The UM Macro Assembler (UMASM) Implementation

Noah Mendelsohn
Tufts University
Email: noah@cs.tufts.edu
Web: http://www.cs.tufts.edu/~noah
Macros
&
Building Extensible Languages
Building extensible languages

- **Software systems are uniquely extensible...this is deeply important and beautiful!**

- **Functions in higher level languages**
  - `printf, sqrt`: supplied with the system
  - `findStudent`: Functions you write or supplied in libraries *become first class features of the environment*
    - Typically compiled, or integrated by the language runtime

- **Macros for high level languages (e.g. C preprocessor)**
  - Text substitution on program source prior to compilation
  - Token substitution; sometimes with parameters
  - Some processors allow looping at compile time, etc.

- **Assembler macros**
  - Traditionally: text substitution in source (though UMASM is different)
  - Appear as pseudo-machine instructions once defined
  - Most assemblers (users can define their own macros – UMASM does not allow this)
UMASM Macros – generating multiple UM instructions from one source instruction

- Some UMASM “instructions” expand to multiple machine instructions
- The framework we provide you implements several “conveniences” that generate multiple UM instructions:
  - More powerful instructions and formats: e.g. output “A whole string”
  - Six conditionals: ==, !=, <s, >s, <=s and >=s usable in conditional move and conditional goto
  - Etc.
- There are six macro instructions for which you will write (in C) the logic to generate the needed UM instructions!
  - MOV, COM, NEG, SUB, AND, OR
Definitions of the macros you will implement

<table>
<thead>
<tr>
<th>Number</th>
<th>Operator</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>Move</td>
<td>$r[A] := r[B]</td>
</tr>
<tr>
<td>15</td>
<td>Bitwise</td>
<td>$r[A] := \neg r[B]</td>
</tr>
<tr>
<td></td>
<td>Complement</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Two’s-Complement</td>
<td>$r[A] := -r[B] mod 2^{32}</td>
</tr>
<tr>
<td></td>
<td>Negation</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Subtraction</td>
<td>$r[A] := (r[B] - r[C]) mod 2^{32}</td>
</tr>
<tr>
<td>18</td>
<td>Bitwise And</td>
<td>$r[A] := r[B] \land r[C]</td>
</tr>
<tr>
<td>19</td>
<td>Bitwise Or</td>
<td>$r[A] := r[B] \lor r[C]</td>
</tr>
</tbody>
</table>

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

\[ x \land y = \neg (\neg (x \land y)). \]
Summary: what the framework will do for you

- Parse all UMASM input
- Assemble the 14 UM machine instructions into binary, calling your `Umsections_emit_word` to output the resulting words
- Call your `Umsections_section()` when the input source switches segments
- Call you to generate code for the 6 new macro instructions (we’ll discuss this next)
- Resolve all labels, including necessary backpatching of instructions
- Call your `Umsections_write()` method to ask you to write the finished program into the output um file.

Also see the class notes linked from today’s calendar entry for more background.
Example of macro

- We cannot assemble this
  \[ r3 := r3 + 1 \]

- But:
  \[ r3 := r3 + 1 \text{ using } r5 \]

becomes

\[
\begin{align*}
  r5 & := 1 \\
  r3 & := r3 + r5
\end{align*}
\]

Tells the assembler r5 is available as a temporary for this instruction.
Permanent temporaries

• This works too:

\[ \text{.temps } r5 \]
\[ r3 := r3 + 1 \]

• becomes

\[ r5 := 1 \]
\[ r3 := r3 + r5 \]
Zero registers

• Give the assembler a zero register

\[
\text{r0} := 0 \quad \text{// can be any register} \\
\text{.zero r0} \quad \text{// tell the assembler} \\
\text{goto L5}
\]

• Can you see why having a register with the value zero will help with generating code for the goto?
How you’ll do it

The framework will call your code when it encounters a call to one if the six macros in a user’s MASM program.

/* Emit a macro instruction into ‘asm’ */
Ummacros_op(Umsections_T asm, Ummacros_Op operator,
            int temporary,
            Ummacros_Reg A, Ummacros_Reg B, Ummacros_Reg C);
How you’ll process macros

The framework will call your code whenever it encounters a call to one of the six macros in a user’s MASM program.

/* Emit a macro instruction into ‘asm’ */
Ummacros_op(Umsections_T asm, Ummacros_Op operator,
             int temporary,
             Ummacros_Reg A, Ummacros_Reg B, Ummacros_Reg C);

Enumeration:
MOV, COM, NEG, SUB, AND, OR
How you’ll process macros

The framework will call your code when it encounters
the six macros in a user’s MASM

/* Emit a macro instruction into ’asm’ */

Ummacros_op(Umsections_T asm, Ummacros_Op operator,
             int temporary,
             Ummacros_Reg A, Ummacros_Reg B, Ummacros_Reg C);
How you’ll process macros

The framework will call your code when it encounters a call to one of the six macros in a user’s MASM program.

/* Emit a macro instruction into ‘asm’ */
Ummacros_op(Umsections_T asm, Ummacros_Op operator,
int temporary,
Ummacros_Reg A, Ummacros_Reg B, Ummacros_Reg C);

Number of a temporary register that’s available for you to use...or -1 if none.
How you’ll process macros

The framework will call your code when it encounters a call to one of the six macros in a user’s MASM program.

/* Emit a macro instruction into ’asm’ */
Ummacros_op(Umsections_T asm, Ummacros_Op operator,
            int temporary,
            Ummacros_Reg A, Ummacros_Reg B, Ummacros_Reg C);

Your UMASM program will emit the necessary UM instructions into the current section of the supplied Umsections.

There’s also a separate call you implement for load literal.
Other UMASM directives
UMASM supports some other *directives*

- **.data** – emits a literal data value into current segment
  - You can assume successive `.data` values emitted in order
- **.space <n>**
  - Gives you n words of zeros

These are implemented for you by the framework
Numbers

- What does the UM specification say about negative or signed numbers?

- The UMASM specification clearly requires a convention for storing negative numbers

- What to do?

- Go back to earlier work on two’s complement. Ask yourself: if we use two’s complement representations, how do we need to change operations like ADD that are already included with the UM?

Be sure you understand why no changes are needed!
(this will likely be on the final exam)
Segments in UM Programs
Section content can be interleaved in UMASM source

These two umasm source files assemble to produce *identical* um object files!
Section content can be interleaved in UMASM source

These two umasm source files assemble to produce *identical* um object files!
Section content can be interleaved in UMASM source

```
.section init
.temp  r7
.output 'E'
r3 := L1
r0 := 0
output m[r0][r3]
output m[r0][r3+1]
halt

.section data
L1: .data 'X'
   .data '

.section init
.temp  r7
.output 'E'
r3 := L1
r0 := 0;
.section data
L1: .data 'X'
   .data '\n'
.section init
r0 := 0
output m[r0][r3]
.output m[r0][r3+1]
halt
```

These two umasm source files assemble to produce identical um object files!
UMASM Segments (called sections)

- UMASM programmers can use the `.section` directive to cause code or data to be put into separate segments

The assembler collects the contents of each section separately before emitting anything

- The section named `.init` is special: the assembler will always emit it first...so code placed there will be run when your program starts

- Standard named sections for UMASM:
  - `'init'` (code that runs to set up for procedure calls, main code)
  - `'text'` (machine code)
  - `'data'` (global variables)
  - `'stk'` (call stack)

- Writing the code to separately gather the contents of these sections and emit them will be one of your main tasks in the UMASM assignment!
Writing the code to assemble segments

- What data will you need to retain as the assembler runs?
- What data structures will you use?

Questions to answer from the above:
- What named sections will contribute output to the .um files you produce?
- In what order will the content of those sections appear in the .um file?
- How will you update your list of sections as umasm source is processed?
- How will you collect the contents of each section?
- What Hanson or other ADTs will you use?

- What invariants will hold after each source instruction is assembled?
The interface you implement

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

/*
 * make the named section current, creating if needed
 * (modifies state of asm)
 */
void Umsections_section(T asm, const char *section);

/*
 * Emit a word into the current section
 */
void Umsections Emit_word(T asm, Umsections_word data);
The interface you implement

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

/**
 * make the named section current, creating if needed
 * (modifies state of asm)
 */
void Umsections_section(T asm, const char *section);

/**
 * Emit a word into the current section
 */
void Umsections_emit_word(T asm, Umsections_word data);
The interface you implement

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

/*
 * make the named section current, creating if needed
 * (modifies state of asm)
 */
void Umsections_section(T asm, const char *section);

/*
 * Emit a word into the current section
 */
void Umsections_emit_word(T asm, Umsections_word data);
Sections in Linux Programs
The process memory illusion (REVIEW)

- Process thinks it's running in a private space
- Separated into segments, from address 0
- Stack: memory for executing subroutines
- Heap: memory for malloc/new
- Global static variables
- Text segment: where program lives

Diagram:
- Stack
- Heap (malloc'd)
- Static uninitialzed
- Static initialized
- Text (code)

Loaded with your program
The process memory illusion

```c
char notInitialized[10000];
char initialized[] = "I love COMP 40";

int main(int argc, char *argvp[]) {
    float f;
    int i;
    // yes, we should check return codes
    char *cp = malloc(10000);
}
```
The process memory illusion

```c
char notInitialized[10000];
char initialized[] = "I love COMP 40";

int main(int argc, char *argvp[]) {
    float f;
    int i;

    // yes, we should check return codes
    char *cp = malloc(10000);
}
```

Loaded with your program
The process memory illusion

```
char notInitialized[10000];
char initialized[] = "I love COMP 40";

int main(int argc, char *argvp[]) {
    float f;
    int i;

    // yes, we should check return codes
    char *cp = malloc(10000);
}
```

Program file tells how much is needed

Loaded with your program
The process memory illusion

```c
char notInitialized[10000];
char initialized[] = "I love COMP 40";

int main(int argc, char *argv[]) {
    float f;
    int i;

    // yes, we should check return codes
    char *cp = malloc(10000);
}
```

Loaded with your program
The process memory illusion

```c
char notInitialized[10000];
char initialized[] = "I love COMP 111";

int main(int argc, char *argvp[]) {
    float f;
    int i;

    // yes, we should check return codes
    char *cp = malloc(10000);
}
```

Loaded with your program

- Stack
- Heap (malloc'd)
- Static uninitialized
- Static initialized
- Text (code)
Of course, the stack enables recursion

```c
int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

Loaded with your program
Of course, the stack enables recursion

```c
int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

Loaded with your program
Individual source files compile to multiple segments

```
#include <stdlib.h>

int main()
{
    puts("Hello, world\n");
    return EXIT_SUCCESS;
}
```

```
$ objdump -s hello.o

hello.o:   file format elf64-x86-64

Contents of section .text:
  0000 554889e5 bf000000 00b80000 0000e800 UH.............
  0010 000000b8 00000000 c9c3 ................

Contents of section .rodata:
  0000 48656c6c 6f2c2077 6f726c64 0a00 Hello, world..

Contents of section .eh_frame:
  0000 14000000 00000000 017a5200 01781001 ...........zR..x..
  0010 030c0708 90010000 1c000000 1c000000 ............A...C.
  0020 00000000 1a000000 00410e10 8602430d ............
  0030 06000000 00000000 ............

Contents of section .comment:
  0000 00474343 202847 4e552920 342e312e GCC: (GNU) 4.1.
  0010 32203230 30383037 30342028 52656420 2 20080704 (Red
  0020 48617420 342e312e 322d3534 2900 Hat 4.1.2-54).

$
Individual source files compile to multiple segments

```c
#include <stdlib.h>

int main()
{
    puts("Hello, world\n");
    return EXIT_SUCCESS;
}
```

```
$ objdump -s hello.o

hello.o:     file format elf64-x86-64

Contents of section .text:
0000 554889e5 bf000000 00b80000 0000e800  UH.............
0010 000000b8 00000000 c9c3  ................

Contents of section .rodata:
0000 48656c6c 6f2c2077 6f726c64 0a00  Hello, world..

Contents of section .eh_frame:
0000 14000000 00000000 017a5200 01781001  ............zR..x..
0010 030c070c 90010000 1c000000 1c000000  ................
0020 00000000 1a000000 00410e10 8602430d  ........A....C.
0030 06000000 00000000  .................

Contents of section .comment:
0000 0047434c 6a202847 4e552920 342e312e  .GCC: (GNU) 4.1.
0010 32203230 30383037 30342028 52656420 2 20080704 (Red
0020 48617420 342e312e 322d3534 2900  Hat 4.1.2-54).
$
Individual source files compile to multiple segments

```c
#include <stdlib.h>

int main()
{
    puts("Hello, world\n");
    return EXIT_SUCCESS;
}
```

```
$ objdump -s hello.o

hello.o:     file format elf64-x86-64

Contents of section .text:
    0000 554889e5 bf000000 00b80000 0000e800   UH............
    0010 000000b8 00000000 c9c3               ...........

Contents of section .rodata:
    0000 48656c6c 6f2c2077 6f726c64 0a00      Hello, world..

Contents of section .eh_frame:
    0000 14000000 00000000 017a5200 01781001   ........zR...x..
    0010 030c0708 90010000 1c000000 1c000000   ................
    0020 00000000 1a000000 00410e10 8602430d   ........A....C.
    0030 06000000 00000000                    ........

Contents of section .comment:
    0000 00474343 2028474e 552920342e312e3220303830373034202852656420202008070420 (Red
    0010 322032330 303830373034202852656420202008070420 (Red
    0020 48617420 342e312e322d35342900  Hat 4.1.2-54).
$ 
```
Individual source files compile to multiple segments

```c
#include <stdlib.h>

int main()
{
    puts("Hello, world\n");
    return EXIT_SUCCESS;
}
```

$ objdump -s hello.o

hello.o: file format elf64-x86-64

Contents of section .text:

```
0000 554889e5 bf000000 00b80000 0000e800 UH...........
0010 000000b8 00000000 c9c3
```

Contents of section .rodata:

```
0000 48656c6c 6f2c2077 6f726c64 0a00 Hello, world..
```

Contents of section .eh_frame:

```
0000 14000000 00000000 017a5200 01781001 ............zR..x.
0110 030c0708 90010000 1c000000 1c000000
0200 00000000 1a000000 00410e10 8602430d ..........A...C.
0300 00600000 00000000
```

Contents of section .comment:

```
0000 00474343 202847 4e552920 342e312e GCC: (GNU) 4.1.
0010 32203230 30383037 30342028 52656420 2 20080704
0020 48657420 342e31e 1f32d3534 2900 Hat 4.1.2-54).
```
Individual source files compile to multiple segments

```c
#include <stdlib.h>

int main()
{
    puts("Hello, world\n");
    return EXIT_SUCCESS;
}
```

```
$ objdump -s hello.o
hello.o: file format elf64-x86-64

Contents of section .text:
  0000 554889e5 bf000000 00b80000 0000e800 UH.............
  0010 000000b8 00000000 c9c3 ............

Contents of section .rodata:
  0000 48656c6c 6f2c2077 6f726c64 0a00 Hello, world..

Contents of section .eh_frame:
  0000 14000000 00000000 017a5200 01781001 ............zR..x..
  0010 03c0709 90010000 1c000000 1c000000 ............
  0020 00000000 1a000000 00410e10 8602430d ........A...C.
  0030 06000000 00000000 ........

Contents of section .comment:
  0000 00474343 3a202847 4e552920 342e312e .GCC: (GNU) 4.1.
  0010 00723230 30383037 30342028 52656420  2 20080704 (Red
  0020 48617420 342e312e 322d3534 2900 Hat 4.1.2-54).
```

Used for C++ exception handling but shows up .o files for .c programs anyway
What you do in UMASM is a lot like what the linker does!

```c
#include <stdlib.h>

int main()
{
    func();
    puts("I called it");
    return EXIT_SUCCESS;
}
```

callfunc.c

```c
void func()
{
    puts("I've been called!");
    return;
}
```

implfunc.c
What you do in UMASM is a lot like what the linker does!

```bash
$ objdump -s callfunc.o

callfunc.o:    file format elf64-x86-64

Contents of section .text:
0000 554889e5 b8000000 00e80000 0000bf00 UN.............
0010 000000e8 00000000 b8000000 00c9c3 .................

Contents of section .rodata:
0000 49206361 6c6c6564 20697421 00 I called it!.

Contents of section .eh_frame:
0000 14000000 00000000 017a5200 01781001 .........zR..x..
0010 030c0708 90010000 1c000000 1c000000 ..........A....C.
0020 00000000 00410e10 8602430d .........A....C.
0030 065a0c07 08000000 .Z.....

Contents of section .comment:
0000 00474343 3a202847 4e552920 342e352e .GCC: (GNU) 4.5.
0010 3200 2. $

0010 3200

$ objdump -s impfunc.o

implfunc.o:    file format elf64-x86-64

Contents of section .text:
0000 554889e5 bf000000 00e80000 008c9c3  UN.............

Contents of section .rodata:
0000 6c6c6564 20697421 00 I've been called
0010 2100  !.

Contents of section .eh_frame:
0000 14000000 00000000 017a5200 01781001 .........zR..x..
0010 030c0708 90010000 1c000000 1c000000 ..........A....C.
0020 00000000 00410e10 8602430d .........A....C.
0030 064b0c07 08000000 .K.....

Contents of section .comment:
0000 00474343 3a202847 4e552920 342e352e .GCC: (GNU) 4.5.
0010 3200 2.$

gcc –o callfunc callfunc.o implfunc.o

$ objdump -s callfunc

callfunc:    file format elf64-x86-64

[...]

Contents of section .rodata:
4005c8 01000200 49206361 6c6c6564 20697421 ....I called it!
4005d8 00492776 65206265 6e206365 616c6c65 .I've been called!
4005e8 642100

4005e8 642100
```


What you do in UMASM is a lot like what the linker does!

As you can see, what you’re learning to do an assembling sections from
declarations taken out of order...
...is exactly what the gcc linker (named ld) does when combining .o files!

```
gcc -o callfunc callfunc.o implfunc.o
```

```
$ objdump -s callfunc

callfunc:     file format elf64-x86-64

[...] Contents of section .rodata:
4005c8 01000200 49206361 6c6c6564 20697421 6c6c6564 20697421 ....I called it! 6c6c6564 20697421 6c6c6564 20697421 .I've been called!
4005d8 00492776 65206265 6e206361 6c6c6564 20697421 6c6c6564 20697421 ....I called it! 6c6c6564 20697421 6c6c6564 20697421 .I've been called!
4005e8 642100
```
Summary

What we’re learning from building UMASM
Lessons from UMASM

- Assemblers generate machine code files from text files
- **Some things assemblers do:**
  - Support symbolic representations of opcodes and operands
  - Support symbolic names for locations (labels)
- **Provide macros: one source instruction represents multiple machine ops**
  - Many assemblers let users define their own macros (think #define)
  - In UMASM, only the C code of the assembler itself can implement them
- **Separately named sections**
  - Allows single pass assembly of data mixed with code
  - Allows operating systems to share read-only segments/sections among users
  - *Real world:*
    - allows multiple source files, compiled to multiple .o files, to contribute to code, data and other segments.
- **Macros and functions blur the distinction between what the machine supports directly and what we “teach” it to do in software**