Lecture 15: x86-64 assembly language, code generation

David Hovemeyer

October 24, 2022

601.428/628 Compilers and Interpreters

Today

- x86-64 assembly language
- x86-64 tips
- Code generation

x64-64 assembly language

x86-64 assembly language

- Your compiler (in Assignments 3–6) will generate x86-64 assembly language
- x86-64 is the dominant instruction set architecture for general purpose computing (laptops, servers, etc.)
  - ARM is making inroads, though
- It's a 64-bit architecture
  - Registers are 64 bits wide
  - Memory addresses are 64 bits

x86-64 registers

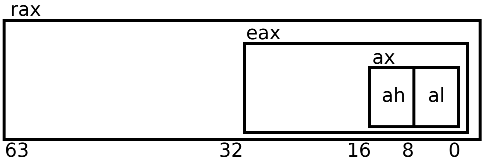
Register(s)	Note
`%rip`	Instruction pointer
`%rax`	Function return value
`%rdi`, `%rsi`	
`%rbx`, `%rcx`, `%rdx`	
`%rsp`, `%rbp`	Stack pointer, frame pointer
`%r8`, `%r9`, ..., `%r15`	

All of these registers are 64 bits (8 bytes)

Aside from `%rip` and `%rsp`, all of these are *general-purpose* registers

“Sub”-registers

- For historical reasons (evolution of x86 architecture from 16 to 64 bits), each data register is divided into
  - Low byte
  - Second lowest byte
  - Lowest 2 bytes (16 bits)
  - Lowest 4 bytes (32 bits)
- E.g., `%rax` register has `%al`, `%ah`, `%ax`, `%eax`:



Naming of sub-registers

Register	Sub-register 32 bit	16 bit	Lowest 8 bit
%rax	%eax	%ax	%al
%rbx	%ebx	%bx	%bl
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rdi	%edi	%di	%dil
%rsi	%esi	%si	%sil
%rsp	%esp	%sp	%spl
%rbp	%ebp	%bp	%bpl
%r8[1]	%r8d	%r8w	%r8b

[1] Same pattern for %r9–%r15

Stack

- The `%rsp` register is the *stack pointer*
  - Contains address of “top” of stack
  - Stack grows down (from high to low addresses), so `%rsp` decreases as stack grows

Assembly language syntax

- Each instruction has a mnemonic (`mov`, `push`, `add`, etc.)
- Most instructions will have one or two *operands* that specify data values (input and/or output)
  - At most **one** operand can be a memory reference
- On Linux, the standard tools use “AT&T” assembly syntax
  - Source is first operand, destination is second
- For instructions that do computations, destination operand is also a source value!
  - I.e., they are destructive
  - This makes code generation a bit interesting

Labels

- A *label* gives a name to the address of a location in memory (code or data)
  - Eventual runtime address generally not known ahead of time, linker and/or dynamic linker will resolve prior to execution
- Used to refer to procedures
- Used to refer to intermediate locations within procedure (local labels)
- Used to refer to global data and constants

Operand size suffixes

- You will notice that instruction mnemonics sometimes use suffixes to indicate the operand size:

Suffix	Bytes	Bits	Note
`b`	1	8	“Byte”
`w`	2	16	“Word”
`l`	4	32	“Long” word
`q`	8	64	“Quad” word

  (Use of `w` to mean 16 bits shows 16-bit origins of x86)
- E.g., `movq` means move a 64 bit value
- You can often omit the operand size suffix, but it's helpful for readability, and can even catch bugs

Assembly operands

Assume `count` and `arr` are global variables, R is a register, N is an immediate, S is 1, 2, 4, or 8

Type	Syntax	Example	Note
Memory ref	*Addr*	`count`	Absolute memory address
Immediate	\$$N$	`$8`, `$arr`	`$arr` is address of `arr`
Register	R	`%rax`	
Memory ref	(R)	`(%rax)`	Address = `%rax`
Memory ref	N(R)	`8(%rax)`	Address = `%rax`+8
Memory ref	(R,R)	`(%rax,%rsi)`	Address = `%rax`+`%rsi`
Memory ref	N(R,R)	`8(%rax,%rsi)`	Address = `%rax`+`%rsi`+8
Memory ref	(,R,S)	`(,%rsi,4)`	Address = `%rsi`×4
Memory ref	(R,R,S)	`(%rax,%rsi,4)`	Address = `%rax`+(`%rsi`×4)
Memory ref	N(,R,S)	`8(,%rsi,4)`	Address = (`%rsi`×4)+8
Memory ref	N(R,R,S)	`8(%rax,%rsi,4)`	Address = `%rax`+(`%rsi`×4)+8

Data movement

90% of assembly code is data movement (made-up statistic)

- `mov`: copy source operand to destination operand
  - Register
  - Memory location (only one operand can be memory location)
  - Immediate value (source operand only)
- Stack manipulation: `push` and `pop` instructions
  - Generally used for saving and restoring register values
  - `push`: decrement `%rsp` by operand size, copy operand to `(%rsp)`
  - `pop`: copy `(%rsp)` to operand, increment `%rsp` by operand size

Data movement examples

Instruction	Note
`movq $42, %rax`	Store the constant value 42 in `%rax`
`movq %rax, %rdi`	Copy 8 byte value from `%rax` to `%rdi`
`movl %eax, 4(%rdx)`	Move 4 byte value from `%eax` to memory at address `%rdx`+4
`pushq %rbp`	Decrement `%rsp` by 8,
	store contents of `%rbp` in memory location `%rsp`
`popq %rbp`	Load contents of memory location `%rsp` into `%rbp`,
	increment `%rsp` by 8

ALU operations

- ALU = "Arithmetic Logic Unit"
- An ALU is a hardware component within the CPU that does computations (of various kinds) on data values
  - Addition/subtraction
  - Logical operations (shifts, bitwise and/or/negation), etc.
- So, ALU instructions are the ones that do computations on values
  - Typically, ALU operates only on integer values
  - CPU will typically have floating-point unit(s) for operations on FP values

lea instruction

- `lea` stands for "Load Effective Address"
- Instructions that allow a memory reference as an operand generally do an *address computation*
  - E.g., `movl 12(%rdx,%rsi,4), %eax`
  - Computed address (for source memory location) is `%rdx`+(`%rsi`×4)+12
- The `lea` instruction computes a memory address, but does *not* access a memory location
  - E.g., `leaq 12(%rdx,%rsi,4), %rdi`
  - Quite similar to the address-of (&) operator in C and C++

Addition, subtraction

- add and sub instructions add and subtract integer values
- Two operands, second operand modified to store the result
  - Note that either operand (but not both) could be a memory reference
- E.g.,

```
movq $1, %r9
movq $2, %r10
addq %r9, %r10
/* %r10 now contains the value 3 */
```

- Overflow is possible!
  - Can detect using condition codes

Other ALU operations

There are lots of other ALU instructions!

- `inc`, `dec` (increment and decrement)
- Multiplication and division
- Logical/bitwise operations

Consult your favorite x86-64 reference for details

Control flow, condition codes

- Intra-procedural control flow: unconditional jump, conditional jump
- Target is the address of an instruction (in the same procedure)
  - Usually specified by a label
- Conditional jump check a *condition code*
  - E.g., “jump if equal”, “jump if less than”, etc.
- Most ALU instructions set condition codes
- Most useful one is the `cmp` instruction

Comparing values

- `cmp` instruction: essentially the same as `sub`, except that it doesn't modify the "result" operand
  - Useful for comparing integer values
- Annoying quirk: AT&T syntax puts the operands in the opposite of the order you might expect
  - E.g., `cmpl %eax, %ebx` computes `%ebx - %eax` and sets condition codes appropriately

Conditional jump

Most often, we want to use the result of a comparison in order to influence a *conditional jump* instruction (used for implementing `if`/`else` logic and eventually-terminating loops)

Examples (^ means XOR, ~ means NOT, & means AND, | means OR):

Instruction	Condition for jump	Meaning
`je`, `jz`	ZF	jump if equal
`jl`	SF ^ OF	jump if less
`jle`	(SF ^ OF) | ZF	jump if less than or equal
`jg`	~(SF ^ OF) & ~ZF	jump if greater
`jge`	~(SF ^ OF)	jump if greater than or equal
`ja`	~CF & ~ZF	jump if above (unsigned)
`jae`	~CF	jump if above or equal (unsigned)
`jb`	CF	jump if below (unsigned)
`jbe`	CF | ZF	jump if below or equal (unsigned)

call and ret

- `call` instruction: calls procedure
  - `%rip` contains address of instruction following `call` instruction
  - Push `%rip` onto stack (as though `pushq %rip` was executed): this is the *return address*
  - Change `%rip` to address of first instruction of called procedure
  - Called procedure starts executing
- `ret` instruction: return from procedure
  - Pop saved return address from stack into `%rip` (as though `popq %rip` was executed)
  - Execution continues at return address

Stack alignment

- The Linux x86-64 calling conventions require `%rsp` to be a multiple of 16 at the point of a procedure call (to ensure that 16 byte values can be accessed on the stack if necessary)
- **Issue:** on entry to a procedure, `%rsp` $\mod 16 = 8$ because the `call` instruction (which called the procedure) pushed `%rip` (the program counter) onto the stack

Ensuring correct stack alignment

- To ensure correct stack alignment:
  - On procedure entry: `subq $8, %rsp`
  - Prior to procedure return: `addq $8, %rsp`
- The Linux `printf` function will segfault if the stack is misaligned

Register use conventions

- Very important issue:
  - There is only one set of registers
  - Procedures must share them
  - *Register use conventions* are rules that all procedures use to avoid conflicts
- Another important issue:
  - How are argument values passed to called procedures?
  - Calling conventions typically designate that some argument values are passed in specific registers
  - Procedure return value is typically returned in a specific register

x86-64 Linux register use conventions

- Arguments 1–6 passed in `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
  - Argument 7 and beyond, and "large" arguments such as pass-by-value struct data, passed on stack
- Integer or pointer return value returned in `%rax`
- Caller-saved registers: `%r10`, `%r11` (and also the argument registers)
- Callee-saved registers: `%rbx`, `%rbp`, `%r12`, `%r13`, `%14`, `%r15`

Caller-saved vs. callee-saved

- What happens to register contents when a procedure is called?
- *Callee-saved* registers: caller may assume that the procedure call will preserve their value
  - In general, all procedures must save their values to memory before modifying them, and restore them before returning
- *Caller-saved* registers: caller must *not* assume that the procedure call will preserve their value
  - In general any procedure can freely modify them
  - A caller might need to save their contents to memory prior to calling a procedure and restore the value afterwards

Using registers

- Using registers correctly and effectively is one of the main challenges of assembly language programming
- Some advice:
  - Use caller-saved registers (`%r10`, `%r11`, etc.) for very short-term temporary values or computations
  - You can use the argument registers as (caller-saved) temporary registers
    - Understand that called procedures could modify them!
  - Use callee-saved registers for longer term values that need to persist across procedure calls
    - Use `pushq`/`popq` to save and restore their values on procedure entry and exit

x86-64 tips

Know where to put things

- The `.section` directive specifies which "section" of the executable program assembled code or data will be placed in
- Put things in the right place!
- Code goes in `.text`
- Read-only data such as string constants go in `.rodata`
- Uninitialized (zero-filled) variables and buffers go in `.bss`
  - Use the `.space` directive to indicate how large these are
- Initialized (non-zero-filled) variables and buffers go in `.data`
  - There are various directives such as `.byte`, `.2byte`, `.4byte`, etc. to specify initialized data values

Labels

- Labels are names representing addresses of code or data in memory
- For functions and global variables, use appropriate names
  - Functions and data exported to other modules must be marked with `.globl`
- For control-flow targets within a function, use *local labels*
  - These are labels which start with `.L` (dot, followed by upper case L)
  - The assembler will not add these to the module's symbol table
  - Using "normal" labels for control flow makes debugging difficult because `gdb` thinks they are functions!

Using gdb

- You can debug assembly programs using gdb!
- "Debugging by adding print statements" is less practical for assembly programs than programs in a high level language
  - Which isn't to say it's not possible or (occasionally) useful
- Being able to use gdb confidently will greatly enhance your ability to develop working assembly language programs

gdb tips

- Set breakpoints (`break main`, `break myProg.S:123`)
- `where`: see current call stack
- `disassemble` (or just `disas`): display assembly code of current function (not necessary if code has debug symbols)
- `step`: step to next instruction
- `next`: step to next instruction (stepping over `call` instructions)
- Use $ prefix to refer to registers (e.g., `$rax`, `$edi`, etc.)
- Use `print` and casts to C data types when inspecting data:
  - Print 64 bit value %rsp points to: `print *(unsigned long *)$rsp`
  - Print character string %rdi points to: `print (char *)$rdi`
  - Print fourth element of array of `int` elements that %r12 points to: `print ((int *)$r12)[3]`

Code generation

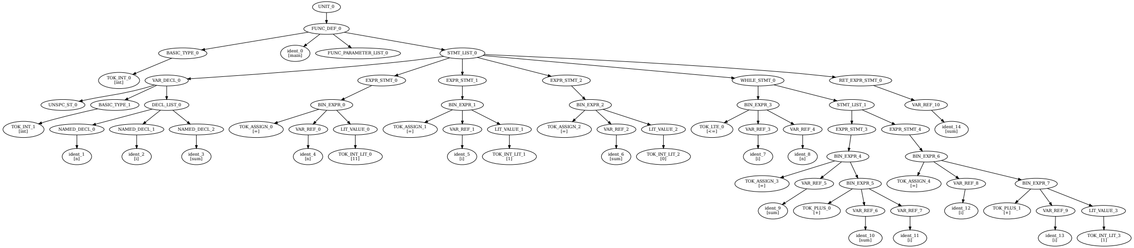
Initial code generation

- Important milestone in compiler development: generate working code
- Goal is to generate *working* code, not necessarily *efficient code*
- Later optimization passes improve code quality
- Approach: use control-flow graph as IR
  - Nodes are basic blocks
  - Each basic block is sequence of instructions
  - Jump instructions must be last
- Could generate "high-level" (machine-independent) instructions
- Or, could generate instructions equivalent to target assembly language

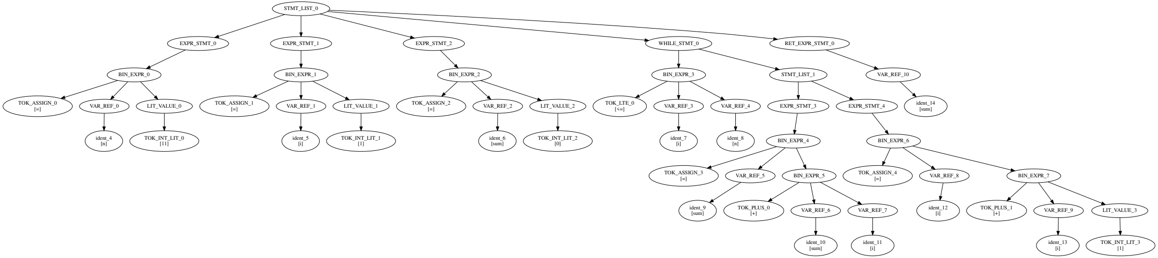
Example program

```
int main(void) {
  int n, i, sum;

  n = 11;
  i = 1;
  sum = 0;

  while (i <= n) {
    sum = sum + i;
    i = i + 1;
  }

  return sum;
}
```

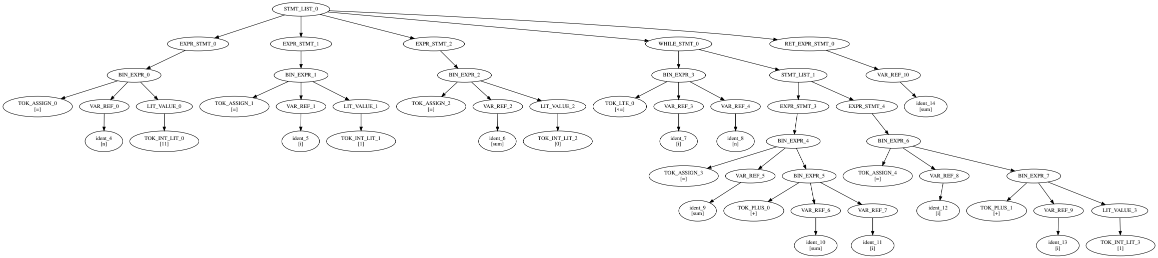
Example program AST



Example program AST (executable statements)



Example program AST (executable statements)



Code generation strategy

- Build symbol tables, determine storage requirements (size and offset) for variables
- Code generator is an AST visitor
  - Code generation is essentially a bottom-up process
- Assume registers can be allocated as needed
- Value computed by each expression is held in a register
- Scalar variables (e.g., `int`) can have virtual registers allocated as their storage

Let's do it!

In class: interactive example, results will be posted to Piazza