RCA COSMAC VIP - Chip-8 Interpreter Disassembly

Address	Code	Labels	Assembler	Comments
0000	91		GHI 1	When a program is initially run R1 points to the end of the last available page of on-card RAM.
0001	BB		PHI B	Sets RB to the display page (this is the highest memory page of on-card RAM)
0002	FF 01		SMI 0x01	Point to the previous page in RAM
0004	B2		PHI 2	Set the high order byte of the stack pointer to this page
0005	B6		PHI 6	Set the high order byte of the the VX pointer to this page
0006	F8 CF		LDI 0xCF	Initialise low order byte of stack pointer
0008	A2		PLO 2	
0009	F8 81		LDI 0x81	These next four instructions set the Program Counter for the interrupt routine in R1 to 0x8146.
000B	B1		PHI 1	
000C	F8 46		LDI 0x46	
000E	A1		PLO 1	
000F	90		GHI 0	Set R4 to 0x001B in preparation for assignment as Program Counter for the call routine.

Address	Code	Labels	Assembler	Comments
0010	B4		PHI 4	
0011	F8 1B		LDI 0x1B	
0013	A4		PLO 4	
0014	F8 01		LDI 0x01	Set R5 to 0x01FC. This register will act as the Chip 8 Program Counter
0016	B5		PHI 5	
0017	F8 FC		LDI 0xFC	
0019	A5		PLO 5	
001A	D4		SEP 4	R4 is now the interpreter program counter. This has no effect on the sequence at this point because R4 points to 001B which is the next instruction pointed to by the old PC in R0.
001B	96	FETCH_DECODE_LOOP:	GHI 6	The Chip 8 Fetch and Decode routine starts here Get the high order byte of the VX pointer ...
001C	B7		PHI 7	... and copy this to the high order byte of the VY pointer
001D	E2		SEX 2	Use the stack pointer (R2) for indirect register addressing operations
001E	94		GHI 4	Copy high order byte of CALL routine pointer (R4) ...

Address	Code	Labels	Assembler	Comments
001F	BC		PHI C	... and copy it to RC (RC will be used later as a pointer into a pair of lookup tables that hold the addresses of the routines that handle each instruction group)
0020	45		LDA 5	Get the first byte of the next Chip-8 instruction and advance the instruction pointer (R5)
0021	AF		PLO F	Copy first byte of Chip-8 instruction to RF.0
0022	F6		SHR	The next four instructions move the most significant digit of the Chip-8 instruction (first byte) - the instruction group code - to the position of the least significant digit. The least significant digit is discarded
0023	F6		SHR	
0024	F6		SHR	
0025	F6		SHR	
0026	32 44		BZ FIRST_DIGIT_0	If op code digit is 0 branch to FIRST_DIGIT_0
0028	F9 50		ORI 0x50	Apply a mask to the instruction group code to turn it into the low-order part of an address that points to an entry in a lookup table (This table is stored from 0x0051 to 0x005F)

Address	Code	Labels	Assembler	Comments
002A	AC		PLO C	RC now points to the correct entry in a lookup table for the instruction group of the current instruction - this table holds the high order byte of the address of the routine that handles that instruction group
002B	8F		GLO F	Retrieve the unaltered copy of the first byte of the Chip-8 instruction from RF.0
002C	FA 0F		ANI 0x0F	Mask the first byte of the Chip-8 instruction to leave only the least significant digit
002E	F9 F0		ORI 0xF0	Apply a mask to the least significant digit of the first byte of the Chip-8 instruction to form the low order byte of a pointer to the relevant variable (These variables are stored in the final page of on-card RAM from 0x0XF0 to 0x0XFF)
0030	A6		PLO 6	The VX pointer (R6) now points to the correct variable for this instruction
0031	05		LDN 5	Get the second byte of the Chip-8 instruction (do not advance the instruction pointer)
0032	F6		SHR	The next four instructions move the most significant digit of the Chip-8 instruction (second byte) - VY - to the position of the least significant digit. The least significant digit is discarded)
0033	F6		SHR	

Address	Code	Labels	Assembler	Comments
0034	F6		SHR	
0035	F6		SHR	
0036	F9 F0		ORI 0xF0	Apply a mask to the VY part of the Chip-8 instruction to form the low order byte of a pointer to the relevant variable (These variables are stored in the final page of on-card RAM from 0x0XF0 to 0x0XFF)
0038	A7		PLO 7	The VY pointer (R7) now points to the correct variable for this instruction
0039	4C		LDA C	Get high-order byte of routine from look-up table
003A	B3		PHI 3	Store this in the high order byte of the interpreter programme counter (R3)
003B	8C		GLO C	Get the low order byte of the address currently pointed to by RC - this will have been moved on by 1 by the LDA instruction...
003C	FC 0F		ADI 0x0F	... so, as the corresponding entries in each table are placed 16 bytes apart, it's just necessary to add 0x0F to the address ...
003E	AC		PLO C	... so that RC now points to the correct place in the second look up table
003F	0C		LDN C	Get the low order byte of the address from the lookup table

Address	Code	Labels	Assembler	Comments
0040	A3	CALL_SUBRO UTINE:	PLO 3	And use this to set the low order byte of the interpreter programme counter (R3)
0041	D3		SEP 3	Now call the interpreter subroutine to handle this instruction group
0042	30 1B		BR FETCH_DECODE_LOOP	On return from the subroutine, loop back and get the next Chip-8 instruction
0044	8F	FIRST_DIGIT _0:	GLO F	This subroutine is entered when the first digit of the instruction is 0x0. This indicates a call to the machine code routine stored in the remaining three digits of the instruction. The routine starts by retrieving the original first byte of the Chip-8 instruction in RF.0
0045	FA 0F		ANI 0x0F	Use a mask to remove the first digit of the instruction (leaving the high order byte of the address to be called)
0047	B3		PHI 3	Use this to set the high order byte of the interpreter programme counter (R3), as this is also used as the programme counter for machine code routines called with this instruction
0048	45		LDA 5	Get the low-order byte of the address to be called directly from memory using the Chip-8 programme counter (R5) and then advance this
0049	30 40		BR CALL_SUBROUTINE	Now return to the main fetch and decode loop and call the relevant subroutine

Address	Code	Labels	Assembler	Comments
004B	22	SWITCH_ON_ DISPLAY:	DEC 2	Subroutine to turn on display R2 is the stack pointer. The 1802 has no push or pop operations, so this has to be done manually. The stack grows downwards in memory, so to push a value onto the stack, the stack pointer has to first be decremented.
004C	69		INP 1	Decrement stack pointer and turn display on (display interrupts are controlled by routine at 8146)
004D	12		INC 2	Increment R2 (Stack pointer)
004E	D4		SEP 4	Return to 0042
004F	00 00		DB 0x00, 0x00	This is filler before the subroutine address lookup tables so that the last digit of the address for each entry corresponds to the digit that indicates the instruction group (i.e. the entry for instruction group 1 is found at 0x0051, the entry for instruction group 2 at 0x0052, etc.)
0051	01 01 01 01 01 01 01 01 01 01 01 01 00 01 01		DB 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x00, 0x01, 0x01	A lookup table holding the high order bytes of the addresses of the subroutines for Chip-8 instruction groups 1 through F
0060	00		DB 0x00	This is filler between the tables so that the second table is also aligned to instruction group numbers (i.e. 1 is at 0x0061, 2 at 0x0062, etc.)

Address	Code	Labels	Assembler	Comments
0061	7C 75 83 88 95 B4 87 BC 91 EB A4 D9 70 99 05		DB 0x7C, 0x75, 0x83, 0x88, 0x95, 0xB4, 0x87, 0xBC, 0x91, 0xEB, 0xA4, 0xD9, 0x70, 0x99, 0x05	Table holding the low bytes for the subroutines selected by the first digit of Chip-8 instructions. So the completed addresses for each digit are: 0x1: 0x017C 0x2: 0x0175 0x3: 0x0183 0x4: 0x018B 0x5: 0x0195 0x6: 0x01B4 0x7: 0x01B7 0x8: 0x01BC 0x9: 0x0191 0xA: 0x01EB 0xB: 0x01A4 0xC: 0x01D9 0xD: 0x0070 0xE: 0x0199 0xF: 0x0105
0070	06	DISPLAY:	LDN 6	Display subroutine (1st digit = 0xD) Get VX(stored at address in R6)
0071	FA 07		ANI 0x07	Mask with 0x07 to save only least significant three bits. These indicate the bit offset of the first bit of sprite data
0073	BE		PHI E	Save these in RE.1
0074	06		LDN 6	Get VX

Address	Code	Labels	Assembler	Comments
0075	FA 3F		ANI 0x3F	Mask with 0x3F to save least significant six bits (max value of X position is 63, which requires only six bits)
0077	F6		SHR	The next three instructions perform an integer division of VX by eight, which gives the position in the pixel row of the first byte that will contain sprite data
0078	F6		SHR	
0079	F6		SHR	
007A	22		DEC 2	Decrement the stack pointer (R2) ready for a push
007B	52		STR 2	Push accumulator (containing most significant three bits of VX) onto the stack
007C	07		LDN 7	Get VY (stored at address in R7)
007D	FA 1F		ANI 0x1F	Mask with 0x1F to save the five least significant bits (max value of Y position is 31, which requires only five bits)
007F	FE		SHL	The next three instructions perform a multiplication of VY by eight, which gives the position in display memory of the first row that will contain sprite data
0080	FE		SHL	
0081	FE		SHL	

Address	Code	Labels	Assembler	Comments
0082	F1		OR	OR the result with the top of the stack. This gives the position in display memory of the first byte that will contain pixel data from the sprite
0083	AC		PLO C	Put the result in RC0
0084	9B		GHI B	Get high order byte of address of display memory
0085	BC		PHI C	Put this in RC1. RC now holds the address of the first byte that will have sprite data written to it
0086	45		LDA 5	Get the second byte of the Chip-8 instruction and advance the Chip-8 programme counter
0087	FA 0F		ANI 0x0F	Mask off the least significant hex digit. This contains the number of bytes (rows) in the sprite pattern
0089	AD		PLO D	Save it in RD - this will be used as a display row counter
008A	A7		PLO 7	Save it in R7 - this will be used as a row counter

Address	Code	Labels	Assembler	Comments
008B	F8 D0		LDI 0xD0	0xD0 is the low order byte of the address of the area of RAM set aside as a Chip-8 work area. This will be used to assemble a two-byte wide copy of the sprite with the sprite data shifted to the correct offset for the position at which the sprite will be displayed
008D	A6		PLO 6	Put this into R6.0 As R6 is normally used as the VX pointer and the variables are stored in the same page, R6.1 will already be set correctly
008E	93	NEXT_SPRITE_ROW:	GHI 3	R3.1 is used as a convenient source of the constant 0x0
008F	AF		PLO F	Set RF.0 to 0x0. The left byte of the reconstructed sprite will be initially assembled here
0090	87		GLO 7	Get the number of rows left to assemble
0091	32 F3		BZ RESET_I_PTR	Branch to the next stage if they are all done
0093	27		DEC 7	Count off one row of sprite data
0094	4A		LDA A	Get one byte of sprite data from the address pointed at by I (RA) and advance I to next byte
0095	BD		PHI D	Put byte in high byte of RD
0096	9E		GHI E	Get the bit offset for the first bit of sprite data (this was saved in RE.1 earlier)

Address	Code	Labels	Assembler	Comments
0097	AE		PLO E	Put these in RE.0. This will be used as a bit counter
0098	8E	SPLIT_SPRITE_ROW:	GLO E	Get the current bit count
0099	32 A4		BZ STORE_SPRITE_ROW	Branch when the bit count is zero, indicating that the sprite data for that row is now correctly split across two bytes (note that this could be immediately if the sprite is positioned at the start of a byte)
009B	9D		GHI D	Get byte to be displayed
009C	F6		SHR	Shift right by 1 bit. his will move a zero into the most significant bit, shift everything else along and move the least significant bit into the carry flag
009D	BD		PHI D	Store shifted byte back in RD.1
009E	8F		GLO F	Get current pattern in second byte
009F	76		SHRC	Shift with carry to the right by one bit. This will move the discarded bit from the first byte into the most significant bit position and shift everything else along
00A0	AF		PLO F	Store the result back in RF.0
00A1	2E		DEC E	Count off another bit
00A2	30 98		BR SPLIT_SPRITE_ROW	Branch back to top of loop

Address	Code	Labels	Assembler	Comments
00A4	9D	STORE_SPRITE_ROW:	GHI D	Get lefthand byte of sprite row to be displayed
00A5	56		STR 6	Store it in the working area in memory
00A6	16		INC 6	Point to the next byte in the working area
00A7	8F		GLO F	Get the righthand byte of the sprite row to be displayed
00A8	56		STR 6	Store it in the working area in memory
00A9	16		INC 6	Point to the next byte in the working area
00AA	30 8E		BR NEXT_SPRITE_ROW	Go back and do next row
00AC	00	DISPLAY_SPRITE:	IDL	Wait for display interrupt
00AD	EC		SEX C	Set the pointer to display memory (RC) to be used for register indirect addressing memory operations
00AE	F8 D0		LDI 0xD0	0xD0 is the low order byte of the address of the area of RAM set aside as a Chip-8 work area. This is where the offset sprite has been assembled
00B0	A6		PLO 6	R6 now points to assembled offset sprite
00B1	93		GHI 3	R3.1 (high-order byte of interpreter programme counter) is a convenient source of the constant 0x0

Address	Code	Labels	Assembler	Comments
00B2	A7		PLO 7	Set R7.0 to zero. This will be used to temporarily store the collision status
00B3	8D	SPRITE_DISPLAY_LOOP:	GLO D	Get the number of rows left to display
00B4	32 D9		BZ SAVE_COLLISION_FLAG	Branch to next stage if all rows done
00B6	06		LDN 6	Get the lefthand byte of sprite data
00B7	F2		AND	AND it with the current byte in display memory at the target position. This will put a 1 in any bit where a set bit overlaps in both the display memory and the sprite data. So any non-zero result indicates that a collision has occurred
00B8	2D		DEC D	Count off one row
00B9	32 BE		BZ DISPLAY_LEFT_BYTE:	Branch forward if no collision occurred
00BB	F8 01		LDI 0x01	Construct a collision flag
00BD	A7		PLO 7	Store this in R7.0
00BE	46	DISPLAY_LEFT_BYTE:	LDA 6	Get the lefthand byte of sprite data and advance the pointer
00BF	F3		XOR	XOR it with the current byte in display RAM
00C0	5C		STR C	Now write it to the display by storing the modified byte back in the display RAM

Address	Code	Labels	Assembler	Comments
00C1	02		LDN 2	Get the x position of the sprite (in bytes) from the stack
00C2	FB 07		XRI 0x07	XOR it with 0x07 to see if it is at position 7 (i.e. the last byte in the row)
00C4	32 D2		BZ DISPLAY_NEXT_ROW	If it is at the right edge of the window, then the second byte would be off screen and there is no point in trying to display it, so skip to the next row
00C6	1C		INC C	Point to the next byte in the display memory
00C7	06		LDN 6	Get the righthand byte of sprite data
00C8	F2		AND	AND it with the current byte in display memory at the target position. This will put a 1 in any bit where a set bit overlaps in both the display memory and the sprite data. So any non-zero result indicates that a collision has occurred
00C9	32 CE		BZ DISPLAY_RIGHT_BYTE	Branch forward if no collision occurred
00CB	F8 01		LDI 0x01	Construct a collision flag
00CD	A7		PLO 7	Store this in R7.0
00CE	06	DISPLAY_RIGHT_BYTE:	LDN 6	Get the righthand byte of sprite data
00CF	F3		XOR	XOR it with current byte in display RAM

Address	Code	Labels	Assembler	Comments
00D0	5C		STR C	Now write it to the display by storing the modified byte back in the display RAM
00D1	2C		DEC C	Reset RC so it points to the first byte in the row with sprite data
00D2	16	DISPLAY_NE XT_ROW:	INC 6	Point R6 the next byte of sprite data
00D3	8C		GLO C	Get the low-order byte of the current position in display RAM
00D4	FC 08		ADI 0x08	Add 0x08 to move it down one row
00D6	AC		PLO C	Put the result back in RC.0
00D7	3B B3		BNF SPRITE_DISPLAY_LOOP	Only display the next row if it is not off the bottom of the screen. This will be indicated because adding 0x08 to the display RAM address will cross a page boundary and generate a carry condition
00D9	F8 FF	SAVE_COLLI SION_FLAG:	LDI 0xFF	0xFF is the low order byte of the address of variable F, where the collision flag will be stored
00DB	A6		PLO 6	R6 now points to variable F
00DC	87		GLO 7	Get the collision flag
00DD	56		STR 6	Store it in variable F
00DE	12		INC 2	Increment R2 (Stack pointer)

Address	Code	Labels	Assembler	Comments
00DF	D4	RETURN_TO_FETCH_LOOP:	SEP 4	Set R4 as program counter (this returns execution to the fetch and decode routine at 0042)
00E0	9B	CLS:	GHI B	Start of routine to erase display page. Get the display page ...
00E1	BF		PHI F	... and store it in RF.1
00E2	F8 FF		LDI 0xFF	... so that RF now points to the final byte in the display page
00E4	AF		PLO F	... so that RF now points to the final byte in the display page
00E5	93	CLEAR_SCREEN_LOOP:	GHI 3	Get zero into the accumulator (D) R3 is the interpreter subroutine/machine code subroutine programme counter. Since this routine is in page 0, R3.1 will contain 0 and it takes just a one byte instruction to get the value from this source than to use a two-byte immediate addressing instruction. This is another example where saving a few bytes of memory here or there was more important than code clarity!
00E6	5F		STR F	Zero the memory location currently pointed to by RF
00E7	8F		GLO F	Get the low order byte of the current address in RF

Address	Code	Labels	Assembler	Comments
00E8	32 DF		BZ RETURN_TO_FETCH_LOOP	If the byte that's just been zeroed is is at address 0x00 in the display page then we're done, so jump to the return instruction
00EA	2F		DEC F	Otherwise point to the previous byte in the display (bytes are zeroed starting at the end and moving backwards through the display memory)
00EB	30 E5		BR CLEAR_SCREEN_LOOP	Jump back to the top of the loop
00ED	00		DB 0x00	Filler
00EE	42	RET:	LDA 2	Start of routine to return from subroutine Pop the high-order byte of the return address off the stack and advance the stack pointer
00EF	B5		PHI 5	Load the high-order byte of the return address into the Chip-8 programme counter (R5)
00F0	42		LDA 2	Pop the low-order byte of the return address off the stack
00F1	A5		PLO 5	Load the low-order byte of the return address into the Chip-8 programme counter
00F2	D4		SEP 4	Return to the fetch and decode routine. The next instruction to be fetched will be at the return address

Address	Code	Labels	Assembler	Comments
00F3	8D	RESET_I_PTR	GLO D	This is part of the display routine used to reset the I pointer to its original value (pointing at the start of the sprite) Get the total number of sprite rows
00F4	A7		PLO 7	Make this into a counter in R7.0
00F5	87	RESET_I_LOOP:	GLO 7	Get number of rows remaining
00F6	32 AC		BZ DISPLAY_SPRITE	If zero (all rows done, so I pointer is reset), branch routine to display sprite
00F8	2A		DEC A	Decrement I pointer (RA)
00F9	27		DEC 7	Decrement row counter
00FA	30 F5		BR RESET_I_LOOP	Branch back to top of the loop
00FC	00 00 00 00		DB 0x00, 0x00, 0x00, 0x00	Filler to end of RAM page
0100	00 00 00 00 00		DB 0x00, 0x00, 0x00, 0x00, 0x00	Filler at start of next page
0105	45	DECODE_F_INSTRUCTIONS:	LDA 5	Start of routine to decode 0xFXXX instructions
0106	A3		PLO 3	Use this to set the interpreter programme counter (R3) to the address of the relevant handler. Execution will continue from there
0107	98	FX07:	GHI 8	Instruction FX07 -> get current value of delay timer into VX Get current value of timer (R8.1)

Address	Code	Labels	Assembler	Comments
0108	56		STR 6	Store it in VX
0109	D4		SEP 4	Return to the fetch and decode routine
010A	F8 81	FX0A:	LDI 0x81	Instruction FX0A -> wait for a key press and store it in VX 0x81 is the high-order byte of the address of a routine in the COSMAC VIP ROM that reads the keyboard
010C	BC		PHI C	Store this in RC.1
010D	F8 95		LDI 0x95	0x95 is the low-order byte of the address of the keyboard routine
010F	AC		PLO C	Put this in RC.0 - RC now contains the full address 0x8195
0110	22		DEC 2	Decrement stack pointer - the ROM routine uses the stack so we need to ensure the stack pointer is pointing at the next empty location before calling it
0111	DC		SEP C	Call the routine to read the keyboard On return the value of the key pressed will be in the accumulator D
0112	12		INC 2	Increment stack pointer
0113	56		STR 6	Store the result in VX
0114	D4		SEP 4	Return to the fetch and decode routine
0115	06	FX15:	LDN 6	Instruction FX15 -> Set timer to VX

Address	Code	Labels	Assembler	Comments
0116	B8		PHI 8	Get VX (designated by R6) and put it into high byte of R8 (timer)
0017	D4		SEP 4	Return to fetch and decode routine
0118	06	FX18:	LDN 6	Instruction FX18 -> Set sound timer to VX Get the value in VX (which is pointed to by R6)
0119	A8		PLO 8	Copy it into the sound timer (R8.0)
011A	D4		SEP 4	Return to fetch and decode routine
011B	64 0A 01	BCD_DENOMINATORS:	DB 0x64, 0x0A, 0x01	Three constants with the decimal values of 100, 10 and 1, used by the BCD instruction FX33
011E	E6	FX1E:	SEX 6	Instruction FX1E -> add variable X to I variable Set register indirect addressing operations to use R6 (VX pointer)
011F	8A		GLO A	Get the low order byte of I (stored in RA.0)
0120	F4		ADD	Add value in VX to low order byte of I
0121	AA		PLO A	Put result back into low order byte of I (RA.0)
0122	3B 28		BNF SAME_PAGE	If no carry as generated (i.e. the addition of the offset didn't cross a page boundary) then there's nothing more to do
0124	9A		GHI A	Get high order byte of I (RA.1)

Address	Code	Labels	Assembler	Comments
0125	FC 01		ADI 0x01	Add 1 to it (to point to next page)
0127	BA		PHI A	Put the result back in high order byte of I (RA.1)
0128	D4	SAME_PAGE:	SEP 4	Return to the fetch and decode routine
0129	F8 81	FX29:	LDI 0x81	Instruction FX29 -> Point I to sprite for hexadecimal character in VX Both the look up table and the sprite data are located in the ROM in page 0x81
012B	BA		PHI A	Store this in the high order byte of I (RA.1)
012C	06		LDN 6	Get the value in VX (R6)
012D	FA 0F		ADI 0x0F	Apply a mask to save just the least significant digit
012F	AA		PLO A	I now points to the correct entry in the look-up table
0130	0A		LDN A	Get the low-order byte of the sprite address from the look-up table
0131	AA		PLO A	I now points to the start of the data for the correct sprite
0132	D4		SEP 4	Return to the fetch and decode routine
0133	E6	FX33:	SEX 6	Instruction FX33 -> Store BCD value of VX at memory pointed to by I Use VX (R6) for register indirect addressing

Address	Code	Labels	Assembler	Comments
0134	06		LDN 6	Get the value to be converted from VX
0135	BF		PHI F	Preserve the original value by temporarily storing it in RF.1
0136	93		GHI 3	Get the high order byte of the address of the BCD denominator constants
0137	BE		PHI E	Store this in RE.1
0138	F8 1B		LDI BCD_DENOMINATORS	Get the low order byte of the address of the BCD denominator constants
013A	AE		PLO E	RE now points to first BCD denominator constant
013B	2A		DEC A	The I pointer (RA) is pointing to first byte of memory to store BCD but it needs to be moved to the byte before that before entering the loop
013C	1A	BCD_LOOP:	INC A	Point I (RA) to location of next BCD digit
013D	F8 00		LDI 0x00	Create a zero value
013F	5A		STR A	Use this to initialise the BCD digit
0140	0E	DIVISION_LOOP:	LDN E	Get the current BCD constant
0141	F5		SD	Subtract it from the current value of VX

Address	Code	Labels	Assembler	Comments
0142	3B 4B		BNF NEXT_DIGIT:	If the result is negative then the current digit is at the correct value, so move on to the next one
0144	56		STR 6	Store the remainder back in VX
0145	0A		LDN A	Get the value of the current BCD digit
0146	FC 01		ADI 0x01	Add 1 to it
0148	5A		STR A	And put it back into memory
0149	30 40		BR DIVISION_LOOP:	Continue to divide VX by current denominator
014B	4E	NEXT_DIGIT:	LDA E	Get current BCD constant and point to next one
014C	F6		SHR	test the least significant bit
014D	3B 3C		BNF BCD_LOOP	If it's not set (i.e. the constant we just using was not 0x01) then loop back and do the next digit
014F	9F		GHI F	Get the preserved original value of VX
0150	56		STR 6	Restore this to VX
0151	2A		DEC A	This and the next instruction restores I (RA) so it is pointing to the first stored BCD digit
0152	2A		DEC A	
0153	D4		SEP 4	Return to the fetch and decode routine

Address	Code	Labels	Assembler	Comments
0154	00		DB 0x00	Filler
0155	22	FX55:	DEC 2	Instruction FX55 -> Store V0 to VX at memory pointed to by I Decrement the stack pointer (R2), ready for a push
0156	86		GLO 6	Get the low order byte of the VX pointer ...
0157	52		STR 2	... and push it onto the stack
0158	F8 F0		LDI 0xF0	0xF0 is the low-order byte of the address of the first variable (V0)
015A	A7		PLO 7	Set this as the low-order byte of the VY pointer (R7)
015B	07	STORE_VARS_LOOP:	LDN 7	Get the value of the next variable
015C	5A		STR A	Store it in the address pointed to by I
015D	87		GLO 7	Get the low-order byte of the address in I
015E	F3		XOR	XOR it with the value at the stack (the low-order byte of the VX pointer). This will result in 0 if they match - indicating that all the requested variables have been stored
015F	17		INC 7	Point VY to the next variable
0160	1A		INC A	Point I to the next address in memory

Address	Code	Labels	Assembler	Comments
0161	3A 5B		BNZ STORE_VARS_LOOP	Return to top of loop if there are still more variables to store
0163	12		INC 2	Pop the low-order byte of VX off the stack
0164	D4		SEP 4	Return to the fetch and decode routine
0165	22	FX65:	DEC 2	Instruction FX65 -> Set V0 to VX to values stored from memory pointed to by I Decrement the stack pointer (R2), ready for a push
0166	86		GLO 6	Get the low order byte of the VX pointer ...
0167	52		STR 2	... and push it onto the stack
0168	F8 F0		LDI 0xF0	0xF0 is the low-order byte of the address of the first variable (V0)
016A	A7		PLO 7	Set this as the low-order byte of the VY pointer (R7)
016B	0A	LOAD_VARS_ LOOP:	LDN A	Get the byte at the address currently pointed to by I
016C	57		STR 7	Store it in the variable currently pointed to by VY
016D	87		GLO 7	Get the low-order byte of the address in I
016E	F3		XOR	XOR it with the value at the stack (the low-order byte of the VX pointer). This will result in 0 if they match - indicating that all the requested variables have been loaded

Address	Code	Labels	Assembler	Comments
016F	17		INC 7	Point VY to the next variable
0170	1A		INC A	Point I to the next address in memory
0171	3A 6B		BNZ LOAD_VARS_LOOP	Return to top of loop if there are still more variables to load
0173	12		INC 2	Pop the low-order byte of VX off the stack
0174	D4		SEP 4	Return to the fetch and decode routine
0175	15	2MMM:	INC 5	Start of handler for instruction group 2MMM: Call subroutine at 0x0MMM Increment the Chip-8 programme counter (R5) to point to the next instruction in sequence
0176	85		GLO 5	Get the low-order byte of the address
0177	22		DEC 2	Decrement the stack pointer (R2)
0178	73		STXD	Push the low-order byte of the address onto the stack and decrement the stack pointer
0179	95		GHI 5	Get the high-order byte of the address
017A	52		STR 2	Push it onto the stack
017B	25		DEC 5	Reset the Chip-8 programme counter to where it was (pointing at the low-order byte of the current instruction)

Address	Code	Labels	Assembler	Comments
017C	45	1MMM:	LDA 5	This is the entry point for the handler for group 1MMM instructions: Branch to instruction at address 0x0MMM Get the low-order byte of the current instruction
017D	A5		PLO 5	Load it into the low-order byte of the Chip-8 programme counter
017E	86		GLO 6	Get the low-order byte of the VX pointer (R6). This contains the most significant digit of the address to be called/branched to
017F	FA 0F		ANI 0x0F	We just need to mask it off to get it
0181	B5		PHI 5	Load this into the high order byte of the Chip-8 programme counter (this now points to the first instruction of the subroutine/ sequence to be branched to)
0182	D4	NO_SKIP:	SEP 4	Return to fetch and decode routine (First instruction fetched on return will be first instruction of subroutine/sequence that has been branched to)

Address	Code	Labels	Assembler	Comments
0183	45	3XNN:	LDA 5	This is the entry point for the 3XNN instruction group (Skip if VX = NN) On entry the Chip-8 programme counter (R5) will be pointing to the second byte of the instruction, which contains the value to be compared. This is loaded into the accumulator before the programme counter is advanced to point to the next Chip-8 instruction
0184	E6	TEST_FOR_EQUALITY:	SEX 6	The VX pointer (R6) will now be used for register indirect addressing operations
0185	F3		XOR	XOR NN with the contents of VX. A number XOR'd with itself will be zero, so if the result of this operation is zero then the the contents of VX is equal to NN
0186	3A 82		BNZ NO_SKIP	If VX does not equal NN (indicated by a non-zero result), then return to the fetch and decode routine
0188	15	SKIP_INSTR:	INC 5	Skip first byte of next instruction
0189	15		INC 5	Skip second byte of next instruction
018A	D4		SEP 4	Return to the fetch and decode routine

Address	Code	Labels	Assembler	Comments
018B	45	4XNN:	LDA 5	This is the entry point for the 4XNN instruction group (Skip if VX ≠ NN) On entry the Chip-8 programme counter (R5) will be pointing to the second byte of the instruction, which contains the value NN to be compared. This is loaded into the accumulator before the programme counter is advanced to point to the next Chip-8 instruction
018C	E6	TEST_FOR_INEQUALITY:	SEX 6	The VX pointer (R6) will now be used for register indirect addressing operations
018D	F3		XOR	XOR NN with the contents of VX. A number XOR'd with itself will be zero, so if the result of this operation is not zero then the the contents of VX is not equal to NN
018E	3A 88		BNZ SKIP_INSTR	If VX does not equal NN (indicated by a non-zero result), then skip the next instruction
0190	D4		SEP 4	Return to the fetch and decode routine
0191	45	9XY0:	LDA 5	This is the entry point for the 9XY0 instruction group (Skip if VX ≠ VY) The LDA 5 instruction used here simply moves the programme counter on to the next instruction, since the value stored in D is not used. It's not clear why the programmer chose to use an LDA rather than an INC instruction here

Address	Code	Labels	Assembler	Comments
0192	07		LDN 7	Get the value in the VY variable into the accumulator
0193	30 8C		BR TEST_FOR_INEQUALITY	Branch to the test for inequality. The operands will be VX (pointed to by R6) and VY, which is now held in D
0195	45	5XY0:	LDA 5	This is the entry point for the 5XY0 instruction group (Skip if VX = VY) Moves the programme counter on to the next instruction. See comment for the instruction at 0x0191
0196	07		LDN 7	Get the value in the VY variable into the accumulator
0197	30 84		BR TEST_FOR_EQUALITY	Branch to the test for inequality. The operands will be VX (pointed to by R6) and VY, which is now held in D
0199	E6	EX9E/EXA1:	SEX 6	This is the entry point for instruction groups EX9E (Skip if VX = current key press) and EXA1 (Skip if VX ≠ current key press) The VX pointer (R6) will now be used for register indirect addressing
019A	62		OUT 2	This will take the value in VX and output it to the keyboard latch. This causes external flag 3 to be set if that key is currently held down or reset if not

Address	Code	Labels	Assembler	Comments
019B	26		DEC 6	OUT instructions cause the register currently selected in X to be automatically advanced. This instruction resets the VX pointer to point to the correct variable. This is a necessary precaution because if the selected variable is VF, the increment will cause the VX pointer to be pointing to the wrong page entirely when the next instruction is fetched
019C	45		LDA 5	Get the second byte of the Chip-8 instruction and advance the Chip-8 programme counter (R5)
019D	A3		PLO 3	The second byte of the instruction is actually the low-order byte of the address of the next part of the handler to be run, depending on whether we are testing for a key being pressed (0x9E) or not being pressed (0xA1). This value is loaded into the interpreter programme counter (R3) so that execution continues from the correct point.
019E	36 88		B3 SKIP_INSTR	External flag 3 will be set if the key indicated in VX is pressed, so jump to the code that skips the next instruction
01A0	D4		SEP 4	Return to the fetch and decode routine

Address	Code	Labels	Assembler	Comments
01A1	3E 88		BN3 SKIP_INSTR	External flag 3 will be clear if the key indicated in VX is not pressed, so jump to the code that skips the next instruction
01A3	D4		SEP 4	Return to the fetch and decode routine
01A4	F8 F0	BMMM:	LDI 0xF0	Start of handler for group BMMM instructions: branch to address 0x0MMM + V0 Create the low-order byte of the address of V0
01A6	A7		PLO 7	Load this into the VY pointer (R7). This is already loaded with the high-order byte of the address, so it now points to V0
01A7	E7		SEX 7	Set the VY pointer to be used for register indirect addressing
01A8	45		LDA 5	Get the low-order byte of the current Chip-8 instruction
01A9	F4		ADD	Add the value in V0 to the low-order byte of the current Chip-8 instruction to form the low-order byte of the address to be branched to
01AA	A5		PLO 5	Load this into the Chip-8 programme counter
01AB	86		GLO 6	The VX pointer has the high-order byte of the address to be branched to

Address	Code	Labels	Assembler	Comments
01AC	FA 0F		ANI 0x0F	We just need to use a mask to set the most significant digit to 0x0
01AE	3B B2		BNF STORE_HIGH_ORDER_BYTE	If the ADD instruction at 0x01A9 did not cause a carry, then we know the addition of the offset has not crossed a page boundary, so we can skip the next instruction
01B0	FC 01		ADI 0x01	If a carry was generated, we need to add 1 so the high-order byte of the address points to the correct page
01B2	B5	STORE_HIGH_ORDER_BYTE:	PHI 5	Load the high order byte of the address into the Chip-8 programme counter
01B3	D4		SEP 4	Return to fetch and decode routine (Next instruction fetched will be at the address branched to)
01B4	45	6XNN:	LDA 5	Instruction 6XNN -> Store NN in VX Get the value in the second byte of the instruction into the accumulator (D) and then advance the Chip-8 programme counter to the next instruction
01B5	56		STR 6	Store the value in VX
01B6	D4		SEP 4	Return to fetch and decode routine

Address	Code	Labels	Assembler	Comments
01B7	45	7XNN	LDA 5	Instruction 7XNN -> Add NN to VX Get the value in the second byte of the instruction into the accumulator (D) and then advance the Chip-8 programme counter to the next instruction
01B8	E6		SEX 6	Set the VX pointer to be used for register indirect addressing instructions
01B9	F4		ADD	Add the value in VX to the accumulator (D), which currently holds the immediate operand from the second byte of the Chip-8 instruction
01BA	56		STR 6	Store the result back in VX
01BB	D4		SEP 4	Return to the fetch and decode routine
01BC	45	8XYN:	LDA 5	Instruction 8XYN -> ALU operations on VX and VY Get the value in the second byte of the instruction into the accumulator (D) and then advance the Chip-8 programme counter to the next instruction
01BD	FA 0F		ANI 0x0F	Mask the byte to save just the second hex digit
01BF	3A C4		BNZ DECODE_AL_INSTR	If the second digit is not zero, it's an arithmetic and logic instruction, so branch to the decode routine for these

Address	Code	Labels	Assembler	Comments
01C1	07		LDN 7	If we fall through to this point, it's an 8XY0 instruction to copy VY into VX Get the value of VY into the accumulator (D)
01C2	56		STR 6	Copy this into VX
01C3	D4		SEP 4	Return to fetch and decode routine
01C4	AF	DECODE_AL_ INSTR:	PLO F	Temporarily save the last digit of the instruction in RF.0
01C5	22		DEC 2	Decrement the stack pointer, ready for a push operation
01C6	F8 D3		LDI 0xD3	Load a 0xD3 1802 instruction (SEP 3) into the accumulator
01C8	73		STXD	Push this onto the stack and decrement the stack pointer
01C9	8F		GLO F	Restore the last digit of the Chip-8 instruction to the accumulator
01CA	F9 F0		ORI 0xF0	OR this with 0xF0 to create a 1802 instruction of the form 0xFN, where N is the last hex digit of the Chip-8 instruction
01CC	52		STR 2	Push this onto the stack.
01CD	E6		SEX 6	Set the VX pointer to be used for register indirect addressing memory instructions
01CE	07		LDN 7	Load the value in VY into the accumulator

Address	Code	Labels	Assembler	Comments
01CF	D2		SEP 2	Execute the two instructions at the top of the stack (the first of these will be the AL instruction, the second will be a SEP 3 instruction to return control to this routine at the instruction following this one
01D0	56		STR 6	Save the result of the operation in VX
01D1	F8 FF		LDI 0xFF	0xFF is the low-order byte of the address of Chip-8 variable VF
01D3	A6		PLO 6	The VX pointer now points to VF
01D4	F8 00		LDI 0x00	Clear the accumulator
01D6	7E		SHLC	Move the carry flag into the least significant bit of the accumulator
01D7	56		STR 6	Save this in VF
01D8	D4		SEP 4	Return to the fetch and decode routine
01D9	19	CXNN:	INC 9	Instruction CXNN -> set VX to random number masked by NN Increment random number seed (R9). This value is incremented 60 times a second by the interrupt routine, but this may not have run since the last random number was generated, so it is also incremented here
01DA	89		GLO 9	Get the low-order byte of the random number seed

Address	Code	Labels	Assembler	Comments
01DB	AE		PLO E	Save this in RE.0
01DC	93		GHI 3	Get the high order byte of the interpreter programme counter (This will be 0x01)
01DD	BE		PHI E	Put this in RE.1. RE now points to a random byte of interpreter code in page 0x01
01DE	99		GHI 9	Get the high order byte of the random number seed
01DF	EE		SEX E	Use RE for register indirect addressing
01E0	F4		ADD	Add value of random byte from interpreter code to the current high-order byte of the random number seed
01E1	56		STR 6	Store this in VX
01E2	76		SHRC	Shift the result one bit to the right. This will effectively divide the full result of the addition by 2 as it takes into account the carry bit generated by the addition
01E3	E6		SEX 6	Use VX pointer for register indirect addressing
01E4	F4		ADD	Add current value in VX to shifted value in accumulator
01E5	B9		PHI 9	Save this as the new high-order byte of the random number seed
01E6	56		STR 6	Put this value in VX

Address	Code	Labels	Assembler	Comments
01E7	45		LDA 5	Get second byte of Chip-8 instruction and advance programme counter
01E8	F2		AND	Use this to mask the random number in VX
01E9	56		STR 6	Put final value in VX
01EA	D4		SEP 4	Return to the fetch and decode routine
01EB	45	AMMM:	LDA 5	Instruction AMMM -> Set I to MMM Get the value in the second byte of the instruction into the accumulator (D) and then advance the Chip-8 programme counter to the next instruction
01EC	AA		PLO A	Set this as the low-order byte of the address in I (RA)
01ED	86		GLO 6	Get low order byte of VX pointer, as this retains the second hex digit of the first instruction byte
01EF	FA 0F		ANI 0x0F	Set the first hex digit to 0x0
01F0	BA		PHI A	Set this as the high-order byte of the address in I
01F1	D4		SEP 4	Return to fetch and decode routine
01F2	00 00 00 00 00 00 00 00 00 00		DB 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00	Filler

Address	Code	Labels	Assembler	Comments
01FC	00 E0		DB 0x00, 0xE0	A chip 8 instruction to call the machine code routine at 00E0. This clears the display. This is executed at the start of every Chip-8 program.
01FE	00 4B		DB 0x00, 0x4B	A chip 8 instruction to call the machine code routine at 004B. This switches on the display. This is executed at the start of every Chip-8 program.