Assemblers, Macros and The UM Macro Assembler (UMASM)

Noah Mendelsohn
Tufts University
Email: noah@cs.tufts.edu
Web: http://www.cs.tufts.edu/~noah
COMP 40 Roadmap

- Ramp up your Programming Skills
- Big programs that teach you abstraction, pointers, locality, machine representations of data
- Intel Assembler Programming
  *The Bomb!*
- Building a Language Processor on your Emulator
- Building Useful Applications in your Language
- Emulating your own hardware in software
Assemblers
Macros
&
Higher Level Languages
Computer languages

- **High level:**
  - Specify the computation or result independent of machine details
  - On statement often translates to many machine instructions...and program as a whole is considered together

- **Assembler language:**
  - Mostly: one statement $\rightarrow$ one machine instruction
  - Exceptions: data declarations, label references, etc

- **Macros:**
  - Directly substituted in place
  - Usually: simple transformation to a few assembler instructions
  - Usually: no looping, etc.
  - The C preprocessor in some ways resembles a macro processor
Grammars
Specifying the syntax of a language

- Goal: distinguish syntactically correct programs from other text strings
- Approach: use a formal specification language for the syntax
- BNF (Backus Normal Form or Backus Naur Form)
  - First developed for ALGOL 58
  - Context-free languages (from the Chomsky hierarchy)
  - E.g. more powerful than regular expressions but harder to match
  - BNF is the spiritual antecedent of most of the grammars used to specify modern programming languages:

Rough example:

```
fish ::= Tuna | Salmon
veggie ::= Broccoli | Carrots
dinner ::= <fish> and <veggie> [plus <veggie>]
```

Matches:
- Tuna and Carrots
- Salmon and Carrots
- Tuna and Broccoli plus Carrots
- Tuna and Broccoli plus Broccoli
Part of the UMASM Syntax

<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>]
[line] ::= {<label>:} [<instr> [using <reg> {, <reg>}] | <directive>]

Matches (for example):

r3 := 5
r6 := r3 + 12546
Part of the UMASM Syntax

\[
\begin{align*}
\text{<ident>} & ::= \text{identifier as in C, except <reserved> or <reg>} \\
\text{<label>} & ::= \text{<ident>} \\
\text{<reg>} & ::= rNN, \text{ where NN is any decimal number} \\
\text{<k>} & ::= \text{<hex-literal> | <decimal-literal> | <character-literal>} \\
\text{<lvalue>} & ::= \text{<reg> | m[<reg>][<rvalue>]} \\
\text{<rvalue>} & ::= \text{<reg> | m[<reg>][<rvalue>] | <k> | <label> | <label> + <k> | <label> - <k>} \\
\text{<relop>} & ::= != | == | <s | >s | <=s | >=s \\
\text{<binop>} & ::= + | - | * | / | \text{nand} | \& | \text{'} | \text{xor} | \text{mod} \\
\text{<unop>} & ::= - | ~ \\
\text{<instr>} & ::= \text{<lvalue> := <rvalue>} \\
& | \text{<lvalue> := <rvalue> <binop> <rvalue>} \\
& | \text{<lvalue> := <unop> <rvalue>} \\
& | \text{<lvalue> := input()} \\
& | \text{<lvalue> := map segment (<rvalue> words)} \\
& | \text{<lvalue> := map segment (string <string-literal>)} \\
& | \text{unmap m[<reg>]}
\end{align*}
\]

Matches (for example):

\[
\begin{align*}
r3 & := 5 \\
r6 & := r3 + 12546
\end{align*}
\]
Part of the UMASM Syntax

<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>]
[...]
(line) ::= {<label>:} [<instr> [using <reg> {, <reg>}] | <directive>]

Matches (for example):

\[
\begin{align*}
  r3 & := 5 \\
  r6 & := r3 + 12546
\end{align*}
\]
Macros
Macros

- Many UMASM commands translate to a single UM instruction:
  - `r4 := 'X'` → `[0xd8000058] r4 := 88;`
  - `output r4` → `[0xa0000004] output r4;`
  - `halt` → `[0x70000000] halt;`
Macros

- Many UMASM commands translate to a single UM instruction:
  
  ```
  r4 := 'X'
  output r4
  halt
  ```

  ```
  [0xd8000058]  r4 := 88;
  [0xa0000004]  output r4;
  [0x70000000]  halt;
  ```

- Many are macros:
  
  ```
  .temp  r4
  output "Hello\n"
  halt
  ```

  ```
  [0xd8000048]  r4 := 72;
  [0xa0000004]  output r4;
  [0xd8000065]  r4 := 101;
  [0xa0000004]  output r4;
  [0xd800006c]  r4 := 108;
  [0xa0000004]  output r4;
  [0xd800006c]  r4 := 108;
  [0xa0000004]  output r4;
  [0xd800006f]  r4 := 111;
  [0xa0000004]  output r4;
  [0x70000000]  halt;
  ```
Macros

- Many UMASM commands translate to a single UM instruction:
  - `r4 := 'X'`
  - `output r4`
  - `halt`

- Many are macros:
  - `.temp r4`
  - `output "Hello\n"`
  - `halt`

```
[0xd8000048]  r4 := 72;
[0xa0000004]  output r4;
[0xd8000065]  r4 := 101;
[0xa0000004]  output r4;
[0xd800006c]  r4 := 108;
[0xa0000004]  output r4;
[0xd800006c]  r4 := 108;
[0xa0000004]  output r4;
[0xd800006f]  r4 := 111;
[0xa0000004]  output r4;
[0x70000000]  halt;
```

.temp directive assures assembler r4 is free for its use as a temporary...your program promises not to care if it gets changed.
Sections
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 of memory segments]

Loaded with your program

Stack

Heap (malloc'd)

Static
uninitialized

Static
initialized

Text (code)

argv, environ

7fffffffffff
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);
}
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

```
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);
}
```

Program file tells how much is needed

Loaded with your program
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[10] = "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
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

- **Stack**
  - `n=4`
  - `n=3`
  - `n=2`
  - `n=1`

- **Heap (malloc’d)**
- **Static uninitialized**
- **Static initialized**
- **Text (code)**

- `argv, environ` - 7fffffffffffffff
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

```
7fffffffffff
```

- Stack
  - n=4
  - n=3
  - n=2
- Heap (malloc'd)
- Static uninitialized
- Static initialized
- Text (code)
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!**