Target Code Generation

Input: intermediate language (IL)

Output: target language program

Target languages:

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

Target code generation must bridge the gap

The gap, if target is machine code

IL	Machine Code
global variables	global static memory
unbounded number of interchangeable local variables	fixed number of registers, of various incompatible kinds, plus unbounded number of stack locations
built-in parameter passing & result returning	calling conventions defining where arguments & results are stored and which registers may be overwritten by callee
statements	machine instructions
statements can have arbitrary subexpression trees	statements have restricted operand addressing modes
conditional branches based on integers representing boolean values	conditional branches based on condition codes (maybe)

Tasks of code generator

Register allocation

- for each IL variable,
  select register/stack location/global memory location(s) to hold it
  - can depend on type of data, which operations manipulate it

Stack frame layout

- compute layout of each function's stack frame

Instruction selection

- for each IL instruction (sequence),
  select target language instruction (sequence)
  - includes operand addressing mode selection

Can have complex interactions

- instruction selection depends on where operands are allocated
- some IL variables may not need a register, depending on the instructions & addressing modes that are selected

Register allocation

Intermediate language uses unlimited temporary variables

- makes ICG easy

Target machine has fixed resources for representing "locals" plus other internal things such as the stack pointer

- MIPS, SPARC: 31 registers + 1 always-zero register
- 68k: 16 registers, divided into data and address regs
- x86: 8 word-sized integer registers (with a number of instruction-specific restrictions on use) plus a stack of floating-point data manipulated only indirectly

Registers *much* faster than memory

Must use registers in load/store RISC machines

Consequences:

- should try to keep values in registers if possible
- must reuse registers for many temp vars
  ⇒ free registers when no longer needed
- must be able to handle more variables than registers
  ⇒ **spill** some variables from register to stack locations
- interacts with instruction selection, on CISCs
  ⇒ a real pain

Classes of registers

What registers can the allocator use?

Fixed/dedicated registers

- stack pointer, frame pointer, return address, ...
- claimed by machine architecture, calling convention, or internal convention for special purpose
- not easily available for storing locals

Scratch registers

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

Allocatable registers

- remaining registers free for register allocator to exploit

Some registers may be overwritten by called procedures
⇒ caller must save them across calls, if allocated

- caller-saved registers, vs. callee-saved registers

Classes of variables

What variables can the allocator try to put in registers?

Temporary variables: easy to allocate

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

Local variables: hard, but doable

- need to determine **last use** of variable in order to free reg
- can easily run out of registers
  ⇒ need to make decision about which variables get regs
- what about assignments to local through pointer?
- what about debugger?

Global variables:
really hard, but doable as a research project

Register allocation in MiniJava

Don't do any analysis to find last use of local variables
⇒ allocate all local variables to stack locations

- each read of the local variable translated into a load from its stack location
- each assignment to a local variable translated into a store into its stack location

Each IL expression has exactly one use
⇒ allocate result value of IL expression to register

- maintain a set of allocated registers
- allocate an unallocated register for each expr result
- free register when done with expr result
- not too many IL expressions "active" at a time
  ⇒ unlikely to run out of registers, even on x86
  - the MiniJava compiler will die if it runs out of registers for IL expressions :(

X86 register allocator:

- `eax`, `ebx`, `ecx`, `edx`: allocatable, caller-save registers
- `esi`, `edi`: scratch registers
- `esp`: stack pointer; `ebp`: frame pointer
- floating-point stack, for `double` values

Stack frame layout

Need space for:

- formals
- local variables
- return address
- (maybe) dynamic link (ptr to calling stack frame)
- (maybe) static link (ptr to lexically-enclosing stack frame)
- other run-time data (e.g. caller-saved registers)

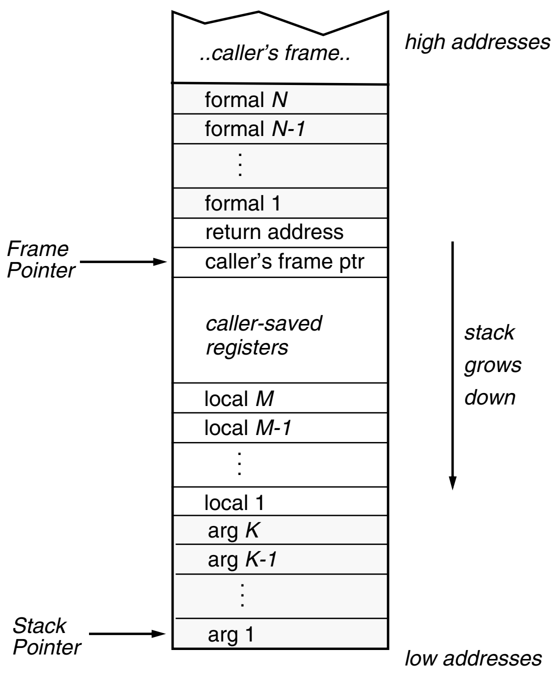
Assign dedicated register(s) to support access to stack frames

- frame pointer (FP): ptr to beginning of stack frame (fixed)
- stack pointer (SP): ptr to end of stack (can move)

Key property:
all data in stack frame is at **fixed, statically computed** offset from FP

- easy to generate fast code to access data in stack frame, even lexically enclosing stack frames
- compute all offsets solely from symbol tables

MiniJava/X86 stack frame layout



Calling conventions

Need to define responsibilities of caller and callee
in setting up, tearing down stack frame

Only caller can do some things
Only callee can do other things
Some things could be done by both

Need a protocol

X86 calling sequence

Caller:

- evaluates actual arguments, pushes them on stack
  - in right-to-left order, to support C varargs
  - alternative: 1st *k* arguments in registers
- saves caller-save registers in caller's stack
- executes call instruction
  - return address pushed onto the stack by hardware

Callee:

- pushes caller's frame pointer on stack
  - the dynamic link
- sets up callee's frame pointer
- allocates space for locals, caller-saved registers
  - order doesn't matter to calling convention
- starts running callee's code...

X86 return sequence

Callee:

- puts returned value in right place
  (`eax` or floating-point stack)
- deallocates space for locals, caller-saved regs
- pops caller's frame pointer from stack
- pops return address from stack and jumps to it

Caller:

- deallocates space for args
- restores caller-save registers from caller's stack
- continues execution in caller after call...

Instruction selection

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

"best" = fastest, shortest, lowest power, ...

Difficulty depends on nature of target instruction set

- RISC: easy
  - usually only one way to do something
  - closely resembles IL instructions
- CISC: hard to do well
  - lots of alternative instructions with similar semantics
  - lots of possible operand addressing modes
  - lots of tradeoffs among speed, size
  - simple RISC-like translation may not be very efficient
- C: easy, as long as C appropriate for desired semantics
  - can leave optimizations to C compiler

Correctness a big issue, particularly if codegen complex

Example

IL 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
```

Target code (x86):

```
movl %eax,%ecx
addl %ebx,%ecx
```

1 IL instruction may expand to several target instructions

Another example

IL code:

```
t1 = t1 + 1;
```

Target code (MIPS):

```
add $1,$1,1
```

Target code (SPARC):

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

Target code (68k):

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

or

```
inc.l d1
```

Target code (x86):

```
addl $1,%eax
```

or

```
incl %eax
```

Can have choices

- it's a pain to have choices; requires making decisions

Yet another example

IL 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+0]
```

Target code (68k):

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

Target code (x86):

```
pushl %eax
```

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

Instruction selection in MiniJava

Expand each IL statement into some number of target machine instructions

- don't attempt to combine IL statements together

In `Target` subdirectory:

```
abstract class Target
abstract class Location
```

- defines abstract methods for emitting machine code for statements, e.g. `emitVarAssign`, `emitFieldAssign`, `emitBranchTrue`
- defines abstract methods for emitting machine code for statements, e.g. `emitVarRead`, `emitFieldRead`, `emitIntMul`
  - return `Location` representing where result is allocated

IL statement and expression classes invoke these operations to generate their machine code

- each IL statement and expression has a corresponding `emit` operation on `Target` class
  - a version of the visitor design pattern

Details of target machines are hidden from IL and the rest of the compiler behind the `Target` and `Location` interfaces

Implementing **`Target`** and **`Location`**

A particular target machine provides a concrete subclass of `Target`, plus concrete subclasses of `Location` as needed

E.g. in `Target/X86` subdirectory:

`class X86Target extends Target`

`class X86Register extends Location`

- for expressions whose results are in (integer) registers

`class X86FloatingPointStack extends Location`

- for expressions whose results are pushed on the floating-point stack

`class X86ComparisonResult extends Location`

- for boolean expressions whose results are in condition codes

Could define `Target/MIPS`, `Target/C`, etc.