Assembly Translation

- **Assembler**
  - a simple translator because of one-to-one correspondence
  - except for preprocessing and function calls
  - more complicated for macro assemblers
  - *Load-and-go* produces executable modules
  - others can produce load, or even objects that could be linked.

- **Module** assemblers often use two-passes (over the source)
  - first pass collects information into *symbol table*
  - it may also collect constants into *literal pool*
  - second pass uses the table to generate target

1 Rudimentary two-pass Assembler

- assume all opcodes and operands are one byte
- assume all arguments are symbolic - translated to addresses

Table 1: Sample instruction set for an accumulator architecture

<table>
<thead>
<tr>
<th>Symbolic</th>
<th>Machine</th>
<th>Length</th>
<th>Num. operands</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>02</td>
<td>2</td>
<td>1</td>
<td><code>Acc &lt;- Acc + arg</code></td>
</tr>
<tr>
<td>COPY</td>
<td>13</td>
<td>3</td>
<td>2</td>
<td><code>arg2 &lt;- arg1</code></td>
</tr>
<tr>
<td>STOP</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td><code>stop</code></td>
</tr>
</tbody>
</table>

- format for the instructions:
  - `[line #] [label] opcode [arg1 [arg2]]`
- assembler directives:
  - `[label] CONST number list`
  - `[label] SPACE`
Translation algorithm overview

**EXAMPLE 1:** Consider example program which uses recursion to compute Fibonacci number

```
1  COPY ZERO OLDER
2  COPY ONE OLD
3  READ LIMIT
4  WRITE OLD
5  CALCNEXT LOAD OLDER
6  ADD OLD
7  STORE NEW
8  SUB LIMIT
9  BRPOS FINAL
10 WRITE NEW
11 COPY OLD OLDER
12 COPY NEW OLD
13 BR CALCNEXT
14 FINAL WRITE LIMIT
15 STOP
16 ZERO CONST 0
17 ONE CONST 1
18 OLDER SPACE
19 OLD SPACE
20 NEW SPACE
21 LIMIT SPACE
```

- replace symbolic addresses by numeric addresses (relative)
- replace symbolic operation codes (not directives) by machine opcodes
- reserve storage for instructions and data
- translate constants to internal representation

Except for load-and-go assembler, only *relative addresses* are needed *(origin)*
Implementation

- Address (location) counter to keep track of current relative address
- Symbol table to keep track of symbols
  - forward references
  - backward references
- Line counter to keep track of what is being processed
- Two passes:
  - build symbol table (plus literal pool)
  - generate target, replacing symbols (plus literals)
- errors in two-passes
  - syntax
  - undefined symbols (end of pass one or only pass two)
  - multiply defined symbols (during pass one)

2 Full Two-Pass Assembler - additional considerations

Additional considerations

- symbolic names can be used for additional purpose as specification of addressing modes
- symbolic addresses can be used in expressions
- storage can be reserved for multiple arguments (as for an array). Storage can be differentiated for byte, word, etc. Allignment must be considered.
- Immediate values
  - immediate mode vs. literal pools
  - protect from changes
- numeric constants can be written as integers, decimals, etc. All must be converted to internal representation specific to the computer.
- address counter may start at 0. Alternatively, it may be reset multiple number of times using ORIGIN. Increasing the counter causes space allocation in the code
- Symbols may be scoped globally, locally in a module, or locally within a module. Global symbols and modular development requires extern-symbol-tables for linker.
● Symbols local within a module is redefinable (multiple definitions). Some rule must be used about usage (e.g., most previous definition). Additional processing is required in pass one, or symbol table format must be modified to allow lists of addresses.

● Address computation can include base registers:
  ○ *Direct addressing*: the address in the argument (loader must modify)
  ○ *Base addressing*: the address is a combination of the argument and a base. *implicit vs. explicit bases*
    address computed on the fly (hardware)

● Addressing mode reflects on address space, size and speed of execution

● Symbol attributes
  ○ address
  ○ relative vs. absolute (not relocated nor modified by base register)
  ○ length
  ○ alignment

● Comments are often allowed, but must be ignored by assembler

● *Listing* is often produced

● Other assembler directives may be used

● Some provisions can be made to ease source writing (macros, etc)

● EQU

○ Implementation of two-pass full assembler
  ● rules for symbol table construction must be modified to allow the additional functions
  ● location counter can be managed on the fly in pass one: ORIGIN would cause reset.
  ● EQU synonyms would cause collapse of multiple rows in symbol table to a single row.
  ● if literals are processed in a literal pool, the pool must be established at the end of code. For long codes, to ensure addressability, it may be dumped partially
  ● pass one can transform the source to *intermediate representation*, with only missing entries for forward references
  ● processing of storage reservation directives calls for increase in address counter and possibly entries in symbol table. Alignment may be needed
  ● for module assembler, a symbol table must be retained for the linker.
3 One-Pass Assembler

- Except for forward references, all could be done in a single pass.
- Forward references can be handled by creating links of references to fill: fill at end or when attributes become known.

4 Load-and-go Assembler

- it must run in a fixed location.
- it cannot be combined with other modules.
- there must be enough memory for the assembler and the code (the code must be in main storage to facilitate filling backward the links)

5 One-pass Module Assembler

- usually, loader/linker will resolve internal forward and external references. For that, tables must be made available to it.