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. The call to the interpreter also stores the callee's frame pointer on the stack, as specified in Section 5.4 of the AppleCore language specification.

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

Every sized instruction has a one-byte unsigned size argument. If the value of the size argument is between 0 and 6 inclusive, then the size argument is stored in the low-order 3 bits of the opcode; otherwise the low-order 3 bits are all ones and the size is stored in the byte immediately following the opcode. Instructions `MTV`, `PHC`, and `VTM` each have an additional argument which is stored in the byte or bytes immediately following the one or two bytes representing the opcode and size.

ADD (Add, Opcodes $08–$0F): Pull two *size*-byte values off the stack, add the values ignoring overflow, and push the result on the stack.

ANL (And Logical, Opcodes $10–$17): Pull two *size*-byte values off the stack, compute the bitwise logical and of the values, and push the result on the stack.

DCR (Decrement Memory, Opcodes $18–$1F): Pull a two-byte address off the stack and decrement the *size*- byte value stored at that address in memory. Push the result on the stack.

DSP (Decrease Stack Pointer, Opcodes $20–$27): Decrease the AppleCore program stack pointer by *size*.

ICR (Increment Memory, Opcodes $28–2F): Pull a two-byte address off the stack and increment the *size*-byte value stored at that address in memory. Push the result on the stack.

ISP (Increase Stack Pointer, Opcodes $30–$37): Decrease the AppleCore program stack pointer by *size*.

MTS (Memory To Stack, Opcodes $38–$3F): Pull a two-byte address off the stack. Push the *size* bytes in memory starting at that address on the stack.

MTV (Memory To Variable, Opcodes $40–$47): Treat the byte following the opcode and size as a zero-page address. Read the value from that address and store it at address `FP` + *size*, where `FP` is the AppleCore frame pointer.

NEG (Arithmetic Negation, Opcodes $48–$4F): Pull a *size*-byte value off the stack, negate the value (i.e., form the two's complement), and push the result on the stack.

NOT (Logical Negation, Opcodes $50–$57): Pull a *size*-byte value off the stack, logically negate the bits of the value, and push the result on the stack.

ORL (Or Logical, Opcodes $58–$5F): Pull two *size*-byte values off the stack, form the logical or of the two values, and push the result on the stack.

ORX (Or Exclusive, Opcodes $60–$67): Pull two *size*-byte values off the stack, form the logical exclusive or of the two values, and push the result on the stack.

PHC (Push Constant, Opcodes $68–$6F): Interpret the *size* bytes following the opcode and size as a *size*-byte constant. Push that constant on the stack.

PVA (Push Variable Address, Opcodes $70–$77): Push the two-byte address given by `FP` + *size* on the stack.

RAF (Return from AppleCore Function, Opcodes $78–$7F): Pull *size* bytes off the stack. Set the stack pointer to the frame pointer. Restore the caller's frame pointer. Push the *size* bytes on the stack. Return control to the address on the 6502 stack.

SHL (Shift Left, Opcodes $80–$87): Pull a single-byte unsigned shift amount off the stack. Pull a *size*-byte value off the stack, shift it left by the shift amount, and push the result on the stack.

STM (Stack To Memory, Opcodes $87–$8F): Pull a two-byte address off the stack. Pull a *size*-byte value off the stack and store it in memory starting at the address.

SUB (Subtract, Opcodes $90–$97): Pull two *size*-byte values off the stack, subtract the second one from the first one, and push the result on the stack.

TEQ (Test Equal, Opcodes $98–$9F): Pull two *size*-byte values off the stack. Push 1 on the stack if they are equal, 0 if they are not equal.

VTM (Variable To Memory, Opcodes $A0–$A7): Treat the byte following the opcode and size as a zero-page address. Read the value at address `FP` + *size*, where `FP` is the AppleCore frame pointer, and store it at the zero-page address.

3.3 Signed Instructions

TODO

3.4 Unused Opcodes

TODO

4 Table of Opcodes

TODO