Apoo32: A Virtual Machine for Apoo Assembly

Pedro Vasconcelos
Departamento de Ciência de Computadores
Faculdade of Ciências, Universidade do Porto
pbv@dcc.fc.up.pt

March 23, 2006

1 Introduction

Apoo [1, 2] is an assembly language teaching environment developed at the Department of Computer Science of the Faculty of Science, University of Porto. It consists of a very simple assembly language with a reduced instruction set plus a graphical user interface for editing, executing and debugging assembly programs. Because the emphasis of Apoo is on programming at the assembly level, no machine code translation is used: Apoo programs are executed as mnemonics.

This document describes Apoo32, a virtual machine code for Apoo programs. We develop this extension with three teaching objectives:

- as an introduction to computer architecture of machine languages e.g. instruction sets, addressing modes, memory segments, etc.;
- as an introduction to virtual machines and byte-code interpreters;
- as a reference design for student's projects, e.g. virtual machine interpreters or language compilers.

The machine is named "Apoo32" because the constant 32 pops up in many of the architectural constraints: there are 32 registers, each register is 32-bits long and data and instruction words are 32-bits long.

2 Apoo32 Architecture

2.1 Programmer level architecture

The Apoo32 programming model has 32 general purpose registers named `R0` to `R31`, each register holding a 32-bit value. Two other special 32-bit registers are used: a *program counter* and a *stack pointer*; these are manipulated by specific instructions.

Data Address	Read	Write
50000	always 0	write ASCII char
50001	read an integer	write an integer
50010	always 0	write CR char

Table 1: Memory-mapped I/O

The instruction set follows a load-store architecture, i.e. all arithmetic and logic operations are done in registers. There are only four addressing modes: immediate (literal), register, direct memory, and indirect memory (through a register). Instructions have zero, one or two operands.

The Apoo32 run-time environment distinguishes three memory areas: the *code segment*, the (static) *data segment* and the *run-time stack*. Each memory segment is a contiguous array of 32-bit words. Addresses for code and data segment are disjoint and start at zero; the only addressable datum is the single word. The stack is not randomly addressable; instead instructions `push`, `pop`, `jsr` and `rtn` address the top of the stack via through the stack pointer. The stack pointer cannot be moved or otherwise directly manipulated by the programmer.

2.2 Memory-Mapped Input/Output

Apoo32 machine includes a rudimentary memory-mapped input/output interface. Table 1 lists the default functionality.

2.3 Instruction encoding

Apoo32 instructions are one or two 32-bits words long, named word 0 and 1. The first word is divided into three sub-fields: the opcode and two (optional) operands. When used, the second word represents a single 32-bit field.

Table 2 describes the encoding for each assembly mnemonic. Fields within a word are represented by a range of bits e.g. [7:0]. Bits are numbered from 0 to 31, with 0 being the least-significant and 31 the most-significant. Entries with '–' represent ignored or unused fields.

2.3.1 Opcode

The opcode is encoded in the lower 8-bits of the first instruction word. Only opcodes `0x0`–`0x17` are defined. This means that the encoding can easily be extended to accommodate a larger instruction set.

2.3.2 Register Operands

Fields arg0 and arg1 in the first word encode one or two register arguments. Each register number 0 to 31 is encoded in 5 bits.

	Word 0			Word 1
	arg1	**arg0**	**opcode**	
Mnemonic	[20:16]	[12:8]	[7:0]	[31:0]
halt	–	–	`0x0`	–
load Addr Ri	–	Ri	`0x1`	Addr
loadn Imm Ri	–	Ri	`0x2`	Imm
loadi Ri Rj	Rj	Ri	`0x3`	–
store Ri Addr	–	Ri	`0x4`	Addr
storer Ri Rj	Rj	Ri	`0x5`	–
storei Ri Rj	Rj	Ri	`0x6`	–
add Ri Rj	Rj	Ri	`0x7`	–
sub Ri Rj	Rj	Ri	`0x8`	–
mul Ri Rj	Rj	Ri	`0x9`	–
div Ri Rj	Rj	Ri	`0xa`	–
mod Ri Rj	Rj	Ri	`0xb`	–
zero Ri	–	Ri	`0xc`	–
inc Ri	–	Ri	`0xd`	–
dec Ri	–	Ri	`0xe`	–
jump Addr	–	–	`0xf`	Addr
jnzero Ri Addr	–	Ri	`0x10`	Addr
jzero Ri Addr	–	Ri	`0x11`	Addr
jpos Ri Addr	–	Ri	`0x12`	Addr
jneg Ri Addr	–	Ri	`0x13`	Addr
jsr Addr	–	–	`0x14`	Addr
rtn	–	–	`0x15`	–
push Ri	–	Ri	`0x16`	–
pop Ri	–	Ri	`0x17`	–

Table 2: Apoo32 instruction encoding

Register fields are aligned with byte boundaries in the word; this allows easier reading of machine code in hexadecimal form (see examples following). Only a total of 18 bits of the first word are used. Thus the encoding could be extended to accommodate extra registers or addressing modes.

2.3.3 Immediate Operands

For instructions with an immediate value or memory or program address, a second 32-bit word is used to encoded its value. Program address are absolute i.e. not relative to the program counter.

2.4 Examples

Example 1 To encode the mnemonic `add R2 R3`:

opcode `add`	=	`0x7`
arg0 `R2`	=	`0x2`
arg1 `R3`	=	`0x3`
word 0	=	opcode`|`(arg0`<<`8)`|`(arg1`<<`16)
	=	`0x30207`

This instruction is one word long.

Example 2 To encode the mnemonic `store R15 0x1000`:

opcode `store`	=	`0x4`
arg0 `R15`	=	`0xf`
arg1 not used		
word 0	=	opcode`|`(arg0`<<`8)
	=	`0xf04`
word 1	=	`0x1000`

This instruction is two words long.

3 Apoo32 Assembler

A prototype Apoo32 assembler is available under the Gnu Public License in source code and compiled binary forms at `http://www.ncc.fc.up.pt/apoo`. The assembler itself is written in the functional language Haskell, making it very concise, easy to understand and modify.[1] The assembler is invoked on a source file with Apoo mnemonics, labels and optional comments:

```
$ apooas file.apoo
```

The output is written to a C-language file `file.apoo.c`:

[1] The source code is about 300-lines long, split into 200-lines for the assembler itself (including code data structures and label resolution) and 100-lines for the Apoo mnemonic parser.

```
N:      const 20
        load  N R1
        loadn 0 R2
        jump cond
loop:   add R1 R2
        dec R1
cond:   jpos R1 loop
        halt
```

Table 3: Sample Apoo assembly source file

```
/* Apoo32 Assembler v1.0.  Copyleft Pedro Vasconcelos, 2006 */
/* This file has been generated automatically; do not edit */

int Apoo_data[] = {0x14};
int Apoo_data_size = 1;
int Apoo_code[] = {0x101,0x0,0x202,0x0,0xf,0x8,0x20107,
   0x10e,0x112,0x6,0x0};
int Apoo_code_size = 11;
```

Table 4: Sample Apoo32 assembler output

```
int Apoo_data[] = { /* data segment values */ };
int Apoo_code[] = { /* code segment values */ };
int Apoo_data_size = .. ; /* number of words in data segment */
int Apoo_code_size = .. ; /* number of words in code segment */
```

We choose the C language for the output format because:

- it allows for easy portability across platforms with different binary formats;
- it avoids the need to introduce students to yet another data format;
- it allows easy integration with student's assignment projects e.g. virtual machine interpreters: the interface between the assembler and the interpreter is done by declaring the the data and code segments as external C symbols.

Example 3 Tables 3 and 4 displays a sample input file and output byte-code of the assembler.

Example 4 No presentation of a toy programming language is complete without an "Hello world" example (Tables 5 and 6).

```
putc:   equ 50000
txt:    string "Hello world!\n"
        # main program
        loadn txt R0
        jsr puts
        halt

        # subroutine to print string at R0
puts:   jump cond
loop:   store R1 putc   # output char
        inc R0
cond:   loadi R0 R1
        jnzero R1 loop
        rtn
```

Table 5: “Hello world” example

```
/* Apoo32 Assembler v1.0. Copyleft Pedro Vasconcelos, 2006 */
/* This file has been generated automatically; do not edit */

int Apoo_data[] = {0x48,0x65,0x6c,0x6c,0x6f,0x20,0x77,0x6f,
   0x72,0x6c,0x64,0x21,0xa,0x0};
int Apoo_data_size = 14;
int Apoo_code[] = {0x2,0x0,0x14,0x5,0x0,0xf,0xa,0x104,0xc350,
   0xd,0x10003,0x110,0x7,0x15};
int Apoo_code_size = 14;
```

Table 6: “Hello world” data and byte-code

References

[1] Rogério Reis and Nelma Moreira. *Apoo: an environment for a first course in assembly language programming.* Technical Report DCC-98-9, DCC-FC & LIACC, Universidade do Porto, 1998.

[2] Rogério Reis and Nelma Moreira. *Apoo: an environment for a first course in assembly language programming.* SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education), 33(2), December 2001.