The AppleCore Virtual Machine Specification, v1.0

Robert L. Bocchino Jr.
Pittsburgh, PA

January 8, 2012

1 Introduction and Rationale

The AppleCore Virtual Machine (AVM) is a virtual machine architecture for the AppleCore programming language. Its purposes are:

1. To provide a virtual instruction set that closely models the stack semantics of the AppleCore language. The virtual instruction set can be used as a single intermediate representation for AppleCore source programs that is translated to either 6502 code or AVM bytecode.
2. To provide a bytecode representation that stores AppleCore programs in a very compact way (using about 2.5x fewer bytes than native code). This allows larger programs to be stored in memory. However, runtime interpretation of AppleCore bytecode is slower than running native code; exactly how much slower remains to be seen.

2 Integration with 6502 Code

The AVM instruction set is designed to represent AppleCore functions in a way that interoperates with 6502 code. In particular, caller code should have no way of knowing whether a called function is implemented in 6502 or AVM instructions. Also, interpreted AVM instructions must be able to call native code, because the AppleCore specification says that calls to AppleCore functions and regular assembly language functions are interchangeable.

These requirements are met as follows. As is typical for virtual machine architectures on the Apple (for example, see Steve Wozniak's Sweet-16 interpreter), interpretation starts when the 6502 code does a `JSR` to the interpreter. The interpreter pulls the "return address" off the Apple II stack and uses it to determine where to start interpreting bytecode: all the bytes following the `JSR` instruction are interpreted as AVM code, up to and including the first `RAF` (Return from AppleCore Function) encountered. Executing `RAF` returns 6502 control to the return address that was on the stack when the interpreter was invoked.

In code translated from AppleCore source, each function starts with a `JSR` to the interpreter, and all statements in the function are translated to AVM code. That way, the function works normally when called from normal 6502 code. Also, subroutine calls within AVM code work the same way regardless of whether the callee function contains AVM or native code.

3 Instruction Set

The AVM instruction set has three kinds of instructions: unsized instructions, sized instructions, and signed instructions.

3.1 Unsized Instructions

BRK (Break, Opcode $00): Causes the AVM interpreter to execute a 6502 `BRK` instruction.

BRF (Branch on Result False, Opcode $01): Causes the AVM interpreter to pull a single byte off the AppleCore program stack. If the byte evaluates to true (i.e., has its low bit set, see Section 4.1 of the AppleCore Language Specification), then interpretation continues with the instruction located three bytes after this one. Otherwise, control branches to the address given by the next two bytes.

BRU (Branch Unconditionally, Opcode $02): Causes control to branch to the address given by the two bytes following the instruction opcode.

CFD (Call Function Direct, Opcode $03): Causes the AVM interpreter to execute a `JSR` to the address given by the two bytes following the instruction opcode. On return from the `JSR`, execution resumes with the third byte following the instruction opcode.

CFI (Call Function Indirect, Opcode $04): Causes the AVM interpreter to pull two bytes off the stack and do a `JSR` to code that branches to the address given by those two bytes. On return from the `JSR`, execution resumes with the byte following the instruction opcode.

NOP (No Operation, Opcode $EA): Causes the AVM interpreter to skip the instruction and continue execution with the byte following the instruction opcode.

3.2 Sized Instructions

TODO

3.3 Signed Instructions

TODO

3.4 Unused Opcodes

TODO

4 Table of Opcodes

TODO