Compiling Techniques

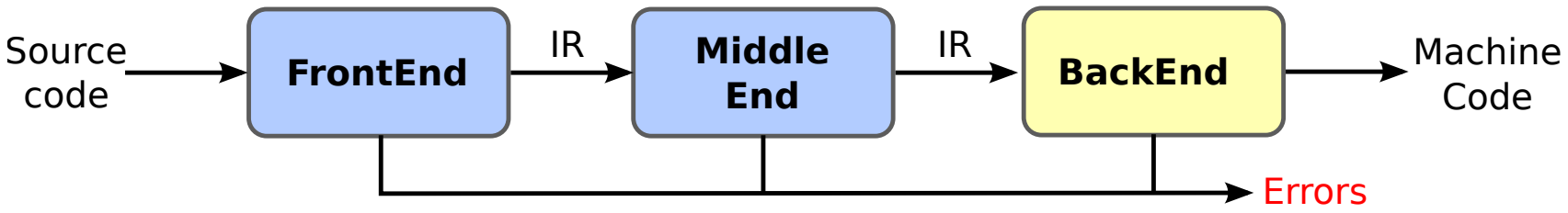
Lecture 11: Introduction to Code Generation

Christophe Dubach

28 Octoberr 2016

Table of contents

Overview



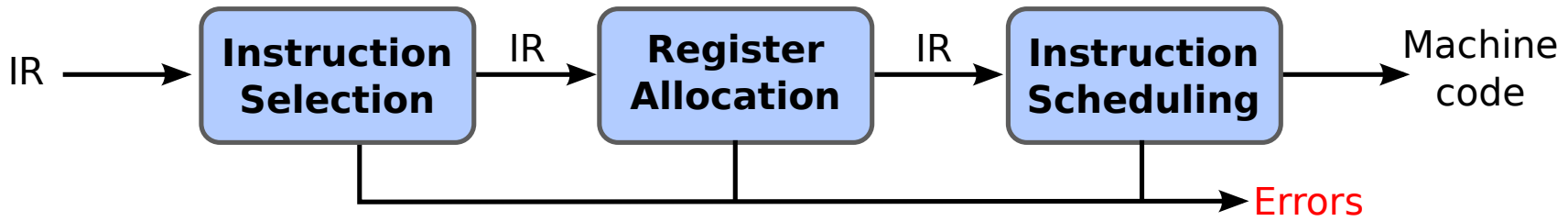
Front-end

- Lexer
- Parser
- AST builder
- Semantic Analyser

Middle-end

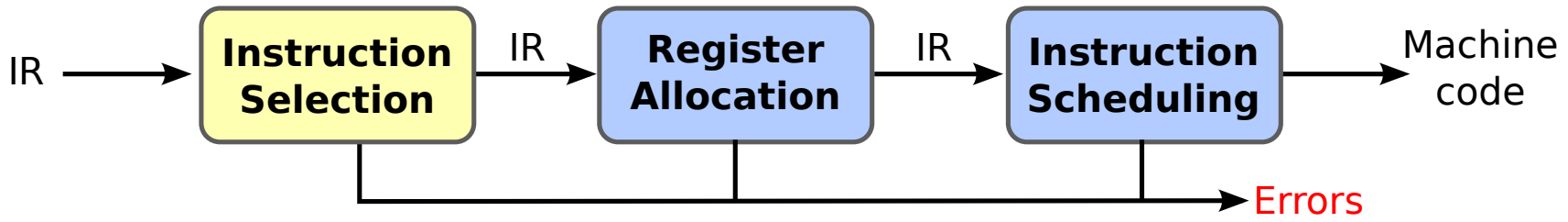
- Optimizations (Compiler Optimisations course)

The Back end



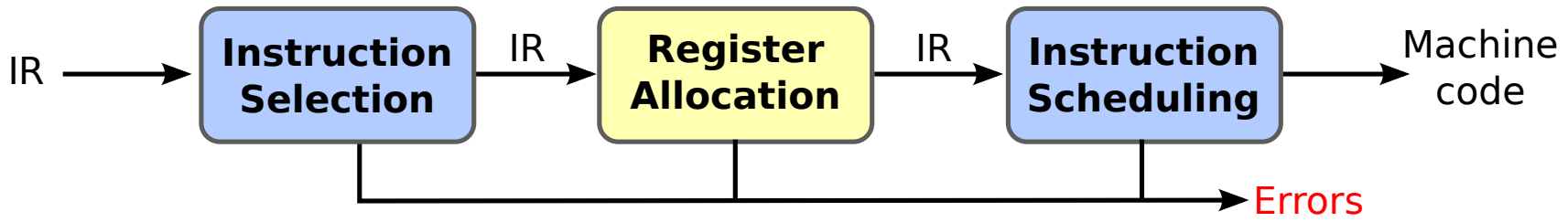
- Translate IR into target machine code
- Choose instructions to implement each IR operation
- Decide which value to keep in registers
- Ensure conformance with system interfaces
- Automation has been less successful in the back end

Instruction Selection



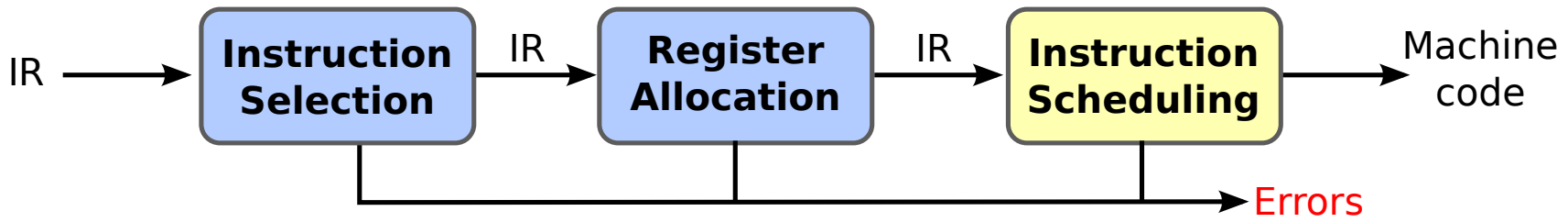
- Mapping the IR into assembly code (in our case AST to MIPS assembly)
- Assumes a fixed storage mapping & code shape
- Combining operations, using address modes

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.

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, 1 register size $\Rightarrow$ 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 some form of pattern matching
  - Assume enough registers or target "important" values
- 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
  - With targeting, focus on "good" priority heuristic

Approximate solutions

Will be important to define good metrics for "close", "good", "enough", ...

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 1 name can reference its value
  - Pointers, parameters, aggregate & 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 go into one
- load or store the others at each reference

Register allocation is key

All this relies on a strong register allocator.

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 register to the finite set of available physical architectural registers (more on this in lecture on register allocation).

Generating Code for Register-Based Machine

Memory

x
y

Example: x+y

```
loadI   @x     → r1 // load the address of x into r1
loadA   r1     → r2 // now value of x is in r2

loadI   @y     → r3 // load the address of y into r3
loadA   r3     → r4 // now value of y is in r4

add     r2,r4  → r5 // r5 contains x+y
```

Exercise

Write down the list of assembly instructions for x+(y*3)

Exercise

Assuming you have an instruction muli (**mul**tiply **i**mmediate), rewrite the previous example.

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

Visitor Implementation for binary operators

Binary operators

```
Register visitBinOp(BinOp bo) {
  Register lhsReg = bo.lhs.accept(this);
  Register rhsReg = bo.rhs.accept(this);
  Register result = nextRegister();
  switch(bo.op) {
    case ADD:
      emit(add lhsReg.id rhsReg.id → result.id);
      break;
    case MUL:
      emit(mul lhsReg.id rhsReg.id → result.id);
      break;
    ...           }
  freeRegister(lhsReg);
  freeRegister(rhsReg);
  return result;
}
```

Visitor Implementation for variables

x

```
loadI   @x    → r1 // load the address of x into r1
loadA   r1    → r2 // now value of x is in r2
```

Var

```
Register visitVar(Var v) {
  Register addrReg = nextRegister();
  Register result = nextRegister();
  emit(loadI v.address → addrReg.id);
  emit(loadA addrReg.id → result.id);
  freeRegister(addrReg);
  return result;
}
```

Visitor Implementation for integer literals

3

```
loadl   3       → r1
```

IntLiteral

```
Register visitIntLiteral(IntLiteral it) {
  Register result = nextRegister();
  emit(loadl it.value → result.id);
  return result;
}
```

Next lecture

Code Shape

- Conditions
- Function calls
- Loops
- If statement

Memory management

- Static/stack/heap allocation
- Data structure memory layout
- Register spilling