Implementation of Floats and Strings in P2

Sean Shi
Sean.Shi@colorado.edu
University of Colorado, Boulder
Boulder, Colorado, USA

Ryan Wagster
Ryan.Wagster@colorado.edu
University of Colorado, Boulder
Boulder, Colorado, USA

Sarah Bian
Sarah.Bian@colorado.edu
University of Colorado, Boulder
Boulder, Colorado, USA

Abstract

This project extends P2 with floating point values, more math operations, and strings including operations on strings such as addition and multiplication. This allows for a wider range of programs and presented some unusual challenges. It required additions to support floating point assembly instructions using SSE and additions to the runtime library to support floating point values.

ACM Reference Format:
Sean Shi, Ryan Wagster, and Sarah Bian. 2025. Implementation of Floats and Strings in P2. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Project Link: https://github.com/csci4555-sp25/final-project-sean-sarah-ryan

1 Introduction

In order to create a more complete python implementation, we will add floating point operations, additional operators like multiplication and division, and we will implement the translation of float literals and strings literals into assembly.

Functionality for floating point literal assignments, floating point comparisons, floating point addition, subtraction, multiplication, and division, as well as printing floating point numbers, will all be supported. Functionality for all equivalent integer and boolean arithmetic will also be supported and explicated to be compatible with floating point numbers.

Our goal with this project is to implement floating point numbers without traditional x86 memory usage and .data sections typically reserved for floating point operations, and implement strict 32 bit floating point numbers (while Python typically uses 64 bit floating point numbers.) This will reduce x86 code size and eliminate the needs for another register allocator for %xmm registers and memory storage and access for floating point values.

2 Floating Points

2.1 Background and History

FPU Operations in x86. To implement floating point operations, we will be using the 32 bit SSE instructions rather than the older x87-FPU instruction set. These instructions use a set of special registers %xmm[0-7] for floating point calculations and data must be moved into or out of these registers with special move instructions namely `movss` and `movd` [1]. The movss instruction is typically used for moving values between %xmm registers and memory, while movd is used for moving doubleword values between %xmm registers and general purpose registers.

The reason for this curious design is because the original x86 processor had a separate floating point processor the x87 which used a stack with 80 bit values. These instructions are kept around for compatibility reasons and useful for certain scientific applications which benefit from the greater precision, but the SSE instructions present a simpler interface for performing SIMD or floating point operations.[2]

2.2 Implementation

2.2.1 Floats in Assembly. To represent floating point values in assembly, we must translate the bytes of a floating point value into an immediate or which we can store in the `.data` section of the assembly. Some example C code[1] for this is shown here:[3]

```
// Quake III source code
float y = number;
// evil floating point bit level hacking
long i = *(long *) &y;
```

The implementation of this in Python is somewhat trickier since it requires use of the `ctypes` module generally used in FFI applications.

```
from ctypes import *
f = 5.5 # choose any value
ptr = pointer(c_float(f))
ptr2 = cast(ptr, POINTER(ctypes.c_int))
data = ptr2.contents.value
```

[1]A version of this code is used to fix a bug in runtime.c

In our intermediate representations (such as .flatpy), the floating point values are represented as immediate integer values (which appear like very large numbers due to the way FPU bit mapping works) that are tagged with the FLOAT tag from runtime.c rather than the INT tag.

Only %xmm0 and %xmm1 are ever used for floating point operations, and a new register allocation method and .data memory assignment was not needed for floats because of this method, drastically reducing code size and reducing memory usage.

In a traditional x86 representation, floating point numbers are stored as data i.e. in the `data` section to be accessed and moved using movss throughout the program.

2.2.2 Explication.

Float - Float. The first element of arithmetic operations that was finished was float on float operations for addition, subtraction, multiplication, and division. During explication, all floating point numbers are typically stored as integer values and tagged with the float tag defined in runtime. Because the value is only stored as an integer but is still a valid floating point number binary string, moving the value back into an %xmm register is all that needs to be done to use the value as a float again.

The x86 floating point instructions can be used exactly the same way that the integer instructions are used, but must be used between two %xmm registers. Because we used the general purpose registers for all value handling, %xmm0 and %xmm1 are always used for floating point operations. After the operation completes the float is moved from %xmm0 back into the general purpose destination register. The movss instruction is incindentally never used because we never needed to directly bridge %xmm registers and memory, as this was handled through general purpose registers by our register allocator algorithm.

Float - Integer / Boolean. In order to properly perform arithmetic operations between a float and a non-float numeric type, the non-float type must always be promoted to a float type. Initially we attempted to solve this problem through the use of new functions in runtime.c that would perform a C-style cast on the input integer or boolean and return a floating point number. In lieu of this we instead used an x86 instruction to accomplise the same thing without the need for a function call. When an integer or boolean needed to be promoted to a boolean, the cvtsi2ssl x86 instruction was performed on the value, storing it in %xmm0, and then immediately moving the value back to a general purpose register.

After this type conversion, the instruction is carried out as normal for float - float operations. The only additional step is the int to float conversion. Support was added for an float to int conversion, but in the environment, every arithmetic operation that involved at least one floating point value argument will resolve as a floating point number, so the float to int conversion is unused in this compiler.

Int - Int. As it would be awkward to allow some arithmetic for floating point numbers but not for integers, binary operation subtraction, multiplication, and division was also added for integer values. For subtraction and multiplication, the standard x86 subl and imul (signed integer multiplication) instructions were used. However, for division, in the Python environment, a division between two integers will produce a floating point number. So, both integer values are promoted to floating point numbers and the divss instruction is used.

2.2.3 Imprecision. To represent floats in our implementation, we will be using the implementation in `runtime.c` which uses the last 2 bits as the relevant tag. This format mostly follows the IEEE 32-bit/single-precision floating point, but has a 21 bit rather than the standard 23 bit value. This does have some implications for precision, notably that many values will be rounded down slightly. This problem is compounded by the fact that we are using single-precision floating points rather than python's double precision.

sign	exponent	significand	tag
1	8	21	2

Figure 1. Layout of modified floating points

As a result of these limitations, our implementation will not provide correct results for many fractions and results may differ slightly from python. It is worth noting though that floating points are somewhat imprecise no matter what implementation is used.

3 Strings

3.1 Implementation

We rely heavily on `runtime.c` in our implementation of strings because implementing string operations in assembly would require extensive use of malloc, interactions with unions and structs, and many other programming constructs which would be somewhat tedious to implement in assembly.

Our representation of strings aligns with C strings which are null-terminated character arrays. To implement these, we added an additional tag to the big_pyobj type enum and employed a struct with the same properties as a list save that we used a pointer to a character array.

In order to emulate python, we also added additional cases to add.

3.1.1 Assembly.

Strings in our implementation are stored using the `.string` assembler directive in the `.rodata` section of an assembly file. These can then be accessed as an immediate or using any other method.

3.2 Explication

Str - Str. To represent strings in assembly, we needed to first create a .data section that saves constants like string literals into memory. For example, for the python expression `print("hello world")`, the label L_0: is used to label the null-terminated string ”hello world”. This ensures that L_0 refers to the memory address of that literal string in the final executable. In this way, every string is given a unique label and a raw pointer to the memory address.

In order to access this memory address, the address for the string is pushed unto the stack, and the runtime function inject_string is called, which takes in the pointer and asserts that it does point to a string, it then processes the tagged string like any other variable, moving it into either memory or a register for reuse depending on coloring and liveness.

3.2.1 String Concatenation.

Strings use the same logic as lists for concatenation, so a similar function was developed for addition explication and added to `runtime.c` as a case for `add`. However, since strings are immutable, we must create an entirely new array and then use memcpy to create the string.

4 Conclusion

The above implementations of float and string values proved a fulfilling challenge for furthering the functionalities and implementations of P2. This project successfully extended P2 to handle both floating point arithmetic using SSE and tagged float representations, as well as string manipulation via tagged pointers and memory-labeled string literals. These additions enable a wider variety of Python programs to be compiled and executed in our system.

More unit tests to test edge cases of float arithmetics and string manipulation could be added to increase the robustness of our modified P2 code. Future functionalities could include additional string manipulation, such as string slicing, and memory management for floating point numbers. String slicing, for example, is a functionality that our code currently does not support. It would require additional functions to read start, end, and step arguments and to save the new string within .data as necessary. Memory management for floating point numbers could be further optimized to isolate them to %xmm registers and memory rather than utilizing general purpose registers for floats.

5 Contributions

Sean Shi

- Emulation of Floats as Integer Representation
- Float Literal Implementation
- Generalization of Explication Function
- Floating Point Add Implementation
- Floating Point Test Programs
- integer to floating point conversion
- runtime.c fix for floats and implementation of string type

Ryan Wagster

- Prototype of Floats with C-style Casts
- Explication of Float Types
- FPU and Int/Bool Sub, Mult, and Div Implementation
- FPU and Add, Sub, Mult, and Div and String Test Programs
- IR to x86 for FPU operations
- Kanban Project Management

Sarah Bian

- String Literal Implementation
- String Concatentation and Splicing Explication
- String Test Programs

Project Repository

This project can be accessed at https://github.com/csci4555-sp25/final-project-sean-sarah-ryan

Acknowledgments

To Professor Joseph Izraelevitz

References

[1] 2021. x86-64 Assembly Language Programming with Ubuntu (Jorgensen). https://eng.libretexts.org/@go/page/19978 [Online; accessed 2025-05-01].

[2] David M. Russinoff. 2022. *x87 Instructions*. Springer International Publishing, Cham, 247–251. https://doi.org/10.1007/978-3-030-87181-9_13

[3] Shaw. 2017. The Legendary Fast Inverse Square Root. https://medium.com/hard-mode/the-legendary-fast-inverse-square-root-e51fee3b49d9