COMP40 Assignment: Macro Instructions in an Assembler

Assignment due Sunday, December 4 at 11:59 PM.
There is no design document.

Contents

1 Purpose and overview

In this assignment you will get more practice working with machine instructions. I have written an assembler and linker, which you will complete.

- 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 be exposed to a *macro assembler*, which combines short sequences of machine instructions to create an assembly language that appears more fully functional than the underlying machine. You will write a few macro instructions.
- You will learn how an executable binary can be compsed from *sections*.

2 Executable binaries in real life

A Universal Machine binary is a sequence of words without metadata. Real executable binaries do have metadata, and they are produced from relocatable object files ("dot-O files") which also have metadata. In this project, you'll be shielded from the details of relocatable object code; you'll focus on structuring the binary into sections and on creating the illusion of a larger instruction set.

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.

2.1 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*.[1] 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.

[1] *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.

- 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.

2.2 Feature creep and the proliferation of sections

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.

3 A 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 * 10   // not representable directly on UM
```

Because you will finish the semester by writing a modest program in UM assembly language, I 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 the Sony Playstation).

A detailed description of the UM Macro Assembly Language[2] is online; for here, it is enough to know that you can write

```
.temp r7
r6 := r6 * 10
r2 := m[r1][1]  // p = p -> next
```

instead of the more faithful but painful

```
r7 := 10
r6 := r6 * r7    // r6 := r6 * 10
r7 := 1
r2 := m[r1][r7]  // p = p -> next
```

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:

4 ⟨*cat-abbrev.ums* 4⟩≡

```
// copy standard input to standard output

.zero r0   // promise this register will always be zero
.temps r7  // the assembler may overwrite
           // this register at will
L:
r1 := input()
// if r1 is all ones, goto exit, else goto write
r2 := r1 nand r1
r3 := exit
if (r2 != 0) r3 := write
goto r3
write: output r1
goto L
exit: halt
```

4 What we expect from you

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 will take several shortcuts:

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

[2]See URL `http://www.cs.tufts.edu/comp/40/handouts/umasm.html`.

- I have implemented 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.
- I'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, I am giving you very little freedom of design. I have determined an architecture and set up two interfaces for you to implement.

4.1 First interface: Assembler sections

The first interface is specified in `/comp/40/include/umsections.h`.

5 ⟨*umsections.h* 5⟩≡

```
#ifndef UMSECTIONS_INCLUDED
#define UMSECTIONS_INCLUDED
/* Full documentation for this interface is at http://tinyurl.com/2uwhhtu */

#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.
  */

⟨declarations of functions exported from umsections.h 6a⟩

#undef T
#endif
```

It should be a checked run-time error to pass a null `Umsections_T` to any of the functions exported from `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 `umsections.c`.

The assembler datatype To represent the `struct Umsections_T` you may include any fields you like. Here's some advice:

- You have to have a *sequence* of sections, but you also have to be able to switch to any section by name. It might help to have *both* a `Seq_T` and a `Table_T`. Don't duplicate elements in the sequence.
- At any point in time, the assembler has to keep track of which section is *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.

6a ⟨*declarations of functions exported from* `umsections.h` 6a⟩≡ (5) 6b ▷

```
T Umsections_new(const char *section,
            int (*error)(void *errstate, const char *message),
            void *errstate);
    /* Create a new assembler which emits into the given section.
       The error function, which is called in case of errors,
       must not return.
    */
```

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.

6b ⟨*declarations of functions exported from* `umsections.h` 6a⟩+≡ (5) ◁ 6a 6c ▷

```
void Umsections_free(T *asmp);  // destroy an old assembler
```

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

6c ⟨*declarations of functions exported from* `umsections.h` 6a⟩+≡ (5) ◁ 6b 6d ▷

```
int Umsections_error(T asm, const char *msg);
  /* call the assembler's error function, using the error state
     passed in at creation time */
```

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.

6d ⟨*declarations of functions exported from* `umsections.h` 6a⟩+≡ (5) ◁ 6c 7a ▷

```
void Umsections_section(T asm, const char *section);
  /* start emitting to the named 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.

7a ⟨*declarations of functions exported from* `umsections.h` 6a⟩+≡ (5) ◁6d 7b▷
```
typedef uint32_t Umsections_word; // Universal Machine word
void Umsections_emit_word(T asm, Umsections_word data);
  /* Emit a word into the current section */
```

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. I need to be able to observe sections and mutate their contents.

7b ⟨*declarations of functions exported from* `umsections.h` 6a⟩+≡ (5) ◁7a 7c▷
```
void Umsections_map(T asm, void apply(const char *name, void *cl), void *cl);
  /* for each section name S in the sequence in 'asm', in order,
     call apply(S, cl) */
int Umsections_length(T asm, const char *name);
  /* if 'name' is a named section in asm, return the number of words
     in that section; otherwise call asm's error function */
Umsections_word Umsections_getword(T asm, const char *name, int i);
  /* Return the word numbered 'i' from the section with the given name.
     The first word is numbered 0.
     If there is no section with the given name, or if i is out of bounds,
     call asm's error function. */
void Umsections_putword(T asm, const char *name, int i, Umsections_word w);
  /* In the section with the given name, replace word 'i' with 'w'.
     If there is no section with the given name, or if i is out of bounds,
     call asm's error function. */
```

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.

7c ⟨*declarations of functions exported from* `umsections.h` 6a⟩+≡ (5) ◁7b
```
void Umsections_write(T asm, FILE *output);
  /* Write the contents of each section stored in asm,
     in the order in which they appear in asm's sequence.
     Write the words to file 'output' in UM format.
  */
```

4.2 Second interface: Universal Machine macro instructions

The second interface you will implement emits macro instructions.

8a ⟨*ummacros.h* 8a⟩≡

```
#ifndef UMMACROS_INCLUDED
#define UMMACROS_INCLUDED
/* Full documentation for this interface is at http://tinyurl.com/2uwhhtu */

#include "umsections.h"
#include "um-opcode.h"

⟨definition of macro opcodes as type Ummacros_Op 8d⟩

typedef enum Ummacros_Reg { r0 = 0, r1, r2, r3, r4, r5, r6, r7 } Ummacros_Reg;
  /* Represents a UM register number, which several functions in this
     interface expect as arguments. */

⟨declarations of functions exported from ummacros.h 9a⟩
#endif
```

It should be a checked run-time error to pass a null `Umsections_T` to any function exported by the `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 `ummacros.c`. You'll think carefully about how to emulate the macro operations using the Universal Machine's 14 instructions.

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.

8b ⟨*um-opcode.h* 8b⟩≡

```
#ifndef UM_OPCODE_INCLUDED
#define UM_OPCODE_INCLUDED

⟨definition of hardware opcodes as type Um_Opcode 8c⟩

#endif
```

Here are the opcode definitions:

8c ⟨*definition of hardware opcodes as type* `Um_Opcode` 8c⟩≡ (8b)

```
typedef enum Um_Opcode {
  CMOV = 0, SLOAD, SSTORE, ADD, MUL, DIV,
  NAND, HALT, ACTIVATE, INACTIVATE, OUT, IN, LOADP, LV
} Um_Opcode;
```

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

8d ⟨*definition of macro opcodes as type* `Ummacros_Op` 8d⟩≡ (8a)

```
typedef enum Ummacros_Op { MOV = LV+1, COM, NEG, SUB, AND, OR } Ummacros_Op;
  /* move, one's complement (~), two's-complement negation (-),
     subtract, bitwise &, bitwise | */
```

The semantics are as follows:

Number	*Operator*	*Action*
14	Move	$\mathtt{\$r}[A] := \mathtt{\$r}[B]$
15	Bitwise Complement	$\mathtt{\$r}[A] := \neg\mathtt{\$r}[B]$
16	Two's-Complement Negation	$\mathtt{\$r}[A] := -\mathtt{\$r}[B] \bmod 2^{32}$
17	Subtraction	$\mathtt{\$r}[A] := (\mathtt{\$r}[B] - \mathtt{\$r}[C]) \bmod 2^{32}$
18	Bitwise And	$\mathtt{\$r}[A] := \mathtt{\$r}[B] \wedge \mathtt{\$r}[C]$
19	Bitwise Or	$\mathtt{\$r}[A] := \mathtt{\$r}[B] \vee \mathtt{\$r}[C]$

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`.

9a ⟨*declarations of functions exported from* `ummacros.h` 9a⟩≡ (8a) 9b▷
```
  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(). */
```

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

$$x \wedge y = \neg(\neg(x \wedge y)).$$

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.)

9b ⟨*declarations of functions exported from* `ummacros.h` 9a⟩+≡ (8a) ◁9a
```
  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 ~k do not fit in 25 bits---in which
       case temporary register may be overwritten.  Checked RTE if
       temporary is needed and is -1 */
```

Your implementation of `Ummacros_load_literal` must *also* be justified by algebraic laws.

4.3 Your design document

Most of the assembler is built for you, and most of the rest is designed for you. Moreover, I also provide the outline of an implementation plan. I therefore see little purpose in having you submit a design document before the assignment is due. But I 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 What we provide for you

5.1 The rest of the assembler

We provide a program that runs the assembler and calls your routines.

10 ⟨*umasm.h* 10⟩≡
```
#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.

5.2 List of interfaces

We provide these interfaces:

`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`).

5.3 Dummy implementations to get you started

I provide “dummy” implementations of the two interfaces; these implementations write to standard error. The purpose of these implementations is to enable you to learn, by experiment, how an assembly-language program affects what functions are called. You can use the dummy implementations by linking with `-lumasm-dummy` immediately *after* `-lumasm`. Code using *only* the dummy implementations will halt at link time with an assertion failure.

5.4 Implementation plan

I 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 I am right, you should be able to test your code on the ⟨*cat-abbrev.ums* 4⟩ presented in class and in this handout on page 4, 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 > /tmp/midmark.ums
   umdump -bare midmark.um | diff - /tmp/midmark.ums
   ```

   You should see no differences. Note well that `umasm` *and* `umdump` *are not half-inverses*.[3] But `umasm` and `umdump` *are* half-inverses *on the output of* `umdump -bare`. That’s why a three-stage test is required.

5. Submit.

[3]Since this point is frequently overlooked, I 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.

6. 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.

7. Submit.

8. 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.

9. Submit. You are finished.

6 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 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.

7 What to submit

7.1 Implementation

By *Sunday, December 4 at 11:59 PM,* 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*