The Programming the UM Macro Assembler

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 Useful Applications in your Language
- Building a Language Processor on your Emulator
- Emulating your own hardware in software
Today

- General techniques for organizing a program using UMASM
- Creating a stack
- Creating and calling functions
Program
Startup
How do C programs get started anyway?

- Some of the work is done by Linux before the process starts, but...
- ...there’s actually a lot done by the “C Runtime” code *linked with your program*, before your main program is called

**Things startup routines do**
- Set up the stack and frame pointers
- Set up for exception handling
- Call constructors to initialize data (for languages like C++)
- *Call the user’s main program*

**Different languages or even different compilers can use different startup routines**

Now we’ll explore typical startup code for UMASM programs
Suggested startup code for UMASM programs

- **Goals:**
  - Set up useful registers
  - Often: set up a stack
  - Let UMASM know about register assignments (.zero and .temps)
  - Invoke or drop through to user code
Simple startup code with no stack

**Goals:**
- Set up useful registers
- Often: set up a stack
- Let UMASM know about register assignments (.zero and .temps)
- Invoke or drop through to user code

```
.section text
.temps r6, r7
.zero r0
start:
  r0 := 0
  goto main linking r1
halt
```

Simple version with no stack
Simple startup code with no stack

- **Goals:**
  - Set up useful registers
  - Often: set up a stack
  - Let UMASM know about register assignments (.zero and .temps)
  - Invoke or drop through to user code

```assembly
.section text
.temps r6, r7
.zer o r0
.start:
  r0 := 0
  goto main linking r1
.halt
```

You could choose other temp or zero registers... 
...but these conventions are common

Simple version with no stack
Simple startup code with no stack

- Set up useful registers
- Often: set up a stack
- Let UMASM know about register assignments
- Invoke or drop through to user code

This allows you to write your program as a separate “main” function, as you do in C. (optional)

Simple version with no stack

```
.section init
.temps r6, r7
.zero r0
start:
r0 := 0
  goto main linking r1
halt
```
Simple startup code with no stack

- Set up useful registers
- Often: set up a stack
- Let UMASM know about register assignments (.zero and .temps)
- Invoke or drop through to user code

We’ll learn later about “goto function linking r1”

Simple version with no stack

```assembly
.section text
.temps r6, r7
.zero r0
start:
  r0 := 0
  goto main linking r1
halt
```


### Startup code with stack

- **Goals:**
  - Set up useful registers
  - Often: set up a stack
  - Let UMASM know about register assignments
  - Invoke or drop through to user code

```
.section init
.temps r6, r7
.section stk
.space 100000
endstack:
.section init
.zero r0
start:
  r0 := 0
  r2 := endstack
  goto main linking r1
halt
```

Richer version *with stack*
## Startup code with stack

### Goals:
- Set up useful registers
- Often: set up a stack
- Let UMASM know about register assignments
- Invoke or drop through to user code

```assembly
.section init
.temps r6, r7
.section stk
.space 100000
endstack:
.section init
.zero r0
start:
  r0 := 0
  r2 := endstack
  goto main linking r1
halt
```

Richer version with stack
**Goals:**
- Set up useful registers
- Often: set up a stack
- Let UMASM know about register assignments
- Invoke or drop through to user code

```assembly
.routine init
    .temps r6, r7
.routine stk
    .space 100000
.endstack:
.routine init
    .zero r0
.start:
    r0 := 0
    r2 := endstack
    goto main linking r1
.halt
```

Richer version *with stack*
Goals:
- Set up useful registers
- Often: set up a stack
- Let UMASM know about register assignments
- Invoke or drop through to user code

Our UM has very few registers...

...putting code and stack in the same segment allows both to be addressed with a single register (.zero r0)

Note, the stack is in segment zero with the program. Why?
Review:
Calling Functions in AMD 64
Why have a standard **AMD64** “linkage” for calling functions?

- Functions are compiled separately and linked together
- We need to standardize enough that function calls can be freely mixed
- We may “cheat” when caller and callee are in the same source file
Linux/AMD 64 function linkage details

- Caller “pushes” return address on stack
- Where practical, arguments passed in registers
- Exceptions:
  - Structs, etc.
  - Too many
  - What can’t be passed in registers is at known offsets from stack pointer!
- Return values
  - In register, typically %rax for integers and pointers
  - Exception: structures
- Each function gets a stack frame
  - Leaf functions that make no calls may skip setting one up
- Caller vs. callee saved registers
Before call

Arguments

???

%rsp

After callq

Arguments

Return address

%rsp

If callee needs frame

Arguments

Return address

Callee vars

Args to next call?

framesize

sub $0x{framesize},%rsp
 Arguments and return values passed in registers when types are suitable and when there aren’t too many

Return values usually in `%rax, %eax, etc.

Callee may change these and some other registers!

MMX and FP 87 registers used for floating point

Read the specifications for full details!
int fact(int n) {
    if (n == 0)
        return 1;
    else
        return n * fact(n - 1);
}

AMD 64 Factorial Revisited

fact:
.LFB2:
pushq  %rbx
.LCFI0:
    movq   %rdi, %rbx
    movl   $1, %eax
testq  %rdi, %rdi
    je .L4
    leaq   -1(%rdi), %rdi
call   fact
    imulq  %rbx, %rax
.L4:
popq   %rbx
ret
Introduction to UMASM
Functions

More details next time
Why have a standard **UMASM** function linkage?

- **Functions are compiled separately and linked together**
- **We need to standardize enough that function calls can be freely mixed**
- **It’s tricky, so doing it the same way every time minimizes bugs and makes code clearer**
- **We may “cheat” when caller and callee are in the same source file**
What do we need to know about calling conventions?

- Which registers hold parameters? [none — use the stack]
- Which registers hold results? [r1]
- Which registers are preserved across calls? [r0, r3, r4]
- Which register is the stack pointer? [r2]
- When we call a procedure, where is the return address? [in r1]
- What is the layout of the stack frame? [r2 points to arguments]
Summarizing register conventions

- r0 is always 0 (points to segment 0, used for goto and for the call stack)
- r1 is the return-address register \textit{and} the result register
- r2 is the stack offset (into segment 0, which is addressed by r0)
- r3, r4 are nonvolatiles
- r5 is a helpful volatile register (good to copy return address)
- r6, r7 are volatile and are normally .temps (but up to each procedure)
Calling a function with no arguments

call somefunc();
Calling a function with no arguments

```
r1 := return_addr  // Offset in seg 0
r7 := somefunc     // Offset in seg 0
goto *r7 in Program m[r0] // Call the function

return_addr:
    ...back in caller...

Use of r1 for return address is required
```
Calling a function with no arguments

```plaintext
r1 := return_addr  // Offset in seg 0
r7 := somefunc     // Offset in seg 0

goto *r7 in program m[r0] // Call the function

return_addr:         // Function returns here

...back in caller...

call somefunc();
```

Use of r7 for function address is programmer choice
Calling a function with no arguments

r1 := return_addr // Offset in seg 0
r7 := somefunc // Offset in seg 0
goto *r7 in program m[r0] // Call the function
return_addr: // Function returns here
...back in caller...

call somefunc();

goto somefunc linking r1 // goto main, setting r1 to the return address

UMASM provides this built-in macro to do the above for you!
Calling a function with no arguments

```
r1 := return_addr // Offset in seg 0
r7 := somefunc // Offset in seg 0
goto *r7 in program m[r0] // Call the function
return_addr: // Function returns here
...back in caller...
```

call somefunc();

goto somefunc linking r1 // goto main, setting r1 to the return address

UMASM provides this built-in macro to do the above for you!

Also remember: r1 is both the linking and the result register. If `somefunc` calls another function, then r1 must be saved.
UMASM
Function Arguments
and the
Call Stack
Why we need a stack

- **A place to store data that doesn’t fit in registers**
  - This machine has very few registers!
  - Local variables for functions
  - Return address stack for non-leaf procedures
  - Save callee-save registers needed in the function

- **Function arguments**
UMASM function arguments

- At entry to a function, the stack must look like this:
  
  \[
  m[r0][r2 + 0] \ comment \ first \ parameter \\
  m[r0][r2 + 1] \ comment \ second \ parameter \\
  m[r0][r2 + 2] \ comment \ third \ parameter
  \]

- The called function may push more data on the stack, which will change \( r2 \)...

- ...when this happens the parameters will appear to be at larger offsets from the (modified) \( r2 \)!
Pushing and popping

- **Push macro**

  ```
  .zero r0
  .temps r6, r7
  push r3 on stack r2
  ```

  Expands to:

  ```
  r6 := -1 // macro instruction (not shown)
  r2 := r2 + r6
  m[r0][r2] := r3
  ```

- **Pop macro**

  ```
  pop r3 off stack r2
  ```

  Expands to:

  ```
  r3 := m[r0][r2]
  r6 := 1
  r2 := r2 + r6
  ```

These instructions can be used to save and restore registers and to put function arguments on the stack!
Summary

- Programming the UM presents interesting and educational challenges:
  - Limited registers
  - Simple instructions
  - These allow us to illustrate with small programs the challenges faced by programmers & compilers building bigger programs on machines like AMD 64

- Like the C language, we need startup code to enable a reasonable programming environment before each program runs
  - Create a stack, establish base registers, etc.

- Both Linux and our UM environment benefit from a standardized function calling sequence – allows mixing and matching of code from different programmers, tools, etc.

Next time: a deeper dive into UMASM programming with functions