COMP40 Assignment: Macro Instructions in an Assembler

No design document is required, but the instructions below suggest that you may want to prepare but not submit a short one covering some specific features anyway.

Contents

1 Purpose and overview

2 Introduction to the macro assembler

3 Sections: Executable binaries in real life
  3.1 Linux Executable binaries
  3.2 Feature creep and the proliferation of sections in Unix and Linux

4 Using the framework we provide
  4.1 Building the main function
  4.2 First interface: Assembler sections
  4.3 Second interface: Universal Machine macro instructions
  4.4 How to Build and link your Macro Assembler
  4.5 Dummy implementations to get you started

5 Planning your work
  5.1 Create a design document – but you won’t submit it
  5.2 Implementation plan
  5.3 Testing the Macro Assembler
  5.4 Avoid common mistakes

6 What to submit
1 Purpose and overview

In this assignment you will get more practice working with machine instructions. Norman Ramsey has written significant parts of an assembler and linker, which you will complete. The result will be a tool that you can use to assemble runnable .um files. You can test your results using your own UM, or you can use a debugged one that we provide for you at /comp/40/bin/um.

We expect that you will achieve the following:

- The assignment will solidify your understanding of binary machine code.
- The assignment will reinforce your understanding of machine-level computation; you will figure out how to use the Universal Machine to implement such computations as two’s-complement subtraction or bitwise and.
- You will gain experience with macro assemblers, which combine short sequences of machine instructions to create an assembly language that appears more fully functional than the underlying machine. You will implement a few macro instructions. Specifically, you will decide which combination of UM instructions are needed for each macro, and you will use interfaces we provide to generate the necessary code.
- You will learn how an executable binary can be composed from sections. By doing this, you will gain some insight into the operation of linkers, such as the Unix/Linux ld program.

Instructions for this assignment are split between this document and the handout http://www.cs.tufts.edu/comp/40/handouts/umaasm.pdf, which gives formal specifications for the UMASM language and for the macro assembler program. In this document:

- The chapter immediately below gives an introduction to why we use macros, with specific examples from the umaasm language.
- Subsequent chapters introduce the concept of umaasm sections, by first exploring similar section concepts as they exist in C and Unix/Linux systems.
- Following that, we provide suggestions and instructions for using the supplied framework to implement your assembler.
- Finally, we describe what we expect in your submission.

2 Introduction to the macro assembler

The Universal Machine has a very limited set of instructions. For example, there is no single Universal Machine instruction that allows you to perform an operation like

\[ r6 := r6 \times 10 \]  // not representable directly on UM

Because you will finish the semester by writing a modest program in UM assembly language, we have created an assembler that borrows temporary registers to make life easier for you in your role as an assembly-language programmer. This technique was first used to great success in the assembler for the MIPS architecture (the CPU of early versions of the Sony Playstation – newer Playstation 4 systems use AMD x86-64 chips).
A detailed description of the UM Macro Assembly Language\(^1\) is online; you should read it carefully now. To get started understanding how things work, it is enough to know that a programmer using UMASM can write:

\[
\text{.temp } r7 \\
r6 := r6 * 10 \\
r2 := m[r1][1] \quad // \ p = p \rightarrow \text{next}
\]

instead of the following, which maps more directly to UM instructions, but is more painful to code:

\[
\text{r7 := 10} \\
r6 := r6 * r7 \quad // \ r6 := r6 * 10 \\
r7 := 1 \\
r2 := m[r1][r7] \quad // \ p = p \rightarrow \text{next}
\]

The assembler will take the shorter input and generate a .um binary file with the longer form, i.e. using only the 14 instructions actually supported by the UM.

In order to use the macro features of the assembler, you have to tell the assembler which register contains zero and which register(s) it may use as temporaries. Here is an example:

\[
\langle \text{cat-abbrev.ums} \rangle \equiv \\
\begin{align*}
&\text{// copy standard input to standard output} \\
&\text{.zero } r0 \quad // \text{promise this register will always be zero} \\
&\text{.temps } r7 \quad // \text{the assembler may overwrite} \\
&\quad \text{// this register at will} \\
&L: \\
&r1 := \text{input()} \\
&\quad // \text{if } r1 \text{ is all ones, goto exit, else goto write} \\
&r2 := r1 \text{ nand } r1 \\
&r3 := \text{exit} \\
&\text{if } (r2 != 0) \ r3 := \text{write} \\
&\text{goto } r3 \\
&\text{write: output } r1 \\
&\text{goto } L \\
&\text{exit: halt}
\end{align*}
\]

We will shortly dive into the details of building your assembler, but first it’s worth considering the concept of sections, by studying how they are used in the Linux systems we use every day.

3 Sections: Executable binaries in real life

As you have seen from earlier projects, .um files contain nothing but a sequence of 32 bit UM instructions and data constants, all exactly as they are to appear in segment 0 when the program starts. These files contain no labels or symbols, no relocation information, and no other metadata.

Real executable binaries, such as the executable files we run on Linux do have metadata, and they are produced from relocatable object files (“dot-O files”) which also have metadata. In this project, you won’t directly implement relocation — we will implement label references for you. You will focus on structuring the binary into sections and emitting those sections in the proper order, as a “real” linker would do.

The topics of object code, binaries, and linking are covered at length (30 pages or so) in Chapter 7 of your text by Bryant and O’Hallaron. What follows is a summary of what you most need to know. Keep in mind that the remainder of this discussion is about C and Unix programs; this is intended to motivate your work on UMASM, to explain why what you’re doing is important, and to give you a feel for how ”real” systems do this sort of thing.

The instructions for the work you will do to implement sections are in the chapter below titled ”Using the Framework We Provide”.

3.1 Linux Executable binaries

An executable binary is divided into sections. Each section identifies a block of memory and the intended use of that block. At load time, sections are mapped (by the operating system) into a running process image:

- The text section contains machine code, and it is typically mapped into the address space read-only and shared.\(^2\) If the memory-management unit includes an execute bit, the execute bit is set to show that it is OK to run machine code in the text section.

- The data section contains initialized data, i.e., initialized global variables in C. It is mapped read/write and not shared, and the initial contents are as specified in the executable binary.

- The bss section contains no data at all! Instead it specifies space that is to be reserved in the running process image to hold uninitialized data, i.e., uninitialized global variables in C. When the operating system creates a process image, the bss section is mapped to read/write, unshared memory and is initialized to zeroes.

Why not just represent a program binary as a single block, as on the Universal Machine? Using three sections saves memory and disk space, and when the three-section format was designed, these resources were scarce.

- The distinction between text and data saves memory by allowing multiple processes to share physical memory for the text section.

- The distinction between data and bss saves disk space by providing a very compact representation of a large block of zeroes.

\(^2\)Shared means that if you and I are both running vim on the Linux server, the virtual addresses of the code in our two different processes are mapped to the same physical memory. Sharing is an important way to enable a lot of processes to share limited physical RAM.
3.2 Feature creep and the proliferation of sections in Unix and Linux

Three sections were good enough for Ritchie and Thompson, but not for Stallman. If you aim `objdump -h` at a relocatable object file or especially at an executable binary, you may be overwhelmed by all the sections that you find. Here’s a short guide (more on page 544 of the book):

- The **rodata** section contains read-only data. This addition is actually justified: being read-only, it can be shared, but (on hardware that supports an execute bit) programs should not be permitted to execute it. In C it is used primarily to store string literals and floating-point literals.

- The **init** and **fini** sections contain code intended to run before **main** and after **exit**, respectively. They play little role in C but are useful for more featureful languages like C++ and Modula-3.

- There are myriad sections containing read-only data intended for the debugger. These sections are often **not** mapped into the running process image, but they are where the compiler leaves tracks telling the debugger what the compiler did with the local variables, how machine-code locations map to source lines, and so on.

- Sections like **got** (global offset table) and **plt** (procedure linkage table), along with a host of others, support dynamic linking, a horrendously complicated subject which is beyond the scope of COMP 40.

- Some sections have been created by people at the Free Software Foundation for their own purposes. For example, in every binary they leave a footprint which marks the binary as created by their tools.

4 Using the framework we provide

Implementing an assembler and linker is a big job. It includes parsing a textual assembly language, which you might have to design yourself; working with an on-disk representation for relocatable object code, part or all of which you might have to design yourself; implementing one or two dozen different kinds of relocation, depending on the number of binary instruction formats in play; searching for undefined symbols in libraries; and implementing the “macro instructions” that make it possible to use a literal constant where the machine expects only a register, for example.

In order to create a project that is interesting, teaches ideas of lasting value, and can be completed in your lifetime, we take a few shortcuts:

- We have designed an assembly language for the Universal Machine, and we have written a parser for it.

- We have implemented in the supplied framework the most delicate of the macro instructions: the ones that permit you to use a segment reference or in some cases a constant where the hardware permits only a register.

- We’ve implemented all the relocation. Although this decision means that you won’t get to learn how relocation works, it simplifies your task dramatically.

It remains to you to implement the following:

- Six simple macro instructions, each of which can be implemented with at most one temporary register.
• Loading of general 32-bit constants despite the fact that the hardware loads only 25-bit constants

• Management of an arbitrary sequence of named sections

• Concatenation of sections into an executable binary and emission of the executable binary to a file in the UM format

Because of the magnitude of the project, we are giving you relatively little freedom of design. We have determined an architecture and set up two interfaces for you to implement.

4.1 Building the main function

You will create a main function, but we give you a function that you should call immediately from that:

```c
// umasm.h

#ifndef UMASM_INCLUDED
#define UMASM_INCLUDED

extern int Umasm_run(int argc, char *argv[]);
/* run the Universal Machine macro assembler as the main program */

#endif
```

Your main function can simply call Umasm_run, passing argc and argv unchanged.
The function \texttt{Umasm\_run} will do all the work of assembling and writing the executable .\texttt{um} file for a program, but from time to time it will call the following functions that you provide to do various steps in the processing.

4.2 First interface: Assembler sections

The first interface is specified in \texttt{/comp/40/include/umsections.h}.

\begin{verbatim}
#include <stdint.h>
#include <stdio.h>

#define T Umsections_T
typedef struct T *T;

/* A value of type T represents a nonempty *sequence* of named sections.
 Each section contains a sequence of Universal Machnine words.
 In addition, each value of type T identifies one section in
 the sequence as the 'current' section, and a value of type T
 encapsulates an error function. */

/* Callers of procedures in this interface need *not* provide
 long-lived pointers to strings. Callers may reuse string memory
 as soon as the call returns. */

/* declarations of functions exported from umsections.h */

#undef T
#endif
\end{verbatim}

It should be a checked run-time error to pass a null \texttt{Umsections\_T} to any of the functions exported from \texttt{umsections.h}.

This interface, although long, is fairly easy to implement. My code takes about 160 lines of C. Please implement this interface, and put your implementation in file \texttt{umsections.c}.

The assembler datatype To represent the \texttt{struct Umsections\_T} you may include any fields you like. Here's some advice:

- You have to have a \texttt{sequence} of sections, but you also have to be able to switch to any section by name. It might help to have both a \texttt{Seq\_T} and a \texttt{Table\_T}. Don't duplicate elements in the sequence.

- At any point in time, the assembler has to keep track of which section is \texttt{current}. You have to be able to append instructions and data to the current section.
The assembler stores an error state and an error function which are passed in at assembler-creation time. All errors should be signalled by calling that function. To create an error message, you may want to use Hanson’s Fmt_String, which combines sprintf and malloc in one convenient package.

When an assembler is created, the caller tells the assembler the name of the first section. This section becomes the current section of the assembler, and it is also the first in the assembler’s sequence of sections. The section is initially empty.

The error and errstate parameters should be stored in the assembler so they can be used to signal errors.

The Umsections_free function, like Hanson’s free functions, takes a pointer to a Umsections_T. It is an unchecked run-time error to pass any Umsections_T to any function after it has been freed.

For the Umsections_error function, implement a simple function that allows you to signal an error.

Sections Like most assemblers, the UM assembler combines “create new section” and “switch to existing section” in a single operation. If the named section is not already part of the assembler, this operation creates a new section and appends it to the assembler’s sequence of sections. The named section is then made the current section.

Emitting words Anything that corresponds to a global variable in C will correspond to initialized data in the assembly language. Initialized data is created by the Umsections_emit_word function, which appends the word to the current section. You will also use Umsections_emit_word to emit instructions.
Support for linking and relocation  You won’t need to implement linking and relocation. But you will need to support my implementation of linking and relocation. We need to be able to observe sections and mutate their contents.

Writing an executable binary  Once all sections are complete, my code will backpatch instructions and data as needed to account for relocation. At that point, my code will call Umsections.write to write binary code to disk.

4.3 Second interface: Universal Machine macro instructions

The second interface you will implement emits macro instructions.
It should be a checked run-time error to pass a null \texttt{UmSections\_T} to any function exported by the \texttt{ummacros.h} interface.

This interface, although shorter than the first interface, is harder to implement. My code is about 110 lines of C, of which about 60 lines are devoted to the two functions that implement macro instructions. Please implement this interface, and put your implementation in file \texttt{ummacros.c}. You'll think carefully about how to emulate the macro operations using the Universal Machine's 14 instructions.

\textbf{Opcodes} The opcode interface requires no implementation; it defines an enumeration type that is shared by the assembler, the disassembler, and the Universal Machine emulator.

\begin{verbatim}
10a \langle \texttt{um-opcode.h} 10a \rangle ≡
    \#ifndef UM_OPCODE_INCLUDED
    \#define UM_OPCODE_INCLUDED

    \langle \text{definition of hardware opcodes as type } \texttt{Um\_Opcode} 10b \rangle

    \#endif

Here are the opcode definitions:

\begin{verbatim}
10b \langle \text{definition of hardware opcodes as type } \texttt{Um\_Opcode} 10b \rangle ≡
    \#define \texttt{CMOV} = 0, \texttt{SLOAD}, \texttt{STORE}, \texttt{ADD}, \texttt{MUL}, \texttt{DIV},
          \texttt{NAND}, HALT, ACTIVATE, INACTIVATE, OUT, IN, LOADP, LV

    \texttt{Um\_Opcode};
\end{verbatim}

By adding the following macro definitions, you will make it far easier to write programs in UM assembly language:

\begin{verbatim}
10c \langle \text{definition of macro opcodes as type } \texttt{Ummacros\_Op} 10c \rangle ≡
    \#define \texttt{MOV} = \texttt{LV+1}, \texttt{COM}, \texttt{NEG}, \texttt{SUB}, \texttt{AND}, \texttt{OR}

    \texttt{Ummacros\_Op};
    \* move, one's complement (\textasciitilde{}), two's-complement negation (-),
    \ subtract, bitwise &\textasciitilde{}, bitwise | */
\end{verbatim}

The semantics are as follows:

\begin{tabular}{|c|c|c|}
    \hline
    \textbf{Number} & \textbf{Operator} & \textbf{Action} \\
    \hline
    14 & Move & \$r[A] := \$r[B] \\
    15 & Bitwise Complement & \$r[A] := \neg \$r[B] \\
    16 & Two's-Complement Negation & \$r[A] := -\$r[B] \mod 2^{32} \\
    17 & Subtraction & \$r[A] := (\$r[B] - \$r[C]) \mod 2^{32} \\
    18 & Bitwise And & \$r[A] := \$r[B] \wedge \$r[C] \\
    19 & Bitwise Or & \$r[A] := \$r[B] \lor \$r[C] \\
    \hline
\end{tabular}
Implementing macro instructions  The Ummacros_op function is used to emit sequences of UM instructions which implement the six “macro instructions.” Some macro instructions, like COM, can be emitted without using a temporary register. Others, like subtraction, require a temporary register. No macro instruction requires more than one temporary register. If a temporary register is available, its number is passed in the argument temporary. If no temporary register is available, the argument temporary contains -1. If a temporary is needed but none is available, this function should call Umsections_error.

11a ⟨declarations of functions exported from ummacros.h⟩ \equiv (9c) 11b

\begin{verbatim}
void Ummacros_op(Umsections_T asm, Ummacros_Op operator, int temporary,
                 Ummacros_Reg A, Ummacros_Reg B, Ummacros_Reg C);
/* Emit a macro instruction into 'asm', possibly overwriting temporary
 register. Argument of -1 means no temporary is available.
 Macro instructions include MOV, COM, NEG, SUB, AND, and OR.
 If a temporary is needed but none is available, Umsections_error(). */
\end{verbatim}

Important: Each macro instruction must be justified by one or more algebraic laws. For example, if we have figured out a way to implement bitwise complement, we can justify an implementation of bitwise AND using the following law, which relies on COM and on the native NAND instruction:

\[ x \land y = \neg (\neg (x \land y)). \]

The laws should appear in the code near the implementations that they justify.

The Ummacros_load_literal function emits code to load a 32-bit literal value. If the literal value k fits in 25 unsigned bits, Ummacros_load_literal can use a single Load Value instruction. Otherwise, it will have to use a sequence of instructions, possibly requiring a temporary register. (Hint: if the complement of k fits in 25 unsigned bits, no temporary register is needed. You must handle this case without a temporary.)

11b ⟨declarations of functions exported from ummacros.h⟩ \equiv (9c) <9a

\begin{verbatim}
void Ummacros_load_literal(Umsections_T asm, int temporary,
                         Ummacros_Reg A, uint32_t k);
/* Emit code to load literal k into register A.
 Must work even if k and \sim k do not fit in 25 bits---in which
 case temporary register may be overwritten. Call the
 error function if temporary is needed but is supplied as -1 */
\end{verbatim}

Your implementation of Ummacros_load_literal must also be justified by algebraic laws. And if your implementation has multiple cases, you need at least one law for each case.

11
4.4 How to Build and link your Macro Assembler

The Macro Assembler framework is written in a combination of C (700 lines) and Lua (800 lines) stuck together with about 500 lines of glue code. In addition to Hanson’s CII library, it also uses the LPEG parsing library implemented by Roberto Ierusalimschy. All that code is just the outer shell which we provide for you; however, without your core assembler of roughly 300–350 lines of C, it doesn’t do anything.

We provide useful interfaces in the following .h files:

- `um-opcode.h` An enumeration type for opcodes
- `umasm.h` Declaration of `Umasm_run`, to be called from `main`
- `umsections.h` One interface you are to implement
- `ummacros.h` The other interface you are to implement

You can compile against these interfaces using `-I/comp/40/include`. To link against them, you will need

```
gcc ... -L/comp/40/lib64 -lumasm -llua5.1 -lpeg -llua `pkg-config --libs cii40` ...
```

You will probably also need to link against the math library (`-lm`).
4.5 Dummy implementations to get you started

We provide “dummy” implementations of the two interfaces that you (eventually) will implement; these implementations don’t do anything useful, except that they write to standard error when called.

The purpose of these implementations is to enable you to learn, by experiment, how an assembly-language program affects what functions are called. That is, you can very early in the development process build a version of umasm that links with the dummies, which will be used in place of any functions you haven’t yet written. Remember: the gcc linker program ld looks in order on the command line for .o files and libraries from which to satisfy external references; if you list the dummy library after the .o files with your own implementations, then the dummies will be linked only for functions that you haven’t yet written! So, do it like this:

```bash
gcc <your UMasm.o, etc.> -L/comp/40/lib64 -lumasm -lumasm-dummy -llua5.1-lpeg -llua 'pkg-config --libs cii40' ...
```

Try assembling different simple umasm programs. Because at first so much is missing, you won’t actually do the assembly, but you will see from the dummy function output which functions are called when.

Note that code using only the dummy implementations will halt at link time with an assertion failure, because no code has been generated.

5 Planning your work

We recommend the following strategy for completing this assignment.

5.1 Create a design document – but you won’t submit it

Most of the assembler is built for you, and most of the rest is designed for you. Moreover, we also provide the outline of an implementation plan. We therefore see little purpose in having you submit a design document before the assignment is due. But we do recommend that before you start coding, you create an abbreviated design document covering just these points:

1. How will you represent the sequence of sections of type Umsections_T?

2. How does your representation of Umsections_T relate to the world of ideas (sections and instructions)?

3. What are the invariants of your representation of Umsections_T?

4. For each function that you have to implement, what part(s) of which representation(s) do you expect it to depend on?

Except for item 4, these questions should be answered in the documentation of the code you eventually submit.

5.2 Implementation plan
We recommend this implementation plan:

1. Link with the dummy implementations, and experiment until you get an idea what is going on.

2. Build the first version of your own code so that it implements sections correctly, uses the dummy version of Ummacros_op, and uses a version of Ummacros_load_literal that handles only 25-bit unsigned literals. If we are right, you should be able to test your code on the (cat-abbrev.ums 3) in this handout on page 3, which you should be able to run.

3. To unit test your code, create an assembly-language program in file test.ums. The program should contains one each of the UM machine instructions, but the program need not do anything when run. *Do not use any macro instructions!* You can test your assembler as follows:

   ```
   use comp40
   ./umasm test.ums | umdump -bare | diff test.ums -
   ```

   If everything is implemented correctly, the “bare dump” of your executable binary should be identical to the input file test.ums, and you should see no output.

4. System test your code by disassembling, reassembling, and then re-disassembling the a binary such as midmark.um:

   ```
   use comp40
   umdump -bare midmark.um | ./umasm | umdump -bare > midmark_temp.ums
   umdump -bare midmark.um | diff - midmark_temp.ums
   ```

   You should see no differences. Note well that umasm and umdump *are not half-inverses.* But umasm and umdump *are half-inverses on the output of umdump -bare.* That’s why a three-stage test is required. Also: you can delete midmark_temp.ums when you’re done testing.

5. Implement Ummacros_load_literal so that it handles fully general 32-bit literals. Unit test it using both positive and negative literals in the assembly source.

6. Implement the macro operations in Ummacros_op. Write a test file that tests all six macro operations for correctness. Assemble, link, and run the file.

7. Submit. You are finished.

---

3Since this point is frequently overlooked, We put it another way: running umdump -bare | umasm is *not* the identity function on UM files. In particular, if you run umdump -bare midmark.um | ./umasm, you will *not* recreate the midmark.um binary. In fact, you will not create any runnable binary at all. Trying to run a binary created from the output of umdump may result in frustration, lost sleep, and other bad outcomes.
5.3 Testing the Macro Assembler

We have unit-tested almost all of the Macro Assembler instructions, with special emphasis on the conditional gotos. Our testing has found enough bugs that we strongly suspect there are many more bugs lurking. Ideally we would create a program to generate a full test suite using randomized inputs.

To begin testing your own assembler, use the output of umdump -bare. (The -bare option dumps only the assembler syntax, and not the usual lists of addresses or hex representations.) Given any binary foo.um, you can try the following sequence of commands:

```
umdump -bare foo.um | tee foo.dump | ./umasm > new-foo.um
umdump -bare new-foo.um | diff foo.dump -
```

If your assembler handles the basic instructions correctly, the diff command should show no differences. Of course, this test addresses only the basic assembly of instructions.

- It does not test labels or relocation.
- It does not test the six macro operators.

To test these operators, you will have to devise your own .ums files.

5.4 Avoid common mistakes

Avoid these common mistakes:

- Don’t forget to document every field of every data type and to state all invariants.
- If you use Hanson’s polymorphic data structures, don’t forget to say how each void * is instantiated.
- Don’t forget that the default key in a Hanson Table T has to be an atom, not just any string.
- Don’t forget that the implementation of each macro instruction must be justified by one or more algebraic laws. Your implementation of loading general 32-bit literals must be similarly justified.
- Don’t assume that source and destination registers are distinct. In practice the destination register is often identical to one or more source registers.
- Don’t forget that testing with umdump -bare is a minimum sanity check only. It’s a common mistake to believe that if your assembler passes this test, it works. Nothing could be less likely. Using umdump
  - Doesn’t test any macro features (yours or mine)
  - Doesn’t test if you use temporary registers properly
  - Doesn’t expose common mistakes in implementing the six macro instructions
- Don’t submit an implementation of the Universal Machine along with your assembler. They are nearly independent and should be treated as such.
- Don’t make the rare mistake of using umdump -bare to disassemble a binary, using umasm to assemble the results, and then to trying to run the resulting binary. It will not work. Although this mistake is rare, it is worth mentioning because the consequences can be devastating: hours spent “debugging” a program that is actually working.

4The new conditional moves have not been tested.
6 What to submit

Use the script submit40-asm to submit

- All .c and .h files you have written, which must include umsections.c and ummacros.c.
  The file ummacros.c must contain a justification, in the form of algebraic laws, for the implementation of each macro instruction, and also for the general case of loading 32-bit values.

- A script called compile that compiles all your .c files into .o files and then links the executable binary umasm.

- A README file which
  - Identifies you and your programming partner by name
  - Acknowledges help you may have received from or collaborative work you may have undertaken with others
  - Identifies what has been correctly implemented and what has not
  - Says approximately how many hours you have spent analyzing the assignment
  - Says approximately how many hours you have spent building your assembler and linker
  - Says approximately how many hours you have spent debugging your assembler and linker