Assemblers & Linkers

Real Programs

- Real programs are broken up into modules (various .c files, for instance).
- Each file might declare global variables.
- Each file might use globals that are (possibly) defined by another file.
- Each file might call functions that are defined in some other file.
- How do the tools we've seen manage these details?

Assembly File Format

- A real assembly file looks like this:

```
.data
   # global data definitions go here.
.text
   # instructions go here
```

- Example:

```
.data
   someArray:       .space   400      # an array 400 bytes long
   x:               .word    13       # a word-sized variable
.text
   lw       $t0, x($gp)
   addi     $t1, $gp, someArray
   lw       $t2, 0($t1)
```

  - someArray and x are *labels*

Labels

- Assemblers let us use *labels* to talk about offsets or addresses in our programs.
- Labels are resolved into actual values by the linker.
- So what does the assembler actually do?
  - Encodes instructions as best it can.
  - Spits out an object file.

Assemblers

- Assemblers let us express global (static) data and instructions.
- They let us talk about locations in terms of labels, rather than numeric values.
- Many assemblers (not the one we'll use in this class) include other bells and whistles you'd want for building large-scale systems:
  - Pseudo-operations (eg. for multiplication)
  - Other addressing modes
  - Flexible syntax

Object File Format

- An object file contains (at least) the following:
  - A text segment (some instructions)
  - A data segment (global data defined by this file)
  - A symbol table: a list of symbols defined and referenced by this file

Linker

- The linker takes a bunch of object files and resolves inter-file and intra-file symbol references.
- It spits out an executable file, which contains (at least):
  - A text segment
  - A data segment
  - Debugging information (possibly)
- After resolving symbol references in the modules, the Linker streams together the text segments of all of the object files followed by the data segments.

Loader

- How do we run an executable?
- The loader is a component of the operating system that knows how to read executable files:
  - It reads the executable from disk.
  - Places the text segment into memory.
  - Places the data segment into memory.
  - Sets up the stack pointer, frame pointer, and global pointer.
  - Kicks off the program by jumping to the first instruction.

Back to Our Example:

- Remember our simple C program:

```
int x, y;

void main( ) {
   x = x + y;
   if (x == y) {
     x = x + 3;
   }
   x = 42 + x * y;
   ...
}
```


Real Assembly Version

```
.data
x:          .word   0
y:          .word   0
.text
            lw      $t1, x($gp)      # t1 holds x
            lw      $t2, y($gp)      # t2 holds y
            add     $t1, $t1, $t2    # x = x + y
            bne     $t1, $t2, L1     # branch if t1 != t2
            addi    $t1, $t1, 3      # x = x + 3
L1:         mult    $t1, $t2         # lo = x * y
            mflo    $t3              # get the result
            add     $t1, $t3, 42
            sw      $t1, x($gp)      # update x
```

- Notice: we use labels for our globals and for the branch instruction. Why is this a good idea?
- Link the program for me (resolve the labels to values).

A More Complicated Example

- A looping C program:

```
int array[100];

void main( ) {
   int i;
   i = 0;
   while (i < 100) {
       array[i] = i;
       i = i + 1;
   }
}
```


Assembly Version

```
.data
array:      .space  400
.text
main:       add     $t0, $0, $0      # use t0 as a counter (i)
            addi    $t1, $gp, array  # t1 holds an address
            addi    $t2, $0, 100     # t2 holds constant 100
start:      slt     $t3, $t0, $t2
            beq     $t3, $0, done
            sw      $t0, 0($t1)      # a[i] = i
            addi    $t0, $t0, 1      # i = i + 1
            addi    $t1, $t1, 4      # why are we adding 4?
            j       start
done:       jr      $ra              # return to caller
```

- Notice the use of labels (for data and for branch offsets)
- Link the program for me (resolve the labels).

Assembly Version (Compiler Generated)

```
.data
array:      .space 400
.text
main:       addi    $sp, $sp, -24
            sw      $31, 16($sp)
            addi    $2, $0, 99
            addi    $3,$gp, array
            addi    $4, $3, 396
L5:         sw      $2, 0($3)
            addi    $2, $2, -1
            addi    $3, $3, -4
            bgez    $2, L5
            lw      $ra, 16($sp)
            addi    $sp, $sp, 24
            jr      $ra
```

- The compiler is pretty smart...