The AppleCore Virtual Machine Specification, v1.0

Robert L. Bocchino Jr.
Pittsburgh, PA

January 9, 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.

3.2 Sized Instructions

Every sized instruction has a one-byte unsigned size argument. If the value of the size argument is between 1 and 7 inclusive, then the size argument is stored in the low-order 3 bits of the opcode; otherwise the low-order 3 bits are all zeros, 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. Counting the low-order bit as 0, bit 2 of the opcode represents the sign (1 is signed 0, is unsigned). If the size is between 1 and 3 inclusive, then the size is stored in the low-order 2 bits of the opcode; otherwise the low-order 2 bits are both zeros, 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 signed 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 right 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 and $E8–$FF are not currently used for AVM instructions. They are reserved for possible future use (for example, to add floating point instructions). The AVM interpreter treats these opcodes as equivalent to $00 (`BRK`).

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	CFI	Call Function Indirect
05	???	Unused
06	???	Unused
07	???	Unused
08 S	ADD $$S$$	Add
09	ADD $01	Add
0A	ADD $02	Add
0B	ADD $03	Add
0C	ADD $04	Add
0D	ADD $05	Add
0E	ADD $06	Add
0F	ADD $07	Add
10 S	ANL $$S$$	And Logical
11	ANL $01	And Logical
12	ANL $02	And Logical
13	ANL $03	And Logical
14	ANL $04	And Logical
15	ANL $05	And Logical
16	ANL $06	And Logical
17	ANL $07	And Logical
18 S	DCR $$S$$	Decrement
19	DCR $01	Decrement
1A	DCR $02	Decrement
1B	DCR $03	Decrement
1C	DCR $04	Decrement
1D	DCR $05	Decrement
1E	DCR $06	Decrement
1F	DCR $07	Decrement
20 S	DSP $$S$$	Decrease SP
21	DSP $01	Decrease SP
22	DSP $02	Decrease SP
23	DSP $03	Decrease SP
24	DSP $04	Decrease SP
25	DSP $05	Decrease SP
26	DSP $06	Decrease SP
27	DSP $07	Decrease SP
28 S	ICR $$S$$	Increment
29	ICR $01	Increment
2A	ICR $02	Increment
2B	ICR $03	Increment
2C	ICR $04	Increment
2D	ICR $05	Increment
2E	ICR $06	Increment
2F	ICR $07	Increment

Hex Bytes	Assembly Format	Instruction Name
30 S	ISP $$S$$	Increase SP
31	ISP $01	Increase SP
32	ISP $02	Increase SP
33	ISP $03	Increase SP
34	ISP $04	Increase SP
35	ISP $05	Increase SP
36	ISP $06	Increase SP
37	ISP $07	Increase SP
38 S	MTS $$S$$	Mem To Stack
39	MTS $01	Mem To Stack
3A	MTS $02	Mem To Stack
3B	MTS $03	Mem To Stack
3C	MTS $04	Mem To Stack
3D	MTS $05	Mem To Stack
3E	MTS $06	Mem To Stack
3F	MTS $07	Mem To Stack
40 S A	MTV $$S$<-$$A$	Mem To Var
41 A	MTV $01<-$$A$	Mem To Var
42 A	MTV $02<-$$A$	Mem To Var
43 A	MTV $03<-$$A$	Mem To Var
44 A	MTV $04<-$$A$	Mem To Var
45 A	MTV $05<-$$A$	Mem To Var
46 A	MTV $06<-$$A$	Mem To Var
47 A	MTV $07<-$$A$	Mem To Var
48 S	NEG $$S$$	Negate
49	NEG $01	Negate
4A	NEG $02	Negate
4B	NEG $04	Negate
4C	NEG $03	Negate
4D	NEG $05	Negate
4E	NEG $06	Negate
4F	NEG $07	Negate
50 S	NOT $$S$$	Not
51	NOT $01	Not
52	NOT $02	Not
53	NOT $04	Not
54	NOT $03	Not
55	NOT $05	Not
56	NOT $06	Not
57	NOT $07	Not
58 S	ORL $$S$$	Or Logical
59	ORL $01	Or Logical
5A	ORL $02	Or Logical
5B	ORL $03	Or Logical
5C	ORL $04	Or Logical
5D	ORL $05	Or Logical
5E	ORL $06	Or Logical
5F	ORL $07	Or Logical

Hex Bytes	Assembly	Name	Hex Bytes	Assembly	Name
60 S	ORX $$S$	Or Exclusive	90 S	SUB $$S$	Subtract
61	ORX $01	Or Exclusive	91	SUB $01	Subtract
62	ORX $02	Or Exclusive	92	SUB $02	Subtract
63	ORX $04	Or Exclusive	93	SUB $04	Subtract
64	ORX $03	Or Exclusive	94	SUB $03	Subtract
65	ORX $05	Or Exclusive	95	SUB $05	Subtract
66	ORX $06	Or Exclusive	96	SUB $06	Subtract
67	ORX $07	Or Exclusive	97	SUB $07	Subtract
68 $S\ C_1 \ldots C_S$	PHC $$C_S \ldots C_1$	Push Constant	98 S	TEQ $$S$	Test Equal
69 C	PHC $$C$	Push Constant	99	TEQ $01	Test Equal
6A $C_1\ C_2$	PHC $$C_2C_1$	Push Constant	9A	TEQ $02	Test Equal
6C $C_1\ C_2\ C_3$	PHC $$C_3C_2C_1$	Push Constant	9C	TEQ $03	Test Equal
6C $C_1 \ldots C_4$	PHC $$C_4 \ldots C_1$	Push Constant	9C	TEQ $04	Test Equal
6D $C_1 \ldots C_5$	PHC $$C_5 \ldots C_1$	Push Constant	9D	TEQ $05	Test Equal
6E $C_1 \ldots C_6$	PHC $$C_6 \ldots C_1$	Push Constant	9E	TEQ $06	Test Equal
6F $C_1 \ldots C_7$	PHC $$C_7 \ldots C_1$	Push Constant	9F	TEQ $07	Test Equal
70 S	PVA $$S$	Push Var Addr	A0 $S\ A$	VTM $$S$->$$A$	Var To Mem
71	PVA $01	Push Var Addr	A1 A	VTM $01->$$A$	Var To Mem
72	PVA $02	Push Var Addr	A2 A	VTM $02->$$A$	Var To Mem
73	PVA $03	Push Var Addr	A3 A	VTM $04->$$A$	Var To Mem
74	PVA $04	Push Var Addr	A4 A	VTM $03->$$A$	Var To Mem
75	PVA $05	Push Var Addr	A5 A	VTM $05->$$A$	Var To Mem
76	PVA $06	Push Var Addr	A6 A	VTM $06->$$A$	Var To Mem
77	PVA $07	Push Var Addr	A7 A	VTM $07->$$A$	Var To Mem
78 S	RAF $$S$	Rtn from AC Fn	A8 S	DIV $$S$	Divide
79	RAF $01	Rtn from AC Fn	A9	DIV $01	Divide
7A	RAF $02	Rtn from AC Fn	AA	DIV $02	Divide
7B	RAF $03	Rtn from AC Fn	AB	DIV $03	Divide
7C	RAF $04	Rtn from AC Fn	AC S	DIV $$S$S	Divide
7D	RAF $05	Rtn from AC Fn	AD	DIV $01S	Divide
7E	RAF $06	Rtn from AC Fn	AE	DIV $02S	Divide
7F	RAF $07	Rtn from AC Fn	AF	DIV $03S	Divide
80 S	SHL $$S$	Shift Left	B0 S	EXT $$S$	Extend
81	SHL $01	Shift Left	B1	EXT $01	Extend
82	SHL $02	Shift Left	B2	EXT $02	Extend
83	SHL $04	Shift Left	B3	EXT $03	Extend
84	SHL $03	Shift Left	B4 S	EXT $$S$S	Extend
85	SHL $05	Shift Left	B5	EXT $01S	Extend
86	SHL $06	Shift Left	B6	EXT $02S	Extend
87	SHL $07	Shift Left	B7	EXT $03S	Extend
88 S	STM $$S$	Stack To Mem	B8 S	MUL $$S$	Multiply
89	STM $01	Stack To Mem	B9	MUL $01	Multiply
8A	STM $02	Stack To Mem	BA	MUL $02	Multiply
8B	STM $03	Stack To Mem	BB	MUL $03	Multiply
8C	STM $04	Stack To Mem	BC S	MUL $$S$S	Multiply
8D	STM $05	Stack To Mem	BD	MUL $01S	Multiply
8E	STM $06	Stack To Mem	BE	MUL $02S	Multiply
8F	STM $07	Stack To Mem	BF	MUL $03S	Multiply

Hex Bytes	Assembly	Name	Hex Bytes	Assembly	Name
C0 S	SHR $$S$$	Shift Right	E0 S	TLT $$S$$	Test Less Than
C1	SHR $01	Shift Right	E1	TLT $01	Test Less Than
C2	SHR $02	Shift Right	E2	TLT $02	Test Less Than
C3	SHR $03	Shift Right	E3	TLT $03	Test Less Than
C4 S	SHR $$S$S	Shift Right	E4 S	TLT $$S$S	Test Less Than
C5	SHR $01S	Shift Right	E5	TLT $01S	Test Less Than
C6	SHR $02S	Shift Right	E6	TLT $02S	Test Less Than
C7	SHR $03S	Shift Right	E7	TLT $03S	Test Less Than
C8 S	TGE $$S$$	Test Greater or Equal	E8	???	Unused
C9	TGE $01	Test Greater or Equal	E9	???	Unused
CA	TGE $02	Test Greater or Equal	EA	???	Unused
CC	TGE $03	Test Greater or Equal	EB	???	Unused
CC S	TGE $$S$$	Test Greater or Equal	EC	???	Unused
CD	TGE $01S	Test Greater or Equal	ED	???	Unused
CE	TGE $02S	Test Greater or Equal	EE	???	Unused
CF	TGE $03S	Test Greater or Equal	EF	???	Unused
D0 S	TGT $$S$$	Test Greater Than	F0	???	Unused
D1	TGT $01	Test Greater Than	F1	???	Unused
D2	TGT $02	Test Greater Than	F2	???	Unused
D3	TGT $03	Test Greater Than	F3	???	Unused
D4 S	TGT $$S$S	Test Greater Than	F4	???	Unused
D5	TGT $01S	Test Greater Than	F5	???	Unused
D6	TGT $02S	Test Greater Than	F6	???	Unused
D7	TGT $03S	Test Greater Than	F7	???	Unused
D8 S	TLE $$S$$	Test Less or Equal	F8	???	Unused
D9	TLE $01	Test Less or Equal	F9	???	Unused
DA	TLE $02	Test Less or Equal	FA	???	Unused
DB	TLE $03	Test Less or Equal	FB	???	Unused
DC S	TLE $$S$S	Test Less or Equal	FC	???	Unused
DD	TLE $01S	Test Less or Equal	FD	???	Unused
DE	TLE $02S	Test Less or Equal	FE	???	Unused
DF	TLE $03S	Test Less or Equal	FF	???	Unused