ECE331 Handout 3- ASM Instructions, Address Modes and Directives

ASM Instructions

Functional Instruction Groups

- Data Transfer/Manipulation
- Arithmetic
- Logic & Bit Operations
- Data Test
- Branch
- Function Call (Subroutine)

Data Transfer/Manipulation

Mnemonic	*Function*	*Operation*	*C*	*V*	*Z*	*N*
LDAA	Load accumulator A	A ← M	--	0	Δ	Δ
LDAB	Load accumulator B	B ← M	--	0	Δ	Δ
LDD	Load accumulator D	A ← M, B ← M+1	--	0	Δ	Δ
LDX	Load index register X	X ← M:M+1	--	0	Δ	Δ
LDY	Load index register Y	Y ← M:M+1	--	0	Δ	Δ
LDS	Load stack pointer (SP)	SP ← M:M+1	--	0	Δ	Δ

Table A. Load instructions

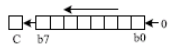
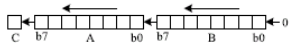
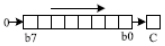
Mnemonic	*Function*	*Operation*	*C*	*V*	*Z*	*N*
STAA	Store accumulator A	A → M	--	0	Δ	Δ
STAB	Store accumulator B	B → M	--	0	Δ	Δ
STD	Store accumulator D	A → M, B → M+1	--	0	Δ	Δ
STX	Store index register X	X → M:M+1	--	0	Δ	Δ
STY	Store index register Y	Y → M:M+1	--	0	Δ	Δ
STS	Store stack pointer (SP)	SP → M:M+1	--	0	Δ	Δ

Table B. Store instructions

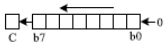
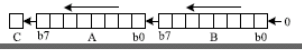
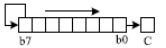
Mnemonic	*Function*	*Operation*	*C*	*V*	*Z*	*N*
TAB	Transfer acc. A to acc. B	B ← A	--	0	Δ	Δ
TBA	Transfer acc. B to acc. A	A ← B	--	0	Δ	Δ
TAP	Transfer acc. A to CCR	CCR ← A	Δ	Δ	Δ	Δ
TPA	Transfer CCR to acc.A	A ← CCR	--	--	--	--
MOVB	Mem to Mem move byte	$(M)_1 \rightarrow (M)_2$	--	--	--	--
MOVW	Mem to Mem move Word	$(M:M+1)_1 \rightarrow (M:M+1)_2$	--	--	--	--

Table C. Move/transfer instructions

Logical shift instructions

Mnemonic	Function	Operation
LSL <opr>	Logical shift left memory	
LSLA	Logical shift left A	
LSLB	Logical shift left B	
LSLD	Logical shift left D	
LSR <opr>	Logical shift right memory	
LSRA	Logical shift right A	
LSRB	Logical shift right B	
LSRD	Logical shift right D	

Arithmetic shift instructions

Mnemonic	Function	Operation
ASL <opr>	Arithmetic shift left memory	
ASLA	Arithmetic shift left A	
ASLB	Arithmetic shift left B	
ASLD	Arithmetic shift left D	
ASR <opr>	Arithmetic shift right memory	
ASRA	Arithmetic shift right A	
ASRB	Arithmetic shift right B	

Rotate instructions

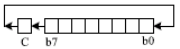
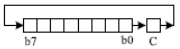
Mnemonic	Function	Operation
ROL <opr>	Rotate left memory thru carry	
ROLA	Rotate left A through carry	
ROLB	Rotate left B through carry	
ROR <opr>	Rotate right memory thru carry	
RORA	Rotate right A through carry	
RORB	Rotate right B through carry	

Table 2.7 ■ Summary of shift and rotate instructions

Shift/Rotate CCR Actions:

- C=Δ, V=Δ, Z=Δ, N=Δ (except LSR where N=0)

Arithmetic

Mnemon	*Function*	*Operation*	*H*	*C*	*V*	*Z*	*N*
ABA	Add acc. A and acc. B	A ← A+B	Δ	Δ	Δ	Δ	Δ
ABX	Add acc. B to index reg. X	X ← B+X	Δ	Δ	Δ	Δ	Δ
ABY	Add acc. B to index reg. Y	Y ← B+Y	Δ	Δ	Δ	Δ	Δ
ADDA	Add acc. A with memory	A ← A+M	Δ	Δ	Δ	Δ	Δ
ADDB	Add acc. B with memory	B ← B+M	Δ	Δ	Δ	Δ	Δ
ADDD	Add acc. D with memory	D ← D+M:M+1	--	Δ	Δ	Δ	Δ
ADCA	ADDA with carry	A ← A+M+C	Δ	Δ	Δ	Δ	Δ
ADCB	ADDB with carry	B ← B+M+C	Δ	Δ	Δ	Δ	Δ

Table D. Addition instructions

Mnemonic	*Function*	*Operation*	*C*	*V*	*Z*	*N*
SBA	Subtract acc. B from acc. A	A ← A-B	Δ	Δ	Δ	Δ
SUBA	Subtract memory from acc. A	A ← A-M	Δ	Δ	Δ	Δ
SUBB	Subtract memory from acc. B	B ← B-M	Δ	Δ	Δ	Δ
SUBD	Subtract memory from acc. D	D ← D-M:M+1	Δ	Δ	Δ	Δ
SBCA	SUBA with borrow	A ← A-M-C	Δ	Δ	Δ	Δ
SBCB	SUBB with borrow	B ← B-M-C	Δ	Δ	Δ	Δ

Table E. Subtraction instructions

Mnemonic	*Function*	*Operation*	*C*	*V*	*Z*	*N*
DEC	Decrement memory by 1	M ← M-1	--	Δ	Δ	Δ
DECA	Decrement A by 1	A ← A-1	--	Δ	Δ	Δ
DECB	Decrement B by 1	B ← B-1	--	Δ	Δ	Δ
DES	Decrement SP by 1	SP ← SP-1	--	--	Δ	--
DEX	Decrement X by 1	X ← X-1	--	--	Δ	--
DEY	Decrement Y by 1	Y ← Y-1	--	--	--	--

Table F. Decrement instructions

Mnemonic	*Function*	*Operation*	*C*	*V*	*Z*	*N*
INC	Increment memory by 1	M ← M+1	--	Δ	Δ	Δ
INCA	Increment A by 1	A ← A+1	--	Δ	Δ	Δ
INCB	Increment B by 1	B ← B+1	--	Δ	Δ	Δ
INS	Increment SP by 1	SP ← SP+1	--	--	Δ	--
INX	Increment X by 1	X ← X+1	--	--	Δ	--
INY	Increment Y by 1	Y ← Y+1	--	--	--	--

Table G. Increment instructions

Logic & Bit Operations

Mnemonic	Function	Operation
ANDA <opr>	AND A with memory	A ← (A) • (M)
ANDB <opr>	AND B with memory	B ← (B) • (M)
ANDCC <opr>	AND CCR with memory (clear CCR bits)	CCR ← (CCR) • (M)
EORA <opr>	Exclusive OR A with memroy	A ← (A) ⊕ (M)
EORB <opr>	Exclusive OR B with memory	B ← (B) ⊕ (M)
ORAA <opr>	OR A with memory	A ← (A) + (M)
ORAB <opr>	OR B with memory	B ← (B) + (M)
ORCC <opr>	OR CCR with memory	CCR ← (CCR) + (M)
CLC	Clear C bit in CCR	C ← 0
CLI	Clear I bit in CCR	I ← 0
CLV	Clear V bit in CCR	V ← 0
COM <opr>	One's complement memory	M ← $FF - (M)
COMA	One's complement A	A ← $FF - (A)
COMB	One's complement B	B ← $FF - (B)
NEG <opr>	Two's complement memory	M ← $00 - (M)
NEGA	Two's complement A	A ← $00 - (A)
NEGB	Two's complement B	B ← $00 - (B)

Table 2.8 ■ Summary of Boolean logic instructions

Mnemonic	Function	Operation
BCLR <opr>[2], msk8	Clear bits in memory	M ← (M) • ($\overline{mm}$)
BITA <opr>[1]	Bit test A	(A) • (M)
BITB <opr>[1]	Bit test B	(B) • (M)
BSET <opr>[2], msk8	Set bits in memory	M ← (M) + (mm)

Note. 1. <opr> can be specified using all except relative addressing modes for BITA and BITB.
2. <opr> can be specified using direct, extended, and indexed (exclude indiriect) addressing modes.
3. msk8 is an 8-bit value.

Table 2.9 Summary of bit operations

Logic CCR Actions:

- AND/OR/EOR: V=0, Z=Δ, N=Δ (except AND/OR CC)
- COM/COMA/COMB: C= 1, V=0, Z=Δ, N=Δ
- NEG/NEGA/NEGB: C= Δ, V=Δ, Z=Δ, N=Δ
- BITA/BITB/BCLR/BSET: V=0, Z=Δ, N=Δ

Data Test

Compare instructions		
Mnemonic	**Function**	**Operation**
CBA	Compare A to B	(A) - (B)
CMPA	Compare A to memory	(A) - (M)
CMPB	Compare B to memory	(B) - (M)
CPD	Compare D to memory	(D) - (M:M+1)
CPS	Compare SP to memory	(SP) - (M:M+1)
CPX	Compare X to memory	(X) - (M:M+1)
CPY	Compare Y to memory	(Y) - (M:M+1)
Test instructions		
Mnemonic	**Function**	**Operation**
TST	Test memory for zero or minus	(M) - $00
TSTA	Test A for zero or minus	(A) - $00
TSTB	Test B for zero or minus	(B) - $00

Table 2.4 ■ Summary of compare and test instructions

Data Test CCR Actions:

- Compare: C= Δ, V=Δ, Z=Δ, N=Δ
- TST/TSTA/TSTB: C= 0, V=0, Z=Δ, N=Δ

Branch

Unconditional Branches

Mnemonic	*Function*	*Comment*
BRN	branch never	useful as delay
BRA	branch always	8-bit signed offset

Simple Conditional Branches

Mnemonic	*"if" Function*	*Condition = Flag*
BCC	carry clear	C=0
BCS	carry set	C=1
BEQ	equal	Z=1
BNE	not equal	Z=0
BMI	minus	N=I
BPI	plus	N=0
BVS	overflow set	V=1
BVC	overflow clear	V=0

Unsigned Conditional Branches

Mnemonic	*"if" Function*	*Condition*	*Flags*
BHS	higher or same	r ≥ 0	C=0
BHI	higher than	r > 0	C+Z=0
BLS	lower or same	r ≤ 0	C+Z=1
BLO	lower than	r < 0	C=1

Signed Conditional Branches

Mnemonic	*"if" Function*	*Condition*	*Flags*
BGE	greater or equal	r ≥ 0	N⊕V=0
BGT	greater than	r > 0	Z+N⊕V=0
BLE	less or equal	r ≤ 0	Z+N⊕V=1
BLT	less than	r < 0	N⊕V=1

r = result

All branch instructions have a "long" version with 16-bit offset. For example, LBRA or LBCC.

CCR Actions: Branches do not effect CCR

Function Call (Subroutine)

Mnemonic	***Function***	***Operation***	***SP***
JSR	Jump to subtroutine	PC ← M:M+1	← SP-2
RTS	Return from subroutine	PC ← M(SP:SP+1)	← SP+2

CCR Actions: Subroutine calls do not effect CCR

68HC12 Addressing Modes

Addressing Mode	Source Format	Abbreviation	Description
Inherent	INST	INH	Operands (if any) are in CPU registers.
Immediate	INST *#opr8i* or INST *#opr16i*	IMM	Operand is included in instruction stream. 8- or 16-bit size implied by context
Direct	INST *opr8a*	DIR	Operand is the lower 8 bits of an address in the range $0000–$00FF.
Extended	INST *opr16a*	EXT	Operand is a 16-bit address
Relative	INST *rel8* or INST *rel16*	REL	An 8-bit or 16-bit relative offset from the current pc is supplied in the instruction.
Indexed (5-bit offset)	INST *oprx5,xysp*	IDX	5-bit signed constant offset from x, y, sp, or pc
Indexed (auto pre-decrement)	INST *oprx3,–xys*	IDX	Auto pre-decrement x, y, or sp by 1 ~ 8
Indexed (auto pre-increment)	INST *oprx3,+xys*	IDX	Auto pre-increment x, y, or sp by 1 ~ 8
Indexed (auto post-decrement)	INST *oprx3,xys–*	IDX	Auto post-decrement x, y, or sp by 1 ~ 8
Indexed (auto post-increment)	INST *oprx3,xys+*	IDX	Auto post-increment x, y, or sp by 1 ~ 8
Indexed (accumulator offset)	INST *abd,xysp*	IDX	Indexed with 8-bit (A or B) or 16-bit (D) accumulator offset from x, y, sp, or pc
Indexed (9-bit offset)	INST *oprx9,xysp*	IDX1	9-bit signed constant offset from x, y, sp, or pc (lower 8-bits of offset in one extension byte)
Indexed (16-bit offset)	INST *oprx16,xysp*	IDX2	16-bit constant offset from x, y, sp, or pc (16-bit offset in two extension bytes)
Indexed-indirect (16-bit offset)	INST [*oprx16,xysp*]	[IDX2]	Pointer to operand is found at... 16-bit constant offset from x, y, sp, or pc (16-bit offset in two extension bytes)
Indexed-indirect (D accumulator offset)	INST [D,*xysp*]	[D,IDX]	Pointer to operand is found at... x, y, sp, or pc plus the value in D

Indexed Addressing Mode Operations

Postbyte Code (xb)	Source Code Syntax	Comments rr: 00 = X, 01 = Y, 10 = SP, 11 = PC

`rr0nnnnn`	`,r`
	`n,r`
	`-n,r`
		5-bit constant offset n = –16 to +15
		r can specify x, y, sp, or pc

`111rr0zs`	`n,r` `-n,r`	**Constant offset** (9- or 16-bit signed) z:0 = 9-bit with sign in LSB of postbyte(s) 1 = 16-bit if z = s = 1, 16-bit offset indexed-indirect (see below) rr can specify x, y, sp, or pc
`111rr011`	`[n,r]`	**16-bit offset indexed-indirect** rr can specify x, y, sp, or pc
`rr1pnnnn`	`n,-r` `n,+r` `n,r-` `n,r+`	**Auto pre-decrement/increment** or **Auto post-decrement/increment**; p = pre-(0) or post-(1), n = –8 to –1, +1 to +8 rr can specify x, y, or sp (pc not a valid choice)

`111rr1aa`	`A,r`
	`B,r`
	`D,r`
		Accumulator offset (unsigned 8-bit or 16-bit)
		aa:00 = A
		01 = B
		10 = D (16-bit)
		11 = see accumulator D offset indexed-indirect
		rr can specify x, y, sp, or pc

`111rr111`	`[D,r]`	**Accumulator D offset indexed-indirect** rr can specify x, y, sp, or pc

Indexed Addressing Mode Operations & Examples

Indexed Addressing Mode	Offset	Reference	Source Code Example	Machine Code	Cycles
5-bit signed constant offset	-16...+15	X, Y, SP, PC	LDAA 15,X	A6 0F	3
9-bit signed constant offset	-256...+255	X, Y, SP, PC	LDAA 255,Y	A6 E8 FF	3
16-bit signed constant offset	-32768...+65535	X, Y, SP, PC	LDAA 4096,PC	A6 FA 10 00	4
8-bit accumulator (A or B) offset	accA, accB	X, Y, SP, PC	LDAA B,SP	A6 F5	3
16-bit accumulator offset	accD	X, Y, SP, PC	LDAA D,X	A6 E6	3
Auto Pre-Increment by +1...+8	1…8	X, Y, SP	LDAA 8,+SP	A6 A7	3
Auto Pre-Decrement by -1…-8	-1…-8	X, Y, SP	LDAA 8,-Y	A6 68	3
Auto Post-Increment by +1...+8	1…8	X, Y, SP	LDAA 1,X+	A6 30	3
Auto Post-Decrement by -1…-8	-1…-8	X, Y, SP	LDAA 4,X-	A6 3C	3
16-bit offset indexed-indirect	-32768...+65535	X, Y, SP, PC	LDAA [4096,X]	A6 E3 10 00	6
D accumulator indexed-indirect	accD	X, Y, SP, PC	LDX [D,PC]	EE FF	6

adapted from "Introducing the MC68HC12" by James M. Sibigtroth, http://elmicro.com/files/intro_mc68hc12.pdf

HC12 ASM Directives

EQU equates symbol with numeric valve
-use to define memory location or constant
-assembler replaces label with correct # value

EX: `LIST EQU $5B`
defines variable 'LIST' = $5B

ORG origin: set memory address of instructions/data that follow
-all programs must specific their ORG

EX: `TOP ORG $6000`
sets program origin at $6000

END set end of program
-any ASM instructions following END are ignored
-can have directives after END

SWI software interrupt
-stops program execution
-use while testing/debugging code

RMB reserve block of memory

EX: `TEMP BMP $10`
reserves 16 ($10) bytes starting at addr. assigned to label TEMP

FCB form constant byte
-reserves block of memory & initiates contents of reserved block

EX: `ABC FCB $11, $12, $13`
reserves 3 bytes w/ valves $11, $12 & $13 at addr. assigned to ABC

FDB form double-byte, same as FCB but 2 bytes per operand

FCC form constant character
-stores ASCII code for alphanumeric characters enclosed in " " symbols

EX: `NAME FCC "MIKE"`
stores 4 ASCII bytes for MIKE

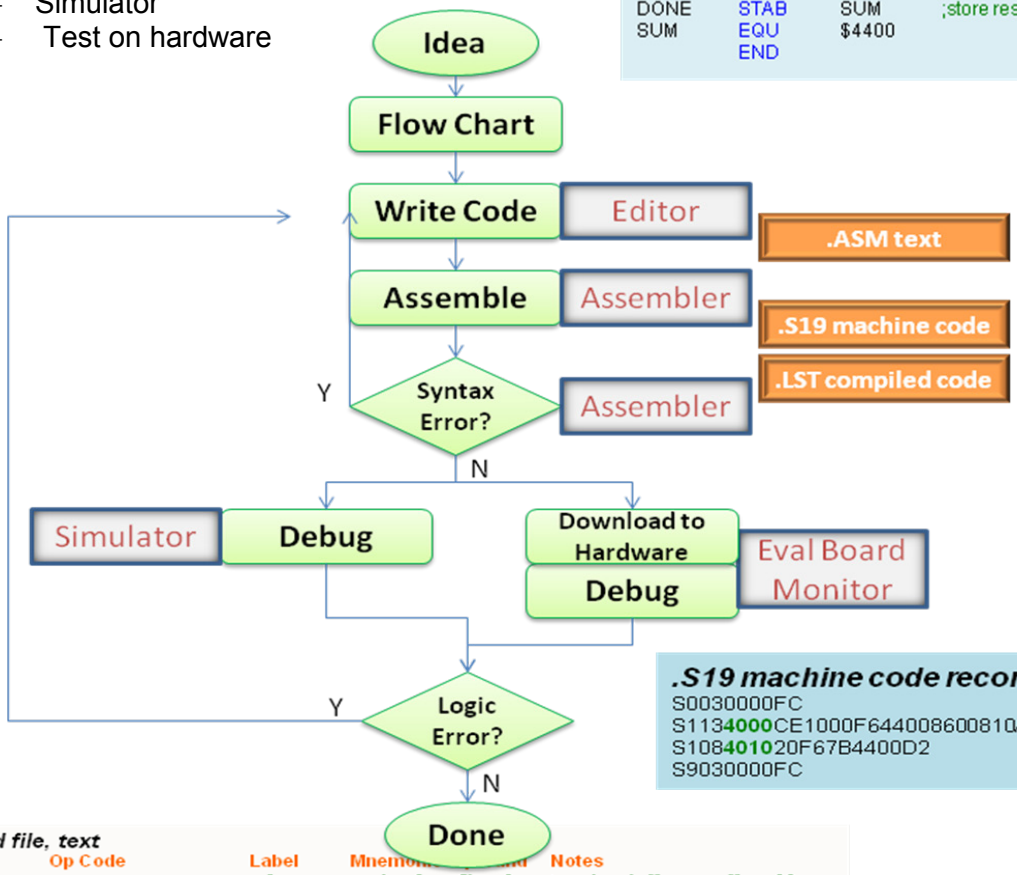
Assembly Process

Assembly Process: The process of converting ASM code into executable machine code.

- Input
  - **.ASM** (text file)
- Outputs
  - **.LST**
    - compiled code
    - program addresses & op-codes
  - **.S19** record
    - HEX file that can be uploaded to μC to store program to memory
- Testing paths
  - Simulator
  - Test on hardware

.ASM code file, text

```
; Loop example for Ch. 3 notes by A.Mason. Mar 09
; Sums 10 values from memory and store result in SUM.
; assumes 10 values to sum are stored at $1000
; assumes prior sum stored at SUM
        ORG     $4000
        LDX     #$1000    ;set x to fist memory address
        LDAB    SUM       ;load staring sum into accB
        LDAA    #$00      ;initialize counter to 0
CHECK   CMPA    #$0A      ; ?added all 10?
        BEQ     DONE      ;  if yes, done
        ADDB    0,X       ;  if no, add # to SUM
        INX               ;increment IX
        INCA              ;increment counter
        BRA     CHECK     ;repeat loop
DONE    STAB    SUM       ;store result
SUM     EQU     $4400
        END
```



.S19 machine code record, binary

```
S0030000FC
S1134000CE1000F644008600810A2706EB00084221
S108401020F67B4400D2
S9030000FC
```

.LST compiled file, text

```
Line    Addr     Op Code          Label   Mnemonic  Operand  Notes
   1:                                                        ; Loop example for Ch. 3 notes by A.Mason. Mar 09
   2:                                                        ; Sums 10 values from memory & store result in SUM
   3:                                                        ; assumes 10 values to sum are stored at $1000
   4:                                                        ; assumes prior sum stored at SUM
   5:          =00004000                       ORG     $4000
   6:     4000 CE 1000                         LDX     #$1000   ;set x to fist memory addr
   7:     4003 F6 4400                         LDAB    SUM      ;load staring sum into accB
   8:     4006 86 00                           LDAA    #$00     ;initialize counter to 0
   9:     4008 81 0A               CHECK       CMPA    #$0A     ; ?added all 10?
  10:     400A 27 06                           BEQ     DONE     ;  if yes, done
  11:     400C EB 00                           ADDB    0,X      ;  if no, add # to SUM
  12:     400E 08                              INX              ;increment IX
  13:     400F 42                              INCA             ;increment counter
  14:     4010 20 F6                           BRA     CHECK    ;repeat loop
  15:     4012 7B 4400             DONE        STAB    SUM      ;store result
  16:          =00004400           SUM         EQU     $4400
  17:                                          END
```

Assembly Process Example

Assembly Code **Assembled Code (.LST file)**

; ECE331 Example of SET/CLR Bit and Branch Instructions

label[1]	*mnemonic*[2] *directive*[1]	*operand*[3]	*prog./data mem. addr.*[4]	*machine opcode*[5]
; main program				
	ORG	**$4000**	**4000**	
	LDAA	**#00**	**4000**	**86 00**
	LDX	**#DATA**	**4002**	**CE 6000**
TOP	**BRSET**	**A,X,$01,ODD**	**4005**	**0E E4 01 0B**
	BSET	**A,X,%00000011**	**4009**	**0C E4 03**
	BCLR	**A,X,%00001100**	**400C**	**0D E4 0C**
	LDAB	**A,X**	**400F**	**E6 E4**
	INCA		**4011**	**42**
	BRA	**TOP**	**4012**	**20 F1**
ODD	**SWI**		**4014**	**3F**
; data storage				
	ORG	**$6000**	**6000**	
DATA	**FCB**	**$EE, $DC, $D0, $F4**	**6000**	**EE DC D0 F4**
	FCB	**$80, $00, $55, $22**	**6004**	**80 00 55 22**
	FCB	**$AA**	**6008**	**AA**
	END			

Notes:

1. Labels and Directives: used by the assembler to simplify writing code and direct the assembly process. During assembly, all labels are replaced with appropriate hexadecimal values. Directives do not result in any program opcodes, although they can generate data to be stored in memory.
2. Mnemonics: shorthand names for assembly instructions. Mnemonics will be converted to hexadecimal opcodes (machine code) during the assembly process that are loaded into microcontroller program memory to form the program that can be executed.
3. Operands: data parameters needed to complete a program instruction or assembly directive statement. Hexadecimal operands must be begin with a '$' so the assembler knows the value is not a decimal. An operand with a '#' specifies an *immediate* address mode; otherwise the operand is an address. Multiple operands can be separated by comma or tabs, depending on the assembler used.
4. Once ASM code is assembled, all instructions and data are assigned to memory addresses. The addresses are determined by ORG directives in the ASM code. It is useful to differentiate between program memory, where instructions (and their operands) are stored and data memory, where non-instruction data fields are stored. Program memory contains the opcodes that will be fetched and decoded during program execution. Data memory contains data values read during program execution or results stored during program execution.
5. Operation codes, or machine opcodes, are hexadecimal values representing instructions. The binary equivalents of these codes tell the microcontroller exactly what do to. Assembly instruction opcode values will be stored in microcontroller program memory at the address indicated beside each line of opcode. For data directives, data values are stored directly to data memory at the address specified before the data values. Data values are not instructions and cannot be executed as program opcodes.

Machine Code Upload Record (.S19 file)

S0030000FC
S11340008600CE60000EE4010B0CE4030DE40CE624
S1084010E44220F13F31
S10C6000EEDCD0F480005522AA64
S9030000FC

Notes:

YELLOW HIGHLIGHTS show memory addresses
GREEN HIGHLIGHTS show data for data memory
BLUE HIGHLIGHTS show ASM program opcodes
to be stored in program memory