CS107 Lecture 14
Introduction to Assembly

Reading: B&O 3.1-3.4

CS107 Topic 5: Assembly

How does a computer **compile** C programs to **assembly code** and then **execute** them? And what does **assembly code look like**?

Why is answering this question important?

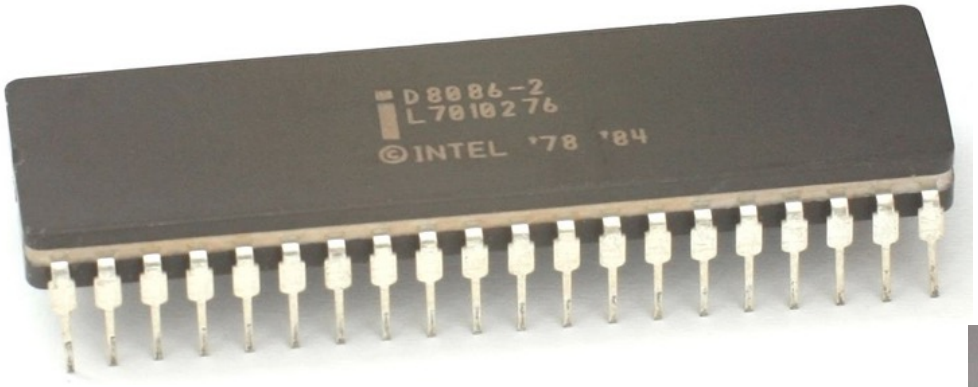
- Learning how our code is translated **helps us write better code**.
- We'll understand how to **reverse engineer programs** given their assembly.
- We'll understand how **program logic is ultimately represented using binary numbers**, just as `int`s, strings, arrays, and records are.

gcc: Our Machine Code Duolingo

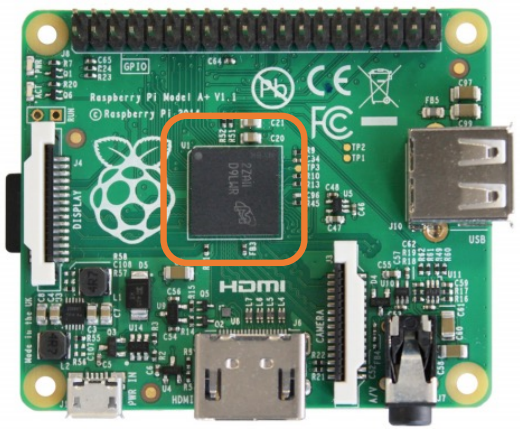
- **gcc** is the compiler that **converts your human-readable code into machine-readable instructions**.
- C and many other languages **define higher-level abstractions** that we can use to **efficiently and elegantly implement complex programs**.
  - Hardware, however, **doesn't understand** aggregate data structures, types, etc.
- **Machine code** is all 1's and 0's. **Truly everything is encoded using bits**.
  - We'll read it using an equivalent, human-readable called **assembly**.
- Often a single C statement **compiles to a single assembly code instruction**, and other times it **compiles to two or more assembly code instructions**.

Central Processing Units (CPUs)

Intel 8086, 16-bit microprocessor ($86.65, 1978)



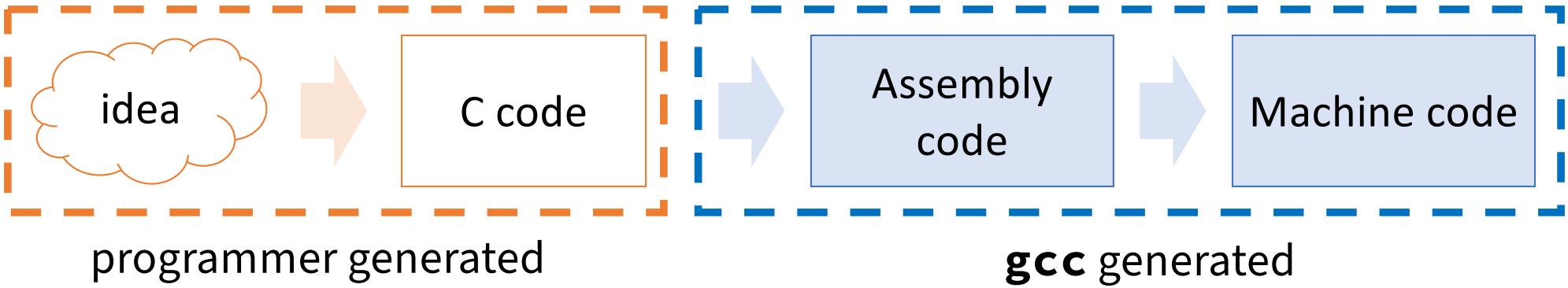
We are going to learn the **x86-64 instruction set architecture**. This is the instruction set is used by Intel and AMD.



Raspberry Pi BCM2836
32-bit **ARM** microprocessor
($35 for everything, 2015)

Intel Core i9-9900K 64-bit
8-core multi-core processor
($449, 2018)

Why are we reading assembly?



Main goal: **understanding** what assembly code does

- We will **not** be writing assembly!
- Rather, we want to translate assembly **back** into equivalent C code.
- Knowing how our C programs are converted into assembly **offers insight into how me might write cleaner, more efficient code**.

Demo: Looking at an Executable (`objdump -d`)

Baby's First Assembly

```
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}
```

What does this look like in assembly?

```
make
objdump -d sum
```

```
0000000000401136 <sum_array>:
  401136:    b8 00 00 00 00          mov    $0x0,%eax
  40113b:    ba 00 00 00 00          mov    $0x0,%edx
  401140:    39 f0                   cmp    %esi,%eax
  401142:    7d 0b                   jge    40114f <sum_array+0x19>
  401144:    48 63 c8                movslq %eax,%rcx
  401147:    03 14 8f                add    (%rdi,%rcx,4),%edx
  40114a:    83 c0 01                add    $0x1,%eax
  40114d:    eb f1                   jmp    401140 <sum_array+0xa>
  40114f:    89 d0                   mov    %edx,%eax
  401151:    c3                      retq
```

Baby's First Assembly

```
0000000000401136 <sum_array>:
  401136:  b8 00 00 00 00          mov    $0x0,%eax
  40113b:  ba 00 00 00 00          mov    $0x0,%edx
  401140:  39 f0                   cmp    %esi,%eax
  401142:  7d 0b                   jge    40114f <sum_array+0x19>
  401144:  48 63 c8                movslq %eax,%rcx
  401147:  03 14 8f                add    (%rdi,%rcx,4),%edx
  40114a:  83 c0 01                add    $0x1,%eax
  40114d:  eb f1                   jmp    401140 <sum_array+0xa>
  40114f:  89 d0                   mov    %edx,%eax
  401151:  c3                      retq
```

Baby's First Assembly

```
0000000000401136 <sum_array>:
  401136:  b8 00 00 00 00        mov    $0x0,%eax
  40113b:  ba 00 00 00 00        mov    $0x0,%edx
                                 p      %esi,%eax
                                 e      40114f <sum_array+0x19>
                                 vslq   %eax,%rcx
                                 d      (%rdi,%rcx,4),%edx
                                 dd     $0x1,%eax
  40114d:  eb f1                 jmp    401140 <sum_array+0xa>
  40114f:  89 d0                 mov    %edx,%eax
  401151:  c3                    retq
```

This is the **name of the function** and the **address in memory** where the code for this function begins.

Baby's First Assembly

```
0000000000401136 <sum_array>:
  401136: b8 00 00 00 00        mov    $0x0,%eax
  40113b: ba 00 00 00 00        mov    $0x0,%edx
  401140: 39 f0                 cmp    %esi,%eax
  401142: 7d                                    y+0x19>
  401144: 48
  401147: 03                                    dx
  40114a: 83
  40114d: eb                    jmp    401140 <sum_array+0xa>
  40114f: 89 d0                 mov    %edx,%eax
  401151: c3                    retq
```

These are the **memory addresses** where each of the instructions live. **Consecutive instructions are sequential in memory**.

Baby's First Assembly

```
0000000000401136 <sum_array>:
  401136:  b8 00 00 00 00
  40113b:  ba 00 00 00 00
  401140:  39 f0
  40114d:  eb f1
  40114f:  89 d0
  401151:  c3
```

This is the **assembly code**: **human readable** versions of each machine code instruction.

```
mov     $0x0,%eax
mov     $0x0,%edx
cmp     %esi,%eax
jge     40114f <sum_array+0x19>
movslq  %eax,%rcx
add     (%rdi,%rcx,4),%edx
add     $0x1,%eax
jmp     401140 <sum_array+0xa>
mov     %edx,%eax
retq
```

Baby's First Assembly

```
0000000000401136 <sum_array>:
  401136:   b8 00 00 00 00
  40113b:   ba 00 00 00 00
  401140:   39 f0
  401142:   7d 0b
  401144:   48 63 c8
  401147:   03 14 8f
  40114a:   83 c0 01
  40114d:   eb f1
  40114f:   89 d0
  401151:   c3
```

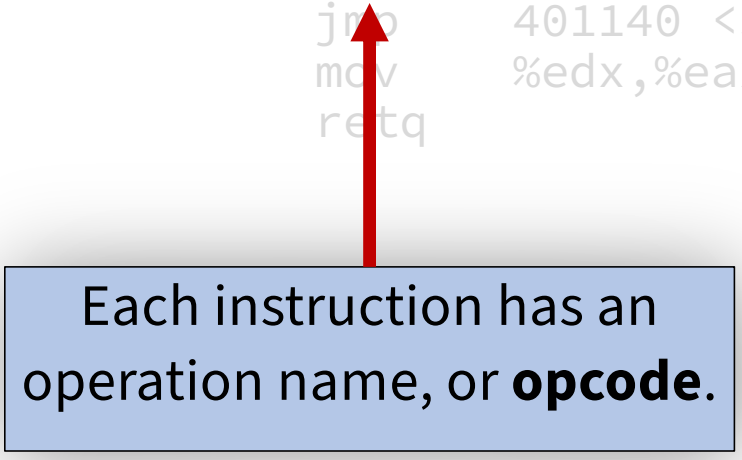
This is the **machine code**: raw hexadecimal encodings that mean something to the computer. Different instructions require a different number of bytes.

Baby's First Assembly

```
0000000000401136 <sum_array>:
  401136:  b8 00 00 00 00          mov    $0x0,%eax
  40113b:  ba 00 00 00 00          mov    $0x0,%edx
  401140:  39 f0                   cmp    %esi,%eax
  401142:  7d 0b                   jge    40114f <sum_array+0x19>
  401144:  48 63 c8                movslq %eax,%rcx
  401147:  03 14 8f                add    (%rdi,%rcx,4),%edx
  40114a:  83 c0 01                add    $0x1,%eax
  40114d:  eb f1                   jmp    401140 <sum_array+0xa>
  40114f:  89 d0                   mov    %edx,%eax
  401151:  c3                      retq
```

Baby's First Assembly

```
0000000000401136 <sum_array>:
  401136:  b8 00 00 00 00          mov    $0x0,%eax
  40113b:  ba 00 00 00 00          mov    $0x0,%edx
  401140:  39 f0                   cmp    %esi,%eax
  401142:  7d 0b                   jge    40114f <sum_array+0x19>
  401144:  48 63 c8                movslq %eax,%rcx
  401147:  03 14 8f                add    (%rdi,%rcx,4),%edx
  40114a:  83 c0 01                add    $0x1,%eax
  40114d:  eb f1                   jmp    401140 <sum_array+0xa>
  40114f:  89 d0                   mov    %edx,%eax
  401151:  c3                      retq
```



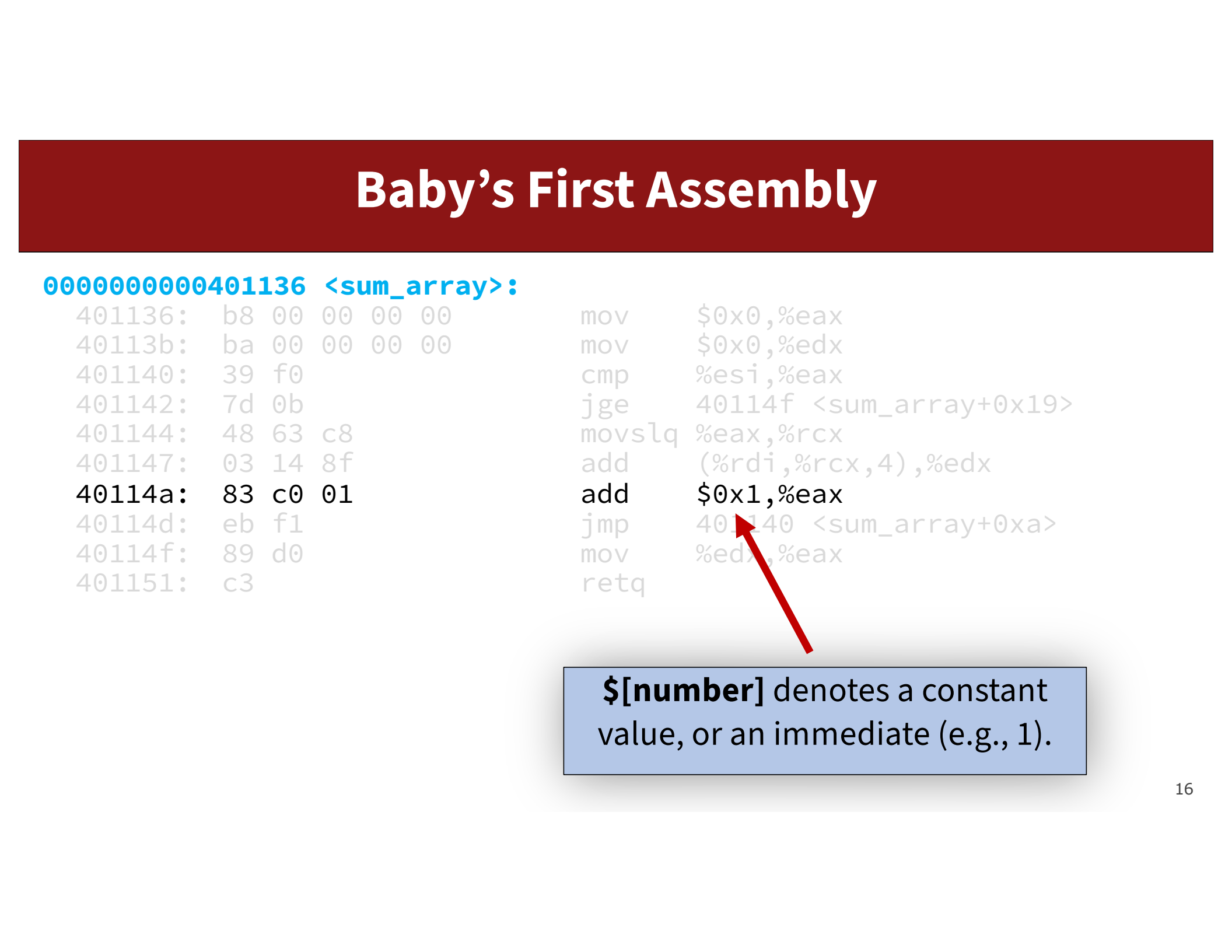
Baby's First Assembly

```
0000000000401136 <sum_array>:
  401136:  b8 00 00 00 00          mov    $0x0,%eax
  40113b:  ba 00 00 00 00          mov    $0x0,%edx
  401140:  39 f0                   cmp    %esi,%eax
  401142:  7d 0b                   jge    40114f <sum_array+0x19>
  401144:  48 63 c8                movslq %eax,%rcx
  401147:  03 14 8f                add    (%rdi,%rcx,4),%edx
  40114a:  83 c0 01                add    $0x1,%eax
  40114d:  eb f1                   jmp    401140 <sum_array+0xa>
  40114f:  89 d0                   mov    %edx,%eax
  401151:  c3
```

Each instruction can also have arguments, or operands.

Baby's First Assembly

```
0000000000401136 <sum_array>:
  401136:  b8 00 00 00 00        mov    $0x0,%eax
  40113b:  ba 00 00 00 00        mov    $0x0,%edx
  401140:  39 f0                 cmp    %esi,%eax
  401142:  7d 0b                 jge    40114f <sum_array+0x19>
  401144:  48 63 c8              movslq %eax,%rcx
  401147:  03 14 8f              add    (%rdi,%rcx,4),%edx
  40114a:  83 c0 01              add    $0x1,%eax
  40114d:  eb f1                 jmp    401140 <sum_array+0xa>
  40114f:  89 d0                 mov    %edx,%eax
  401151:  c3                    retq
```



Baby's First Assembly

```
0000000000401136 <sum_array>:
  401136:  b8 00 00 00 00          mov    $0x0,%eax
  40113b:  ba 00 00 00 00          mov    $0x0,%edx
  401140:  39 f0                   cmp    %esi,%eax
  401142:  7d 0b                   jge    40114f <sum_array+0x19>
  401144:  48 63 c8                movslq %eax,%rcx
  401147:  03 14 8f                add    (%rdi,%rcx,4),%edx
  40114a:  83 c0 01                add    $0x1,%eax
  40114d:  eb f1                   jmp    401140 <sum_array+0xa>
  40114f:  89 d0                   mov    %edx,%eax
  401151:  c3                      retq
```

%[name] identifies a **register** on the CPU (e.g., **%eax**).

Assembly Abstraction

C **abstracts away** machine-level details.

- Instead of manipulating registers and raw bytes, we program with variables, types, functions, and control structures. The compiler translates those higher-level constructs into the specific instructions and data layouts required by the hardware.

C is **largely portable** across machines.

- C code can be compiled for many different processor architectures and operating systems, and the compiler manages the machine-specific translation. Provided you code in standard C and avoid system-specific features, your code can run on most machines.

Machine code is **just bytes**.

- A program is a sequence of instruction encodings and data values stored in memory. There are no variable names, no types, no safety checks—only raw opcodes and operands interpreted directly by the processor.

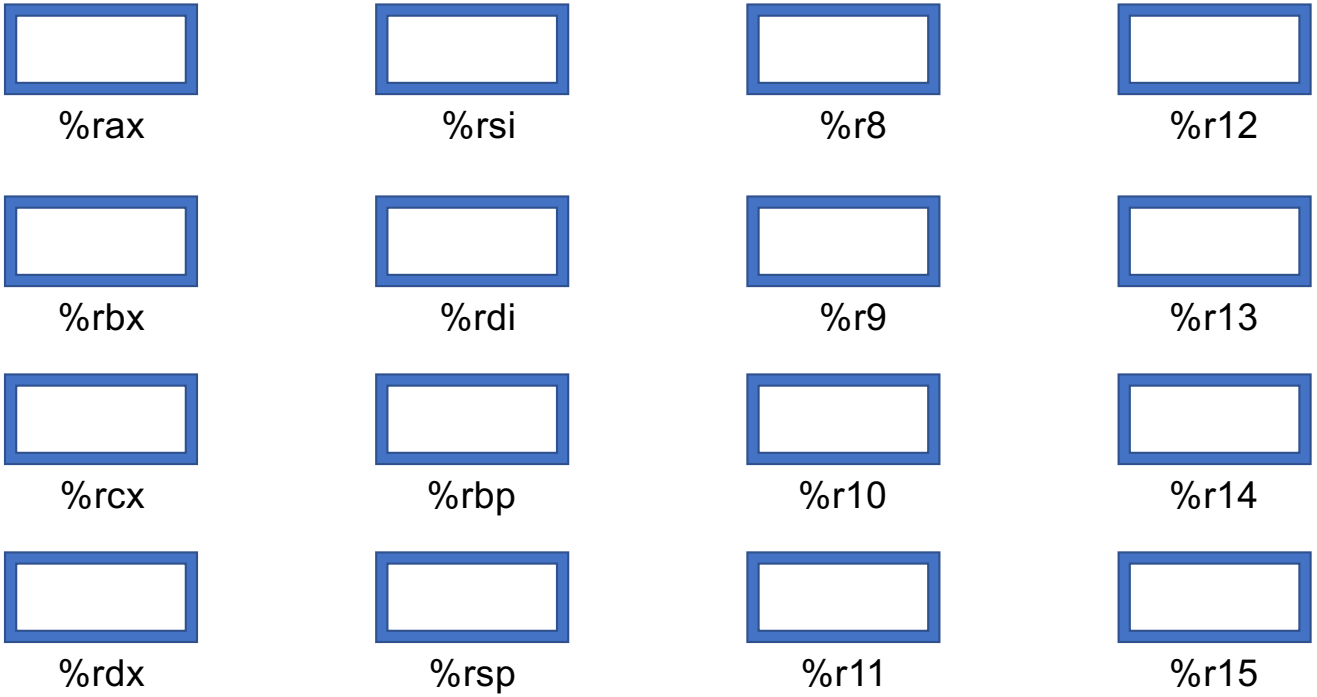
Assembly is **processor-specific** and **very close to the hardware**.

- Each assembly instruction maps closely to a particular machine instruction and exposes registers, memory addresses, and calling conventions explicitly. Because every processor has its own instruction set, assembly programs generally only run on one architecture.

Registers

A **register** is a fast **read/write memory slot right on the CPU** that can hold variable values.
Registers are **not** located in memory.

Registers



Registers

- A **register** is a 64-bit slot on the processor.
- There are **16 such registers**, each with its own name.
- Registers operate like **scratch paper**. Data items being accessed by your programs are generally moved into registers.
  - Most ALU operations—that is, **arithmetic logic unit operations**—require operands be stored in registers first.
- Registers are used to **pass values to functions** and **return values** from them.
- Once a value is loaded into a register, it can be accessed by the ALU **very, very quickly**—typically in a single clock cycle.
- Processor instructions typically **load values** into registers, **perform any necessary operations** on them there, and **store results back** to memory.
  - No matter how clever the original C code is, it **ultimately executes in this form**.

gcc and Assembly

gcc compiles your program – it **decides how stack frames should be built up and torn down** and **generates assembly code instructions** to read and update memory reachable from those stack frames.

Here's what the **assembly abstraction** of C code might look like:

C	Assembly Abstraction
`int sum = x + y;`	1) Copy **x** into register 1 2) Copy **y** into register 2 3) Add register 2 to register 1 4) Write register 1 to memory for **`sum`**