The Universal Machine Macro Assembler

Norman Ramsey

Fall 2011

Introduction

With only 14 instructions, the Universal Machine is a Spartan environment for even the most seasoned assembly-language programmer. The Universal Machine Macro Assembler, called `umasm`, is a front end that extends the Universal Machine to create a more usable assembly language.[1]

- There are a few more arithmetic operations. Most you will have implemented yourself, but modulus and exclusive-or are built into my front end.
- Most instructions can operate not only on registers but also on words in memory. Registers, memory words, literals, and relocatable addresses are all acceptable as operands. As an important special case, you can `goto` a label directly.
- To implement these conveniences requires "temporary" registers. An individual instruction can be given temporary registers through a `using` clause (see the grammar in Figure 1 on page 3), but it is also possible to use the `.temps` directive to set aside temporary registers. I typically use

  ```
  .temps r6, r7
  ```

  in most of my code, which means that the assembler may destroy the contents of register `r6` or `r7` at any time.

[1] In assembly-language jargon, a "macro" is something that appears to be a single instruction but actually expands to a *sequence* of machine instructions. A true macro assembler would let you, the programmer, define your own macros. Maybe next year.

- As you know, the Universal Machine has no `goto` instruction; the Load Program instruction requires a segment identifier. Since zero is the common case, it is possible to designate a register as the zero register. The obvious convention is that "register zero is always zero:"

  ```
  .zero r0
  ```

  This declaration constitutes a *promise* to the assembler; although register zero is indeed initially zero, if you overwrite it you have to put it back to zero before claiming it stays zero. The advantage is that the Macro Assembler, relying on `r0` being zero, can implement `goto` using a single Load Program instruction.

  It is also possible to turn this feature off and to use register zero as a temporary register, e.g.,

  ```
  .zero off
  .temps r0, r6, r7
  ```

  There is a minor performance penalty: to implement a goto, the Macro Assembler must now load zero into a register.

- The Macro Assembler provides a full set of six conditionals: `==`, `!=`, `<s`, `>s`, `<=s`, and `>=s`. Comparisons are signed. These conditionals may be used both in conditional move and conditional goto. *Conditionals require a lot of temporary registers*, sometimes up to four.

Notable features of the Macro Assembler

Figure 1 on page 3 gives the full language accepted by the Macro Assembler; the start symbol of the grammar is `<program>`, at the bottom. The nonterminals of major interest are `<instr>` and `<directive>`:

- Instructions may assign to any `<lvalue>` and read from any `<rvalue>`.
- The `output` instruction may write not only a register but also a character or string literal. This facility requires a temporary register.

```
 <comment> ::= from # or // to end-of-line
<reserved> ::= if | m | goto | map | segment | nand | xor | string
             |  unmap | input | output | in | program | using
             |  off | here | halt | words | push | pop | on | off | stack
   <ident> ::= identifier as in C, except <reserved> or <reg>
   <label> ::= <ident>
     <reg> ::= rNN, where NN is any decimal number
       <k> ::= <hex-literal> | <decimal-literal> | <character-literal>
  <lvalue> ::= <reg> | m[<reg>][<rvalue>]
  <rvalue> ::= <reg> | m[<reg>][<rvalue>]
             |  <k> | <label> | <label> + <k> | <label> - <k>
   <relop> ::= != | == | <s | >s | <=s | >=s
   <binop> ::= + | - | * | / | nand | & | '|' | xor | mod
    <unop> ::= - | ~
   <instr> ::= <lvalue> := <rvalue>
             |  <lvalue> := <rvalue> <binop> <rvalue>
             |  <lvalue> := <unop> <rvalue>
             |  <lvalue> := input()
             |  <lvalue> := map segment (<rvalue> words)
             |  <lvalue> := map segment (string <string-literal>)
             |  unmap m[<reg>]
             |  output <rvalue>
             |  output <string-literal>
             |  goto <rvalue> [linking <reg>]
             |  if (<rvalue> <relop> <rvalue>) goto <rvalue>
             |  if (<rvalue> <relop> <rvalue>) <lvalue> := <rvalue>
             |  push  <rvalue> on   stack <reg>
             |  pop  [<lvalue> off] stack <reg>
             |  halt
             |  goto *<reg> in program m[<reg>]
<directive> ::= .section <ident>
             |  .data <label> [(+|-) <k>]
             |  .data <k>
             |  .space <k>
             |  .string <string-literal>
             |  .zero <reg> | .zero off              // identify zero register
             |  .temps <reg> {, <reg>} | .temps off  // temporary regs
    <line> ::= {<label>:} [<instr> [using <reg> {, <reg>}] | <directive>]
 <program> ::= {<line> (<comment> | newline | ;)}
```

Figure 1: Grammar for the Universal Machine Macro Assembler

- The Macro Assembler's conditional `goto` provides equality, inequality, and signed integer comparisons on arbitrary `rvalues`. A fully general `goto` requires *four* temporary registers; thus a `using` clause will normally be required.
- The conditional move is equally flexible but may require fewer temporary registers. Beware! Except for the native conditional move, the *conditional-move instructions have not been thoroughly tested.*
- Macro instructions `push` and `pop` store and retrieve words from segment zero at the location pointed to by the given stack pointer. After a `push`, the stack pointer points to the newly pushed word; before a `pop`, the stack pointer points to the word about to be popped.

  To use the stack instructions, you must choose a register to serve as stack pointer, allocate space in segment zero for a stack, and set the stack pointer to point to the space you have allocated.
- The `.section` and `.data` directives call directly into the `Umasm_section` and `Umasm_emit_*` functions defined in your core assembler.
- The `.space` k directive emits k words of zeroes. You could use it to allocate space (at assembly time) in segment zero.
- The `.string` directive emits the characters of a string literal, one word per character, followed by a word containing all one bits.

Using the Macro Assembler

Usage of the Macro Assembler is straightforward:

```
umasm [-help] [-grammar] [-o out.um] [source.ums ...]
```

The `-help` option prints a longer explanation of options, including several options that are intended only for debugging the Macro Assembler itself. The `-grammar` option prints the input language of the Macro Assembler. The `-o` option names a file to which the binary UM code should be written; if not given, the Assembler writes to standard output.

Building the Macro Assembler

The Macro Assembler 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; without your core assembler of roughly 300–350 lines of C, it doesn't do anything. To create a Universal Machine Macro Assembler, you need to link your code against these four libraries:

```
... -lumasm -llua5.1-lpeg -llua ‘pkg-config --libs cii40‘
```

At that point you can start testing.

Testing the Macro Assembler

I have unit-tested almost all of the Macro Assembler instructions, with special emphasis on the conditional gotos.[2] My testing has found enough bugs that I am convinced there are many more bugs lurking. Ideally I 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`. 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.

[2] The new conditional moves have *not* been tested.