VSP88: 8-bit virtual processor, language and assembler

César Parent

Abertay University

BSc Computing

2014/2015

ABSTRACT

This paper gives a report of the definition, implementation and testing of a virtual computer architecture similar on the first generation of microprocessors. The goal was to design an instruction set, assembler, virtual processor and machine that worked in a way similar to that of 1970-era computers.

After a phase of research and study of early micro-processors, the Motorola 6800 and derived MOS Technology 6502 were chosen as models. The implementation was divided in three phases: designing the machine language and instruction set, writing an assembler, and finally write the virtual machine to test it.

Overall, the project was successful. The assembler can parse code and translate it to machine instructions, and VSP88, the machine, can execute it. To prove that the instruction set was complete enough, a RAM monitor, a simple command prompt and a Pong game were built in assembly language.¶

TABLE OF CONTENTS

I. INTRODUCTION

The microprocessor is considered the heart of a computer. Even though modern computers make use of several *co-processors* — for graphics for example — the *Central Processing Unit* or CPU is the computer's director: it orchestrate the execution, dispatching commands to the rest of the computer's components [4].

A microprocessor is an integrated circuit made of several — nowadays, billions [14] — transistors arranged to operate in a very particular way. It takes input and outputs data on a data bus in a binary form, addresses memory circuits using the address bus, and stores temporary data in registers.

Each processor architecture has a certain amount of operations it can carry, called the instruction set. Each instruction is mapped to a binary-encoded value that can be stored in memory, as part of a *programme* [5]. When the processor runs, it operates in cycles. Each cycle consists of three steps: reading the instruction and its operand from memory, decode it, and execute the operation, hence the name, Fetch-Decode-Execute cycle (FDX) [1].

Programmes are a sequence of instructions and data stored in memory. To make it easier to write those programmes, assembly languages map binary instruction to more human readable instructions. A special software, the assembler, then parses assembly code and translates it into binary machine instructions that can be ran [2].

Modern computer architectures are complex. Processors have very large instruction sets that can be hard to understand in their entirety, documentations spanning multiple several-hundred-page volumes [6]. Peripherals are numerous and very complex. For those reasons, a modern computer is not the optimal platform to understand the basics of computer architecture.

1970-era computers on the other hand were simple machines. The main processors of the first generation of personal computers, mostly derived from the Motorola 6800, are simple circuits. They had a handful of registers at most [11], small and understandable instruction sets [7] — basic arithmetic and logic operations, memory handling and flow control [12]. Peripherals were mapped to memory addresses [11], making communication with the processor easy. For these reasons those architectures are good platforms to learn how a processor and assembler work.

The goal of this project was to get a better understanding of the workings of a processor and assembler by designing and implementing an assembly language, an assembler and a virtual 8-bit machine.¶

II. PROCEDURE

The project was divided in three phases. The first one consisted in designing the virtual processor, its machine & assembly language and the surrounding virtual machine. The second step was to implement the assembler and then the virtual machine in C. The third phase was meant to assess wether the processor was complete enough by attempting to write a simple command prompt and a Pong game.

TOOLS USED

The design and development process was done on a Mac OS X machine. The tools used in the development of the assembler and virtual machine were:

- C programming language: the assembler and virtual machine are written in C;
- SDL2: graphics library used to simulate the virtual machine's video circuit;
- Xcode 6: Mac OS X development environment for C-based languages;
- Clang/LLVM Compiler: C-based languages compiler (v6.0/SVN3.5);
- xxd: hexadecimal/binary dump command line tool bundled with OS X;
- TextMate: code text editor with custom language definitions for assembly;
- Numbers: spreadsheet application used to organise the instruction set.

COMMON VOCABULARY

- *Address (in memory)*: unique index, usually counting upwards from 0, that points to a cell of memory (RAM or ROM).
- *Bit*: one digit in binary arithmetic, can be 0 or 1.
- Bus: system used to transfer data between components of a computer. The *bus width* is the amount of data (int bits or bytes) that can be transferred at one time on the bus.
- *Byte*: eight consecutive bits.
- *Mnemonic*: string of character denoting an assembly instruction.
- *Random Access Memory (RAM)*: data storage system that keeps its data only when powered and can be read and written to. RAM is organised in cells, each mapped to an address and containing a very small amount of data.
- *Read Only Memory (ROM)*: Similar to RAM, except that it cannot be written to, and the data is kept even when the memory is powered down.
- *Register*: very small data storage unit present inside the processor.¶

II.1. PHASE 1: ARCHITECTURE AND LANGUAGE DEFINITION

The first phase of the project was to define the architecture of the processor and virtual machine.

The overall design of both software and virtual hardware are heavily inspired by computers from the 1970 era, particularly the MOS Technology 6502, and the processor it was derived from, the Motorola 6800. The low cost of the 6502 at the time made it very popular — a version of it is still produced in 2015 — and documentation is thus very easy to find. To keep the project relatively simple, VSP88 was designed without any support for interrupts, even though the 6502 processor handled them [12].

II.1.1. PROCESSOR BUSES

VSP88 is based around an 8-bit architecture. The data input/output bus is height bit wide, and each address of memory contains an 8-bit value. All operations are carried in 8-bit arithmetic, and operation codes are 8-bit long. The address bus, used to address memory locations, is 16-bit wide, and VSP88 can thus access upon to 65,536 bytes (64KB) of memory.

II.1.2. PROCESSOR REGISTERS

VSP88 has four special purpose registers and two general purpose register, detailed below on Fig.1:

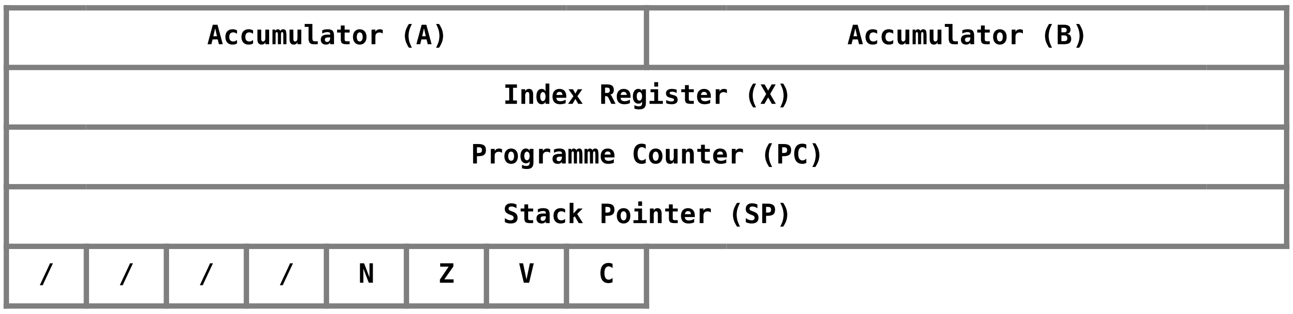


FIG.1 - VSP88 REGISTERS

The six registers and their purposes are:

- *Accumulators (A, B)*: two 8-bit registers available for general use. This means little storage for calculations, but allows smaller instructions.

- *Index Register (X)*: 16-bit register used to as an memory address offset in certain modes. This can be helpful when looping through data structures like arrays or strings;

- *Programme Counter (PC)*: 16-bit register pointing to the memory address of the next machine instruction; necessary for any processor.

- *Stack Pointer (SP)*: 16-bit register pointing to the memory address of the top of the stack. The stack is a last-in, first-out data, used as a temporary storage space in memory. Data from registers can be pushed onto the stack, and pulled from it.
- *Flags Registers (F)*: 8-bit register storing information on the result of arithmetic and logic operations. the four high bits are not used.
  - Bit 0 (C) is the carry flag. It is set when an operation triggers a carry into the most significant bit of the result, and cleared otherwise.
  - Bit 1 (V) is the overflow flag. It is set when the result of an operation is bigger than the maximum value stored in 8-bit two's complement, and cleared otherwise.
  - Bit 2 (Z) is the zero flag. It is set when the result of an operation is zero, and cleared otherwise.
  - Bit 3 (N) is the negative flag. It is set when the result of an operation is negative, and cleared otherwise.

II.1.3. INSTRUCTION SET

Defining the instruction set took some time, because it needs to be balanced: too big, and implementing the assembler and the virtual machine would have been too complex. Too small, and writing even simple programme would be hard, if not impossible due to the lack of some features.

Instructions can be sorted into four categories: Utility, Memory Operations, Arithmetic & Logic and Flow Control.

- Utility: there needs to be a way to set the Stack Pointer, and one to halt the processor execution.
- Memory Operations: loading data from memory into registers, and storing data from registers into memory locations. This includes pushing values to and pulling values from the stack.
- Arithmetic & Logic: the number operations had to be limited: there is no "multiply" or "divide" instructions, since they can be reproduced with a combination of shift and add/subtract operations. Likewise, increment and decrement were left out. This left add, subtract, shift, rotate, logical and, logical or and comparison.
- Flow Control: unconditional and conditional branches (Zero flag set, Zero flag clear, Carry flag set, Carry flag clear, Negative flag set, Negative flag clear) are available. There is also the jump to subroutine, that pushes the current instruction's address on the stack before jumping, and return from subroutine, which pulls the return address from the stack into the Instruction Pointer.

In assembly, instructions are represented by a three-letter code, called mnemonics. Instructions that affect one of the two accumulators have its letter (A or B) added at the end. Instructions take zero or one parameter.

The final Instruction Set is made of 32 instructions, with most Arithmetic & Logic operations available only on general purpose registers. The complete instruction set listing is presented in appendix 1.

```
label:  LDAA      #42
        ADDA      #1
        STRA      $ff
        LDAB      X,$ff
        BNE       label
        HCF
```

FIG.2 - VSP88 ASSEMBLY EXAMPLE

II.1.4. ADDRESSING MODES

Most instructions need an operand. This operand can be an immediate value, stored directly in the programme, but also an memory address to get the value from. To make sure relatively complex programmes could be written, three addressing modes were initially planned:

Addressing Mode	Example	Description
Immediate	`LDAA #42`	The value is loaded directly from the programme.
Direct	`LDAA $00ff`	The value is loaded from the address written in the programme. *(ex: load value stored at 00ff)*
Indexed Direct	`LDAA X,$00ff`	The address from the programme is added to the value of the Index Register, and the value is loaded from that new address. *(ex: load value stored at (00ff+X))*

After a few attempts to write software in assembly, a fourth indexing mode was added to allow the use of pointers (memory content pointing to other locations).

Indexed Indirect	`LDAA X^$00ff`	An address (pointer) is loaded from the two bytes at the address written in the programme. This value is added to the value of the Index Register, and the value is loaded from that new address. *(ex: load value stored at ([00ff][0100]+X))*

II.1.5. INSTRUCTION FORMAT

To keep the processor execution logic simple, every instruction *must* fit into one memory cell, and thus be no longer than eight bits. Those eight bits should indicate the instruction, what accumulator is targeted, and what addressing mode is used.

- One bit indicates the target accumulator: 0 for A, 1 for B;
- two bits are reserved to indicated the addressing mode: 00 (0) for immediate, 01 (1) for Direct, 0b10 (2) for Indexed Direct, 0b11 (3) for Indexed Indirect;
- the remaining five bits are available for the instruction (00000 through 11111 (32)).

Operation Code					**addressing mode**		**Target register**
`0`	`0`	`0`	`0`	`0`	`0`	`0`	`0`

FIG.3 - VSP88 OPCODE FORMAT

Once the instructions codes and the instruction format were chosen, a map of all the operation codes (OpCodes) was made to aid the development process. It is presented in Appendix 2.

II.1.6. INPUT/OUTPUT DESIGN

To keep inputing and outputting data simple, memory handling on VSP88 is very similar to that used in the Apple II [15]. Everything, RAM, ROM and peripherals is considered memory and is addressed through the main address bus.

Whenever a key is pressed on the keyboard, its ASCII value is placed in memory location 0xEFFF, and the most significant bit is set to signal any programme running that a new value is waiting.

The display works with RAM as well, and copies almost exactly the text and “low-res” modes of the Apple II. Location 0x07FF controls the display mode: if it contains 0, the display is set to text, otherwise it switches to graphics mode.

In text mode (24 lines of 40 characters), address 0x0400 holds the ASCII code for the character at the top left of the screen, and the next 960 (40×24) addresses hold the characters, column by column, line by line. Each time the display logic runs, it reads those 960 addresses and displays them.

In graphics mode (40 by 48 pixels), each of those 960 bytes stores two pixels, stacked on top of each other. The four least significant bits store the colour code for the bottom pixel while the four most significant bits store the colour code for the top pixel. This gives 16 (2^4) possible colours for each pixel. The number denotes the address of a colour in a 16-colour palette similar to that of the Apple II [13].

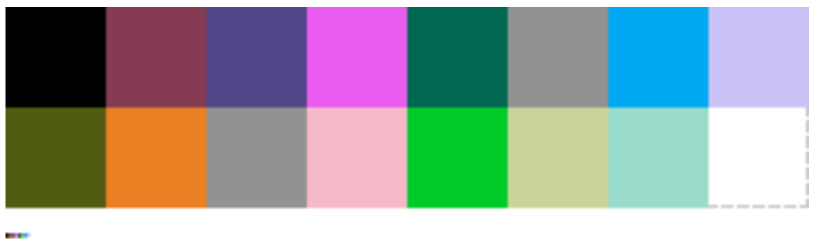


FIG.4 - VSP88/APPLE II COLOR PALETTE

II.1.7. MEMORY MAP

As previously discussed, the display memory occupies addresses 0x0400 to 0x07FF. Likewise, 0xEFFF is reserved for input handling. The ROM, which is not writable at run-time, is located at the top of the address space, from 0xF000 to 0xFFFF (4KB). When VSP88 is started, it will start executing the instruction located at address 0xF000.

User/Programme memory	`0x0000`
	`0x03FF`

Display memory	`0x0400`
	`0x07FF`

User/Programme memory	`0x0800`
	`0xEFFE`

Input memory	`0xEFFF`

ROM (OS/routines)	`0xF000`
	`0xFFFF`

FIG.5 - VSP88 ADDRESS MAP

II.2. PHASE 2: BUILDING THE ASSEMBLER

An assembler is a translator software: it takes in a text file containing assembly code mnemonics, and translates those to valid machine code (binary) instructions. Since each assembly mnemonic maps to one and only one machine instruction, the translation can be done independently for each line.

Translating is done in two phases: the first aims at separating a line of code in assembly “words”, called tokens [3]. The second phase, parsing, is the most complex: each token is matched to its function (instruction, operand, comment, label), and then translated to a valid binary value. During this phase, the assembler also checks that the instruction is valid and the operand is allowed for that instruction.

Certain instructions (jumps and branches) take the address where the processor should jump as an operand. It can be cumbersome for the programmer to keep track of the start address of a routine. To solve that problem, the assembler also introduces labels: a line can optionally be marked with a label, and the assembler will keep track of the address of each label. That way, labels can be used as operands, and the assembler will resolve the real address whilst assembling the programme.

```
label:   LDAA      #42
         ADDA      #1
         JMP       label
```

FIG.6 - USE OF LABELS IN VSP88 ASSEMBLY

This poses one problem: the assembler needs to know about all the labels of the programme even while parsing the first line: otherwise, it would impossible to programme jumps to further portions of the code since the assembler would not know about them yet. For this reason, a two-pass assembler design [9] was chosen: during the first assembly pass, the software assembles any instruction that does not call any label, and adds each label it finds and its address to a symbol table. During the second pass, any instruction that could not be assembled during the first pass is completed using the symbol table.

II.2.1. TOKENISER

The tokeniser attempts to reproduce what the human brain does first when reading a line of text: divide the line in words. To achieve this, a list of word separators is defined: any number of spaces or tabulations can be present between tokens.

The presence of a label is detected by the last character of a token being a colon. In that case, the line’s address is added to the symbol table. Any portion of the line stating with a semicolon is considered a comment and is ignored. Once all the tokens are identified, their value is stored in a temporary buffer for the next steps to use.

Assembly line	`START:`	`JMP`	`A_LABEL`	`; load 42 into A`
Tokens	Label	Mnemonic	Operand	Comment, ignore

FIG.7 - EXAMPLE ASSEMBLY LINE'S TOKENS

II.2.2. MNEMONIC PARSER

The assembler contains a table mapping valid mnemonics to their binary value and allowed addressing modes. When a line has been tokenised, the assembler tries to match the mnemonic to one of the table's entries. If it cannot be done, an error is thrown and the assembler exits.

If the mnemonic is valid, the parser adds the binary value of the instruction to the line's data buffer.

Mnemonic	Operand	Bin. Instr.	Complete	Addr Mode	Bin. Oper.	Target Reg.
JMP	A_LABEL	10100	FALSE	?	?	None

FIG.8 - EXAMPLE ASSEMBLY LINE'S DATA BUFFER AFTER MNEMONIC PARSING STEP

II.2.3. OPERAND PARSER

Parsing a line's operand happens in two steps: first the assembler checks the format of the operand to deduce the addressing mode and check if it is allowed with the line's instruction. Then, the operand is converted from string to a numeric value and stored in the line's data buffer.

To determine what addressing mode is used, the assembler was set to look at the first two characters of the operand.

First characters	Addressing Mode
`'#'`	Immediate addressing
Digit or `'$'`	Direct addressing
`'X,'`	Direct Indexed Addressing
`'X^'`	Indirect Indexed Addressing

FIG.9 - ADDRESSING MODES PREFIXES IN ASSEMBLY

Once the mode is determined, the first character of the actual operand tells the assembler if the value is decimal (no prefix) or hexadecimal ('$' prefix). Knowing that, the assembler can use C standard library's functions to parse the string into a numeric value.

Mnemonic	Operand	Bin. Instr.	Complete	Addr Mode	Bin. Oper.	Target Reg.
JMP	A_LABEL	10100	FALSE	Direct(Label)	?	None

FIG.10 - EXAMPLE ASSEMBLY LINE'S DATA BUFFER AFTER OPERAND PARSING STEP

II.2.4. SECOND PASS AND FINALISATION

Once each line has been parsed and the symbol table filled, the second pass starts. The assembler only checks lines tagged as incomplete and fills in the operand field using the symbol table.

Mnemonic	Operand	Bin. Instr.	Complete	Addr Mode	Bin. Oper.	Target Reg.
JMP	A_LABEL	10100	FALSE	Direct(Label)	$0f7f	None

FIG.11 - EXAMPLE ASSEMBLY LINE'S DATA BUFFER AFTER SECOND PASS

From there on, the assembler knows enough about the programme to write the binary file. The complete binary code for each line is constructed by combining the instruction code, the target register and the addressing mode, and the values are written to the output file given as a command line parameter.

II.2.5. STRING DIRECTIVES

After writing some assembly programme, it was found that outputting strings was tedious: each character had to be loaded into memory one by one, using its ascii code. As shown in Fig.11, the longer the string, the longer the programme. Maintaining such programmes would be more than tedious.

```
label:  LDAA    #$68     ; 0x68 = h
        JSR     PRCHAR   ; print
        LDAA    #$65     ; 0x65 = e
        JSR     PRCHAR   ; print
        LDAA    #$6c     ; 0x6c = l
        JSR     PRCHAR   ; print
        JSR     PRCHAR   ; print
        LDAA    #$6f     ; 0x6f = o
        JSR     PRCHAR   ; print
        HLT
```

FIG.12 - PRINTING "HELLO" WITHOUT DIRECTIVES

String directives were added to the assembler: directives are instructions that do not map to machine code, but tell the assembler to do some kind of operation. The string directive is mapped to the mnemonic ".str", and the operand must be a string in double quotes.

```
string:       .str      “hello”

              LDX       string     ; load the address of string
              JSR       pr_str     ; call print string routine
end:          HLT

pr_str:       LDAA      X,0        ; load current character in A
              BEQ       pr_str_end ; stop if character = \0
              JSR       PRCHAR     ; print character
              ADX       #1         ; increment index
              JMP       pr_str     ; loop
pr_str_end:   RTS
```

FIG.13 - PRINTING “HELLO” WITH DIRECTIVES

When the assembler encounters a string directive, the label of the string is added to the symbol table. This way, the string’s location in memory can be easily accessed by the programme. Then, the string’s characters are written in the programme’s binary file, right after the main programme’s machine code. A null character (0x00) is placed after the string to mark its end.¶

II.3. PHASE 3: BUILDING THE VIRTUAL MACHINE

The goal of the Virtual Machine was to reproduce the macro level execution of a processor and 1970 era computer. The electronic circuits were not simulated, as it would have made the implementation much more complex.

The work was focused on the processor and its fetch, read and and execute cycle. The workings of the RAM, ROM and video unit were somewhat simplified to keep the scope of the project small enough.

The virtual machine was built using the C programming language. Drawing to the screen and handling input was achieved through the SDL 2 library.

The processor's state is held by a C struct object, with fields for each of the registers, as well as house-keeping variables.

When the machine starts, the assembled programme given as a parameter is copied into the C array that represents the RAM. Then, a loop runs at one megahertz. Each loop represents one instruction cycle and is split in three phases: fetch, decode and execute. The loop runs until the use explicitly quits the programme.

II.3.1. INSTRUCTION CYCLE: FETCH

Each instruction cycle of the processor starts with the *fetch* operation: on a real machine, the address contained in the Programme Counter is put on the address bus, which allows the processor to read the next Operation Code from memory.

On the virtual machine, the RAM is represented as an array of bytes. The fetch function reads the byte at the index stored in the programme counter, and puts it in the instruction register of the CPU state object for the decoding and execution units to use.

II.3.2. INSTRUCTION CYCLE: DECODE

When the processor reaches the decode phase, the only thing it knows is the Operation Code. As discussed in the design phase, this Operation Code combines information about the instruction, but also what register it targets and what addressing mode is used.

During the decoding phase, the instruction number and then the addressing mode and target registers are extracted from the Operation Code. this allows the processor to know wether there is an operand or not and fetch it from memory. Once this has been done, the processor can determine how many bytes of memory the complete instruction (one byte Operation Code and possible operand) uses, and increment the Programme Counter to point to the next instruction.

II.3.3. INSTRUCTION CYCLE: EXECUTE

On the virtual machine, a table maps each instruction code to the C function that simulates it. When the processor reaches the execution phase, it uses the map to call the function corresponding to the decoded instruction code. Each function does what was specified during the design phase (add the two accumulators together for example), and possibly changes the processor's flags register.

II.3.4. DISPLAY AND INPUT

The part of the programme handling display is written using SDL. When the virtual machine is started, a window is created. Each time the display must be rendered, the function loops through the display section of the machine's RAM, and displays either a character in each slot, or a pixel which colour is taken from an 16-colour array.

Whenever a key is pressed, SDL detects it, and the programme puts its equivalent ASCII value in the processor's RAM array. Due to performance issues, the machine had to be made a little less realistic: the processor loop runs in one thread at the processor's clock speed, while the main thread is used only for display and input. This allows the main thread to run at a comparatively slow speed.¶

II.4. PHASE 4: TEST PROGRAMMES

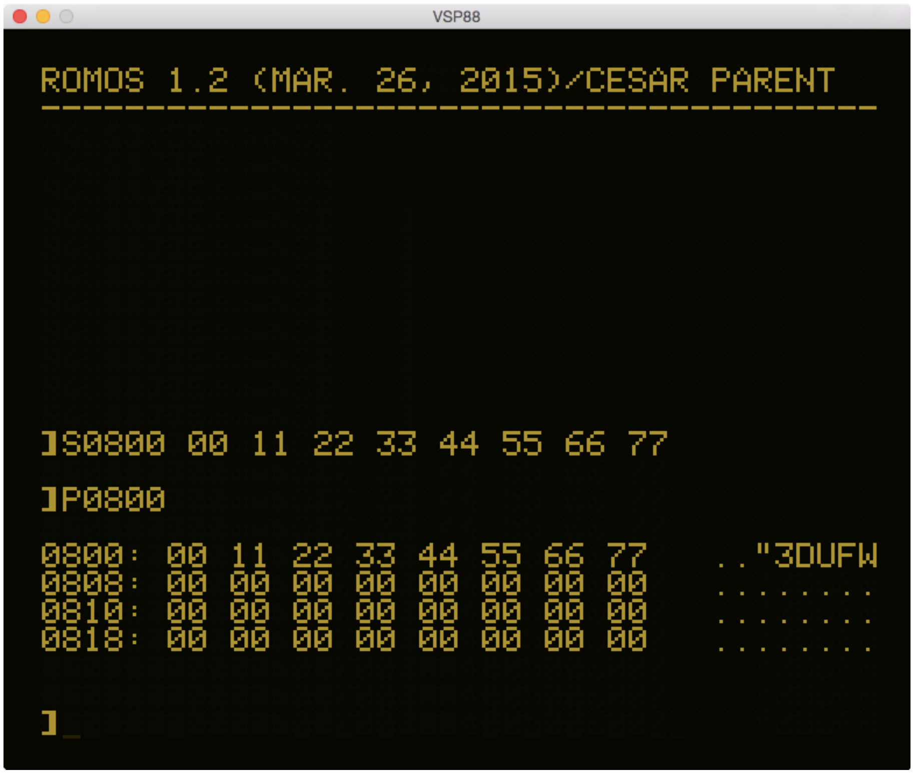
In order to test the virtual machine and verify that the instruction set was complete enough, two test programmes were written. A basic programme and its equivalent machine code are presented in appendix 3.

II.4.1 ROMOS/MONITOR

The first programme was a RAM monitor like 1970-era computers used to have: the user can display a portion of the memory, put data in it and run programmes starting at a given address.

Writing the Monitor uncovered the need of pointers, and thus the fourth and last addressing mode (indirect indexed). This allowed to write a simple monitor, that would not be considered finished (no error checking) but was good enough to debug other programmes. The monitor has three commands: P$$$$ (where $$$$ is an address) to print memory content, S$$$$ ## to set memory content at address $$$$ to ##, and R$$$$ to run a programme stored at address $$$$.

The monitor is stored in the rom (starting at 0xF000) and exposes common routines (print strings, characters and pixels to the screen) available to any other programme running. A listing of the Monitor's source code is presented in the first part of appendix 5.



```
ROMOS 1.2 (MAR. 26, 2015)/CESAR PARENT
----------------------------------------



]S0800 00 11 22 33 44 55 66 77
]P0800
0800: 00 11 22 33 44 55 66 77   .."3DUFW
0808: 00 00 00 00 00 00 00 00   ........
0810: 00 00 00 00 00 00 00 00   ........
0818: 00 00 00 00 00 00 00 00   ........

]_
```

FIG.14 - ROM OS AND MONITOR

II.4.2. PONG

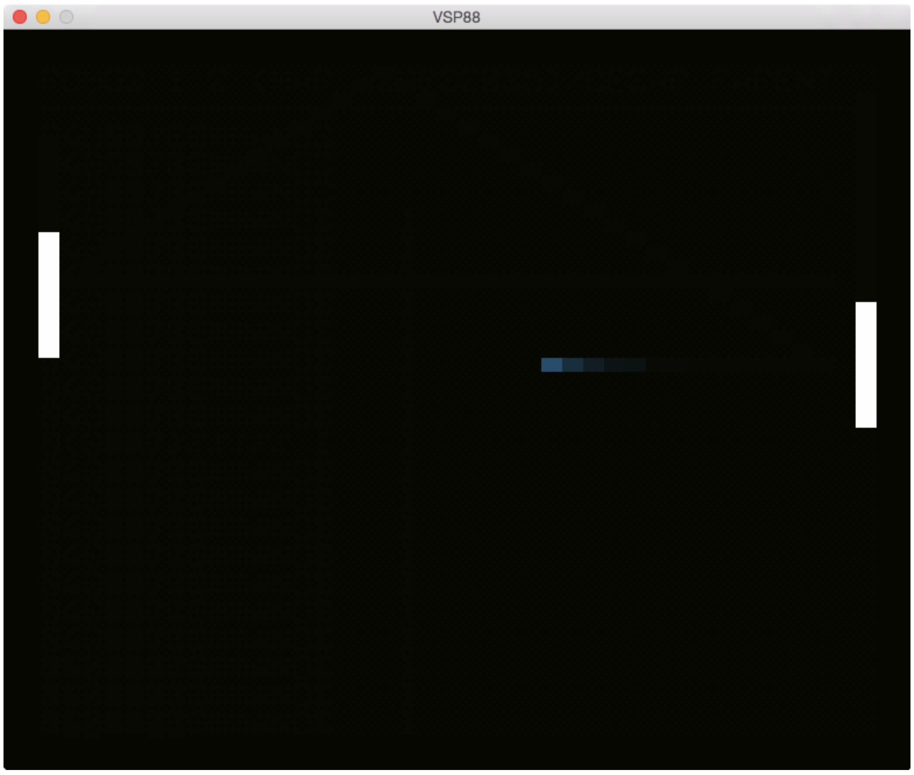


FIG.15 - PONG RUNNING ON VSP88

The second test programme that was written aimed at testing the graphics system. It is a very basic replica of the classic game Pong. This game was simple enough that it could be written before the addition of the Indirect Indexed addressing mode.

Both paddle's positions are stored in fixed places in memory, as is that of the ball. Each time the game loop runs, it checks if A (up) or Z (down) was typed on the keyboard, and if so moves the player's (left) paddle. Since only the last pressed keyboard key is stored in memory, it was not possible to have a second human player. Instead, the programme checks for the difference between the opponent paddle's and the ball's vertical position, and then moves the right paddle to try to catch the ball.

When the ball reaches the left or right wall, the game displays "you won" or "you lost" and exits to the main command prompt.

A listing of Pong's source code is presented in the second part of appendix 5.¶

III. DISCUSSION AND CONCLUSION

Overall, the project was successful. The main goals were reached: the assembler successfully parses, checks and translates programmes into machine code binary files, and VSP88 boots, executes machine code and outputs data and graphics.

The lack of an indirect addressing mode in the first version of the instruction set proved quite problematic when writing programmes in assembly: jumping to an address chosen by the user at runtime was tedious, if not impossible. Thanks to the modular architecture chosen for the assembler source code, adding the new addressing mode was relatively easy, and made the machine more able.

INSTRUCTION SET

For the chosen (simple) demonstration programmes, the Instruction Set proved complete enough. The absence of Exclusive OR and Not AND instruction made writing some routines a little hard, but overall no major problem was encountered.

The instruction set proved to be very limiting in one regard: the main registers (A, B and X) are almost completely isolated. There is no easy way to transfer the value of A to B or X and reverse. The only possible transfer is from B to A, by setting A to 0 and then adding B to A. The 6502 instruction set, for example provides instructions to facilitate those transfers (TXA, TAX, TYA, TAY...).

Transfers between A and B can be done using the RAM as a temporary data storage. Transferring from X is not possible since there is no instruction to store the value contained in X to memory. Some programmes would be hard to write because of that limitation of the Instruction Set.

A solution to this problem would be to add instructions to the set. However, the format chose to encode instructions prevents it: the instruction itself is encoded on five bits only, capping the amount to 32. Since most instructions only use some of the available addressing modes, some Operation Codes are not used. Out of the 256 possibilities (8-bit integer, from 0 to 2^8), only 107 are actually used, as shown in the Operation Codes Map presented in appendix 2.

The way around would be to proceed like Motorola and MOS Technology proceeded for the 6800 and 6502 processors: hardcode each combination of instruction, addressing mode and target register into a value ranging from 0x00 to 0xFF. That way, the 6800 has 72 instructions for 197 Operation Codes [10].

VIRTUAL MACHINE

The first version of the machine ran into a performance issue. The screen was redrawn each time a processor cycle was finished. SDL is designed for games, and while refresh speeds of a few hundred herts (updates per second), it is not capable to reach the speed the processor needed (around one megahertz). This caused the processor to be slowed down by waiting for the screen update to be finished.

This problem was solved by separating the processor's logic and the display logic in two separated threads, as discussed in section II.3.4. This allowed the processor to reach its target speed without straining the host computer's graphics processor, at the detriment of realism.

Due to the simplistic design, chosen to keep the project focused, the machine deals with peripheral in an unrealistic way, without any kind of hardware delays or interrupt system.

FURTHER DEVELOPMENT

As discussed above, some parts of the virtual machine and the assembler would need to be rewritten to allow for a more flexible and complete instruction set. It would also be interesting to try to make the virtual machine handle simulated hardware in a more realistic way, by adding interrupt support [8].

An interesting development, albeit complex and quite expensive, would be to try to create a physical version of VSP88 using a FPGA (Field Programmable Gate Array) and RAM/ROM chips to build an actual computer.

Another way to go further would be to build a virtual machine and assembler that mimic completely the MOS Technology 6502. This would thus allow to simulate a large number of computers from the 1970s like the Apple II and the Commodore 64¶

IV. REFERENCES

[1] ALISTAIR SURALL. 2013. The fetch execute cycle [Online]. Cambridge University Press. Available: http://www.cambridgegcsecomputing.org/modules_lms/more?id_module=134652&id_course=134785&id_course_inquiry=&id_section=4032 [Accessed Feb 15th, 2015].

[2] COHEN, D. 2011. CSE 351, section 2: Designing an Assembler [Online]. Available: http://courses.cs.washington.edu/courses/cse351/11wi/section/section3.html [Accessed Feb 20th, 2015].

[3] FARELL, J. A. 1995. Anatomy of a Compiler: The Tokenizer [Online]. Available: http://www.cs.man.ac.uk/~pjj/farrell/comp3.html#COMPILER_ANATOMY [Accessed Feb 21st, 2015].

[4] FERGUSON, I. 2013. Computer Hardware Architecture and Operating System - lecture 1. Dundee, UK: Abertay University.

[5] HAFEZ, A. 2013. Stored Program and Instruction code [Online]. Available: http://www.ahabdulhafez.net/hkucourses/CAO_L11_InstructCode.pdf [Accessed Feb 16th, 2015].

[6] Intel® 64 and IA-32 Architectures Software Developer's Manual, 2015. Intel.

[7] JESSOP, P. M. 1978. The Motorola 6800 Instruction Set. BYTE. New York City, NY, USA: McGraw-Hill.

[8] JOHNATAN VALVANO & RAMESH YERRABALLI. Chapter 12: Interrupts [Online]. Available: http://users.ece.utexas.edu/~valvano/Volume1/E-Book/C12_Interrupts.htm [Accessed Feb 15th, 2015].

[9] KARONIS, N. 2003. Simpletron Assembler [Online]. Available: http://faculty.cs.niu.edu/~karonis/Classes/03fall/551/sas/sas.html [Accessed Feb 21st, 2015].

[10] M6800 Programming Reference Manual, 1976. Phoenix, AZ, USA, Motorola Semiconductor Procucts Inc.

[11] MOS TECHNOLOGY 1976. MCS6500 Microcomputer Family Hardware Manual, Norristown, PA, USA, MOS Publication.

[12] MOS TECHNOLOGY 1976. MCS6500 Microcomputer Family Programming Manual, Norristown, PA, USA, MOS Publication.

[13] OWAD, T. 2005. Apple I Replica Creation: Back to the Garage, Newville, PA, USA, Syngress.

[14] SHIMPI, A. L. 2013. The Haswell Review: Intel Core i7-4770K & i5-4670K Tested [Online]. Available: http://www.anandtech.com/show/7003/the-haswell-review-intel-core-i74770k-i54560k-tested/5 [Accessed Mar 21st, 2015].

[15] WOZNIAK, S. 1977. System Description: The Apple-II by Stephen Wozniak. BYTE. May 1977 ed. New York City, NY, USA: McGraw- Hill.

V. TABLE OF FIGURES

VI. APPENDICES

1. VSP88 INSTRUCTION SET

Function	Mnemonic	Binary	Description
Halt	`HCF`	`00000`	Halts processor execution until reset is triggered

Load	`LDA[A|B] #value`
	`LDA[A|B] location`
		`00001`	Loads either a value or the content of a memory address in accumulator.

Store	`STR[A|B] location`	`00010`	Stores the value of accumulator in a memory address.

Add	`ADD[A|B] #value`
	`ADD[A|B] location`
		`00011`	Add a value or the content of a memory address to accumulator.

Add A&B	`ADD`	`00100`	Add accumulators together into accumulator A.

Subtract	`SUB[A|B] #value`
	`SUB[A|B] location`
		`00101`	Subtract a value or the content of a memory address from accumulator.

Subtract A&B	`SUB`	`00110`	Subtract accumulator B from A into accumulator A.

Logical AND	`AND[A|B] #value`
	`AND[A|B] location`
		`00111`	Logical AND between accumulator and value or content of a memory address.

Logical AND A&B	`AND`	`01000`	Logical AND between accumulators A and B.

Logical OR	`LOR[A|B] #value`
	`LOR[A|B] location`
		`01001`	Logical OR between accumulator and value or content of a memory address.

Logical OR A&B	`LOR`	`01010`	Logical OR between accumulators A and B.

Compare	`CMP[A|B] #value`
	`CMP[A|B] location`
		`01011`	Logically compare accumulator with value or content of a memory address.
			- $>$: Zero fl. 1, Carry fl. 0
			- $<$: N:1, C:
			- $==$: Z:1, C:0, N:0

Compare ab	`CMP`	`01100`	Locally compare accumulators A and B.
Left Shift	`ASL[A|B]`	`01101`	Arithmetic shift A or B left by one.
Right Shift	`ASR[A|B]`	`01110`	Arithmetic shift A or B right by one.
Left Rotate	`ROL[A|B]`	`01111`	Logically rotate A or B left by one.
Right Rotate	`ROR[A|B]`	`10000`	Logically rotate A or B right by a value.
Push on Stack	`PSH[A|B]`	`10001`	Push the value of an accumulator on the stack.
Pull from Stack	`PUL[A|B]`	`10010`	Pull a value from the stack into an accumulator.
Stack Pointer set	`SSP location`	`10011`	Sets the stack pointer to address.

Function	Mnemonic	Binary	Description
Jump	`JMP location`	10100	Jump to location. If the ram flag was set to low, it is set to high and the content of the kernel are copied in the ram.
Branch on zero	`BEQ location`	10101	Branch to location if Zero flag is set.
Branch not zero	`BNE location`	10110	Branch to location if Zero flag is clear.
Branch on carry	`BCS location`	10111	Branch to location if Zero flag is set.
Branch not carry	`BCC location`	11000	Branch to location if Carry flag is clear.
Branch on negative	`BMI location`	11001	Branch to location if Negative flag is set.
Branch on positive	`BPL location`	11010	Branch to location if Negative flag is clear.
Jump to subroutine	`JSR location`	11011	Jump to virtual address label and push high and low bits of the programme counter on the stack.
Return from sub.	`RTS`	11100	Pull high and low bits from the stack into the programme counter.
Load X	`LDX location`	11101	Copy location into the index register.

Add to X	`ADX #value`
	`ADX location`
		11110	Add a value or the content of a memory address to Index Register.

Subtract from X	`SBX #value`
	`SBX location`
		11111	Subtract value or content of a memory address from the Index Register

2. VSP88 MACHINE OPCODES MAP

	None	A,none A,Imm.	B,none B,Imm.	Direct A,Direct	B,Direct	Index. A,Index.	B,Index.	A,Index. Indirect	B,Index. Indirect
HCF	**00**								
LDA[acc]		**08**	**09**	**0A**	**0B**	**0C**	**0D**	**0E**	**0F**
STR[acc]		**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
ADD[acc]		**18**	**19**	**1A**	**1B**	**1C**	**1D**	**1E**	**1F**
ADD	**20**								
SUB[acc]		**28**	**29**	**2A**	**2B**	**2C**	**2D**	**2E**	**2F**
SUB	**30**								
AND[acc]		**38**	**39**	**3A**	**3B**	**3C**	**3D**	**3E**	**3F**
AND	**40**								
LOR[acc]		**48**	**49**	**4A**	**4B**	**4C**	**4D**	**4E**	**4F**
LOR	**50**								
CMP[acc]		**58**	**59**	**5A**	**5B**	**5C**	**5D**	**5E**	**5F**
CMP	**60**								
ASL[acc]		**68**	**69**						
ASR[acc]		**70**	**71**						
ROL[acc]		**78**	**79**						
ROR[acc]		**80**	**81**						
PSH[acc]	**88**		**89**						
PUL[acc]	**90**		**91**						
SSP				**9A**					
JMP				**A2**		**A4**		**A6**	
BEQ				**AA**		**AC**		**AE**	
BNE				**B2**		**B4**		**B6**	
BCS				**BA**		**BC**		**BE**	
BCC				**C2**		**C4**		**C6**	
BMI				**CA**		**CC**		**CE**	
BPL				**D2**		**D4**		**D6**	
JSR				**DA**		**DC**		**DE**	
RTS	**E0**								
LDX				**EA**		**EC**		**EE**	
ADX		**F0**		**F2**					
SBX		**F8**		**FA**					

Number of Operation Codes: 107

3. ASSEMBLY AND ASSEMBLED PROGRAMME

Assembly Code	Assembled Machine Code
`.SOME_STRING "This programme returns the length of a string"`	
`_START:       LDAA   #0            ; clear A`	`08 EA`
`              LDX    SOME_STRING   ; load the string's address`	`EA 00 14`
`LOOP:         LDAB   X,0           ; load the character at X`	`0D 00 00`
`              BEQ    END           ; if B = 0, branch to END`	`AA 00 10`
`              ADDA   #1            ; increment the count by 1`	`18 01`
`              JMP    LOOP          ; jump to loop`	`A2 00 05`
`END:          HCF                  ; end execution`	`00`
	`54 68 69 73 20 70 72 6f`
	`67 72 61 6d 6d 65 20 63`
	`6f 75 6e 74 73 20 74 68`
	`65 20 63 68 61 72 61 63`
	`74 65 72 73 20 6f 66 20`
	`61 20 73 74 72 69 6e 67`
	`00`

4. SOURCE CODE AND VIDEO DEMONSTRATIONS

Source code for the virtual machine and for the assembler (written in C, compiled on a Mac OS X 10.10 machine) are available on GitHub:

- Apple II Bitmap Font: https://github.com/cesarparent/Apple-Bitmap-Font
- CPAsm & VSP88 Source Code: https://github.com/cesarparent/VSP88

Demonstration videos for the RomOS and Monitor, and for the pong test game have been uploaded to Youtube:

- VSP88's first assembled programme: https://youtu.be/5rU_sCumNyA
- VSP88/Monitor demonstration: https://youtu.be/12y-e6m0d0g
- VSP88/Pong demonstration: https://youtu.be/1gloIblb0D0

5. ROMOS/MONITOR AND PONG SOURCE CODE

The following pages are listings, in VSP88 assembly, of the two test programmes, RomOS/Monitor (described in II.4.1) and Pong, a player-versus-AI replica of the classic paddle game (described in II.4.2).

```
;.............................................................................
;   BOOTROM.S
;   VSP88 RomOS
;   Version 1.2
;   2015-03-26
;
;   RomOS and Monitor are loaded at address $f000 (in VSP88's ROM)
;.............................................................................
boot_msg:   .str    "VPS 88"
boot_ver:   .str    "RomOS 1.2 (MAR. 26, 2015)/Cesar
Parent\n----------------------------------------"
welcome:    .str    "Valid commands:\nR(UN)$$$$\nP(RINT)$$$$\nN(EXT)\nS(ET)$$$$ $$
[...]\nC(LEAR)\nH(ELP)\nQ(UIT)"
invalid:    .str    "Invalid command"
stop_msg:   .str    "Goodbye!"

; initialise processor
            LDAA    #-10
            LDX     $0400
            SSP     $3ff

; initialise display system
            JSR     DTXT
            LDAA    #17         ; start displaying text at X=17
            STRA    $07fd
            LDAA    #0          ; start displaying text at Y=0
            STRA    $07fe

            LDX     boot_msg    ; display boot message
            JSR     PRSTR
            LDAA    #127
            LDAB    #127
            JSR     DELAY
            JSR     CLRSCR
            LDX     boot_ver
            JSR     PRTITLE
            LDX     welcome
            JSR     PRSTR
            LDAB    #13
            JSR     VTAB

cmprompt:   LDAA    #93         ; jump-back point for command routines
            JSR     PRCHAR
            JSR     CLRFNBUF
            LDX     $0

            ; B SHOULD BE PRESERVED AT ALL COSTS: IT HOLDS THE CURRENT AMOUNT OF
CHARACTERS

inputl:     JSR     CRSBLK
            LDAA    $efff       ; load the keyboard mailbox
            BPL     inputl      ; check if MSB is set (new keystroke)
            ANDA    #$7f        ; unset MSB to get key's ASCII code
            STRA    $efff       ; store in memory to prevent infinite keystrokes
            STRA    X,$0100
            CMPA    #10
            BEQ     parse       ; if return was pressed, parse the
commandADX      #1
            ADX     #1
            JSR     PRCHAR  ; print character
            CMPA    #8
            BNE     _input_bp
            SBX     #2
            BPL     _input_bp
            LDX     $0
_input_bp:  JMP     inputl

;................................................................................
;   MONITOR
;................................................................................

CRSBLK:     PSHA
            PSHB
            LDAA    #20
            LDAB    #1
            JSR     DELAY
            LDAA    $0201
            SUBA    #1
            STRA    $0201
            CMPA    #$80
            BNE     _crs_end

            LDAA    $0200
            BEQ     _crs_show
            JMP     _crs_hide

_crs_show:  LDAB    #1
            STRB    $0200
            LDAA    #$5f
            LDAB    #$7f
            STRB    $0201
            STRA    $07fc
            JSR     DRWCHR
            JMP     _crs_end

_crs_hide:  LDAB    #0
            STRB    $0200
            LDAA    #32
            LDAB    #$7f
            STRB    $0201
            STRA    $07fc
            JSR     DRWCHR
            JMP     _crs_end

_crs_end:   PULB
            PULA
            RTS
;................................................................................
;   CLRSCR the function buffer
```

```
;..............................................................................
CLRFNBUF:   PSHB
            LDX     $0
            LDAA    #0
            LDAB    #127
_CLRBUF_l:  STRA    X,$0100
            ADX     #1
            SUBB    #1
            BNE     _CLRBUF_l
            PULB
            RTS

;..............................................................................
;   ROM USER FUNCTION
;..............................................................................
parse:      LDAB    #2
            JSR     VTAB
            LDAA    $0100
            CMPA    #$63
            BEQ     c_CLRSCR
            CMPA    #$68
            BEQ     c_help
            CMPA    #$6e
            BEQ     c_next
            CMPA    #$70
            BEQ     c_prtmon
            CMPA    #$72
            BEQ     c_run
            CMPA    #$71
            BEQ     c_exit
            CMPA    #$73
            BEQ     c_setmon
            JMP     c_invalid

_parse_end: LDAB    #2
            JSR     VTAB
            JMP     cmprompt

c_setmon:   LDX     $0101
            JSR     decodeaddr
            LDX     $0106
_setm_l:    JSR     PRSBYTE
            JSR     _setm_sta
            ADX     #3
            LDAA    X,0
            BNE     _setm_l
            JMP     cmprompt

_setm_sta:  LDX     0
            STRA    X^$000e
            JSR     incraddr
            RTS

c_prtmon:   LDX     $0101
            JSR     decodeaddr
            LDAA    #4
_prtmon_e:  PSHA
            LDAA    $000e
            JSR     PRBYTE
            LDAA    $000f
            JSR     PRBYTE
            LDAA    #$3a
            JSR     PRCHAR
            LDAA    #$20
            JSR     PRCHAR

            LDX     0
            LDAB    #8
_prtmon_l:  LDAA    X^$000e
            JSR     PRBYTE
            LDAA    #$20
            JSR     PRCHAR
            ADX     #1
            SUBB    #1
            BNE     _prtmon_l
            LDAA    #$20
            JSR     PRCHAR
            JSR     PRCHAR
            LDX     0
            LDAB    #8
_prtm_asc:  LDAA    X^$000e
            CMPA    #32
            BPL     _prtm_pr

            LDAA    #$2e
_prtm_pr:   JSR     PRCHAR
            ADX     #1
            SUBB    #1
            BNE     _prtm_asc
            JSR     incraddr8
            PULA
            SUBA    #1
            BNE     _prtmon_e

            JMP     _parse_end

c_next:     LDAA    #4
            JMP     _prtmon_e

c_CLRSCR:   LDX     boot_ver
            JSR     CLRSCR
            JSR     PRTITLE
            JMP     _parse_end

c_run:      LDX     $0101
            JSR     decodeaddr
            LDX     $0
            JSR     CLRSCR
            JSR     X^$000e
            LDX     boot_ver
            JSR     PRTITLE
            JMP     _parse_end

c_exit:     JSR     CLRSCR
            LDAA    #16
            STRA    $07fd
            LDAA    #12
            STRA    $07fe
            LDX     stop_msg
            JSR     PRSTR
            HCF

c_help:     LDX     welcome
            JSR     PRSTR
            JMP     _parse_end

c_invalid:  LDX     invalid
            JSR     PRSTR
            JMP     _parse_end

PRTITLE:    LDAA    #0
            STRA    $07fe
            LDAA    #0
            STRA    $07fd
            JSR     PRSTR
            LDAA    #23         ; switch insertion point to the bottom
            STRA    $07fe
            RTS

;...............................................................................
;   Decode address
;   decode a four-character hex address starting at X and puts it in $0f,$10
;...............................................................................
decodeaddr: PSHA
            JSR     PRSBYTE
            STRA    $000e
            ADX     #2
            JSR     PRSBYTE
            STRA    $000f
            PULA
            RTS

;...............................................................................
;   Increment address by 1
;   Increment the 16-bit address $0f
;...............................................................................
incraddr:   PSHA
            LDAA    $000f   ; low-order nibble
            CMPA    #$ff    ; if we're going to overflow, increment high-order
            BEQ     _iad_ho
            ADDA    #1
            STRA    $000f
            PULA
            RTS
_iad_ho:    LDAA    #0
```

```
            STRA    $000f
            LDAA    $000e   ; increment high order nibble and store it
            ADDA    #1
            STRA    $000e
            PULA
            RTS

;...............................................................................
;   Increment address by 8
;   Increment the 16-bit address $0f
;...............................................................................
incraddr8:  PSHA
            LDAA    $000f   ; low-order nibble
            CMPA    #$f7    ; if we're going to overflow, increment high-order
            BCC     _iad8_lo
            CMPA    #0
            BEQ     _iad8_lo
            BCS     _iad8_lo
            ADDA    #8
            STRA    $000f
            LDAA    $000e   ; increment high order nibble and store it
            ADDA    #1
            STRA    $000e
            PULA
            RTS

_iad8_lo:   ADDA    #8
            STRA    $000f
            PULA
            RTS
;...............................................................................
;   Parse byte from hex to memory
;   parses the two characters starting at X and put the value in A
;...............................................................................

PRSBYTE:    LDAA    #0
            JSR     parse_nib
            ASLB
            ASLB
            ASLB
            ASLB
            LOR
            ADX     #1
            JSR     parse_nib
            LOR
            RTS


parse_nib:  LDAB    X,0
            CMPB    #$30    ;compare with '0'
            BCC     _prsnb_err
            CMPB    #$3a
            BCC     _prsnb_dig
            CMPB    #$61
            BCC     _prsnb_err
            CMPB    #$67
            BCC     _prsnb_hex
            JMP     _prsnb_err

_prsnb_dig: SUBB    #48
            RTS
_prsnb_hex: SUBB    #87
            RTS
_prsnb_err: LDAB    #0
            RTS

;...............................................................................
;   String length
;   Return the length of the string pointed to by IDX in Accumulator B
;...............................................................................
strlen:     PSHA
            LDAB    #-1

_strll:     ADDB    #1
            ADX     #1
            LDAA    X,$0000
            BNE     _strll

_strl_end:  PULA
            RTS

;...............................................................................
;   String comparison
;   Compare strings at $00ff and $01ff
;   Zero flag is set if strings are indentical, CLRSCR otherwise
;       /param  $200    first string to compare
;       /param  IDX     second string to compare.
;...............................................................................
strcmp:     LDX     $0000       ; if the first character is '\0', return
_strcmp_lp: LDAA    X,$0000
            LDAB    X,$0000
            CMP
            BNE     _strcmp_nq
            ADX     #1
            CMPA    #0
            BNE     _strcmp_lp
_strcmp_o:  LDAA    #1
            RTS
_strcmp_nq: LDAA    #0
            RTS
;...............................................................................
;   Character comparison
;   Compares character in A to by X,$0300
;   !Destroys A!
;   A set to 0 if characters are equal, 1 otherwise
;...............................................................................
charcmp:    ADX     $0300
            LDAB    X,$0
            CMP
            BMI     _chcmp_nq
            BEQ     _chcmp_eq
_chcmp_eq:  LDAA    #0
            RTS
_chcmp_nq:  LDAB    #1
            RTS

;................................................................................
;   Display String
;   Print a string whose first character is at the address pointed at by
;   IDX, until a null character is encountered.
;   IDX is released to 0x0000 at the end of the execution
;       /param  0x07fd  Starting point X coordinate
;       /param  0x07fe  Starting point Y coordinate
;................................................................................
PRSTR:      PSHA
            LDAA    X,$0000

_dstrl:     JSR     PRCHAR
            ADX     #1
            LDAA    X,$0000
            CMPA    #$0a
            BEQ     _dstrl
            CMPA    #$1f
            BPL     _dstrl

_dstr_end:  PULA
            RTS

;................................................................................
;   Horizontal tab. Switch the insertion point by the number in B
;................................................................................
HTAB:       PSHA
            PSHB
            LDAA    #32
_HTABL:     JSR     PRCHAR
            SUBB    #1
            BNE     _HTABL
            PULB
            PULA
            RTS

VTAB:       PSHA
            PSHB
            LDAA    #10
_VTABL:     JSR     PRCHAR
            SUBB    #1
            BNE     _VTABL
            PULB
            PULA
            RTS
;................................................................................
;   Display Character
;   Start at address
;       /param  0x07fc  ASCII code
;       /param  0x07fd  X coordinate
;       /param  0x07fe  Y coordinate
;...............................................................................
DRWCHR:     PSHA
            PSHB
            LDX     $0          ; place IDX at start of Screen Memory
            LDAB    $07fe       ; load Y coordinate

            BEQ     _chloopend
_dchloop:   ADX     #40
            SUBB    #1
            BNE     _dchloop
_chloopend: ADX     $07fd       ; Add X coordinate

            LDAA    $07fc
            STRA    X,$0400
            PULB
            PULA
            RTS

;...............................................................................
;   Print Character
;   Display the character in A at the end of the currently occupied screen space
;       /param  Acc A   ASCII code
;...............................................................................
PRCHAR:     PSHA
            PSHB
            LDAB    #0
            STRB    $07fc
            JSR     DRWCHR
            CMPA    #31
            BPL     _pch_vis
            CMPA    #8
            BEQ     _pch_bs
            CMPA    #10
            BNE     _pch_end
            LDAA    #42
            JMP     _pch_wrap

_pch_end:   PULB
            PULA
            RTS

_pch_bs:    LDAA    $07fd
            CMPA    #2
            BMI     _pch_end
            LDAB    #0
            STRB    $07fc
            JSR     DRWCHR
            SUBA    #1
            STRA    $07fd
            JSR     DRWCHR
            JMP     _pch_end


_pch_wrap:  CMPA    #40
```

```
            BMI     _pch_end
            LDAA    #0
            LDAB    $07fe
            ADDB    #1
            CMPB    #23
            BPL     _pch_scrl
            STRA    $07fd
            STRB    $07fe
            JMP     _pch_end

_pch_scrl:  JSR     SCRLL
            LDAA    #0
            LDAB    #23
            STRA    $07fd
            STRB    $07fe
            JMP     _pch_end

_pch_vis:   STRA    $07fc
            JSR     DRWCHR
            LDAA    $07fd
            ADDA    #1
            STRA    $07fd
            JMP     _pch_wrap


;.................................................................................
;   SCRLL
;   Nudge the whole displayed screen by one line upwards, CLRSCRing a new
;   line for text/command entry and display
;.................................................................................
SCRLL:      PSHA
            PSHB
            LDX     $0
            LDAB    #21

_SCRLC: PSHB
            LDAB    #40
_SCRLL: LDAA    X,$0478
            STRA    X,$0450
            ADX     #1
            SUBB    #1
            BNE     _SCRLL
            PULB
            SUBB    #1
            BNE     _SCRLC

            LDAA    #0
            LDAB    #40
_SCRLCLR:   STRA    X,$0450
            ADX     #1
            SUBB    #1
            BNE     _SCRLCLR

            PULB
            PULA
            RTS

;.............................................................................
;   CLRSCR
;   CLRSCR the whole display
;.............................................................................
CLRSCR:     PSHA
            PSHB
            LDX     $0
            LDAB    #24

_CLRSCRCOL: PSHB
            LDAB    #40
            LDAA    #0
_CLRSCRL:   STRA    X,$0400
            ADX     #1
            SUBB    #1
            BNE     _CLRSCRL
            PULB
            SUBB    #1
            BNE     _CLRSCRCOL

            LDAA    #0
            LDAB    #23
            STRA    $07fd
            STRB    $07fe
            PULB
            PULA
            RTS


;.............................................................................
;   print the byte stored in A as hexadecimal
;.............................................................................
PRBYTE: PSHA
            ASRA
            ASRA
            ASRA
            ASRA
            JSR     _PRBYTEHX
            PULA

_PRBYTEHX:  ANDA    #$0f
            LORA    #$30    ;'0'
            CMPA    #$3a
            BCC     _ECHO
            ADDA    #7

_ECHO:      JSR     PRCHAR
            RTS

;-----------------------------------------------------------------------------
; Draw a pixel
; A Stores the X coordinate
; B Stores the Y coordinate
; $07FC stores the colour
;--------------------------------------------------------------------------------
DRPX:       PSHA
            PSHB
            STRA    $07fd       ; save X for later
            STRB    $07fe
            LDX     $0400       ; start at the beginning of screen memory
            ASRB                ; divide B by two (interlaced pixels)
            BEQ     _DRPXLEND   ; if line 0, bypass the loop
_DRPXL:     ADX     #40         ; add one line
            SUBB    #1          ; dec counter
            BNE     _DRPXL
_DRPXLEND:  ADX     $07fd

            LDAA    $07FC       ; load the colour
            LDAB    $07fe       ; Load actual Y coordinate
            ANDB    #$01        ; check if LSB is set (odd number)
            BEQ     _DPXEVEN
            BNE     _DPXODD

_DRPXEND:   LOR                 ; interlace both pixels' colours
            STRA    X,0         ; write pixel data to memory
            PULB                ; restore A and B, return
            PULA
            RTS

_DPXODD:    LDAB    X,0         ; load the two-pixel value
            ANDB    #$0F        ; clear the pixel that will be reset
            ASLA                ; shift the colour value to its right position
            ASLA
            ASLA
            ASLA
            JMP     _DRPXEND

_DPXEVEN:   LDAB    X,0         ; load the two-pixel value
            ANDB    #$F0        ; clear the pixel that will be reset
            ;ASLB               ; AND THE NEW PIXEL WITH ITS INTERLACED TWIN.
            JMP     _DRPXEND

;...............................................................................
;   Display mode switches
;...............................................................................
DTXT:       PSHA
            LDAA    #$00
            JMP     _DMODE

DGRAPH:     PSHA
            LDAA    #$FF
            ; fall through
_DMODE:     STRA    $07FF
            PULA
            RTS
;...............................................................................
;   Multiply A by 10
;...............................................................................
```

```
MUL10:      PSHB
            ASLA            ; multiply by 2
            PSHA            ; save x*2
            ASLA            ; multply by 2, x*4
            ASLA            ; multply by 2, x*8
            PULB            ; pull saved 2x in B
            ADD             ; add into A, A = x*8 + x*2
            PULB
            RTS
;...............................................................................
;   DELAY
;...............................................................................
DELAY:      PSHA
            PSHB
_DELAYL1:   ADDA    #0
_DELAYL2:   ADDB    #0
            SUBA    #1
            BNE     _DELAYL2
            SUBB    #1
            BNE     _DELAYL1
            PULB
            PULA
            RTS
```

```
;...............................................................................
;   VSP88 Pong + AI
;   Version 1.2
;   2015-03-26
;...............................................................................
lost_str:   .str    "You lost"
won_str:    .str    "You won!"

            ldaa    #$ff
            stra    $07FF

            ldaa    #5
            stra    $07D0
            stra    $07D1
            stra    $07FC

            ldaa    #15
            stra    $07EE
            stra    $07EF

            ldaa    #1
            stra    $07EC
            ldaa    #0
            stra    $07ED

game_loop:  JSR     paddle_1
            JSR     paddle_2
            JSR     draw_p1
            JSR     draw_p2
            JSR     draw_ball
            JSR     game_delay
            jmp     game_loop

game_loose: ldaa    #0
            JSR     CLRSCR      ; ROM clear
            stra    $07ff       ; switch to text mode
            ldab    #11
            strb    $07fe
            ldab    #16
            JSR     HTAB        ; ROM htab
            ldx     lost_str
            jmp     game_end

game_win:   ldaa    #0
            JSR     CLRSCR      ; ROM clear
            stra    $07ff       ; switch to text mode
            ldab    #11
            strb    $07fe
            ldab    #16
            JSR     HTAB        ; ROM htab
            ldx     won_str
            ; fall through
game_end:       JSR     PRSTR
            ldaa        #127
            ldab        #50
            JSR     DELAY
            JSR     CLRSCR
            ldaa        #0
            stra        $efff
            jsr     PRTITLE
            jmp     cmprompt

;.................................................................................
;  Pong paddle moving routine.
;  Update the position of the first paddle
;.................................................................................

paddle_1:   psha
            pshb

            ldab    $07D0
            ldaa    $EFFF
            anda    #$80
            beq     _pad1_o
            ldaa    $EFFF
            anda    #$7f
            stra    $EFFF

            cmpa    #113         ; compare with q, going up
            beq     _pad1_up

            cmpa    #122
            beq     _pad1_down

_pad1_o:    pulb
            pula
            RTS

_pad1_up:   cmpb    #1
            bmi     _pad1_o
            subb    #1
            strb    $07D0
            pulb
            pula
            RTS

_pad1_down: cmpb    #39
            beq     _pad1_o
            addb    #1
            strb    $07D0
            pulb
            pula
            RTS

;.................................................................................
;  Pong paddle 2 ai
;  Update the position of the first paddle (stored in address data segment+0)
;.................................................................................
```

```
paddle_2:   psha
            pshb
            ldaa    $07ee
            cmpa    #13
            bmi     _pad2_o

            ldab    $07D1
            ldaa    $07EF
            suba    #4

            cmp
            beq     _pad2_o
            bpl     _pad2_down
            bmi     _pad2_up

_pad2_o:    pulb
            pula
            RTS

_pad2_up:   cmpb    #2
            bmi     _pad2_o
            subb    #1
            strb    $07D1
            jmp     _pad2_o
            pulb
            pula
            RTS

_pad2_down: cmpb    #39
            beq     _pad2_o
            addb    #1
            strb    $07D1
            jmp     _pad2_o



;................................................................................
;  Game Routine
;  clear and draw the first player's paddle
;................................................................................

draw_p1:    psha
            pshb

            ldab    #0      ; colour
            strb    $07fc
            strb    $07fd
            ldaa    $07d0
            beq     _dp1_pad
            suba    #1
            stra    $07fe
            JSR     draw_pixel
            adda    #1
```

```
_dp1_pad:   ldab    #15
            strb    $07fc   ;store paddle colour
            ldab    #9

_dp1_loop:  stra    $07fe
            JSR     draw_pixel
            adda    #1
            subb    #1
            bne     _dp1_loop

            ldab    #0
            strb    $07fc
            stra    $07fe
            JSR     draw_pixel

_dp1_end:   pulb
            pula
            RTS

;.................................................................................
;  Game Routine
;  clear and draw the second player's paddle
;.................................................................................

draw_p2:    psha
            pshb

            ldab    #0      ; colour
            strb    $07fc
            ldab    #39
            strb    $07fd
            ldaa    $07d1
            beq     _dp2_pad
            suba    #1
            stra    $07fe
            JSR     draw_pixel
            adda    #1

_dp2_pad:   ldab    #15
            strb    $07fc   ;store paddle colour
            ldab    #9

_dp2_loop:  stra    $07fe
            JSR     draw_pixel
            adda    #1
            subb    #1
            bne     _dp2_loop

            ldab    #0
            strb    $07fc
            stra    $07fe
            JSR     draw_pixel
```

```
_dp2_end:   pulb
            pula
            RTS

;...............................................................................
;  ROM Utility Routine
;  Draw vertical line of pixels
;  starts at the coordinate stored in $07FD (X), $07FE (Y)
;  draws downloads for $07FA pixels.
;       $07EC stores x speed
;       $07ED stores y speed
;       $07EE stores x position
;       $07EF stores y position
;...............................................................................

draw_ball:  psha
            pshb

            ldaa    #0
            stra    $07FC

            ldaa    $07EE       ; load x
            ldab    $07EF       ; load y
            JSR     swap_vert   ; change X speed if needed
            JSR     swap_hor    ; change Y speed if needed
            stra    $07FD       ; clear the previous position (draw a black pixel)
            strb    $07FE
            JSR     draw_pixel

            ldaa    #6          ; load ball colour
            stra    $07FC       ; store it for draw_pixel to use

            ldaa    $07EE       ; load x
            adda    $07EC       ; add the velocity to X,Y
            addb    $07ED
            stra    $07FD       ; store the position in draw_pixel's mailbox
            strb    $07FE
            JSR     draw_pixel

            stra    $07EE
            strb    $07EF

            pulb
            pula
            RTS

;...............................................................................
;  Game Routines
;  flip the ball's direction when it hits a wall
;  X is in accumulator A
;  Y is in accumulator B
;...............................................................................

swap_vert:  cmpb    #47
```

```
            bpl     swap_up
            cmpb    #1
            bmi     swap_down
            RTS

swap_up:    pshb
            ldab    #-1
            strb    $07ed
            pulb
            RTS

swap_down:  pshb
            ldab    #1
            strb    $07ed
            pulb
            RTS

swap_dir:   pula
            pshb
            ldab    #0
            strb    $07ed
            pulb
            RTS

swap_hor:   cmpa    #37
            bpl     swap_left
            cmpa    #2
            bmi     swap_right
            RTS

swap_left:  JSR     check_won
            psha
            ldaa    #-1
            stra    $07ec
            ldaa    $07d1
            adda    #4
            ldab    $07ef
            cmp
            beq     swap_dir
            pula
            bcs     swap_up
            jmp     swap_down
            RTS

swap_right: JSR     check_lost
            psha
            ldaa    #1
            stra    $07ec
            ldaa    $07d0
            adda    #4
            ldab    $07ef
            cmp
            beq     swap_dir
            pula
            bcs     swap_up
            jmp     swap_down
            RTS

check_won:  psha                ; save A to make space for AI paddle
            ldaa    $07D1
            cmp
            bpl     game_win
            adda    #8
            cmp
            bmi     game_win
            pula
            RTS

check_lost: psha                ; save A to make space for AI paddle
            ldaa    $07D0
            cmp
            bpl     game_loose
            adda    #8
            cmp
            bmi     game_loose
            pula
            RTS


game_delay: psha
            pshb
            ldaa    #10
oLoop:      ldab    #127
dLoop:      adda    #0
            subb    #1
            bne     dLoop
            suba    #1
            bne     oLoop
            pulb
            pula
            RTS

;...................................................................................
;  ROM Utility Routine
;  Draw pixel at given coordinates
;  /param   $07FC   color/character
;  /param   $07FD   X coordinate (0 to 40)
;  /param   $07FE   Y coordinate (0 to 48)
;...................................................................................

draw_pixel: psha
            pshb
            ldx     $0400
            ldab    $07FE       ; load the Y Coordinate
            asrb                ; divide Y by two to interlace pixels (addresses 0x0400 ->
0x07c0)
            pshb
            beq     _pxloopend
_dpxloop:   adx     #40         ; add a line to the Index register (20, two pixels per byte)
            subb    #1          ; remove 1 from the counter
            bne     _dpxloop    ; Divide A by two (two pixels per byte)
_pxloopend: adx     $07FD

            ldaa    $07FC       ; load the colour in A
            pulb
            aslb                ; multiply Y by two. If (A/2)*2 = Y, then the Y coordinate
is even
            cmpb    $07FE
            beq     _px_even
            bne     _px_odd

_px_end:    lor
            stra    X,$0000
            pulb
            pula
            RTS

_px_odd:    ldab    X,$0000     ; load the current value into B
            andb    #15
            asla
            asla
            asla
            asla
            jmp     _px_end

_px_even:   ldab    X,$0000     ; load the current value into B
            andb    #-16        ; clear the current pixel
            aslb                ; and the new pixel with its interlaced twin.
            jmp     _px_end
```