Computer Science 3 - 2013

Programming Language Translation

Practical for Week 26, beginning 21 October 2013

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, Alan Herbert and Sean Pennefather have 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 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 you 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

A version of Parva with auxiliary classes for connecting to the PAM system is available in Java and C# versions. These kits are essentially the sames as were supplied to you last week, with two modifications - string are handled differently, and the system can scan decimal, binary and hexadecimal constants. You can use this kit "as is", but if you wish you can make a few modifications to your own Parva system so that you can try out programs that use the extnsions suggested in last weeks practical. A summary of these changes - which you can make by suitably cutting and pasting from a supplied file into the various components of your system - is given as an appendix. As usual, there are several files that you can use to build a working system, zipped up in the file `PRAC26.ZIP` (Java) or `PRAC26C.ZIP` (C#).

- Immediately after logging on, get to the command line level by using the `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 ones like:

```
J:\prac26\library
J:\prac26\Parva
J:\prac26\Native
J:\prac26\Pamflit
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`.

Task 2 - Try out the system with a simple program

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);
      } // reply = 'I'
      else if (reply == 'R') {
        if (!loaded) {
          loaded = true;                              // So we can rerun without reloading
          NativeLink.initNatives();                   // Initialise the libraries we are using
          ProgramImage pi = new ProgramImage();       // Copy the codes to the PAM
          pi.writeCodes(0, codes);
          pi.programToDevice();
          System.err.println("Program Loaded");
        }
        System.err.println("Initialising...");
        InteractiveConsole ic = new InteractiveConsole();
        ic.startBlocking();
      } // reply = 'R'
    } while (reply != 'Q');
  }
```

Don't be over ambitious at the start. CMAKE the compiler and then run it with a command like

```
PARVA  first.pav
```

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

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

Use the interpreter and also the PAM computer. Get the demonstrators to help you at the start.

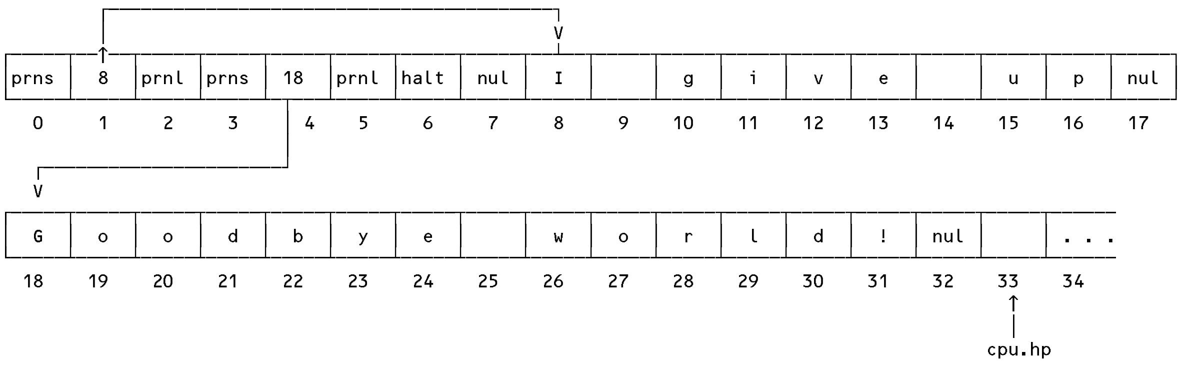
Task 3 - Let's untangle that heap of string

By now you should be well aware that any strings encountered in a Parva program have been stacked at the top end of memory, whence they could be retrieved by the PRNS opcode in the PVM interpreter.

There is an alternative model, one that is used in the PAM. 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 forthcoming examinations?
  writeLine("I give up");
  writeLine("Goodbye world!");
}
```

would be stored in memory like this



The wy in which this is achieved can be seen in the modified code generator and interpreter in the kit. The system tries to "optimize" - if a string appears in the source code more than once, only one copy of the string is patched into the "object code". Perhaps the PVM should have been defined like this when it was first invented in 2001!

Try running a program like this and make sure that if you have modified your own compiler, that you have done so correctly. Once again, use the PVM interpreter and also the PAM hardware. Here's another one: try to think of a way to show how duplicates are really handled.

```
void main () {
// heapstr.pav
// Strings on the heap (how do we check that we eliminate duplicates?)
  writeLine("I give up");
  writeLine("Goodbye cruel world - I am off to join the circus");  // pop tune of my era
  writeLine("Are you sure about that?");
  writeLine("Quite sure - ", "I give up", ", I said " , "I give up");
}
```

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
    0x600A : Interrupt vector for button B
    0x600B : 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 DIP 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 - compile it and then run it on the PAM. 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 meant 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 Profound Program that you could try out:

```
void main() { $C+
// Read the first 3 words in memory and display them
  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, so have fun.

Task 4 - Help debug PAM

The kit contains various other test programs, and last week you will probably have written some of your own. Probably 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 in the kit, of course, and similarly, modify the code generator and interpreter.

And, better still, do your own thing and invent some more examples for us! Last year Sean Pennefather came up with nice examples that show how to handle the interrupts that can be generated when you press the buttons on the PAM motherboard. They are in the kit - speak to Sean to discuss these further.