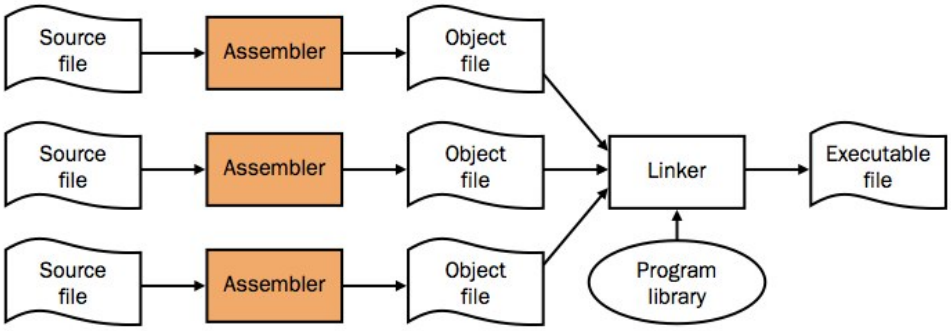

Figure 1: Assembler's process

Assembly Language

- Assembler translates the assembly language *source* into binary instructions in an *object file*.

- Programs usually contain multiple assembly/HLL source files, called **modules**, each compiled and assembled independently. Also, precompiled routines in **library**. So, an object module (.o files for e.g.) contains *external* references to routines in **other modules** and **libraries** and therefore can't be executed since these are **unresolved references**. Linker combines .o files and libraries to produce an **executable**.

- Fig. 2 show machine code for a program, Fig. 3 in raw AL, Fig. 4 in AL with labels for addresses. Names beginning with a dot are **assembler directives**.
  - `.text` says that following lines are instructions
  - `.align n` says *align lines on a* 2^n bytes boundary. `.align 2` means align on a word length (each line is a word long).
  - `.data` indicates that data lines follow
  - `.asciiz` stores a null terminated string in memory, `.globl` declares a procedure available to other object files

- AL serves dual purposes in practice:
  - **output of HLL.** the result of compilation, $\therefore$ the *target* language to the *source* language of HLL
  - **language of programming.** if
    - *speed is critical.* for e.g., in real-time systems. Also embedded systems (the computer is part of a device). Compiler may translate HLL code in indeterminate ways so it is difficult to judge the time cost. AL code however is the (almost) actual code that runs on the CPU.
    - *exploit hardware features* often hidden to HLL, for e.g., low-level memory addressing.

```
00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000001110011100000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100
00000000000000000111100000010010
00000011000011111100100000100001
00010100001000001111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
00001100000100000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
00000011111000000000000000001000
00000000000000000001000000100001
```

Figure 2: Machine code for adding integers between 0 and 100

```
addiu  $29, $29, -32
sw     $31, 20($29)
sw     $4,  32($29)
sw     $5,  36($29)
sw     $0,  24($29)
sw     $0,  28($29)
lw     $14, 28($29)
lw     $24, 24($29)
multu  $14, $14
addiu  $8,  $14, 1
slti   $1,  $8, 101
sw     $8,  28($29)
mflo   $15
addu   $25, $24, $15
bne    $1,  $0, -9
sw     $25, 24($29)
lui    $4,  4096
lw     $5,  24($29)
jal    1048812
addiu  $4,  $4, 1072
lw     $31, 20($29)
addiu  $29, $29, 32
jr     $31
move   $2,  $0
```

Figure 3: Assembly language

```
        .text
        .align  2
        .globl  main
main:
        subu    $sp, $sp, 32
        sw      $ra, 20($sp)
        sd      $a0, 32($sp)
        sw      $0,  24($sp)
        sw      $0,  28($sp)
loop:
        lw      $t6, 28($sp)
        mul     $t7, $t6, $t6
        lw      $t8, 24($sp)
        addu    $t9, $t8, $t7
        sw      $t9, 24($sp)
        addu    $t0, $t6, 1
        sw      $t0, 28($sp)
        ble     $t0, 100, loop
        la      $a0, str
        lw      $a1, 24($sp)
        jal     printf
        move    $v0, $0
        lw      $ra, 20($sp)
        addu    $sp, $sp, 32
        jr      $ra


        .data
        .align  0
str:
        .asciiz "The sum from 0 .. 100 is %d\n"
```

Figure 4: Assembly language with labels

- Another approach: code in both HLL and AL. Use **program profiling** (automatic or manual) to find time-critical parts of program (most time spent). Can improve so much by *using better data structures or algorithms.* Best: *do these parts in AL.*
- Compilers are usually better at producing *uniform high quality code.* Programmers can however consider writing (AL) code in many different ways thus coming up with a best technique.
  - Compilers are becoming smarter though.
- Also AL is the only lang. available on some legacy or embedded systems
- Problems with using AL to code a program:
  - Tightly bound to the architecture for which it was generated. May become obsolete. HLL code can be compiled for any architecture that has a compiler.
  - Longer. ∴, less productive; difficult to understand (try finding out what the loop does in sample code) since no structure, conditionals or loops must be built from scratch (using basically *goto*s); more bugs.
  - difficult to verify the correctness of program. Ada was developed as a HLL for embedded devices (similarly J2ME).

Assemblers

- assembly code → object file w/ binary instructions and data. Two steps:
  - find labels with memory locations (keeps this bookkeeping info (*relocation information*) so local references can be resolved)
  - translate each AL instruction to (binary) numeric instruction
- *object files do not have external references resolved* (assembler only resolves in-file references) so can't be run directly. (although it IS in binary format)
- **Linker** combines multiple objects by resolving external references in between them. Assembler helps by keeping a **symbol table**, a list of undefined external references.
- Assembler's 1st pass: build up symbol table.
  - break instruction into **lexemes**: `ble $t0, 100, loop` contains 6.
  - If line begins with label (loop labels, procedure id labels), records label and the memory word address for instructn in **symbol table**.
  - records instruction's and data's size
- 2nd pass: examines each line again; actually builds m/c code using symbol table
  - Instructions are translated to binary (opcode + operand), including those that use local labels defined in symbol table.
  - If label is for external reference, leaves it be, since symbol table doesn't have it.

- Assembler assumes memory starts from 0.
- Some instructions (`jal, lwl/h/b, sw/h/b`) require absolute addresses of operand data or instruction, and assembler doesn't know actual addresses upon loading. A file's data and instructions are stored contigously in memory but starting address is not known beforehand to assembler.
  - Assembler $\therefore$ builds **relocation information**. List of instructions in .o file that need abs. addresses. These are relocated by the linker.

Facilities

- Assemblers provide some extra facilitites. E.g.: directives: which are instructions meant for assembler (not part of AL)
- e.g. **data layout directives**:

```
.asciiz \The sum from 0 .. 100 is %d\n"
```

- **Macros.** pattern matching and code replacement
  - Like methods/functions. However, only *replacement*, not actual *method call*.
  - For e.g., a print program that prints, in this case, the value in register $7 ($t9)

```
.data
int_str:.asciiz "%d"
.text
la $a0, int_str # Load string address
# into first arg
mov $a1, $7 # Load value into
# second arg
jal printf  # Call the printf routine
```

Can be implemented using a *macro*

```
.data
int_str:.asciiz "%d"
.text
.macro print_int($arg)
la $a0, int_str # Load string address into
# first arg
mov $a1, $arg  # Load macro's parameter
# ($arg) into second arg
jal printf  # Call the printf routine
.end_macro

print_int($7)
```