CSCI 210: Computer Organization
Lecture 2: Assembly Language

Stephen Checkoway
Oberlin College
Slides from Cynthia Taylor

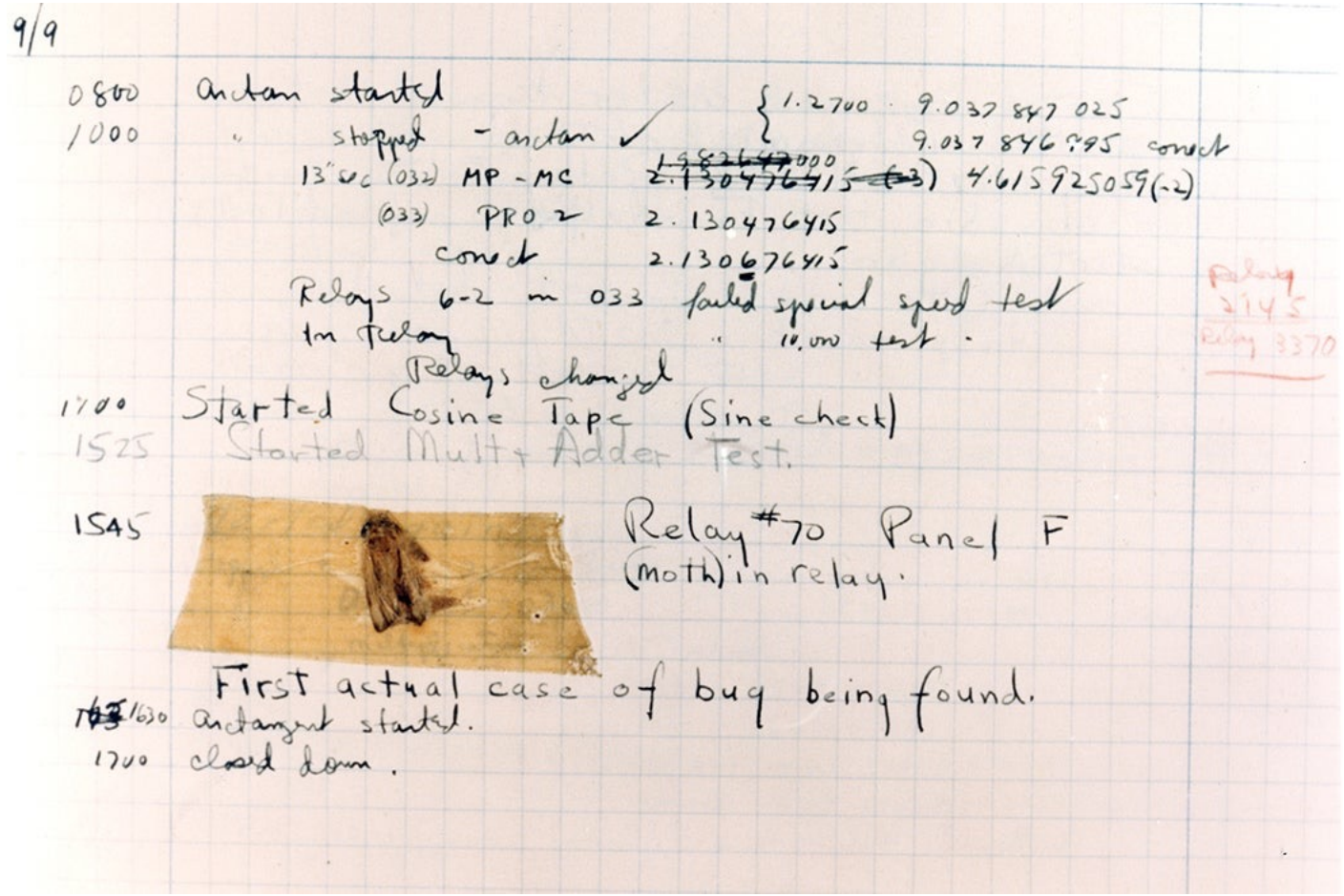
Announcements

- Reading due before class, linked from blackboard
- Problem set 0 due next Friday at 23:59
  - On GradeScope, linked from blackboard

CS History: Rear Admiral Grace Hopper

- Invented the compiler
- Conceptualized machine-independent programming languages.
- Popularized term “debugging”

Not actually the first use of “bug” but a good story nevertheless



https://daily.jstor.org/the-bug-in-the-computer-bug-story/

How to Speak Computer?

```
ld x14,0(x10)
ld x15,8(x10)
sd x14,8(x10)
sd x15,0(x10)
```

1

```
00000000000001010011011100000011
00000000100001010011011110000011
00000000111001010011010000100011
00000000111101010011000000100011
```

2

```
temp = v[0];
v[0] = v[1];
v[1] = temp;
```

3

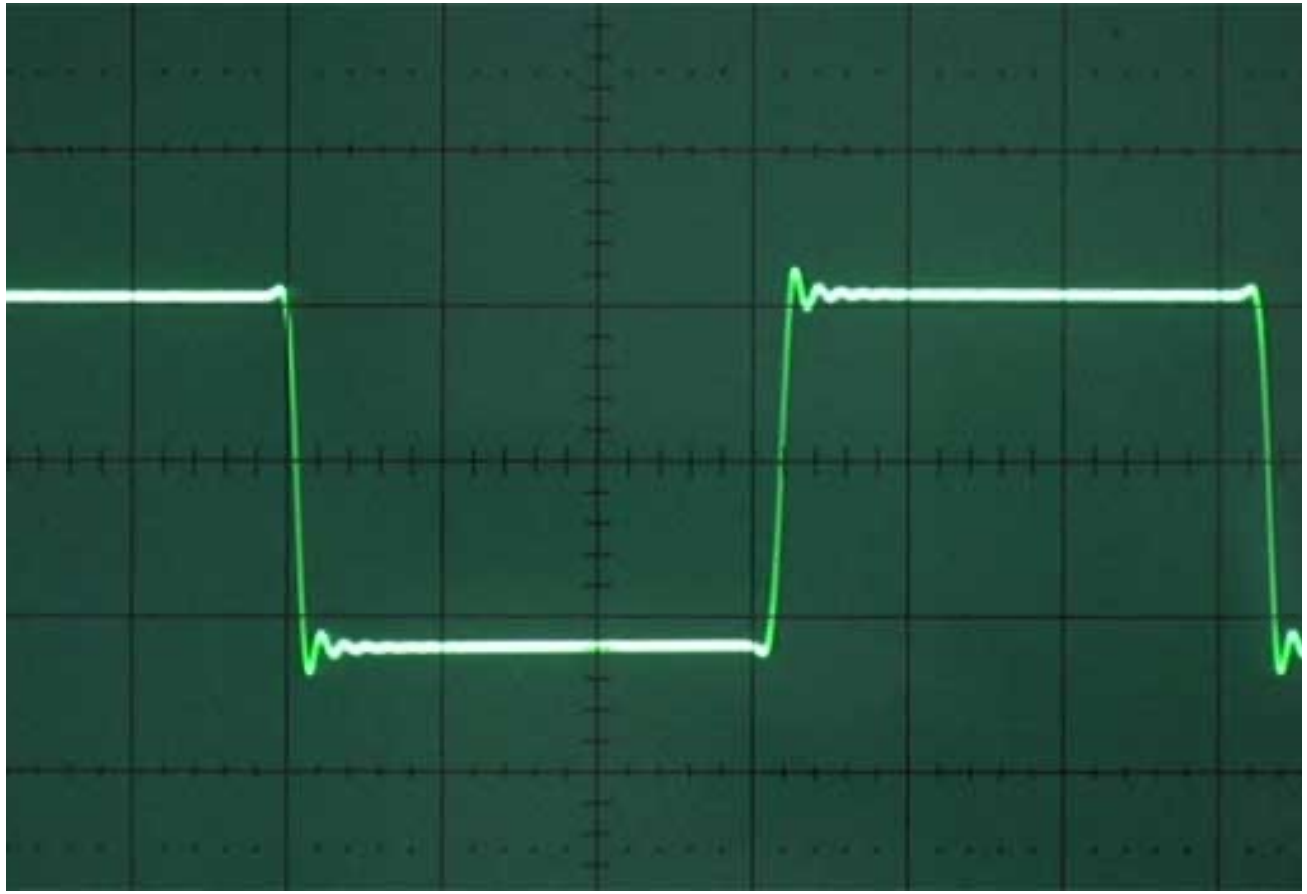
Selection	High Level Language	Assembly	Machine Language
A	3	2	1
B	3	1	2
C	2	1	2
D	1	2	2
E	None of the above		

What Your CPU Understands

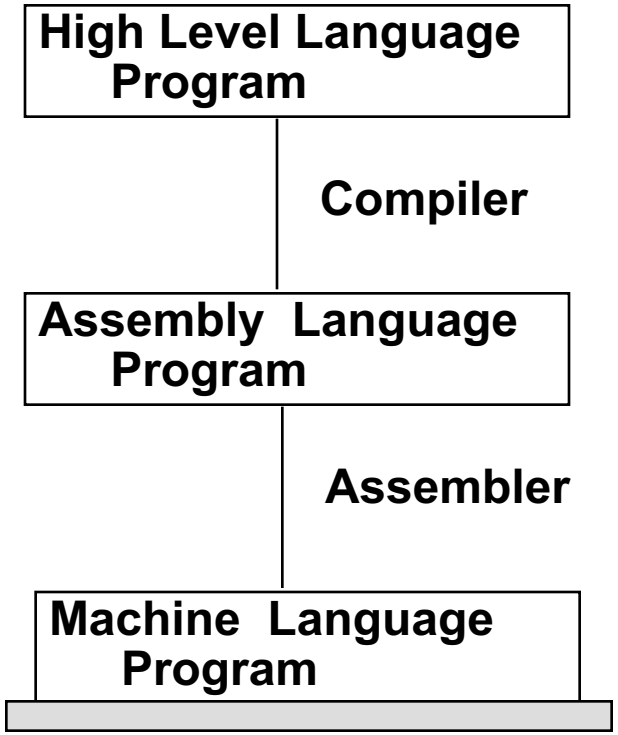
Electricity

Ones and zeros

Problem: People don’t like writing programs in ones and zeros



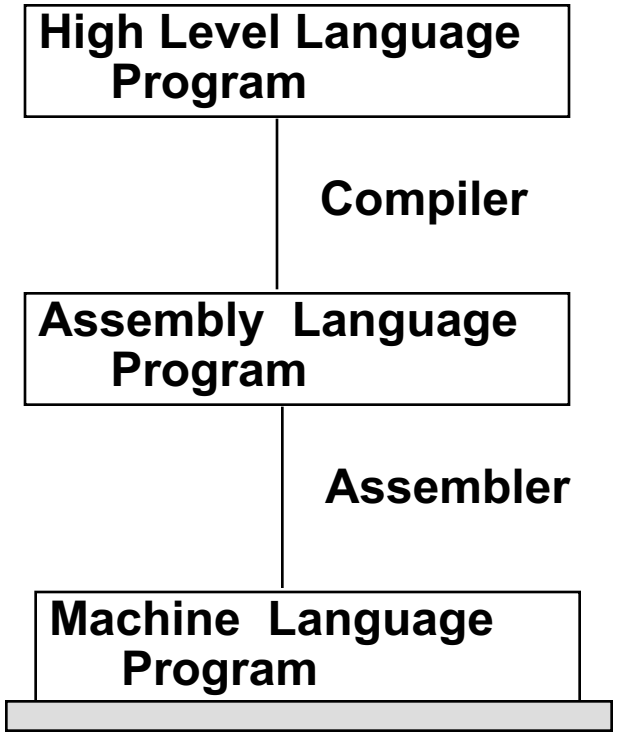
How to Speak Computer




```
temp = v[0];
v[0] = v[1];
v[1] = temp;
```

Machine Interpretation

How to Speak Computer

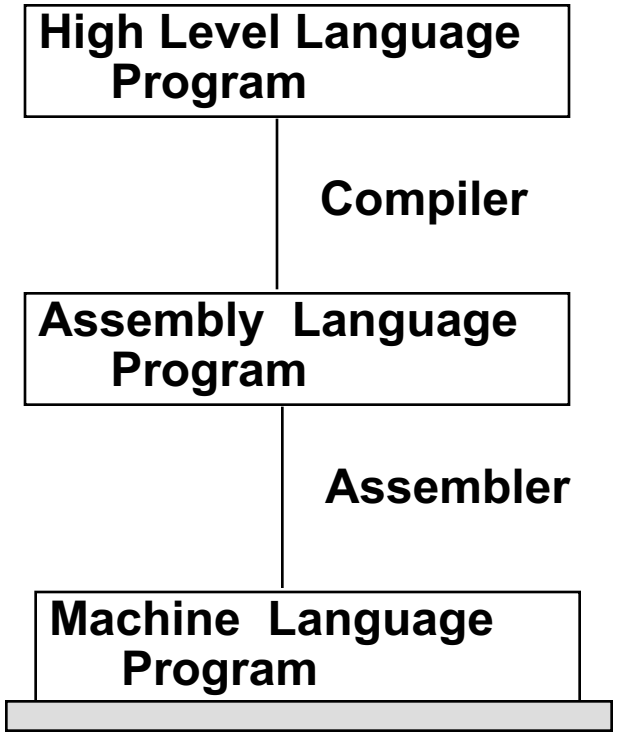



```
temp = v[0];
v[0] = v[1];
v[1] = temp;
```

```
ld x14,0(x10)
ld x15,8(x10)
sd x14,8(x10)
sd x15,0(x10)
```

Machine Interpretation

How to Speak Computer



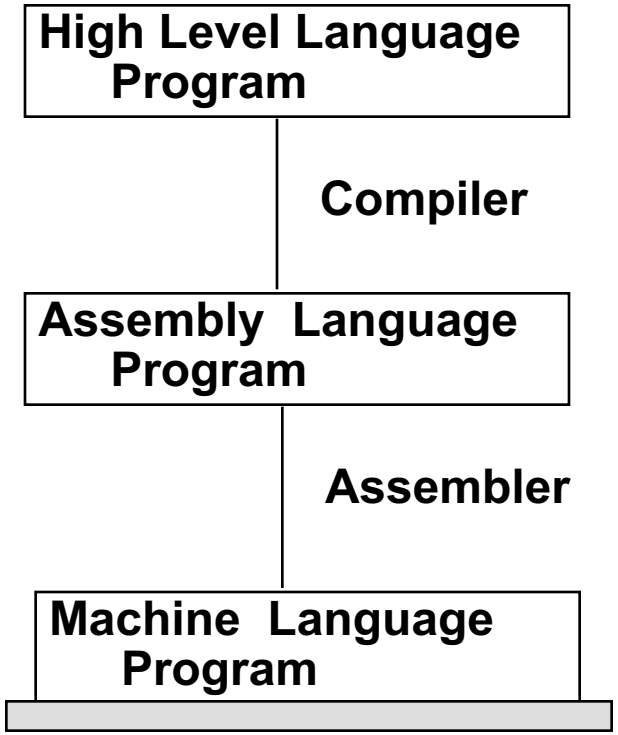

```
temp = v[0];
v[0] = v[1];
v[1] = temp;
```

```
ld x14,0(x10)
ld x15,8(x10)
sd x14,8(x10)
sd x15,0(x10)
```

```
00000000000010100110111100000011
00000000100001010011011110000011
00000000111001010011010000100011
00000000111101010011000000100011
```

Machine Interpretation

How to Speak Computer




```
temp = v[0];
v[0] = v[1];
v[1] = temp;
```

```
ld x14,0(x10)
ld x15,8(x10)
sd x14,8(x10)
sd x15,0(x10)
```

```
00000000000001010011011100000011
00000000100001010011011110000011
00000000111001010011010000100011
00000000111101010011000000100011
```

Machine Interpretation

Machine does something!

Let’s look at these in reverse order

- Starting with “machine does something”
- Build up to high-level languages
- At each step, we’re building **abstractions** that the next higher-level can use

- WARNING: Everything I’m about to say is mostly correct but definitely not complete!

Machine does something

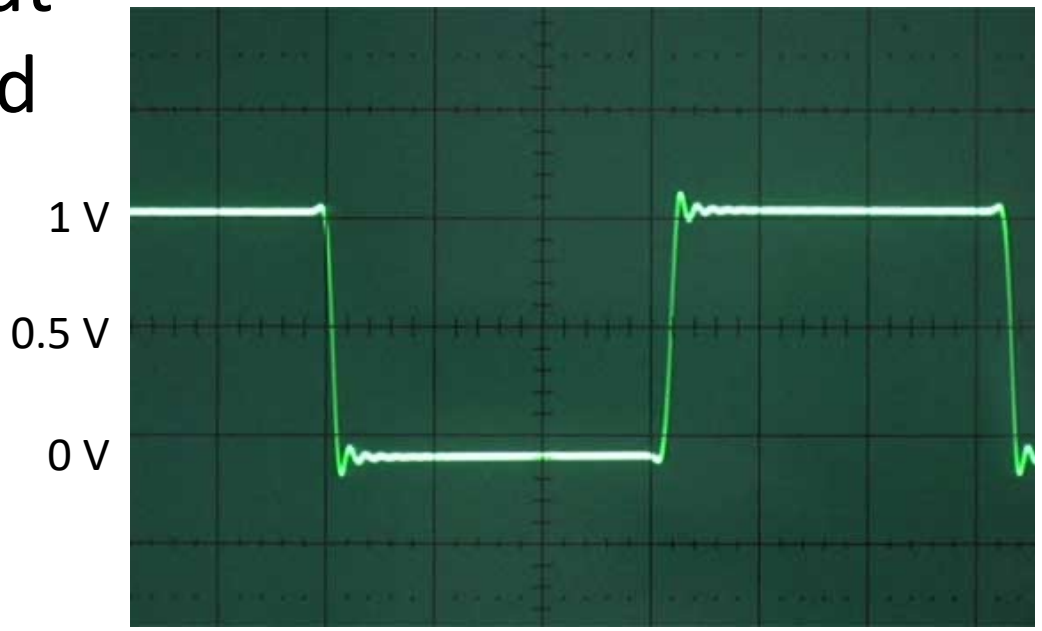
- At the lowest level (that we'll talk about) we have **transistors** which act (sort of) like electrical switches
- These transistors are organized into groups and connected together to perform operations like "add numbers—which are represented by a sequence of electrical voltages—together"
- A modern processor—a CPU—is built from billions of transistors

Central Processing Unit (CPU)

- The CPU operates by processing a stream of machine-language instructions which, **exactly like** numbers, are represented by a sequence of electrical voltages
- The instructions dictate which operation (like addition or multiplication) to perform and what data to perform it on
- The CPU contains a **very** small amount of memory called **registers** (built out of transistors!) to store the data it operates on
  - How small? The CPU has about 30 registers, each holds one number

Correspondence between instructions/numbers and sequences of voltages

- The CPU works with voltages but humans work with numbers and instructions
- We represent numbers/instructions as sequences of 0s and 1s
- These correspond to voltages:
  - 0 corresponds to a voltage < 0.5 V
  - 1 corresponds to a voltage > 0.5 V



Registers

- (Very) Small amount of memory inside the CPU
- Data is put into a register before it is used by an instruction
- Manipulated data is then stored back in main memory (RAM).

Aside: Multi-core CPUs

- Modern CPUs contains one or more “cores,” each of which executes instructions independently from the other cores (we’re going to only focus on single-core CPUs but the same ideas apply to multi-core CPUs)

Machine Language

Machine language program

```
00000000000001010011011100000011
00000000100001010011011110000011
00000000111001010011010000100011
00000000111101010011000000100011
```

- Abstracts from voltage levels to 0s and 1s
- A machine language program tells the CPU what to do
- It consists of a sequence of individual instructions
- Each instruction is a sequence of 0s and 1s
  - In this class, each instruction is a sequence of exactly 32 0s and 1s

Typical Machine Language Operations (with corresponding machine language instruction)

- Load data from main memory (RAM) into a register
  - 00000000000001011011010100000011
- Store the contents of a register into main memory
  - 00000000101001011011000000100011
- Compute the sum (or difference) of two registers, store the result in a register
  - 00000000110001011000010100110011
- Change which instruction runs next (e.g., for a loop)
  - 11111111010111111111000001101111
- Change which instruction runs next based on register values (e.g., for if)
  - 11111110101001011100100011100011

Machine-language is the lowest level abstraction programmer can use to program a particular machine

- We used to toggle physical switches to load machine-language into the computer

- This is painful!

Instruction Set Architecture (ISA)

- The definition (specification) of a machine language supported by a CPU
- Encompasses all the information necessary to write a machine language program, including instructions, registers, memory access, …
- Usually defines a human-readable assembly language which has a 1-1 correspondence with machine-language
  - No more writing code in 0s and 1s!

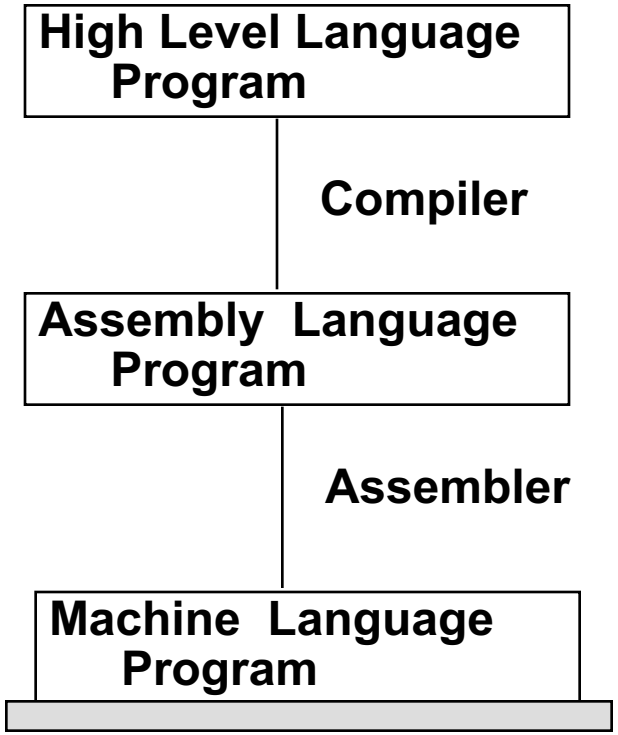
Examples of ISAs

- Intel x86, x86_64 — Most common for desktops, servers, and (non-Apple) laptops
- ARM: A32 (32-bit ARM), A64 (64-bit ARM), T32 (Thumb), Apple Silicon — Phones, tablets, (Apple) laptops
- **RISC-V** — We're going to use RISC-V in this class
- MIPS32, MIPS64 — Previous versions of this class used MIPS32
- Power ISA (PowerPC)

Which of the following statements is generally true about ISAs?

Select	Statement
A	Some models of processors support exactly one ISA, others support multiple (usually related) ISAs
B	An ISA is unique to one model of processor.
C	Every processor supports multiple ISAs.
D	Each processor manufacturer has its own unique ISA.
E	None of the above

How to Speak Computer




```
temp = v[0];
v[0] = v[1];
v[1] = temp;
```

```
ld x14,0(x10)
ld x15,8(x10)
sd x14,8(x10)
sd x15,0(x10)
```

```
00000000000001010011011100000011
00000000100001010011011110000011
00000000111001010011010000100011
00000000111101010011000000100011
```

Machine Interpretation

Machine does something!

Assembly Language

- Abstraction of machine language
  - From 1s & 0s to symbolic names
- Allows direct access to architectural features (registers, memory)
- Symbolic names are used for
  - operations (mnemonics)
  - memory locations (variables, branch labels)
- There's usually a single assembly language corresponding to a machine language
  - x86 has at least 2 distinct assembly languages (Intel and AT&T) with multiple variants of each; x86 is *weird* in a bunch of respects

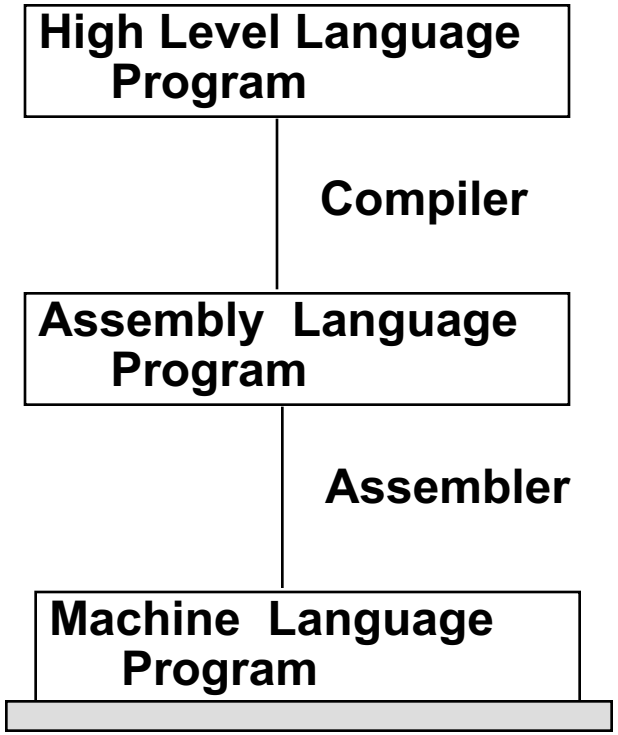
Assembler

- A program called an **assembler** converts assembly-language programs to their equivalent machine-language programs
- The input is a text file containing an assembly program
- The output is a binary file containing a machine language program

Aside

- Sometimes CPUs support undocumented or even unintended instructions
- These instructions often don’t have an official symbolic name and so have to be written in machine language
- Such instructions were common in old CPUs which didn’t check for illegal instructions
- Modern CPUs will (usually) detect an illegal instruction and prevent the program from continuing

How to Speak Computer




```
temp = v[0];
v[0] = v[1];
v[1] = temp;
```

```
ld x14,0(x10)
ld x15,8(x10)
sd x14,8(x10)
sd x15,0(x10)
```

```
00000000000001010011011100000011
00000001000001010011011110000011
00000000111001010011010000100011
00000000111101010011000000100011
```

Machine Interpretation

Machine does something!

High-level code

```
x = 4;
y = 5;
x = x + y;
```

RISC-V code

```
addi t0, zero, 4 #set t0 to 4
addi t1, zero, 5 #set t1 to 5
add  t0, t0, t1  #perform the add
```

Usually, 1 line of high-level code is translated to multiple assembly instructions; these are very simple

Looking ahead: register names

- RISC-V has 32 registers named `x0`, `x1`, …, `x31`
- Registers `x0` always holds the value 0 (it cannot be changed)
- There are **conventions** about how registers get used
- E.g., registers `x5`, `x6`, `x7`, `x28`, `x29`, `x30`, and `x31` are used to hold temporary values
- Each register has a “friendly” name that corresponds to these conventions
- The 7 temporary registers have the friendly names `t0`, `t1`, …, `t6`
- `x0` has the friendly name `zero`
- The book mostly uses the x- names, the slides will use both

Compiler

- A program called a **compiler** translates high-level code like C or Rust into assembly language
- The input is a text file containing a high-level program
- The output is a text file containing an assembly program

- Some compilers (like clang or rustc) incorporate an assembler and go directly from high-level programs to machine language programs; others (like gcc) run the assembler as a separate program

Abstractions recap

- Transistors operating via electricity
- Abstraction: machine-language specifying operations in 0s and 1s
- Abstraction: assembly-language specifying operations in human-readable text
- Abstraction: high-level language specifying algorithms

Group Discussion: What are some advantages to a high-level language over programming in assembly?

A single program written in a high-level language can be compiled into _____ assembly language programs

A. Exactly one

B. Multiple

C. At most three

A single program written in assembly can be assembled into _____ machine language programs

A. Exactly one

B. Multiple

C. At most two

High-level language program (in C)

```
void swap(long v[], long k) {
  long temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

C compiler — one-to-many

Assembly language program (for RISC-V)

```
swap:
  slli  a1, a1, 3
  add   a0, a0, a1
  ld    a4, 0(a0)
  ld    a5, 8(a0)
  sd    a4, 8(a0)
  sd    a5, 0(a0)
  ret
```

assembler — one-to-one

Machine (object, binary) code (for RISC-V)

```
00000000001101011001010110010011
00000000101101010000010100110011
00000000000000101001101111000011
00000000100000101001101111000011
00000000111001010011010000100011
00000000111101010011000000100011
00000000000000001000000001100111
```

Reading

- Next lecture: Hardware!
  - Sections 1.4 and 1.5

- Problem set 0 due next Friday at 11:59 p.m.