# Target Code Generation

David Notkin
Autumn 2008

## Schedule

- Project D: intermediate code generation
  - Due: December 3
- Due December 10, 5PM (available Monday 11/17)
  - Project E: target code generation [MiniJava++]
  - Written assignment [MiniJava--]
- Final on December 11 (one hour, backend focused)



## Target Code Generation

- Input: intermediate language (IL)
- Output: target language program
- Target languages include
  - absolute binary (machine) code
  - relocatable binary code
  - assembly code
  - C
- Using the generated intermediate code, convert to instructions and memory characteristics of the target machine
  - Target code generation must bridge the gap

## Gap: 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 | instructions have restricted operand addressing |
| 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 based on it's type and lifetime
- Stack frame layout
- Instruction selection
  - for each IL instruction (sequence), select target language instruction (sequence); must consider operand addressing mode selection

## These tasks interact

- 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
- Stack frame layout may depend on instruction set
- …

## Register Allocation

- Intermediate language uses unlimited temporary variables – this intentionally makes ICG easy
- Target machine has fixed resources for representing locals plus other internal things such as stack pointer
  - MIPS, SPARC: 31 registers + 1 always-zero register
  - 68k: 16 registers, divided into data and address
  - x86: 8 word-sized integer registers (with instruction-specific restrictions on use) plus a stack of floating-point data
- Registers are 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, implies free registers after use
- Must handle more variables than registers, implies spill
- Interacts with instruction selection on CISC, implies it's a real pain



## Classes of Registers

- Fixed/dedicated registers
  - Stack pointer, frame pointer, return address, ...
  - Claimed by machine architecture, calling convention, or internal convention for special purpose
  - Some registers may be overwritten by called procedures so caller must save them across calls, if allocated
    - caller-saved registers vs. callee-saved registers
- Scratch registers
  - registers kept around for temps (e.g., loading a spilled value from memory to operate on it)
- Free registers
  - remaining registers free for register allocator to use

## Classes of Variables

- What variables can the allocator put in registers?
- Temporary variables: easy to allocate
  - Defined and used exactly once, during expression evaluation, implies allocator can free up register when done
  - Usually not too many in use at one time implies less likely to run out of registers
- Local variables: hard, but doable
  - need to determine last use of variable to free register
  - can easily run out of registers so must make decision about which variables get register allocation
  - what about assignments to local through pointer?
  - what about debugger?
- Global variables: really hard, but doable as a research project

## Register Allocation in MiniJava

- Allocate all local variables to stack locations
  - No need for analysis to find last use of local variables
  - Each read of the local variable translated into a load from stack
  - Each assignment to a local translated to a store into its stack location

## Register Allocation in MiniJava

- Each IL expression has exactly one use so can allocate result value of IL expression to register
  - Maintain set of allocated registers
  - Allocate an unallocated register for each expression result
  - Free register when done with expression result
  - Not too many IL expressions "active" at a time implies unlikely to run out of registers, even on x86
    - MiniJava compiler dies if it runs out of registers for IL expressions

## Register Allocation in MiniJava

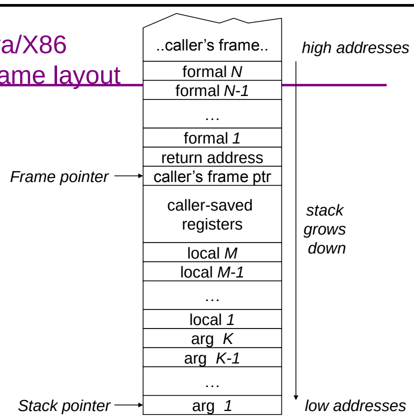
- X86 register allocator
  - `eax, ebx, ecx, edx`: allocatable, caller-save registers
  - `esi, edi`: scratch registers
  - `esp`: stack pointer; `ebp`: frame pointer
  - floating-point stack, for `double` values



## 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
  - FP: ptr to beginning of stack frame (fixed)
  - SP: ptr to end of stack (can move)
- All data in stack frame is at fixed, statically computed offset from FP
  - Compute all offsets solely from symbol tables

## MiniJava/X86 stack frame layout



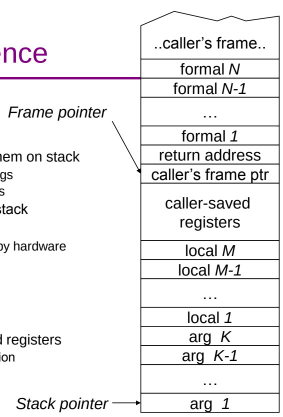

## Calling Conventions

- 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
- So, need a protocol – just like in the IL

## 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

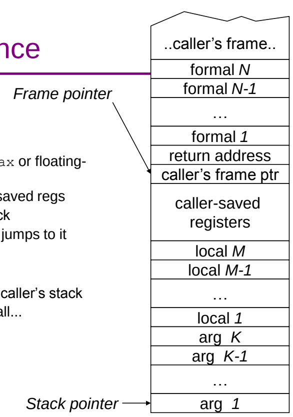
Callee:
- pushes caller's frame pointer on stack
  - the dynamic link
- sets up callee's frame pointer
- allocates space for locals, caller-saved registers
  - order doesn't matter to calling convention
- starts running callee's code...




## 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-saved 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, ...
- Correctness a big issue, particularly if codegen is complex

## Codegen difficulty depends on target

- 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



## 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
```

One 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: 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 instructions can combine to one target instruction

## 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 classes `Target` and `Location`
  - define abstract methods for emitting machine code for statements and data access: `emitVarAssign, emitFieldAssign, emitBranchTrue, 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 the `Target` class
- 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
- For example, 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.

## An Example X86 emit method

```
Location emitIntConstant(int value) {
   Location result_location =
      allocateReg(ILType.intILType());
       emitOp("movl",
        intOperand(value),
        regOperand(result_location));
   return result_location;
}
Location allocateReg(ILType):
```
*allocate a new register to hold a value of the given type*
```
void emitOp(String opname, String arg1, ...):
```
*emit assembly code*
```
String intOperand(int):
```
*return the asm syntax for an int constant operand*
```
String regOperand(Location):
```
*return the asm syntax for a reference to a register*

## An Example X86 Target emit method

- What x86 code to generate for **`arg1 +.int arg2`**?
- x86 int add instruction: **`addl %arg, %dest`**
  - semantics: **`%dest = %dest + %arg;`**

- emit **`arg1`** into **`register%arg1`**
- emit **`arg2`** into **`register%arg2`**
- then?

## An Example X86 Target emit method

```
Location emit IntAdd(ILExprarg1,ILExprarg2) {
   Location arg1_location=arg1.codegen(this);
   Location arg2_location=arg2.codegen(this);
   emitOp("addl",
        regOperand(arg2_location),
        regOperand(arg1_location));
   deallocateReg(arg2_location);
   return arg1_location;
}
void deallocateReg(Location):
```
*deallocate register,*
*make available for use by later instructions*

## An Example X86 Target emit method

- What x86 code to generate for var read or assignment?
- Need to access var's home stack location
- x86 stack reference operand: **`%ebp(offset)`**
  - semantics: **`*(%ebp + offset);`**
  - **`%ebp = `*`frame pointer`***

## An Example X86 Target emit method

```
Location emitVarRead(ILVarDecl var) {
  int var_offset = var.getByteOffset(this);
  ILType var_type = var.getType();
  Location result_location =
      allocateReg(var_type);
  emitOp("movl",
         ptrOffsetOperand(FP, var_offset),
         regOperand(result_location));
  return result_location;
}
```

## Continued

```
void emitVarAssign(ILVarDecl var,
                    Location rhs_location) {
  int var_offset = var.getByteOffset(this);
  emitOp("movl",
         regOperand(rhs_location),
         ptrOffsetOperand(FP, var_offset));
}

String ptrOffsetOperand(Location, int):
```

*return the asm syntax for a reference to a "ptr + offset" memory location*



## An Example X86 Target emit method

```
void emitAssign(ILAssignableExpr lhs,
                ILExpr rhs) {
  Location rhs_location =
      rhs.codegen(this);
  lhs.codegenAssign(rhs_location, this);
  deallocateReg(rhs_location);
}
```

Each **ILAssignableExpr** implements **codegenAssign**

- invokes appropriate **emitAssign** operation, e.g. **emitVarAssign**

## Generation for Comparisons

- What code to generate for **arg1 <.int arg2**
- MIPS: use an **slt** instruction to compute boolean-valued int result into a register
- x86 (and most other machines): no direct instruction
- Have comparison instructions, which set condition codes
  - e.g. **cmpl %arg2, %arg1**
- Later conditional branch instructions can test condition codes
  - e.g. **jl, jle, jge, jg, je, jne label**
- What instructions to generate?

## Generation for Compares

```
Location emitIntLessThanValue(ILExpr arg1,ILExpr arg2) {
   Location arg1_location=arg1.codegen(this);
   Location arg2_location=arg2.codegen(this);
   emitOp("cmpl",regOperand(arg2_location),…);
   deallocateReg(arg1_location);…
   Location result_location =
     allocateReg(ILType.intILType());
  String true_label = getNewLabel();
  emitOp("jl", true_label);
  emitOp("movl", intOperand(0),regOperand(result_location));
  String done_label = getNewLabel();
  emitOp("jmp", done_label);
  emitLabel(true_label);
  emitOp("movl", intOperand(1),regOperand(result_location));
  emitLabel(done_label);
  return result_location;
}
```

## Generation for Branch

- What code to generate for
  **iftrue test goto label**

```
void emitConditionalBranchTrue(ILExpr
  test,ILLabeltarget){
  Location test_location=test.codegen(this);
      emitOp("cmpl", intOperand(0),
               regOperand(test_location));
      emitOp("jne", target.getName());
   }
```

## Generation for Branch

- What is generated for
  **iftrue arg1 <.int arg2 goto label**

```
    <emit arg1 into %arg1>
    <emit arg2 into %arg2>
    cmpl %arg2, %arg1
    jl true_label
    movl $0, %res
    jmp done_label
  true_label:
    movl $1, %res
  done_label:
    cmpl $0, %res
    jne label
```

- Can we do better?

## Optimized Branches

- Idea: boolean-valued IL expressions can be generated two ways, depending on their consuming context
  - for their value or for their condition code
- Existing code gen operation on IL expression produces its value
- New **`codegenTest`** operation on IL expression produces its condition code
  - **`X86ComparisonResultLocation`** represents this result
- Now conditional branches can evaluate their test expression in the "for condition code" style

## Optimized Branches

```
void emitConditionalBranchTrue(ILExpr test,
                               ILLabeltarget){
   Location test_location=test.codegen(this);
   X86ComparisonResultLoc cc =
       (X86ComparisonResultLoc) test_location;
   emitOp("j" + cc.branchTrueOp(),
   target.getName());
}
```

## IL `codegenTest` Default Behavior

```
class ILExpr extends ILExpr {
   ...
   Location codegenTest(Target target) {
      return target.emitTest(this);
   }
}
```

In `X86Target` class:

```
  Location emitTest(ILExpr arg) {
    Location arg_location = arg.codegen(this);
    emitOp("cmpl", intOperand(0),
           regOperand(arg_location));
    deallocateReg(arg_location);
    return new X86ComparisonResultLoc("ne");
  }
```

## IL `codegenTest` Specialized Behavior

```
class ILIntLessThanExpr extends ILExpr {
  …
  Location codegenTest(Target target) {
     return target.emitIntLessThanTest(arg1, arg2);
  }
}
```

*In `X86Target` class:*

```
Location emitIntLessThanTest(ILExpr arg1,ILExpr arg2) {
    Location arg1_location=arg1.codegen(this);
    Location arg2_location=arg2.codegen(this);
    emitOp("cmpl",regOperand(arg2_location), …);
    deallocateReg(arg1_location);
    …
    return new X86ComparisonResultLoc("l");
}
```

## Register Allocation: Cool Algorithm

- How to convert the infinite sequence of temporary data references, t1, t2, … into finite assignment register numbers $8, $9, …, $25
- Goal: Use available registers with minimum spilling
- Problem: Minimizing the number of registers is NP-complete … it is equivalent to chromatic number--minimum colors to color nodes of graph so no edge connects same color

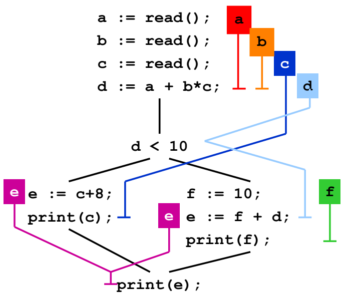
## Begin With Data Flow Graph

- procedure-wide register allocation
- only live variables require register storage

**dataflow analysis**: a variable is live at node N if *the value* it holds is used on some path further down the control-flow graph; otherwise it is dead

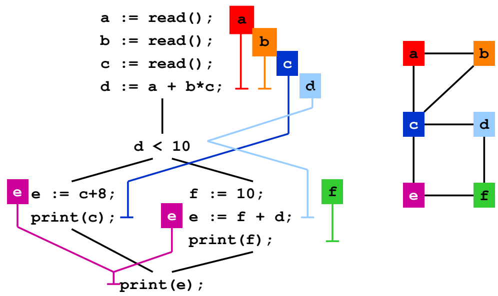
- two variables(values) interfere when their live ranges overlap

## Live Variable Analysis



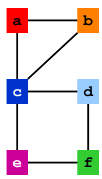

```
a := read();
b := read();
c := read();
d := a + b*c;
if (d < 10 ) then
    e := c+8;
    print(c);
else
    f := 10;
    e := f + d;
    print(f);
fi
print(e);
```

## Register Interference Graph

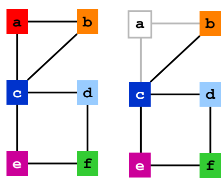



## Graph Coloring

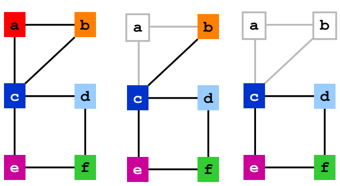
- NP complete problem
- Heuristic: color easy nodes last
  - find node *N* with lowest degree
  - remove *N* from the graph
  - color the simplified graph
  - set color of *N* to the first color that is not used by any of *N*'s neighbors
- Basics due to Chaitin (1982)



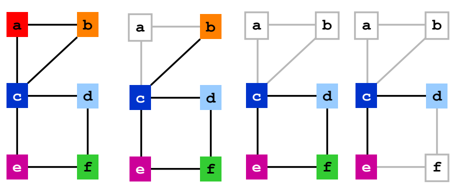

## Apply Heuristic



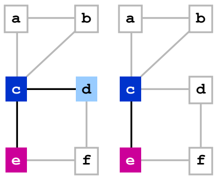
## Apply Heuristic



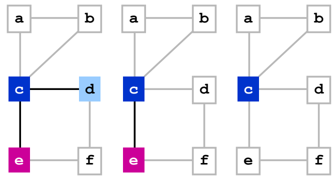
## Apply Heuristic



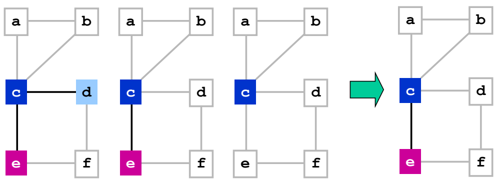
## Continued



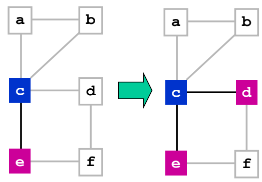

## Continued



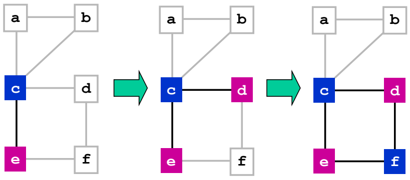

## Continued



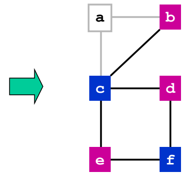

## Continued



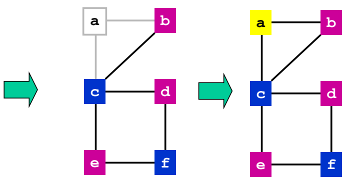

## Continued



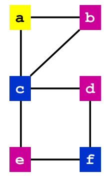

## Continued



## Continued




## Final Assignment




```
a := read();
b := read();
c := read();
d := a + b*c;
if (d < 10 ) then
   e := c+8;
   print(c);
else
   f := 10;
   e := f + d;
   print(f);
fi
print(e);
```

## What is the O(running time)?

- Acceptable?



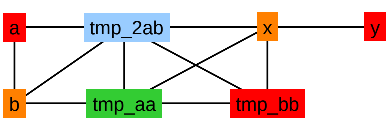
## Example: for small groups

```
{  int tmp_2ab = 2*a*b;
   int tmp_aa = a*a;
   int tmp_bb = b*b;

   x := tmp_aa + tmp_2ab + tmp_bb;
   y := tmp_aa - tmp_2ab + tmp_bb;
}
```

given that a and b are live on entry and dead on exit, and that x and y are live on exit:
(a) construct the register interference graph
(b) color the graph; how many registers are needed?

## 4 Registers Needed




## Code Generation Summary

- Code generation is
  - Machine specific
  - Error prone
  - Least “elegant” of the compilation process
- Code generation is
  - Place where key transformation takes place in the compiler
  - Most visible impact on performance

## Generation to Optimization: data-flow

- The data-flow analysis sketched for register allocation via coloring gives a feel for many of the techniques at the basis of optimization
- Data-flow analysis gathers information about the possible set of values calculated at various points in program, using a control-flow graph (CFG) representation
- Data-flow analysis usually works by setting up dataflow equations for the CFG node, solving these equations by reaching a fixpoint
  - Due to Kildall (1973) – UW CSE PhD #7 (1972)



## Sensitivity

- Data-flow analysis is flow-sensitive – the order of statement in the CFG matters
- But almost always path-insensitive – doesn't consider the values of predicates at conditionals
- Can be context-sensitive – that is, some analyses care about which calling context occurs



## Forward data-flow

- The classic example of data-flow analysis is *reaching definitions* – which definitions may reach a given point in the code
- Dataflow equations for each block in CFG
  - $Reach_{in}[S] = \cup_{p \in pred(S)} Reach_{out}[p]$
  - $Reach_{out}[S] = Gen[S] \cup (Reach_{in}[S] - Kill\ [S])$
- Need
  - **Gen[d: y is assigned] = {d}**
  - **Kill[d: y is assigned] = Defs[y] – {d}**
    - **Defs[y]** is the set of definitions that assign to **y**

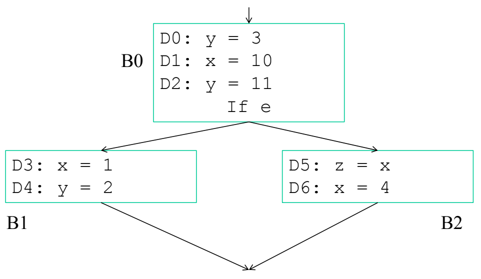


## Boring Example (wikipedia)

```
1: if b==4 then
2:    a = 5;
3: else
4:    a = 3;
5: endif
6:
7: if  a < 4 then
8:    ...
```



## Another example: from Stanford web






$[\![st]\!]\langle C, I, E\rangle = \langle C', I', E'\rangle$, *where:*

$$C' = \begin{cases} (C - kill) \cup gen \cup I & \text{if } strong \\ C \cup gen \cup I & \text{if not } strong \end{cases}$$

$$I' = I$$

$$E' = E \cup gen$$

For all *s ∈ Sasgn, sa ∈ Salloc, se ∈ Sentry, i ∈ I :*
[JOIN] *Res(•s) i = Fs∈pred(s) Res(s•) i*
[TRANSF] *Res(s•) i = Fi∈I ([[s]](ρ, (i, Res(•s) i))) i*
[ALLOC] *Res(sa•) ia ha, where [[sa]]gen(ρ) = (ia, ha)*
[ENTRY] *Res(•se) i ao i*