COEN 175

Lecture 21: Floating Point

Intel Floating Point

- Intel originally used a coprocessor for floating-point.
- The coprocessor uses a stack for computation, so an expression is evaluated as if written in postfix.
- Consider the expression `a * b + c * d`:

```
fldl    a       # load/push a
fldl    b       # load/push b
fmulp           # multiply and pop
fldl    c       # load/push c
fldl    d       # load/push d
fmulp           # multiply and pop
faddp           # add and pop
```

- Any return value is left on the top of the stack.

Modern Floating Point

- Modern Intel processors (since the Pentium III) use SSE (Streaming SIMD Extensions) for floating point.
  - The SSE register names are `%xmm0` – `%xmm7`.
  - They are all **caller-saved** registers.
- The arithmetic opcodes are:
  - `movsd` – move scalar, double-precision
  - `addsd` – add scalars, double-precision
  - `subsd` – subtract scalars, double-precision
  - `mulsd` – multiply scalars, double-precision
  - `divsd` – divide scalars, double-precision
  - `ucomisd` – (unordered) compare scalars, double-precision

Example: Addition

- Assume the registers are `%xmm0`, `%xmm1`, etc.
- Assume that all variables are 64-bit reals and are global variables so they can be referred to by name.
- Generate code for `a * b + c * d`.

```
movsd  a, %xmm0           # load: %xmm0 allocated
mulsd  b, %xmm0
movsd  c, %xmm1           # load: %xmm1 allocated
mulsd  d, %xmm1
addsd  %xmm1, %xmm0       # %xmm1 deallocated
```

- Compare this with our code for integer arithmetic.

Putting It All Together

- Let’s write a complete function for addition.

```
void Add::generate()
{
    // Generate code for both the left child
    // and the right child

    // If the left child is not in a register,
    // then allocate a register and load it

    // Perform the operation such as
    // "addl right, left"

    // If the right operand is in a register,
    // then deallocate it
}
```

Putting It All Together

- Let's write a complete function for addition.

```
void Add::generate()
{
    _left->generate();
    _right->generate();

    if (_left->_register == nullptr)
        load(_left, FP(_left) ? fp_getreg() : getreg());

    cout << "\tadd" << suffix(_left);
    cout << _right << ", " << _left << endl;

    assign(_right, nullptr);
    assign(this, _left->_register);
}
```

Two sets of registers

Utility Functions

- Let's write the utility functions we've used.

```
# define FP(expr)     ((expr)->type().isReal())
# define BYTE(expr)   ((expr)->type().size() == 1)

static string suffix(Expression *expr) {
    return FP(expr) ? "sd\t" : (BYTE(expr) ? "b\t" : "l\t");
}

Register *fp_getreg() {
    for (unsigned i = 0; i < fp_registers.size(); i ++)
        if (fp_registers[i]->_node == nullptr)
            return fp_registers[i];

    load(nullptr, fp_registers[0]);
    return fp_registers[0];
}
```

Floating-Point Comparison

- The `ucomisd` instruction sets the flags register based on the floating-point status word.
- The zero and carry flag are set, but not the sign flag.
- Therefore, we must use the **unsigned** `set` and `jump` instructions:
  - `setb/jb` – set/jump if below
  - `seta/ja` – set/jump if above
  - `setbe/jbe` – set/jump if below or equal
  - `setae/jae` – set/jump if above or equal

Example: Comparison

- Assume all variables are 64-bit reals.
- Generate floating-point code for `x > y * z`.

```
movsd      y, %xmm0        # %xmm0 allocated
mulsd      z, %xmm0
movsd      x, %xmm1        # %xmm1 allocated
ucomisd    %xmm0, %xmm1    # %xmm0 and %xmm1 deallocated
seta       %al             # %eax allocated
movzbl     %al, %eax
```

- Note that the result has type `int` and is stored in an integer register, not a floating-point register.

Floating-Point Conversion

- To convert a 32-bit integer to a 64-bit real:
  - `cvtsi2sd` – convert scalar integer to scalar double
- To convert a byte to a 64-bit real, we must first sign extend the byte into a 32-bit integer register.
- To convert a 64-bit real to a 32-bit integer:
  - `cvttsd2si` – convert scalar double to scalar integer
- To convert a 64-bit real to a byte, we first convert the real into a 32-bit integer register and then use the corresponding byte register.

Example: Conversion

- Assume `n` is a 32-bit integer and `x` is a 64-bit real.
- Generate code for `n = (char) (x + n)`.
  - With no coercions: `n = (int) (char) (x + (double) n)`.

```
cvtsi2sd   n, %xmm0
movsd      x, %xmm1
addsd      %xmm0, %xmm1
cvttsd2si  %xmm1, %eax
movsbl     %al, %eax
movl       %eax, n
```

- Note that the cast to a `char` generates no code beyond the conversion to `int`, but we do use `%al` for the next operation.

Floating-Point Literals

- Floating-point literals must be stored in memory and referenced using an assembler label.
- The `.double` directive is used for this purpose:

```
.L0:    .double    3.14159
.L1:    .double    2.989792e+8
```

- Note that like the `.asciz` directive for strings, the `.double` directive must be used in the `.data` section of the assembly file.

Floating-Point Zero

- A floating-point value of zero is useful for:
  - Determining a truth value;
  - Negating a floating-point value.
- Although floating-point literals must be stored in memory, there is a simple way to generate zero:

```
pxor   %xmm0, %xmm0
```

- Generate code for `–x + y`.

```
pxor   %xmm0, %xmm0
subsd  x, %xmm0
addsd  y, %xmm0
```

Floating-Point Arguments

- Floating-point arguments are pushed on the stack.
  - However, we cannot simply “push” them because they are 8-byte values, not 4-byte values.
  - Instead, we must first adjust the stack pointer and move them ourselves, effectively mimicking a push instruction.
- Generate code for `f(x + y, z)`.

```
movsd  z, %xmm0
subl   $8, %esp
movsd  %xmm0, (%esp)
movsd  x, %xmm0
addsd  y, %xmm0
subl   $8, %esp
movsd  %xmm0, (%esp)
call   f
addl   $16, %esp
```

Floating-Point Return

- For compatibility, a floating-point return value is still placed on the top of the coprocessor stack.
- Since there is no direct path between the SSE registers and the coprocessor, we must use memory.

Spill and reload

```
# return x + y

movsd   x, %xmm0
addsd   y, %xmm0
movsd   %xmm0, t_0
fldl    t_0
jmp     f.exit
```

```
# x * f() + y

call    f
fstpl   t_1
movsd   x, %xmm0
mulsd   t_1, %xmm0
addsd   y, %xmm0
```

Spill

Summary

- Within our framework, floating-point values are treated no differently than integer values.
  - Our `load()` function is intelligent enough to handle either integer or floating-point loads.
  - We just need to tell it the correct register.
  - The arithmetic and comparison code generation functions require little to no modification.
- Code generation for type conversion is tricky.
- Function calls and returns require using the coprocessor stack and reloading from memory.