The UM Macro Assembler (UMASM)  
What’s Going on Inside?

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

output "Hello\n"

Generating multiple UM instructions from one source instruction
Temporary scratch registers

- We cannot assemble this
  \[
  r3 := r3 + 1
  \]
- But:
  \[
  r3 := r3 + 1 \text{ using } r5
  \]

becomes

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

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

• This works too:

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

• becomes

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

Tells the assembler r5 is available as a temporary from now on
Zero registers

• Give the assembler a zero register

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

• Can you see why having a register with the value zero will help with generating code for the goto?
Zero registers

- Give the assembler a zero register

  \[
  r0 := 0  \quad \text{// can be any register}
\]

  \[
  .zero \ r0  \quad \text{// tell the assembler}
\]

  \[
  \text{goto L5}
\]

- Can you see why having a register with the value zero will help with generating code for the goto?

Assure the assembler that r0 will contain zero

Does not set r0 to 0
UMASM Macros – generating multiple UM instructions from one source instruction

- As we’ve seen, some UMASM “instructions” expand to multiple machine instructions

- UMASM implements several “conveniences” that generate multiple UM instructions:
  - More powerful instructions and formats: e.g. output “A whole string”
  - Six conditionals: ==, !=, <, >, <=, and >= usable in conditional move and conditional goto
  - Etc.

- To see how UMASM does this, there are six macro instructions for which you will write (in lab) the logic using a different assembler 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>$r[A] := r[B]</td>
</tr>
<tr>
<td>15</td>
<td>Bitwise Complement</td>
<td>$r[A] := \lnot r[B]</td>
</tr>
<tr>
<td>16</td>
<td>Two’s-Complement</td>
<td>$r[A] := -r[B] mod 2^{32}</td>
</tr>
<tr>
<td>17</td>
<td>Subtraction</td>
<td>$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:

\[ x \land y = \lnot(\lnot(x \land y)). \]
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
Negative Numbers
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)
Sections in UM Programs
Section content can be interleaved in UMASM source

These two umasm source files assemble to produce identical um object files!
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!
UMASM Sections *(these are not segments)*

- 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 would you need to retain as the assembler runs?
- What data structures would 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 would you update a list of sections as umasm source is processed?
  - How would you collect the contents of each section?
  - What Hanson or other ADTs would you use to implement these UMASM features?
- *What invariants will hold after each source instruction is assembled?*
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

Loaded with your program

Stack

Heap (malloc'd)

Static uninitialized

Static initialized

Text (code)
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

```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

- **Stack**
- **Heap** (malloc'd)
- **Static uninitialized**
- **Static initialized**
- **Text (code)**
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);
}
```

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

- **Stack**: Text (code)
- **Heap**: Static uninitialized
- **Stack**: Static initialized
- **Stack**: argv, environ

`7fffffffffff`
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);
}
```
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 C 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 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 030c0708 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 32203230 30383037 30342028 52655642 2 20080704 (Red
0020 48617420 342e312e 322d3534 2900 Hat 4.1.2-54).
```

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 ............
  0020 00000000 1a000000 00410e10 8602430d ............A....C.
  0030 06000000 00000000 ........

Contents of section .comment:
  0000 00474343 3a202847 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"e8000000 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  .................
    0000 00000000 1a000000 00410e10 8602430d  ..........A....C.
    0030 00000000 00000000                      ........
Contents of section .comment:
    0000 00474343 3a202847 4e552920 342e312e .GCC: (GNU) 4.1.
    0010 32203230 30383037 30342028 52656420 2 20080704 (Red
    0020 48617420 342e312e 322d3534 2900  Hat 4.1.2-54).

$ 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  .................
    0000 00000000 1a000000 00410e10 8602430d  ..........A....C.
    0030 00000000 00000000                      ........
Contents of section .comment:
    0000 00474343 3a202847 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 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).
```

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

```bash
$ objdump -s callfunc.o
callfunc.o:    file format elf64-x86-64
Contents of section .text:
0000 554889e5 b8000000 00e80000 0000bf00 UH............
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 .............
0020 00000000 12000000 00410e10 8602430d ........A...C.
0030 065a0c07 08000000 .............
0040 00000000 00000000 017a5200 01781001 ........zR..x..
0050 030c0708 90010000 1c000000 1c000000 .............
0060 00000000 12000000 00410e10 8602430d ........A...C.
0070 064b0c07 08000000 .............
Contents of section .comment:
0000 00474343 3a202847 4e552920 342e352e .GCC: (GNU) 4.5.
0010 3200                    2. $  
```

implfunc.c

```bash
$ objdump -s implfunc.o
implfunc.o:    file format elf64-x86-64
Contents of section .text:
0000 554889e5 bf000000 00e80000 0000c9c3 UH............
Contents of section .rodata:
0000 49277665 20626566 6e206361 6c6c6564 I've been called  
0010 0000 49277665 20626566 6e206361 6c6c6564 I've been called  

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

$ objdump -s callfunc.o

callfunc.o:   file format elf64-x86-64

Contents of section .text:
  0000 554889e5 b8000000 00e80000 0000bf00  UH..............
  0010 00000008 00000000 b8000000 00c9c3  ...............1

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 03c0708 90010000 1c000000 1c000000 .............
  0020 00000000 1f000000 00410e10 8602430d .......A....C.
  0030 065a0c07 08000000 .Z......

Contents of section .comment:
  0000 00474343 3a202847 4e552920 342e352e GCC: (GNU) 4.5.
  0010 00000000 00000000 00000000 00000000 .
  0010 030c0708 90010000 1c000000 1c000000 .........
  0020 00000000 10000000 00410e10 8602430d .......A....C.
  0030 064b0c07 08000000 .K....

Contents of section .comment:
  0000 00474343 3a202847 4e552920 342e352e GCC: (GNU) 4.5.
  0010 00000000 00000000 00000000 00000000 .
  0010 03c0708 90010000 1c000000 1c000000 .........
  0020 00000000 10000000 00410e10 8602430d .......A....C.

$ objdump -s impfunc.o

implfunc.o:   file format elf64-x86-64

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

Contents of section .rodata:
  0000 49277665 2062656e 63616c6c65 2100 I've been called

Contents of section .eh_frame:
  0000 14000000 00000000 017a5200 01781001 .........zR..x.
  0010 03c0708 90010000 1c000000 1c000000 .............
  0020 00000000 1f000000 00410e10 8602430d .......A....C.
  0030 064b0c07 08000000 .K....

Contents of section .comment:
  0000 00474343 3a202847 4e552920 342e352e GCC: (GNU) 4.5.
  0010 00000000 00000000 00000000 00000000 .
  0010 03c0708 90010000 1c000000 1c000000 .........
  0020 00000000 10000000 00410e10 8602430d .......A....C.

$ 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 6e63616c6c65 6c6c6564 20697421 ....I called it!
  4005e8 642100 .I've been called!
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 206971421 ....I called it!
4005d8 00492776 65206265 6d656e2063 6c6c65 .I've been called!
4005e8 642100
Summary

What we’re learning from thinking about building an assembler
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