Lecture 6: Machine-level program representation

David Hovemeyer

February 4, 2022

601.229 Computer Systems Fundamentals

Compiling and executing a C program

Compilation

- There are many high-level programming languages (Java, Python, C, C++, ...)

Compilation

- There are many high-level programming languages (Java, Python, C, C++, ...)
- A computer can only directly execute *machine code*

Compilation

- There are many high-level programming languages (Java, Python, C, C++, ...)
- A computer can only directly execute *machine code*
- So, translation from high-level language code to machine code is necessary
- Strategies:

Compilation

- There are many high-level programming languages (Java, Python, C, C++, ...)
- A computer can only directly execute *machine code*
- So, translation from high-level language code to machine code is necessary
- Strategies:
  - Interpretation: a program "interprets" the high-level code and carries out the specified computation

Compilation

- There are many high-level programming languages (Java, Python, C, C++, ...)
- A computer can only directly execute *machine code*
- So, translation from high-level language code to machine code is necessary
- Strategies:
  - Interpretation: a program "interprets" the high-level code and carries out the specified computation
  - Compilation: a *compiler* program translates the high-level code into machine code

Compilation

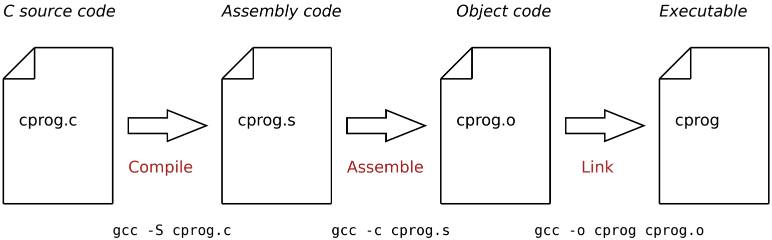
- There are many high-level programming languages (Java, Python, C, C++, ...)
- A computer can only directly execute *machine code*
- So, translation from high-level language code to machine code is necessary
- Strategies:
  - Interpretation: a program “interprets” the high-level code and carries out the specified computation
  - Compilation: a *compiler* program translates the high-level code into machine code
  - Hybrid strategies are possible (e.g., Java Virtual Machine)

Compiling C code

Example C program:

```
#include <stdio.h>
#include <stdlib.h>

long times10(long x) {
    long result = (x << 3) + (x << 1);
    return result;
}

int main(void) {
    printf("Enter value: ");
    long x;
    scanf("%ld", &x);
    long y = times10(x);
    printf("Result=%ld\n", y);
    return 0;
}
```

Compiling a C program



Compile and assemble steps are often combined (convert .c to .o), but they are still separate steps

C vs. assembly code

C code:

```
long times10(long x) {
    long result =
      (x << 3) + (x << 1);
    return result;
}
```

Assembly code:

```
times10:
    leaq (%rdi,%rdi), %rax
    leaq (%rax,%rdi,8), %rax
    ret
```

Assembly vs. machine code

Assembly code must be *assembled into machine code*:

Assembly code:	Machine code:
`times10:`	
`leaq (%rdi,%rdi), %rax`	`48 8d 04 3f`
`leaq (%rax,%rdi,8), %rax`	`48 8d 04 f8`
`ret`	`c3`

The CPU can directly decode and execute machine instructions

x86-64 assembly programming

Why learn assembly language?

- Since compilers exist, why learn how to write assembly code?

Why learn assembly language?

- Since compilers exist, why learn how to write assembly code?
  - Have complete control over hardware

Why learn assembly language?

- Since compilers exist, why learn how to write assembly code?
  - Have complete control over hardware
  - Understand hardware-level program execution
    - Important for understanding security vulnerabilities, and how to avoid introducing them

Why learn assembly language?

- Since compilers exist, why learn how to write assembly code?
  - Have complete control over hardware
  - Understand hardware-level program execution
    - Important for understanding security vulnerabilities, and how to avoid introducing them
  - Optimize performance-critical code

Why learn assembly language?

- Since compilers exist, why learn how to write assembly code?
  - Have complete control over hardware
  - Understand hardware-level program execution
    - Important for understanding security vulnerabilities, and how to avoid introducing them
  - Optimize performance-critical code
  - Implement code generators (compilers, JIT compilers)

x86-64 architecture

Selected “x86” processors

CPU	Vendor	Year	Bits	Note
8086	Intel	1978	16	
80386	Intel	1985	32	32-bit, virtual memory
Pentium	Intel	1993	32	
Pentium Pro	Intel	1995	32	
Pentium III	Intel	1999	32	
Pentium 4	Intel	2004	32	
Opteron	AMD	2003	64	First 64-bit x86 (“AMD64”)

Subsequent Intel CPUs adopted the AMD64 architecture (calling it “EM64T”)

Often called “x86-64” or just “x64”

x86-64 registers

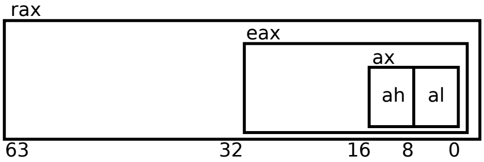
Register(s)	Note
`%rip`	Instruction pointer
`%rax`	Function return value
`%rdi`, `%rsi`	
`%rbx`, `%rcx`, `%rdx`	
`%rsp`, `%rbp`	Stack pointer, frame pointer
`%r8`, `%r9`, ..., `%r15`	

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`:



Memory

- Conceptually, memory is a big array of byte-sized storage locations
- Each location has an address
- In x86-64, addresses are 64 bit, so 2^{64} addresses
- In reality, there are additional details:
  - Actual x86-64 processors don't use all of the address bits
  - *Virtual memory* creates an arbitrary mapping of address to physical memory
  - Virtual memory is mapped "sparsely": only some ranges of addresses are mapped to actual memory

A C program

```
#include <stdio.h>

char buf[1000];
int arr[21];

int main(void) {
    int i, j;
    fgets(buf, 1000, stdin);
    for (i = 0; i < 21; i++)
        sscanf(buf + i*2, "%2x", &arr[i]);
    for (i = 0; i < 21; i++)
        printf("%c%s", arr[i], (i+1)%7 == 0 ? "\n" : "");
    return 0;
}
```

Running the C program

```
$ gcc -o art art.c

$ ./art
7C5C2D2D2D2F7C7C206F5F6F207C205C5F5E5F2F20
|\---/|
| o_o |
 \_^_/
```

:-)

Memory layout of C program

Using the pmap command to inspect the memory map of the running program:

```
29208:   ./art
0000562d71c36000      4K r-x-- art
0000562d71e36000      4K r---- art
0000562d71e37000      4K rw--- art
0000562d735fc000    132K rw---   [ anon ]
00007f7b5b9a5000   1948K r-x-- libc-2.27.so
00007f7b5bb8c000   2048K ----- libc-2.27.so
00007f7b5bd8c000     16K r---- libc-2.27.so
00007f7b5bd90000      8K rw--- libc-2.27.so
00007f7b5bd92000     16K rw---   [ anon ]
00007f7b5bd96000    156K r-x-- ld-2.27.so
00007f7b5bfa0000      8K rw---   [ anon ]
00007f7b5bfbd000      4K r---- ld-2.27.so
00007f7b5bfbe000      4K rw--- ld-2.27.so
00007f7b5bfbf000      4K rw---   [ anon ]
00007fff84484000    132K rw---   [ stack ]
00007fff845d4000     12K r----   [ anon ]
00007fff845d7000      8K r-x--   [ anon ]
ffffffffff600000      4K r-x--   [ anon ]
 total             4512K
```

Stack

- The *stack* is an extremely important runtime data structure
- Is a stack of *activation records*, a.k.a. “stack frames”
- A stack frame represents an in-progress function call, and contains
  - Return address (address of instruction where control should return when function returns)
  - Local variables
  - Temporary data
- 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!

- Assembly code = sequence of instructions
  - Executed sequentially

Assembly language!

- Assembly code = sequence of instructions
  - Executed sequentially (kind of, see Chapter 5)

Assembly language!

- Assembly code $=$ sequence of instructions
  - Executed sequentially (kind of, see Chapter 5)
- Each instruction has a mnemonic (`mov`, `push`, `add`, etc.)

Assembly language!

- Assembly code = sequence of instructions
  - Executed sequentially (kind of, see Chapter 5)
- 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)

Assembly language!

- Assembly code = sequence of instructions
  - Executed sequentially (kind of, see Chapter 5)
- 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)
- On Linux, the standard tools use "AT&T" assembly syntax
  - Source is first operand, destination is second

Assembly code structure, labels

- Assembly code generally specifies both code and data
  - Much like code written in a high level language
- A *label* marks the location of a chunk of code and/or data
  - Syntax:

```
nameOfLabel:
    labeled code or data
```

- When the assembly code eventually runs, its code and data are loaded into memory
- So, labels are synonymous with *memory addresses*
- In general, you can use labels as memory addresses in your assembly code

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 labels indicating the addresses of 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`	Content of memory location specified by 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)`	Copy 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

Clicker quiz!

Clicker quiz omitted from public slides

Assigning 32 bit value to 64 bit register

- Each 64 bit register has an alias for the lower 32 bits
  - `%rax`, `%eax`
  - `%rdi`, `%edi`
  - `%r10`, `%r10d`
  - etc.
- Storing a value in the low 32 bits *clears the upper 32 bits*
- E.g.:

```
movq $0xffffffffffffffff, %rax /* %rax initially contains ffffffffffffffff */
movl $1, %eax                  /* %rax now contains 1 */
```

Zero-extension, sign-extension

- When moving a smaller source value to a larger destination, sign-extension (copying sign bit to high bits of result) is necessary to preserve the value of a signed value
- E.g., representation of -16381 as 16 bit and 32 bit values:

Bits	Representation
16	1100000000000011
32	11111111111111111100000000000011

- Data movement with sign-extension: `movsbw`, `movsbl`, `movswl`, etc.
  - E.g., `movswl %ax, %edi`
- For unsigned values, data movement with zero-extension (copying 0 into high bits of result): `movzbw`, `movzbl`, `movzwl`, etc.

Example C program

```c
#include <stdio.h>

void addLongs(long x, long y, long *p) {
    *p = x + y;
}

int main(void) {
    long a, b, result;
    scanf("%ld", &a);
    scanf("%ld", &b);
    addLongs(a, b, &result);
    printf("Result is %ld\n", result);
    return 0;
}
```

Example assembly program

```
.section .rodata

longIntFmt:
    .string "%ld"
resultFmt:
    .string "Result is %ld\n"

.section .text

    .globl addLongs
addLongs:
    addq %rdi, %rsi
    movq %rsi, (%rdx)
    ret
```

```
    .globl main
main:
    pushq %rbp
    subq $32, %rsp
    movq %rsp, %rbp

    movq $longIntFmt, %rdi
    leaq 0(%rbp), %rsi
    call scanf

    movq $longIntFmt, %rdi
    leaq 8(%rbp), %rsi
    call scanf

    movq 0(%rbp), %rdi
    movq 8(%rbp), %rsi
    leaq 16(%rbp), %rdx
    call addLongs

    movq $resultFmt, %rdi
    movq 16(%rbp), %rsi
    call printf

    addq $32, %rsp
    popq %rbp
    ret
```

Example assembly program

```
.section .rodata

longIntFmt:
    .string "%ld"
resultFmt:
    .string "Result is %ld\n"

.section .text

    .globl addLongs
addLongs:
    addq %rdi, %rsi
    movq %rsi, (%rdx)
    ret
```

```
    .globl main
main:
    pushq %rbp
    subq $32, %rsp
    movq %rsp, %rbp

    movq $longIntFmt, %rdi
    leaq 0(%rbp), %rsi
    call scanf

    movq $longIntFmt, %rdi
    leaq 8(%rbp), %rsi
    call scanf

    movq 0(%rbp), %rdi
    movq 8(%rbp), %rsi
    leaq 16(%rbp), %rdx
    call addLongs

    movq $resultFmt, %rdi
    movq 16(%rbp), %rsi
    call printf

    addq $32, %rsp
    popq %rbp
    ret
```

Things to note:

Example assembly program

```
.section .rodata

longIntFmt:
    .string "%ld"
resultFmt:
    .string "Result is %ld\n"

.section .text

    .globl addLongs
addLongs:
    addq %rdi, %rsi
    movq %rsi, (%rdx)
    ret
```

```
    .globl main
main:
    pushq %rbp
    subq $32, %rsp
    movq %rsp, %rbp

    movq $longIntFmt, %rdi
    leaq 0(%rbp), %rsi
    call scanf

    movq $longIntFmt, %rdi
    leaq 8(%rbp), %rsi
    call scanf

    movq 0(%rbp), %rdi
    movq 8(%rbp), %rsi
    leaq 16(%rbp), %rdx
    call addLongs

    movq $resultFmt, %rdi
    movq 16(%rbp), %rsi
    call printf

    addq $32, %rsp
    popq %rbp
    ret
```

Things to note:

- The first three function parameters are passed in `%rdi`, `%rsi`, and `%rdx`

Example assembly program

```
.section .rodata

longIntFmt:
    .string "%ld"
resultFmt:
    .string "Result is %ld\n"

.section .text

    .globl addLongs
addLongs:
    addq %rdi, %rsi
    movq %rsi, (%rdx)
    ret
```

```
    .globl main
main:
    pushq %rbp
    subq $32, %rsp
    movq %rsp, %rbp

    movq $longIntFmt, %rdi
    leaq 0(%rbp), %rsi
    call scanf

    movq $longIntFmt, %rdi
    leaq 8(%rbp), %rsi
    call scanf

    movq 0(%rbp), %rdi
    movq 8(%rbp), %rsi
    leaq 16(%rbp), %rdx
    call addLongs

    movq $resultFmt, %rdi
    movq 16(%rbp), %rsi
    call printf

    addq $32, %rsp
    popq %rbp
    ret
```

Things to note:

- The first three function parameters are passed in `%rdi`, `%rsi`, and `%rdx`
- `(%rdx)` means the memory location pointed-to by `%rdx` (like pointer dereference)

Example assembly program

```
.section .rodata

longIntFmt:
    .string "%ld"
resultFmt:
    .string "Result is %ld\n"

.section .text

    .globl addLongs
addLongs:
    addq %rdi, %rsi
    movq %rsi, (%rdx)
    ret
```

```
    .globl main
main:
    pushq %rbp
    subq $32, %rsp
    movq %rsp, %rbp

    movq $longIntFmt, %rdi
    leaq 0(%rbp), %rsi
    call scanf

    movq $longIntFmt, %rdi
    leaq 8(%rbp), %rsi
    call scanf

    movq 0(%rbp), %rdi
    movq 8(%rbp), %rsi
    leaq 16(%rbp), %rdx
    call addLongs

    movq $resultFmt, %rdi
    movq 16(%rbp), %rsi
    call printf

    addq $32, %rsp
    popq %rbp
    ret
```

Things to note:

- The first three function parameters are passed in `%rdi`, `%rsi`, and `%rdx`
- `(%rdx)` means the memory location pointed-to by `%rdx` (like pointer dereference)
- `8(%rbp)` means the memory location at address `%rbp`+8

Example assembly program

```
.section .rodata

longIntFmt:
    .string "%ld"
resultFmt:
    .string "Result is %ld\n"

.section .text

    .globl addLongs
addLongs:
    addq %rdi, %rsi
    movq %rsi, (%rdx)
    ret
```

```
    .globl main
main:
    pushq %rbp
    subq $32, %rsp
    movq %rsp, %rbp

    movq $longIntFmt, %rdi
    leaq 0(%rbp), %rsi
    call scanf

    movq $longIntFmt, %rdi
    leaq 8(%rbp), %rsi
    call scanf

    movq 0(%rbp), %rdi
    movq 8(%rbp), %rsi
    leaq 16(%rbp), %rdx
    call addLongs

    movq $resultFmt, %rdi
    movq 16(%rbp), %rsi
    call printf

    addq $32, %rsp
    popq %rbp
    ret
```

Things to note:

- The first three function parameters are passed in `%rdi`, `%rsi`, and `%rdx`
- `(%rdx)` means the memory location pointed-to by `%rdx` (like pointer dereference)
- `8(%rbp)` means the memory location at address `%rbp`+8
- `leaq 16(%rbp), %rdx` means compute the address `%rbp`+16 and store it in `%rdx` (like address-of)

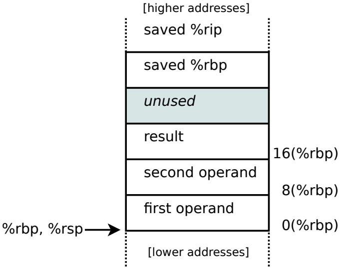
Example assembly program (continued)

```
.section .rodata

longIntFmt:
    .string "%ld"
resultFmt:
    .string "Result is %ld\n"

.section .text

    .globl addLongs
addLongs:
    addq %rdi, %rsi
    movq %rsi, (%rdx)
    ret
```

```
    .globl main
main:
    pushq %rbp
    subq $32, %rsp
    movq %rsp, %rbp

    movq $longIntFmt, %rdi
    leaq 0(%rbp), %rsi
    call scanf

    movq $longIntFmt, %rdi
    leaq 8(%rbp), %rsi
    call scanf

    movq 0(%rbp), %rdi
    movq 8(%rbp), %rsi
    leaq 16(%rbp), %rdx
    call addLongs

    movq $resultFmt, %rdi
    movq 16(%rbp), %rsi
    call printf

    addq $32, %rsp
    popq %rbp
    ret
```

Things to note:

- 40 bytes are allocated within `main`'s stack frame, including 24 bytes for local variables:



`%rbp` is used to access the local variables