback function will increment the number of completed animations by one. When all animations are completed, the callback function for the animation sequence will be executed, indicating that this group of animations has been completed. The `move_animation` method in the code is used to execute an individual animation in the animation group, and its implementation is as presented in Listing 4.35.

Listing 4.35: Execution of Animation Sequence - Individual Animation

```
1   pub fn move_animation<F>(&mut self, data: ElementAnimationData, is_layout: bool,
      callback: F)
2       where
3           F: FnOnce() + Send + 'static,
4   {
5       elements_vec[data.source.2][data.source.3].target_position = target_data.0;
6       if elements_vec[data.source.2][data.source.3].order ⩽ target_data.1 {
7           elements_vec[data.source.2][data.source.3].order = target_data.1.get_higher
    ();
8       }
9       let sender = self.finish_sender.clone();
10      if let Some(callback_in_data) = data.callback {
11          elements_vec[data.source.2][data.source.3].set_animation_finished_callback(
    move |element| {
12              callback_in_data(element);
13              callback();
14              sender.send(ElementAnimationFinishMsg::SetTarget(data.source, data.
    target)).unwrap();
15          });
16      } else {
```

```
17          elements_vec[data.source.2][data.source.3].set_animation_finished_callback
    (|_| {
18              callback();
19          });
20      }
21  }
```

This method will set the target position and display order of the animation into the elements, then set the callback function after the animation finishes according to the animation's configuration, and finally execute the animation. After the animation is completed, the callback function of the animation will be executed to execute the next animation.

Overall, the implementation of the animation sequence part is based on three parts: animation sequence (include several animation groups), animation group (include several individual animations), and individual animation. Among them, the difference of animation sequence and animation group is: the groups in the animation sequence must be executed one after another, but the individual animations in the animation group are performed simultaneously. The three of them realize the execution of the animation sequence through their nested relationship.

End Detection of Animation Sequence

Listing 4.36: End Detection of Animation Sequence

```
1  pub fn move_animation_finish(&mut self, ctx: &Context) {
2      loop {
3          match self.finish_receiver.try_recv() {
4              Ok(ElementAnimationFinishMsg::SetTarget(source, target)) => {
5                  self.set_target_for_move_animation_finish(source, target);
6                  ctx.request_repaint();
7              }
8              Err(TryRecvError::Empty) => {
9                  break;
10                 /* Do nothing */
11             }
12             Err(_) => {
13                 /* Error */
14             }
15         }
16     }
17 }
```

The code of end detection of animation sequence is as shown in Listing 4.36. This section of the code is called every rendering loop to check whether the animation sequence is finished. This part of the

code will receive messages from the receiver of the animation end message, if a message is received, it will execute the cleanup work after the end of the animation sequence according to the information in the message, and then request a redraw of the interface. If no message is received, it will break out of the loop, that is, the end of the animation sequence check.

4.4 Instruction Executor

According to the design in Chapter 3, executing instructions needs to go through four parts: creating the layout of the register copies required for the animation, executing the animation, updating CPU data, and deleting the copies and layout of the register. Therefore, the instruction executor is implemented based on a finite state machine to execute these four parts. The states of this finite state machine are as shown in Listing 4.37.

Listing 4.37: The status of FSM in Instruction Executor

```
pub enum AnimationFSMState {
    Idle,
    CreateLayout,
    RunAnimation,
    UpdateData,
    DestroyLayout,
}
impl AnimationFSMState {
    pub fn next(&mut self) {
        *self = match self {
            AnimationFSMState::Idle => AnimationFSMState::CreateLayout,
            AnimationFSMState::CreateLayout => AnimationFSMState::RunAnimation,
            AnimationFSMState::RunAnimation => AnimationFSMState::UpdateData,
            AnimationFSMState::UpdateData => AnimationFSMState::DestroyLayout,
            AnimationFSMState::DestroyLayout => AnimationFSMState::Idle,
        }
    }
}
```

The states of this status machine are sequentially from top to bottom: idle status (**Idle**), layout creation status (**CreateLayout**), animation execution status (**RunAnimation**), data update status (**UpdateData**), layout deletion status (**DestroyLayout**). Among them, the idle status is the initial state. When the instruction executor receives an instruction, it will start executing from the idle state. In each state, the instruction executor will perform the corresponding operations and then switch the state to the next one. After the instruction executor completes the layout deletion status, it will switch

the state back to the idle status, waiting for the arrival of the next instruction.

Listing 4.38: Implementation of the FSM in Instruction Executor

```
pub enum FSMCtrlMsg {
    ToIdle,
    Next,
}
pub struct AnimationFSM {
    state: AnimationFSMState,
    create_layout: Option<Box<dyn FnOnce(&mut Self) + Send + 'static>>,
    run_animation: Option<Box<dyn FnOnce(&mut Self) + Send + 'static>>,
    update_data: Option<Box<dyn FnOnce(&mut Self) + Send + 'static>>,
    destroy_layout: Option<Box<dyn FnOnce(&mut Self) + Send + 'static>>,
    pub sender: Sender<FSMCtrlMsg>,
    receiver: Receiver<FSMCtrlMsg>,
}
```

Listing 4.38 shows the implementation of the state machine. In this, the `FSMCtrlMsg` enum type is the event type for user control of the state machine states, and the `AnimationFSM` struct is used for the implementation of the state machine. This struct includes the state of the state machine, the function executed when creating the layout state, the function executed when running the animation state, the function executed when updating data state, the function executed when destroying the layout state, the sender of control messages and the receiver of control messages. Among them, receiver for the state machine control messages is used to receive control messages from the instruction executor (sender), which controls the state of the state machine.

Listing 4.39: Execution of the FSM in Instruction Executor

```
pub fn run(&mut self) {
    match self.state {
        AnimationFSMState::Idle => {}
        AnimationFSMState::CreateLayout => {
            if let Some(f) = self.create_layout.take() {
                f(self);
            }
        }
        AnimationFSMState::RunAnimation => {
            if let Some(f) = self.run_animation.take() {
                f(self);
            }
        }
        AnimationFSMState::UpdateData => {
            if let Some(f) = self.update_data.take() {
                f(self);
```

```
17              }
18          }
19          AnimationFSMState::DestroyLayout => {
20              if let Some(f) = self.destroy_layout.take() {
21                  f(self);
22              }
23          }
24      }
25      match self.receiver.try_recv() {
26          Ok(FSMCtrlMsg::ToIdle) => {
27              self.state = AnimationFSMState::Idle;
28              self.create_layout = None;
29              self.run_animation = None;
30              self.update_data = None;
31              self.destroy_layout = None;
32          }
33          Ok(FSMCtrlMsg::Next) => {
34              self.state.next();
35          }
36          Err(_) => {}
37      }
38  }
```

Listing 4.39 shows the execution of the state machine. When the state machine is executed, the corresponding function will be first executed according to the state of the state machine. Then, the control message reception end of the state machine will be checked. If a control message is received, the state of the state machine will be switched according to the type of the control message. Among them, `FSMCtrlMsg::ToIdle` is used to switch the state machine to the idle state, and `FSMCtrlMsg::Next` is used to switch the state machine to the next state.

Based on this finite state machine, instructions can be executed in a fixed order. For example, when the instruction executor receives an instruction, it will switch the state machine to the layout creation state, then execute the function of the layout creation state, which will create the layout of the register copy. Then it will switch the state machine to the animation execution state, then execute the function of the animation execution state, which will execute the animation. Then, it will switch the state machine to the data update state, then execute the function of the data update state, which will update the CPU data, then switch the state machine to the layout deletion state, then execute the function of the layout deletion state, which will delete the layout of the register copy, and finally switch the state machine to the idle state, waiting for the arrival of the next instruction. This finite state machine sets a custom function for each state instead of executing through a fixed program, which makes the function of the

instruction executor more flexible. For example, users can implement different animation effects or implement more operations in a single step through custom functions.

5

Evaluation

This chapter evaluates the visualization method and prototype program proposed earlier. This chapter first discusses the difficulties of evaluating visualization methods, and then proposes to evaluate the visualization method proposed in this thesis through user research based on the schemes in the references. The second section records the evaluation results. The last section provides some SIMD algorithm example questions implemented in assembly language, which can be used to verify the effectiveness of the implemented functionality in Chapter 4.

5.1 Evaluation Methods

5.1.1 The Challenges of Evaluating Visualization Methods

It is generally believed that new technologies and applications must be formally verified to be considered a significant contribution. Therefore, visualization researchers also need to find objective indicators that can show the value of their methods[31]. However, the evaluation of visualization methods is considered challenging, mainly for the following reasons:

- *There is no effective general quantitative evaluation standard*: unlike tests in other fields such as software performance, the evaluation of visualization methods is often subjective, without effective quantitative evaluation standards. Different visualization methods focus on different points, and the schemes often have diversity, which makes it difficult for others to repeat the evaluation of visualization methods and to find a common evaluation method that can evaluate various visualization methods.
- *User diversity*: Different users may have different background knowledge, skills, preferences, and needs. This means that a visualization method may be very effective for some users, but not so effective, or even counterproductive, for others.

- *User subjectivity*: The evaluation of visualization methods is often subjective, which means that the evaluation results may be influenced by the evaluator's subjectivity. This subjectivity may cause the inaccuracy or even unfairness of the evaluation results.

Current ways to evaluate the effectiveness of visualization methods usually rely on user research[31]. These methods are not only not general quantitative standards, the methods themselves are easily affected by respondent bias, respondent diversity; and the difficulty of selection and implementation of the sample during the evaluation usually brings difficulties to the evaluation.
Unfortunately, although five key challenges in empirical visualization research have been proposed and possible methods to solve these issues have been suggested in C. Ziemkiewicz, M. Chen, D. H. Laidlaw, B. Preim, and D. Weiskopf. Open challenges in empirical visualization research. *Foundations of Data Visualization*:243–252, 2020, the implementation of these methods is still very challenging. Moreover, user research is still the main approach for evaluating visualization methods and is an important means of translating laboratory research into practical applications[32].
User research usually involves various techniques, from informal survey questionnaires, to crowdfunded user research, to stringent laboratory research with a small number of participants. However, in many situations, relying solely on laboratory research methods that collect evaluation results from a small number of participants may result in a lack of statistical reliability[32]. Therefore, due to the lack of conditions for professional crowdfunded user research, this thesis ultimately adopted a user research method based on a wide range of survey questionnaires to evaluate the visualization method proposed in this thesis.

5.1.2 Evaluation Method Employed in this Thesis

This thesis employs empirical methods, specifically surveys, to collect information and conduct social research on the following aspects:

1. *Basic situation*: the survey questionnaire did not collect sensitive personal information of the respondents, including name, gender, age, etc. The questionnaire only collected information such as the level of understanding of the computer field, experience with SIMD programming, and experience with visualization tools.
2. *Understanding of advanced programming languages*: using a C language as an example, the survey questionnaire assessed respondents' recognition of the C language code. A C language code was shown to respondents, their understanding of the C language was determined by asking

whether they had a clear understanding of the values of specific variables in the program. The C language programs shown ranged from simple to complex, including:

- Scalar variable operations and assignment statements;
- Vector operations and assignment statements using SIMD intrinsics;

3. *Understanding of assembly language*: The survey questionnaire showed respondents a piece of assembly language code, and determined their understanding of the assembly language by asking whether they had a clear understanding of the values of specific registers in the program. This part of the test was essentially just testing the user's understanding of the display of a single SIMD instruction in assembly language, and did not contain a complete algorithm.
4. *Testing static visualization method*: the survey questionnaire showed respondents a static visualization method (i.e., picture) with the same semantics as SIMD instructions in assembly language, to test whether this scheme could help users understand SIMD instructions, deepen their understanding of SIMD instructions, or help users who already understand SIMD computations understand SIMD instructions faster.
5. *Testing dynamic visualization method*: the survey questionnaire showed respondents a dynamic visualization method (i.e., animation) with the same semantics as SIMD instructions in assembly language, to test whether this scheme could help users understand SIMD instructions, deepen their understanding of SIMD instructions, or help users who already understand SIMD computations understand SIMD instructions faster.
6. *Summary*: The final part of the survey questionnaire directly asked respondents about their views on SIMD computations and visualizations, including:
    - Respondents' understanding of the difficulty of SIMD computations versus scalar computations;
    - Whether respondents received assistance from the visualization method when understanding code;
    - Respondents' preference for static and dynamic visualization methods in different situations;
    - Respondents' preference for pure code versus visualization methods when understanding code;

5.2 Evaluation Result

Before analyzing the results of the survey questionnaire, it must be noted that this test has limitations. As mentioned in the previous section, compared to general quantitative evaluations in other fields, user research-based evaluations of visualization methods are relatively limited and easily influenced by user subjectivity. There are the following obvious limitations in this evaluation:

- The number of respondents participating in this survey questionnaire is relatively small compared to large-scale Crowdsourced user research, as only 97 people participated in the completion of this survey questionnaire, which may result in a lack of statistical reliability in the results;
- The vast majority of respondents participating in this survey questionnaire are from Asia, which may lead to regional bias in the survey results;
- This survey did not collect information such as the age and gender of the respondents, but the differences in age and gender could potentially impact human cognition to some extent, which may limit the results of the survey;
- This survey primarily targets specific SIMD instructions in assembly language. Even though we have selected ones with computation and data permuting, two typical types of SIMD instructions, it still cannot represent all SIMD instructions, which could potentially constrain the survey results.

Therefore, the results of this survey may not be universally applicable, but they still can, to a certain extent, work as preliminary evaluation for the visualization method proposed in this thesis.

5.2.1 Implementation of the Survey Questionnaire

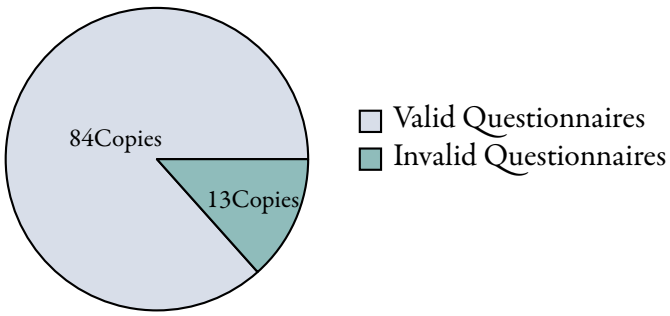
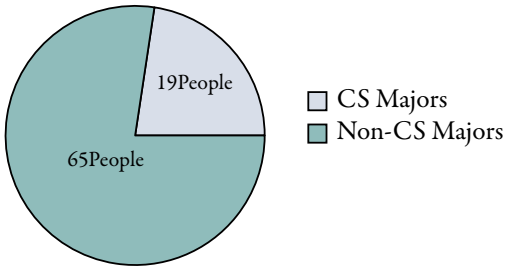


Figure 5.1: Implementation of the Survey Questionnaire

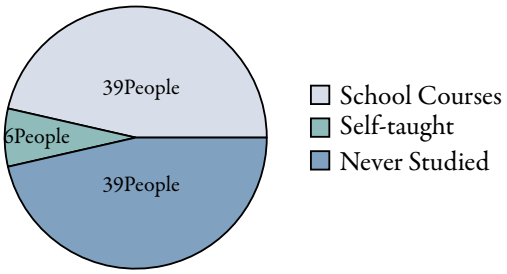
As shown in Figure 5.1, a total of 97 questionnaires were collected in this survey. Among them, 84

were valid and 13 were invalid. The recovery rate of valid questionnaires was 86.6%. The criteria for excluding invalid questionnaires were as follows: respondents showed obvious logical errors in their answers to consecutive questions. For example, in the assessment of code cognition, they answered "can understand," but later indicated "unable to understand the code" in subsequent responses.

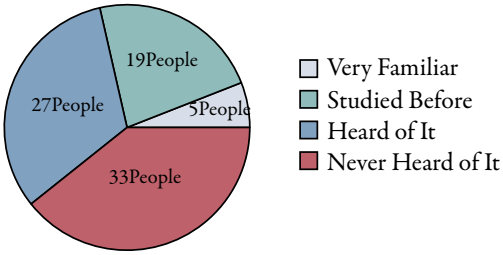
5.2.2 Basic Information of Survey Respondents



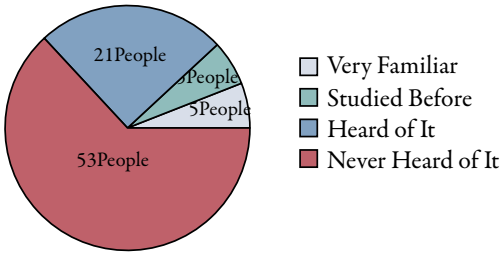
(a) Proportion of Respondents with Computer Science Education



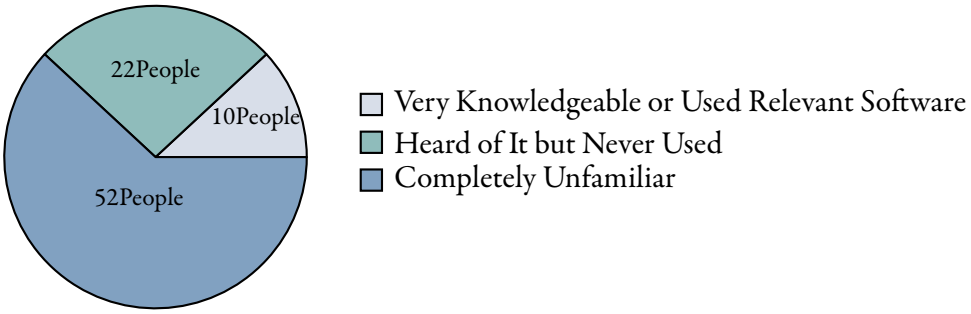
(b) Proportion of Respondents Who Have Learned Programming



(c) Proportion of Respondents Familiar with Assembly Language



(d) Proportion of Respondents Familiar with SIMD



(e) Proportion of Respondents Familiar with Visualization or Who Have Used Related Software

Figure 5.2: Basic Information of Survey Respondents

As shown in Figure 5.2, the basic information of the survey respondents is presented. Figure 5.2a shows the proportion of respondents who have received computer science education, which includes majors such as Computer Science and Technology, Software Engineering, Information Security, etc. It

can be seen that 19.6% of the respondents have received computer science education, meaning that the majority of the respondents have not received such education. Figure 5.2b shows the proportion of respondents who have learned programming, including those who have attended school courses or taught themselves programming. It can be seen that 53.6% of the respondents have learned programming, indicating that more than half of the participants have received some form of programming education. This suggests that within the scope of the survey, even among non-computer science students, a considerable number have studied programming.

Figure 5.2c and Figure 5.2d show the proportion of respondents familiar with assembly language and SIMD computing. It can be seen that 28.6% are familiar with assembly language and 11.9% are familiar with SIMD computing, indicating that the vast majority of respondents are not familiar with these more advanced or challenging technologies. Even among those with formal computer science education, the familiarity is not high.

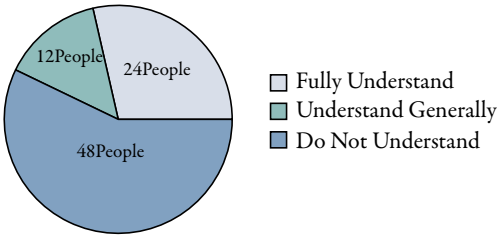
Finally, Figure 5.2e shows the proportion of respondents familiar with visualization or who have used related software. It can be seen that 38.1% have some understanding of visualization, with only 11.9% being very knowledgeable or having used relevant visualization software. This suggests that most of the respondents were unfamiliar with visualization prior to the survey, and even fewer had used related software. This indicates that the prevalence of visualization within the surveyed group is not high.

5.2.3 Survey Respondents' Understanding of Advanced Programming Languages

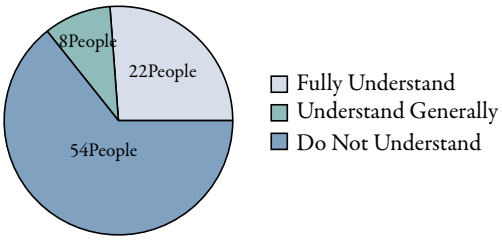
Figure 5.3 shows the level of understanding of advanced programming languages among the survey respondents. Figure 5.3a displays the proportion of respondents who understand ordinary scalar computations in C language, with 28.6% fully understanding, 14.3% understanding the general meaning, and 57.1% not understanding at all. It is evident that only a small proportion, 28.6%, fully understand ordinary scalar computations in C language.

Figure 5.3b shows the proportion of respondents who understand complex scalar computations in C language. Here, 26.2% fully understand, 9.5% understand the general meaning, and 64.3% do not understand at all. Again, a relatively small proportion, 26.2%, fully understand complex scalar computations in C language.

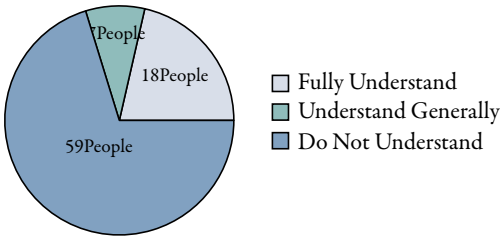
Figure 5.3c presents the proportion of respondents who understand SIMD computations in C language, with 21.4% fully understanding, 8.3% understanding the general meaning, and 73.8% not understanding at all. This indicates that only 21.4% of respondents have a full understanding of SIMD



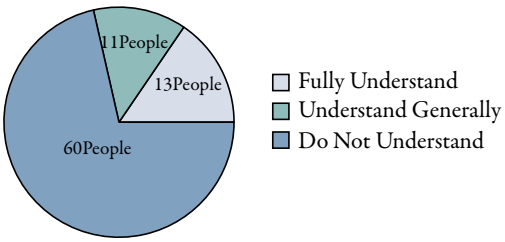
(a) Proportion of Respondents Who Understand Ordinary Scalar Computations in C Language



(b) Proportion of Respondents Who Understand Complex Scalar Computations in C Language



(c) Proportion of Respondents Who Understand SIMD Computations in C Language



(d) Proportion of Respondents Who Understand SIMD Permuting in C Language

Figure 5.3: Survey Respondents' Understanding of Advanced Programming Languages

computations in C language.

Finally, Figure 5.3d shows the proportion of respondents who understand SIMD permuting in C language. Here, 15.4% fully understand, 13.1% understand the general meaning, and 71.4% do not understand at all. This demonstrates that only a small fraction, 15.4%, fully understand SIMD permuting in C language.

According to the above statistics, as shown in Figure 5.4, it can be observed that as the difficulty of the code increases (from ordinary scalar calculations to SIMD permuting, with the level of code abstraction gradually increasing, thus considered more challenging to understand), the number of people who can understand the code behavior decreases. This confirms the problem raised in Chapter 1: SIMD instruction code is indeed more difficult to understand compared to ordinary scalar operations.

In the following statistics, we will analyze the situations of both professional and non-professional programmers separately. Here, individuals who have received professional computer science education are considered "professional programmers," while those who have not received computer science education but possess certain programming skills are considered "non-professional programmers." In the following statistics, "fully understand" and "understand the general meaning" are collectively referred to as "understand."

Figure 5.5 displays the understanding level of professional programmers regarding advanced program-

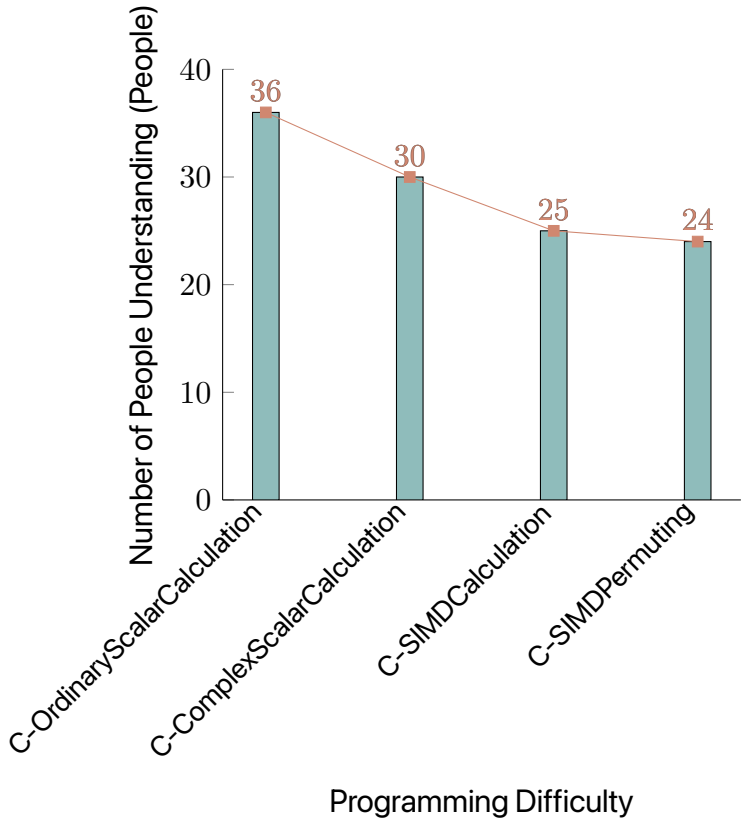
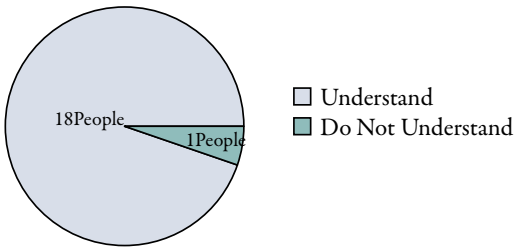
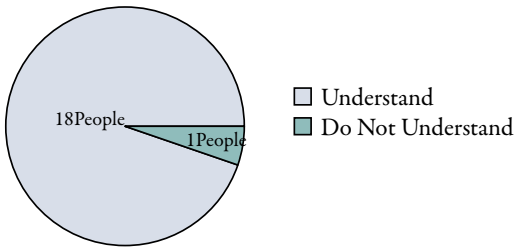


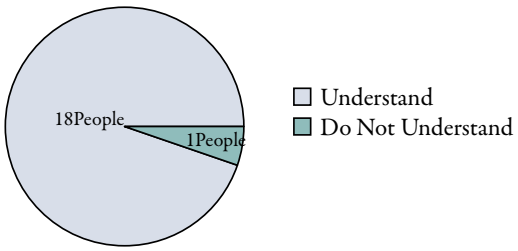
Figure 5.4: Influence of Programming Difficulty on Respondents' Understanding of Advanced Programming Languages



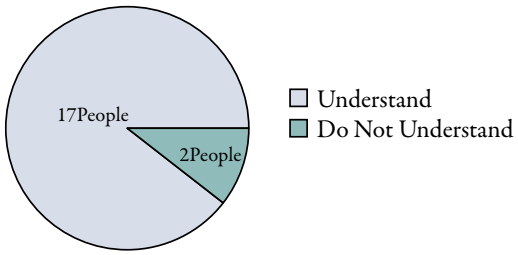
(a) Proportion of Professional Programmers Who Understand Ordinary Scalar Computations in C Language



(b) Proportion of Professional Programmers Who Understand Complex Scalar Computations in C Language



(c) Proportion of Professional Programmers Who Understand SIMD Computations in C Language



(d) Proportion of Professional Programmers Who Understand SIMD Permuting in C Language

Figure 5.5: Understanding of Advanced Programming Languages by Professional Programmers

ming languages. According to Figure 5.5a, Figure 5.5b, and Figure 5.5c, the proportion of professional programmers who understand ordinary scalar, complex scalar, and SIMD computations in C language reaches 94.7%; according to Figure 5.5d, the proportion of professional programmers who understand the more complex SIMD permuting in C language is also as high as 89.5%.

The data from Figure 5.6 show that, after receiving professional computer science education, the vast

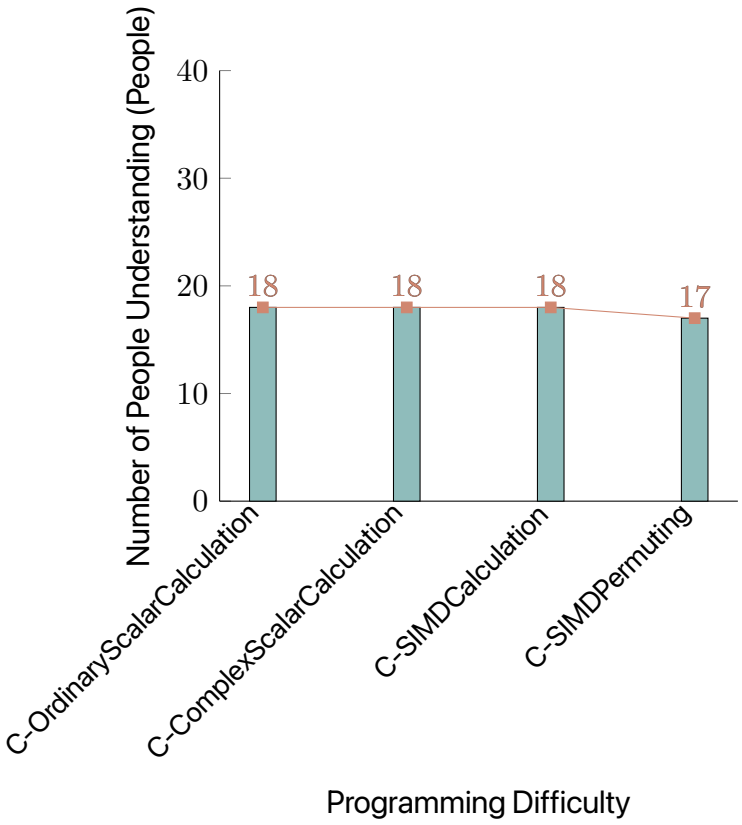
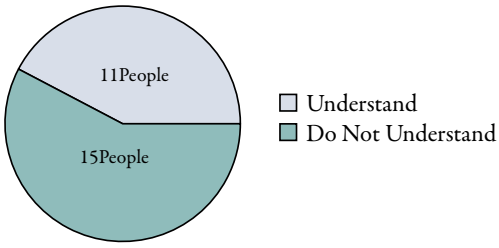


Figure 5.6: Influence of Programming Difficulty on the Understanding of Advanced Programming Languages by Surveyed Professional Programmers

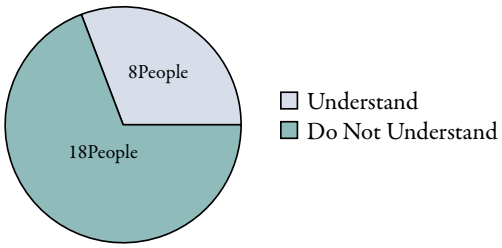
majority of students can understand the use of advanced programming languages, and the difficulty of the code does not significantly affect their understanding of C language code.

Figure 5.7 illustrates the understanding level of non-professional programmers regarding advanced programming languages. According to the data from Figure 5.7a, Figure 5.7b, Figure 5.7c, and Figure 5.7d, combined with Figure 5.8, it is evident that as the difficulty increases, the number of non-professional programmers who can understand C language code behavior decreases. Moreover, the majority of them have a weak ability to understand C language code behavior.

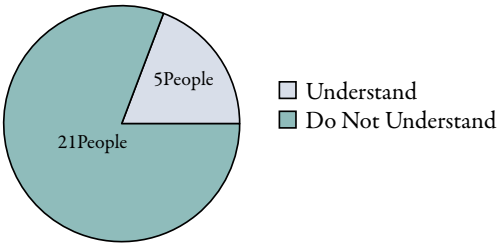
According to the data from Figure 5.9a, Figure 5.9b, Figure 5.9c, and Figure 5.9d, a very small portion of people who have not studied programming at all can still understand C language code through semantics. As indicated by Figure 5.10, the number of people who can understand code behavior gen-



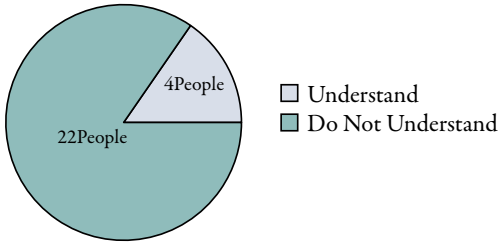
(a) Proportion of Non-Professional Programmers Who Understand Ordinary Scalar Computations in C Language



(b) Proportion of Non-Professional Programmers Who Understand Complex Scalar Computations in C Language



(c) Proportion of Non-Professional Programmers Who Understand SIMD Computations in C Language



(d) Proportion of Non-Professional Programmers Who Understand SIMD Permuting in C Language

Figure 5.7: Understanding of Advanced Programming Languages by Non-Professional Programmers

erally follows the trend that fewer people understand as the code difficulty increases.

Based on the above analysis, one point worth noting is that for individuals who have received computer science education, their ability to understand C language code is generally not affected by code difficulty. However, for those who have not received computer science education, even if they have studied programming or possess some programming ability, their understanding of C language code is generally affected by code difficulty. This might suggest that they are more accustomed to reading source code and know how to extract information from it than others. Additionally, they may be able to use reference materials (like documentation) more effectively to aid their understanding of the code.

5.2.4 Survey Respondents' Understanding of Assembly Language

According to the results of Figure 5.11, for both questions regarding the understanding of assembly language, nearly 80% of respondents are unable to understand assembly language code. This percentage is significantly higher than those who cannot understand C language code, confirming that understanding assembly language is indeed more difficult than understanding high-level languages.

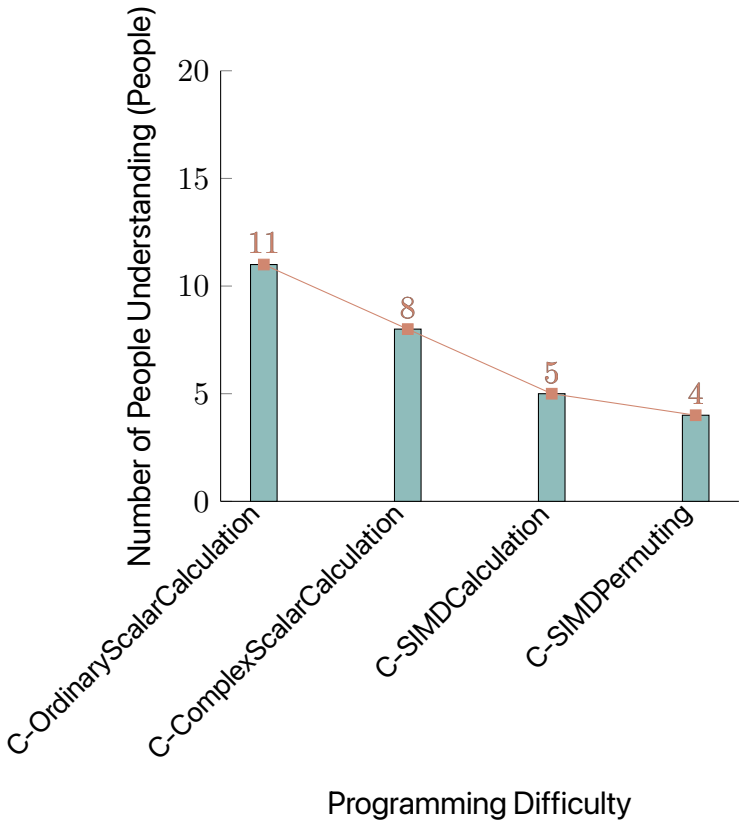


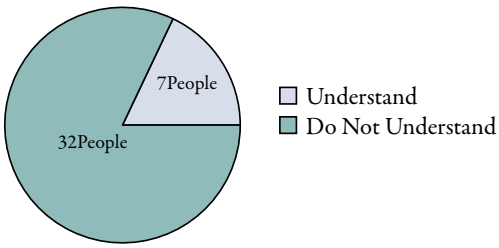
Figure 5.8: Influence of Programming Difficulty on the Understanding of Advanced Programming Languages by Surveyed Non-Professional Programmers

According to the results of Figure 5.12, it is evident that professional programmers, those who have received computer science education, generally have a high level of understanding of assembly language. However, this is still lower compared to the number who understand C language.

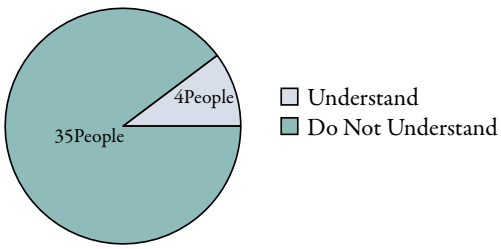
For non-professional programmers, understanding assembly language is very challenging. This is evident from Figure 5.13, where, among all surveyed non-professional programmers, only one person was able to understand assembly language code for both questions.

According to Figure 5.14, for people who have not studied programming, their situation is similar to that of non-professional programmers, with almost no one being able to understand assembly language code.

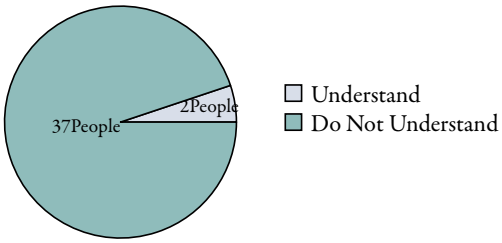
Based on these results, it is not difficult to see that understanding assembly language is more challenging compared to understanding C language code. This confirms the issue raised in Chapter 1: assembly language code is indeed more difficult for people to understand compared to higher-level languages. However, for those who have received computer science education, their ability to understand assembly language is still at a relatively high level, although it is lower compared to the number who understand C language.



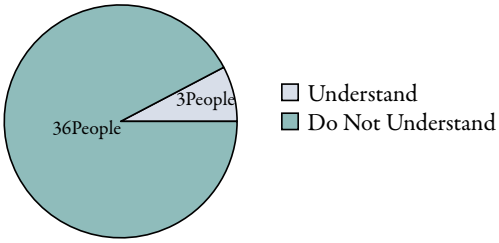
(a) Proportion of People Who Have Not Studied Programming and Understand Ordinary Scalar Computations in C Language



(b) Proportion of People Who Have Not Studied Programming and Understand Complex Scalar Computations in C Language



(c) Proportion of People Who Have Not Studied Programming and Understand SIMD Computations in C Language



(d) Proportion of People Who Have Not Studied Programming and Understand SIMD Permuting in C Language

Figure 5.9: Understanding of Advanced Programming Languages by People Who Have Not Studied Programming

5.2.5 Survey Respondents' Understanding of Static Visualization Methods

Help of Static Visualization in Understanding SIMD Computations in Assembly Language

Figure 5.15 shows the level of understanding among survey respondents regarding static visualization methods for SIMD computations. It can be seen that 54.8% of respondents are able to understand static visualization, indicating that a majority of the surveyed individuals can comprehend these visualization methods.

Figure 5.16a indicates that 43.3% of respondents could not understand the code when reading it, but could comprehend it through static visualization. This suggests that static visualization indeed helps some people in understanding the code. Furthermore, Figure 5.16b shows that the vast majority believe static visualization is beneficial in understanding SIMD computations in assembly language.

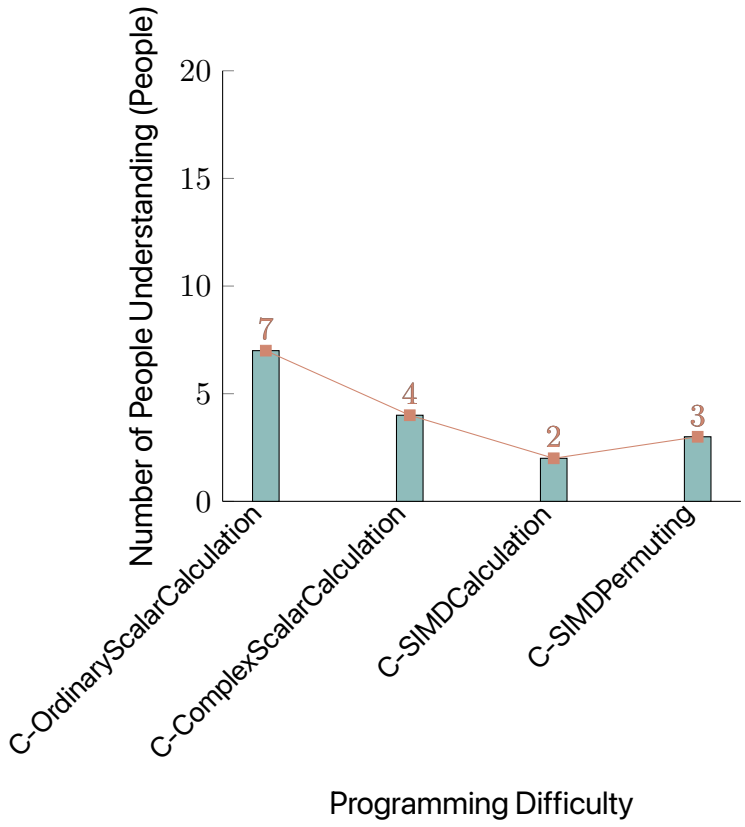
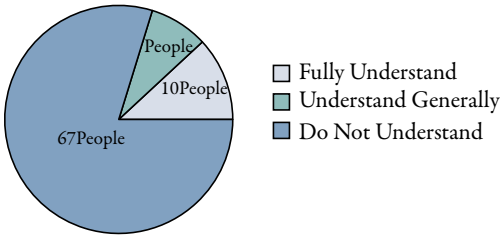
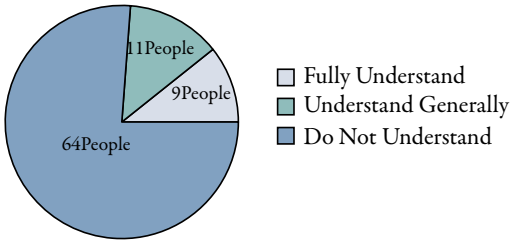


Figure 5.10: Influence of Programming Difficulty on the Understanding of Advanced Programming Languages by People Who Have Not Studied Programming



(a) Proportion of Respondents Who Understand SIMD Computations in Assembly Language

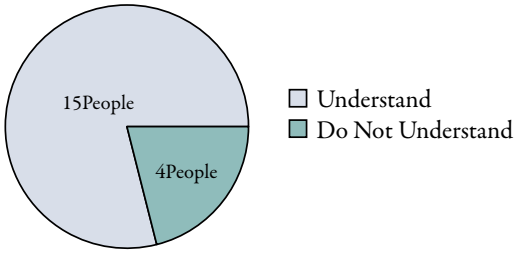


(b) Proportion of Respondents Who Understand SIMD Permuting in Assembly Language

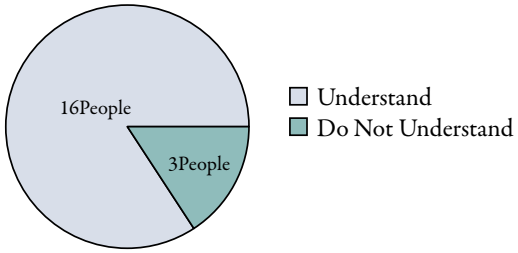
Figure 5.11: Survey Respondents' Understanding of Assembly Language

Help of Static Visualization in Understanding SIMD Permuting in Assembly Language

Figure 5.17 demonstrates the understanding level of survey respondents regarding static visualization methods for SIMD permuting. It can be seen that 48.8% of respondents can understand static visualization, which is lower than the proportion for simpler SIMD computations. This indicates that the effectiveness of static visualization methods diminishes for more complex code like SIMD permut-

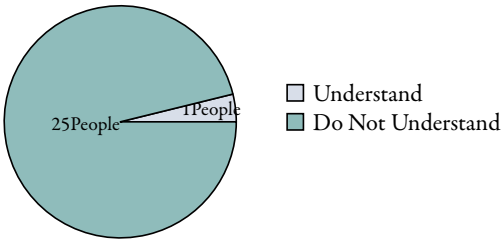


(a) Proportion of Professional Programmers Who Understand SIMD Computations in Assembly Language

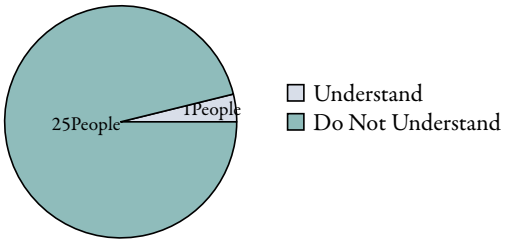


(b) Proportion of Professional Programmers Who Understand SIMD Permuting in Assembly Language

Figure 5.12: Understanding of Assembly Language by Professional Programmers

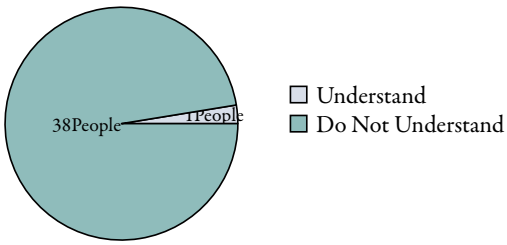


(a) Proportion of Non-Professional Programmers Who Understand SIMD Computations in Assembly Language

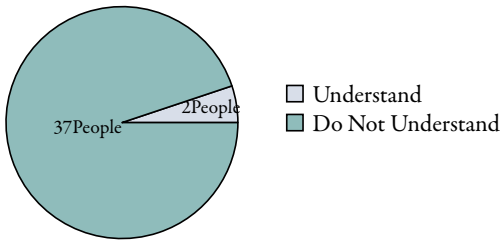


(b) Proportion of Non-Professional Programmers Who Understand SIMD Permuting in Assembly Language

Figure 5.13: Understanding of Assembly Language by Non-Professional Programmers



(a) Proportion of People Who Have Not Studied Programming and Understand SIMD Computations in Assembly Language



(b) Proportion of People Who Have Not Studied Programming and Understand SIMD Permuting in Assembly Language

Figure 5.14: Understanding of Assembly Language by People Who Have Not Studied Programming

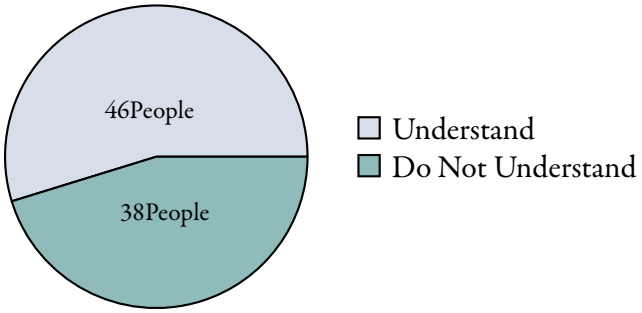
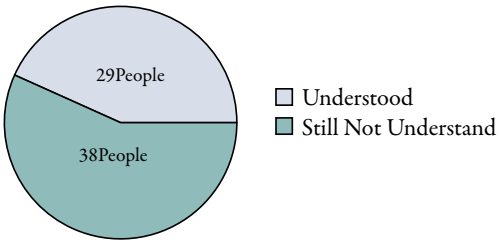
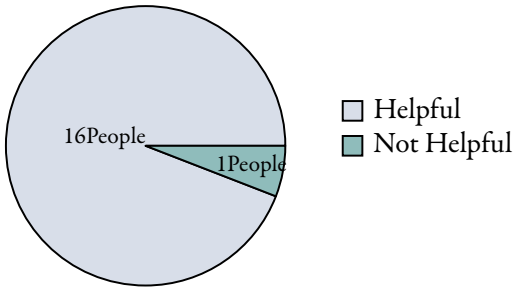


Figure 5.15: Proportion of Respondents Who Can Understand Static Visualization of SIMD Computations



(a) Proportion of Respondents Who Did Not Understand the Program by Reading Code, But Understood SIMD Computations Through Static Visualization



(b) Proportion of Respondents Who Understood the Program by Reading Code, and Found Static Visualization Helpful in Understanding SIMD Computations

Figure 5.16: Help of Static Visualization in Understanding SIMD Computations

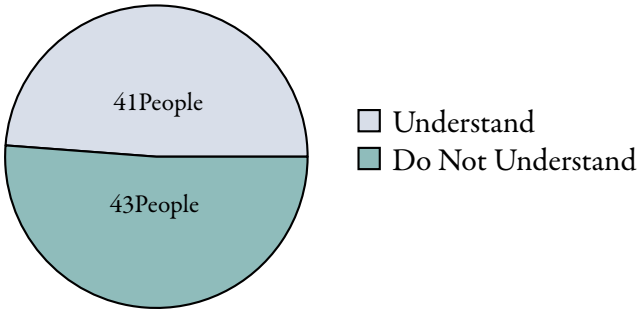
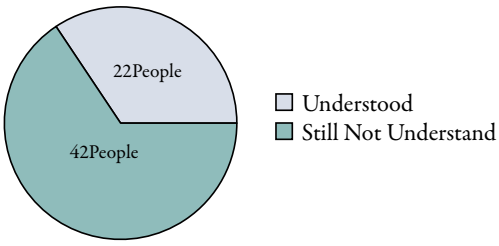


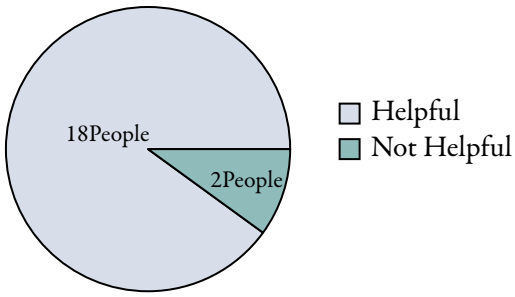
Figure 5.17: Proportion of Respondents Who Can Understand Static Visualization of SIMD Permuting

ing.

Figure 5.18a reveals that 34.4% of respondents were unable to understand the meaning of the code



(a) Proportion of Respondents Who Did Not Understand the Program by Reading Code, But Understood SIMD Permuting Through Static Visualization



(b) Proportion of Respondents Who Understood the Program by Reading Code, and Found Static Visualization Helpful in Understanding SIMD Permuting

Figure 5.18: Help of Static Visualization in Understanding SIMD Permuting

when reading it, but could comprehend it through static visualization. This implies that static visualization methods can indeed assist some individuals in understanding the code, although the proportion of people it helps is slightly lower for more complex SIMD permuting compared to simpler SIMD computations. Additionally, Figure 5.18b shows that the vast majority believe static visualization is helpful in understanding SIMD permuting in assembly language.

5.2.6 Survey Respondents' Understanding of Dynamic Visualization Methods

Help of Dynamic Visualization in Understanding SIMD Computations in Assembly Language

Figure 5.19 shows the level of understanding among survey respondents regarding dynamic visualization methods for SIMD computations. It can be seen that 47.6% of respondents are able to understand dynamic visualization, indicating that a significant portion of the surveyed individuals can comprehend these visualization methods. However, the number of respondents who understand dynamic visualization for SIMD computations is fewer than those who understand static visualization.

Figure 5.20a indicates that 34.3% of respondents could not understand the meaning of the code when reading it, but could comprehend it through dynamic visualization. This suggests that dynamic visualization methods do help some people understand the code, although the proportion of people it assists

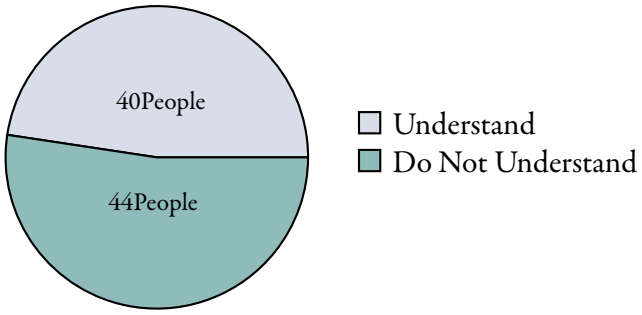
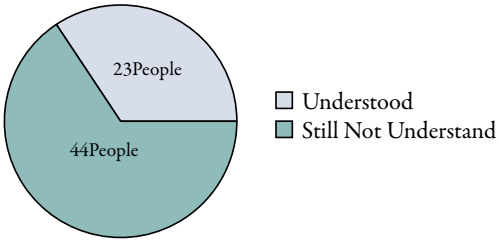
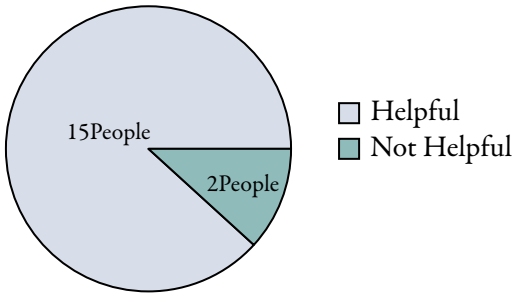


Figure 5.19: Proportion of Respondents Who Can Understand Dynamic Visualization of SIMD Computations



(a) Proportion of Respondents Who Did Not Understand the Program by Reading Code, But Understood SIMD Computations Through Dynamic Visualization



(b) Proportion of Respondents Who Understood the Program by Reading Code, and Found Dynamic Visualization Helpful in Understanding SIMD Computations

Figure 5.20: Help of Dynamic Visualization in Understanding SIMD Computations

is slightly lower compared to static visualization. Additionally, Figure 5.20b shows that the majority believe dynamic visualization is beneficial in understanding SIMD computations in assembly language, but this proportion is still lower compared to static visualization.

Help of Dynamic Visualization in Understanding SIMD Permuting in Assembly Language

Figure 5.21 demonstrates the level of understanding among survey respondents regarding dynamic visualization methods for SIMD permuting. It can be seen that 56.0% of respondents can understand dynamic visualization, suggesting that a majority of the surveyed individuals can comprehend these methods. Interestingly, the number of respondents who understand dynamic visualization for SIMD permuting is higher than those for SIMD computations; dynamic visualization seems to be more effective for more complex SIMD permuting instructions compared to static visualization.

Figure 5.22a indicates that 42.2% of respondents could not understand the meaning of the code when

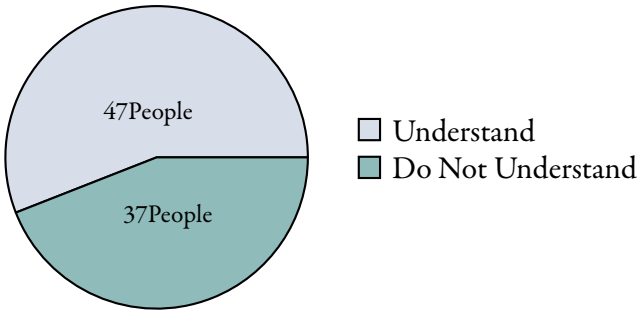
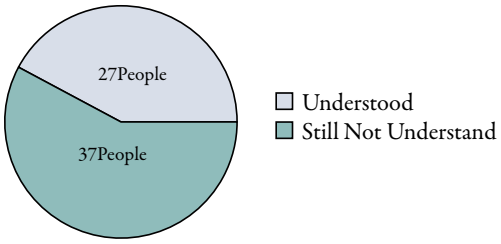
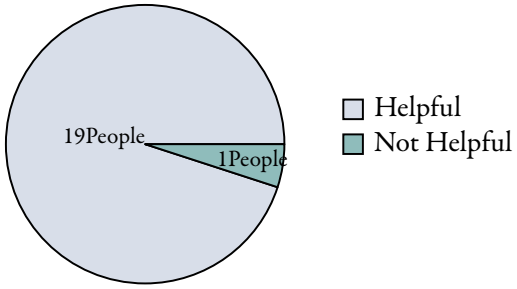


Figure 5.21: Proportion of Respondents Who Can Understand Dynamic Visualization of SIMD Permuting



(a) Proportion of Respondents Who Did Not Understand the Program by Reading Code, But Understood SIMD Permuting Through Dynamic Visualization

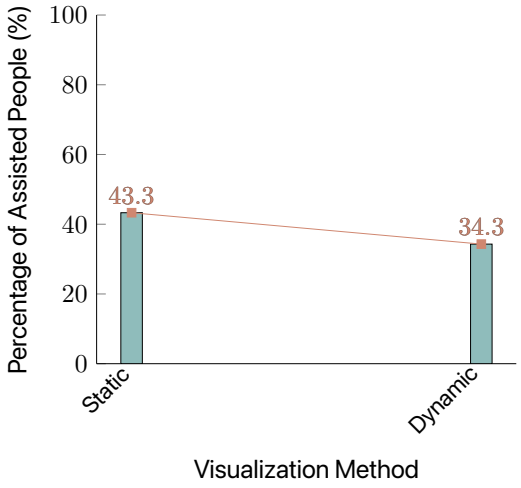


(b) Proportion of Respondents Who Understood the Program by Reading Code, and Found Dynamic Visualization Helpful in Understanding SIMD Permuting

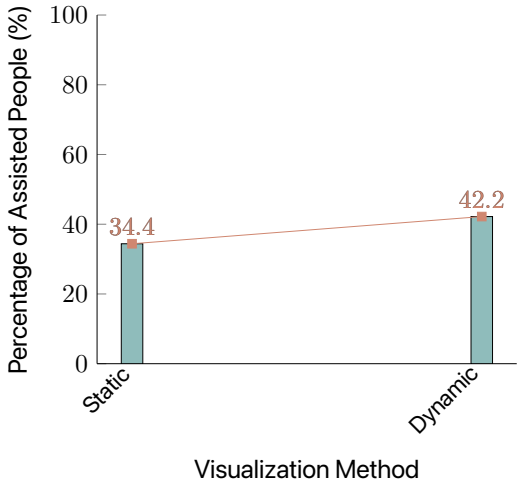
Figure 5.22: Help of Dynamic Visualization in Understanding SIMD Permuting

reading it, but could comprehend it through dynamic visualization. This suggests that dynamic visualization methods can indeed assist some people in understanding the code, and the proportion of assistance is higher for more complex SIMD permuting compared to static visualization. Furthermore, Figure 5.22b shows that the majority believe dynamic visualization is beneficial in understanding SIMD permuting in assembly language, and this proportion is greater than those who find static visualization helpful.

According to the analysis, for simpler SIMD computations, static visualization appears to be more helpful in understanding the code; however, for more complex SIMD permuting instructions, dynamic visualization offers greater assistance. Figure 5.23 visually presents this conclusion more clearly.



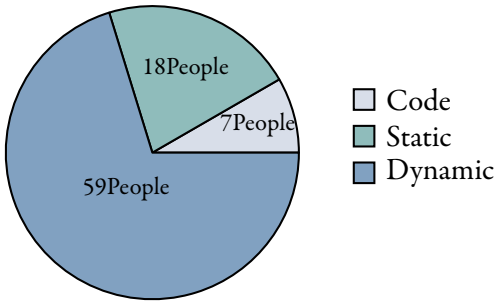
(a) Comparison of Static and Dynamic Visualization for SIMD Computation Programs



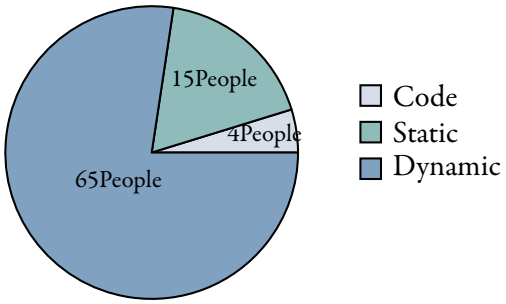
(b) Comparison of Static and Dynamic Visualization for SIMD Permuting Programs

Figure 5.23: The Degree of Assistance Provided by Static and Dynamic Visualization in Understanding Different Levels of Code Complexity

5.2.7 Survey Respondents' Preferences for Visualization Methods



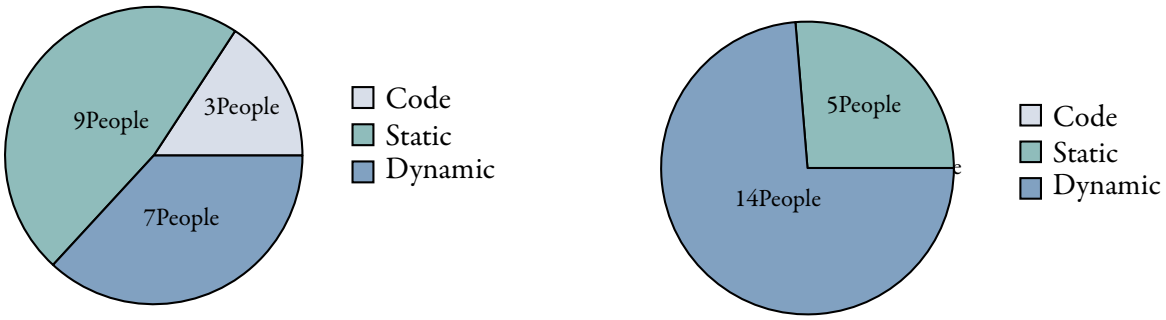
(a) Preferences of All Respondents for Simple Code



(b) Preferences of All Respondents for Complex Code

Figure 5.24: Preferences of All Respondents for Visualization Methods

According to the results of Figure 5.24, overall, respondents tend to use visualization methods to aid their understanding of code. For simple code, 70.2% of respondents prefer dynamic visualization methods, while for complex code, this proportion rises to 77.4%. However, for those with a computer science education, a greater number prefer using code and static visualization for understanding simple code, while the preference for complex code aligns with the general trend favoring dynamic visualization. This conclusion can be drawn from Figure 5.25.



(a) Preferences of Professional Programmers for Simple Code

(b) Preferences of Professional Programmers for Complex Code

Figure 5.25: Preferences of Professional Programmers for Visualization Methods

5.2.8 Conclusion

Overall, based on the results of this survey, visualization can assist people in understanding the meaning of code to a certain extent. However, the effectiveness of static and dynamic visualizations varies with the complexity of the code. For relatively simple code, static visualization is more helpful than dynamic visualization; for more complex code, dynamic visualization is more effective than static visualization. The dynamic visualization method used in the survey corresponds to the method proposed in Chapter 3. Thus, it can be concluded that the visualization approach presented in this thesis can help people understand the meaning of code to a certain extent, and this approach is particularly effective in assisting the understanding of complex SIMD instructions, while for simple instructions, it is not as intuitive as static visualization.

Additionally, individuals with computer science education generally have a higher ability to understand code. For those without a computer science education, even if they have studied programming or possess some programming skills, their overall ability to understand code is on par with the general public. This supports the view mentioned in the literature B. Helmlinger, M. Sommer, M. Feldhammer-Kahr, G. Wood, M. E. Arendasy, and S. E. Kober. Programming experience associated with neural efficiency during figural reasoning. *Scientific Reports*, 10(1):13351, 2020, which states that "people with high programming experience may develop a unique programming mindset distinct from others."

5.3 Supplementary Evaluation

The supplementary evaluation provided a small group of interviewees with two versions of software, "Prototype1" and "Prototype2", allowing them to thoroughly experience and evaluate these two pro-

totype versions and their visualization methods. Due to the distribution of software copies, this survey was conducted on a small scale, so the results may not be a statistically significant evaluation. Moreover, the participants in this survey were predominantly Asian, so regional preferences might affect the accuracy of the final results. Despite these limitations, the supplementary evaluation still reflects the interviewees' opinions on "Prototype1" and "Prototype2" as well as the two different visualization methods to a certain extent, and the results still hold some reference value.

Similar to the previous evaluation, the participants in this survey included students majoring in computer science, students who have studied programming, and laypeople who are not familiar with programming. This survey differed from the above questionnaire in that it used a matrix multiplication algorithm as an example, allowing users to actually use the tool to execute the algorithm process.

Figure 5.26 shows the overall results of the supplementary evaluation.It can be seen that among the

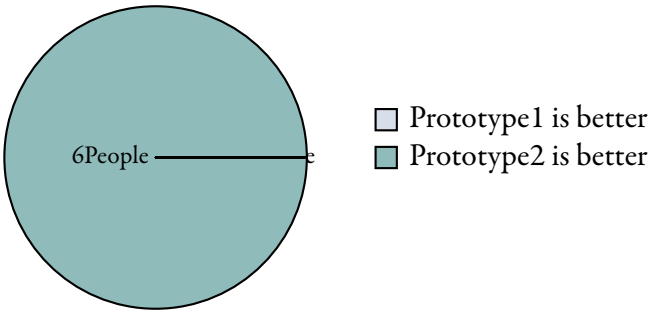


Figure 5.26: Supplementary Evaluation: Overall

6 participants in the survey, all of them considered Prototype2 to be generally superior. Based on the feedback, they identified the following advantages of Prototype2 over Prototype1:

- Prototype2 can display complete computational results, which is important for understanding SIMD (Single Instruction, Multiple Data) operation instructions.
- Prototype2 offers adjustable animation speeds, allowing users to quickly skip instructions that do not require understanding or to slow down for instructions that need careful observation.
- Prototype2 provides clear guidance during animation execution, enabling users to see the starting and ending points of movements.
- Prototype2 does not use an excessive amount of colors, making it more user-friendly for those with color vision deficiencies.
- Prototype2 supports more features than Prototype1, such as loops, memory access, and writing, etc.

However, in comparing the GUIs, half of the participants preferred the GUI of Prototype1, as shown

in Figure 5.27. They found that having the visualization window on one side was more intuitive for users, and that the window in Prototype2 was somewhat cumbersome.

Additionally, based on user feedback, there are still areas for improvement in Prototype2 in the following aspects:

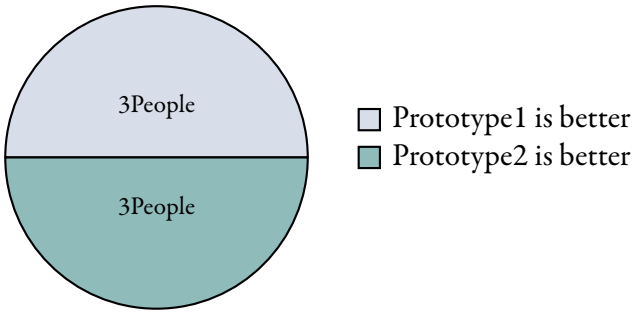


Figure 5.27: Supplementary Evaluation: GUI

- Replace the window mode with a more intuitive left-right layout, which is more favored by beginners.
- Include options for Undo and Skip, allowing users to bypass animations directly without waiting for them to complete when not needed.
- Consider using more contrasting colors in the color scheme to differentiate between different sections.

5.4 Examples

In addition to evaluating with surveys, this thesis provides some examples to verify the effectiveness of the proposed visualization scheme. This section will provide some examples of SIMD algorithms implemented in assembly language.

5.4.1 Prefix Sum

Algorithm

Listing 5.1: Prefix Sum algorithm (`_start` Part)

```
1  section .data
2  array dd 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
3  result dd 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
4
5  section .text
```

```
6  global _start
7  _start:
8      vmovdqu32 zmm0, [array]
9      call _prefix_sum
10     vmovdqu32 [result], zmm0
11     ; exit
12     xor rdi, rdi
13     mov rax, 60
14     syscall
```

Listing 5.2: Prefix Sum Algorithm (Algorithm Part)

```
1  _prefix_sum: ; arg: zmm0
2      vpxorq zmm2, zmm2, zmm2
3      valignd zmm1, zmm0, zmm2, 15
4      vpaddd zmm0, zmm0, zmm1
5      valignd zmm1, zmm0, zmm2, 14
6      vpaddd zmm0, zmm0, zmm1
7      valignd zmm1, zmm0, zmm2, 12
8      vpaddd zmm0, zmm0, zmm1
9      valignd zmm1, zmm0, zmm2, 8
10     vpaddd zmm0, zmm0, zmm1
11     ret
```

The code in Listing 5.2 shows the assembly language version of the Prefix Sum algorithm mentioned in W. Zhang, Y. Wang, and K. A. Ross. Parallel prefix sum with simd, 2023. arXiv: 2312.14874 [cs.DC]. Using the `_start` function and the data provided in Listing 5.1, you can run the function for testing.

Visualization Results of PixelAssemblySIMD

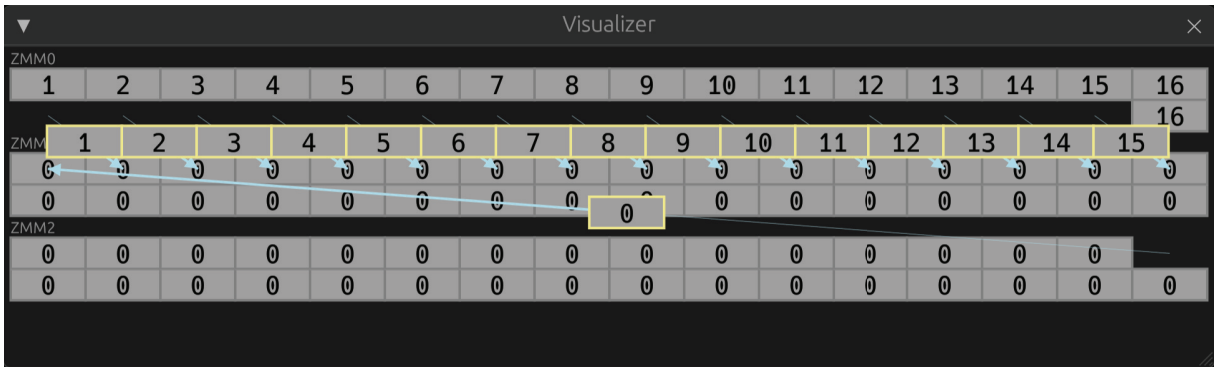


Figure 5.28: Visualization process of `valignd zmm1, zmm0, zmm2, 15`

Figure 5.28 shows the process of visualizing the instruction `valignd zmm1, zmm0, zmm2,`

15 using the PixelAssemblySIMD tool. This instruction moves the last 32-bit integer of ZMM2 to the first 32-bit integer of ZMM1, and then moves the first to fifteenth 32-bit integers of ZMM0 to the second to sixteenth 32-bit integers of ZMM1. The image shows PixelAssemblySIMD moving the corresponding numbers from ZMM2 and ZMM0 to ZMM1, with the elements in motion highlighted by a yellow border to alert the user that these elements are being animated; additionally, PixelAssemblySIMD draws a sky-blue arrow from the starting point to the endpoint of the animation, to remind the user of the starting and ending positions of the elements. For clarity, the line of the arrow becomes lighter at the positions that have already been moved, allowing the user to clearly see the main content behind while still understanding the starting and ending points of the animation.

Figure 5.29 shows the final visualization results of the Prefix Sum algorithm.

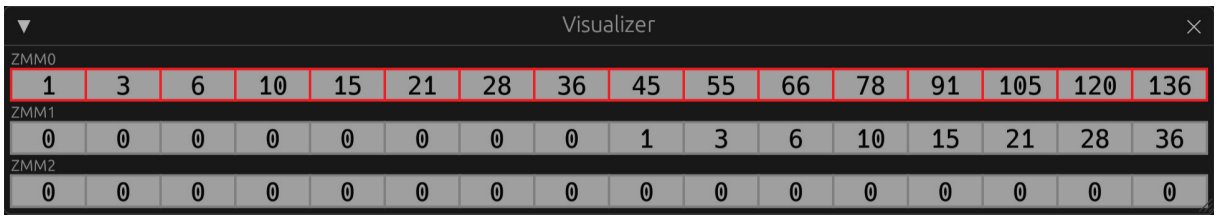


Figure 5.29: Visualization results of the Prefix Sum algorithm

5.4.2 Matrix Transpose

Algorithm

Listing 5.3: Matrix Transpose Algorithm (Data Declaration)

```
1  section .data
2  matrix dd 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0
3       dd 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0
4       dd 17.0, 18.0, 19.0, 20.0, 21.0, 22.0, 23.0, 24.0
5       dd 25.0, 26.0, 27.0, 28.0, 29.0, 30.0, 31.0, 32.0
6       dd 33.0, 34.0, 35.0, 36.0, 37.0, 38.0, 39.0, 40.0
7       dd 41.0, 42.0, 43.0, 44.0, 45.0, 46.0, 47.0, 48.0
8       dd 49.0, 50.0, 51.0, 52.0, 53.0, 54.0, 55.0, 56.0
9       dd 57.0, 58.0, 59.0, 60.0, 61.0, 62.0, 63.0, 64.0
10 result dd 0, 0, 0, 0, 0, 0, 0, 0
11      dd 0, 0, 0, 0, 0, 0, 0, 0
12      dd 0, 0, 0, 0, 0, 0, 0, 0
13      dd 0, 0, 0, 0, 0, 0, 0, 0
14      dd 0, 0, 0, 0, 0, 0, 0, 0
15      dd 0, 0, 0, 0, 0, 0, 0, 0
16      dd 0, 0, 0, 0, 0, 0, 0, 0
```

```
17      dd 0, 0, 0, 0, 0, 0, 0, 0
```

Listing 5.4: Matrix Transpose Algorithm (`_start` Part)

```
1  section .text
2  global _start
3  _start:
4      vmovaps ymm0, [matrix]
5      vmovaps ymm1, [matrix + 0x20]
6      vmovaps ymm2, [matrix + 0x40]
7      vmovaps ymm3, [matrix + 0x60]
8      vmovaps ymm4, [matrix + 0x80]
9      vmovaps ymm5, [matrix + 0xA0]
10     vmovaps ymm6, [matrix + 0xC0]
11     vmovaps ymm7, [matrix + 0xE0]
12     call _matrix_transpose
13     vmovaps [result], ymm8
14     vmovaps [result + 0x20], ymm9
15     vmovaps [result + 0x40], ymm12
16     vmovaps [result + 0x60], ymm13
17     vmovaps [result + 0x80], ymm10
18     vmovaps [result + 0xA0], ymm11
19     vmovaps [result + 0xC0], ymm14
20     vmovaps [result + 0xE0], ymm15
21     ; exit
22     xor rdi, rdi
23     mov rax, 60
24     syscall
```

Listing 5.5: Matrix Transpose Algorithm (Algorithm Part)

```
1  _matrix_transpose:
2      vunpcklps ymm8, ymm0, ymm1
3      vunpcklps ymm9, ymm2, ymm3
4      vunpcklps ymm10, ymm4, ymm5
5      vunpcklps ymm11, ymm6, ymm7
6      vunpckhps ymm12, ymm0, ymm1
7      vunpckhps ymm13, ymm2, ymm3
8      vunpckhps ymm14, ymm4, ymm5
9      vunpckhps ymm15, ymm6, ymm7
10     vshufps ymm0, ymm8, ymm9, 0b01000100
11     vshufps ymm1, ymm8, ymm9, 0b11101110
12     vshufps ymm2, ymm10, ymm11, 0b01000100
13     vshufps ymm3, ymm10, ymm11, 0b11101110
14     vshufps ymm4, ymm12, ymm13, 0b01000100
15     vshufps ymm5, ymm12, ymm13, 0b11101110
```

```
16      vshufps ymm6, ymm14, ymm15, 0b01000100
17      vshufps ymm7, ymm14, ymm15, 0b11101110
18      vperm2f128 ymm8, ymm0, ymm2, 0x20
19      vperm2f128 ymm9, ymm1, ymm3, 0x20
20      vperm2f128 ymm10, ymm0, ymm2, 0x31
21      vperm2f128 ymm11, ymm1, ymm3, 0x31
22      vperm2f128 ymm12, ymm4, ymm6, 0x20
23      vperm2f128 ymm13, ymm5, ymm7, 0x20
24      vperm2f128 ymm14, ymm4, ymm6, 0x31
25      vperm2f128 ymm15, ymm5, ymm7, 0x31
26      ret
```

An assembly language implementation of the Matrix Transpose algorithm is provided in Listing 5.5. You can run the program to complete the test using the `_start` function given in Listing 5.4 and data given in Listing 5.3.

Visualization Results of PixelAssemblySIMD

Figure 5.30 shows the final visualization results of the Matrix Transpose algorithm. The final results are stored in `YMM8-YMM15`. In the Matrix Transpose algorithm, all are shift instructions similar to the `valignd` instruction, therefore, the execution process will not be elaborated further.

5.4.3 Matrix Multiplication

Algorithm

Listing 5.6: The Matrix Multiplication algorithm

```
1  section .data
2  MatrixA dq 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14
     .0, 15.0, 16.0
3  MatrixB dq 16.0, 15.0, 14.0, 13.0, 12.0, 11.0, 10.0, 9.0, 8.0, 7.0, 6.0, 5.0, 4.0,
     3.0, 2.0, 1.0
4  MatrixC dq 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0
     .0, 0.0
5
6  section .text
7  global _start
8  _start:
9       vmovapd ymm0, [MatrixB]
10      vmovapd ymm1, [MatrixB + 32]
11      vmovapd ymm2, [MatrixB + 64]
12      vmovapd ymm3, [MatrixB + 96]
```

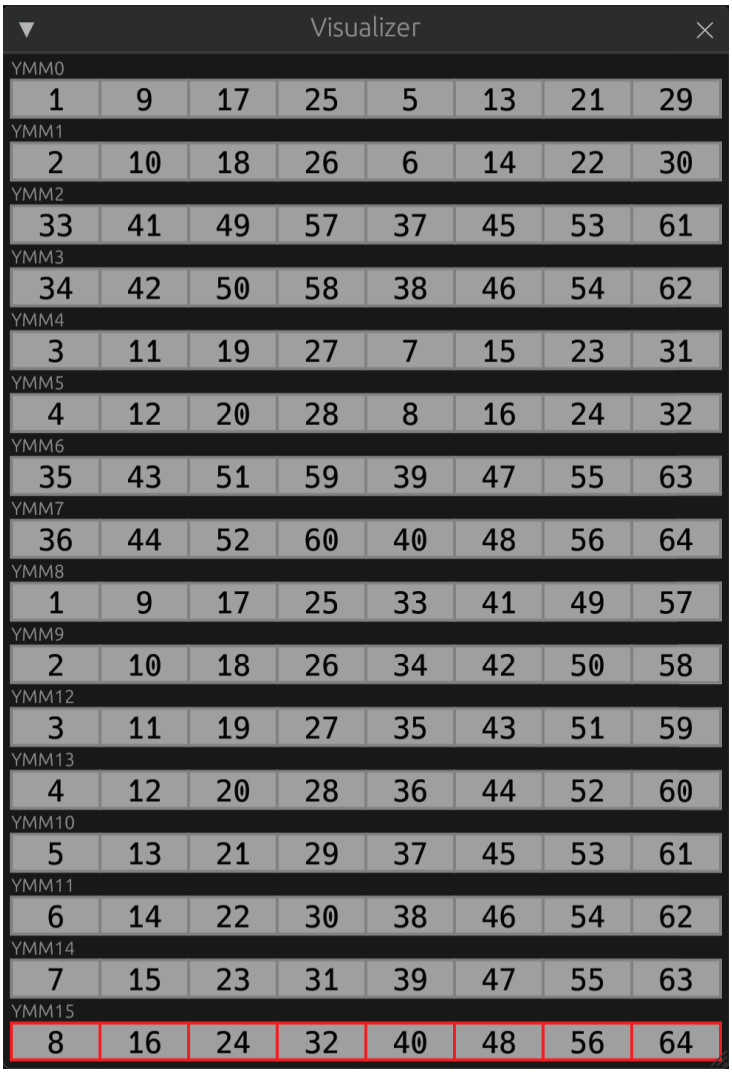


Figure 5.30: Visualization results of the Matrix Transpose algorithm

```
13      mov rdi, 0
14  __loop:
15      vmovapd ymm15, [MatrixA + rdi]
16      vextractf128 xmm14, ymm15, 0
17      vbroadcastsd ymm4, xmm14
18      shufpd xmm14, xmm14, 1
19      vbroadcastsd ymm5, xmm14
20      vextractf128 xmm14, ymm15, 1
21      vbroadcastsd ymm6, xmm14
22      shufpd xmm14, xmm14, 1
23      vbroadcastsd ymm7, xmm14
24      vmulpd ymm7, ymm3, ymm7
25      vfmadd213pd ymm6, ymm2, ymm7
26      vfmadd213pd ymm5, ymm1, ymm6
27      vfmadd213pd ymm4, ymm0, ymm5
28      vmovapd [MatrixC + rdi], ymm4
```

```
29      add rdi, 32
30      cmp rdi, 128
31      jne __loop
32      ; exit
33      xor rdi, rdi
34      mov rax, 60
35      syscall
```

Listing 5.6 is an implementation of the Matrix Multiplication algorithm in assembly language.

Visualization Results of PixelAssemblySIMD

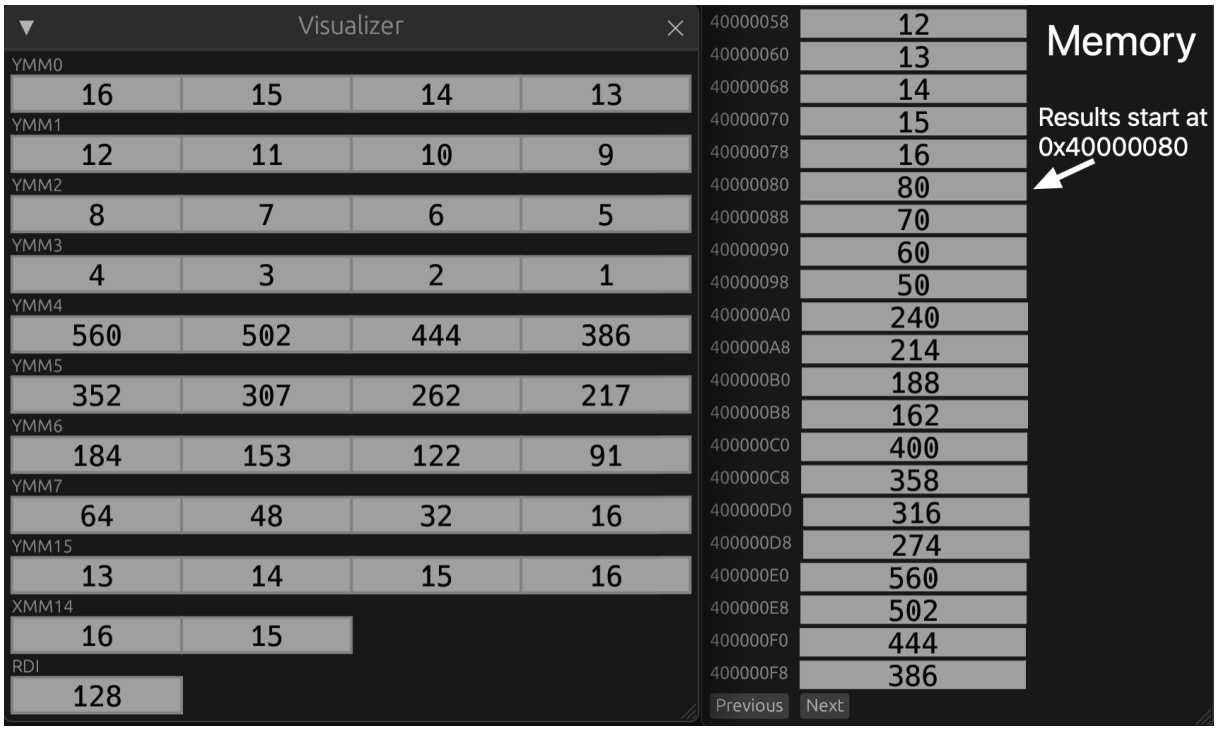


Figure 5.31: Visualization results of the Matrix Multiplication algorithm

Figure 5.31 shows the final visualization results of the Matrix Multiplication algorithm. Since the algorithm ultimately stores the result in memory, the diagram also shows the visualization of the memory, with the final result starting from memory address `0×40000080`. The Matrix Multiplication algorithm utilizes more complex computational instructions such as `vfmadd213pd`. Figure 5.32 shows the computational process of `vfmadd213pd`. It can be seen that PixelAssemblySIMD demonstrates how the final result is computed through a specific formula, which is very helpful for users to understand SIMD computational instructions.

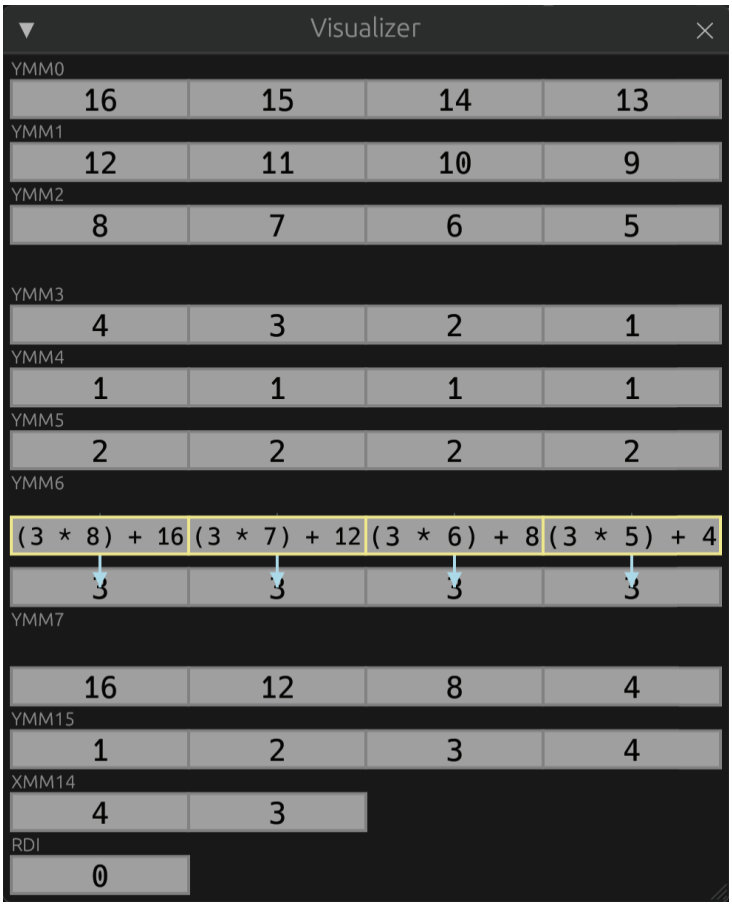


Figure 5.32: Visualization process of `vfmadd213pd ymm6, ymm2, ymm7`

6

Comparison with Related Research

This chapter mainly states related studies and compares the method proposed in this thesis with them. Although there are many common visualization software at present, compared with them, there are relatively few visualization tools specifically for SIMD. Some existing SIMD visualization tools are listed in this chapter, they are briefly introduced and compared with the visualization method and implementation proposed in this thesis.

6.1 SIMDGiraffe

6.1.1 Project Overview

SIMDGiraffe[33] is a project for visualizing SIMD instructions. The aim of the project is to make SIMD code understandable through various visualization methods so that anyone with basic understanding of algebra can quickly and stress-free understand any given SIMD instruction or its intrinsic functions. The basic features of SIMDGiraffe include capturing expert explanations, semantic SIMD visualization, graphical visualization, and explanation of SIMD instructions/intrinsic functions.
This tool works according to the view chosen by the user (beginner's view or expert view), and most operations are done by clicking the mouse. The expert view is for explaining vector instructions, and the beginner's view is for looking at the traces left by experts while explaining through the expert view. SIMDGiraffe offers a model of describing vector code behavior on a target vector architecture. This model can describe the behavior of vector code as a path from the entry point of the function to the exit point. The path can isolate independent or weakly related code blocks, where independent code blocks can run independent of the rest of the function, and weakly related code blocks can execute within the function, but their return values will be read by the function. The model also defines a relationship that connects register read and write operations. Through this model, users can segment vector code

by setting a starting point and an endpoint and establish an interactive correspondence between source code and visualization representation[6].

6.1.2 Comparison

The differences are:

- SIMDGiraffe visualizes the code through its code behavior model, but this model is evidently based on the control flow of the program. PixelAssemblySIMD abstracts instructions into different degrees of data transfer, which is a visualization based on data flow.
- SIMDGiraffe does not support newer SIMD instruction sets, such as AVX-512. But PixelAssemblySIMD has already supported these instruction sets.
- The usage of the SIMDGiraffe tool does not provide clear guidance, it is hard for users to use it like PixelAssemblySIMD without prior learning.

6.2 SIMD-Visualiser

6.2.1 Project Overview

Listing 6.1: Example Code for SIMD-Visualiser: Prefix Sum Algorithm

```
#include <x86intrin.h>

__m128i PrefixSum(__m128i curr) {
    __m128i Add = _mm_slli_si128(curr, 4);
    curr = _mm_add_epi32(curr, Add);
    Add = _mm_slli_si128(curr, 8);
    return _mm_add_epi32(curr, Add);
}
```

SIMD-Visualiser[34] is a tool for graphically displaying SIMD code, with its user interface consisting of a code editor on the left half and a visualization window on the right half. SIMD-Visualiser provides an example code for the Prefix Sum algorithm, as shown in Listing 6.1. The visualization process of executing this code is illustrated in Figure 6.1. This visualization tool aims to help humans understand SIMD code designed for machines. Through the use of animation, color, and graphics, SIMD-Visualiser aims to simplify and reveal SIMD code, making it easier for people with basic computer science knowledge to understand.

The main features of SIMD-Visualiser include graphical visualisation, the ability to handle Abstract

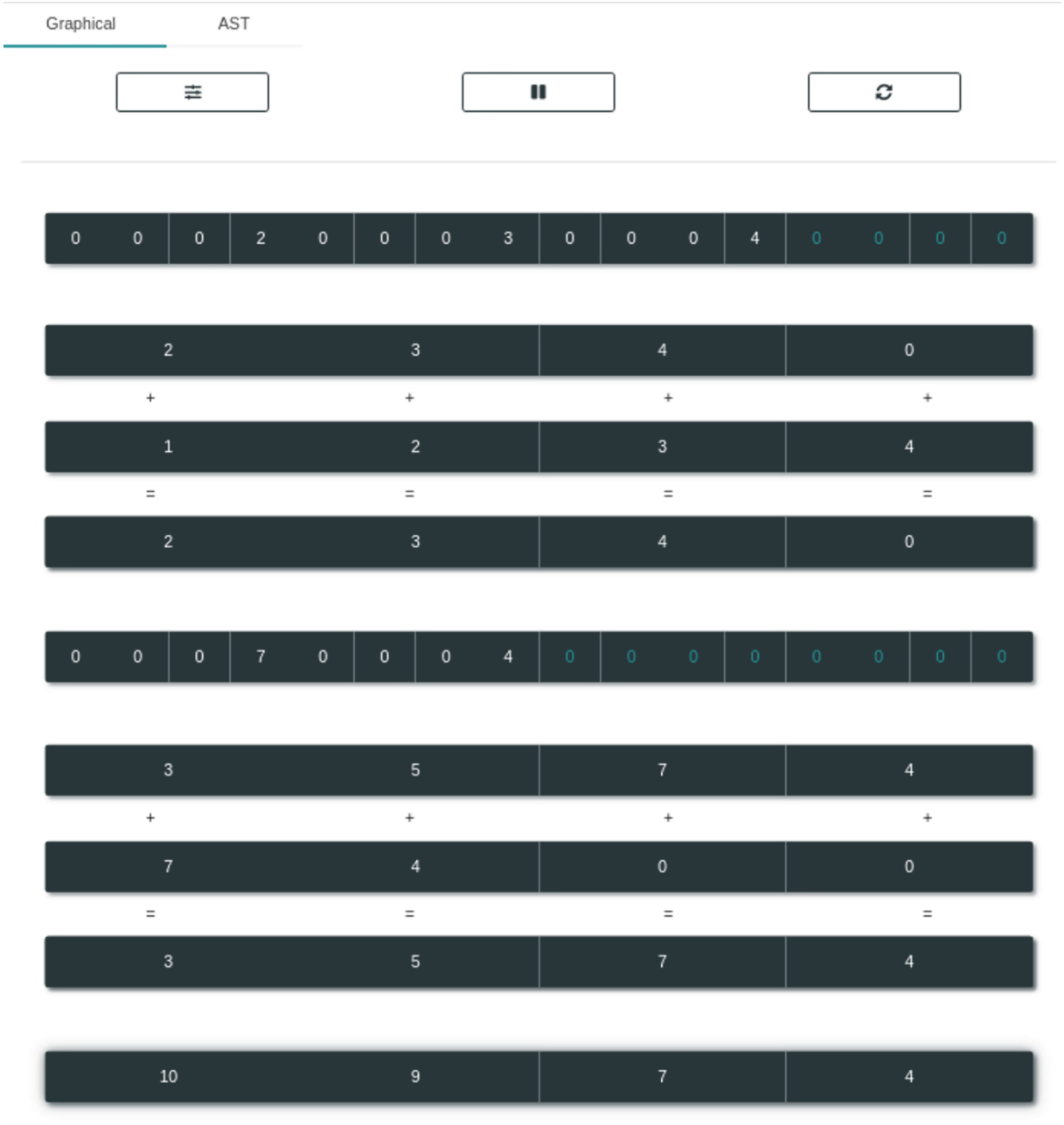


Figure 6.1: Visualization of the Prefix Sum Algorithm in SIMD-Visualiser

Syntax Tree (AST), and the functionality to write, compile and debug SIMD code. The tool uses the Clang compiler to compile SIMD code into assembly language, then utilises JavaScript libraries such as React (for the user interface) and Anime.js (for animation) for parsing and visualisation. SIMD-Visualiser is capable of not only displaying visual results of SIMD intrinsic functions but also the ASTs generated by Clang as shown in Figure 6.2.

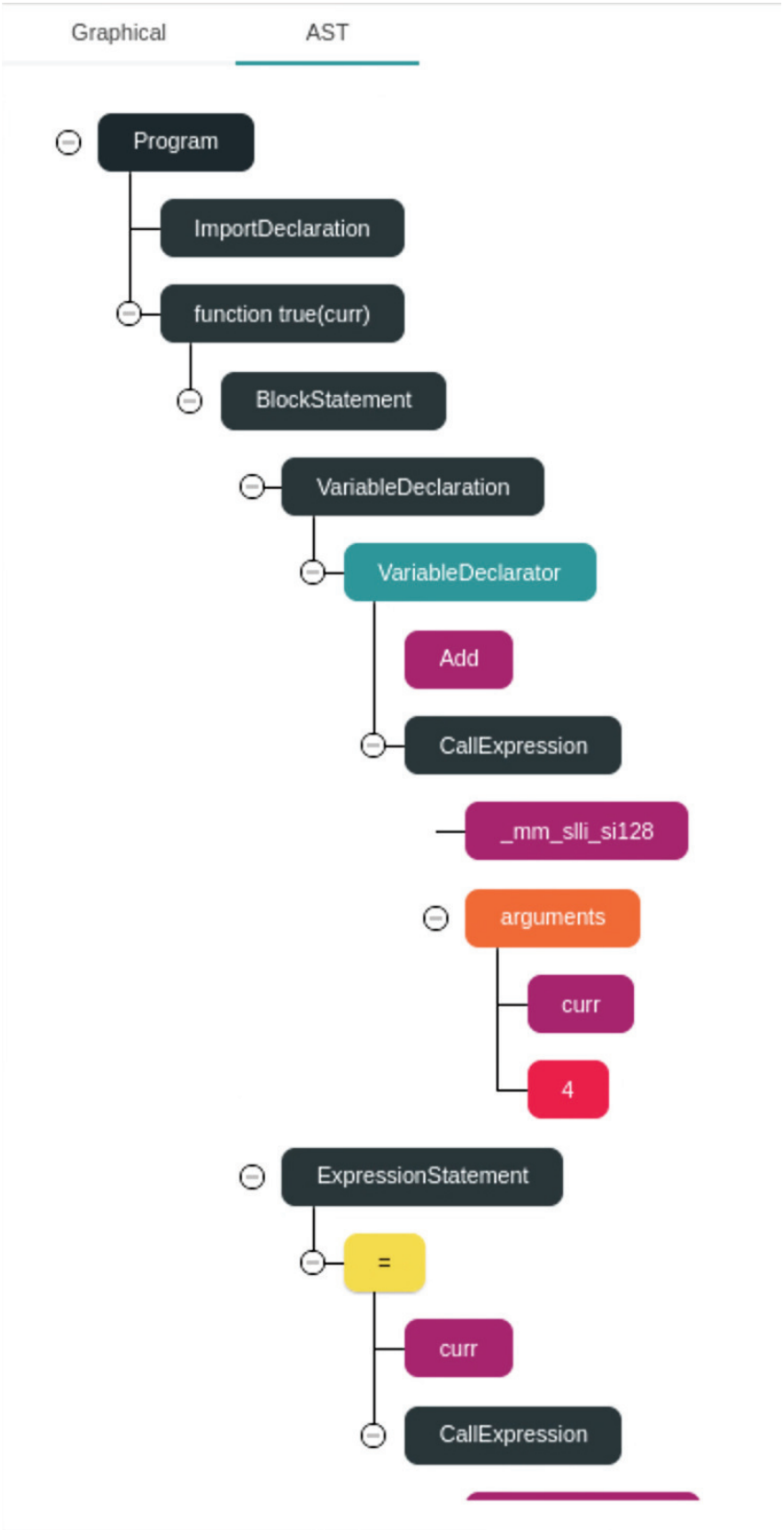


Figure 6.2: AST generated by SIMD-Visualiser

6.2.2 Comparison

Similarities: Although SIMD-Visualiser aims to visualise SIMD intrinsic functions, it visualises them after compiling the functions into assembly language using Clang, hence, this tool fundamentally employs the same low-level visualisation as PixelAssemblySIMD. Both systems utilise animation for dynamic visualisation of code.

Differences:

- SIMD-Visualiser is a web-based tool, made with JavaScript based on React and Anime.js. This is very similar to the tech route of the first version of PixelAssemblySIMD. Therefore, they share the same problem of having lower data reliability. From the demonstration of SIMD-Visualiser, it can be inferred that its data might be incorrect during complex modifications. However, the second version of PixelAssemblySIMD, using a CPU emulation library cpulib to manage CPU data, has solved this issue.
- SIMD-Visualiser visualises the procedural operation of each SIMD intrinsic function from top to bottom, which leads to a problem: it ties the operation of SIMD to the control flow of the data. However, SIMD should not rely on the program's control flow[2], therefore PixelAssemblySIMD's visualisation based on data flows conforms more to SIMD's characteristics.
- SIMD-Visualiser sequentially presents multiple visualisation results, but there are no clear distinctions between them. This makes it difficult for users to discern which are the visualisation results of the same instruction. PixelAssemblySIMD's visualisation strategy based on data flows does not encounter this problem.
- SIMD-Visualiser lacks support for more modern SIMD instruction sets such as AVX-512, while PixelAssemblySIMD already supports visualisation of these instruction sets.
- SIMD-Visualiser generates beautiful dynamic visualisation animations with Anime.js. However, PixelAssemblySIMD uses a very simple animation system, resulting in a gap in animation effects. In terms of code implementation, Anime.js is easy to use, but PixelAssemblySIMD's animator is more difficult to program, which poses certain challenges to adding new instruction visualisation.

6.3 NEVADA

6.3.1 Project Overview

NEVADA[35] is a tool developed in collaboration between University of Szeged and ARM, specifically designed for ARM's NEON[*1] instruction set, which appears as shown in Figure 6.3 during operation. The NEVADA tool enables developers to visualise and understand the execution of NEON instructions, crucial for optimising software performance in embedded devices.

By allowing users to observe and modify the behaviour of NEON code snippets in a web browser without a full development environment, NEVADA simplifies learning process. This makes the experimentation with NEON much more convenient and accessible. The tool is implemented in Java and translated into HTML/JavaScript by the Google Web Toolkit, making it accessible through a web browser. It emulates a simplified ARM CPU, inclusive of NEON and ARM registers, and includes a linear memory space for loading and storing data. The tool's interface is similar to a simplified debugger, allowing users to modify register or memory content, and execute code in different modes, including step execution.

This emulator supports a wide range of NEON instructions, such as logical, arithmetic, comparison, and conversion operations. It provides features like breakpoints, detailed views of NEON and ARM core register sets, visualisation of memory data.

6.3.2 Comparison

Similarities: Both NEVADA and PixelAssemblySIMD provide assembly-level visualisation.

Differences:

- NEVADA is designed for ARM's NEON instruction set, while PixelAssemblySIMD is designed for x86_64's SSE, AVX, and AVX-512 instruction sets.
- PixelAssemblySIMD emphasizes the description of instruction behavior based on animation, while NEVADA displays the status of registers and memory in ARM in a graphical way, more like a debugger with a graphical interface. When executing complex SIMD data shuffle operations, it may not be possible to know the real source of the data through NEVADA's graphical interface, while the animation of PixelAssemblySIMD can clearly show the changes in data.

[*1] NEON is the SIMD extension of the ARM processor

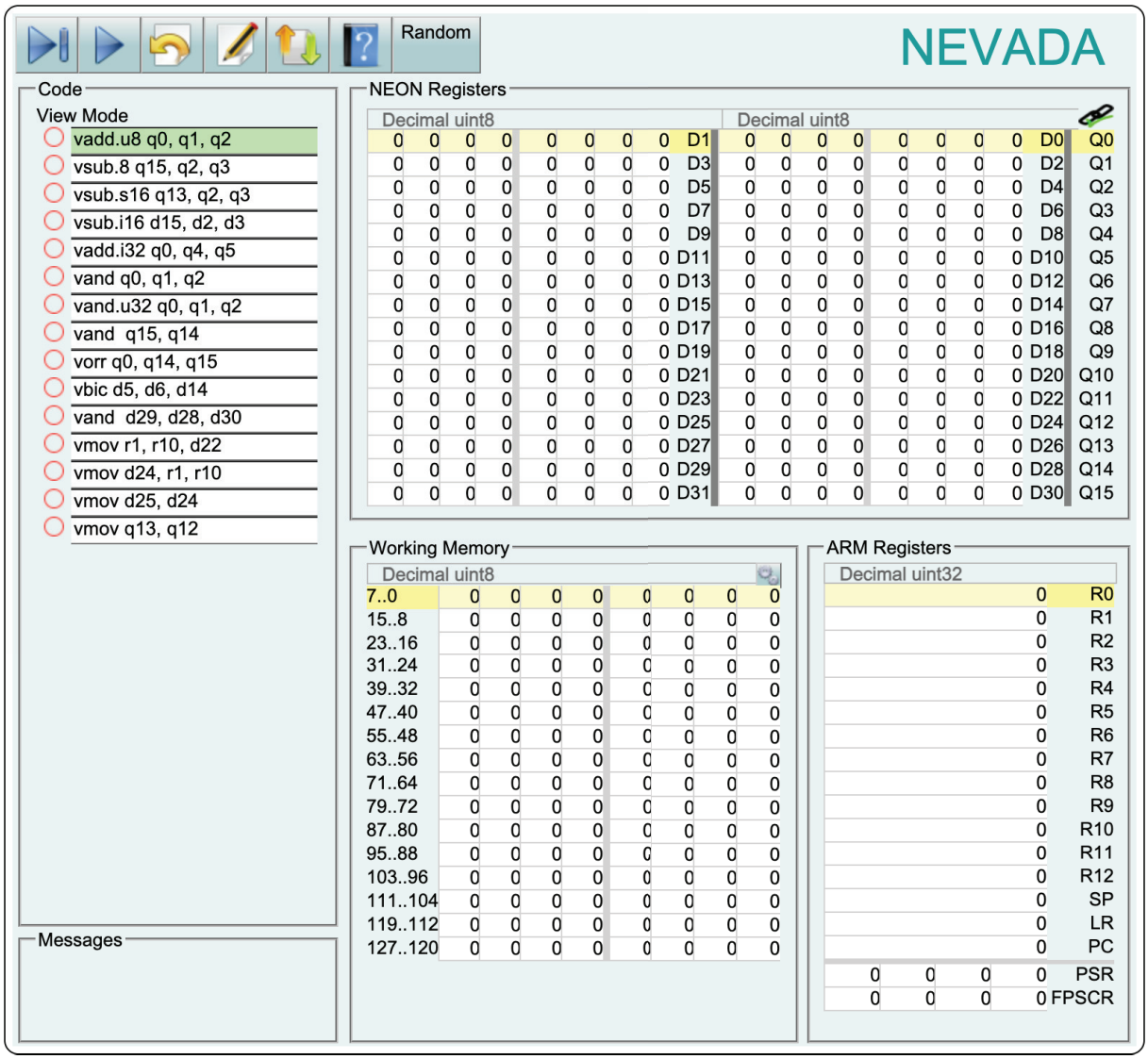


Figure 6.3: NEVADA

- As NEVADA does not need to show animations, if the user does not need to have a deep understanding of the SIMD instruction execution process, then NEVADA's graphical interface can show data changes faster, helping programmers debug faster. Therefore, NEVADA is more suitable for programmers to debug, while PixelAssemblySIMD is more suitable for programmers to learn.

7

Conclusions and Further Work

This chapter concludes the work of the thesis and proposes the direction of subsequent work. This chapter first summarizes the main work of this paper and then looks forward to future work.

7.1 Conclusions

This thesis first proposed the challenges humans face in understanding SIMD computing, and analyzed it from multiple aspects, positing that ”the conflict between different cognitive roles of humans” and ”the duality of programmer cognition” might be the deeper causes. Therefore, the thesis proposes the view that ”assisting humans in SIMD computing is necessary”. Subsequently, the thesis mentioned that visualization is an effective means to understand abstract and complex concepts, thereby proposing a method to assist humans in understanding SIMD calculations, that is, visualizing the execution process of SIMD calculations through visualization technology to help humans understand SIMD calculations.

Then, the thesis proposed a general data flow-based visualization method design, and implemented a tool prototype that implemented this visualization method design, which named PixelAssemblySIMD. The thesis first mentioned an initial design, analyzed its shortcomings, and proposed improvements, finally achieving a usable and reliable tool prototype. During the design of the tool prototype, the thesis introduced a CPU emulation framework cpulib designed to ensure the correctness of visualization data, and a method to implement asynchronous behavior animation in synchronous systems.

Next, the study used user research methods to evaluate the proposed visualization scheme. The results showed that the visualization scheme proposed in the thesis could help humans understand SIMD calculations; it also revealed the shortcomings of the visualization scheme, that it is not as intuitive as static visualization methods in assisting simple code. In the user research-based evaluation, the paper also confirmed through the results, the help of programming thinking in understanding abstract prob-

lems. At the end of the evaluation, the thesis provided some algorithm examples of assembly language using SIMD calculations, which can be used to test the correctness of the visualization tool.
Finally, the thesis introduced three different SIMD visualization tools, analyzed their strengths and weaknesses, and compared them with the method proposed in this thesis.

7.2 Further Work

7.2.1 CPU Emulator

Function Extension

The current cpulib only completes the emulation of the CPU's registers and memory, and the function of instruction execution is implemented in the visualization tool. However, the goal of cpulib is to serve as a stand-alone CPU emulator, so it is necessary to incorporate the function of instruction execution into cpulib. As part of future work, cpulib will implement a JIT technology-based instruction execution function similar to QEMU, so that cpulib can efficiently execute instructions independently of the visualization tool. In addition, cpulib needs to implement the emulation of other important components in the CPU, such as the interrupt controller, clock, etc. These parts are not needed in the visualization tool, but they are essential for a complete CPU emulator.
In addition, continuing to optimize the performance of cpulib is also part of future work. Currently, cpulib's performance is sufficient to support the use of visualization tools, but if cpulib were to function as a stand-alone CPU emulator, its performance is still subpar. Therefore, the performance optimization of cpulib is also part of future work.

Integration with WebAssembly

Currently, the Rust language can be easily compiled into WebAssembly[29], and the runtime environment for WebAssembly has also matured. There are even some virtual machine software based on WebAssembly[36]. Moreover, PixelAssemblySIMD can now run in WebAssembly based on browsers, which means cpulib can also be compiled into WebAssembly. cpulib can run in browsers without the need to install any software. At the same time, it can also serve as a WebAssembly library, used by other WebAssembly programs. Therefore, utilizing web-based animation technologies for creating visualization tools is one of the potential future works, allowing for more attractive animations in browsers while ensuring data accuracy. Secondly, a new subject of interest is a virtual machine and debugging environment that can run in browsers, which is an example of cpulib being used independently. This

could become a new target for future work.

7.2.2 Visualization Method

According to the evaluation results shown in Chapter 5, the visualization method proposed in this thesis has a significant impact on assisting human understanding of SIMD calculations. However, the visualization method proposed in this thesis is less intuitive than static visualization methods when explaining simple code. Therefore, part of the future work is to improve the proposed visualization method, incorporate support for static visualization while maintaining the current dynamic visualization, to accommodate different user groups and understanding levels.

7.2.3 PixelAssemblySIMD

Instructions Supported

Currently, PixelAssemblySIMD only supports common SIMD instructions. To make it a tool that can be effectively used in most programs, part of the future work is to add support for more instructions.

In addition, PixelAssemblySIMD does not currently change the value of the flag register when executing an instruction, which to some extent limits the functionality of PixelAssemblySIMD. However, cpulib itself already supports the flag register, so adding support for the flag register is also part of the future work. After support for the flag register is added, PixelAssemblySIMD can natively support instructions such as jumps without the need to implement this feature in the debugger, which is a more logical design.

Visualization of Memory

The current visualization of memory and the execution of memory access instructions in PixelAssemblySIMD lacks special animations or even misses some key animations during execution. Therefore, in the future, support for dynamic memory visualization could be added to PixelAssemblySIMD, allowing memory access instructions to be understood more intuitively by users.

Acknowledgements

First and foremost, I would like to express my profound gratitude to my mentor, Prof. Kazunori Ueda. As someone who ventured alone to study in Japan, you have provided me with invaluable guidance academically and immense support and care in my personal life. I deeply regret not grasping the true essence of research methods and what genuine scientific inquiry entails earlier; however, I am even more grateful for your consistent, meticulous, and patient teaching. Without it, I might have struggled to find my direction. I am especially thankful for your assistance and concern when I fell ill, which was a tremendous comfort to me, being sick and alone in a foreign land. Here, I extend my sincerest thanks and highest respect to you.

Secondly, I extend special thanks to all my colleagues, especially to Mr. Naoki Yamamoto and Mr. Masato Gocho, who were instrumental in my initial days in Japan. Mr. Yamamoto's assistance upon my arrival and his continual feedback were invaluable, while Mr. Gocho's insights into SIMD greatly clarified my understanding and piqued my interest. Despite my personal challenges in fully integrating into the group, I deeply appreciate your inclusiveness and the rich learning experience you all provided. I regret missing out on more shared experiences and cultural explorations with you. I sincerely wish each of you a prosperous future and remarkable success.

Furthermore, a heartfelt thanks to my friends in China. Your companionship and support, despite the distance, have been a source of immense comfort. I am particularly grateful to Mr. Yucheng Zhao for his company during my time in Japan, and wish him all the best in his forthcoming study abroad journey. My appreciation also extends to Mr. Lingzhi Zhang, my college roommate, whose reunion in Japan was a source of great joy. Your presence and support have been invaluable to me, and I cherish each of you deeply.

Lastly, I must express my deepest gratitude to my family, my strongest supporters. Thank you for your selfless love and endless support. Through every phase of my education and life, you have been my steadfast pillar. Your wisdom and experience have guided me through challenges in academics and life. Your understanding and tolerance allowed me to feel the warmth of home even while abroad. Your sacrifices and efforts are the driving forces behind my continual progress. Every achievement I have made is inseparable from your nurturing and encouragement. In the days to come, I will strive harder to live up to your expectations and repay you for your kindness and upbringing.

January 2024 Zhong ZHONG

References

[1] A. Fog. Optimizing subroutines in assembly language: an optimization guide for x86 platforms, 2008 (cited on page 1).

[2] M. Hassaballah, S. Omran, and Y. B. Mahdy. A review of simd multimedia extensions and their usage in scientific and engineering applications. *The Computer Journal*, 51(6):630–649, 2008 (cited on pages 3, 6, 7, 115).

[3] K. R. Wadleigh and I. L. Crawford. *Software optimization for high-performance computing*. Prentice Hall Professional, 2000 (cited on page 3).

[4] W. Zhang, Y. Wang, and K. A. Ross. Parallel prefix sum with simd, 2023. arXiv: 2312.14874 [cs.DC] (cited on pages 3, 104).

[5] M. Friedell, M. LaPolla, S. Kochhar, S. Sistare, and J. Juda. Visualizing the behavior of massively parallel programs. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 472–480, 1991 (cited on pages 6, 11).

[6] P.-M. Ntang and D. Lemire. Simdgiraffe: visualizing simd functions. In *VISIGRAPP (3: IVAPP)*, pages 147–154, 2021 (cited on pages 6, 112).

[7] A. Ortiz. Teaching the simd execution model: assembling a few parallel programming skills. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 74–78, 2003 (cited on page 6).

[8] B. Helmlinger, M. Sommer, M. Feldhammer-Kahr, G. Wood, M. E. Arendasy, and S. E. Kober. Programming experience associated with neural efficiency during figural reasoning. *Scientific Reports*, 10(1):13351, 2020 (cited on pages 10, 101).

[9] B. A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing*, 1(1):97–123, 1990 (cited on pages 10, 11).

[10] K. Ala-Mutka. Problems in learning and teaching programming-a literature study for developing visualizations in the codewitz-minerva project. *Codewitz needs analysis*, 20, 2004 (cited on page 11).

[11] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009 (cited on page 11).

[12] N. Fraser. Ten things we've learned from blockly. In *2015 IEEE blocks and beyond workshop (blocks and beyond)*, pages 49–50. IEEE, 2015 (cited on page 11).

[13] P. Khaloo, M. Maghoumi, E. Taranta, D. Bettner, and J. Laviola. Code park: a new 3d code visualization tool. In *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 43–53. IEEE, 2017 (cited on page 11).

[14] N. A. Quynh and D. H. Vu. Unicorn: next generation cpu emulator framework. *BlackHat USA*, 476, 2015 (cited on pages 16, 18).

[15] W. M. Zabołotny. Qemu-based hardware/software co-development for daq systems. *Journal of Instrumentation*, 17(04):C04004, 2022 (cited on page 17).

[16] Stephan Venter. Understanding, using, and patching qemu, 2023. https://tuxcare.com/blog/understanding-using-and-patching-qemu/, Last accessed on 2023-12-31 (cited on page 18).

[17] W. Bugden and A. Alahmar. Rust: the programming language for safety and performance, 2022. arXiv: 2206.05503 [cs.PL] (cited on page 19).

[18] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. Safe systems programming in rust. *Communications of the ACM*, 64(4):144–152, 2021 (cited on page 19).

[19] W. Bugden and A. Alahmar. The safety and performance of prominent programming languages. *International Journal of Software Engineering and Knowledge Engineering*, 32(05):713–744, 2022 (cited on page 19).

[20] A. Saligrama, A. Shen, and J. Gjengset. A practical analysis of rust's concurrency story, 2019. arXiv: 1904.12210 [cs.DC] (cited on page 19).

[21] Z. Yu, L. Song, and Y. Zhang. Fearless concurrency? understanding concurrent programming safety in real-world rust software, 2019. arXiv: 1902.01906 [cs.PL] (cited on page 19).

[22] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. Rustbelt: securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, 2017 (cited on page 19).

[23] Bybydev. Why rust has steep learning curve, 2023. https://byby.dev/rust-learning-curve, Last accessed on 2024-1-2 (cited on page 20).

[24] David Karger, Lea Verou, Amy X. Zhang. Reading 5: UI Software Architecture, 2018. https://web.mit.edu/6.813/www/sp18/classes/05-ui-sw-arch/, Last accessed on 2024-1-2 (cited on pages 20, 21).

[25] Amaury Ricardo. Imperative v declarative ui design - Is Declarative Programming the future?, 2022. https://www.rootstrap.com/blog/imperative-v-declarative-ui-

design-is-declarative-programming-the-future, Last accessed on 2024-1-2 (cited on page 21).

[26] Tyler McGinnis. Imperative vs declarative programming, 2016. https://ui.dev/imperative-vs-declarative-programming, Last accessed on 2024-1-2 (cited on pages 21, 22).

[27] Adam Siwiec. Week 6: The Egui Rust Framework, 2023. https://siwiec.us/blog/week-6-the-egui-rust-framework/, Last accessed on 2024-1-2 (cited on pages 22, 23).

[28] Solomon Esenyi. The state of rust gui libraries, 2022. https://blog.logrocket.com/state-of-rust-gui-libraries/, Last accessed on 2024-1-2 (cited on page 23).

[29] K.-I. D. Kyriakou and N. D. Tselikas. Complementing javascript in high-performance node.js and web applications with rust and webassembly. *Electronics*, 11(19):3217, 2022 (cited on pages 43, 119).

[30] Roman Chumak. Egui_code_editor, egui Code Editor widget with numbered lines and syntax highlighting, 2023. https://github.com/p4ymak/egui_code_editor, Last accessed on 2023-12-16 (cited on page 44).

[31] C. Ziemkiewicz, M. Chen, D. H. Laidlaw, B. Preim, and D. Weiskopf. Open challenges in empirical visualization research. *Foundations of Data Visualization*:243–252, 2020 (cited on pages 82, 83).

[32] S. Liu, W. Cui, Y. Wu, and M. Liu. A survey on information visualization: recent advances and challenges. *The Visual Computer*, 30:1373–1393, 2014 (cited on page 83).

[33] Pierre Marie Ntang. Simdgiraffe, Visualize SIMD instructions, 2021. https://github.com/pmntang/SIMDGiraffe, Last accessed on 2023-12-16 (cited on page 111).

[34] Jeremie Piotte. Simd-visualiser, A tool to graphically visualize SIMD code, 2018. https://github.com/piotte13/SIMD-Visualiser/, Last accessed on 2023-12-16 (cited on page 112).

[35] DSE of U-Szeged. Nevada the arm-neon visualization tool, 2012. https://github.com/szeged/nevada/, Last accessed on 2023-12-16 (cited on page 116).

[36] Leaning Technologies. Webvm, virtual machine for the web, 2022. https://github.com/leaningtech/webvm/, Last accessed on 2023-12-31 (cited on page 119).

A

Source Code Structure

The structure of the source code is shown below.

A.1 cpulib

- /src/lib.rs

  External interface
- /src/registers.rs

  Register-related data and operations
- /src/memory.rs

  Memory-related data and operations
- /src/utilities.rs

  Utility functions

A.2 PixelAssemblySIMD

- /src/main.rs

  App entry point
- /src/visualizer_setting.rs

  Visualization settings window
- /src/reg_visualizer_data.rs

  Visualization data
- /src/reg_visualizer.rs

  Register visualization window
- /src/mem_visualizer.rs

Memory visualization window

- /src/animation_fsm.rs

  FSM for instruction execution

- /src/instruction_actuator.rs

  Instruction executor

- /src/utilities.rs

  Utility functions