Assembly Languages

What is an Assembly Language?
- The development of assembly language was a major milestone in the evolution of computer technology. First step to the high-level language.

  High-level language program (C, Java, Python) →
  Compiler →
  Binary machine language program (1111100001000000)

  Assembly language program (LOAD 8 RA) →
  Assembler →
  Binary machine language program (1111100001000000)

- Binary programs (machine code) are tedious and very error-prone process.
- Assembly languages provide a user-friendly way to program.

- A sample assembly program: sum the first ten non-zero odd numbers
  # set up
  MOVE 1 RA
  MOVE 0 RB
  MOVE 2 RC
  MOVE 10 RD

  # loop
  loop:
  ADD RA RB RB
  ADD RA RC RA
  ADD -1 RD RD
  BRZ breakout
  BRA loop

  breakout: #RB contains the sum
  OPORT RB
Assembly program:
- use **symbolic name of each instruction**
- use **symbolic address** (label)
- **assembly language is hardware dependent**, with a different assembly language for each type of processor. (reference to specific registers, opcodes supported by the processor, bit length of registers and operand of machine languages)
- Four elements of a statement in a typical assembly language

<table>
<thead>
<tr>
<th>Label</th>
<th>Mnemonic</th>
<th>Operand(s)</th>
<th>;comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>Opcode name</td>
<td>Zero or more</td>
<td>Optional</td>
</tr>
</tbody>
</table>

- The first element is the symbolic address; some lines have no symbolic address, implying that the address of that line is one more than the address of the previous line; If the operand(s) is(are) for memory-referencing instruction, it contains the symbolic address.

Why use label?
- **make a program location easier to find and remember**
- **instructions can easily be moved to correct the program**. The assembler will automatically change the address in all instructions that use the label when the program is reassembled.
- **programmers do not have to calculate relative or absolute memory address**, but just uses labels as needed.

Two-Pass Assembler
- The two-pass assembler parses the input file containing the assembly program twice. Before parse the assembly program, the first step is to tokenize the program.
  - During the tokenization, the whitespace and comments in the program are discarded.
  - The outcome of tokenization is a list of lists. Each sublist contains the elements on a line of code. For example, [['move', '1', 'RA'], ['move', '0', 'RB'], ...].

- First pass is to construct a symbol table that contains a list of all labels and their associated line number (address).
  - The labels are usually used for loop.
  - Examine the assembly program line by line (aka the list of lists after tokenization), determine the length of the corresponding assembly program, and therefore determine the line number for the labels.
• The outcome of the first pass is a symbol table that associate the line number with each label.

- Second pass read the program (the list of lists) again from the beginning. Each instruction is translated into the appropriate binary machine code. The translation includes:
  • Translate the mnemonic (opcode) into binary
  • Use the opcode to determine the format of instruction and length of the various fields in the instruction
  • Translate each operand name into appropriate register or memory code.
  • Translate each immediate value into binary string
  • Translate any references to labels into the appropriate line number using the symbol table generated by the first pass.