COEN 175

Lecture 19: Managing Registers

Managing Registers

- So far, we have not discussed how we allocate and deallocate registers while generating code.
- Suppose we have two C++ classes:
  - `Expression` – a base class for expressions;
  - `Register` – a class representing an Intel register.
- We need the following:
  - A way to find out if an expression is in a register;
  - A way to find out which expression is using a register;
  - A pool of available registers.

Linking the Two Classes

- Let’s assume the following C++ class definitions:

```
class Expression {
    …
public:
    class Register *_register;
    std::string _operand;
};
```

```
class Register {
    …
public:
    class Expression *_node;
};
```

- Given an `Expression`, we can access its register, which will be `nullptr` if it is not loaded into one.
- Given a `Register`, we can access its expression, which will be `nullptr` if it is not being used by one.

Managing the Links

- Let’s write a simple function to manage the links.

```
void assign(Expression *expr, Register *reg)
{
    if (expr != nullptr) {
        if (expr->_register != nullptr)
            expr->_register->_node = nullptr;

        expr->_register = reg;
    }

    if (reg != nullptr) {
        if (reg->_node != nullptr)
            reg->_node->_register = nullptr;

        reg->_node = expr;
    }
}
```

What Our Function Does

- The `assign` function does the following:
  - Disassociates any associated register from the node;
  - Disassociates any associated node from the register.
- The `assign` function does not:
  - Emit any assembly code to load values into registers;
  - Perform any spills if the register is already in use.
- All it does is maintain the proper mappings.
  - In other words, it is at the lowest level.
  - Policy decisions will be made at a higher level.

Register Allocation

- We need a pool of available registers.
- For simplicity, we will just use the caller-saved registers and allocate the first available register.

```
Register *eax = new Register("%eax", "%al");
vector<Register *> registers = { eax, … };

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

    abort();
}
```

Fail if no register is available

Printing Expressions

- Let’s overload the output stream operator for expressions for convenience.
  - If the expression is in a register, then we use it. Otherwise, we use its `_operand` field which references memory.
  - Assume that our `Register` class has a function `name(`*n*`)` that returns the *n*-byte register name.

```
ostream &operator <<(ostream &ostr, Expression *expr)
{
    if (expr->_register == nullptr)
        return ostr << expr->_operand;

    unsigned size = expr->type().size();
    return ostr << expr->_register->name(size);
}
```

Register Loads

- Finally, we need a function to load an expression into a given register.

```
void load(Expression *expr, Register *reg)
{
    if (reg->_node != expr) {
        assert(reg->_node == nullptr);

        if (expr != nullptr) {
            unsigned size = expr->type().size();
            cout << "\tmov" << suffix(expr) << expr;
            cout << ", " << reg->name(size) << endl;
        }

        assign(expr, reg);
    }
}
```

Fail if register is allocated

Use `suffix(x)` to return the correct opcode suffix for x

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, getreg());

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

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

Unresolved Issues

- Our new infrastructure works well and is very intuitive as it closely matches our algorithm.
- But, we still have a number of issues:
  - What if we call `getreg()` and no register is available?
  - What if we call `load()` to load an expression into a specific register and it is already allocated?
  - What happens to the caller-saved registers when we make a function call?
- Fortunately, we can fix all these issues at once by introducing spills.

Adding Temporaries

- To introduce spills, we will need to be able to create temporaries on the run-time stack.
- For simplicity, we will just assign temporaries the next available offset on the stack, just like locals.

```
void assigntemp(Expression *expr)
{
    stringstream ss;

    offset = offset – expr->type().size();
    ss << offset << "(%ebp)";
    expr->_operand = ss.str();
}
```

Smarter Loads

- Now let’s modify `load()` to perform spills.

```
void load(Expression *expr, Register *reg)
{
    if (reg->_node != expr) {
        if (reg->_node != nullptr) {
            unsigned size = reg->_node->type().size();

            assigntemp(reg->_node);
            cout << "\tmov" << suffix(reg->_node);
            cout << reg->name(size) << ", ";
            cout << reg->_node->_operand << endl;
        }

        /* rest of function same as before */
    }
}
```

Spill instead of failing

Smarter Allocation

- We can now write a smarter version of **`getreg()`**.
- If no register is available, then we will simply spill the first register and return it.

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

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

Spill the first register so it's available