Compiled on 08/05/2023 at 11:16am

Loading and Linking

1 Loaders

The operating system controls the programs that run on a processor. To run a program, the operating system must load the program from disk storage into main memory (RAM) and prepare it for execution. The operating system typically uses another program, called a *loader*, to achieve this. The following shows pseudo-code of a simple operating system and loader:

```
operating system ver. 1.0
repeat:
  p <- next program to run
  loader(p)
  jalr $0
  beq $0, $0, repeat
```

```
loader ver. 1.0
for (i=0; i<codeLength; ++i) {
   MEM[4i] = file[i];
}
```

In the pseudo-code above, once the program to run has been chosen, the loader loads it by copying the code from the `file` to `MEM`. Notice that the loader above loads the selected program starting at address `0x00`. We use `codeLength` as a count of number of words in a program. `file[i]` is pseudo-code to read the $\texttt{i}^{\texttt{th}}$ word. The $\texttt{i}^{\texttt{th}}$ word is loaded into the 4 bytes starting at `4i`. Once the program has been loaded, the operating system jumps to address `0x00`. Once the program returns, the operating system looks for the next program to run.

The assumption made in the earlier module on MIPS assembly language and the code above is that a MIPS program is always loaded starting at address `0x00`. This is of course a simplistic and unrealistic view of how things really are. For one thing, the operating system itself is a program that is running in memory and there might be other programs also running at the same time. It is not possible for a program to always be loaded at the start of RAM memory. Instead, we can have the loader search for an appropriate amount of RAM memory, load the program at that memory and return to the operating system the starting address of the memory assigned to the program. We show an updated version of the pseudo-code for the operating system below:

```
operating system ver. 2.0
repeat:
    p  <-- choose program to run
    $3 <-- loader(p)
    jalr $3
    beq $0, $0, repeat
```

Notice in the pseudo-code above that the loader returns a value. This is the starting address of the program that was just loaded. The operating system then jumps to that address. The pseudo-code for the loader is shown below, where the variable `N` represents the amount of memory the loader decides to allocate to the program. In our memory model for a program, the memory assigned to the program includes memory needed for the instructions in the program, and memory for the stack and heap. In other memory models, you would have more sections (such as memory for read-only data and global values). Now let's update the loader as well:

```
loader ver. 2.0
α <-- findFreeRAM(N)
for (i=0; i<codeLength; ++i) {
   mem[α+4i] = file[i];
}
$30 <-- α + N
return α to OS
```

There are two differences between loader version 1 and 2. The big change is that in version 2 the loader searches for a block of memory that is big enough (`N` contiguous bytes of memory) to use for the program. In the code above, the starting address of this block is denoted by α. The loader then loads the program starting at α. The other difference is that the loader sets \$30 to just past the last word assigned to the program. Recall that this was as assumption we had made in earlier modules, i.e., the loader initializes \$30 to the top of the stack.

The loader's algorithm above would indeed load an assembled MIPS program into the chosen memory location. The problem is that there is a high chance that the program will not execute correctly.

What would cause an assembled MIPS program when loaded at starting address α by the loader fail to execute correctly?

The problem is that the MIPS assembler assumed that the program would be loaded at starting address `0x00`. This assumption is no longer true. But what features of MIPS assembly are affected? There is just one: labels. The following example illustrates the problem.

```
Assembly       Address   Instruction
lis $3            0x00   0x00001814
.word 0xabc       0x04   0x00000abc
lis $1            0x08   0x00000814
.word A           0x0c   0x00000018
jr $1             0x10   0x00200008
B:
jr $31            0x14   0x03e00008
A:
beq $0, $0, B     0x18   0x1000fffe
.word B           0x1c   0x00000014
```

An assembly program and its assembled equivalent are shown above. What the program does is immaterial. Labels can be used in two places: in a branch instruction (`beq $0, $0, B` above) and as part of the `.word` directive (`.word A` and `.word B`). Recall from our discussion of uses of labels in branches, we had an equation which computed the offset that is needed by the MIPS machine code. Since the equation computes the relative difference between the use and definition of a label, the actual memory address of where these labels appear do not matter. Therefore, labels used in branches are not affected if the loader loads the program at an address other than `0x00`.

The other use of labels is as part of `.word` directives. There, recall, the assembler simply replaces the `.word` with the **memory address** of where the label was defined. This memory address was computed by the assembler by assuming that the program was loaded at starting address `0x00`. In the example above, `.word A` was assembled to `0x0000018` because the label `A` was defined at location `0x18` assuming that the first instruction was at address `0x00`. Similarly, `.word B` was assembled to `0x14` because the label `B` was defined at address `0x14` if we were to assume that the

first instruction is at address `0x00`.

Now that the loader is not loading the first address at `0x00`, any word in the assembled file, which used to represent a `.word label` will be off by exactly α, the actual starting address of the loaded program. Since the actual starting address only becomes apparent once the loader finds the chunk of memory it will use for the program, it is the loader which must **relocate** words that used to represent `.word label` in the original assembly program. The actual relocation is straightforward: just add α to the value at that location.

Fine! We have decided that our loader will relocate. However, there is still a problem. The problem is that the content that the loader is reading (from a binary file) looks as follows [1]:

```
0x00001814
0x00000abc
0x00000814
0x00000018
0x00200008
0x03e00008
0x1000fffe
0x00000014
```

The problem is that the assembled file that the loader is going to load is simply a stream of bits, i.e., the loader has no way of finding which words were produced from the directive `.word label`. And if the loader cannot find which words to relocate, it can of course not relocate them. The solution to this is to have the assembler encode information of what locations contained instances of a `word` used with a label. Therefore, instead of generating just pure machine code, the assembler should generate *object code*.

Definition 1 *Object code contains machine code for the assembly program plus additional information needed by the loader and (later) the linker.*

In CS241, we use a homegrown object code format named `MERL`: MIPS Executable Relocatable Linkable. In the next section, we introduce this file format.

2 MERL Format

The assembler generates MERL output for the MIPS assembly program being assembled. Much like other assembled programs, the output is binary. The MERL format contains three sections: a header, the assembled code and a footer. The assembled code section of the output is exactly as before. The header is exactly three words as described below:

1. Word 1 is the binary encoding of the assembly instruction `beq $0, $0, 2`, i.e., an unconditional jump over the next two words.

2. Word 2 is the address just past the end of the MERL file, i.e., it indicates where the MERL footer ends.

3. Word 3 is the address just past the end of the assembled code, i.e., it indicates where the MERL footer begins.

[1] Well, not really. The content is in binary, we are using hexadecimal because it is easier to read.

The E in MERL stands for executable; MERL object files are executable as is. To achieve this, the first instruction in the header jumps over the remaining MERL header and begins executing the actual MIPS program. Of course, the program only works correctly in this way if loaded at address `0x00`. The next two words in the header specify the end and begin addresses of the MERL footer respectively. The MERL footer (the third and last section of the MERL file) contains relocation entries [2].

Each relocation entry in the MERL footer comprises of two words. The first word is the format specifier and has the value 1 to indicate that this is a relocation entry. The second word in a relocation entry is the address of a word that must be relocated, i.e., α, the starting address should be added to the word at the address specified in the second word of a relocation entry.

The table below shows the same assembly program as before. Like before, we also show the non-relocatable assembled code. However, this time we also show the relocatable MERL output and what this MERL output would look like in assembly (which makes it easier to read). An interesting observation to make is that since we added the MERL header before the code segment, the words in the code segment appear 12 bytes (3 words) ahead of where they used to in the non-MERL version of the code. For example, the `.word A` would have appeared as the fourth word (address `0x0c`) but now appears as word 7 (address `0x18`). The relocation entries obviously use the updated locations.

Assembly	Address	ML Not Relocatable	Address	MERL Relocatable	MERL In Assembly
			0x00	0x10000002	beq $0, $0, 2
			0x04	0x0000003c	.word endModule
			0x08	0x0000002c	.word endCode
lis $3	0x00	0x00001814	0x0c	0x00001814	lis $3
.word 0xabc	0x04	0x00000abc	0x10	0x00000abc	.word 0xabc
lis $1	0x08	0x00000814	0x14	0x00000814	lis $1
.word A	0x0c	0x00000018	0x18	0x00000024	reloc1: .word A
jr $1	0x10	0x00200008	0x1c	0x00200008	jr $1
B:					B:
jr $31	0x14	0x03e00008	0x20	0x03e00008	jr $31
A:					A:
beq $0, $0, B	0x18	0x1000fffe	0x24	0x1000fffe	beq $0, $0, B
.word B	0x1c	0x00000014	0x28	0x00000020	reloc2: .word B
					endCode:
			0x2c	0x00000001	.word 1
			0x30	0x00000018	.word reloc1
			0x34	0x00000001	.word 1
			0x38	0x00000028	.word reloc2
			0x3c		endModule:

[2] The full MERL format also contains other entries in the MERL footer which we will discuss later in this module.

Convert the following MIPS assembly program into a MIPS assembly program that is in the MERL file format (similar to the rightmost column shown above).

```
lis $1
.word 0x1000
lis $2
.word A
A: jr $2
beq $0, $1, B
B: jr $31
```

[3]

We have a number of tools available.

cs241.linkasm is an assembler which will create MERL object code files.

```
cs241.linkasm < input.asm > output.merl
```

cs241.merl is a convenience tool which can be used to relocate a MERL file and make it ready to be loaded at a specified address. Its output is a non-relocatable MIPS binary file (the MERL header and footer are stripped, and all relocation entries relocated).

```
cs241.merl 0x1234 < output.merl > output.mips
```

The file `output.mips` can be loaded at address `0x1234`.

The simulators `mips.twoints` and `mips.array`, that we have previously used, can actually be given a second command line argument; the address at which to load a MIPS program.

```
mips.twoints output.mips [hex-address]
```

The command above will load the MIPS program at starting address `0x1234`.

[3] The answer is found in Section 6.1 at the end of this module.

3 Loader Relocation Algorithm

Recall we were discussing how the loader would know which words in the assembled file to relocate. Now that we have discussed the MERL object code format, let's look at the algorithm for relocation:

```
read_word()                    //skip the first word in the MERL file
endMod ← read_word()           // second word is the address of end of MERL file
codeSize ← read_word() - 12 // Use the third word to compute size of the code section
α ← findFreeRAM(codeSize)    // find starting address α

for(int i=0; i<codeSize; i+= 4)  // load the actual program (not the header and footer)
  MEM[α + i] ← read_word()       // starting at α

i ← codeSize + 12      //start of relocation table
while (i < endMod)
  format ← read_word()
  if (format == 1)     // indicates relocation entry
    rel ← read_word() // address to be relocated: this is relative to start of header
                      // not start of code, location to update is MEM[α + rel - 12]
    MEM[α + rel - 12] += α - 12  // go forward by α but also back by 12
                                 // since we did not load the header
  else
    ERROR // unknown format type
  i += 8  // update to next entry in MERL footer
```

We begin by reading and ignoring the first word in the input MERL file, which is there so that MERL files are directly executable if loaded at starting address `0x00`. The next word gives us the address for the very end of the MERL file. The third word gives us the address where the code segment ends (which coincides with the start of the MERL footer). The algorithm computes the size of the code, `codesize`. Since the word we just read is the address where the code segment ended, the size of the code is computed by subtracting the size of the header (since it appears before the code segment); 3 words or 12 bytes. The algorithm then determines the starting address α where the code segment will be loaded. We do not discuss how this contiguous block of size `codeSize` is found.

The algorithm then loads the code segment by reading `codeSize` number of words and placing them in memory starting at address α. Notice that any words in this code that were originally `.word label` have not yet been relocated. That comes in the next loop. This loop begins at the start of the relocation table (computed by adding back the 12 bytes we had subtracted to obtain `codeSize`).

The loop goes through the entire MERL footer (`i < endMod`). We are only expecting relocation entries. A word is read from the footer and checked that it is the value 1; the MERL format uses the value 1 to indicate that the next word is a relocation address. Later, we will discuss other entries that might exist in the MERL footer. Once the value 1 has been read, the algorithm retrieves the next word. This is the address that has to be relocated. But this address is the address of the word when there was a MERL header. Since we stripped the MERL header, we should subtract 12 from the address. Also, the address assumed that the first word was at address `0x00`. But since we now have α, the actual address of the word that needs to be relocated is found by adding α. Since the program has been loaded into memory, the value at `MEM[`α `+ rel - 12]`, where `rel` is the value we read from the relocation entry, needs to be relocated.

Relocating involves adding α. However, since we removed the header, we also must subtract 12. The algorithm then moves to the next entry in the MERL footer (incrementing the counter i by 8

since two words from the footer were read). This completes our discussion of loaders.

The Loader Relocation Algorithm video goes step by step using an example program to illustrate this algorithm.

We summarize what we have covered so far in this module. We have eliminated the restriction that a MIPS program must be loaded starting at address `0x00`. To achieve this, we needed an updated assembler which, instead of generating pure machine code, generates object code. This was necessary because pure machine code does not tell us which words originated from an occurrence of `.word label`, which is the only MIPS assembly construct affected by the starting address. We chose to use a custom object code format, MERL and discussed how the loader, given a MERL file, would load and relocate affected words.

One important caveat to be aware of is that hard-coding memory addresses in MIPS assembly is a really bad idea. For example, consider the following code:

```
lis $2
.word 12
jr $2
jr $31
```

This code loads the address 12 into register 2 and then jumps to that address. If this program is loaded starting at address `0x00`, the instruction at address 12 will be the return instruction `jr $31` and the program will terminate normally. However, if the program is loaded at any other address, the program will still attempt to jump to address 12, which might not even be in the memory segment assigned to this program by the loader. There will be no relocation entry for `.word 12` because there is no indication that the 12 here is the address 12 and not the constant 12. There is no way for the loader to solve this dilemma; the only solution is not to cause this problem in the first place. That is, never hard-code addresses and always use labels when referring to locations in the code.

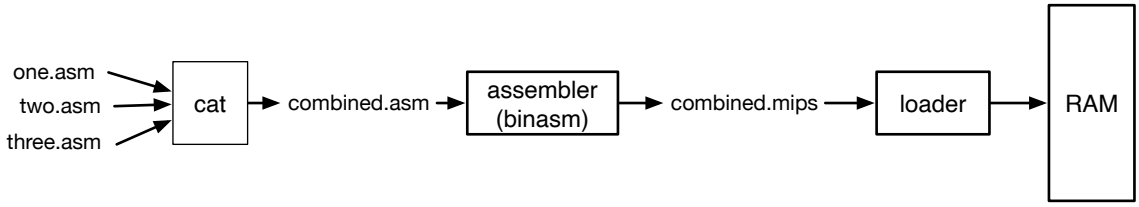
4 Linkers

In the code generation module, we touched upon relying on a runtime environment in the form of code made available to a program. We briefly talked about the need to link the files that contained the runtime code with the code generated by the compiler. In this section, we discuss the process of linking in more detail.

When we discussed MIPS assembly programming, we never considered separating the program into multiple files. That made sense since the programs we were writing were relatively small. However, programs do tend to get large and therefore it is useful to have the ability to split a program into multiple files. Additionally, much like the `print` and `alloc` modules that we relied on during code generation, it is useful to be able to rely on existing libraries when available. Finally, when working in a team, it is useful for team members to be working on their own files within the scope of the same project. While working with multiple files, one file would refer to code in a different file (using labels) and rely on the assembler to jump to that label.

This poses the question: how would the assembler resolve a reference to a label when the label is defined in a different file? One simplistic solution is to first combine the different assembly (`.asm`)

files into a single file and then assemble it. This is shown diagrammatically below:

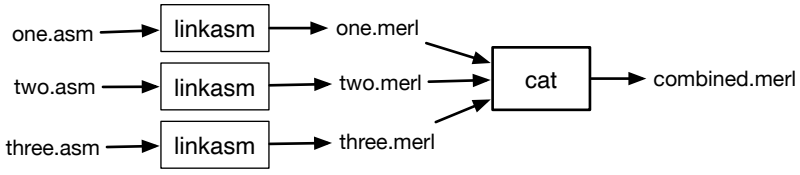


This could be as simple as running a command such as:

```
cat one.asm two.asm three.asm | cs241.binasm > combined.mips
```

And from a practical point of view this actually works. It does suffer from some limitations. First, it requires assembling all the code in one go, i.e., there is no ability to have pre-assembled versions of some files. Second, it assumes that all the source code is available. This is often not possible when using external libraries which might only be made available as pre-assembled files. Third, the different assembly files might have used the same labels. By concatenating these files, we might end up getting duplicate labels.

To avoid these limitations, one can consider assembling first and then concatenating. However, we cannot just create assembled MIPS binary; the output needs to be relocatable (after all, not all of these files, if any, can be loaded at starting address `0x00`). Luckily, we already had to solve the relocation problem for the loader! We can use our relocating assembler (`linkasm`) to produce MERL files for each assembly file, and then concatenate them. The following shows this process diagrammatically:



The problem with the above approach is that concatenating MERL files does not produce a valid MERL file. The `combined.merl` file would simply contain all the contents of `one.merl` followed by `two.merl` and then `three.merl`. But that is not valid MERL; it has multiple headers and footers interspersed through the code! We would need a smarter form of concatenation. This smarter concatenation algorithm is implemented as a program called the *linker*. We will discuss this shortly after we discuss an even bigger problem: what is our relocating assembler going to do when it encounters a label that is not defined in the same file?

The solution to handling externally defined references to labels is to update the assembler once again. Whenever the assembler encounters a label that cannot be resolved, the assembler will create an *External Symbol Reference* (ESR) entry in the MERL footer. It will then become the job of the linker to ensure that this label is resolved when it links the different MERL files.

One question that arises is whether the assembler should create an ESR entry every time it cannot resolve a label. Imagine that the programmer was not planning to use a label from another file and instead just typed a label incorrectly. If the assembler creates an ESR, then it is forcing the linker to resolve this label, which is likely not going to happen since it was a typo (indeed, if it

did resolve, it would be to an unexpected label!). There is a simple solution to avoid this; require the programmer to tell the assembler when to expect labels to be resolved at the linking stage. In MIPS assembly, we can do this using the `.import` directive (we discussed this earlier, in the code generation module).

```
.import print
```

Given this import directive, now if the assembler encounters a `print` label, it will generate an ESR. If a label cannot be resolved, and there is no import directive for it, then this causes an assembler error like before.

To illustrate the MERL format for ESRs, we will use the following short example:

```
; File: one.asm
.import proc
lis $1
.word proc
jalr $1
```

The complete MERL file is shown below in assembly and binary. **Recall that while the MERL format is a binary format, we typically show it in assembly format to make it easier to understand.** We also show the memory address where each word would reside if loaded at a starting address of `0x00`:

Address	MERL in assembly:	MERL output
0x00	`beq $0, $0, 2`	0x10000002
0x04	`.word endModule`	0x00000034
0x08	`.word endCode`	0x00000018
0x0c	`lis $1`	0x00000814
0x10	`use1: .word 0 ; placeholder for proc`	0x00000000
0x14	`jalr $1`	0x00200009
	`endCode:`	
0x18	`.word 0x11 ; format code for ESR`	0x00000011
0x1c	`.word use1 ; address where proc is used`	0x00000010
0x20	`.word 4 ; length of label`	0x00000004
0x24	`.word 112 ; ASCII for p`	0x00000070
0x28	`.word 114 ; ASCII for r`	0x00000072
0x2c	`.word 111 ; ASCII for o`	0x0000006f
0x30	`.word 99 ; ASCII for c`	0x00000063
0x34	`endModule:`	

The MERL file begins with the standard MERL header that has an instruction to jump over the next two words. The next two words are the addresses for where the module ends (`0x34`) and where the code segment ends (`0x18`). After the header is the code segment. The second word in this code segment is the use of `.word proc`. The assembler will of course discover that this label has not been declared in the assembly file. However, since the `import` directive is there, an ESR entry in the MERL footer will be created. The assembler must still produce some output for this word, as a placeholder for when the linker finds the actual address of `proc`. By convention, the assembler outputs the value 0 at this location, which will be replaced by the linker once the actual address of `proc` is known. After the code segment has been assembled, there is the MERL footer containing the ESR entry for `proc`, beginning at address `0x18`. The first word in the ESR entry is the format code for the ESR, the value `0x11`. This is followed by the address of the location where the label was used. Since `.word proc` appears at address `0x10`, that value is used. Next comes the length of the label, in this case 4 (p, r, o and c). Followed by the length, are the ASCII values for each

character in the label.

Let's now consider the other file; the file that is going to provide the label that will be linked (`proc` in our example above). One thing should be clear to the reader: the file providing the label would also need to be a MERL file, with some type of MERL entry that indicates where the provided label was in the assembly file. Recall that an assembled program does not have any labels, since labels are just assembler directives and do not exist in machine code. In the MERL format, entries for such labels are called *External Symbol Definitions* (ESD). This brings up the following question: which labels should have ESD entries? One could imagine creating an ESD for every label, but this would mean making labels visible that you never intended to (imagine every internal loop label being made available to link to). A better approach is to expect the programmer to specify which labels they want to be linkable, i.e., the programmer specifically *exports* the labels. This is done through the `.export` directive. For an example, we show the file `two.asm`, which exports the label `proc` that our previous file `one.asm` imported.

```
; File: two.asm
.export proc
proc:
jr $31
```

The assembled MERL file created for this file is shown below. Like before, we show the file in assembly for ease of reading, and we also show addresses for ease of referencing:

Address	MERL in assembly:	MERL output
0x00	`beq $0, $0, 2`	0x10000002
0x04	`.word endModule`	0x0000002c
0x08	`.word endCode`	0x00000010
	`proc:`	
0x0c	`jr $31`	0x03e00008
	`endCode:`	
0x10	`.word 0x05 ; format code for ESD`	0x00000005
0x14	`.word proc ; address where proc is defined`	0x0000000c
0x18	`.word 4 ; length of label`	0x00000004
0x1c	`.word 112 ; ASCII for p`	0x00000070
0x20	`.word 114 ; ASCII for r`	0x00000072
0x24	`.word 111 ; ASCII for o`	0x0000006f
0x28	`.word 99 ; ASCII for c`	0x00000063
0x2c	`endModule:`	

The MERL file, as always, begins with the header. Then comes the code segment which in this case is the trivial procedure with just the return instruction. At the end is the MERL footer, beginning at address `0x10`. The MERL format code for an ESD is the value `0x05` followed by the location where the exported label is defined. In our example this is the address `0x0c`. This is followed by the length of the label being exported, followed by the ASCII values for each character in the label.

Now that we have covered the complete MERL format, we are in a position to discuss the linker algorithm.

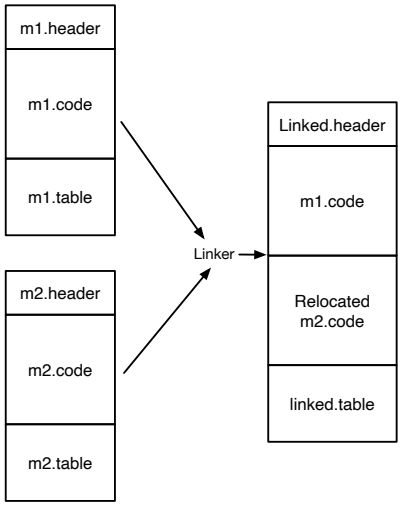
4.1 Linking Algorithm

We give a detailed algorithm for linking two MERL files `m1` and `m2` in pseudo-code. We first discuss the steps at a high level and then dive deeper into a concrete algorithm that the reader should

have no difficulty converting into actual code. In the following, we use m1.code and m1.table as shorthand to refer to just the code segment or table segment of the `m1` MERL file, respectively (similarly for the `m2` MERL file).

Task 1: Check for duplicate exports. Confirm that the two MERL files do not both export the same symbol(s).

Task 2: Combine code segments. We have framed the linker as a smart concatenation of MERL files. This means that the linker must concatenate the code segments, `m1.code` and `m2.code`. One important consideration is that the order in which the MERL files are provided to the linker matters. Linking files `m1` and `m2`, in this order, will result in `m1.code` appearing before `m2.code` in the linked file's code segment. The diagram on the right shows the intended outcome of the linker. Notice how m2.code follows m1.code. We also need to relocate m2.code; we discuss the reason in tasks 3 and 4. In task 2, we simply concatenate m1.code and m2.code without modifying either code segment; future tasks will be need to reach the final outcome shown in the diagram on the right.



Task 3: Relocate m2.table. A MERL table contains REL (relocation), ESD (External Symbol Definition) and ESR (External Symbol Reference) entries, each of which contain hard-coded addresses for where labels were defined or used with a `.word` directive. Since task 2 will move m2's code segment to below m1's code segment, we will need to update the addresses in REL, ESD and ESR entries in m2's table. This will require determining a relocation offset which will be equal to where m1's code segment ends minus the size of the header (12), since we will not be duplicating the header from m2 in the linked file. In other words, every table entry in m2 must now be updated by adding this relocation offset to the address that the entry contains.

Task 4: Relocate m2.code using REL entries. Each REL entry specifies an address in the code which needs to be relocated: it represents a "`.word` label" occurrence where the label value is known. In task 2, we moved m2.code so that it appears after m1.code, which changed the starting address of m2.code. In task 3, we changed the entries in m2.table so they are relative to the new starting address. However, there is still one more step: the lines of code that REL entries refer to need to be adjusted to accord with the new starting address. We need to go through each REL entry in the modified m2.table, and update the corresponding lines of m2.code by adding the relocation offset computed in task 3 to each such line. It might be hard to understand why this is necessary right now; we will see an example later.

Task 5: Resolve imports for m1. For each import (ESR) in m1.table, check if m2.table has a corresponding export (ESD). If such an export is found, the import can be resolved. Resolving an import requires updating the location of where the ESR occurs in m1's code with the address of where the exported label is defined. But that is not all! Recall that the ESR had been created because there was an unresolved label used with a `.word`. Since we have now resolved the label, we now have the use of a `.word` with a resolved label, i.e., we need a relocation entry. Therefore, we

must change the ESR entry we just resolved into an REL entry since if the label's definition was to move (due to relocation), the address would need to be relocated.

Task 6: Resolve imports for m2. We must resolve any imports (ESR) in m2.table by looking for matching exports in m1.table, i.e., we must do exactly the same as what we did in task 5 but with m1 and m2 tables swapped.

Task 7: Create the table for the linked MERL file. The table for the linked MERL file is the concatenation of all the ESD and REL entries along with any unresolved ESR entries from the tables from m1 and m2. We are guaranteed that there will be no duplicates. The order in which the table entries are stored does not matter.

Task 8: Compute the header for the linked MERL file. Recall that the MERL header requires outputting the total size of the MERL file (`endModule`) and the location where the code segment ends (`endCode`). The value for `endCode` is simply the size of the header (12) plus the size of the combined code segment concatenated in task 2. The value for `endModule` is `endCode` plus the size of the table that was created for the linked MERL file in task 7.

Task 9: Output the linked MERL file. This task is trivial. Output the header which requires outputting first the MERL Cookie (`beq $0, $0, 2` or `0x10000002`) followed by `endModule` and `endCode`. Then output the combined code that was produced in task 2, followed by the table that was produced in task 7.

We now give the algorithm discussed above as tasks in a more concrete pseudocode form below:

```
// Step 1: Check for duplicate export errors
for each ESD in m1.table {
  if there is an ESD with the same name in m2.table {
    ERROR (duplicate exports)
  }
}

// Step 2: Combine the code segments for the linked file
// The code for m2 must appear after the code for m1.
// We treat linked_code as an array of words containing just the
//   concatenation of the code segments
linked_code = concatenate m1.code and m2.code

// Step 3: Relocate m2's table entries
reloc_offset = end of m1.code - 12
for each entry in m2.table {
  add reloc_offset to the number stored in the entry
}

// Step 4: Relocate m2.code
// It is essential that this happen after Step 3
for each relocation entry in m2.table {
  index = (address to relocate - 12) / word size
  add relocation offset to linked_code[index]
}

// Step 5: Resolve imports for m1
for each ESR in m1.table {
```

```
  if there is an ESD in m2.table with a matching name {
    index = (address of ESR - 12) / word size
    overwrite linked_code[index] with the exported label value
    change the ESR to a REL
  }
}

// Step 6: Resolve imports for m2
Repeat Step 5 for imports from m2 and exports for m1

// Step 7: Combine the tables for the linked file
linked_table = concatenate modified m1.table and modified m2.table

// Step 8: Compute the header information
endCode = 12 + linked_code size in bytes
endModule = endCode + linked_table size in bytes

// Step 9: Output the MERL file
output merl cookie
output endModule
output endCode
output linked_code
output linked_table
```

4.2 Tracing Through the Linking Algorithm

We trace through the different steps of this algorithm using the following m1.asm and m2.asm files. Doing this manually is tedious and is going to take a while. While this can technically be skipped since everything you need to know about the algorithm has already been discussed, tracing step by step is important for truly understanding the algorithm, so we do it.

The description below can be considered a transcript for the Linking Algorithm video.

```
; m1.asm
.import foo
.export bar
sw $31, -4($30)

lis $29
.word foo
jalr $29

lis $3
.word bar
lw $3, 0($3)

lw $31, -4($30)
jr $31

bar: .word 0
```

```
; m2.asm
.export foo
.import bar
foo:    ; expects positive number in $1
lis $2
.word -1
lis $28
.word loop
lis $29
.word bar

loop:
lw $3, 0($29)    ; load current value at bar
add $3, $1, $3   ; add current value in $1 to it
sw $3,0($29)     ; store new value at bar
add $1, $1, $2   ; subtract $1 by 1
bne $1, $0, skip ; skip if not equal to 0
jr $31   ; if equal to 0 return
skip:
jr $28   ; jump to loop
```

Can you tell what the procedure `foo` in m2.asm does?

[4]

Next, let's take a look at the MERL files, m1.merl and m2.merl respectively.

```
; m1.merl
0x00: 10000002
0x04: 0000006c
0x08: 00000034
0x0c: afdfffc
0x10: 0000e814
0x14: 00000000
0x18: 03a00009
0x1c: 00001814
0x20: 00000030
0x24: 8c630000
0x28: 8fdfffc
0x2c: 03e00008
0x30: 00000000
; m1.table
; next column
```

```
; m1.table
0x34: 00000001;REL
0x38: 00000020
0x3c: 00000011;ESR foo
0x40: 00000014
0x44: 00000003
0x48: 00000066
0x4c: 0000006f
0x50: 0000006f
0x54: 00000005;ESD bar
0x58: 00000030
0x5c: 00000003
0x60: 00000062
0x64: 00000061
0x68: 00000072
```

```
; m2 merl
0x00: 10000002
0x04: 00000078
0x08: 00000040
0x0c: 00001014
0x10: ffffffff
0x14: 0000e014
0x18: 00000024
0x1c: 0000e814
0x20: 00000000
0x24: 8fa30000
0x28: 00231820
0x2c: afa30000
0x30: 00220820
0x34: 14200001
0x38: 03e00008
0x3c: 03800008
; m2.table
; next column
```

```
; m2.table
0x40: 00000001;REL
0x44: 00000018
0x48: 00000011;ESR bar
0x4c: 00000020
0x50: 00000003
0x54: 00000062
0x58: 00000061
0x5c: 00000072
0x60: 00000005;ESD foo
0x64: 0000000c
0x68: 00000003
0x6c: 00000066
0x70: 0000006f
0x74: 0000006f
```

It is worth taking some time to inspect the MERL files. In m1.merl, in m1's table (which spans addresses 0x34 to 0x68), there is one REL (words at addresses 0x34 and 0x38), one ESR for foo (words at addresses 0x3c to 0x50) and one ESD for bar (word at addresses 0x54 to 0x68). Similarly, m2's merl table (which spans addresses 0x40 to 0x74) has one REL (words at addresses 0x40 to 0x44), one ESR for bar (words at addresses 0x48 to 0x5c) and one ESD for foo (words at addresses 0x60 to 0x74).

We begin tracing through the steps discussed in the Linking algorithm. In step 1, there is one ESD for bar in m1.table but there is no ESD for bar in m2.table. So, step 1 is complete as no duplicate exports were found. step 2 is a mere concatenation of m1.code and m2.code. The code segment from m1's MERL file are all the words within addresses 0x0c to 0x30 inclusive in the m1.merl file. Similarly, the code segment from m2's MERL file are all the words within addresses 0x0c to 0x3c inclusive in the m2.merl file. We imagine that these words are copied into an array of words which we call `linked_code` in the algorithm, with the first word from m1's code segment `afdffffc` residing at index 0 within `linked_code`. After the 10 words that comprise m1's code segment (index 0 to 9 shown below), the subsequent 13 words (index 10 to 22) come from m2's code segment (with the first word in m2's code segment, `00001014` residing at index 10 in `linked_code`. We have on purpose kept the m1.code and m2.code in separate rows for clarity. Beside each index, we show the address where the code will appear in the final file; the addresses start at `0x0c` because of the header.

index	0	0x0c	1	0x10	2	0x14	3	0x18	4	0x1c	5	0x20	6	0x24	7	0x28	8	0x2c	9	0x30					
word	afdffffc		0000e814		00000000		03a00009		00001814		00000030		8c630000		8fdfffc		03e00008		00000000						
index	10	0x34	11	0x38	12	0x3c	13	0x40	14	0x44	15	0x48	16	0x4c	17	0x50	18	0x54	19	0x58	20	0x5c	21	0x60	22	0x64
word	00001014		ffffffff		0000e014		00000024		0000e814		00000000		8fa30000		00231820		afa30000		00220820		14200001		03e00008		03800008	

[4] foo computes the sum of all numbers from $1 to 0 and stores the result at the address associated with bar.

In step 3, we compute `reloc_offset`. The end of m1.code is the address `0x34` (52 in decimal) as given by word 3 in m1's header. This means our computed `reloc_offset` is 40 in decimal (52-12). We must add this value to each address in entries within m2.table. These addresses are found at location `0x44`, `0x4c` and `0x64` of the original m2 table. The modified m2 table, once 40 has been added to the identified addresses, is shown on the right.

```
; m2.table relocated
0x40: 00000001 ; REL
0x44: 00000040 ; originally 0x18
0x48: 00000011 ; ESR bar
0x4c: 00000048 ; originally 0x20
0x50: 00000003
0x54: 00000062
0x58: 00000061
0x5c: 00000072
0x60: 00000005 ; ESD foo
0x64: 00000034 ; originally 0x0c
0x68: 00000003
0x6c: 00000066
0x70: 0000006f
0x74: 0000006f
```

We are now at step 4, where we must relocate m2.code that is now part of the concatenated code that we call `linked_code`. Looking at the m2 table, we see that there is just one relocation entry and the address to relocate (from the modified m2 table) is `0x00000040`, 64 in decimal. We compute the index into our `linked_code` as (64 - 12) / 4 = 52/4 = 13. In other words, we must modify the value at index 13 which currently contains `00000024`, in hexadecimal. We add the relocation offset (40 in decimal) to this value. Since `0x00000024` is 36 in decimal, the new value is 36+40 = 76 in decimal or `0x0000004c`. The updated `linked_code` array of concatenated code is shown below:

index	0 0x0c	1 0x10	2 0x14	3 0x18	4 0x1c	5 0x20	6 0x24	7 0x28	8 0x2c	9 0x30			
word	afdfffc	0000e814	00000000	03a00009	00001814	00000030	8c630000	8fdfffc	03e00008	00000000			
index	10 0x34	11 0x38	12 0x3c	13 0x40	14 0x44	15 0x48	16 0x4c	17 0x50	18 0x54	19 0x58	20 0x5c	21 0x60	22 0x64
word	00001014	ffffffff	0000e014	0000004c	0000e814	00000000	8fa30000	00231820	afa30000	00220820	14200001	03e00008	03800008

Does what we just did all make sense? If we look at the original assembly code for m2, there was a ".word loop" referring to a label later in the file. In the MERL file, this became the hex value `0x24`, located at address `0x18`, and thus there was a REL entry for `0x18`.

After constructing `linked_code` in step 2, address `0x18` became `0x40`, and address `0x24` became `0x4c`. So, it does actually make sense: in step 3 we updated m2.table so that the REL entry points to `0x40` instead of `0x18`. And in step 4 we used this updated entry to change the value `0x24` to `0x4c`. The label reference now points to the right location in `linked_code`!

In step 5, we try to resolve the imports (ESR) for m1. There is one ESR in m1's table for a label named foo. We check and find that m2's table does have a corresponding export (ESD) for this label. The address of ESR, from m1's table is `0x14`. This means the index into our `linked_code` is (20-12)/4 = 2, i.e., the third word in the array. We find the exported label's value from the **modified** m2's table. This is `0x34`. This value is stored at `linked_code[2]`. The resulting concatenated code is shown below. Also, the ESR entry in m1's table is replaced by the following REL entry:

```
00000001 ; REL
00000014 ; same address as ESR
```

index	0 0x0c	1 0x10	2 0x14	3 0x18	4 0x1c	5 0x20	6 0x24	7 0x28	8 0x2c	9 0x30			
word	afdfffc	0000e814	00000034	03a00009	00001814	00000030	8c630000	8fdfffc	03e00008	00000000			
index	10 0x34	11 0x38	12 0x3c	13 0x40	14 0x44	15 0x48	16 0x4c	17 0x50	18 0x54	19 0x58	20 0x5c	21 0x60	22 0x64
word	00001014	ffffffff	0000e014	0000004c	0000e814	00000000	8fa30000	00231820	afa30000	00220820	14200001	03e00008	03800008

In step 6, we are again resolving ESRs but this time for m2. There is one ESR in m2's modified table for a label named bar. The address for this label is `0x48` (remember this was relocated). The

index becomes (72-12)/4 = 15. In m1's table there is an ESD for bar with the exported label's value being `0x30`. This value is stored at `linked_code[15]`. The resulting concatenated code is shown below. Also, the ESR entry in m2's table is replaced by the following REL entry:

```
00000001 ; REL
00000048 ; same address as ESR
```

index	0 0x0c	1 0x10	2 0x14	3 0x18	4 0x1c	5 0x20	6 0x24	7 0x28	8 0x2c	9 0x30			
word	afdfffc	0000e814	00000034	03a00009	00001814	00000030	8c630000	8fdfffc	03e00008	00000000			
index	10 0x34	11 0x38	12 0x3c	13 0x40	14 0x44	15 0x48	16 0x4c	17 0x50	18 0x54	19 0x58	20 0x5c	21 0x60	22 0x64
word	00001014	ffffffff	0000e014	0000004c	0000e814	00000030	8fa30000	00231820	afa30000	00220820	14200001	03e00008	03800008

We are in the home stretch now. In step 7 we concatenate the two modified tables from m1 and m2. We have split them across two columns just to save vertical space.

```
00000001;REL                    00000001;REL
00000020                        00000040
00000001;REL                    00000001;REL
00000014                        00000048
00000005;ESD bar                00000005;ESD foo
00000030                        00000034
00000003                        00000003
00000062                        00000066
00000061                        0000006f
00000072                        0000006f
```

In step 8 we compute the values for `endCode` and `endModule`. Since the size of `linked_code` is 92 (23 words each taking 4 bytes), `endCode` is 92 + 12 = 104. This is `0x68` in hexadecimal. To compute `endModule`, we add `endCode` to the size of the table we just created, which is 80 (there are 20 words). This totals 184 which is `0xb8`.

Finally, we output the linked MERL file in step 9.

```
0x00: 10000002                  ; linked table
0x04: 000000b8                  0x68: 00000001;REL
0x08: 00000068                  0x6c: 00000020
0x0c: afdfffc                   0x70: 00000001;REL
0x10: 0000e814                  0x74: 00000014
0x14: 00000034                  0x78: 00000005;ESD bar
0x18: 03a00009                  0x7c: 00000030
0x1c: 00001814                  0x80: 00000003
0x20: 00000030                  0x84: 00000062
0x24: 8c630000                  0x88: 00000061
0x28: 8fdfffc                   0x8c: 00000072
0x2c: 03e00008                  0x90: 00000001;REL
0x30: 00000000                  0x94: 00000040
0x34: 00001014                  0x98: 00000001;REL
0x38: ffffffff                  0x9c: 00000048
0x3c: 0000e014                  0xa0: 00000005;ESD foo
0x40: 0000004c                  0xa4: 00000034
0x44: 0000e814                  0xa8: 00000003
0x48: 00000030                  0xac: 00000066
0x4c: 8fa30000                  0xb0: 0000006f
0x50: 00231820                  0xb4: 0000006f
0x54: afa30000                  0xb8:
0x58: 00220820
0x5c: 14200001
0x60: 03e00008
0x64: 03800008
; linked table in
; next column
```

This must have been a painful read. It was painful writing it. However, following the algorithm step by step using a non-trivial example should have clarified the tasks the algorithm must perform.

Using the Linking algorithm just discussed, produce the resulting MERL file when the MERL files for `one.asm` and `two.asm` that we showed previously are linked in that order.
[5]

5 Concluding remarks

With the discussion of loading and linking now complete, we have now eliminated any unrealistic assumptions from our MIPS assembly programs. We are now free to write MIPS programs that span multiple files using the `.import` and `.export` directives to refer to use and define labels externally. These directives cause appropriate ESR and ESD entries to be created in the MERL object file. At the same time, our assembler will ensure that it creates REL entries for any uses of resolved labels with the `.word` directive. This ensures that if the location of that word is moved (due to linking or relocation), the word will be updated appropriately by the linker.

[5] The answer is found in Section 6.2 at the end of this module.

6 Answers to longer self-practice questions

6.1 Self-practice answer 1

Convert the following MIPS Assembly program into a MIPS assembly program that is in the MERL file format.

```
lis $1
.word 0x1000
lis $2
.word A
A: jr $2
beq $0, $1, B
B: jr $31
```

```
beq $0, $0, 2
.word endModule
.word endCode
lis $1
.word 0x1000
lis $2
reloc1:
.word A
A: jr $2
beq $0, $1, B
B: jr $31
endCode:
.word 0x1
.word reloc1
endModule:
```

6.2 Self-practice answer 2

Using the Linking algorithm just discussed, produce the resulting MERL file when the MERL files for `one.asm` and `two.asm` that we showed previously are linked in that order

Address	MERL in Assembly:	MERL output
0x00	`beq $0, $0, 2`	0x10000002
0x04	`.word endModule`	0x00000040
0x08	`.word endCode`	0x0000001c
0x0c	`lis $1`	0x00000814
0x10	`use1: .word 0x18`	0x00000018
0x14	`jalr $1`	0x00200009
	`proc:`	
0x18	`jr $31`	0x03e00008
	`endCode:`	
0x1c	`.word 0x05 ; format code for ESD`	0x00000005
0x20	`.word proc ; address where proc is defined`	0x00000018
0x24	`.word 4 ; length of label`	0x00000004
0x28	`.word 112 ; ASCII for p`	0x00000070
0x2c	`.word 114 ; ASCII for r`	0x00000072
0x30	`.word 111 ; ASCII for o`	0x0000006f
0x34	`.word 99 ; ASCII for c`	0x00000063
0x38	`.word 0x01 ; format code for REL`	0x00000001
0x3c	`.word use1 ; location to relocate`	0x00000010
0x40	`endModule:`	