# Intermediate Code & Local Optimizations

## Lecture Outline

- What is "Intermediate code" ?
- Why do we need it?
- How to generate it?
- How to use it?
- Optimizations
  - Local optimizations



## Code Generation Summary

- We have so far discussed
  - Runtime organization
  - Simple stack machine code generation
  - Improvements to stack machine code generation
- Our compiler goes directly from the abstract syntax tree (AST) to assembly language...
  - ... and does not perform optimizations

Most real compilers use intermediate languages



## Why Intermediate Languages?

**ISSUE:** Reduce code complexity

- Multiple front-ends
  - gcc can handle C, C++, Java, Fortran, Ada, ...
  - each front-end translates source to the same generic language (called GENERIC)
- Multiple back-ends
  - gcc can generate machine code for various target architectures: x86, x86_64, SPARC, ARM, …
- **One Icode to bridge them!**
  - Do most optimization on intermediate representation before emitting machine code

## Why Intermediate Languages?

**ISSUE:** When to perform optimizations

- On abstract syntax trees
  - Pro: Machine independent
  - Con: Too high level
- On assembly language
  - Pro: Exposes most optimization opportunities
  - Con: Machine dependent
  - Con: Must re-implement optimizations when re-targeting
- On an intermediate language
  - Pro: Exposes optimization opportunities
  - Pro: Machine independent



## Kinds of Intermediate Languages

High-level intermediate representations:

- closer to the source language (structs, arrays)
- easy to generate from the input program
- code optimizations may not be straightforward

Low-level intermediate representations:

- closer to target machine: GCC's RTL, 3-address code
- easy to generate code from
- generation from input program may require effort

"Mid"-level intermediate representations:

- programming language and target independent
- Java bytecode, Microsoft CIL, LLVM IR, ...



## Intermediate Code Languages: Design Issues

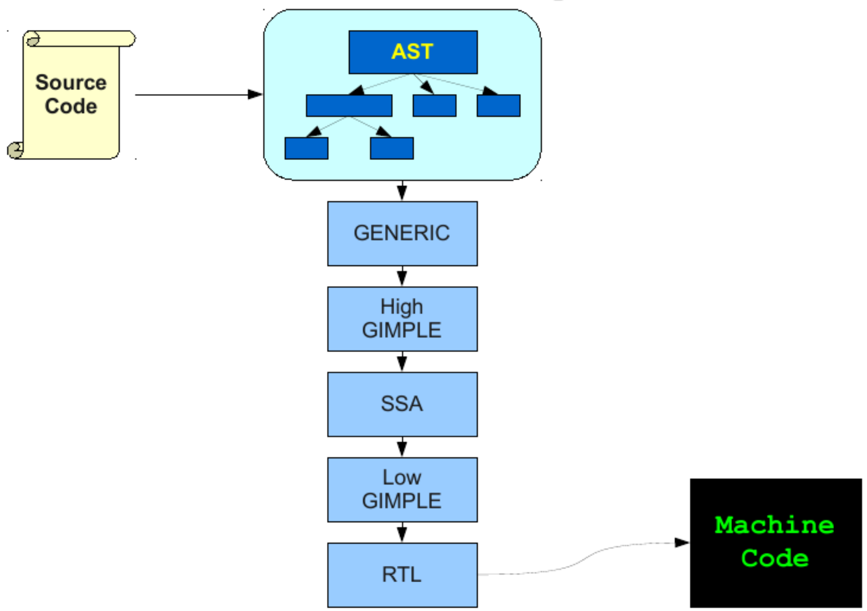
- Designing a good ICode language is not trivial
- The set of operators in ICode must be rich enough to allow the implementation of source language operations
- ICode operations that are closely tied to a particular machine or architecture, make retargeting harder
- A small set of operations
  - may lead to long instruction sequences for some source language constructs,
  - but on the other hand makes retargeting easier



## Intermediate Languages

- Each compiler uses its own intermediate language
- Nowadays, usually an intermediate language is a high-level assembly language
  - Uses register names, but has an unlimited number
  - Uses control structures like assembly language
  - Uses opcodes but some are higher level
    - E.g., push translates to several assembly instructions
    - Most opcodes correspond directly to assembly opcodes

## Architecture of gcc






## Three-Address Intermediate Code

- Each instruction is of the form

  $x := y \text{ op } z$

  - $y$ and $z$ can only be registers or constants
  - Just like assembly
- Common form of intermediate code
- The expression $x + y * z$ gets translated as

  $t_1 := y * z$

  $t_2 := x + t_1$

  - temporary names are made up for internal nodes
  - each sub-expression has a "home"



## Generating Intermediate Code

- Similar to assembly code generation
- Major difference
  - Use any number of IL registers to hold intermediate results

**Example:** `if (x + 2 > 3 * (y - 1) + 42) then z := 0;`

$t_1 := x + 2$
$t_2 := y - 1$
$t_3 := 3 * t_2$
$t_4 := t_3 + 42$
if $t_1$ =< $t_4$ goto L
$z := 0$
L:



## Generating Intermediate Code (Cont.)

- igen(e, t) function generates code to compute the value of e in register t
- Example:

  igen($e_1 + e_2$, t) =
  igen($e_1$, $t_1$) ($t_1$ is a fresh register)
  igen($e_2$, $t_2$) ($t_2$ is a fresh register)
  $t := t_1 + t_2$
- Unlimited number of registers

  $\Rightarrow$ simple code generation

## From ICode to Machine Code

This is almost a macro expansion process

| ICode | MIPS assembly code |
|---|---|
| x := A[i] | load i into *r1*<br>**la** *r2*, A<br>**add** *r2*, *r2*, *r1*<br>**lw** *r2*, (*r2*)<br>**sw** *r2*, x |
| x := y + z | load y into *r1*<br>load z into *r2*<br>**add** *r3*, *r1*, *r2*<br>**sw** *r3*, x |
| if x >= y goto L | load x into *r1*<br>load y into *r2*<br>**bge** *r1*, *r2*, L |



## Basic Blocks

- A *basic block* is a maximal sequence of instructions with:
  - no labels (except at the first instruction), and
  - no jumps (except in the last instruction)
- Idea:
  - Cannot jump into a basic block (except at beginning)
  - Cannot jump out of a basic block (except at end)
  - Each instruction in a basic block is executed after all the preceding instructions have been executed



## Basic Block Example

Consider the basic block

```
L:                  (1)
  t := 2 * x        (2)
  w := t + x        (3)
  if w > 0 goto L'  (4)
```

- No way for (3) to be executed without (2) having been executed right before
  - We can change (3) to w := 3 * x
  - Can we eliminate (2) as well?



## Identifying Basic Blocks

- Determine the set of *leaders*, i.e., the first instruction of each basic block:
  - The first instruction of a function is a leader
  - Any instruction that is a target of a branch is a leader
  - Any instruction immediately following a (conditional or unconditional) branch is a leader
- For each leader, its basic block consists of itself and all instructions up to, but not including, the next leader (or end of function)

## Control-Flow Graphs

A *control-flow graph* is a directed graph with

- Basic blocks as nodes
- An edge from block A to block B if the execution can flow from the last instruction in A to the first instruction in B

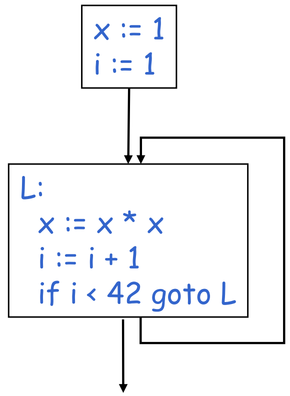
  E.g., the last instruction in A is goto $L_B$

  E.g., the execution can fall-through from block A to block B

Frequently abbreviated as CFGs



## Control-Flow Graphs: Example




- The body of a function (or method or procedure) can be represented as a control-flow graph
- There is one initial node
- All "return" nodes are terminal



## Constructing the Control Flow Graph

- First identify the basic blocks of the function
- There is a directed edge between block $B_1$ to block $B_2$ if
  - there is a (conditional or unconditional) jump from the last instruction of $B_1$ to the first instruction of $B_2$ or
  - $B_2$ immediately follows $B_1$ in the textual order of the program, and $B_1$ does not end in an unconditional jump.



## Optimization Overview

- Compiler "optimizations" seek to improve a program's utilization of some resource
  - Execution time (most often)
  - Code size
  - Network messages sent
  - (Battery) power used, etc.
- Optimization should not alter what the program computes
  - The answer must still be the same
  - Observable behavior must be the same
    - this typically also includes termination behavior

## A Classification of Optimizations

For languages like C there are three granularities of optimizations

(1) Local optimizations
- Apply to a basic block in isolation

(2) Global optimizations
- Apply to a control-flow graph (function body) in isolation

(3) Inter-procedural optimizations
- Apply across method boundaries

Most compilers do (1), many do (2) and very few do (3)



## Cost of Optimizations

- In practice, a conscious decision is made **not** to implement the fanciest optimizations
- Why?
  - Some optimizations are hard to implement
  - Some optimizations are costly in terms of compilation time
  - Some optimizations are hard to get completely right
  - The fancy optimizations are often hard, costly, and difficult to get completely correct
- Goal: maximum improvement with minimum cost

## Local Optimizations

- The simplest form of optimizations
- No need to analyze the whole procedure body
  - Just the basic block in question
- Example: algebraic simplification



## Algebraic Simplification

- Some statements can be deleted

  x := x + 0
  
  x := x * 1
- Some statements can be simplified

  x := x * 0 ⇒ x := 0
  
  y := y ** 2 ⇒ y := y * y
  
  x := x * 8 ⇒ x := x << 3
  
  x := x * 15 ⇒ t := x << 4; x := t - x
  
  (on some machines << is faster than *; but not on all!)

## Constant Folding

- Operations on constants can be computed at compile time
- In general, if there is a statement

  x := y op z

  - And y and z are constants
  - Then y op z can be computed at compile time
- Example: x := 20 + 22 ⇒ x := 42
- Example: if 42 < 17 goto L can be deleted



## Flow of Control Optimizations

- Eliminating unreachable code:
  - Code that is unreachable in the control-flow graph
  - Basic blocks that are not the target of any jump or "fall through" from a conditional
  - Such basic blocks can be eliminated
- Why/how would such basic blocks occur?
- Removing unreachable code makes the program smaller
  - And sometimes also faster
    - Due to memory cache effects (increased spatial locality)



## Single Assignment Form

- Some optimizations are simplified if each register occurs only once on the left-hand side of an assignment
- Intermediate code can be rewritten to be in *single assignment* form

| | | |
|---|---|---|
| x := z + y | | x := z + y |
| a := x | ⇒ | a := x |
| x := 2 * x | | b := 2 * x |

(b is a fresh temporary)

- More complicated in general, due to control flow (e.g. loops)
  - *Static single assignment (SSA)* form



## Common Subexpression Elimination

- Assume
  - A basic block is in single assignment form
  - A definition x := is the first use of x in a block
- All assignments with same RHS compute the same value
- Example:

| | | |
|---|---|---|
| x := y + z | | x := y + z |
| … | ⇒ | … |
| w := y + z | | w := x |

(the values of x, y, and z do not change in the … code)

## Copy Propagation

- If w := x appears in a block, all subsequent uses of w can be replaced with uses of x

- Example:

```
b := z + y              b := z + y
a := b        ⇒         a := b
x := 2 * a              x := 2 * b
```

- This does not make the program smaller or faster but might enable other optimizations
  - Constant folding
  - Dead code elimination



## Copy Propagation and Constant Folding

- Example:

```
a := 5                  a := 5
x := 2 * a      ⇒       x := 10
y := x + 6              y := 16
t := x * y              t := 160
```



## Dead Code Elimination

If

w := RHS appears in a basic block

w does not appear anywhere else in the program

Then

the statement w := RHS is dead and can be eliminated

- Dead = does not contribute to the program's result

Example: (a is not used anywhere else)

```
x := z + y         x := z + y         x := z + y
a := x       ⇒     a := x       ⇒     b := 2 * x
x := 2 * x         b := 2 * x
```



## Applying Local Optimizations

- Each local optimization does very little by itself

- Typically optimizations interact
  - Performing one optimization enables another

- Optimizing compilers repeatedly perform optimizations until no improvement is possible
  - The optimizer can also be stopped at any time to limit the compilation time

## An Example

Initial code:

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

assume that only f and g are used in the rest of program



## An Example

Algebraic simplification:

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```



## An Example

Algebraic simplification:

```
a := x * x
b := 3
c := x
d := c * c
e := b << 1
f := a + d
g := e * f
```



## An Example

Copy and constant propagation:

```
a := x * x
b := 3
c := x
d := c * c
e := b << 1
f := a + d
g := e * f
```

## An Example

Copy and constant propagation:

```
a := x * x
b := 3
c := x
d := x * x
e := 3 << 1
f := a + d
g := e * f
```



## An Example

Constant folding:

```
a := x * x
b := 3
c := x
d := x * x
e := 3 << 1
f := a + d
g := e * f
```



## An Example

Constant folding:

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```



## An Example

Common subexpression elimination:

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

## An Example

Common subexpression elimination:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```



## An Example

Copy and constant propagation:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```



## An Example

Copy and constant propagation:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f
```



## An Example

Dead code elimination:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f
```

## An Example

Dead code elimination:

a := x * x

f := a + a
g := 6 * f

This is the final form



## Peephole Optimizations on Assembly Code

- The optimizations presented before work on intermediate code
  - They are target independent
  - But they can be applied on assembly language also

*Peephole optimization* is an effective technique for improving assembly code

- The "peephole" is a short sequence of (usually contiguous) instructions
- The optimizer replaces the sequence with another equivalent (but faster) one



## Implementing Peephole Optimizations

- Write peephole optimizations as replacement rules

  $i_1, \ldots, i_n \rightarrow j_1, \ldots, j_m$

  where the RHS is the improved version of the LHS
- Example:

  move \$a \$b, move \$b \$a → move \$a \$b
  - Works if move \$b \$a is not the target of a jump
- Another example:

  addiu \$a \$a i, addiu \$a \$a j → addiu \$a \$a i+j



## Peephole Optimizations

- Redundant instruction elimination, e.g.:

```
  . . .              . . .
  goto L      ⇒    L:
L:                   . . .
  . . .
```

- Flow of control optimizations, e.g.:

```
    . . .                . . .
    goto L1      ⇒      goto L2
    . . .                . . .
L1: goto L2         L1: goto L2
    . . .                . . .
```

## Peephole Optimizations (Cont.)

- Many (but not all) of the basic block optimizations can be cast as peephole optimizations
  - Example: addiu \$a \$b 0 $\rightarrow$ move \$a \$b
  - Example: move \$a \$a $\rightarrow$
  - These two together eliminate addiu \$a \$a 0

- Just like for local optimizations, peephole optimizations need to be applied repeatedly to get maximum effect



## Concluding Remarks

- Multiple front-ends, multiple back-ends via intermediate codes

- Intermediate code is helpful for many optimizations

- Many simple optimizations can still be applied on assembly language

- Next time: *global* optimizations