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 provides a uniform representation for AppleCore source programs that may be 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 2x–3x 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.

Note that in most cases an instruction with size zero is actually a no-op (for example, `ADD` with size zero does nothing). However, in some cases (for example `PVA`) the size-zero instruction is meaningful.

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, 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, 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 (Negate, 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 (Not, 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

Signed instructions work similarly to sized instructions, except that the there is a sign argument and a size argument. Bit 2 of the opcode represents the sign (1 is signed 0, is unsigned). If the size is between 0 and 2 inclusive, then the size is stored in the low-order 2 bits of the opcode; otherwise the low-order 2 bits are all ones and size is stored in the byte immediately following the opcode.

DIV (Divide, Opcodes $A8–$AF): Pull a *size*-byte dividend and then a *size*-byte divisor off the stack. Do signed or unsigned division, and push the quotient on the stack.

EXT (Extend, Opcodes $B0–$B7): Sign- or zero-extend the value on the the top of the stack by *size* bytes: Push *size* `$00` or `$FF` bytes on the stack; push `$FF` if and only if the operation is sign extension and the top byte on the stack has its high bit set.

MUL (Multiply, Opcodes $B8–$BF): Pull two *size*-byte values off the stack, do signed or unsigned multiplication of the values, and push the result on the stack.

SHR (Shift Right, Opcodes $C0–$C7): 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. The shift is either signed (i.e., shift in the sign bit) or unsigned (i.e., shift in zero).

TGE (Test Greater or Equal, Opcodes $C8–$CF): Pull two *size*-byte values off the stack. Do a signed or unsigned comparison of the values. Push 1 on the stack if the second value pulled is greater than or equal to the first value pulled; push 0 otherwise.

TGT (Test Greater Than, Opcodes $D0–$D7): Pull two *size*-byte values off the stack. Do a signed or unsigned comparison of the values. Push 1 on the stack if the second value pulled is greater than the first value pulled; push 0 otherwise.

TLE (Test Less or Equal, Opcodes $D8–$DF): Pull two *size*-byte values off the stack. Do a signed or unsigned comparison of the values. Push 1 on the stack if the second value pulled is less than or equal to the first value pulled; push 0 otherwise.

TLT (Test Less Than, Opcodes $E0–$E7): Pull two *size*-byte values off the stack. Do a signed or unsigned comparison of the values. Push 1 on the stack if the second value pulled is less than the first value pulled; push 0 otherwise.

3.4 Unused Opcodes

Opcodes $05–$07, $E8–$E9, and $EB-$FF are not used for AVM instructions. The AVM interpreter treats these opcodes as equivalent to $EA (`NOP`).

4 Table of Instructions

Hex Bytes	Assembly	Instruction Name
00	BRK	Break
01 L H	BRF $$HL$$	Branch False
02 L H	BRU $$HL$$	Branch Unconditional
03 L H	CFD $$HL$$	Call Function Direct
04 L H	CFI $$HL$$	Call Function Indirect
05	???	Unused
06	???	Unused
07	???	Unused
08	ADD $0	Add
09	ADD $1	Add
0A	ADD $2	Add
0B	ADD $3	Add
0C	ADD $4	Add
0D	ADD $5	Add
0E	ADD $6	Add
0F S	ADD $$S$$	Add
10	ANL $0	And Logical
11	ANL $1	And Logical
12	ANL $2	And Logical
13	ANL $3	And Logical
14	ANL $4	And Logical
15	ANL $5	And Logical
16	ANL $6	And Logical
17 S	ANL $$S$$	And Logical
18	DCR $0	Decrement
19	DCR $1	Decrement
1A	DCR $2	Decrement
1B	DCR $3	Decrement
1C	DCR $4	Decrement
1D	DCR $5	Decrement
1E	DCR $6	Decrement
1F S	DCR $$S$$	Decrement
20	DSP $0	Decrease SP
21	DSP $1	Decrease SP
22	DSP $2	Decrease SP
23	DSP $3	Decrease SP
24	DSP $4	Decrease SP
25	DSP $5	Decrease SP
26	DSP $6	Decrease SP
27 S	DSP $$S$$	Decrease SP
28	ICR $0	Increment
29	ICR $1	Increment
2A	ICR $2	Increment
2B	ICR $3	Increment
2C	ICR $4	Increment
2D	ICR $5	Increment
2E	ICR $6	Increment
2F S	ICR $$S$$	Increment

Hex Bytes	Assembly Format	Instruction Name
30	ISP $0	Increase SP
31	ISP $1	Increase SP
32	ISP $2	Increase SP
33	ISP $3	Increase SP
34	ISP $4	Increase SP
35	ISP $5	Increase SP
36	ISP $6	Increase SP
37 S	ISP $$S$$	Increase SP
38	MTS $0	Mem To Stack
39	MTS $1	Mem To Stack
3A	MTS $2	Mem To Stack
3B	MTS $3	Mem To Stack
3C	MTS $4	Mem To Stack
3D	MTS $5	Mem To Stack
3E	MTS $6	Mem To Stack
3F S	MTS $$S$$	Mem To Stack
40 A	MTV $0<-$$A$	Mem To Var
41 A	MTV $1<-$$A$	Mem To Var
42 A	MTV $2<-$$A$	Mem To Var
43 A	MTV $3<-$$A$	Mem To Var
44 A	MTV $4<-$$A$	Mem To Var
45 A	MTV $5<-$$A$	Mem To Var
46 A	MTV $6<-$$A$	Mem To Var
47 S A	MTV $$S$<-$$A$	Mem To Var
48	NEG $0	Negate
49	NEG $1	Negate
4A	NEG $2	Negate
4B	NEG $4	Negate
4C	NEG $3	Negate
4D	NEG $5	Negate
4E	NEG $6	Negate
4F S	NEG $$S$$	Negate
50	NOT $0	Not
51	NOT $1	Not
52	NOT $2	Not
53	NOT $4	Not
54	NOT $3	Not
55	NOT $5	Not
56	NOT $6	Not
57 S	NOT $$S$$	Not
58	ORL $0	Or Logical
59	ORL $1	Or Logical
5A	ORL $2	Or Logical
5B	ORL $3	Or Logical
5C	ORL $4	Or Logical
5D	ORL $5	Or Logical
5E	ORL $6	Or Logical
5F S	ORL $$S$$	Or Logical

Hex Bytes	Assembly	Name	Hex Bytes	Assembly	Name
60	ORX $0	Or Exclusive	90	SUB $0	Subtract
61	ORX $1	Or Exclusive	91	SUB $1	Subtract
62	ORX $2	Or Exclusive	92	SUB $2	Subtract
63	ORX $4	Or Exclusive	93	SUB $4	Subtract
64	ORX $3	Or Exclusive	94	SUB $3	Subtract
65	ORX $5	Or Exclusive	95	SUB $5	Subtract
66	ORX $6	Or Exclusive	96	SUB $6	Subtract
67 S	ORX $$S$	Or Exclusive	97 S	SUB $$S$	Subtract
68	PHC	Push Constant	98	TEQ	Test Equal
69 C	PHC $$C$	Push Constant	99	TEQ $1	Test Equal
6A C_1 C_2	PHC $$C_2C_1$	Push Constant	9A	TEQ $2	Test Equal
6C C_1 C_2 C_3	PHC $$C_3C_2C_1$	Push Constant	9C	TEQ $3	Test Equal
6C $C_1 \ldots C_4$	PHC $$C_4 \ldots C_1$	Push Constant	9C	TEQ $4	Test Equal
6D $C_1 \ldots C_5$	PHC $$C_5 \ldots C_1$	Push Constant	9D	TEQ $5	Test Equal
6E $C_1 \ldots C_6$	PHC $$C_6 \ldots C_1$	Push Constant	9E	TEQ $6	Test Equal
6F S $C_1 \ldots C_S$	PHC $$C_S \ldots C_1$	Push Constant	9F S	TEQ $$S$	Test Equal
70	PVA $0	Push Var Addr	A0 A	VTM $0->$$A$	Var To Mem
71	PVA $1	Push Var Addr	A1 A	VTM $1->$$A$	Var To Mem
72	PVA $2	Push Var Addr	A2 A	VTM $2->$$A$	Var To Mem
73	PVA $4	Push Var Addr	A3 A	VTM $4->$$A$	Var To Mem
74	PVA $3	Push Var Addr	A4 A	VTM $3->$$A$	Var To Mem
75	PVA $5	Push Var Addr	A5 A	VTM $5->$$A$	Var To Mem
76	PVA $6	Push Var Addr	A6 A	VTM $6->$$A$	Var To Mem
77 S	PVA $$S$	Push Var Addr	A7 S A	VTM $$S->$$A$	Var To Mem
78	RAF $0	Rtn from AC Fn	A8	DIV $0	Divide
79	RAF $1	Rtn from AC Fun	A9	DIV $1	Divide
7A	RAF $2	Rtn from AC Fun	AA	DIV $2	Divide
7B	RAF $3	Rtn from AC Fun	AB S	DIV $$S$	Divide
7C	RAF $4	Rtn from AC Fun	AC	DIV $0S	Divide
7D	RAF $5	Rtn from AC Fun	AD	DIV $1S	Divide
7E	RAF $6	Rtn from AC Fun	AE	DIV $2S	Divide
7F S	RAF $$S$	Rtn from AC Fun	AF S	DIV $$SS	Divide
80	SHL $0	Shift Left	B0	EXT $0	Extend
81	SHL $1	Shift Left	B1	EXT $1	Extend
82	SHL $2	Shift Left	B2	EXT $2	Extend
83	SHL $4	Shift Left	B3 S	EXT $$S$	Extend
84	SHL $3	Shift Left	B4	EXT $0S	Extend
85	SHL $5	Shift Left	B5	EXT $1S	Extend
86	SHL $6	Shift Left	B6	EXT $2S	Extend
87 S	SHL $$S$	Shift Left	B7 S	EXT $$SS	Extend
88	STM $0	Stack To Mem	B8	MUL $0	Multiply
89	STM $1	Stack To Mem	B9	MUL $1	Multiply
8A	STM $2	Stack To Mem	BA	MUL $2	Multiply
8B	STM $3	Stack To Mem	BB S	MUL $$S$	Multiply
8C	STM $4	Stack To Mem	BC	MUL $0S	Multiply
8D	STM $5	Stack To Mem	BD	MUL $1S	Multiply
8E	STM $6	Stack To Mem	BE	MUL $2S	Multiply
8F S	STM $$S$	Stack To Mem	BF S	MUL $$SS	Multiply

Hex Bytes	Assembly	Name
C0	SHR $0	Shift Right
C1	SHR $1	Shift Right
C2	SHR $2	Shift Right
C3 S	SHR $$S$	Shift Right
C4	SHR $0S	Shift Right
C5	SHR $1S	Shift Right
C6	SHR $2S	Shift Right
C7 S	SHR $$S$S	Shift Right
C8	TGE $0	Test Greater or Equal
C9	TGE $1	Test Greater or Equal
CA	TGE $2	Test Greater or Equal
CC S	TGE $$S$	Test Greater or Equal
CC	TGE $0S	Test Greater or Equal
CD	TGE $1S	Test Greater or Equal
CE	TGE $2S	Test Greater or Equal
CF S	TGE $$S$S	Test Greater or Equal
D0	TGT $0	Test Greater Than
D1	TGT $1	Test Greater Than
D2	TGT $2	Test Greater Than
D3	TGT $$S$	Test Greater Than
D4	TGT $0S	Test Greater Than
D5	TGT $1S	Test Greater Than
D6	TGT $2S	Test Greater Than
D7 S	TGT $$S$S	Test Greater Than
D8	TLE $0	Test Less or Equal
D9	TLE $1	Test Less or Equal
DA	TLE $2	Test Less or Equal
DB S	TLE $$S$	Test Less or Equal
DC	TLE $0S	Test Less or Equal
DD	TLE $1S	Test Less or Equal
DE	TLE $2S	Test Less or Equal
DF S	TLE $$S$S	Test Less or Equal

Hex Bytes	Assembly	Name
E0	TLT $0	Test Less Than
E1	TLT $1	Test Less Than
E2	TLT $2	Test Less Than
E3 S	TLT $$S$	Test Less Than
E4	TLT $0S	Test Less Than
E5	TLT $1S	Test Less Than
E6	TLT $2S	Test Less Than
E7 S	TLT $$S$S	Test Less Than
E8	???	Unused
E9	???	Unused
EA	NOP	No Operation
EB	???	Unused
EC	???	Unused
ED	???	Unused
EE	???	Unused
EF	???	Unused
F0	???	Unused
F1	???	Unused
F2	???	Unused
F3	???	Unused
F4	???	Unused
F5	???	Unused
F6	???	Unused
F7	???	Unused
F8	???	Unused
F9	???	Unused
FA	???	Unused
FB	???	Unused
FC	???	Unused
FD	???	Unused
FE	???	Unused
FF	???	Unused