CSE 401 – Compilers

Lecture 20: x86-64, GNU Assembler, and Project Code Generation, Part II
Michael Ringenburg
Winter 2013

Reminders/Announcements

• Midterms are graded
  – If you haven’t picked yours up yet, you can stop by during my office hours today (2:30-3:30)
• Project part 3 due this Friday
• Part 4 will be due on Friday, March 15 (last day of class). I will put the assignment out this afternoon.
• Laure out of town next week – no office hours.
Review: boot.c

- We will provide a small bootstrap named boot.c with the part 4 assignment.
  - A tiny C program that calls your compiled code as if it were an ordinary C function (assumes your main label is asm_main).
- It also contains some functions that compiled code can call as needed
  - This is a mini “runtime library”
    - Leverages gcc’s C runtime for program startup/initialization, I/O, memory management, etc.
    - A tiny MiniJava interface layer on top for access to input, output, memory allocation
  - Add to this if you like
    - Sometimes simpler to generate a call to a newly written library routine instead of generating in-line code

Bootstrap Program Sketch

```c
#include <stdio.h>
extern void asm_main(); /* label for your compiled code */
/* execute compiled program */
void main() { asm_main(); }
/* return next integer from standard input */
long get() { ... }
/* write x to standard output */
void put(long x) { ... }
/* return a pointer to a block of memory at least nBytes large (or null if insufficient memory available) */
char* mjmalloc(long nBytes) { return malloc(nBytes); }
```
Review: Library Calls

- To call these library functions (get, put, mjmalloc, and anything you might add), just follow x86-64 calling conventions. On Linux, call target label is just the function name (Windows and OS X add a preceding _).
- E.g., a code template for `System.out.println(exp)` (MiniJava’s “print” statement) might be:

  ```
  <compile exp; result in %rax>
  movq %rax,%rdi ; load argument register
  call put ; call external put routine
  ```

- If the stack is not kept 16-byte aligned, calls to external C or library code are the most likely place for a runtime error

Assembler File Format

- GNU .s file syntax is roughly this (sample code will be provided with part 4 of the project)

  ```
  .text       # code segment
  .globl asm_main       # label for main program
  asm_main:
  ...
  class1$method1:
  ...
  .data  # generated method tables
  ...
  # generated method tables
  # generated method tables
  # repeat .text/.data as needed
  ```
Let’s Take A Look at the Bootstrap, and a sample .s file

Generating .asm Code

• Suggestion: isolate the actual assembly output operations in a handful of routines
  – Modularity & saves some typing
  – 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 L to .asm output as “L:”
    void genLabel(String L) { ... }
  – 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
  – Visitor may traverse the tree in ad-hoc ways depending on sequence that parts need to appear in the code (based on code recipes/templates we studied for particular syntax constructs/AST nodes).
• Treat the x86 as a 1-register machine with a stack for additional intermediate values
  – Except for function calls (due to register-based calling convention on x86-64)
  – Don’t have to worry about register allocation

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
  – Means you won’t necessarily get the right overflow behavior for ints (supposed to be 32-bit in Java), but that is okay (you’ll still get full credit).
    • MiniJava was originally designed for 32-bit machines.
x86 as a Stack Machine

- Idea: Use x86-64 stack for expression evaluation with %rax as the “top” of the stack
- 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 some details about where code generation fits

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 offset(%rbp),%rax")
  ```
  
  – Recall simplifying assumption that everything is 64-bit in MiniJava
Example: Generate Code for \(\text{exp1} + \text{exp2}\)

- Visit \(\text{exp1}\)
  - generate code to evaluate \(\text{exp1}\) with result in %rax
  - `gen("pushq %rax")`
  - push \(\text{exp1}\) result onto stack
- Visit \(\text{exp2}\)
  - generate code for \(\text{exp2}\); result in %rax
  - `gen("popq %rdx")`
  - pops \(\text{exp1}\) result into %rdx (also cleans up stack)
  - `gen("addq %rdx,%rax")`
  - perform the addition; result in %rax

Example: \(\text{var} = \text{exp}; (1)\)

- Assuming that \text{var} is a local variable
  - Visit node for \text{exp}
    - Generates code that leaves the result of evaluating \text{exp} in %rax
    - `gen("movq %rax,offset_of_variable(%rbp)")"`)
Example: Simple main()

• With this, we can now generate code for a simple main method:

```java
public static void main() {
    int x;
    {
        x = 5;
        System.out.println(x + 1);
    }
}
```

Possible AST
Possible AST

MainMethodDecl
  DeclList
  Statement
  ...
  Block
  Assign
  Print
  Plus
  x
  5
  x
  1

#prologue
push %rbp
movq %rsp,%rbp
subq $16,%rsp

You’ll likely have a method to generate prologues, e.g., genPrologue(int numLocals). Call it before generating the method’s statement list. Also, recall suggestion to round up frame size to multiples of 16.

Possible AST

MainMethodDecl
  DeclList
  Statement
  ...
  Block
  Assign
  Print
  Plus
  x
  5
  x
  1

#prologue
push %rbp
movq %rsp,%rbp
subq $16,%rsp

#Assign right: 5
movq $5,%rax
Possible AST

MainMethodDecl
  \[ ... \]
  \[ \text{DeclList} \]
  \[ \text{Statement} \]
    \[ \text{Block} \]
      \[ \text{Assign} \]
        \[ x \]
        \[ 5 \]
      \[ Print \]
        \[ Plus \]
          \[ x \]
          \[ 1 \]

#prologue
push %rbp
movq %rsp,%rbp
subq $16,%rsp

#Assign right: 5
movq $5,%rax

#Assign
movq %rax,-8(%rbp)
Possible AST

MainMethodDecl
  \text{DeclList} \quad \text{Statement}

... Block

Assign \quad \text{Print}
  x \quad 5 \quad \text{Plus}
  x \quad 1

Possible AST

MainMethodDecl
  \text{DeclList} \quad \text{Statement}

... Block

Assign \quad \text{Print}
  x \quad 5 \quad \text{Plus}
  x \quad 1

#Plus exp1
movq -8(%rbp),%rax
pushq %rax

#Plus exp2
movq $1,%rax

#Plus
popq %rdx
addq %rdx,%rax
Possible AST

MainMethodDecl

DeclList

Statement

Block

Assign

x 5

Print

Plus

x 1

Possible AST

MainMethodDecl

DeclList

Statement

Block

Assign

x 5

Print

Plus

x 1

... #Plus exp1
movq -8(%rbp),%rax
pushq %rax

#Plus exp2
movq $1,%rax

#Plus
popq %rdx
addq %rdx,%rax

#Print
movq %rax,%rdi
call put

... #Print
movq %rax,%rdi
call put

#Epilogue
leave
ret

Generate epilogue after statements.
Suggestion

• Build your code generator incrementally.
  – Start with enough functionality to compile very simple programs.
  – Then, add functionality to compile slightly more complex programs.
  – Rinse Test (thoroughly) and repeat.

• The last step is key.
  – Debugging code generators is hard (basically, it comes down to debugging assembly code).
  – By doing small pieces, and testing thoroughly after each one, you make your life much easier.

• The assignment will have a (time-tested) approach to incrementally building your code generator.

Example: var = exp; (2)

• If var is a more complex expression (object or array reference, for example)
  – visit var
    • Since it’s going to be used as a store target, you want to evaluate the address, not the value. For objects this may be default, but probably not fields/array elements.
    • MiniJava has a limited set of “var” possibilities, so you could possibly special case them if you wanted.
  – gen(pushq %rax)
    • push address of object/field/array element/etc
  – visit exp – leaves rhs value in %rax
  – gen(popq %rdx)
  – gen(movq %rax,appropriate_offset(%rdx))
Example: Generate Code for 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 register like %rax
    • gen(“movq (%rdi),%rax”)
  – generate call instruction with indirect jump
    • gen(“call *M(%rax)”), where M is offset of f in method table

Method Call Complications

• Big one: code to evaluate any argument might clobber argument registers (i.e., method call in some parameter value)
  – Possible strategy to cope on next slides, other solutions may be possible
• Not quite so bad: what if a method has more than 6 parameters?
  – Traditionally, supporting extra parameters hasn’t been required in this course, so I won’t either.
  – Not hard, and a reasonable extension to attempt for some extra credit.
  – Requires extra bookkeeping in caller and callee (especially when combined with our strategy for dealing with the above issue, due to evaluation order rules).
Method Calls in Parameters

- Suggestion to avoid trouble:
  - Evaluate parameters and push them on the stack
  - Right before the call instruction, pop the parameters into the correct registers
    - Works if we are dealing with at most 6 parameters.
    - If attempting extension: later parameters should be evaluated after earlier parameters, so parameters 7+ will normally be in the way of popping first 6.
    - Could use free registers to hold them temporarily, and repush (but requires a register allocator to track free regs).
    - Or could leave all the parameters in storage and copy the first 6 into registers, then deallocate everything after return
    - But....

Stack Alignment (1)

- Above strategy works provided we don’t call a method while an odd number of parameter values are pushed on the stack!
  - (violates 16-byte alignment on method call...)
- We have a similar problem if an odd number of intermediate/temporary values are pushed on the stack when we call a function in the middle of evaluating an expression
Stack Alignment (2)

• Workable solution: keep a counter in the code generator of how much has been pushed on the stack. If needed, gen(pushq %eax) to align the stack before generating a call instruction
  – Be sure to generate a popq afterwards, iff you pushed
• Another solution: make stack frame big enough and use movq instead of pushq to store arguments and temporaries
  – What most real compilers do – also frees up %rbp
  – Will need some extra bookkeeping to allocate space for arguments and temporaries

In Summary ...

• 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 register allocator
• For project, you are only required to handle up to 6 parameters.
  – But you may try to do more as an extension, if you wish.
Code Gen for Method Definitions

- Generate label for method
  - Classname$methodname:
- Walk list of declarations
  - Assign offsets from %rbp for each local: -8, -16, -24, etc. Store in variable’s symbol table entry.
- Generate method prologue
  - Push rbp, copy rsp to rbp, subtract frame size from rsp
- Visit statements in order
  - Method epilogue is normally generated as part of each return statement (or return statements branch to epilogue)
  - In MiniJava the return is generated after visiting the method body to generate its code
    - MiniJava only allows a single return at the end of the method.
    - Main special case: Generate epilogue without return

Example: return exp;

- Visit exp; leaves result in %rax where it should be
- Generate method epilogue to unwind the stack frame; end with ret instruction
Control Flow: Unique Labels

- Needed: a String-valued method that returns a different label each time it is called (e.g., L1, L2, L3, ...)
  - Allows us to create unique labels for control flow.
  - Variation: 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, ...; fi1, fi2, ... .)

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
  - There’s a few ways you can do this
    - Visitor object can store state: parent can store for child. Make sure visit method remembers the state when it was called (may make nested calls with different state, e.g. x < y && x < z, or !exp).
    - Or, can augment the accept/visit methods for expressions to pass additional parameters – and pass null state (or whatever) for non-boolean expressions, since it shouldn’t be used.
    - Or, have parent visitor store context in child’s AST node.
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: exp1 < exp2

- Similar to other binary operators
- Difference: 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 (jl, jnl) depending on sense of test
Boolean Operators

- `&&` (and *if you include it*)
  - Follow the same recipes as the IA-32 examples from last week, except in Gnu x86-64
  - Create label needed to skip around the two 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

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 reach a join point
  - i.e., the paths through an if-else statement must not leave a different number of words 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 a simple 1-accumulator model of code generation, this should generally be true without needing extra work; with better use of registers this becomes an issue
  - Stack of temporary values should be empty after each statement
And That’s It…

- We’ve now got enough on the table to complete the compiler project
- Coming Attractions
  - Survey of optimization: analysis and transformations
  - More sophisticated code generation
  - Two guest lectures:
    - Real-world parsing (it’s not just for compilers!)
    - Real register allocators
    - Yes, they will be on the final 😊