

Programming the Universal Machine

Norman Ramsey

Fall 2008

Introduction

For the assembly-language programmer, the most salient facts about the Universal Machine are these:

- The machine does not have many registers.
- Extra registers are needed to implement even the simplest operations, like comparing registers, or subtracting.
- The only way to implement goto is with a Load Program instruction, which requires a register containing zero to indicate the program being loaded.

With these limitations in mind, I have recommended some best practices for programming the Universal Machine. The most important of these practices is a convention for implementing procedure calls, known here as the UM Stack Convention.

A note on efficiency

With only 13 instructions and 8 registers, the Universal Machine is can be somewhat difficult to program. *You should not try to make your assembly code efficient:*

- It is nearly impossible to manage the registers efficiently by hand.

- In a realistic situation, you would create efficient assembly code by having a compiler do part of the work.

Instead, write code that is clear and correct, with no regard for efficiency.

The stack

The UM Stack convention uses a stack just like the AMD64. Register 2 is reserved for use as a stack pointer, and stack space is allocated in array zero by the following code, found in file `stack.ums`: This code also initializes the stack pointer to point to the old end of the stack. The stack grows downward, toward smaller addresses. Most assembly-language programmers will take their chances with stack overflow.

Register usage calls and returns

At the start of a procedure,

- Register `r0` contains zero.
- Register `r1` contains the return address—where control should go when the function has ended.
- Register `r2` is the stack pointer. Memory locations `a[r2-1]`, `a[r2-2]`, and so on are available for use by the procedure.
- Any arguments are on the stack at locations `a[r2]` (the first argument), `a[r2+1]`, `a[r2+2]`, and so on.
- Registers `r3` and `r4` contain values that belong to the caller—if these registers are used, the values must be saved and restored.
- Registers `r5` to `r7` are “scratch registers.” The procedure may use them any way it likes.

At the end of a procedure,

- Register `r0` contains zero.

- Register r1 contains the value returned by the procedure, if any.
- Register r2, the stack pointer, has exactly the same value as it had on entry to the procedure.
- Registers r3 and r4 contain the values they had on entry to the procedure.
- Registers r5 to r7 contain arbitrary values.

Programming idioms for call sites

To implement a call that in C would look like

```
x = f(a, b, c);
```

I recommend that you write

```
push c on stack r2
push b on stack r2
push a on stack r2
goto f linking r1
x := r1
r2 := r2 + 3 // pop a, b, and c off the stack
```

After this sequence you should remember that values in r5, r6, and r7 may have been destroyed.

Idioms for use of registers

Many of the interesting macro instructions require at least two temporaries; some of the conditional branches can require four or more. As a matter of convention, I suggest that you reserve r0 to be zero and r6 and r7 as temporaries. You can do this by

```
.zero r0
.temps r6, r7
```

If you need more temporaries, the Macro Assembler will work just as well if r0 is a temporary:

```
.zero off
.temps r0, r6, r7
```

But this tactic creates an additional obligation; *before returning from a procedure, you must restore r0*:

```
.temps r6, r7
r0 := 0
.zero r0
```

Finally, for a conditional branch, you might need extra temporaries:

```
if (r3 <s a[r0][r2+1]) goto next using r1, r4, r5;
```

The using clause applies only to that instruction, but values in r1, r4, and r5 will be lost.

Programming idiom for a big procedure

If you're writing a big procedure, you'll want maximum use of all the registers. Here's a very general idiom for a procedure f that returns a result called result:

```
.zero r0
.temps r6, r7
f: // entry point for the function f
push r1 on stack r2 // save return address
push r3 on stack r2
push r4 on stack r2
// first argument is now in a[r0][r2+3]

... body of procedure f, which computes results ...
```

```

_f_end:
r1 := result
pop r4 off stack r2
pop r3 off stack r2
pop r5 off stack r2 // restore return address into r5
goto r5

```

If you need even more registers, you can make r0 a temporary and end with

```

...
pop r5 off stack r2
r0 := 0
.zero r0
goto r5

```

Idiomatic use for sections

Most programs will get by with four sections:

- Section `text` holds code.
- Section `data` holds initialized data.
- Section `stk` holds the stack.
- Section `init` is used for initialization and to call the main function.