1 Overview

Your assignment is to modify a MIPS assembler to relocate (patch) the addresses of symbols (labels). The source for a simple assembler can be found in /home/cs340/prog6. You must add the code that creates the list of instructions to patch during the first pass of the linker, and then patch those instructions during the second pass. You shouldn’t have to look at the rest of the code or the assembler; this handout should contain all the information you need to interface your patching code with the assembler properly.

2 asm340

asm340 is a simple MIPS assembler. The command-line syntax is

    asm340 foo.s foo.o.

The original version of the assembler does not patch instructions, so that the object file contains ‘0’ for all addresses and offsets.

3 asm340.h

The file asm340.h in /home/cs340/prog6 contains the definitions needed to patch the instructions. These include the following:

```
enum segment { TEXT = 0, DATA = 1, BSS = 2, NONE = -1 };
typedef struct symbol {
    char* label;                     // symbol (label) name
    enum segment segment_loc;        // segment to which it belongs
    unsigned int offset;             // byte offset within segment
    unsigned int symbol_index;       // unique index of symbol
    unsigned int defined: 1;         // 1 if defined, 0 otherwise
    unsigned int imported: 1;        // 1 if imported, 0 otherwise
    unsigned int exported: 1;        // 1 if exported, 0 otherwise
} symbol;
#define MAX_PATCH 1024               // max. patch list entries

void add_patch_entry(enum segment segment, unsigned int offset, symbol *symbolPtr);
```
void apply_patches(void);
unsigned int io_read_word(unsigned int address);
void io_write_word(unsigned int address, unsigned int word);
unsigned int get_segment_base(enum segment segment);

The definitions are used for more than code patching, so you won’t need everything in the file. In particular, you won’t use the BSS and NONE segments, nor the imported and exported fields in the symbol structure.

4 Patch routines

You have to write the following routines:

    void add_patch_entry(enum segment segment, unsigned int offset, symbol *symbolPtr)
    As asm340 processes the source file it calls this routine each time it encounters an instruction or data directive that uses a label. The segment parameter is either TEXT or DATA, and indicates which segment contains the word/instruction to be patched. offset is the byte offset of the word to be patched from the beginning of the segment. symbolPtr is a pointer to the symbol table information for the symbol (defined in asm340.h). This routine should remember this information for use during the patch phase. Note that you cannot call io_read_word and io_write_word (see below) from this routine; final addresses are not computed until the source file is fully processed. Similarly, the information in the symbol structure may not be valid as the symbol may not yet be defined. Simply save the pointer to the symbol structure for use when applying the patches. For simplicity you may assume that at most MAX_PATCH instructions/words will need to be patched (defined in asm340.h). This allows you to implement the patch list with a global array. If you want to practice your C programming you could implement your patch list as a linked-list and malloc each node of it.

    void apply_patches(void)
    asm340 calls this routine after it has read the source file and partially translated the instructions. This routine uses the information stored in the patch list to patch the appropriate words in the output file. You should use the routines io_read_word and io_write_word to read and write words in the output file, respectively (see below for descriptions). All defined symbols will have valid offsets in their symbol structures; print an error message if an instruction to be patched uses an undefined symbol (defined is 0). The address of the symbol is its offset within the segment plus the base address of the segment (use get_segment_base to get the base). Instructions are patched differently depending on their format — see the MIPS instruction encodings.

    Words in the data segment are patched by writing the symbol’s address into the output file at the location to be patched.

    Don’t forget that branch instructions are relative; they are patched with the difference (in instructions) between the symbol’s address and the instruction’s address.

    Print an error message if the address is too large to be patched into the instruction. For example, many instructions contain the low-order 16 bits of the address. If the address won’t fit in the instruction then print an error and indicate the address of the offending instruction, what label it uses, and the address of the label.
5 I/O routines

Use the following routines to read and write the memory locations within the object file being patched. Only call these when applying patches; final addresses are not known while processing the source file. Note that the address parameters refer to addresses in object file, not the assembler itself. That’s why they are of type unsigned int and you must use special routines to read and write the object file.

unsigned int io_read_word(unsigned int address)
    Reads the word at address in the object file and returns it as an unsigned int.

void io_write_word(unsigned int address, unsigned int word)
    Writes the word to the object file at address.

unsigned int get_segment_base(enum segment segment)
    Returns the base address of the segment in object file. This routine can only be called after the pass through the source file.

6 An Example

Suppose the assembly file contains the instruction:

```
jal   foo
```

and the label foo is defined somewhere else in the file. When asm340 reaches this line it will call the add_patch_entry function you’ve written with segment set to TEXT, offset set to number of bytes from the beginning of the text segment to this instruction, and symbolPtr pointing to the symbol structure for foo. asm340 will translate the instruction into MIPS machine code using ‘0’ as foo’s address. The resulting instruction is 0xC0000000 (see the MIPS encoding sheet). At some point while processing the source file asm340 will find where foo is defined, and fill in its symbol structure with its segment and offset.

After processing the source file asm340 will call your apply_patches routine. From your patch list you will realize that you need to patch the above instruction. Suppose that foo is 4 instructions beyond the instruction to be patched (determined by looking at foo’s address in the symbol structure and the address of the instruction to be patched). You must then patch the instruction so that it is 0xC000004. Use io_read_word to read the instruction to be patched, and io_write_word to write it back out after you’ve patched it.

7 Displaying object file contents

You can display the values of the words in the object file produced by asm340 using the od Unix command. od reads input files in byte, half-word, or word-sized chunks, and can print the hexadecimal, binary, or octal representations of that chunk.

For example, assume that you have the file hello.txt containing the text

```
Hello
```
You can use `od` to display the contents of this file grouped into four-byte words, as such:

```plaintext
od -Ax -X hello.txt
```

and the program will display:

```
0000000 48656c6c 6f0a0000
0000006
```

which is the hexadecimal representation of the first six bytes in the file `hello.txt`, grouped into 4-byte (word) chunks. The first set of digits on the line represents the starting offset in the file where the data on that line came from (in hex), and each subsequent set of eight hex digits represents the value of a 4-byte word in the file. The offset in the file corresponds to the address of the data once they are loaded into memory, so in this example the value 0x48656c6c will be stored at address 0.

8 Testing and Turnin

Put all your code in the file `patch.c`. A template is included to get you started. To simplify testing your program the `asm340` sources include the file `test1.c`, which isn’t really part of the assembler. Instead, it is a simple test program. Typing

```plaintext
make test1
```

will make a program called `test1` that you can run to test your patching code. The proper output can be found in `test1.out`. Please make sure that your patching code works on this test case, plus others of your own, before testing the full `asm340` program. It will be much easier to debug the test program than the full `asm340`.

To build `asm340` type `make asm340`. Several sample input files called `input*.s` are provided to test your program; the correct outputs are found in `input*.out`. Only turn in the files you created or modified (probably only `patch.c`) using `ca340prog6`. Typing `make turnin` will do it for you.