Target code generation

Input: program as intermediate representation

- three-address code
- ASTs

Output: program as target code

- absolute binary (machine) code
- relocatable binary code
- assembly code
- C

Requirement: must generate correct code

Differences in generated code quality & time to generate

Task of code generator

Bridge the gap between:

- intermediate code (machine independent)
- target code (machine dependent)

Instruction selection

- for each IR instruction (or sequence),
  select target language instruction (or sequence)

Register allocation

- for each IR variable,
  select target language register/stack location

Code scheduling

- decide the order of the target language instructions

Instruction selection

Given one or more IR instructions,
pick "best" sequence of target machine instructions
with same semantics

"best" = fastest, usually fewest

Difficulty depends on nature of target instruction set

- **CISC: hard**
  - lots of alternative instructions with similar semantics
  - lots of tradeoffs among speed, size
  - often not completely orthogonal
- **RISC: easy**
  - usually only one choice
  - closely resembles IR instructions
- **C: easy if C is appropriate for the desired semantics**
  - ex: many high-level languages require check for integer overflow

Example

1 IR instruction can require several target instructions

IR code:

```
t3 := t1 + t2
```

Target code (MIPS):

```
add $3,$1,$2
```

Target code (SPARC):

```
add %1,%2,%3
```

Target code (68k):

```
mov.l d1,d3
add.l d2,d3
```

Another example

Can have choices of which instruction(s) to select

IR code:

```
t1 := t1 + 1
```

Target code (MIPS):

```
addi $1,$1,1
```

Target code (SPARC):

```
add %1,1,%1
```

Target code (68k):

```
add.l #1,d1
```

or

```
inc.l d1
```


Yet another example

Several IR code instructions can combine to 1 target instruction
⇒ **hard!**

IR code:

```
// push x onto stack
sp  := sp - 4
*sp := t1
```

Target code (MIPS):

```
sub $sp,$sp,4
sw $1,0($sp)
```

Target code (SPARC):

```
sub %sp,4,%sp
st %1,[%sp]
```

Target code (68k):

```
mov.l d1,-(sp)
```

A final example

Source code:

```
a++;// "a" is a global variable
```

IR code:

```
t1 := a
t1 := t1 + 1
a := t1
```

Target code:

```
lw $1, 0(address of a)
add $1, $1, 1
sw $1, 0(address of a)
```


Instruction selection in PL/0

Do very simple instruction selection,
as part of generating code for AST node

Interface to target machine: `assembler` class

- function for each kind of target instruction
- hides details of assembly format, etc.

Register allocation

IR uses unlimited temporary variables

- makes intermediate code generation easy
- makes intermediate code machine-independent

Target machine has fixed & few resources for holding values

Registers *much* faster than memory

Consequences:

- should try to keep values in registers if possible
- want to choose most frequently accessed values
- want to choose values accessed in a small program range
- must free registers when no longer needed
- must be able to handle out-of-registers condition
  $\Rightarrow$ **spill** some registers to home stack locations
- must interact with instruction selection on CISCs
  $\Rightarrow$ makes both jobs harder

Classes of registers

What registers can the allocator use?

Dedicated registers

- claimed by instruction set architecture (hardware) for a special purpose
  - register hardwired to 0, SP, return address, ...
- claimed by calling convention (software)
  - FP, argument registers 1-4, ...
  - not easily available for storing locals
- examples
  - MIPS, SPARC: 32 registers, but not all are general purpose
  - 68k: 16 registers, divided into data and address regs
  - x86: 4 data registers, plus 12 special-purpose registers

Scratch registers

- couple of registers kept around for temporary values
  - e.g. loading a spilled value from memory in order to operate on it

Allocatable registers are the ones that are left!

Classes of variables

What variables can the register allocator try to put in registers?

Temporary variables: easy to allocate

- defined & used exactly once, during expression evaluation
  ⇒ allocator can free up register after used
- usually not too many in use at one time
  ⇒ less likely to run out of registers

Local variables: hard, but doable

- more of these & their **lifetimes** are longer
  ⇒ need to make decision about which variables get registers
- need to free a register when its value is not longer needed
  ⇒ need to determine the **last use** of the variable
- what about assignments to a local through pointer?
- what about debugging?

Global variables: really hard

- have to analyze whole program

Pl/0 register allocator

Keep set of allocated registers as codegen proceeds

- `RegisterBank` class in PL/0

During codegen, allocate one from set

- `Reg temp = rb->getNew();`
- modify register bank to record that `temp` is taken
- what if no registers available?

When done with register, release it

- `rb->free(temp);`
- modify register bank to record that `temp` is free

“Real” register allocators

Register allocation

(1) Decide **which variables** should go into registers

- how frequently they are used
  allocate most frequently used variables to regs first
- how long they are used
- if two variables don’t overlap, then give to same register

(2) Decide how long they should stay there (**register spilling**)

Register assignment

(3) Decide which variable goes into **which register**

Technique is call **register coloring**

Why it’s difficult:

- optimal register assignment is NP-complete
- some registers can only be used for special purposes
- some registers must be used in consecutive pairs

Code scheduling

Iterate for each basic block:

(1) determine all instructions that are ready to execute

- operands have been computed

```
add $3, $2, $2 (assume ready)
sub $6, $5, $4 (assume ready)
mult $7, $3, $6 (not ready))
```

(2) put them in the **ready list**

(3) pick one on the **critical path**

- example heuristic: instruction that has the longest chain of dependent instructions

```
add $1, $7, $8
sub $2, $1, $8
mult $3, $2, $8
add $4, $3, $8
sub $5, $3, $8
add $6, $7, $8
```

- example heuristic: instruction with the longest latency

List scheduling

Some codegen routines

```
Reg IntLiteral::codegen(Scope* s, RegBank* rb) {
   Reg dest = rb->getNew();
   TheAssembler->loadImmediate(dest, _value);
   return dest;
}

Reg BinOp::codegen(Scope* s, RegBank* rb) {
   Reg r1 = _left->codegen(s, rb);
   Reg r2 = _right->codegen(s, rb);
   rb->free(r1);
   rb->free(r2);
   Reg dest = rb->getNew();
   TheAssembler->arith(_op, dest, r1, r2);
   return dest;
}

void AssignStmt::codegen(Scope* s, RegBank* rb) {
   Reg result = _expr->codegen(s, rb);
   int offset;
   Reg base = _lvalue->codegen_addr(s, rb, offset);
   TheAssembler->store(result, base, offset);
   rb->free(base);
   rb->free(result);
}
```


An example using PI/0

Source code:

```
var x;
...
x := x + 2 * (x - 1);
```

Function call codegen routine

```
Reg FuncCall::codegen(Scope* s, RegBank* rb) {
   // evaluate & push arguments
   foreach arg, right to left {
      Reg a;
      if (pass by value) {
         a = arg->codegen(s, rb);
      } else {
         // pass by reference
         int offset;
         Reg base = arg->codegen_addr(s, rb, offset);
         Reg o = rb->getNew();
         TheAssembler->loadImmediate(o, offset);
         rb->free(base);
         rb->free(o);
         a = rb->getNew();
         TheAssembler->arith(PLUS, a, base, o);
      }
      TheAssembler->push(SP, a);
      rb->free(a);
   }

   ...
```



```
   ...

   // evaluate & push static link
   Reg link = s->getFPOf(enclosingScope, rb);
   TheAssembler->push(SP, link);
   rb->free(link);

   // save any allocated regs across call
   rb->saveRegs(s);

   // call
   TheAssembler->call(_ident);

   // restore saved regs
   rb->restoreRegs(s);

   // pop off args & static link
   TheAssembler->popMultiple(SP,(#args+1)*sizeof(int));

   // allocate temp reg for result of call
   Reg dest = rb->getNew();
   TheAssembler->move(dest, RET0);

   // return result
   return dest;
}
```

Another example

Source code:

```
x := y + 4;
z := x * 8;
```