



CSE P 501 – Compilers

x86-64, Running MiniJava,
Basic Code Generation and Bootstrapping
Hal Perkins
Autumn 2011



Some x86-64 References

(Links on course web)

- **x86-64 Machine-Level Programming**
 - Earlier version of sec. 3.13 of *Computer Systems: A Programmer's Perspective* 2nd ed. by Bryant & O'Hallaron (CSE351 textbook)
- **From www.x86-64.org:**
 - System V Application Binary Interface AMD64 Architecture Processor Supplement
 - Gentle Introduction to x86-64 Assembly
- **x86-64 Instructions and ABI**
 - Handout for University of Chicago CMSC 22620, Spring 2009, by John Reppy



Compiler Target

- Compiler output is an assembly-language file that is linked to the “real” main program written in C
 - Lets the C library set up the stack, heap; handle I/O, etc.
- Default target is Linux x86-64
 - Should not need big changes for Microsoft MASM, but not tried yet
 - Help each other out – discussion board, etc.
 - Examples below use gnu/linux asm notation



Intel vs. GNU Assembler

- The GNU assembler uses AT&T syntax. Main differences:

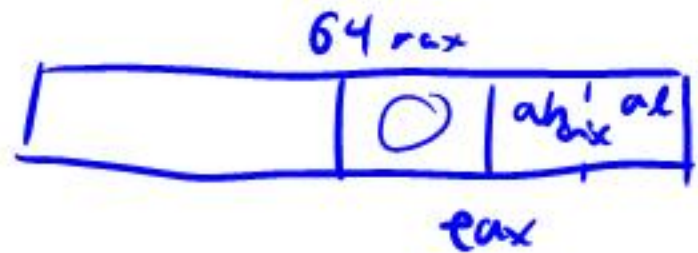
	Intel/Microsoft	AT&T/GNU as
Operand order: <u>op</u> <u>a</u> , <u>b</u>	<u>a</u> = a op b (dst first)	<u>b</u> = a op b (dst last)
Memory address	[baseregister+offset]	offset(baseregister)
Instruction mnemonics	mov, add, push, ...	movl, addl, pushl [operand size is added to end]
Register names	eax, ebx, ebp, esp, ...	<u>%eax</u> , %ebx, %ebp, %esp, ...
Constants	17, 42	<u>\$17</u> , \$42
Comments	; to end of line	# to end of line or /* ... */



x86-64 Main features

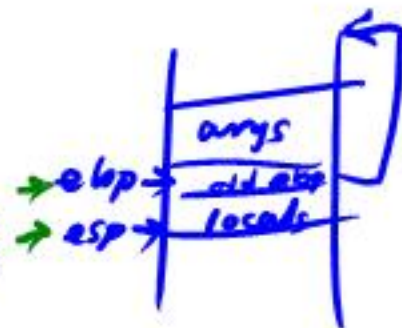
- 16 64-bit general registers; 64-bit integers (but `int` typically defaults to 32 bits; `long` is 64 bits)
- 64-bit address space; pointers are 8 bytes
- 8 additional SSE registers (total 16); used instead of x87 floating point by default
- Register-based function call conventions
- Additional addressing modes (pc relative)
- 32-bit legacy mode
- Some pruning of old features

x86-64 registers



- 16 64-bit general registers
 - `%rax`, `%rbx`, `%rcx`, `%rdx`, `%rsi`, `%rdi`, `%rbp`, `%rsp`, `%r8`-`%r15`
- Registers can be used as 64-bit ints or pointers, or 32-bit ints (upper half set to 0 automatically)
 - Also possible to reference low-order 16- and 8-bit chunks

x86-64 Function Calls



- First 6 arguments in registers, rest on the stack
- ✓ ■ int/pointer result returned in %rax
- Stack frame should be 16-byte aligned when call instruction is executed (i.e., %rsp value is 0xddddddddddddd0; pushed return address has that address minus 8)
- We'll use %rbp as frame pointer, but compilers often adjust %rsp once on function entry and reference locals relative to %rsp using a fixed-size stack frame

x86-Register Usage



■ `%rax` – function result

■ Arguments 1-6 passed in these registers

→ ■ `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`

■ "this" pointer is first argument, in `%rdi`

■ `%rsp` – stack pointer; value must be 8-byte aligned always and 16-byte aligned when calling a function

■ `%rfp` – frame pointer (optional use)



x86-64 Register Save Conventions

- A called function must preserve these registers (or save/restore them if it wants to use them)

→ ■ `%rbx, %rbp, %r12-%r15`

- `%rsp` isn't on the "callee save list", but needs to be properly restored for return
- All other registers can change across a function call

`gcc -S foo.c ⇒ foo.S`

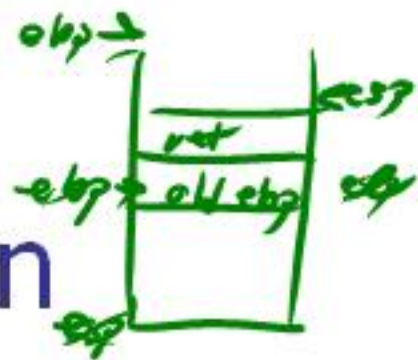
x86-64 Function Call



- Caller places up to 6 arguments in registers, rest on stack, then executes call instruction (which pushes 8-byte return address)
- On entry, called function prologue is like the 32-bit version:

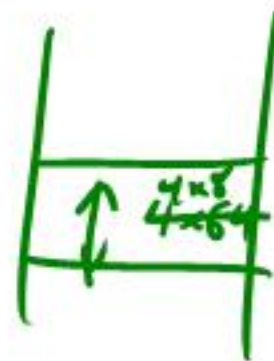
```
pushq %rbp  
movq %rbp, %rsp movq %esp, %ebp  
subq $framesize, %rsp
```

x86-64 Function Return



- Called function puts result in `%rax` (if any) and restores any callee-save registers if needed
- Called function returns with:
 - `movq %rbp,%rsp # or leave instead of`
 - `popq %rbp # movq/popq`
 - `ret`
 - Same logic as 32-bit
- If caller allocated space for arguments it deallocates as needed

The Nice Thing About Standards...




-
- A hand-drawn diagram of a stack frame. It consists of a vertical rectangle with a horizontal line near the top. An upward-pointing arrow is drawn to the left of the horizontal line, and the text 'return address' is written to the right of the arrow. Below the horizontal line, there is another horizontal line, and the text 'return address' is written between these two lines.
- The above is the System V/AMD64 ABI convention (used by Linux, OS X)
 - Microsoft's x64 calling conventions are slightly different (sigh...)
 - First four parameters in registers `%rcx`, `%rdx`, `%r8`, `%r9`; rest on the stack
 - Stack frame needs to include empty space for called function to save values passed in parameter registers if desired



Running MiniJava Programs

- To run a MiniJava program
 - Space needs to be allocated for a stack and a heap
 - `%rsp` and other registers need to have sensible initial values
 - We need some way to allocate storage (new) and communicate with the outside world



```
main() {  
  run-main(); ← minijava  
}
```

Bootstrapping from C

- Idea: take advantage of the existing C runtime library
- Use a small C main program to call the MiniJava main method as if it were a C function
- C's standard library provides the execution environment and we can call C functions from compiled code for I/O, malloc, etc.

C `asm_main()`

call `asm-main`

Assembler File Format

- GNU syntax is roughly this (sample code will be provided with codegen phase of the project)

```
→ .text                                # code segment
   .globl asm_main                     # start of compiled static main
   ;; generated code                    # repeat .code/.data as needed
asm_main:                              # start of compiled "main"
   ...
→ .data
   ;; generated method tables # repeat .text/.data as
   needed
   ...
end
```

asm-main()

call _asm-main



External Names

- In a Linux environment, an external symbol is used as-is (*xyzy*)
- In Windows and OS X, an external symbol *xyzy* is written in asm code as *_xyzy* (leading underscore)
- Adapt to whatever environment you're using



Generating .asm Code

- Suggestion: isolate the actual compiler 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
- Treat the x86 as a 1-register machine with a stack for additional intermediate values



(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

$a + b * c$



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)
```


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



- Visit exp1
 - generate code to evaluate exp1 with result in $\%rax$
- `gen(pushq $\%rax$)`
 - push exp1 onto stack
- Visit exp2
 - generate code for exp2 ; result in $\%rax$
- `gen(popq $\%rdx$)`
 - pop left argument into $\%rdx$; clean up stack
- `gen(addq $\%rdx, \%rax$)`
 - perform the addition; result in $\%rax$



Example: `var = exp;` (1)

- 
- Assuming that `var` is a local variable
 - Visit node for `exp`
 - Generates code that leaves the result of evaluating `exp` in `%rax`
 - `gen(movq %rax,offset_of_variable(%rbp))`



Example: var = exp; (2)

- If var is a more complex expression (object or array reference, for example)
 - visit var
 - gen(pushq %rax)
 - push reference to variable or object containing variable onto stack
 - visit exp – leaves rhs value in %rax
 - gen(popq %rdx)
 - gen(movq %rax, appropriate_offset(%rdx))

Example: Generate Code for $\text{obj.f}(e_1, e_2, \dots, e_n)$



- 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 e_1, e_2, \dots, e_n . For each argument,
 - `gen(movq %rax, correct_argument_register)`
 - generate code to load method table pointer located at `0(%rdi)` into register like `%rax`
 - generate call instruction with indirect jump



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, but better solutions would be welcome
- Not quite so bad: what if a method has more than 6 (or 4 for Microsoft folks) parameters?
 - Let's punt that one and restrict the number of parameters to the number of parameter registers
 - Looks like the test programs are all ok here



Method Calls in Parameters

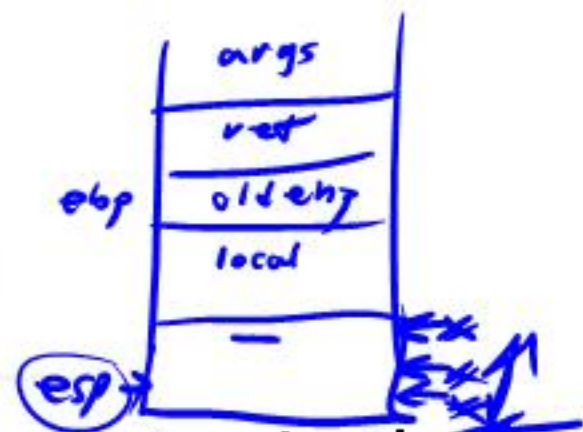
- Suggestion to avoid trouble:
 - Evaluate parameters and push them on the stack
 - Right before the call instruction, pop the parameters into the correct registers
 - Or leave the parameters in storage and copy them into registers, then deallocate after return
- But....



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 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 function in the middle of evaluating an expression
- (But we may get away with it if it only involves calls to our generated, not library, code)

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
- Another solution: make stack frame big enough and use `movq` instead of `pushq` to store arguments and temporaries
 - Will need some extra bookkeeping to allocate space for arguments and temporaries



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 register allocator
- better ideas for handling x86-64 function calls in MiniJava are most welcome



Code Gen for Method Definitions

int foo () {

- Generate label for method
- Generate method prologue
- Visit statements in order
 - Method epilogue is normally generated as part of each return statement (next)
 - In MiniJava the return is generated after visiting the method body to generate its code





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



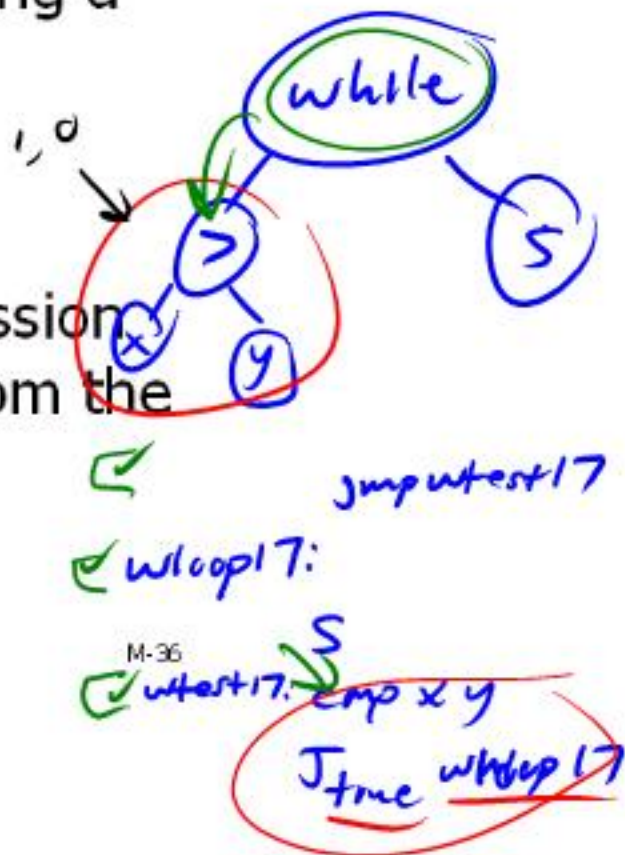
Control Flow: Unique Labels

- Needed: a String-valued method that returns a different label each time it is called (e.g., L1, L2, L3, ...)
 - 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,)

$$x > y$$

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





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: $exp1 < exp2$

1) true/false
Jump

2) target

- 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 depending on sense of test



Boolean Operators

- `&&` (and `||` if you include it)
 - 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



Bootstrap Program

- The bootstrap is a tiny C program that calls your compiled code as if it were an ordinary C function
- It also contains some functions that compiled code can call as needed
 - Mini “runtime library”
 - Add to this if you like
 - Sometimes simpler to generate a call to a newly written library routine instead of generating in-line code – implementer tradeoff



Bootstrap Program Sketch

```
#include <stdio.h>
extern void asm_main(); /* 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) { printf ... }
/* return a pointer to a block of memory at least nBytes
   large (or null if insufficient memory available) */
→ char* minijavaalloc(long nBytes) { return malloc(nBytes); }
```




Main Program Label

- Compiler needs special handling for the static main method label
 - Label must be the same as the one declared extern in the C bootstrap program and declared `.globl` in the `.s` asm file
 - `asm_main` used above
 - Could be changed, but probably no point
 - Why not "main"? (Hint: what is/where is the *real* main function?)



Interfacing to “Library” code


- Trivial to call “library” functions
- Evaluate parameters using the regular calling conventions
- Generate a call instruction using the function label
 - (External names need a leading _ in Windows, OS X)
 - Linker will hook everything up



System.out.println(exp)

- MiniJava's "print" statement

```
<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



And That's It...

- We've now got enough on the table to complete the compiler project
- Coming Attractions
 - Lower-level IR and control-flow graphs
 - Back end (instruction selection and scheduling, register allocation)
 - Middle (optimizations)