Translating an Assembly Language Program (Review)

- The *assembler* performs initial translation of an assembly language module into machine language
  - A *module* is a file that contains all or part of a program
  - The *source* module is translated into an *object* module
- The *linker* links a set of assembled modules and libraries together to form a complete program (an executable file)
  - Resolves *external references* — symbols defined in one module and used in another
- The *loader* loads the completed program into memory where it can be executed
  - Usually capable of loading the program at an arbitrary memory location (*relocation*)

A Simple One-Pass Assembler

```
void main (void)
{
    /* construct an empty symbol table */
    make_empty_table (sym_tab);

    /* initialize the location counter */
    location = LOAD_ADDR;

    /* process each line in the source file */
    while (!eof(source_file)) {
        read_line (sourcefile, this_line);

        /* check for a new label definition */
        label = new_label (this_line);
        if (label != NULL)
            enter (sym_tab, label, location);

        /* translate the instruction on this line */
        mach_inst = translate (this_line, location);
        if (mach_inst != NULL) {
            write (object_file, mach_inst);
            location = location + size_of(mach_inst);
        }
    }
}
```


Working with Pointers (Preview)

The C code:

```
ptra = &a[0];
ptrb = &b[0];
for (i=1 ; i<=10 ; i++)       /* use register */
{                             /* indirect */
  *ptrb = *ptra;              /* addressing */
  ptra++; ptrb++
 }
```

The assembler code:

```
            LOAD      R2,#a  ; R2 = ptra
            LOAD      R3,#b  ; R3 = ptrb

            LOAD      R1,#1  ; R1 = i
test2:      BRLE      R1,#10,for2
            JUMP      endfor2
for2:       STORE     @R3,@R2
            ADD       R2,R2,#4
            ADD       R3,R3,#4
            ADD       R1,R1,#1
            JUMP      test2
endfor2:
```


A Two-Pass Assembler

```
void main (void)
{
    /*********** the first pass ***********/
    make_empty_table (sym_tab);
    location = LOAD_ADDR;
    while (!eof(source_file)) {
        this_line = read_line (source_file);
        label = new_label (this_line);
        if (label != NULL)
            enter (sym_tab, label, location);
        location = location + bytes_needed(this_line);
    }

    /*********** the second pass ***********/
    rewind_file (source_file);
    location = LOAD_ADDR;
    while (!eof(source_file)) {
        this_line = read_line (sourcefile);
        mach_inst = translate (this_line, location);
        if (mach_inst != NULL) {
            write (object_file, mach_inst);
            location = location + size_of(mach_inst);
        }
    }
}
```

A Two-Pass Assembler With a Patch List

```
void main (void)
{
    /*********** the first pass ***********/
    make_empty_table (sym_tab);
    location = LOAD_ADDR;
    while (!eof(source_file))  {
        this_line = read_line (source_file);
        label = new_label (this_line);
        if (label != NULL)
            enter (sym_tab, label, location);
        mach_inst = translate (this_line, location);
        if (mach_inst != NULL)  {
            if (incomplete (mach_inst))  {
                patch_item = make_patch
                        (mach_inst, location);
                add_to_end (patch_list, patch_item);
            }
            write (object_file, mach_inst);
            location = location + size_of(mach_inst);
        }
    }
    /*********** the second pass ***********/
    while (!is_empty(patch_list))  {
        patch_item = remove_first (patch_list);
        apply_patch (object_file, patch_item);
    }
}
```


Notes on Two-Pass Assembler With Patch List

- In the first pass, the assembler
  - Translates each instruction into machine language and puts it into the object file, even if it has to leave “holes” where it should put an address
  - Enters labels into symbol table, with an indication if they’re undefined
  - Builds a patch list
    - Instructions that need to be patched
- After it finishes that pass, then it knows all the labels and addresses (the symbol table is complete), so…
- In the second pass, the assembler goes over (only) the object file
  - “patching” the “holes” in the instructions that use forward references with the actual addresses

Assembler With Segments

- Maintains separate base address and location counter for each segment
- Initially writes translated text and data to separate object files
  - Does not write .bss segment to object file (no need to store uninitialized space!)
- Then writes final object file:
  - Header (size of each segment, address of first instruction to be executed, etc.)
  - Text segment, data segment
  - Symbol table, patch list(s)
- Two alternatives to determining length of text and data segments:
  - Three-pass assembler
  - Use offsets instead of addresses in symbol table

Linking

- The *linker* links a set of assembled modules and libraries together to form a complete program (an executable file)
  - Resolves *external references* — symbols defined in one module and used in another
- Assembler and linker can work together:
  - Assembler makes a single pass:
    - Translates each instruction (w/ holes)
    - Builds symbol table (incl. undefined labels)
    - Builds two patch lists (text, data)
    - All symbol references cause a patch entry
  - Linker makes three passes
    - Pass 1 — Combine text, data, and bss segments into a single executable file
    - Pass 2 — Build private symbol table for unexported symbols in each file, public symbol table for exported symbols
    - Pass 3 — Apply all patches to executable file

Loading

- The *loader* loads the completed program into memory where it can be executed
  - Loads text and data segments into memory at specified location
  - Returns value of start address to operating system (from header — address of first instruction to be executed)
- Alternatives in loading
  - Absolute loader — loads executable file at fixed location
  - Relocatable loader — capable of loading the program at an arbitrary memory location
    - Assembler and linker assume program will start at location 0
    - When program is loader, loader modifies all addresses by adding the real start location to those addresses

Homework #4 — Due 10/26/98 (Part 2)

2. Write a complete assembly language program in the book's LOAD / STORE architecture that reserves space for 5 integers, counts the number of those integers that are odd (we'll assume that someone else somehow loads values into those memory locations), and stores the result in a memory space named "count". Your program should also:
   - Use a loop to examine the 5 integers
   - Use bit minipulation instructions to determine if each integer is odd or even
   - Contain all necessary segments (text, data, bss, etc.)
   - Be written in good programming style, including comments, etc.

   This program counts 3/5 of this homework grade.