Computer Science 3 - 2014

Programming Language Translation

Practical 2, Week beginning 31 March 2014 - Solutions

There were some very good solutions submitted, and some very energetic ones too - clearly a lot of students had put in many hours developing their code. This is very encouraging. Do learn to put your names into the introductory comments of programs that you write.

Full source for the solutions summarized here can be found in the ZIP file on the Web page - `PRAC2A.ZIP` (Java) and `PRAC2AC.ZIP` (C#).

Task 2 - A look at PVM code

Task 2 was to examine the PVM code for a simple Parva program that demonstrated de Morgan's Laws.

The code does not use short circuit evaluation, which as we shall soon see makes it easier to translate without requiring what one of my compiler-writing friends and fellow authors, John Gough, calls "jumping code".

Since *false* and *true* are represented internally by 0 and 1, to get the table in numeric form requires only that one change the PRNB instructions into PRNI instructions!

Have a look at how I have commented this, using "high level" code, rather than detailed line by line commentary of the form "load address of X". Some of the submissions had "commentary" that was, frankly, almost useless. Try the following test for assembler code: Cover over the real code with a piece of paper and read only the comments. Does what you read make sense on its own? I maintain that it should. The easiest way to achieve this is by using a high level algorithmic notation.

```
                   ; Demonstrate de Morgan's Laws
                   ; P.D. Terry, Rhodes University, 2014

  0 DSP    2       ; bool x is v0, y is v1
  2 PRNS   "   X      Y    (X.Y)\' X\'+Y\'  (X+Y)\' X\'.Y\'\n\n"
  4 LDA    0       ;                               47 OR              ;
  6 LDC    0       ;                               48 NOT             ;
  8 STO            ; x = false;                    49 PRNI            ;     write(!(x || y));
  9 LDA    1       ; repeat                        50 LDA       0     ;
 11 LDC    0       ;                               52 LDV             ;
 13 STO            ;   y = false;                  53 NOT             ;
 14 LDA    0       ;   repeat                      54 LDA       1     ;
 16 LDV            ;                               56 LDV             ;
 17 PRNI           ;     write(x);                 57 NOT             ;
 18 LDA    1       ;                               58 AND             ;
 20 LDV            ;                               59 PRNI            ;     write(!x && !y);
 21 PRNI           ;     write(y);                 60 PRNS      "\n"  ;     writeLine();
 22 LDA    0       ;                               62 LDA       1     ;
 24 LDV            ;                               64 LDA       1     ;
 25 LDA    1       ;                               66 LDV             ;
 27 LDV            ;                               67 NOT             ;
 28 AND            ;                               68 STO             ;     y = !y;
 29 NOT            ;                               69 LDA       1     ;
 30 PRNI           ;     write(!(x && y));         71 LDV             ;
 31 LDA    0       ;                               72 NOT             ;
 33 LDV            ;                               73 BZE       14    ;   until (!y);
 34 NOT            ;                               75 LDA       0     ;
 35 LDA    1       ;                               77 LDA       0     ;
 37 LDV            ;                               79 LDV             ;
 38 NOT            ;                               80 NOT             ;
 39 OR             ;                               81 STO             ;   x = !x;
 40 PRNI           ;     write(!x || !y);          82 LDA       0     ;
 41 LDA    0       ;                               84 LDV             ;
 43 LDV            ;                               85 NOT             ;
 44 LDA    1       ;                               86 BZE       9     ; until (!x);
 46 LDV            ;                               88 HALT            ; System.Exit();
```

Task 4

Task 4 was to hand-compile the Factorial program into PVM code. Most people got a long way towards this. Once again, look at how I have commented this, using "high level" code.

```
 0 DSP    3       ; n is v0, f is v1, i is v2
 2 LDA    0
 4 LDC    1
 6 STO            ; n = 1;
 7 LDA    0
 9 LDV
10 LDC    20      ; // max = 20, constant
12 CLE            ; while (n <= max) {
13 BZE    78
15 LDA    1
17 LDC    1
19 STO            ;   f = 1;
20 LDA    2
22 LDA    0
24 LDV
25 STO            ;   i = n;
26 LDA    2
28 LDV
29 LDC    0
31 CGT            ;   while (i > 0) {
32 BZE    55
34 LDA    1
36 LDA    1
38 LDV
39 LDA    2
41 LDV
42 MUL
43 STO
44 LDA    2       ;     f = f * i;
46 LDA    2
48 LDV
49 LDC    1
51 SUB
52 STO            ;     i = i - 1;
53 BRN    26      ;   }
55 LDA    0
57 LDV
58 PRNI           ;   write(n);
59 PRNS   "! = "  ;   write("! = ");
61 LDA    1
63 LDV
64 PRNI           ;   write(f);
65 PRNS   "\n"    ;   write("\n") (or use PRNL)
67 LDA    0
69 LDA    0
71 LDV
72 LDC    1
74 ADD
75 STO            ;   n = n + 1;
76 BRN    7       ; }
78 HALT
```

Note that max is a constant, not a variable. There is no need to assign it a variable loaction and store 20 into this - simply build the value of 20 into the instructions that need to use it.

Task 5 - Trapping overflow

Checking for overflow in multiplication and division was not always well done. You cannot easily multiply and then try to check overflow (it is too late by then) - you have to detect it in a more subtle way. Here is one way of doing it -note the check to prevent a division by zero. This does not use any precision greater than that of the simulated machine itself. Note that it is necessary to check for "division by zero" in the rem code as well!

```
case PVM.mul:           // integer multiplication
  tos = pop();
  sos = pop();
  if (tos != 0 && Math.abs(sos) > maxInt / Math.abs(tos)) ps = badVal;
  else push(sos * tos);
  break;

case PVM.div:           // integer division (quotient)
  tos = pop();
  if (tos == 0) ps = divZero;
  else push(pop() / tos);
  break;

case PVM.rem:           // integer division (remainder)
  tos = pop();
  if (tos == 0) ps = divZero;
  else push(pop() % tos);
  break;
```

It is possible to use an intermediate long variable (but don't forget the casting operations or the abs function):

```
case PVM.mul:           // integer multiplication
  tos = pop();
  sos = pop();
  long temp = (long) sos * (long) tos;
  if (Math.abs(temp) > maxInt) ps = badVal;
  else push(sos * tos);
  break;
```

Task 6 - Arrays

The code as supplied for tracking students' attendance at a practical suffered from various defects - a number of zero is useless, even though it would be accepted quite happily, a student is able to clock in more than once, the constant StudentsInClass has a misleading value, and if a large negative number is supplied the program crashes. A few simple changes will fix some or all of these. I was happy to accept just one or two of these changes, but here is a rather radical rewrite that embraces them all, and uses the value 0 to terminate the program, just so that you can have a look at how this would have been translated. (`STUDENTS1.PAV`):

```
void main () {
// Track students as they clock in and out of a practical - improved version
// P.D. Terry, Rhodes University, 2014
// Improved version

  const StudentsInClass = 100;
  bool[] atWork = new bool[StudentsInClass + 1];

  int student = 1;
  while (student <= StudentsInClass) {
    atWork[student] = false;
    student = student + 1;
  }

  read("Student? (> 0 clocks in, < 0 clocks out, 0 terminates) ", student);
  while (student != 0) {
    bool clockingIn = true;
    if (student < 0) {
      clockingIn = false;
      student = -student;
    }
    if (student > StudentsInClass)
      write("Invalid student number\n");
    else if (clockingIn)
      if (atWork[student])  write(student, " has already clocked in!\n");
      else atWork[student] = true;
    else
      if (!atWork[student]) write(student, " has not yet clocked in!\n");
      else atWork[student] = false;
    read("Student? (> 0 clocks in, < 0 clocks out, 0 terminates) ", student);
  }

  write("The following students have still not clocked out\n");
  student = 1;
  while (student <= StudentsInClass) {
    if (atWork[student]) write(student);
    student = student + 1;
  }

} // main
```

A translation into PVM code is a little tedious, and it is easy to leave some of the code out and get a corrupted solution:

```
                  ; Track students as they clock in and out pf a practical
                  ; P.D. Terry, Rhodes University, 2014
                  ; bool[] atwork is v0, int student is v1

  0 DSP      3     ;
  2 LDA      0     ;
  4 LDC      100   ;
  6 LDC      1     ;
  8 ADD            ;
  9 ANEW           ;
 10 STO            ; bool[] atWork =  new bool[...]
 11 LDA      1     ;
 13 LDC      1     ;
 15 STO            ; int student = 1;
 16 LDA      1     ;
 18 LDV            ;
 19 LDC      100   ;
 21 CLE            ;
 22 BZE      45    ; while (student <= 100) {
 24 LDA      0     ;
 26 LDV            ;
 27 LDA      1     ;
 29 LDV            ;
 30 LDXA           ;
 31 LDC      0     ;
 33 STO            ;   atWork[Student] = false;
 34 LDA      1     ;
 36 LDA      1     ;
 38 LDV            ;
 39 LDC      1     ;
 41 ADD            ;   student = student + 1;
 42 STO            ;
```

```
106 LDXA           ;
107 LDV            ;
108 BZE      118   ;    if (atWork[student])
110 LDA      1     ;
112 LDV            ;      write (student)
113 PRNI           ;
114 PRNS     " has already clocked in!\n"
116 BRN      128   ;
118 LDA      0     ;    else
120 LDV            ;
121 LDA      1     ;
123 LDV            ;
124 LDXA           ;
125 LDC      1     ;
127 STO            ;      atWork[student] = true;
128 BRN      159   ;
130 LDA      0     ;   else
132 LDV            ;
133 LDA      1     ;
135 LDV            ;
136 LDXA           ;
137 LDV            ;
138 NOT            ;
139 BZE      149   ;     if (!atWork[student]
141 LDA      1     ;
143 LDV            ;       write(student)
144 PRNI           ;
145 PRNS     " has not yet clocked in!\n"
```

```
 43 BRN      16    ; }
 45 PRNS     "Student? (> 0 clocks in, < 0  ...
 47 LDA      1     ;
 49 INPI           ; read(student);
 50 LDA      1     ;
 52 LDV            ;
 53 LDC      0     ;
 55 CNE            ;
 56 BZE      166   ; while (student != 0) {
 58 LDA      2     ;
 60 LDC      1     ;
 62 STO            ;   bool clockingIn = true;
 63 LDA      1     ;
 65 LDV            ;
 66 LDC      0     ;
 68 CLT            ;
 69 BZE      83    ;  if (student < 0) {
 71 LDA      2     ;
 73 LDC      0     ;
 75 STO            ;    clockingIn = false;
 76 LDA      1     ;
 78 LDA      1     ;
 80 LDV            ;
 81 NEG            ;
 82 STO            ;    student = - student
 83 LDA      1     ;  }
 85 LDV            ;
 86 LDC      100   ;
 88 CGT            ;
 89 BZE      95    ;  if (student > StudentsInClass)
 91 PRNS     "Invalid student number"
 93 BRN      159   ;
 95 LDA      2     ;
 97 LDV            ;
 98 BZE      130   ;  else if (clockingIn)
100 LDA      0     ;
102 LDV            ;
103 LDA      1     ;
105 LDV            ;
```

```
147 BRN      159   ;
149 LDA      0     ;
151 LDV            ;     else
152 LDA      1     ;
154 LDV            ;
155 LDXA           ;
156 LDC      0     ;
158 STO            ;      atWork[student] = false];
159 PRNS     "Student? (> 0 clocks in, < 0 ...
161 LDA      1     ;
163 INPI           ;   read(student)
164 BRN      50    ; } // while (student != 0)
166 PRNS     "The following students have still not ...
168 LDA      1     ;
170 LDC      1     ;
172 STO            ; student = 1;
173 LDA      1     ;
175 LDV            ;
176 LDC      100   ;
178 CLE            ;
179 BZE      206   ; while (student <= 100
181 LDA      0     ;
183 LDV            ;
184 LDA      1     ;
186 LDV            ;
187 LDXA           ;
188 LDV            ;
189 BZE      195   ;   if (atWork[student])
191 LDA      1     ;
193 LDV            ;
194 PRNI           ;     write(student);
195 LDA      1     ;
197 LDA      1     ;
199 LDV            ;
200 LDC      1     ;
202 ADD            ;
203 STO            ;     student = student + 1;
204 BRN      173   ; } // while (student <= 100
206 HALT           ; System.Exit()
```

Task 7 - Your lecturer is quite a character

Reading and writing characters was trivially easy, being essentially a simple variation on the cases for numeric input and output. However, the output of numbers was arrranged to have a leading space; this is not as pretty when you see i t a p p l i e d t o c h a r a c t e r s , i s i t - which is why the call to `results.write` uses a second argument of 1, not 0 (this argument could have been omitted). Note the use of the modulo arithmetic to ensure that only sensible ASCII characters will be printed:

```
case PVM.inpc:          // character input
  mem[pop()] = data.readChar();
  break;
case PVM.prnc:          // character output
  if (tracing) results.write(padding);
  results.write((char) (Math.abs(pop()) % (maxChar + 1)), 1);
  if (tracing) results.writeLine();
  break;
```

Extending the machine and the assembler still further with opcodes `CAP`, `INC` and `DEC` was also straightforward. However, many people had not considered the hint that one should not limit the `INC` and `DEC` opcodes to cases where they can handle only assignment statements like `X++;`. In some programs you might want to have statements like `List[N+6]++;`.

Hence, the opcodes for the equivalent of a ++ or -- operation produced interesting answers. There are clearly two approaches that could be used: either increment the value at the top of the stack, or increment the variable whose address is at the top of the stack. Of course, the latter is more useful if you are to have but one of these, and if we regard a statement like `n++;` as an assignment statement, the first step in hand translation for such a statement is to arrange for the address of the "target" to be pushed onto the top of stack. All fits beautifully, does it not? (One could, of course, provide both versions of the opcodes, but we don't need them here.). Here is my suggestion (devoid of precautionary checking - see if you can make it safer for yourself):

```
case PVM.cap:           // toUpperCase
  push(Character.toUpperCase((char) pop()));
  break;
```

```
        case PVM.inc:          // ++
          mem[pop()]++;
          break;
        case PVM.dec:          // --
          mem[pop()]--;
          break;
```

In terms of these opcodes `SENTENCE.PVM` is quite easily written as follows:

```
                  ; Read a sentence and write it in reverse in UPPER CASE
                  ; P.D. Terry, Rhodes University, 2014
                  ; char[] sentence is v0; leng is v1

 0 DSP    2   ;                              33 LDXA       ;
 2 LDA    0   ;                              34 LDV        ;
 4 LDC    256 ;                              35 LDC    46  ;
 6 ANEW       ;                              37 CEQ        ;
 7 STO        ;  sentence = new char[256];   38 BZE    13  ; until (sentence[leng-1] = '.');
 8 LDA    1   ;                              40 LDA    1   ;
10 LDC    0   ;                              42 LDV        ;
12 STO        ;  leng = 0;                   43 LDC    0   ;
13 LDA    0   ;  repeat {                    45 CGT        ; while (leng > 0) {
15 LDV        ;                              46 BZE    63  ;
16 LDA    1   ;                              48 LDA    1   ;
18 LDV        ;                              50 DEC        ;   leng--;
19 LDXA       ;                              51 LDA    0   ;
20 INPC       ;    read(sentence[leng]);     53 LDV        ;
21 LDA    1   ;                              54 LDA    1   ;
23 INC        ;    leng++;                   56 LDV        ;
24 LDA    0   ;  }                           57 LDXA       ;
26 LDV        ;                              58 LDV        ;
27 LDA    1   ;                              59 CAP        ;
29 LDV        ;                              60 PRNC       ;   write(upper(sentence[leng]);
30 LDC    1   ;                              61 BRN    40  ; }
32 SUB        ;                              63 HALT       ; System.Exit()
```

Task 8 - Improving the opcode set

This is straightforward, if a little tedious, and it is easy to leave some of the changes out and get a corrupted solution. The `PVMAsm` class requires modification in the *switch* statement that recognizes two-word opcodes:

```
    case PVM.brn:                           // all require numeric address field
    ...
    case PVM.ldc:
    case PVM.ldl:  // +++++++++++++++++  addition
    case PVM.stl:  // +++++++++++++++++  addition
      codeLen = (codeLen + 1) % PVM.memSize;
      if (ch == '\n')                       // no field could be found
        error("Missing address", codeLen);
      else {                                // unpack it and store
        PVM.mem[codeLen] = src.readInt();
        if (src.error()) error("Bad address", codeLen);
      }
      break;
```

The `PVM` class requires several additions. We must add to the enumeration of the machine opcodes:

```
    public static final int // Machine opcodes
      ...
      ldl    = 63,     // +++++++++++++++++  additions
      stl    = 64,
      lda_0  = 65,
      ...
```

We must add to the *switch* statement in the `trace` method (several submissions missed this):

```
    static void trace(OutFile results, int pcNow) {
      switch (cpu.ir) {
        ...
        case PVM.ldl:  // +++++++++++++++++  addition
        case PVM.stl:  // +++++++++++++++++  addition
      }
      results.writeLine();
    }
```

and we must provide case arms for all the new opcodes. A selection of these follows; the rest can be seen in the solution kit. Notice that for consistency all the "inBounds" checks should be performed on the new opcodes too (several submissions missed this).

```
        case PVM.ldc_m1:        // push constant -1
          push(-1);
          break;
        case PVM.ldc_0:         // push constant 0
          push(0);
          break;
        case PVM.ldc_1:         // push constant 1
          push(1);
          break;
        ...

        case PVM.lda_0:         // push local address 0
          adr = cpu.fp - 1;
          if (inBounds(adr)) push(adr);
          break;
        case PVM.lda_1:         // push local address 1
          adr = cpu.fp - 2;
          if (inBounds(adr)) push(adr);
          break;
        ...

        case PVM.ldl:           // push local value
          adr = cpu.fp - 1 - next();
          if (inBounds(adr)) push(mem[adr]);
          break;
        case PVM.ldl_0:         // push value of local variable 0
          adr = cpu.fp - 1;
          if (inBounds(adr)) push(mem[adr]);
          break;
        case PVM.ldl_1:         // push value of local variable 1
          adr = cpu.fp - 2;
          if (inBounds(adr)) push(mem[adr]);
          break;
        ...

        case PVM.stl:           // store local value
          adr = cpu.fp - 1 - next();
          if (inBounds(adr)) mem[adr] = pop();
          break;
        case PVM.stl_0:         // pop to local variable 0
          adr = cpu.fp - 1;
          if (inBounds(adr)) mem[adr] = pop();
          break;
        case PVM.stl_1:         // pop to local variable 1
          adr = cpu.fp - 2;
          if (inBounds(adr)) mem[adr] = pop();
          break;
```

We must add to the method that lists out the code (several submissions missed this). :

```
    public static void listCode(String fileName, int codeLen) {
        ...
        case PVM.brn:
        case PVM.ldc:
        case PVM.ldl: // +++++++++++++++++  addition
        case PVM.stl: // +++++++++++++++++  addition
          i = (i + 1) % memSize; codeFile.write(mem[i]);
          break;
```

Finally we must add to the section that initializes the mnemonic lookup table:

```
    public static void init() {
      ...
      mnemonics[PVM.ldl]    = "LDL";    // +++++++++++++++++  additions
      mnemonics[PVM.stl]    = "STL";
      mnemonics[PVM.lda_0]  = "LDA_0";
      ...
```

As an example of using the new opcodes, here is the Factorial program recoded in considerably fewer operations. Some submissions only used some of the new opcodes, ignoring the `INC, DEC` and `STL` ones, for example.

```
 0 DSP    3       ; n is v0, f is v1, i is v3
 2 LDC_1
 3 STL_0          ; n = 1;
 4 LDL_0
 5 LDC    20      ; // max = 20, constant
 7 CLE            ; while (n <= max) {
 8 BZE    39
10 LDC_1
11 STL_1          ;   f = 1;
12 LDL_0
13 STL_2          ;   i = n;
14 LDL_2
15 LDC_0
16 CGT            ;   while (i > 0) {
17 BZE    27
19 LDL_1
20 LDL_2
21 MUL
22 STL_1          ;     f = f * i;
23 LDA_2
24 DEC            ;     i--;
25 BRN    14      ;     i = i = 1;
27 LDL_0
28 PRNI           ;   write(n);
29 PRNS   "! = "  ;   write("! = ");
31 LDL_1
32 PRNI           ;   write(f);
33 PRNS   "\n"    ;   write("\n") (or use PRNL)
35 LDA_0
36 INC            ;   n++;
37 BRN    4
39 HALT           ;   n = n + 1;
```

and here is `SENTENCE1.PVM`, which uses 40 words of memory, compared with 63 for `SENTENCE.PVM`.

```
                ; Read a sentence and write it in reverse in UPPER CASE
                ; P.D. Terry, Rhodes University, 2014
                ; char[] sentence is v0; leng is v1

 0 DSP      2     ;
 2 LDC      256   ;
 4 ANEW           ;  sentence = new char[256];
 5 STL_0          ;
 6 LDC_0          ;
 7 STL_1          ;  leng = 0;
 8 LDL_0          ;  repeat {
 9 LDL_1          ;
10 LDXA           ;
11 INPC           ;    read(sentence[leng]);
12 LDA_1          ;
13 INC            ;    leng++;
14 LDL_0          ;  }
15 LDL_1          ;
16 LDC_1          ;
17 SUB            ;
18 LDXA           ;
19 LDV            ;
20 LDC      46    ;
22 CEQ            ;
23 BZE      8     ; until (sentence[leng-1] = '.');
25 LDL_1          ;
26 LDC_0          ;
27 CGT            ; while (leng > 0) {
28 BZE      40    ;
30 LDA_1          ;
31 DEC            ;   leng--;
32 LDL_0          ;
33 LDL_1          ;
34 LDXA           ;
35 LDV            ;
36 CAP            ;
37 PRNC           ;   write(upper(sentence[leng]);
38 BRN      25    ; }
40 HALT           ; System.Exit();
```

(The code for `STUDENTS1.PVM` can be found in detail in the solution kit.)

Task 9 - Do "improvements" necessarily make things "better"?

Surprisingly, no. In the prac worksheet the suggestion was made that you study the original source to see that the original opcodes had been mapped onto the numbers 30 .. 62. This meant that you could map the new opcodes onto a set of numbers below 30, or above 62. In the prac solution kit you will find four versions of the interpreter in which this has been done.

The following table shows various timings obtained on the four systems for two encodings of the infamous Sieve of Eratosthenes, differing only in that one used the compact opcodes where possible. The behaviour is quite remarkable. The optimized opcode set resulted in the execution of about 33% fewer instructions over counts running into millions, and **when the optimized opcodes were mapped onto "high" internal numbers the overall execution speed improved to about 84%. However, when mapped onto low numbers the code using the unoptimized opcode set took far longer to run, while that using the optimized opcode set slightly less time to run**. Since the only difference in the source code of the **interpreter** was to be found in this numerical mapping, one is forced to conclude that the underlying implementation of the large switch statement plays a key role in the performance one can expect. Several submissions suggested that the differences could be explained away by the longer list of opcodes and the (relatively) slow lookup process that forms the basis of the `opCode` method in the `PVM.java` file (at least, that is what I think the authors were trying to say; some explanations were very badly expressed!). But this has nothing to do with it - that method is used by the *assembly* process when the source code is read in, and not at all by the *interpretation/execution* process when the program is "run".

In a really serious implementation of an interpreter it would be worth carrying out further experiments to determine the optimal mapping, based, for example, on benchmarks carried out on a variety of programs. (These timings were done fairly roughly on a stopwatch; one should really have run the simulations many times over and for higher numbers of iterations, but the effects show up readily enough.)

Only one team came up with any suggestions for how the interpreter could be improved still further. This can be done in various ways, for example by "inlining" the code that is currently executed by calls to the `next`, `push` and `pop` routines, and it was disappointing that nobody bothered to try this. Of course it means quite a lot of changes have to be made. The solution kits show this in detail.

```
Java - 1000 iterations, 4000 upper limit, times in seconds (Win XP, 3GHz machine)

                              S1.PVM       S2.PVM

Opcode set                    Limited      Extended
High numbers                    6.26          5.24     (84%)
Low numbers                     8.99          4.74     (53%)
High numbers, checks removed    2.71          3.65     (134%)
Low numbers, checks removed     5.70          3.16     (60%)

Operations                  394,334,033   263,191,026  (67%)
```

The "checks removed" figures were obtained using variations of the interpreter source in which all the checks that CPU.SP remained in bounds had been suppressed, as well as the calls to next, push and pop (their effect was achieved by "inlining" the equivalent code. One can see that an insistence on safety results in a considerable loss of run-time efficiency.

I ran the simulations again using C# implementations of the system - the source code is to all intents and purposes identical:

```
C# - 1000 iterations, 4000 upper limit, times in seconds (Win XP, 3GHz machine)

                              S1.PVM       S2.PVM

Opcode set                    Limited      Extended
High numbers                    9.55          6.78     (71%)
Low numbers                     9.73          7.02     (72%)
High numbers, checks removed    3.95          1.99     (49%)
Low numbers, checks removed     4.31          2.25     (52%)
Operations                  394,334,033   263,191,026  (67%)
```

Interestingly, the C# system is sometime "slower" and sometimes "faster" than the Java one, and there is less variation in timing between the "high" and "low" number mappings of the opcodes. The extended opcode set always resulted in shorter times.

Great minds think alike - "Make it as simple as you can, but no simpler"