Topic 20 Programming Languages – Summary

Vocabulary

instruction set (n): the list of all possible commands a particular CPU knows how to carry out.

machine code (n): the binary codes representing each of the instructions in the instruction set.

source code (n): the text of the program that a programmer writes. This may be assembly code, or code written in a high-level language.

object code (n): the translated source code. This will often be machine code, but might also be an intermediate code (such as Java bytecode) which needs to be further translated before it can be executed.

mnemonic (n): a memory aid or technique designed to help encode, retain, and recall information more effectively. This includes abbreviations, rhymes, diagrams, etc. They leverage the brains natural preference for patterns and connections. In the case of assembly language, each instruction is an easy-to-remember abbreviation of what the instructions does, such as `SUB` for subtract or `SLL` for shift left logical.

abbreviation (n): any shortened form of a word or phrase

assembly language (n): a low-level languages written in mnemonics. Assembly language is architecture specific (for an instruction set, there is an assembly language). Although there are some exceptions, there is a near one-to-one mapping between an assembly language mnemonic, such as `ADD` or `LOAD`, and a machine code instruction for the processor.

acronym (n): a specific type of abbreviation that forms a pronounceable word from the initial letters of a series of words, for example: SCUBA for **s**elf-**c**ontained **u**nderwater **b**reathing **a**paratus, LASER for **l**ight **a**mplification by **s**timulated **e**mission of **r**adiation, or RADAR for **ra**dio **d**etection **a**nd **r**anging. Note that the Pearson textbook calls the assembly language mnemonics acronyms, but most, such as `SUB` or `SLL` are abbreviations but not acronyms.

low-level programming language (n): a programming language that is closely related to the CPU's machine code.

translator (n): a program that converts source code into machine code. *Assemblers*, *compilers* and *interpreters* are all translators.

assembler (n): a translator that converts the mnemonics of assembly language into machine language instructions for the microprocessor to carry out.

interpreter (n): a translator that converts high-level language source code into object code, often machine code and executes the code as it translates.

compiler (n): a translator that converts high-level language source code into object code, often machine code. The source code is translated "all at once" and saved to be executed later.

emulator (n): hardware or software that allows one type of computer system to behave like another

reboot (n): to start a device again (e.g.: turn off the computer and turn it on again).

Concepts

Machine Code

Also called machine language. Understand that the machine code for one architecture (one instruction set) of processor will be very different from the machine code for another architecture. For example, Intel and AMD make processors that implement the x86 instruction set and includes instructions of variable length (some instructions are longer than 32 bits) and complex instructions that perform multi-step operations.

Interpreters

Although the Pearson textbook says interpreted code is "translated and executed one line at a time",(so you may write this on the exams), do know this is an over simplification, and for modern interpreters, the code may be compiled and optimized in entire logical blocks like loops or functions, and perhaps cached for use if the section of code is run again.

Assembly versus Machine Code

It is sometimes said that there is a one-to-one mapping from assembly language instruction to machine code. This is true for many assembly language instructions, however, there are cases where the assembly language instruction represents a task that actually requires more than one machine code instruction to implement. This is to simplify commonly-used machine code patterns. For example, RISC-V assembly includes an instruction to load an immediate 32-bit value into a register:

```
li t0, 0x12345678   # load value into register t0 (x5)
```

It should be obvious that it is not possible to load a 32-bit immediate value into a register using only one RISC-V machine code instruction because the length of each RISC-V instruction is only 32 bits in total!

The li assembly language instruction is actually a pseudo-instruction that corresponds to two separate assembly instructions that do have machine code equivalents:

```
lui t0, 0x12345      # load value into upper bits of register t0
addi t0, t0, 0x678   # add value to register t0 (lower bits)
```

The first instruction uses 20 bits of the instruction to store the immediate value `0x12345`. The contents of register `t0` (which is an alias for register `x0`) after the instruction will be `0x12345000`.

The second instruction adds the value `0x678` to the contents of register `t0`, resulting in the final value of `t0` to be `0x12345678`.

In summary, although there is not an exact one-to-one mapping from assembly language instructions to machine code, the mapping is often so direct and predictable that it is commonly described as one-to-one.