

CSE 401/M501 – Compilers

x86-64, Running MiniJava,
Basic Code Generation and Bootstrapping
CSE 401/M501 staff – Section Slides
Spring 2025

Generating .asm Code

- Suggestion: isolate the actual compiler output operations in a handful of routines

- Usual modularity reasons & saves some typing
- Some possibilities

```
// write code string s to .asm output
void gen(String s) { ... }
// write "op src,dst" to .asm output
void genbin(String op, String src, String dst) {...}
// write label lbl to .asm output as "lbl:"
void genLabel(String lbl) { ... }
```

- A handful of these methods should do it

A Simple Code Generation Strategy

- Goal: quick 'n dirty correct code; “optimize” later if time
- Traverse AST primarily in execution order and emit code during the traversal
 - Codegen visitor might want to traverse the tree in ad-hoc ways depending on sequence that parts need to appear in the asm code
- Treat the x86-64 as a 1-register machine with a stack to hold additional intermediate values(!)
 - Ugly code, but will work – better later if there's time

(The?) Simplifying Assumption

- Store all values (reference, int, boolean) in 64-bit quadwords
 - Natural size for 64-bit pointers, i.e., object references (variables of class types)
 - C's "long" size for integers
 - Use `int64_t` or `uint64_t` in any C code that interacts with MiniJava generated code to guarantee size (declared in `<stdint.h>`)
 - This can produce different results from Java `ints` in edge cases, mostly involving overflow. We'll ignore.

Before Codegen Visitor Pass...

- Need an initial pass through class and method symbol tables to assign locations to variables
 - Method local variables: successive offsets in the stack frame relative to `%rbp` (-8, -16, ...)
 - Also for parameters – allocate space to store copies in stack frame when needed (or always, to keep things simple)
 - Object instance variables: successive offsets from the start of the object (+0 is vtable pointer, instance variables at +8, +16, ...)
- This will also compute the size of each stack frame and object which is needed later
- Also assign vtable offsets for method pointers in this initial pass

x86 as a Stack Machine

- Idea: Use x86-64 stack for expression evaluation with `%rax` as the logical “top” of the stack (initially empty)
- Invariant: Whenever an expression (or part of one) is evaluated at runtime, the generated code leaves the result in `%rax`
- If a value needs to be preserved while another expression is evaluated, push `%rax`, evaluate, then pop when first value is needed
 - Remember: ***always pop what you push***
 - Will produce lots of redundant, but correct, code
- Examples below follow code shape examples, but with more details about code generation

Example: Generate Code for Constants and Identifiers

Integer constants, say 17

```
gen(movq $17,%rax)
```

- leaves value in %rax

Local variables (any type – int, bool, reference)

```
gen(movq varoffset(%rbp),%rax)
```

Instance variables (“this.var”)

```
gen(movq varoffset(%rdi),%rax)
```

- (assumes %rdi still contains unaltered “this” ptr; use different register or saved copy if %rdi has changed)

Example: Generate Code for $\text{exp1} + \text{exp2}$

Visit exp1

- generates code to evaluate exp1 with result in $\%rax$

`gen(pushq $\%rax$)`

- push exp1 onto stack

Visit exp2

- generates code for exp2 ; result in $\%rax$

`gen(popq $\%rdx$)`

- pop left argument into $\%rdx$; cleans up stack

`gen(addq $\%rdx, \%rax$)`

- perform the addition; result in $\%rax$

- Note: Java requires operands must be evaluated left to right. Makes a difference if some operand has side effects (method call with read/write, change global variable, etc.). Be sure to do that.

Example: `var = exp;` (1)

Assuming that `var` is a local variable

Visit node for `exp`

- Generates code to eval `exp` and leave result in `%rax`

`gen(movq %rax,offset_of_variable(%rbp))`

Similar code if `var` is part of an object, but use pointer to the object instead of `%rbp`

Example: `var = exp;` (2)

If `var` is a more complex expression (object instance variable or array element, for example)

visit `var`

`gen(pushq %rax)`

- push lvalue (address) of variable or object containing variable onto stack

visit `exp`

- leaves rhs value in `%rax`

`gen(popq %rdx)`

`gen(movq %rax, appropriate_offset(%rdx))`

Example: call obj.f(e1,e2,...en)

In principal the code should work like this:

Visit obj

- leaves reference to object in %rax

gen(movq %rax,%rdi)

- “this” pointer is first argument

Visit e1, e2, ..., en. For each argument,

- gen(movq %rax,%correct_argument_register)

generate code to load method table pointer located at 0(%rdi) into some register, probably %rax

generate call instruction with indirect jump

Method Call Complications

- Big one: code to evaluate any argument might clobber argument registers (i.e., computing an argument value might require a method call)
 - Possible strategy to cope on next slides, but feel free to do something better
- And more: clobbers *current* method's %rdi (this ptr)
 - Save it on method entry; reload after call (or on every use)
- Other one: what if a method has too many parameters?
 - OK for CSE 401/M501 to assume all methods have ≤ 5 parameters plus “this” – do better if you want

Method Calls in Parameters

- Suggestion to avoid trouble:
 - Evaluate parameters in order left to right, and push them on the stack as each one evaluated
 - Left to right ordering required by Java language specification
 - Right before the call instruction, pop the parameters into the correct registers
- But....

Stack Alignment (1)

- Above ~~idea~~ hack works provided we don't call a method while an odd number of parameter values are pushed on the stack!
 - (violates 16-byte %rsp alignment on method call...)
- We have a similar problem if an odd number of intermediate values are pushed on the stack when we call a method while evaluating an expression
 - (We might get away with it if it only involves calls to our own generated, not library, code, but it would be wrong* to do that)
 - *i.e., might “work”, but it's not the right way to solve the problem

Stack Alignment (2)

- Workable solution: keep a counter in the code generator of how much has been pushed on the stack. If needed, emit extra `gen(pushq %rax)` (or some other register) to push a useless value and align the stack before generating a call instruction
 - Be sure to pop it after!!
- Another (cleaner, but more work) solution: make stack frame big enough and use `movq` instead of `pushq` to store arguments and temporaries
 - Needs extra bookkeeping to keep track of how much to allocate for stack frame and how temps are used and where they are in the frame

Sigh...

- Multiple registers for method arguments is a big win compared to pushing on the stack, but complicates our life since we do not have a ~~fancy~~ decent register allocator
- Feel free to do better than this simple push/pop scheme – but remember, simple and works wins over fancy and not finished or broken

Code Gen for Method Definitions

- Generate label for method
 Classname\$methodname:
- Generate method prologue
 push %rbp, copy %rsp to %rbp, subtract frame size
 (multiple of 16) from %rsp
- Visit statements in order
 - Method epilogue is normally generated as part of a return statement (details shortly)
 - In MiniJava the return is generated after visiting the rest of the method body to generate its code

Registers again...

- Method parameters are in registers
- But code generated for methods also will be using registers, even if there are no calls to other methods
- So how do we avoid clobbering parameters?
- Suggestion: Allocate space in the stack frame and save copies of all parameter registers on method entry. Use those copies as local variables when you need to reference a parameter.

Example: return exp;

- Visit exp; this leaves result in %rax where it should be
- Generate method epilogue (copy %rbp to %rsp, pop %rbp) to unwind the stack frame; follow with ret instruction
 - Can use leave instead of movq/popq to unwind the stack, but the separate instructions might be a little easier to trace/debug if something isn't right

Control Flow: Unique Labels

- Needed in code generator: a String-valued method that returns a different label each time it is called (e.g., L1, L2, L3, ...)
 - Improvement: a set of methods that generate different kinds of labels for different constructs (can really help readability of the generated code)
 - (while1, while2, while3, ...; if1, if2, ...; else1, else2, ...; endif1, endif2,)

Control Flow: Tests

- Recall that the context for compiling a boolean expression is:
 - Label or address of jump target
 - Whether to jump if true or false
- So the visitor for a boolean expression should receive this information from the parent node visitor

Example: while(exp) body

- Assuming we want the test at the bottom of the generated loop...

gen(jmp testLabel)

gen(bodyLabel:)

visit body

gen(testLabel:)

visit exp (condition) with target=bodyLabel and sense="jump if true"

Example: $\text{exp1} < \text{exp2}$

- Similar to other binary operators
- Difference: surrounding (parent) context is a target label and whether to jump if true or false
- Code
 - visit exp1
 - gen(pushq %rax)
 - visit exp2
 - gen(popq %rdx)
 - gen(cmpq %rdx,%rax)
 - gen(condjump targetLabel)
 - appropriate conditional jump depending on sense of test

Boolean Operators

&& (and | | if you add it)

- Create label(s) needed to skip around the parts of the expression
- Generate subexpressions with appropriate target labels and conditions

!exp

- Generate exp with same target label, but reverse the sense of the condition

Reality check

- Lots of projects in the past have evaluated all booleans to get 1 or 0, then tested that value for control flow
- Would be nice to do better (as above), but “simple and works...”
- (And we need to be able to generate the 0/1 anyway for storable boolean expressions)

Join Points

- Loops and conditional statements have join points where execution paths merge
- Generated code must ensure that machine state will be consistent regardless of which path is taken to get there
 - i.e., the paths through an if-else statement must not leave a different number of values pushed onto the stack
 - If we want a particular value in a particular register at a join point, both paths must put it there, or we need to generate additional code to move the value to the correct register
- With our simple 1-accumulator model of code generation, this should usually be true without needing extra work; with better use of registers it becomes a bigger issue
 - With more registers, would need to be sure they are used consistently at join point regardless of how we get there