The Universal Machine Macro Assembler

Introduction

This document is the specification for the Universal Machine Macro Assembler Language and for the UMASM Macro Assembler program.

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]

Here are some of the features that the macro assembler adds; that is, these are capabilities that are not available from individual UM instructions, but which are available to programmers using the UMASM language. When the following features are used, the UMASM assembler emits multiple machine instructions:

- There are a few more arithmetic operations.
- Most instructions are extended so they operate not only on registers (which can be done using a single UM instruction) 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. A suggested convention is to use:

[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.

```
.temps r6, r7
```

in a given UMASM program, which means that the assembler may destroy the contents of register `r6` or `r7` at any time.

- The Universal Machine has no `goto` instruction; the Load Program instruction, which does provide a "jump" or "goto" capability, requires a segment identifier. Since zero is the common case, it is possible with the `.zero` directive to designate a register that will always hold the number zero. 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 issuing any macro call (such as `goto`) that would depend on it. The advantage of declaring a register, e.g. `r0` as zero, is that the Macro Assembler 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 are used in the `if` macro to implement a 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. Note that the `<string-literal>`, `<hex-literal>`, `<decimal-literal>` and `<character-literal>` productions are missing from the grammer. Each matches the equivalent C literal syntax: for example, `<string-literal>` accepts a C-style double-quoted string. The nonterminals of major interest are `<instr>` and `<directive>`:

```
   <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>
              |  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

- 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.
- The Macro Assembler's conditional `goto` instruction (i.e., the one using the `if` macro) provides equality, inequality, and signed integer comparisons on arbitrary `rvalues`. A fully general conditional instruction requires *four* temporary registers; thus a `using` clause will normally be required to specify which additional registers can be used as these temporaries (in addition to any specified with the `.temps` directive).
- 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; it must hold either the address in segment zero of the last item pushed, or if the stack is empty, the address of the word "off the end of" the stack. (More formally, if stack space from indices N through M is available, $M \geq N$, and the stack is empty, then the stack pointer register must be set to $M + 1$).
- The `.section` directive indicates the section into which subsequent instructions, data, and space (from the space directive) are to be emitted. This directive is effective until another `.section` directive is encountered.

  The content of each section is gathered separately and emitted continguously into the UM program. If the same sections are named repeatedly, e.g.

  ```
  .section A
     ...section A part 1...
  .section B
     ...section B part 1...
  .section A
     ...section A part 2...
  .section B
     ...section B part 2...
  ```

  then the resulting program is:

```
...section A part 1...
...section A part 2...
...section B part 1...
...section B part 2...
```

The contents (if any) of the distinguished section named "init" are emitted ahead of any other sections; other sections are then emitted in order of first mention in the umasm source.

- The `.data` directive causes one word of data to be emitted; you may label the emitted data to reference that data in later instructions. It is the programmer's responsibility to appropriately keep data and instructions separate; one way to do this is to put all data into one or more separate data sections.
- The `.space` k directive emits k words of zeroes. This facility is useful for allocating space (at assembly time) in segment zero. One common use for such space is for a call stack.
- The `.string` directive emits the characters of a string literal, one word per character, followed by a word containing all one bits.

The Macro Assembler Program

Usage of the Macro Assembler program 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.