Apoo: an Environment for a First Course in Assembly Language Programming

Rogério Reis   Nelma Moreira

Technical Report Series: DCC-98-9

Departamento de Ciência de Computadores – Faculdade de Ciências

&

Laboratório de Inteligência Artificial e Ciência de Computadores

Universidade do Porto

Rua do Campo Alegre, 823 4150 Porto, Portugal

Tel: +351+2+6078830 – Fax: +351+2+6003654

http://www.ncc.up.pt/fcup/DCC/Pubs/treports.html

Apoo: an Environment for a First Course in Assembly Language Programming

Rogério Reis Nelma Moreira
{rvr,nam}@ncc.up.pt
DCC-FC & LIACC, Universidade do Porto
R. do Campo Alegre 823, 4150 Porto, Portugal

November 1998

Abstract

Teaching the very basic concepts of a computer architecture, instruction set and operation based on a real microprocessor is usually an unfruitful task as the essential notions are obscured by the specific details of its architecture. A machine emulator has the benefit of providing a portable environment that can be run in several platforms and that can be easily adapted for pedagogical purposes. **Apoo**[1] is a virtual machine with a very simple architecture and instruction set that mimics almost all the essential features of a modern microprocessor. The **Apoo Interface** is an graphical environment that monitors the state of the machine during the execution of a program and allows the writing/editing/execution of programs in assembly language. The **Apoo Tutor** is a module that aims the grading of student programs based on a description of what should be the execution of the program for specified input data sets. This module was used to evaluate students programming skills in an interactive learning/grading system, not only allowing the detection of errors but trying to give extra information in order that the student could understand and correct his/her mistakes more easily.

1 Introduction

In the past few years at the Faculty of Sciences of the University of Porto, at an introductory course of Computer Science (for Computer Science majors) we teached the **MC68000** assembly language. Initially using **MC68000** based computers, then, when they become too obsolete and thus impossible to maintain we turned to the **e68k**[2], an emulator of the **MC68000** processor with the obvious benefits of robustness and tracing facilities [Fil92][3]. Nevertheless the actual architecture of the **MC68000** with its 14 address modes and more than 50 mnemonics[4] was still in the way of the first year students mind to grasp the essentials of computing programming. Issue was not to update the model micro-processor to a more recent architecture like **MIPS R2000** as it is done by the **SPIM** emulator [PH94], but to create an abstract processor for teaching purposes only, in the spirit of the **MIX** [Knu73] or the **SCRAM** [Dew93]. **MIX** is too archaic and cumbersome, and the **SCRAM** although used to explain the circuit layout of a micro-processor, has not the expressiveness required to teach basic algorithms (easily).

[1]Although **Apoo** could be an achronym for an Abstract Processor with an Original Orientation, in fact it is just the name of a yellowish vagabond cat that keeps poking at our back door.

[2]Available at `http://www.ncc.up.pt/~mig/e68k`

[3]A graphical interface is also available in `http://www.ncc.up.pt/~nam/Tke68k`

[4]And the e68k implementation is notably accurate.

This led us to design a simple assembly language with special emphasis on flow control and basic data manipulation, with an user friendly environment. The implementation language chosen was Python[Lut96] basically because of the following reasons:

- as a very high level object oriented language, the code for **Apoo** and **Apoo Tutor** is only about 20K bytes long;
- all the code of the virtual machine could be written in such a way that the introduction of new machine instructions is a matter of writing only a couple of lines work (literally);
- it is highly portable (it worked without any modification in a *Windows 95*);
- it provided an interface to the `Tk` graphical toolkit, that allowed the easy building of a graphical environment;
- it's neat...

In the next section, the basic architecture of the virtual processor and the new assembly language are described, as well as some implementation details. Section 3 introduces the graphical environment and its functionalities. In Section 4, the tutor module and its integration in a grading system is discussed. Some comments on the practical experience with our students and future work are sketched in Section 5.

2 Apoo Virtual Machine

Apoo has a set of general purpose registers[5] (8 in the default configuration), a data memory area, a program memory area, a system stack and a program counter register.

Each register or memory cell can hold a 32-bits[6] integer. Registers are named `Ri`, where `i` ranges from `0` up to the number of registers minus 1.

Memory cells are created as needed by means of two different pseudo-instructions, as tabled in Figure 1:

- the `mem` pseudo-instruction reserves an array of cells;
- the `const` pseudo-instruction reserves individual cells initializing them with the given values.

Every data memory address referred must have been reserved by one of the previous pseudo-instructions.

As **Apoo** is a virtual machine, which is emulated, and the aim was to teach assembly language, there is no machine code associated with its instruction set. Each program memory cell will hold a whole instruction. In particular, a program in memory will begin in program memory address `0`. The program counter, as usual, will contain the address of the next instruction to be executed, i.e., the number of the next instruction.

Finally, the system stack is used to implement subroutines and argument passing.

2.1 The Assembly Language

Each assembly instruction has the following form:

```
(<Label>:) Operation <Operand1> <Operand2>
```

where

Label is any string of letters or digits; if present, must begin in the first column of a line, and will be associated, as usual, with the address of the instruction in that line.

Operation a mnemonic of its functionality.

[5] The number of available registers, is set when a new instance of the virtual processor is created.

[6] As that is the precision of integers in Python

Pseudo-instruction	Meaning
(`Label:`) `mem Num`	Reserves `Num` memory cells and associates `Label` (if present) with the address of the first cell
`Label: const Num1`	Reserves `1` memory cell whose address associates with `Label` and whose contents is `Num1`
`const Num2`	Reserves `1` memory cell whose contents is `Num2`
⋮	

Figure 1: Pseudo-instructions

Operands can be numeric (data or addresses), labels or registers (`R0`...`R(`$nregs-1$`)`).

A line beginning with a `#` is ignored by the parser; so it can be used to write *comments* in the program.

The instructions of the **Apoo** assembly language are listed in Figure 2. They can be divided in three categories: data transfer, arithmetic and program control.

In data transfer instructions, the first operand refers to the source and the second to the destination. This are addressing modes used:

Immediate the first operand is a constant:

Ex.: `loadn 100 R1`

Direct both operands are registers or memory cell addresses:

- **memory-register** Ex.: `load 100 R1`
- **register-memory** Ex.: `store R1 100`
- **register-register** Ex.: `storer R1 R2`

Indirect the effective address is the contents of the source (respectively destination) operand

- **memory-register** Ex.: `loadi R2 R1`
- **register-memory** Ex.: `storei R1 R2`

In all the arithmetic instructions the operands can only be registers.

Example 1 *The following program loads the values* `3`*,* `4` *and* `5` *in registers* `R1`*,* `R2` *and* `R3`*, respectively, adds its contents and stores the result in memory cell whose address is* `3`*:*

```
loadn   3     R1
loadn   4     R2
loadn   5     R3
add     R1    R2
add     R2    R3
store   R3    3
```

•

The control flow instructions change the value of the program counter (`PC`), in order to alter the normal flow of the program. They are divided in the following groups:

Control Transfer

Unconditional are always executed.

Ex.: `jump 100`

Conditional its operation is conditioned by the value of a register.

Ex.: `jzero R1 100`

Subroutines

Call to subroutine Ex.: `jsr 100`

Return from subroutine Ex.: `rtn`

Save and retrieve arguments in the system stack These instructions can also be used anywhere else...

Ex.: `push R1`

Ex.: `pop R1`.

Halt halts the execution of a program. It must always be present in a program.

And that is all!!!

Let us see some examples of trivial **Apoo** programs.

Example 2 *This program adds the `n` values stored in memory starting at memory cell labeled `Ni`, saving the result in the cell labelled `Patang`.*

```
Ni:     const   3
        const   4
        const   5
n:      const   3
Patang: mem     1

begin:  zero    R4
        loadn   Ni      R1
        load    n       R2
loop:   jzero   R2      end
        loadi   R1      R4
        add     R4      R3
        dec     R2
        inc     R1
        jump    loop
end:    store   R3      Patang
```

•

Example 3 *The following program calls a subroutine that finds the maximum value of a sequence of values stored in memory.*

```
N:      const   3
val:    const   7
        const   8
        const   20
MAX:    mem     1
        #R4=maximum; R2=position
        load    N       R1
        loadn   val     R2
        #R1=number of values
        #R2=beginning of the sequence
        jsr     max
        store   R4      MAX
        halt
```

```
max:    loadi   R2      R4
loop:   inc     R2
        dec     R1
        jzero   R1      cont
        #loads in  R3 the next value
        loadi   R2      R3
        #makes a copy in order to compare with the  maximum
        storer  R3      R5
        sub     R4      R5
        #if R4-R5 > 0 continue
        jpos    R5      loop
        #stores in  the new maximum
        storer  R3      R4
        jump    loop
cont:   rtn
```

•

In appendix A a more detailed explanation of each instruction can be found.

2.2 Implementation

The virtual machine is implemented as a Python class named `Vpu`. The main attributes are:

`nreg` the number of registers.

`reg` the list of internal registers.

`RAM` the memory data cells, the address of each cell corresponds to its position on this list.

`PC` the program counter.

`stack` the system stack.

`labelp` a dictionary that associates program labels with program instruction numbers.

`labelm` a dictionary that associates memory data labels with memory addresses.

`code` a dictionary that associates each mnemonic with the corresponding code written in Python. For instance, the mnemonic `load` has the following `code` associated:

```
if type(A1) == type(''):
  try: add = self.labelm[A1]
  except KeyError: raise LabelError
else: add = A1
try: foo = RAM[add]
except IndexError: raise OutOfMemory(add)
Reg[A2] = foo
incPC()
```

where `A1` and `A2` stand for the operands of the operation. First it is tested if the first operand is a label. If that is the case, the associated memory address is retrieved. The contents of the associated memory cell is then fetched and, if no error occurs, its value stored in the register which number is given by the second argument.

`Prog` the list of instructions of the program in memory.

`BreakP` a list of the instructions numbers which have a breakpoint set in.

`time` the amount of time used by the execution until now.

Operation	Operand1	Operand2	Meaning
Data to Register Transfer			
`load`	`Mem`	`Ri`	Loads the contents of memory address `Mem` into register `Ri`; Mem can be a label
`loadn`	`Num`	`Ri`	Loads number `Num` into register `Ri`; `Num` can be a label, in which case it represents an address
`loadi`	`Ri`	`Rj`	Loads the contents of memory whose address is the contents of `Ri` into `Rj` (indirect load)
Data to Memory Transfer			
`store`	`Ri`	`Mem`	Stores the contents of `Ri` at memory address `Mem`; `Mem` can be a label
`storer`	`Ri`	`Rj`	Stores the contents of `Ri` into `Rj`
`storei`	`Ri`	`Rj`	Stores the contents of `Ri` into memory cell whose address is the contents of `Rj`
Data to the System Stack Transfer			
`push`	`Ri`		Pushes the contents of `Ri` into the top of the stack
`pop`	`Ri`		Pops the element from the top of the stack into `Ri`
Two Operand Arithmetic			
`add`	`Ri`	`Rj`	Add the contents of register `Ri` to the contents of register `Rj` and stores the result in `Rj` (`Rj=Ri+Rj`)
`sub`	`Ri`	`Rj`	Subtracts the contents of register `Rj` from the contents of register `Rj` and stores the result in `Rj` (`Rj=Ri-Rj`)
`mul`	`Ri`	`Rj`	Multiplies the contents of register `Ri` and the contents of register `Rj`, and stores the result in `Rj` (`Rj=Ri*Rj`)
`div`	`Ri`	`Rj`	Stores in `Rj` the quotient of integer division of the contents of register `Ri` by the contents of register `Rj` (`Rj=Ri/Rj`)
`mod`	`Ri`	`Rj`	Stores into `Rj` the rest of integer division of the contents of register `Ri` by the contents of register `Rj` (`Rj=Ri%Rj`)
One Operand Arithmetic			
`zero`	`Ri`		Stores 0 in `Ri` (`Ri=0`)
`inc`	`Ri`		Increments by `1` the contents of `Ri` (`Ri=Ri+1`)
`dec`	`Ri`		Decrements by `1` the contents of `Ri` (`Ri=Ri-1`)
Control Transfer			
`jump`	`Addr`		Jumps to instruction at address `Addr`; `Addr` can be a label
`jzero`	`Ri`	`Addr`	Jumps to instruction at address Addr, if the contents of `Ri` is zero; `Addr` can be a label
`jpos`	`Ri`	`Addr`	Jumps to instruction at address `Addr`, if the contents of `Ri` is positive; `Addr` can be a label
`jneg`	`Ri`	`Addr`	Jumps to instruction at address `Addr`, if the contents of `Ri` is negative; `Addr` can be a label
`jsr`	`Addr`		Pushes the contents of `PC` into the stack and jumps to instruction at address `Addr`
`rtn`			Pops an address from the top of the stack into the `PC`
`halt`			Stops execution; Every program must have this instruction in order to end gracefully, otherwise an `Out of Program` error will occur

Figure 2: **Apoo** Assembly Language

Loading a program The parsing process – `Load` – is accomplished in a single step. Each line of the program, unless it is a comment, is considered an instruction. If it has a label in it, the label is associated with a memory cell. If the operation is a pseudo-instruction `mem` or `const`, data memory cells are created accordingly (with `0`'s in the case of a `mem`) and the label is associated with the first memory address. If the operation is one of the admissible mnemonics, the label is associated with the current program instruction number.

Each instruction is parsed ensureing the correct number and type of its operands and stored in the `Prog` list along with its operands. Syntactic errors stop the whole process without trying to recover.

Running a program At the beginning, the program counter is set to `0`. Step by step, each instruction in the `Prog` list is executed, according to the `code` corresponding to its `mnemonic`. Currently, possible infinite loops are detected having a pre-defined limit for the number of steps in a program run. As it is explained in the next section, a program can run:

- until a `halt` instruction is reached;
- until a previously defined breakpoint is reached;
- until a run-time execution error occurs;
- in a "step-by-step" fashion.

The system timer, started in the beginning of the execution, is automatically stopped every time any of these events occur.

3 The Apoo Interface

Look at Figure 3 and Figure 4!

The **Apoo Interface** is an environment to programming in assembly language and to monitor the execution of the **Apoo** virtual machine, providing an easy way to write/edit/debug/execute **Apoo** assembly programs.

During the execution of a program, it shows the contents of the program counter, registers and memory data. In accordance with what was said concerning the program memory segment, the program in memory is displayed in assembly language (not in machine code).

To execute a program we must first either

- Enter in `Edit` mode and write its instructions.
- Open a text file with its instructions.

After that, we can try to `Load` it. If a parsing error occurs, we can enter `Edit` mode and correct it; the interface will show the text line in which the error occurred.

When the program is successfully loaded (into memory), it can be executed in three different ways:

Run executes the program, until a `halt` instruction is reached or a run-time execution error occurs; at the end the display of the state of the machine is updated(i.e., the values of the program counter, registers and memory data).

Step executes the next instruction and the the display of the state of the machine is updated.

Continue continues the execution until a *breakpoint* is reached.

In an instruction line, a breakpoint can be set or cleared as follows:

- to set a breakpoint: “Double-click” `Button1` that line (the foreground becomes green)
- to clear a breakpoint: “Double-click” `Button2` again (the foreground will return to black)

In `Edit` mode we can change the text code of a program or create a new one. To enter this mode one must press the `Edit` button or the `New` button. After editing we can `Save` or `SaveAs` the current edited text. We leave `Edit` mode by loading the program – pressing the `Load` button – or opening a new file.

4 Apoo Tutor

In order to evaluate student's programming skills in the **Apoo** assembly language, we designed a new module that given a program (made by a student) and a script, valuates the program by “executing” the script. This allows to test if the program has syntactic errors and if given some initial input values it correctly calculates the expected results. This module will not verify the correctness of the program against some specification, but will detect, locate and explain a set of possible errors. In this way, not only the grading process is simplified but also it can help the students in correcting their mistakes. As it will be apparent from the commands of a script file this kind of evaluation takes much benefit from the fact that we have full access to the interpreter of the language.

An example of a tutor script is shown in Figure 5. This module can be executed, from an UNIX shell (or by other program) with the following arguments:

```
$ vpu_tutor tutor-script apoo-program
```

A tutor script can have the following commands:

`load` Tries to load the program and reports syntactic errors, if any.

`initial` This command has the following syntax:

```
initial [label:num[,num]*]* [label;size]*
```

The first set of labels corresponds to memory cells defined with the `const` pseudo-instruction, and the second one to memory cells defined with the `mem` pseudo-instruction; `num` and `size` must be integers. This verifies if the program initializes that set of labels with the corresponding values.

`init` Attributes values, distinct from the initial ones, to memory cells and registers. It can be used several times in a script, for diferent sets of initial input values, for different runs. Its syntax is as follows:

```
[init [RI:num]* [label:num]*]
```

where `RI` stands for one of the registers and `num` for an integer.

`exec` This command tries to execute (`run`) the program in memory if used without argument.

```
exec [label]
```

Used with an argument, it tries to execute the subroutine with name `label`.

`final` This command verifies if the values of the referred memory cells or registers are as specified after an execution. It has the following syntax:

```
final [RI:num]* [label:num[,num]*]*
```

where `RI` is as before and `nums` are the values to be matched.

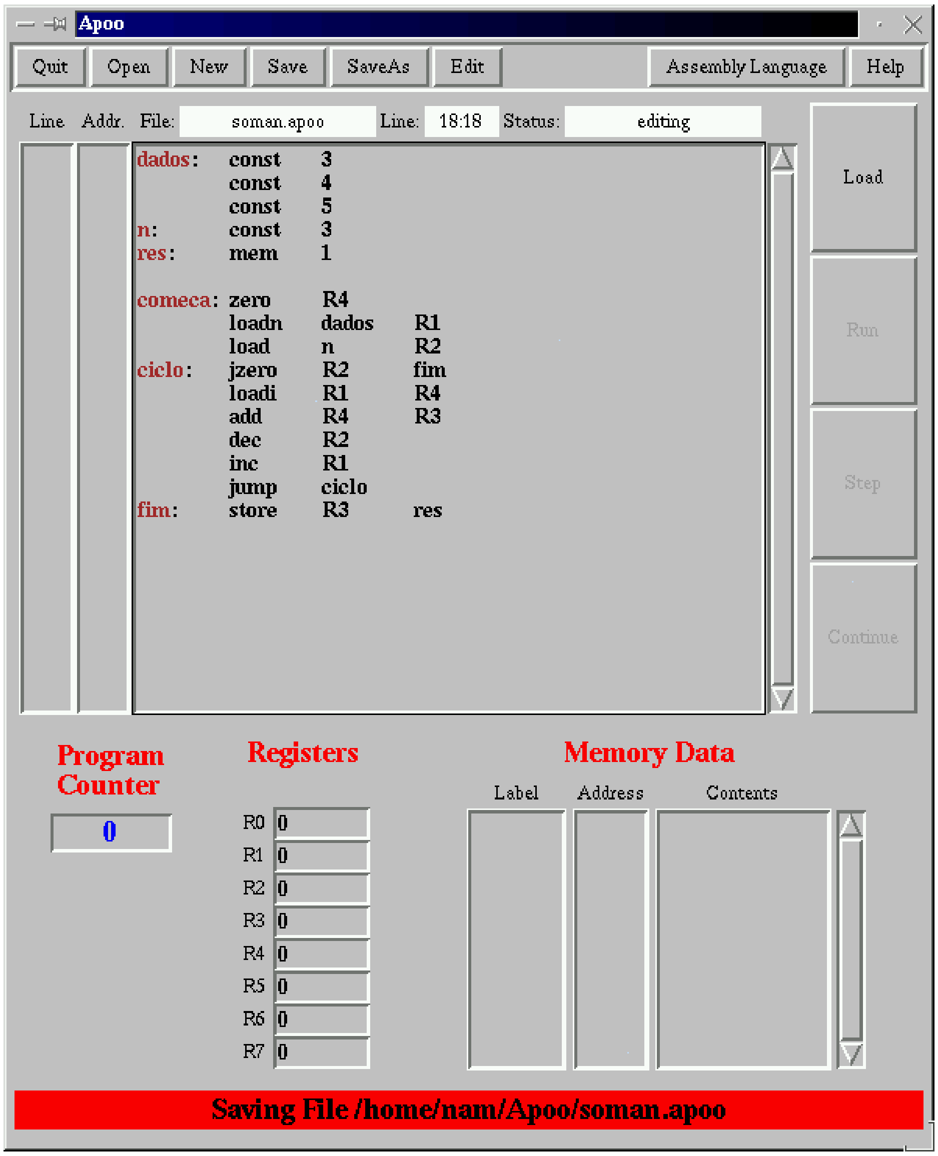


Figure 3: **Apoo Interface**

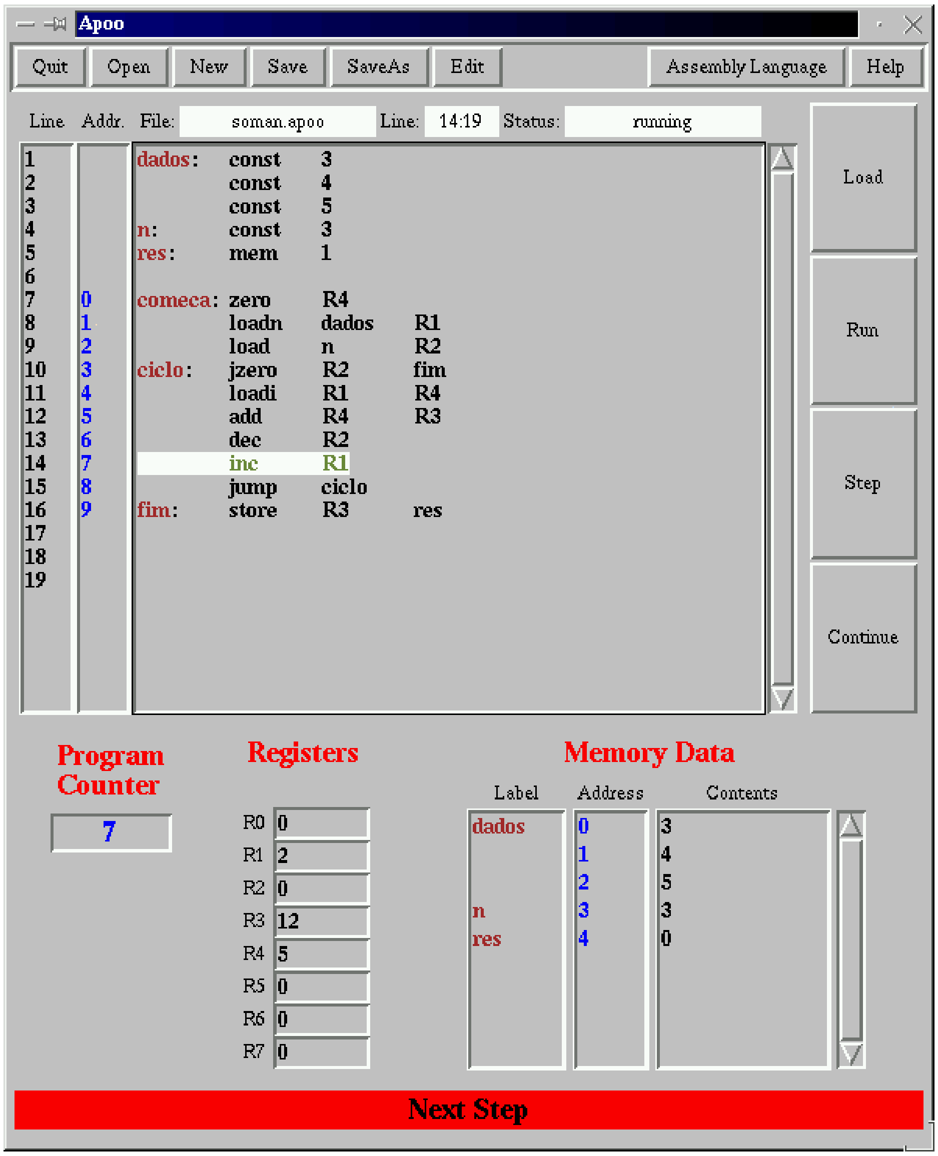


Figure 4: Running a Program

`value` This command must always have an integer as argument, that represents the grading of the previous command, if it was successful:

```
value num
```

`end` Marks the end of the script.

Whenever an error is detected or a `end` command is reached the tutor exits writing to the standard output the evaluated *grade* and a *message* reporting an error or the sucess in solving the problem.

```
load
value 20
initial a:1,-3,40,0 b;1
value 10
exec
value 20
final a:1,-3,40,0 b:38 R3:38
value 20
init a:9,10,10,0
exec
value 10
final b:39 R3:39
value 20
end
```

```
a:      const   1
        const   -3
        const   40
        const   0
b:      mem     1

        loadn   a       R1
        zero    R3
loop:   loadi   R1      R2
        jzero   R2      end
        add     R2      R3
        inc     R1
        jump    loop
end:    store   R3      b
        halt
```

Figure 5: An example of a tutor script and a correspondent correct program.

4.1 Integrated Grading System

The **Apoo Tutor** and the **Apoo Interface** were integrated in the Ganesh grading system [LM98], that is in development in our department.

This system provides, among other features, a **solving environment** that allows a student to be individually examined in a specific domain. In our case the domain is the **Apoo** assembly language. For each examination the student is given a set of problems to solve, by writing programs and submitting them for automatic grading. **Apoo Tutor** is called to grade the program and the student can debug and test his programs with the **Apoo Interface** (with the file functionalities disabled to avoid importing and exporting alien files).

During the examination, the student can submit the programs as many times as he want, until the problem is correctly solved (as far as the tutor script is concerned...) or the time for the examination expires.

5 Future Work

Taking in consideration the experience of the last year's course, in which only in 12 hours (lecture and practice time) were needed for the students to grasp all the important aspects of an assembly language like **Apoo**, it seems that the inclusion of logical and bitwise operations as well as different sized operands can be included in the *curriculum* of the course with great benefit. Thus low level implementation of acces to memory cells of different sizes is required for the inclusion of those type of instructions in the **Apoo** assembly language.

A **Apoo** machine code and its automatic generation and interpretation can be useful to give as an example, the same way as it is done for **SCRAM** in [Dew93].

Likewise, an abstract circuit layout for **Apoo** would be a neat pedagogical tool.

6 Acknowledgments

We would like to thank Sabine Broda and Zé Paulo Leal who were really β-testers of the system and kindly accepted to teach and grade the students with it. We are also very grateful to our students who were patient enough with the inevitable bugs.

References

[Dew93] A. K. Dewdney. *The (New) Turing Omnibus.* Computer Series Press. W.H. Freeman and Company, 1993.

[Fil92] Miguel Filgueiras. A Portable Environment for Programming in mc68000 Assembly. Technical report, Centro de Informática da Universidade do Porto, 1992.

[Knu73] Donald Knuth. *The Art of Computer Programming. Fundamental Algorithms*, volume 1. Addison-Wesley, second edition, 1973.

[LM98] José Paulo Leal and Nelma Moreira. Automatic Grading of Programming Exercices. Technical Report DCC-98-4, DCC-FC & LIACC, Universidade do Porto, July 1998.

[Lut96] Mark Lutz. *Programming Python.* Nutshell Handbook. O'Reilly & Associates, Inc, 1996.

[PH94] David A. Patterson and John L. Hennsyy. *Computer Organization & Design. The Hardware/Software Interface.* Morgan Kaufmann Publishers, 1994.

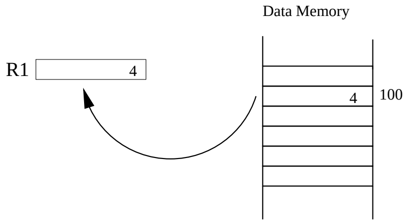
A Apoo Assembly Language

Data Transfer

1. Direct (memory-register)

```
load <memory address> <Ri>
```

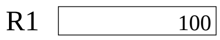
Loads the contents of memory whose address is Mem into register `Ri`.
Example: `load 100 R1`



2. Immediate

```
loadn <number> <Ri>
```

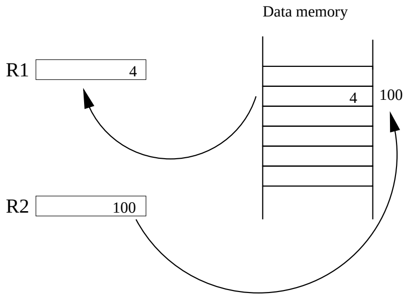
Loads number `number` into register `Ri`.
Example: `loadn 100 R1`



3. Indirect (memory-register)

```
loadi <Ri> <Rj>
```

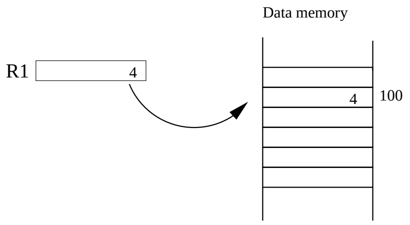
Loads the contents of memory cell whose address is the contents of register `Ri` into register `Rj`.
Example: `loadi R2 R1`



4. Direct (register-memory)

```
store  <Ri> <memory address>
```

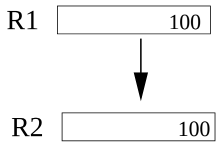
Stores the contents of register `Ri` in the memory cell whose address is given.
Example: `store R1 100`



5. Direct (register-register)

   ```
   storer <Ri> <Rj>
   ```

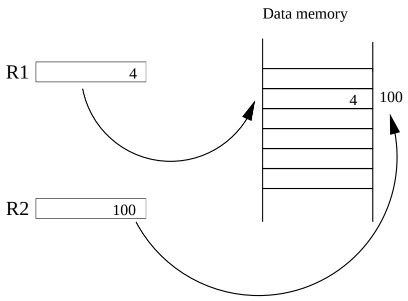
   Stores the contents of register `Ri` into register `Rj`.
   Example: `storer R1 R2`



6. Indirect (register-memory)

   ```
   storei <Ri> <Rj>
   ```

   Stores the contents of register `Ri` into the memory cell whose address is the contents of register `Rj`.
   Example: `storei R1 R2`



Arithmetic Only between registers.

1. Add (register-register)

   ```
   add <Ri> <Rj>
   ```

   Action: `Rj=Ri+Rj`
   Example: `add R1 R2`
   if `R1=100` and `R2=50`, after the instruction `R2=150`
2. Subtraction (register-register)

   ```
   sub <Ri> <Rj>
   ```

   Action: `Rj=Ri-Rj`
   Example: `sub R1 R2`
3. Multiplication (register-register)

   ```
   mul <Ri> <Rj>
   ```

   Action: `Rj=Ri*Rj`
   Example: `mul R1 R2`
4. Integer division (register-register)

   ```
   div <Ri> <Rj>
   ```

   Action: `Rj=Ri/Rj`
   Example: `div R1 R2`
5. Remainder of integer division (register-register)

```
mod <Ri> <Rj>
```

Action: `Rj=Ri%Rj`
Example: `mod R1 R2`

6. Set to zero (register)

```
zero <Ri>
```

Action: `Ri=0`
Example: `zero R1`

7. Increment (register)

```
inc <Ri>
```

Action: `Ri=Ri+1`
Example: `inc R1`

8. Decrement (register)

```
dec <Ri>
```

Action: `Ri=Ri-1`
Example: `dec R1`

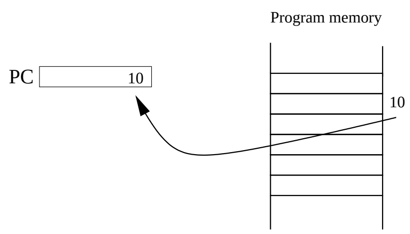
Control Flow

Modify the value of the program counter (`PC`) in order to alter the normal flow of the program.

1. Unconditional

```
jump <address>
```

Loads the address `address` into the `PC`. This address must correspond to a memory cell whose contents is a program instruction. Usually this address is given by a label.
Example: `jump loop`
If the label `loop` corresponds to the program address `100` we have:



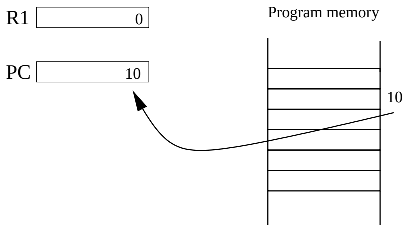
2. Conditional: its execution is conditioned by the value of a register.

(a) if zero

```
jzero <Ri> <address>
```

Loads the address `address` into the `PC`, if the contents of register `Ri` is zero.

Example: `jzero R1 loop`
As in the previous example:

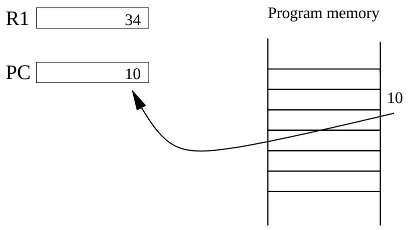


if $\texttt{R1} \neq \texttt{0}$ the instruction will do nothing and, as usual, the contents of the `PC` will be incremented to point to the next instruction in memory.

(b) if positive

```
jpos <Ri> <address>
```

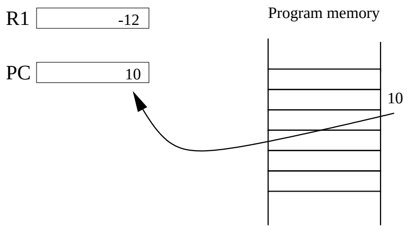
Loads the address `address` into the `PC`, if the contents of register `Ri` is a positive integer.
Example: `jpos R1 loop`



(c) if negative

```
jneg <Ri> <address>
```

Loads the address `address` into the `PC`, if the contents of register `Ri` is a negative integer.
Example: `jneg R1 loop`



3. Execution halt

```
halt
```

Stops the execution of the program. Every program must have this instruction on order to end properly, otherwise an `Out of Program` error will occur.