Computer Science 3 - 2012

Programming Language Translation

Practical for Week 26, beginning 22 October 2012

This practical is designed to explore Michael Andersen's PAM computer (and possibly help find bugs and flaws in it). PAM is an acronym for "Parva Actual Machine", just as PVM is an acronym for "Parva Virtual Machine".

This has been an interesting project, and I am very grateful to Michael for his initiative, for the work he has done and the fact that we (and future classes) will be able to benefit from it. You may have noticed that the "magic numbers" associated with the set of PVM opcodes you have needed have some omissions, and don't form a simple sequence. This is because the PAM has some extra opcodes that we do have not needed in this practical.

There is nothing to hand in, although any contributions or examples of code will be appreciated.

Objectives:

In this practical you are to

- familiarize yourself with the PAM computer, a small single board computer that is microcoded to execute PVM code directly.
- Test the extended Parva compiler by compiling some small programs and then interpreting them as usual or executing them on the PAM computer.
- Extend the Parva Compiler to allow one to access the hardware features of the PAM (the dip switches and LEDs in particular)

This prac sheet is also available at `http://www.cs.ru.ac.za/courses/CSc301/Translators/trans.htm`.

Outcomes:

When you have completed this practical you should

- understand a little more about machine architecture;
- have had some fun;
- have helped contribute to the future use of PAM in CSC 301.

Task 1 - Create a working directory and unpack the prac kit

The PAM system is currently available only in a Java version. As usual, there are several files that you need, zipped up in the file `PRAC26.ZIP` (Java).

- Immediately after logging on, get to the command line level by using the `Start -> All Programs -> Accessories -> Command prompt` option from the tool bar.
- Copy the prac kit into a newly created directory/folder in your file space

```
J:
md  prac26
cd  prac26
copy  i:\csc301\trans\prac26.zip
unzip  prac26.zip
```

  This will create several other directories "below" the `prac26` directory, including:

```
J:\prac26\library
J:\prac26\Parva
J:\prac26\Native
J:\prac26\Examples
```

containing the Java classes for the I/O library, and for the code generator and symbol table handler.

You will also find the executable version of Coco/R and batch files for running it, frame files, and various sample data, code and the Parva grammar, contained in files with extensions like

`*.ATG,     *.PAV    *.JAR`   etc

- As usual, you can use the `CMAKE` and `CRUN` commands to build and run the compiler. The kit also supplies a `PARVA.BAT` file to allow you to give a command like `Parva voter.pav` more easily than by using `CRUN`. There is also a `RUN.BAT` file which allows you to run a precompiled version of the extended Parva system, which you can use to run some of the examples before you modify the `PARVA.ATG` and other support files yourself .

Task 2 - Try out the system with the precompiled version

The `Parva.frame` file in the kit has been extended to allow you the choice of either interpreting a Parva program after compilation, as usual, or of downloading the memory image to the PAM and running it there.

The relevant part of the frame file will show you how this is achieved

```
do {
  System.err.print("\n\nI)nterpret R)un or Q)uit ");
  reply = (InFile.StdIn.readLine() + " ").toUpperCase().charAt(0);
  if (reply == 'I')
    PVM.interpret(codeLength, initSP);
  else if (reply == 'R') {
    if (!loaded) {
      NativeLink.initNatives();                        // Initialise the libraries we are using

      ArrayList<Integer> codes = new ArrayList<Integer>(); // Build up the list of codes
      for (int i = 0; i < codeLength; i++)
        codes.add(PVM.mem[i]);

      ProgramImage pi = new ProgramImage();            // Copy the codes to the PAM
      pi.writeCodes(0, codes);
      pi.programToDevice();

      System.err.println("Loaded");
      loaded = true;                                   // So we can rerun without reloading
    }

    InteractiveConsole ic = new InteractiveConsole();
    System.err.println("Running");
    ic.startBlocking();
  } // reply = 'R'
} while (reply != 'Q');
```

However, PAM requires that the strings like "Hello world" have been loaded into memory above the code, rather than in the old stringpool at the top of memory. The system in the kit does not do this. That was the last task in Practical 25, as you are reminded below, which you may not have completed yet.

If your Parva programs do not try to output any strings the PAM will be able to run them quite easily, however.

To get started, it is suggested that you try the precompiled version of a fairly complete system with commands like

```
RUN first.pav       (simply writes out the first 100 integers)
RUN voter.pav       (from the text book - finds the average age of som
RUN heapstr.pav     (simple greeting program)
```

where, for example, the Parva code for `first.pav` reads

```
void main() { $C+
  int i = 0;
  while (i <= 100) {
    write(i);
    i = i + 1;
  }
}
```

Task 3 - Learn about the hardware gadgets

Like other machines in its class, the PAM has some special memory addresses that allow one to read or alter the state of various devices "located" at those addresses. From Michael's online documentation:

Hardware Abstraction Layer Space

The memory from 0x6000 to 0x600B is occupied by virtual registers that can control the hardware. These registers are:

```
0x6000 : Blue LEDs
0x6001 : Red in RGB LED
0x6002 : Green in RGB LED
0x6003 : Blue in RGB LED
0x6004 : DIP Switches
0x6005 : Push buttons (ABC)
0x6006 : LDR value (not implemented)
0x6007 : Temperature sensor (not implemented)
0x6008 : Delay X jiffies (a jiffy is about 10ms)
0x6009 : Interrupt vector for button A
0x6010 : Interrupt vector for button B
0x6011 : Interrupt vector for button C
```

They can only be written to using a 'sto' instruction and can only be read from using an 'ldv' instruction.

Here is an example of a Parva program that reads the state of the switches and reflects them on the LEDs

```
void main() { $C+
// Read the dip switches and display the readings on the leds of the PAM
// Clearly this does not work on the PVM

  const DIP = 06004H;
  const LED = 06000H;

  while (peek(DIP) != 0)
    poke(LED, peek(DIP));
}
```

Try out this program:

```
RUN readdip.pav
```

Interpreting it will achieve very little (although you can, if you are curious!)

This code has assumed the existence of a new possibility for a *Factor* - `peek(address)` is to push the word found at the absolute `address` in memory onto the evaluation stack of the PAM/PVM, while `poke(address, value)` is a new variation on *Statement* that will pop the stack and store the `value` at the absolute `address` specified as the second argument.

It is not particularly difficult to add these extensions to the Parva compiler - the existing opcode set is all you need (though you will have to be creative in how you use it!).

Here is another silly program that you could try out

```
void main() { $C+
// Read the first 3 words in memory
  writeLine(peek(0));
  writeLine(peek(1));
  writeLine(peek(2));
}
```

and perhaps you can extend this to look around at other memory in a new light <grin>

As a slightly more interesting exercise, see if you can write a program that write out the first 100 numbers, not to the screen, but to the LED display (hint: you will have to build in a delay as you go round the loop, or the program will just produce a blur.

as another exercise, you could try changing the colour of the RGB LED as well. The possibilities are endless.

Task 4 - Help debug PAM

The precompiled version of the extended Parva system for PAM incorporates several of the features you should have added to the system for yourself, as well as a set of examples.

Not all of these will run, but we would be grateful if you could try as many as possible, and see whether you find any problems that might relate to PAM itself.

You could patch in some of your own solutions to Prac 25 into the Parva.atg file, of course, and similarly, modify the code generator and interpreter.

And, of course, do your own thing and invent some more examples for us!

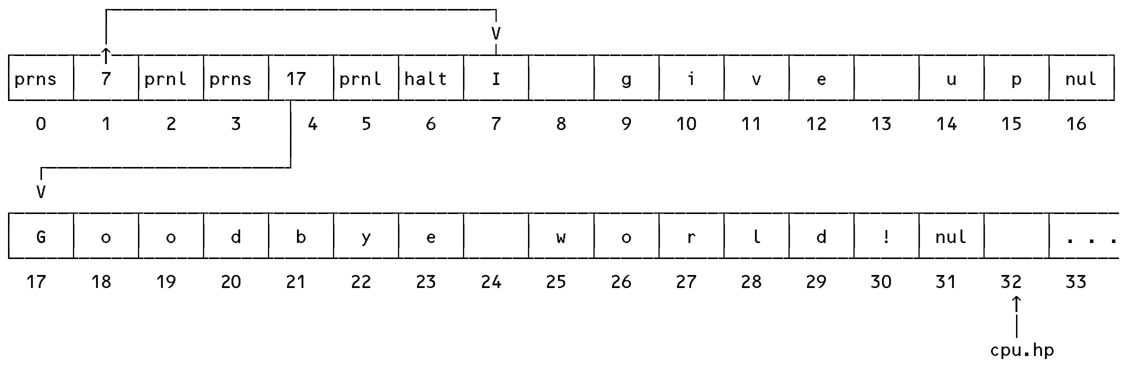
Practical 25, Task 15 - Let's untangle that heap of string

By now you should be well aware that any strings encountered in a Parva program are stacked at the top end of memory whence they are extracted by the PRNS opcode in the PVM.

There is an alternative model, one that is used in the PAM (Parva Actual Machine). In this model the strings are stored above the program code - effectively in space that the PVM would otherwise have used for its heap. A very simple program like

```
void main () {
// A fatalistic approach to the oncoming examinations?
  writeLine("I give up");
  writeLine("Goodbye world!");
}
```

would be stored in memory like this



Modify the code generator and the PVM to use this model.

Hint: You might like to associate a label with each PRNS operation, since at the time you want to generate the opcode you won't know exactly where the string will be stored. If you like developing tight code suited to a small machine like the PAM, you might think of optimising the analysis and code generation so that if the same string appears in the program in two or more places, only one copy is loaded in memory.