The x86-64 instruction set architecture (ISA) is used by most laptop and desktop processors. We will be embedding assembly into some of our C++ code to explore programming in assembly language.

Depending on the source, the name for this instruction set is generally known as x86-64, x86_64, AMD64, and x64.

Some of the material here has been drawn from the GCC Inline Assembly HOWTO `https://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html`.

1 Registers

We will focus only on the 16 general purpose (integer) registers storing 64-bit values. Other registers do exist such as those for SIMD and floating point operations, but they are beyond the scope of this class.

The original 8086 (1976) had the eight 16–bit registers `ax`, `bx`, `cx`, `dx`, `si`, `di`, `bp`, `sp`. IA-32 (1986) extended these eight registers to 32–bit, and gave the 32–bit sizes names `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp`, and `esp`. It also preserved accessing the lower 16 bits using the names devised by the 8086. After Intel went off in its own direction for a 64–bit processor with the Itanium, AMD created the x86-64 ISA when it introduced the Opteron in 2003. The Itanium ended up being a commercial failure, and ultimately Intel also adopted the x64-64 ISA. It extends the registers of the 8086 and IA-32 to 64–bits under the names `rax`, `rbx`, `rcx`, `rdx`, `rsi`, `rdi`, `rbp`, and `rsp`. It also introduced eight additional registers `r8`, `r9`, `r10`, `r11`, `r12`, `r13`, `r14`, and `r15`. Addressing the smaller forms were also preserved, and can be accessed.

For the original eight registers, the lowest byte can be accessed by replacing the letter `x` with the letter `l` in their register name, e.g. `al`. For the eight new registers, they can be accessed by appending the letter `b` to the register name, e.g. `r8b`.

The lowest 16–bits, known as a word in this context, can be accessed for the original eight registers by their 8086 names, e.g. `ax`. For the new registers, they are accessed by appending the letter `w` to their name, e.g. `r8w`.

The lowest 32–bits, known as a double word in this context, can be accessed for the original eight registers by their IA-32 name which prepends the letter `e` to their 8086 names, e.g. `eax`. For the new registers, they are accessed by appending the letter `d` to their names, e.g. `r8d`.

The full 64–bits, known as a quad word in this context, can be accessed for the original eight registers by prepending the letter `r` to their 8086 names, e.g. `rax`. For the new registers, they are accessed by just their name with no suffixes, e.g. `r8`.

Many of the operations allow for specifying which registers to use, but some require their operands in specific registers.

2 Embedding Assembly into GCC

Generally speaking, C++ compilers allow for a programmer to directly embed assembly into the middle of a function. It is not often used modernly because in most situations a compiler will write more performant code and assembly is not portable between processors. The rare cases that I still see it are to access specific hardware features or new operations that a compiler does not support. These happen in things like an operating system kernel, and

very specialized code where efficiency is paramount like video decoders. Although we will not discuss these in this class, if you find yourself in a situation where you think you need straight assembly, first look into the compiler intrinsics that are supported by your compiler. Often those will allow you to accomplish what you need in a simpler manner.

The method of embedding assembly is not part of ISO C++ (or ISO C), so we will focus on how GCC extends the language to allow for direct assembly. Since a design goal for Clang is to be a drop-in replacement for GCC, this syntax is also supported there. The general structure is as follows

```
asm ( assembler code
    : output operands                   /* optional */
    : input operands                    /* optional */
    : list of clobbered registers       /* optional */
    );
```

The assembler code is in AT&T syntax by default. This is in contrast to Intel syntax which is seen more often. Translating between the two syntaxes is generally not too difficult. The main differences are

1. Source–Destination Ordering. In Intel syntax an operation is normally ordered destination–source, whereas AT&T uses source–destination.

2. Register Naming. In AT&T syntax, register names are prefixed by a `%`, such as `%eax`.

3. Immediate Operands. In AT&T syntax, immediate operands (constants) are prefixed by a `$`.

4. Operand Size. In AT&T syntax the size of memory operands is determined from the last character of the op-code name. Op-code suffixes of `b`, `w`, and `l` specify byte(8-bit), word(16-bit), and long(32-bit) memory references. Intel syntax accomplishes this by prefixing memory operands (not the op-codes) with byte ptr, word ptr, and dword ptr.

   Thus, Intel "`mov al, byte ptr foo`" is "`movb foo, %al`" in AT&T syntax.

Other differences do exist. To learn about them consult the manual for GNU gas `https://www.gnu.org/software/binutils/manual/`.

The code itself can be specified in several ways, but the simplest to use is to put each op-code into its own double quoted string with semicolons at the end. For example

```
int negateIntAsm(int in) {
  asm("xor $0xffffffff, %0;"
      "add $0x1, %0;"
      : "=r" (in) // output
      : "0" (in) // input
      : // clobbered registers
      );
  return in;
}
```

is a simple function that corresponds to the following C++ function.

```
int negateInt(int in) {
  in = ~in + 1;

  return in;
}
```

3 Determining the op-codes to use

There is some inherit difficulty in translating code to assembly. You have to have a large knowledge of the specific ISA to select the right op-codes. So we will see how to get the compiler to tell us.

First we create a simple source code file that has just the function we wish to translate, say

```
int negateInt(int in) {
  in = ~in + 1;

  return in;
}
```

We then compile this with no optimizations and the debugging symbols. On a Linux system this would be "`g++ -O0 -g -std=c++11 -c -o file.o file.cpp`". This will create a file named `file.o` which is the compiled form of `file.cpp`. Next we run objdump on it as "`objdump -S file.o > file.S`". This will create a file named `file.S` which will contain the annotated assembly.

```
test.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <_Z9negateInti>:
int negateInt(int in) {
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   89 7d fc                mov    %edi,-0x4(%rbp)
  in = ~in + 1;
   7:   f7 5d fc                negl   -0x4(%rbp)

  return in;
   a:   8b 45 fc                mov    -0x4(%rbp),%eax
}
   d:   5d                      pop    %rbp
   e:   c3                      retq
```

We see the lines of assembly interspersed with the original C++ code. Note the `negl` operation. This is the GCC performing an optimization on our code even though we requested

that it not do so. Unfortunately there is not much we can do with this, GCC will optimize it regardless. However, if we use Clang to compile instead we get a difference assembly listing.

```
test.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <_Z9negateInti>:
int negateInt(int in) {
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   89 7d fc                mov    %edi,-0x4(%rbp)
  in = ~in + 1;
   7:   8b 7d fc                mov    -0x4(%rbp),%edi
   a:   83 f7 ff                xor    $0xffffffff,%edi
   d:   83 c7 01                add    $0x1,%edi
  10:   89 7d fc                mov    %edi,-0x4(%rbp)

  return in;
  13:   8b 45 fc                mov    -0x4(%rbp),%eax
  16:   5d                      pop    %rbp
  17:   c3                      retq
```

In this listing we can see the explicit xor and add instructions. It is from this Clang listing we can see the translation of the function to embedded assembly.

```
int negateIntAsm(int in) {
  asm("xor $0xffffffff, %0;"
      "add $0x1, %0;"
      : "=r" (in) // output
      : "0" (in) // input
      : // clobbered registers
      );
  return in;
}
```

Since writing assembly from scratch is tedious, we will instead use this approach of compiling the code, looking at the result, and translating it back to embedded assembly.

4 Common Structures and How They Translate to Assembly

Since we are "cheating" by using the compiler, we really only need to recognize how some structures will translate to a simplified C++ syntax. From there we can guess how our statement translated by comparing the original to the generated assembly.

To get precise definitions of what each operation does, consult the Intel Software Developer Manuals: `https://www-ssl.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`

4.1 Conditional Statements

Given a conditional statement

```
if (test-expr)
  then-statement
else
  else-statement
```

this translates to a psuedo C++ syntax as

```
  t = test-expr;
  if (!t)
    goto false;
  then-statement
  goto done;
false:
  else-statement
done:
```

Note that this is still mostly valid C++, but it is closer to how the assembly will be. Then the actual assembly is easy to see with gotos changed to various jump statements (jmp, je, jne, etc).

4.2 Loops

Given a basic loop

```
do
  body-statement
  while (test-expr);
```

this translates to a pseudo C++ syntax as

```
loop:
  body-statement
  t = test-expr;
  if (t)
    goto loop;
```

Again, the translation from here to assembly should be straightforward.