CSE 351 Lecture 8 – x86 Programming I

Instruction Set Architectures (ISAs)

An ***instruction set architecture (ISA)*** is defined as the combination of parts of the processor design that one needs to understand to write *assembly code*. It is the contract between programmers and hardware designers to guarantee consistent execution of the chosen assembly language on an implementation of the ISA. The ISA consists of:

- The system's ***state*** – all of the information that uniquely defines the culmination of past instructions,
- The ***instruction set*** – the list and format of all instructions that the CPU can execute, and
- The ***effect*** on the system state of each instruction.

There are two main ISA design philosophies, though the distinction is somewhat arbitrary and all ISAs fall somewhere on a spectrum between these two:

- Complex Instruction Set Computer (CISC): Continue to add more and more elaborate and specialized instructions over time as needed. Good for backwards compatibility and giving programmers tons of tools to work with, but bad for the hardware designers.
- Reduced Instruction Set Computer (RISC): Restrict the ISA to a small set of very general instructions. This makes it easier to build faster hardware, but complex software tasks must now be composed of many smaller instructions.

We will be studying x86-64 in this class, which is on the far end of CISC. To narrow the scope of x86-64 for this course, we will look only at *integral* data and use "*AT&T*" syntax.

---

x86-64 Instructions and Operands

In x86-64 assembly, instructions are specified using a short *instruction name* followed by 0-3 *operands*, separated by commas. In this class, we will primarily look at instructions that take 1 or 2 operands. In AT&T syntax, these generally fit the forms (note that '#' indicates a comment in AT&T syntax):

```
instr op        # e.g. "negq %rsi" negates the value in %rsi
instr src, dst  # e.g. "addq %rdi, %rax" does %rax = %rax + %rdi
```

Most x86-64 assembly instructions can be broadly clumped into the following categories:

1) **Data transfer** – copying data from one place to a specified location
2) **Arithmetic and logical operations**
3) **Control flow** – affects which instruction to execute next

A list of the most common assembly instructions we will use will be found on the exam reference sheets or lists can be found in the textbook in sections CSPP § 3.4-3.7.2.

The operands for x86-64 assembly instructions can be one of three types:

1) ***Immediates*** – constant integer data (look for the prefix '`$`').
2) **Registers** – the name of any of the 16 general purpose registers (look for the prefix '`%`').
3) **Memory** – a specified address that is usually dereferenced (look for parentheses '`()`').

For binary instructions (*i.e.* those that take two operands), these can be used in any combination *except*:

- You cannot use an Immediate as your destination operand.
- You cannot do a direct memory to memory operation (*i.e.* both operands can't be Memory).

---

x86-64 Registers

A ***register*** is a location in the CPU that stores a small amount of data (a word size), which can be accessed very quickly. There are a fixed number of them in every architecture and they are referred to by *name*. In x86-64, there are only 16 general purpose registers that the programmer can use. Register names are given to the word-size (8 bytes) data as well as smaller divisions as follows:

%rax	%eax	%ax	%al	%r8	%r8d	%r8w	%r8b
%rbx	%ebx	%bx	%bl	%r9	%r9d	%r9w	%r9b
%rcx	%ecx	%cx	%cl	%r10	%r10d	%r10w	%r10b
%rdx	%edx	%dx	%dl	%r11	%r11d	%r11w	%r11b
%rsi	%esi	%si	%sil	%r12	%r12d	%r12w	%r12b
%rdi	%edi	%di	%dil	%r13	%r13d	%r13w	%r13b
%rsp	%esp	%sp	%spl	%r14	%r14d	%r14w	%r14b
%rbp	%ebp	%bp	%bpl	%r15	%r15d	%r15w	%r15b
8 bytes	4 bytes	2 bytes	1 byte	8 bytes	4 bytes	2 bytes	1 byte

---

Memory Addressing Modes

The most general way to express a Memory operand has 4 parts and the form: `D(Rb,Ri,S)`

- *Displacement* (`D`) – constant displacement value (must be an immediate)
- *Base register* (`Rb`) – name of the register whose value will act as the "base" of our address calculation
- *Index register* (`Ri`) – name of the register whose value will be scaled and added to the base
- *Scale factor* (`S`) – scales the value in Ri by the specified number, which can only be 1, 2, 4, or 8

The computed address is **`Reg[Rb]+Reg[Ri]*S+D`**, where `Reg[]` means "the value in the register specified." Most instructions will dereference this address when passed a Memory operand. When any of the parts are omitted, their corresponding values default to `Reg[Rb]` = `0`, `Reg[Ri]` = `0`, `S` = `1`, `D` = `0`.