

Compiler Design

Lecture 12: Introduction to Code Generation

Christophe Dubach

Winter 2025

Timestamp: 2025/02/10 14:27:00

Table of contents

Introduction

Overview

The Backend

The Big Picture

Code Generation

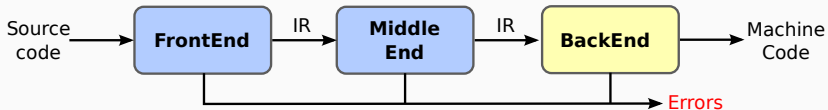
Code Generator for Arithmetic Expressions

Introduction

Introduction

Overview

Overview



Front-end

- Lexer
- Parser
- AST builder
- Semantic Analyser

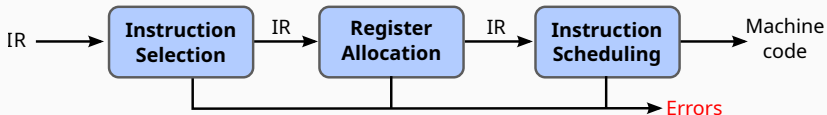
Middle-end

- Optimizations (Compiler Optimisations course)

Introduction

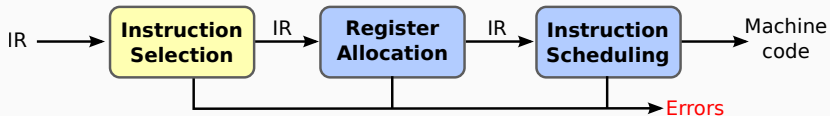
The Backend

The Back end



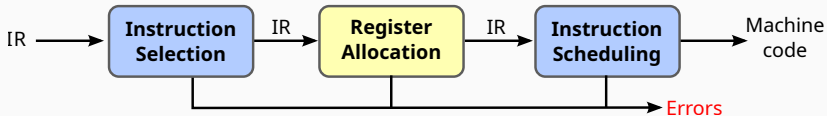
- Translate IR into target machine code
- Choose instructions to implement each IR operation
- Decide which value to keep in registers
- Ensure conformance with system interfaces
- Automation has been less successful in the back end

Instruction Selection



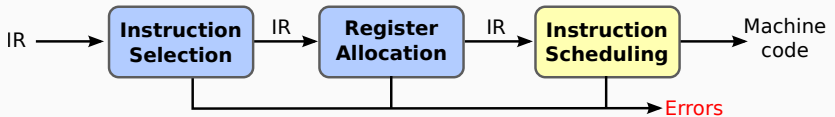
- Mapping the IR into assembly code (in our case AST to MIPS assembly)
- Combining operations (e.g. using addressing modes)

Register Allocation



- Deciding which value reside in a register
- Minimise amount of spilling

Instruction Scheduling



- Avoid hardware stalls and interlocks
- Reordering operations to hide latencies
- Use all functional units productively

Instruction scheduling is an optimisation

Improves quality of the code. Not strictly required.

Introduction

The Big Picture

The Big Picture

How hard are these problems?

- Instruction selection
 - Can make locally optimal choices, with automated tool
 - Global optimality is NP-Complete
- Instruction scheduling
 - Single basic block $\Rightarrow$ heuristic work quickly
 - General problem, with control flow $\Rightarrow$ NP-Complete
- Register allocation
 - Single basic block, no spilling $\Rightarrow$ linear time
 - Whole procedure is NP-Complete (graph colouring algorithm)

These three problems are tightly coupled!

However, conventional wisdom says we lose little by solving these problems independently.

How to solve these problems?

- Instruction selection
 - Use fix instruction template or some form of pattern matching
 - Assume enough registers
- Instruction scheduling
 - Within a block, list scheduling is “close” to optimal
 - Across blocks, build framework to apply list scheduling
- Register allocation
 - Start from virtual registers & map “enough” into k registers
 - With targeting, focus on “good” priority heuristic

Approximate solutions

Will be important to define good metrics for “close”, “good”, “enough”,

Code Generation

Register-based machine

- Most real physical machine are register-based
- Instruction operates on registers.
- The number of architecture register available to the compiler can vary from processor to processors.

The **first phase** of code generation usually **assumes an unlimited numbers of registers** (virtual registers).

Later phases (register allocator) **converts** these virtual registers to the **finite set** of available physical architectural registers (more on this in lecture on register allocation).

Generating Code for Register-Based Machine

The key code quality issue is holding values in registers

When can a value be safely allocated to a register?

- when only one name can reference its value
- pointers, structs & arrays all cause trouble

When should a value be allocated to a register?

- when it is both **safe** & **profitable**

Encoding this knowledge into the IR

- assign a virtual register to anything that goes into one
- load or store the others at each reference

Register allocation is key

All this relies on a strong register allocator.

Generating Code for Register-Based Machine

Memory

...	
addr_x	x
...	
addr_y	y
...	

Example: x+y

```
lw    $t0, addr_x # load content of memory at addr_x into $t0
lw    $t1, addr_y # load content of memory at addr_y into $t1
add   $t2, $t0, $t1
```

Exercise

Write down the list of equivalent assembly instructions for $4+x*y$

Exercise

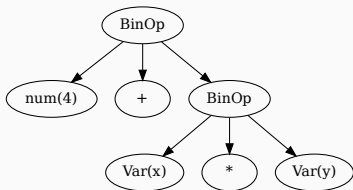
Assuming you have an instruction **addi** (**add** immediate), rewrite the previous example.

This illustrates the instruction selection problem (more on this in following lectures).

Code Generator for Arithmetic Expressions

Generating Code for Arithmetic Expressions

4 + x * y



Main idea: all expressions produce a value in a register.

Algorithm:

- Traverse AST depth first, post-order;
- After traversing a subtree, returns the register that contains the result of evaluating the subtree.

We will assume an unlimited number of registers is available to us (virtual registers).

Two helper functions:

- **newVirtualRegister** to obtain a unique register
- **emit** to produce an instruction

The following example shows how to implement each pattern-matched case to produce code that **evaluates** expressions.

Code Generator for Expressions

```
class ExprValCodeGen {  
  
    Register visit(Expr expr) {  
        return switch(expr) {  
            case ... ->  
            case ... ->  
        }  
    }  
}
```

Pattern-Matching expressions

IntLiteral

```
case IntLiteral it -> {  
    Register resReg = newVirtualRegister();  
    emit("li", resReg, it.value);  
    yield resReg;  
}
```

Var

```
case Var v -> {  
    Register resReg = newVirtualRegister();  
    emit("lw", resReg, v.label);  
    yield resReg;  
}
```

Here we assume our variables are all **integer** and global.

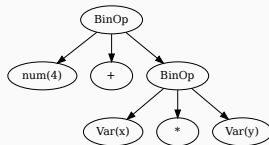
We will see how to deal with arrays/structs and stack allocated variables in another lecture.

Binary operators

```
case BinOp bo -> {
  Register lhsReg = visit(bo.lhs);
  Register rhsReg = visit(bo.rhs);
  Register resReg = newVirtualRegister();
  switch(bo.op) {
    case ADD:
      emit("add", resReg, lhsReg, rhsReg);
      break;
    case MUL:
      emit("mult", lhsReg, rhsReg);
      emit("mflo", resReg);
      break;
  }
  yield resReg;
}
```

Let see all this in action for $4+x*y$

```
case IntLiteral it -> {
    Register resReg = newVirtualRegister();
    emit("li", resReg, it.value);
    yield resReg;
}
case Var v -> {
    Register resReg = newVirtualRegister();
    emit("lw", resReg, v.label);
    yield resReg;
}
case BinOp bo -> {
    Register lhsReg = visit(bo.lhs);
    Register rhsReg = visit(bo.rhs);
    Register resReg = newVirtualRegister();
    switch(bo.op) {
        case ADD:
            emit("add", resReg, lhsReg, rhsReg);
            break;
        case MUL:
            emit("mult", lhsReg, rhsReg);
            emit("mflo", resReg);
            break;
    }
    yield resReg;
}
```



More about code generation:

- Logical and Relational Operators
- Control flow (if-then-else, loops, switch statement)