COMP40 Assignment: Assembly-Language Programming

Contents

1 Purpose and overview

The purpose of this assignment is to deliver on the second half of the course title: you get to do some assembly-language programming. You will consolidate and solidify your knowledge of machine-level programming by implementing a calculator that uses Reverse Polish Notation[1], like the immortal HP 15C[2].

2 An RPN calculator

The COMP 40 RPN calculator reads commands from standard input and prints results to standard output. Like all RPN calculators, it works with a *value stack*. In this case, a value on the stack is one Universal Machine word. The command set is shown in Figure 1; Figure 2 shows an example interaction. You will find a complete reference implementation in file `/comp/40/www/homework/calc.c`[3], and you can run a binary in `/comp/40/bin/calc40`.

Your assignment is to implement this calculator in Universal Machine Assembly language. *Your calculator must duplicate the output of the reference implementation exactly.*

The implementation of the calculator is mostly straightforward: the only persistent state is the value stack, and this value stack is manipulated by each command independently of the others, using purely local reasoning. There is one dirty trick, however: in order to make it possible to read the digits of a numeral one character at a time, the calculator uses a finite-state machine with two states: *waiting* and *entering*. The normal state, which is also the initial state, is *waiting*. The *entering* state is used only when the entry of a numeral is in progress.

- If the machine is *waiting* and it sees a digit, it treats that digit as the start of a numeral, pushes the *value* of that digit, then transitions to the *entering* state.
- If the machine is *entering* and it sees a digit, that digit *continues* a numeral that was already pushed. The machine therefore takes the number on the top of the stack, multiplies it by 10, and adds the value of the next digit.
- In either state, if the machine sees a nondigit, it performs the command associated with that nondigit (if any), then transitions to the *waiting* state.

Here are two examples:

- If the machine sees the string "`42`", it first pushes the number 4 (value of the digit `'4'`), then transitions into the *entering* state. It then sees the digit `'2'` while still in the *entering* state, so it pops 4 and pushes $10 \times 4 + 2$, that is, 42. The result is the single number 42 on the stack.
- If the machine sees the string "`4 2`", with a space between the digits, it first pushes the number 4 (value of the digit `'4'`), then transitions into the *entering* state. It then sees the space character while still in the *entering* state. Because the space character is not a digit, the machine performs the associated command (doing nothing) and transitions back to the *waiting* state. Finally, while in the *waiting state*, it sees the digit `'2'`, so it pushes the number 2. The result is *two* numbers on the stack: 2 on top and 4 on the bottom.

[1] See URL `http://en.wikipedia.org/wiki/Reverse_Polish_notation`.
[2] See URL `http://hp15c.org/hp15c.php`.
[3] See URL `http://www.cs.tufts.edu/comp/40/homework/calc.c.txt`.

Command	*Function*
n	Push n onto the value stack, where n is a numeral (sequence of digits).
space	Does nothing, but may be used to separate numerals, as in the command sequence "`6 7*`."
newline	Print the contents of the value stack
`+`	Pop y from the value stack, then pop x from the value stack, then push $x + y$.
`-`	Pop y from the value stack, then pop x from the value stack, then push $x - y$.
`*`	Pop y from the value stack, then pop x from the value stack, then push $x \times y$.
`/`	Pop y from the value stack, then pop x from the value stack, then push $x \div y$. If y is zero, print an error message and leave the stack unchanged.
`|`	Pop y from the value stack, then pop x from the value stack, then push $x \vee y$, where $\vee$ stands for bitwise or.
`&`	Pop y from the value stack, then pop x from the value stack, then push $x \wedge y$, where $\wedge$ stands for bitwise and.
`c`	(Change sign.) Pop x from the value stack, then push $-x$.
`~`	Pop x from the value stack, then push $\neg x$, where $\neg$ stands for bitwise complement.
`s`	Swap the two values on top of the value stack (exchange x and y).
`d`	Duplicate the value on the top of the stack. (The HP 15C uses the ENTER key.)
`p`	Pop a value off the value stack and discard it.
`z`	Remove all values from the value stack (zero stack).

Figure 1: Calculator commands

```
sunfire31{nr}403: calc40
6 7 *
>>> 42
2 +
>>> 44
11 /
>>> 4
c
>>> -4
p
466 319sd+240c807c   sd-
>>> 0
>>> -807
>>> 932
>>> 319
```

Figure 2: Interacting with the RPN calculator

You can see for yourself the difference between `42` with no space and `4 2` with a space:

```
42
>>> 42
p
4 2
>>> 2
>>> 4
```

The C code keeps track of the state through the position of the program counter, using the two labels `entering` and `waiting`. To avoid duplicating the implementations of any commands, if the code for the *entering* state does not see a digit, it uses a `goto` to reuse the same code used in the *waiting* state.

3 Technical information

3.1 Useful macro instructions

Some critically important macro instructions were explained only briefly if at all in your class notes:

`push r3 on stack r2`	Register `r2` points to a stack, and this instruction subtracts 1 from `r2`, then stores `r3` at offset `r2` in segment 0.
`pop r5 off stack r2`	Register `r2` points to a stack, and this instruction loads register `r5` from offset `r2` in segment 0, then adds 1 to `r2`.
`pop stack r2`	Adds 1 to register `r2`.
`goto p linking r1`	Sets register `r1` to the offset of the instruction immediately following the `call` macro, then transfers control to the instruction labelled `p` in segment 0. Used to implement procedure calls.

3.2 Recommended calling convention

You may choose any calling convention you like, but for general purposes we recommend the following convention:

1. Arguments are passed on the call stack, which is pointed to by register `r2`. The callee sees the first argmument is at the lowest address (`m[0][r2]`), with subsequent arguments at higher addresses. In this convention, if you examine a sequence of `push` instructions in the caller, you'll see that the caller pushes the first argument last.

2. Register `r0` is always zero.

3. On entry to a procedure, register `r1` holds the return address. If you write a procedure that itself makes a call, you will have to save and restore the procedure's return address.

   If a procedure returns a result, the result should be returned in register `r1`.

4. Register `r2` is the stack pointer.

5. Registers `r3` and `r4` are nonvolatile general-purpose registers. If you use either of these registers in a procedure, you must save and restore them.

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

// return address in r1, which gets result
// stack pointer in r2
// nonvolatiles r0, r3, r4
// r0 is zero
double:
  push r1 on stack r2 // save return address
  push r3 on stack r2 // save nonvolatile registers
  push r4 on stack r2

  r3 := m[r0][r2+3] // load argument into r3
  r1 := r3 + r3     // result goes into register

  pop r4 off stack r2 // restore nonvolatile registers
  pop r3 off stack r2
  pop r5 off stack r2 // put return address in r5
  goto r5 // return
```

Figure 3: An assembly procedure that returns double its argument

6. Registers `r5`, `r6`, and `r7` are volatile registers and are not saved and restored by procedure calles.

We also recommend that you dedicate registers `r6` and `r7` for use as temporaries.

Using this convention, Figure 3 shows a slightly paranoid procedure that doubles its argument. In Figure 3, it is not really necessary to save `r3` and `r4`, since everything could have been done using `r5`, but the model works in the general case.

(New for Spring 2014): The class notes and slides recommend creating `urt0.ums` with startup code to create a call stack and to initialize register 0. The design proposal below comes from Norman Ramsey and it uses a slightly different approach, with a file named `stack.ums` providing a similar service. As with other conventions, it's your choice for the RPM calculator whether to handle initialization using something like `urt0.ums`, `stack.ums`, or some other way that you find appealing.

4 Design and implementation plan

4.1 Sections

Norman Ramsey's assembly code is explained here. It uses these sections:

`text`	Contains procedure definitions, including the definition of `main`.
`data`	Contains a preallocated call stack and other data structures.
`rodata`	Contains jump tables.
`init`	Contains setup code, including code to set up the stack, code to initialize jump tables, and code to call `main` when setup is complete.

4.2 Modules

Norman Ramsey's implementation is split into four assembly-language source files:

1. File `stack.ums` allocates space for the call stack (in the data section) and initializes the stack pointer (with code in the `init` section). Not counting blank lines or comments, his implementation of this module is only 6 lines of assembly code.
2. File `printd.ums` contains a function for printing Universal Machine words in decimal.
3. File `calc40.ums` contains calculator-related functions.
4. File `callmain.ums` puts code in the `init` section which makes the initial call to `main`, then halts. Not counting blank lines or comments, the implementation of this module is only 5 lines of assembly code.

It is important that `stack.ums` come first and `callmain`.ums come last, so that the stack pointer is initialized before any other code runs, and so that `main` is not called until all the other code in the `init` section runs. For example,

```
umasm stack.ums calc40.ums printd.ums callmain.ums > calc40.um
```

Note that the `umasm` framework concatenates and assembles as one all the `.ums` source files listed on the command line. [4]

4.3 Data structures

There is really only one data structure in the program, which is the value stack. We recommend that you reserve space in segment 0 so that you can take advantage of the `push` and `pop` instructions. (We will be testing your calculator on random inputs, so *be sure that your value stack is capable of holding at least ten thousand values*.) Another alternative is to use segments to make a linked list.

4.4 Implementation of the print module

The print module is among the more challenging modules in the calculator. Printing Universal Machine words as numbers requires three or four cases:

- Zero is the only number that is printed with a leading zero, so we recommend you handle it as a separate case.

[4] Any `umasm` program you built for COMP 40 automatically inherits this capability from the framework.

- Positive and negative numbers are separate cases; only negative numbers are printed with leading minus signs.
- The most negative number, `0x80000000`, causes all sorts of pain. The Universal Machine lacks a fully functional comparator, and the best comparator we've been able to simulate allows this number to compare as *both* greater than *and* less than zero. You can either treat it as a special case or take extraordinary care with your comparisons.

The reason the print module is difficult is that the number you start with is binary, but of course you need to print it as a decimal. Accessing the decimal digits to be printed, and especially getting them in the order you need them, can be tricky. We are aware of a few kinds of solutions:

- Accumulate digits into some kind of data structure, then print them. Norman Ramsey used a linked list made up of two-word Universal Machine segments, but you may use any data structure you like. His implementation of this solution is about 40 lines of assembly code.
- Write a recursive print function:
  - To print a 1-digit number, print the digit
  - To print an n-digit number, print the most significant $n-1$ digits, then print the least significant digit

  The recursive print function takes about 35 lines of assembly code.
- We are aware of at least one strictly iterative approach involving no dynamic memory allocation and no recursion. Our implementation of this approach is about 38 lines of assembly code; the math that underlies this one is a little trickier than the others.

Any of the above approaches, or any other approach with code complexity and performance not much worse than these is acceptable.

4.5 Implementation of the calculator module

The implementation of the calculator module is about 250 lines of assembly code, but most of these lines are very repetitive—there are fifteen commands, and each one has to check for operands on the stack, do some manipulation, and some control flow. The codes are all quite similar. We recommend you take advantage of these tricks:

- There aren't very many registers, but you can afford to reserve a couple for key variables and data structures. Norman Ramsey reserves one register to hold the value stack and another to hold the character read in (only for as long as needed).

  Two temporaries will be enough for most purposes, but you will occasionally need more. Unless the character read in is a digit, once you have dispatched through the jump table, you can reuse your input-character register as a temporary.
- We recommend that you implement the `switch` statement for the *waiting* state using a jump table with 256 entries. The jump table is declared like this:

```
waiting:
  r1 := input()
waiting_with_character:
  ... test to see if r1 signals end of file,
      and if so, go to end of procedure ...

  // branch indirect through jump table
  r5 := jumptable + r1
  r5 := m[r0][r5]
  goto r5
```

The code ensures that every possible entry in the jump table is meaningful—all 255 values.

To initialize the jump table, the code uses the `init` section aggressively:

- The module begins with `init`-section code that sets every entry in the jump table to the label `input_error`. The code associated with this label prints the "unknown character" error message, then goes back to the *waiting* state.
- After initializing every entry in the table to `input_error`, The code overwrites the ten entries associated with the digits `0` through `9`. Since each of these works the same way, each entry points to the same `digit` label.
- Since the space character does nothing but force the machine to transition to the *waiting* state, the `waiting` label is assigned directly into the jump table:

  ```
  m[r0][jumptable + ' '] := waiting
  ```

- Norman Ramsey's strategy was to implement operators one at a time. For each operator, he used the same pattern. Here's an example for multiply:

  ```
  ////////////// multiply
  .section init
  m[r0][jumptable + '*'] := mul
  .section text

  mul:
    ... check to make sure there are two operands on the value stack ...
    ... pop the two operands and push the product ...
    goto waiting
  ```

  By switching back and forth between the `init` and `text` sections, he makes the implementation of each operator self-contained.

- Almost every operator has to make sure there are enough operands on the stack. The C code employs a general-purpose procedure called "`has`.', but in his assembly code, Norman Ramsey uses what he calls a "really dirty trick": he defines labels `check1` and `check2`, and transfers control using the `goto... linking...` construct. If a check succeeds, he transfers control back to the point of origin, using the link register. If a check fails, he prints an error message and issues as `goto waiting`.
- Most operators are very easy to implement, but the newline operator (print stack) requires a loop, and the signed-division operator requires a lot of case analysis (just as in the C code).

- We recommend that you implement the parts of your calculator module in this order:

  1. The code to initialize the jump table, plus the main loop of the calculator function, which reads a character, checks for EOF, and transfers control via the jump table
  2. Entry of single digits only
  3. The space command
  4. The newline command, which prints the stack—and which will enable you to see your first useful output, provided you avoid multi-digit numerals
  5. Digits for the *entering* state, so that you can read multi-digit numerals[5]
  6. A couple of binary operators like `+` and `*`, including operand checking
  7. A couple of unary operators like `c` and `~`.
  8. The rest of the operators, doing signed division last

5 Debugging techniques

Assembly code is hard to debug. You will need to add some debugging code to your Universal Machine. One option is to make your debugging code conditional on an environment variable such as `UMTRACE`. You can implement this by making a single check when your Universal Machine starts to see if you should be tracing:[6]

```
bool trace = getenv("UMTRACE") != NULL;
```

Then, in the execution loop, you can print information conditioned on the trace:

```
if (trace) {
  Um_instruction instruction = *pc;
  char *asm = (char *)Um_disassemble(instruction);
  if (OP(instruction) == LV)
    fprintf(stderr, "%7" PRIdPTR ": %s\n", pc - prog, asm);
  else
    fprintf(stderr, "%7" PRIdPTR ": %s  (r%d = %d, r%d = %d, r%d = %d)\n",
            pc - prog, asm,
            A(instruction), RA, B(instruction), RB, C(instruction), RC);
  FREE(asm);
}
```

This code prints each PC and instruction before it is executed, along with the values of the registers mentioned in the instruction.

[5] Norman didn't bother with a jump table here; he just checked to see if the input character c was in the range `'0'` $\leq c \leq$ `'9'`. For the comparisons, this required an extra temporary register, which is identified with `using` in the code.

[6] `#include <stdbool.h>` to get the `bool` type.

Here's some advice:

- Run

  ```
  umdump calc40.um | less
  ```

  in one terminal window and

  ```
  UMTRACE=1 valgrind ./um calc40.um 2>&1 | less
  ```

  in another window.

  (If you're stuck using a stupid "C shell," you'll have to use `setenv` and `unsetenv` to control the value of the `UMTRACE` environment variable.)

- Many, many bugs occur when the call stack is not properly adjusted—for example, you push an argument onto the call stack, then after the call returns, you forget to take the argument off the call stack. Keep an eye on the stack pointer to make sure it has the proper values as you call and return.

- In the heat of coding it's easy to forget about proper control flow. Consider organizing your assembly code into short blocks such that each block ends with a `goto`. That way you will never "fall through" and execute code (or data) unintentionally.

- When in doubt, blast `output` macros into your code. The `halt` instruction is also your friend.

- If you fall into a hole, *stop digging*. Get help.

6 What we provide for you

Your mission is to implement the RPN calculator in Universal Machine assembly language. Here's the support you get from us:

- We provide a reference implementation in C whose functionality you must duplicate *exactly*. Source code is in `/comp/40/www/homework/calc.c`, and you can run the binary as `/comp/40/bin/calc40`.

- We provide a random-input generator; the command is `random-calc40`. With no argument, it emits 100 random operators. With an argument, it emits a given number of operators. Here are a couple of examples (newlines have been added for clarity):

  ```
  $ random-calc40
  812 106cd~d690c943d+ dp253c980c879   &957c&d  / 142c/ &c- 757
    49c+| ~~835 846c 225c |d |c&d/ dd& 655* 434c914 +d  *&  d 361
    486/d&|*-* s c509~| s s   ~    d191ds  ~ d|dcd-d d*     pd+391|pd-
    ~  868cs dp&c  c+
  $ random-calc40 5
    d340c5ds
  ```

  A couple of notes:

  - The random-input generator *will* emit operations that fail, but it's not very likely.
  - The probability distributions are skewed so that if there are no errors, the value stack tends to stay close to 10 values. But when there are errors, the value stack grows proportional to the number of tests. This is why *you need a value stack that can handle at least ten thousand elements*.
  - The generator counts only "interesting" operators, so your hand count may not be identical to the argument. You can see what's interesting by examining the source code at `/comp/40/bin/random-calc40`.

- We provide you with a test script that will compare the results of your UM binary with the reference implementation. It takes two arguments: the name of your `.um` file and the number of random operators to test. Here's an example:

```
$ time calc40-test calc40.um 1000
Results identical -- test passed
$ time calc40-test calc40.um 1000000
Results identical -- test passed
```

  With our solution-grade Universal Machine we can test a million-operator inputs in a few seconds.

  Be aware that *the random-test generator does not find many error cases*.

- As soon as the last Macro Assembler is turned in, we will provide a working Macro Assembler.

7 What we expect from you

7.1 Documentation

Assembly code requires the same kinds of documentation as C code, but in more places.

- Representation is still the essence of programming, so we expect you to document your data structures. This means explaining the representations at the machine level.

- In assembly code, registers play key roles; *we expect you to document the use of each register*. Register documentation may be global (e.g., register `r0` always holds zero), may be specific to one procedure (e.g., in this procedure, register `r5` holds the number to be printed), or may be specific just to a few parts of one procedure (e.g., in this region, register `r1` holds the input character). *We expect you to document your use of machine registers*.

- We expect that *an assembly-language procedure will be documented in the same way as a C procedure*, that is:
  - You will document the type and meaning of each argument.
  - You will document the type of the result, if any.
  - You will document the function's *contract*.
  - You will *not* narrate a sequence of events performed by the function.

- Not all source files will define or contain procedures, but if a source file *does* contain one or more procedures, that source file *must* include brief documentation of the calling convention. *Even if you are using the standard calling convention everywhere, we expect you to place a brief summary in each relevant source file.* For an example, see Figure 3.

- We expect you to document important internal labels. (Labels used to implement purely local `if` statements or `loops` need little if any documentation, but a label that is used far away must be documented.)

  The documentation of labels should be connected to the organization of your assembly code into *blocks*, as discussed in class. (A block begins with a label and ends with an unconditional `goto`.)[7] We expect you to document the label of each block with its *contract*. Again, a contract is *not* a narration of the events performed within a block. Here are some examples:

  - *Poor contract:* "print a minus sign and goto `L7`." (We could see this from the code.)
  - *Fair contract:* "print a negative number"
  - *Good contract:* "print a negative number and return"
  - *Very Good contract:* "print the value of register `r5` in decimal, then return, where `r5` must be negative"

  We hope it will help you to remember that *the purpose of the contracts is to enable modular reasoning*. In particular, you should be able to debug each individual block by knowing only the contract of that block and the contracts of any labels it may branch to. If, for example, there is a label `print_pos:` with the contract "print the decimal representation of `r5`, where `r5` must be positive, then return", then we know that the following block is correct:

```
print_neg: // print r5 in decimal, then return (r5 must be negative)
  output '-'
  r5 := -r5
  goto print_pos
```

7.2 "Design"

On this project, you don't get to do much design. There will be no design document submitted, but please *think through* these questions, which would appear in a design document if there were one:

- Make a short sketch of what data structures you will need and how they will be represented on the Universal Machine.

  If you wish to change the calling convention or to design your own calling convention, sketch it out before writing code.

- As a first test, write a file `printd.ums`, which stands for "print decimal," and contains a a *documented* procedure that prints a positive number.

Again, there is no design document, but we are asking you to write some documentation for your own purposes.

[7] N.B. Code in an `init` section may have internal labels and gotos, but it should not *end* in a goto. Every `init` section except the last should end simply by continuing ("falling through") to the next `init` section. The last `init` section should call `main` and then `halt`.

7.3 Final submission

By the deadline use the script `submit40-asmcoding` to submit

- All the assembly code you have written. *Each assembly file must have a name that ends in* `.ums`.
- A script called `compile` that assembles all your source files and creates a Universal Machine binary called `calc40.um`. This script should call `umasm` *without* a dot. The script should not be more than one or two lines long.

  If you want to use your own assembler, begin your script with the additional lines

  ```
  PATH=".:$PATH"
  export PATH
  ```

  which will cause the script to look for `umasm` in the current directory first.

- A `README` file which
  - Identifies you and your programming partner by name
  - Acknowledges help you may have received from or collaborative work you may have undertaken with others
  - Identifies what has been correctly implemented and what has not
  - Explains any departures from the recommended calling convention
  - Explains in one sentence how you chose to implement the print module
  - Says approximately how many hours you have spent *analyzing the assignment*
  - Says approximately how many hours you have spent *writing assembly code*
  - Says approximately how many hours you have spent *debugging your calculator*