CSE 351 Section 4 – x86-64 Assembly

Hi there! Welcome back to section, we're happy that you're here ☺

x86-64 Assembly Language

Assembly language is a human-readable representation of machine code instructions (generally a one-to-one correspondence). Assembly is machine-specific because the computer architecture and hardware are designed to execute a particular machine code instruction set.

x86-64 is the primary 64-bit instruction set architecture (ISA) used by modern personal computers. It was developed by Intel and AMD and its 32-bit predecessor is called IA32. x86-64 is designed for complex instruction set computing (CISC), generally meaning it contains a larger set of more versatile and more complex instructions.

For this course, we will utilize only a small subset of x86-64's instruction set and omit floating point instructions.

x86-64 Instructions

The subset of x86-64 instructions that we will use in this course take either one or two operands, usually in the form: `instruction operand1, operand2`. There are three options for operands:

- Immediate: constant integer data (*e.g.* `$0x400`, `$-533`) or an address/label (*e.g.* `Loop`, `main`)
- Register: use the data stored in one of the 16 general purpose registers or subsets (*e.g.* `%rax`, `%edi`)
- Memory: use the data at the memory address specified by the addressing mode `D(Rb,Ri,S)`

The operation determines the effect of the operands on the processor state and has a suffix ("`b`" for byte, "`w`" for word, "`l`" for long, "`q`" for quad word) that determines the bit width of the operation. Sometimes the operation size can be inferred from the operands, so the suffix is omitted for brevity.

Control Flow and Condition Codes

Internally, condition codes (Carry, Zero, Sign, Overflow) are set based on the result of the previous operation. The `j*` and `set*` families of instructions use the values of these "flags" to determine their effects. See the table provided on your reference sheet for equivalent conditionals.

An *indirect jump* is specified by adding an asterisk (*) in front of a memory operand and causes your program counter to load the address stored at the computed address. (e.g. `jmp *%rax`) This is useful for switch case statements

Procedure Basics

The instructions `push`, `pop`, `call`, and `ret` move the stack pointer (`%rsp`) automatically.

`%rax` is used for the return value and the first six arguments go in `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
("**D**iane's **S**ilk **D**ress **C**ost $**89**").

x86 instructions	English equivalent
`movq $351, %rax`	Move the number 351 into 8-byte (quad) register "rax"
`addq %rdi, %rsi`	Add the 64-bit value of %rdi to %rsi
`movq (%rdi), %r8`	Move the 64-bit data at the address stored in %rdi to %r8
`leaq (%rax,%rax,8), %rax`	Compute 9 * %rax, and store the 64-bit result in %rax

Exercises:

1. [CSE351 Au14 Midterm] Symbolically, what does the following code return?

```
movl   (%rdi), %eax           # %rdi -> x;   r = *x
leal   (%eax,%eax,2), %eax    # %rax -> r;   r = (*x) * 3
addl   %eax, %eax             #              r = (*x)*3 + (*x)*3
andl   %esi, %eax             # %rsi -> y;   r = ((*x)*6) & y
subl   %esi, %eax             #              r = (((*x)*6) & y) - y
ret
```

(((*x) * 6) & y) - y

2. [CSE351 Au15 Midterm] Convert the following C function into x86-64 assembly code. You are not being judged on the efficiency of your code – just the correctness.

```
long happy(long *x, long y, long z) {
    if (y > z)
        return z + y;
    else
        return *x;
}
```

```
happy:
    cmpq   %rdx, %rsi
    jle    .else
    leaq   (%rdx, %rsi), %rax
    ret
.else:
    movq   (%rdi), %rax
    ret
```

Multiple other possibilities (*e.g.* switch ordering of if/else clauses, replace `lea` with `mov`/`add` instruction pair).

3. Write an equivalent C function for the following x86-64 code:

```
mystery:
1    testl    %edx, %edx              # %edx is 3rd argument (z)
2    js       .L3                     # jump to .L3 if z<0
3    cmpl     %esi, %edx              # %esi is 2nd argument (y)
4    jge      .L3                     # jump to .L3 if y<=z
5    movslq   %edx, %rdx              # sign-extend 3rd argument (z)
6    movl     (%rdi,%rdx,4), %eax     # %rdi is 1st argument (x), calc *(x + z*4)
7    ret
.L3:
8    movl     $0, %eax                # return 0
9    ret
```

```
int mystery(int *x, int y, int z) {
    if (z >= 0 && z < y)
        return x[z];
    else
        return 0;
}
```

Notes:

- If either conditional is True, then we jump to the "else" clause, so in C we execute the "if" clause only when the complement of both of them are True.
- Line 6 indicates that the return type is 4 bytes (`int`). Line 8 is ambiguous since it zeros out the entire 8 bytes of `%rax`.
- Argument variable names are arbitrary. Based on usage, could perhaps have used `x`→`ar`, `y`→`n`, `z`→`k`.
- First argument had to point to `int` based on scale factor in Line 6. Both `int *x` and `int x[]` work.

4. [CSE351 Wi17 Midterm] Consider the following x86-64, (partially blank) C code, and memory diagram. Addresses and values are 64-bit. Fill in the C code based on the given assembly.

```
foo:
  movl    $0,     %eax

L1:
  testq   %rdi,   %rdi
  je      L2
  movq    (%rdi), %rdi
  addl    $1,      %eax
  jmp     L1

L2:
  ret
```

```
int foo(long* p) {
  int result = 0;
  while (p != NULL) {
    p = *(long**)p;
    result = result + 1;
  }
  return result;
}
```

Part 2: Follow the execution of foo in assembly, where `0x1000` is passed in to `%rdi`

Write the values of `%rdi` and `%eax` in the columns. If the value doesn't change, you can leave it blank

Instruction	%rdi (hex)	%eax (decimal)
`movl`	`0x1000`	`0`
`testq`		
`je`		
`movq`	`0x1030`	
`addl`		`1`
`jmp`		
`testq`		
`je`		
`movq`	`0x0`	
`addl`		`2`
`jmp`		
`testq`		
`je`		
`ret`		

Address	Value
`0x1000`	`0x1030`
`0x1008`	`0x1020`
`0x1010`	`0x1000`
`0x1018`	`0x0000`
`0x1020`	`0x1030`
`0x1028`	`0x1008`
`0x1030`	`0x0000`
`0x1038`	`0x1038`
`0x1040`	`0x1048`
`0x1048`	`0x1040`

5. Log on to Gradescope and start the "GDB Tutorial (optional)" assignment.
   This includes the basic workflow on how to use GDB, and should prove very useful for Lab 2 and beyond (Q4 even includes a walkthrough of Lab 2 Phase 1).