Project Report

Shujun Liu (username: liushuj)

1. Introduction

This project tries to extend the language by adding to it partial floating point support implemented using SIMD instructions. Floating point operations are very commonly used and well supported by current x86 processors, so this project tries to make the language able to do some of floating point arithmetic, concretely:

1. Assignment;
2. Addition;
3. Subtraction;
4. Comparison

, and at the same time make the compiler able to compile these FP operations into corresponding SSE1&2 SIMD instructions in the generated assembly code.

Currently the floating point support is not implemented for dynamic types so only the passes after type checking is used.

2. Type checking

The language is added a new type named "Double" for double-precision FP numbers; and the syntax of the language has a new kind of terminal symbol: floating point numbers. As boundary condition for the type checker, the Racket built-in function "flonum?" is used to recognize this new kind of terminals.
For all arithmetic and comparison operations, the types of the two operands are checked. If there is one FP operand, then the operation is turned into a FP operation. For the sake of instruction selecting, FP operators uses different symbol than integer operators. The new operators include:
+f -f >f <f >=f <=f feq? int double
The +f, -f, >f, <f, >=f, <=f, and feq? is the FP version of corresponding operations, while "int" and "double" is used for casting between integer and double types. This is useful because even a same number as integer and FP value have different representation in machine level. The actual conversion is done later in select-instructions.
Some examples of the behavior of type checker:

```
(+ 1.1 2) ->
(has-type
         (+f
          (has-type 1.1 Double)
          (has-type (double (has-type 2 Integer)) Double)
Double)

(int (+ 10.5 10.5)) ->
(has-type
         (int
         (has-type
                  (+f
                 (has-type 10.5 Double)
```

```
                    (has-type 10.5 Double))
            Double))
Integer))
```

3. Shrink and FP constants

The shrink pass processes the new +f, etc. operations using the same approach as those operations for integers. Besides, there is a new task accomplished in "Shrink" : uncover all the floating point constants.
SSE provide moving and arithmetic operations similar to those for integers: movsd, addsd, subsd, etc. The problem with them is that their operand cannot be immediate floating point values; in other words, an instruction like

```
addsd $1.234, xmm0
```

is not valid.
However, a FP constant can be stored in the program and accessed using RIP-relative addressing, for example:

```
const1:
        .double 10.5
.globl main
main:
        pushq %rbp
        movq %rsp, %rbp
        movsd const1(%rip), %xmm0
        addsd const1(%rip), %xmm0
        addsd const1(%rip), %xmm0
        addsd const1(%rip), %xmm0
        cvtsd2si %xmm0, %rax
        popq %rbp
        retq
```

will return 42.
To support FP constant values, this compiler uncovers all such values appeared in the program in the shrink pass, generate a label for each of them, put them in a list (here called *fppool*) and the list is later used to generate labels and assembler directives[1] when generating assembly code.

4. Select Instructions

A big part of the work contained in this project lies in the "select instructions" pass. For FP operations, the processor uses different instructions and registers. Here in this project I chose the SIMD operations to implement these operations in order to get rid of difficulty brought by different architecture of x87 FPU.
As mentioned above, the new operations that need to be handled include:
+f -f >f <f >=f <=f feq? int double

The new instructions used include:
(selection of these instructions referred to disassembly result of gcc generated FP programs)
movsd: Move or Merge Scalar Double-Precision Floating-Point Value [2]
addsd: Add Scalar Double-Precision Floating-Point Values

subsd: Subtract Scalar Double-Precision Floating-Point Value
cvtsi2sd: Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value; used in “double”
cvtsd2si: Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer used in “int”
ucomisd: Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS. Used in comparison.

0. Registers
   SSE instructions uses a different set of registers, totally 8 are used in the project, named xmm0 – xmm7. In the project, among them xmm0 and xmm1 are preserved for temporary uses such as moving from memory to memory; at the same time xmm0 is used for return values.
1. Processing of FP constants
   As mentioned in 3, each FP constant is given a label for RIP-relative addressing. In the instructions that contains FP constant as operand, this label is looked up from the table and turned into a “global-value” kind of value, which will be later turned into label(%rip). The corresponding code is like:
   ```
   ((? flonum? f)
      (define lbl (dict-ref fppool f))
      `((movsd (global-value ,lbl) (var ,var))))
   ```
   For example: “movsd (global-value fpconst1) (var v1)”
   There is no need to see if one operand of fp operations is an integer because it is added an casting operation and turned into a variable in the former operations.
   For example, (+f 1.1 2) becomes (let ((v (double 2))) (+f 1.1 v))
2. +f
   Similar to integer +, first movsd one operand to the target variable then addsd the other operand on it.
3. -f (change sign)
   I didn’t find an instruction similar to *neg* in SSE1&2, so changing sign is done by moving 0.0 to the target variable then subtract it with the operand. Because of this, the fppool is initialized with a pair (0.0 . fpconst0).
4. Comparisons
   Comparisons are shrinked before so that there only remains “feq?” and “<f”. These are handled similar to the integer version, the difference being that *ucomisd* instruction is used instead of *cmp*. Also they have similar behavior when setting status registers.
5. “int” and “double”
   Conversion between FP value and integers are done using the machine instruction “cvtsd2si” and “cvtsi2sd”. They take an integer (or Double) from a register or memory, convert to Double (or integer) representation, and save it in a register. Because the target must be a register, xmm0 is used first then the value is moved into the target variable; because the source cannot be immediate value, immediate integers are first moved into rax.
6. Functions
   The new problems for functions are that: 1. FP arguments must be passed using FP registers; 2. Return value of a Double typed function should be put in xmm0.

For 1, when handling “call” and “tail-jmp” during instruction selection, the list of types of the function arguments is first looked up, the list is iterated through, and the 1st integer arg seen the list is put into the 1st integer register, the 1st FP arg is put into the 1st FP register, the 2nd integer arg put into the 2nd integer register… etc. Same procedure is done for the caller and the callee to keep the order of the args.
For 2, the return type of the function is looked up; if it is Double, after the call returns, xmm0 is copied into the target variable using movsd; otherwise rax is moved into the target as before.

5. Uncover-live and Register Allocation

The new instructions need to be handled in uncover-live. "addsd” and “subsd” is handled similar to “add”, “movsd”, “cvtsd2si” and “cvtsi2sd” similar to “mov”, “ucomisd” similar to “cmp”. They are handled similarly when building interference graph.
At the end of register allocation, when actually choosing registers, the type of the variable is looked up (from “locals” in the info field), if it is FP variable then an xmm register is allocated to it, otherwise a regular register is used.
All xmm registers are considered caller-saved register.[3]
The other parts of uncover live, build interference and register allocation is not changed, which means that the same set of numbers are used during graph coloring, e.g. if an integer variable interfering with a FP variable, they will be given different numbers. The reason is that I don’t think this will hurt the correctness of the result although hurting efficiency. Interference between an int variable and an FP variable does not affect interferences between same type of variable. On the other hand, if there is not interference between two vars of different type, e.g. a FP var and an integer are both using a same number, in the end they will be in different registers.

6. Patch instructions

For the new instructions, when copying from memory to memory, the xmm0 register is used as a temporary station instead of rax.

7. Other passes

Trivial changes are made to other passes so that they accept floating point numbers as a new kind of terminal.

8. Print x86

“fppool” is formatted here into labels and assembler directives as mentioned in 3. For example,
((0.0 . fpconst0) (1.1 . fpconst1))
is turned into
fpconst0: .double 0.0
fpconst1: .double 1.1

9. Current status

By the midnight of Dec 17, 2019, all the functions mentioned above are implemented; however, the compiler is not thoroughly tested yet.

Currently it's known that it may work for simple programs that contains branch and arithmetic. Following is an example program and the generated assembly code:

```
(program ()
        (int
          (if (> 5.5 4.5)
            (+ 1 (+ 10.5 30.5))
            0.0))
```

Which will generate assembly code as follows:

```
fpconst430525: .double 0.0
fpconst430524: .double 30.5
fpconst430523: .double 10.5
fpconst430522: .double 5.5
fpconst430521: .double 4.5
fpconst0: .double 0.0

block430932:
        cvtsd2si %xmm2, %rax
        jmp mainconclusion
block430931:
        movsd fpconst430525(%rip), %xmm2
        jmp block430932
block430930:
        movq $1, %rax
        cvtsi2sd %rax, %xmm0
        movsd %xmm0, %xmm2
        movsd fpconst430524(%rip), %xmm3
        addsd fpconst430523(%rip), %xmm3
        addsd %xmm3, %xmm2
        jmp block430932
mainstart:
        movsd fpconst430521(%rip), %xmm2
        movsd fpconst430522(%rip), %xmm3
        ucomisd %xmm3, %xmm2
        jg block430930
        jmp block430931

        .globl main
        .align 16
main:
        pushq %rbp
        movq %rsp, %rbp
        pushq %rbx
        pushq %r12
        pushq %r13
        pushq %r14
        subq $0, %rsp
        movq $16384, %rdi
        movq $16384, %rsi
        callq initialize
        movq rootstack_begin(%rip), %r15
        jmp mainstart
mainconclusion:
        subq $0, %r15
        addq $0, %rsp
        popq %r14
        popq %r13
        popq %r12
        popq %rbx
        popq %rbp
        retq
```

10. Known issues

With the above implementation the compiler cannot handle FP values inside a vector. To support them, implementation of vector-ref and vector-set should be changed such that the correct instruction and register is chosen according to type of the vector element.

A possible workaround for users (if there is any) is to pack FP value in a function before putting it into a vector.

References:

[1] Assembler Directives - x86 Assembly Language Reference Manual. https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html

[2] x86 and amd64 instruction reference. https://www.felixcloutier.com/x86/

[3] Basics of SSE Assembly Programming. http://homes.sice.indiana.edu/rcwhaley/teach/iseHPO_F18/LEC/06ASSE1_ho.pdf