An Assembler

Daniel Bastos
`dbastos@toledo.com`

August 9th 2006

Abstract

This is an educational assembler I studied when reading "The AWK Programming Language" by Brian Kernighan. It's very simple, but quite educational. It's written in AWK, a marvelous language.

1 Introduction

We're writing an assembler that implements a simple assembly language for a simple hypothetical computer. Very simple. The computer has one register — which we often make reference to by using the symbol `%r` —, ten instructions and a memory capable of a thousand words. Each word is composed of 5 bits, each bit is represented by a digit. If a word is a machine instruction, then the first two bits encode the operation and the last three digits are the memory address. The memory addresses range from 0 to 999.

Each program is a sequence of statements, where each statement is separated by a new line character. Each statement is composed of three fields: a label, an operation and an operand. Spaces or tabs, in any amount, separate the fields. Comments start anywhere after the character `#` and end once a new line is found.

We implement the following instructions: `const`, `get`, `put`, `ld`, `st`, `add`, `sub`, `jp`, `jz`, `j`, `halt`. These instructions, respectively, have the operation codes 00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 10. The keyword `const` isn't precisely an instruction. It is an assembler directive to define a constant.

The instruction `get` reads a number from the input into the register. The instruction `put` writes the data from the register to the output. The instruction `ld mem` loads the register with the data in memory location `mem`. The instruction `st` saves the data contained in the register at the memory location `mem`. The instruction `add` adds the number from the location `mem` to the number in the register. The instruction `sub` substracts the number from the location `mem` to the number in the register. The instruction `jp` jumps execution to `mem` if the number in the register is positive. The instruction `jz` jumps execution to `mem` if the number in the register is zero. The instruction `j` jumps execution to `mem` unconditionately.

2 Implementing the assembler

The assembler takes the first argument of the command line as the given to be assembled. Object code is written to `.o` files. The array `op` stores the operation codes defined in the `split` line. In the first pass, we remove comments, load the symbols into the `sym` array, and save the object code in the `.o` file.

The way we load the symbols into the `sym` array is by making the symbol name the index of the array, and the value of the index is the memory position of the symbol.

2 ⟨*das.awk* 2⟩≡

```
BEGIN {

  if (ARGC < 2) { print "Usage: das program.das"; exit 0; }

  src = ARGV[1]; sub(/\..*/,"",ARGV[1]);
  exe = ARGV[1]; obj = exe ".o"; ARGV[1] = "";

  n = split("const get put ld st add sub jp jz j halt", x);
  for (i = 1; i <= n; ++i) op[x[i]] = i - 1;

  FS = "[ \t]+"; # first pass
  while (getline < src > 0) {
    if ($0 ~ /^[ \t]*#/ || $0 ~ /^[ \t]*$/) continue;
    human[c++] = $0; sub(/\#.*/, ""); sym[$1] = nextmem;
    if ($2 != "") { print $2 "\t" $3 > obj; ++nextmem; }
  }

  close(obj);

  nextmem = c = 0; # second pass
  while (getline < obj > 0) {
    if ($2 !~ /^[0-9]*$/) { $2 = sym[$2]; }
    mem[nextmem++] = 1000 * op[$1] + $2;
    printf "%4d:  %05d  %s\n", c, mem[c], human[c]; ++c;
  }

  e = --c;

  for (c = 0; c <= e; ++c) print mem[c] > exe
  system("chmod 0755 " exe);
}
```

On the second pass, if the second argument of the line is not a number, then it's because the programmer used a label, so we must recover the memory address of that label in the array `sym` which we computed in the first pass.

The next step is the generation of machine code; for our hypothetical computer, we encode instructions by bundling the operation code with the the argument's memory address. Each bundle occupies a word, where the first two bits are the instruction itself and the last three are the memory address. The bundling is simple: we multiply the operation code by 1000, so we're always adding three zero digits after the operation code. The memory layout ranges from 0 to 999, so we need only three digits to save the highest block of memory. So, we can simply add the argument's memory address.

Before we write the executable, we create a memory dump together with the human assembly code. The first field of the memory dump is the memory address of that instruction, so we start at zero. The second field is the machine code, and the rest of the line is the original source code. It's easy to debug programs this way.

3 Implementing the machine

In the execution, we have all the machine code stored in the array `mem`, which we obtain by reading the program. To extract the instruction from the encoded bundle, we need only divide the word by 1000 and get rid of a possible fractional part. For the argument's memory address, we extract the remainder of the division by 1000. The rest is just the implementation of each instruction.

3 ⟨*vm.awk* 3⟩≡

```
BEGIN {

  if (ARGC < 2) { print "Usage: vm program"; exit 0; }

  exe = ARGV[1]; ARGV[1] = "";

  do {
    r = getline < exe;

    if ($0 !~ /^[0-9]+$/) {
      print "fatal: illegal instruction:", $0; exit 1;
    }

    mem[i++] = $0;
  } while (r > 0);

  n = split("const get put ld st add sub jp jz j halt", x);
  for (i = 1; i <= n; ++i) op[x[i]] = i - 1;

  for (ip = 0; ip >= 0;) { # execution
    addr = mem[ip] % 1000; code = int(mem[ip++] / 1000);

    if (code == op["get"]) { getline r; }
    else if (code == op["put"]) { print r; }
```

```
    else if (code == op["st"]) { mem[addr] = r; }
    else if (code == op["ld"]) { r = mem[addr]; }
    else if (code == op["add"]) { r += mem[addr]; }
    else if (code == op["sub"]) { r -= mem[addr]; }
    else if (code == op["jp"]) { if (r > 0) ip = addr; }
    else if (code == op["jz"]) { if (r == 0) ip = addr; }
    else if (code == op["j"]) { ip = addr; }
    else if (code == op["halt"]) { ip = -1; }
    else { ip = -1; }
  }

  exit 0;
}
```

4 Examples

The first example is cute. All it does is read a number into the register, then it keeps incrementing that number and printing it; nothing else. Note that this program never ends on purpose. Hit `^C` to stop.

4 ⟨*seq.das* 4⟩≡

```
# prints a sequence of numbers

        get             # read first number
loop    add     one     # increment
        put
        j       loop    # again
one     const   1

# this program never ends
```

By the default, the assembler prints a debug information after assembling the program; for example, the following session serves as an example.

```
%./das seq.das
   0:  01000          get            # read first number
   1:  05004  loop    add     one    # increment
   2:  02000          put
   3:  09001          j       loop   # again
   4:  00001  one     const   1

%ls -l seq
-rwxr-xr-x  1 dbastos  dbastos  22 Aug 18 08:05 seq*

%./vm seq
```

```
-3          # entered by the user
-2
-1
0
1
2
^C
```

Although we create an executable program, that's just a text file with machine code. To execute the program, you must call `vm` passing the program as the first argument. It would be nice to have `vm` as an interpreter, so that we can generate programs in which the first line would be `#!/path/to/vm` and they'd be true UNIX executables.

Now let's write a smarter program; we will call it `rev`; `rev` will print the sequence of numbers always in reverse order until it reaches zero. Once it reaches zero, we end execution.

5 ⟨*rev.das* 5⟩≡

```
# prints a descending sequence of numbers

        get           # read first number
loop    put
        jp      pos   # jump to pos if positive
        add     one   # increment %r
        j       test
pos     sub     one   # decrement %r
test    jz      done  # if it's zero, halt
        j       loop

done    halt
one     const   1
```

Notice that we have to use a roundabout way to decrement or increment a number in the register. We must allocate memory for the number 1, and then make reference to it by using it's memory location with the instruction sub. The same happened in `seq` when we wanted to increment the number in the register. It would be nice to have a instruction or a change in the syntax of the language that allows us to simply increment or decrement the number in the register by some constant. If we add this recourse, then we could create an assembler instruction — not a machine instruction — to increment and decrement the register by one; something like `inc` and `dec`.