How does an assembler work?

In a two-pass assembler

PASS 1: Symbol table generation
PASS 2: Code generation

Illustration of the two passes (follow the class lecture)

```
.data
L1: .word 0x2345  # some arbitrary value
L2: .word 0x3366  # some arbitrary value
Res: .space 4
.text
.globl main
main: lw $t0, L1($0)  # load the first value
     lw $t1, L2($0)  # load the second value
     and $t2, $t0, $t1  # compute the bit-by-bit AND
     or $t3, $t0, $t1  # compute the bit-by-bit OR
     sw $t3, Res($0)  # store result at location in memory
     li $v0, 10  # code for program end
     syscall
```
Other architectures

Not all processors are like MIPS.

Example. Accumulator-based machines

A single register, called the accumulator, stores the operand before the operation, and stores the result after the operation.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load x</td>
<td># into accumulator from memory</td>
</tr>
<tr>
<td>Add y</td>
<td># add y from memory to the acc</td>
</tr>
<tr>
<td>Store z</td>
<td># store acc to memory as z</td>
</tr>
</tbody>
</table>

Can we have an instruction like

```
add z, x, y  # z := x + y, (x, y, z in memory)
```

For some machines, YES, but not in MIPS! What are the advantages and disadvantages of such an instruction?
**Load-store machines**

MIPS is a **load-store architecture**. Only load and store instructions access the memory, all other instructions use registers as operands. What is the motivation?

*Register access is much faster than memory access, so the program will run faster.*

**Reduced Instruction Set Computers (RISC)**

- The instruction set has only a small number of frequently used instructions. This lowers processor cost, without much impact on performance.
- All instructions have the same length.
- Load-store architecture.

Non-RISC machines are called CISC (Complex Instruction Set Computer). Example: Pentium
Another classification

3-address  add r1, r2, r3  \( (r1 \leftarrow r2 + r3) \)
2-address  add r1, r2  \( (r1 \leftarrow r1 + r2) \)
1-address  add r1  (to the accumulator)
0-address or stack machines  (see below)

Example of stack architecture

Consider evaluating \( z = x \times (y + z) \)

Push x

Push y

Push z

Add

Multiply

Pop z
How to represent negative integers? The most widely used convention is 2’s complement representation.

\[ +14 = 0,1110 \]
\[ -14 = 1,0010 \]

Largest integer represented using n-bits is \( + (2^{n-1} - 1) \)

Smallest integer represented using n-bits is \( - 2^{n-1} \)

So, using 4-bits (that includes 1 sign bit),

the largest integer is 0,111 (=7), and
the smallest integer is 1,000 (= -8)

Review binary-to decimal and binary-to-hex conversions.
Review BCD (Binary Coded Decimal) and ASCII codes.
How to represent fractions?
**Overflow**

\[
\begin{align*}
+12 &= 0, 1100 & +12 &= 0, 1100 \\
+2 &= 0, 0010 & +7 &= 0, 0111 \\
\text{add} &= \phantom{0}0 & \phantom{0}0 \\
+14 &= 0, 1110 & \text{?} &= 1, 0011 (\text{WRONG})
\end{align*}
\]

Addition of a positive and a negative number does not lead to overflow. **How to detect overflow?** Here is a clue.

\[
\begin{align*}
0 & \oplus 0 = 0 (OK) & 0 & \oplus 1 = 1 (\text{NOT OK}) \\
+12 &= 0, 1100 & +12 &= 0, 1100 \\
+2 &= 0, 0010 & +7 &= 0, 0111 \\
\text{add} &= \phantom{0}0 & \phantom{0}0 \\
+14 &= 0, 1110 & \text{?} &= 1, 0011 (\text{WRONG})
\end{align*}
\]
The following sequence of MIPS instructions can detect overflow in signed addition of $t1$ and $t2$:

```
addu $t0, $t1, $t2  # add unsigned
xor $t3, $t1, $t2  # check if signs differ
slt $t3, $t3, $zero  # $t3=1 if signs differ
bne $t3, $zero, no_overflow
xor $t3, $t0, $t1  # sum sign = operand sign?
slt $t3, $t3, $zero  # if not, then $t3=1
bne $t3, $zero, overflow
no_overflow:
  ...
  ...
overflow:
  <Do something to handle overflow>
```