

Module 11 – Assembly and machine language programming

Overview

The processor on which this course is based is the Motorola MC68HC11 micro-controller, however, earlier Motorola microprocessors and peripherals may be used to illustrate some concepts. A micro-controller, such as the MC68HC11, is programmed using a dedicated set of instructions. These instructions specify the operation to be performed and the addressing mode to utilise when performing the operation.

Stored program design (concept map)

Objectives

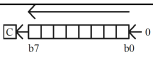

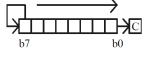
At the completion of this module you will be able to:

- describe the MC68HC11 Instruction Set and the interpretation of its major operations through considerations of several typical examples
- identify the essential components of problem solving techniques and demonstrate their use in assembly and machine program development.

11.1 The MC68HC11 Instruction Set

The following tables list the MC68HC11 instruction set. These tables may appear complex, but once you become familiar with their contents they are fairly straightforward. You are not expected to remember every detail of each instruction, but you must be able to read and understand the tables in order to extract the required information.

Table 11.1: Accumulator and memory instructions

Mnemonic	Operation	Boolean expression	Addressing mode for operand	Machine coding (hexadecimal)		Bytes	Cycles	Condition codes							
				Opcode	Operand(s)			S	X	H	I	N	Z	V	C
ABA	Add Accumulators	$A + B \rightarrow A$	INH	1B		1	2	-	-	↕	-	↕	↕	↕	↕
ABX	Add B to X	$IX + 00:B \rightarrow IX$	INH	3A		1		-	-	-	-	-	-	-	-
ABY	Add B to Y	$IY + 00:B \rightarrow IY$	INH	18 3A		2		-	-	-	-	-	-	-	-
ADCA (opr)	Add with carry to A	$A + M + C \rightarrow A$	A IMM A DIR A EXT A IND,X A IND,Y	89 ii 99 dd B9 hh ll A9 ff 18 A9 ff	2 2 2 3 3 4 2 4 3 5			-	-	↕	-	↕	↕	↕	↕
ADBC (opr)	Add with carry to B	$B + M + C \rightarrow B$	B IMM B DIR B EXT B IND,X B IND,Y	C9 ii D9 dd F9 hh ll E9 ff 18 E9 ff	2 2 2 3 3 4 2 4 3 5			-	-	↕	-	↕	↕	↕	↕
ADDA (opr)	Add Memory to A	$A + M \rightarrow A$	A IMM A DIR A EXT A IND,X A IND,Y	8B ii 9B dd BB hh ll AB ff 18 AB ff	2 2 2 3 3 4 2 4 3 5			-	-	↕	-	↕	↕	↕	↕
ADDB (opr)	Add Memory to B	$B + M \rightarrow B$	B IMM B DIR B EXT B IND,X B IND,Y	CB ii DB dd FB hh ll EB ff 18 EB ff	2 2 2 3 3 4 2 4 3 5			-	-	↕	-	↕	↕	↕	↕
ADDD	Add 16-Bit to D	$D + M:M + 1 \rightarrow D$	IMM DIR EXT IND,X IND,Y	C3 jj kk D3 dd F3 hh ll E3 ff 18 E3 ff	3 4 2 5 3 6 2 6 3 7			-	-	-	-	↕	↕	↕	↕
ANDA (opr)	AND A with Memory	$A \bullet M \rightarrow A$	A IMM A DIR A EXT A IND,X A IND,Y	84 ii 94 dd B4 hh ll A4 ff 18 A4 ff	2 2 2 3 3 4 2 4 3 5			-	-	-	-	↕	↕	0	-
ANDB (opr)	AND B with Memo	$B \bullet M \rightarrow B$	B IMM B DIR B EXT B IND,X B IND,Y	C4 ii D4 dd F4 hh ll E4 ff 18 E4 ff	2 2 2 3 3 4 2 4 3 5			-	-	-	-	↕	↕	0	-
ASL (opr)	Arithmetic Shift Left		EXT IND,X IND,Y INH INH	78 hh ll 68 ff 18 68 ff 48 58	3 6 2 6 3 7 1 2 1 2			-	-	-	-	↕	↕	↕	↕
ASLD	Arithmetic Shift Left Double		INH	05	1 3			-	-	-	-	↕	↕	↕	↕
ASR	Arithmetic Shift Right		EXT IND,X IND,Y INH INH	77 hh ll 67 ff 18 67 ff 47 57	3 6 2 6 3 7 1 2 1 2			-	-	-	-	↕	↕	↕	↕
BCLR (opr) (mask)	Clear Bit(s)	$M \bullet (mm) \rightarrow M$	DIR IND,X IND,Y	15 dd mm 1D ff mm 18 1D ff mm	3 6 3 7 4 8			-	-	-	-	↕	↕	0	-
BITA (opr)	Bit(s) test A with Memory	$A \bullet M$	A IMM A DIR A EXT A IND,X A IND,Y	85 ii 95 dd B5 hh ll A5 ff 18 A5 ff	2 2 2 3 3 4 2 4 3 5			-	-	-	-	↕	↕	0	-
BITB (opr)	Bit(s) test B with Memory	$B \bullet M$	B IMM B DIR B EXT B IND,X B IND,Y	C5 ii D5 dd F5 hh ll E5 ff 18 E5 ff	2 2 2 3 3 4 2 4 3 5			-	-	-	-	↕	↕	0	-
BSET (opr) (msk)	Set Bit(s)	$M + mm \rightarrow M$	DIR IND,X IND,Y	14 dd mm rr 1C ff mm rr 18 1C ff mm rr	3 6 3 7 4 8			-	-	-	-	↕	↕	0	-
CBA	Compare A to B	$A - B$	INH	11		1	2	-	-	-	-	↕	↕	↕	↕

Mnemonic	Operation	Boolean expression	Addressing mode for operand	Machine coding (hexadecimal)		Bytes	Cycles	Condition codes							
				Opcode	Operand(s)			S	X	H	I	N	Z	V	C
CLR (opr)	Clear Memory Byte	$0 \rightarrow M$	EXT IND,X IND,Y	7F 6F 18 6F	hh ll ff ff	3 2 3	6 6 7	-	-	-	-	0	1	0	0
CLRA	Clear Accumulator A	$0 \rightarrow A$	A INH	4F		1	2	-	-	-	-	0	1	0	0
CLRB	Clear Accumulator B	$0 \rightarrow B$	B INH	5F		1	2	-	-	-	-	0	1	0	0
CMPA (opr)	Compare A to Memory	$A - M$	A IMM A DIR A EXT A IND,X A IND,Y	81 91 B1 A1 18 A1	ii dd hh ll ff ff	2 2 3 2 3	2 3 4 4 5	-	-	-	-	↕	↕	↕	↕
CMPB (opr)	Compare B to Memory	$B - M$	B IMM B DIR B EXT B IND,X B IND,Y	C1 D1 F1 E1 18 E1	ii dd hh ll ff ff	2 2 3 2 3	2 3 4 4 5	-	-	-	-	↕	↕	↕	↕
COM (opr)	1's Complement Memory Byte	$\$FF - M \rightarrow M$	EXT IND,X IND,Y	73 63 18 63	hh ll ff ff	3 2 3	6 6 7	-	-	-	-	↕	↕	0	1
COMA	1's Complement A	$\$FF - A \rightarrow A$	A INH	43		1	2	-	-	-	-	↕	↕	0	1
COMB	1's Complement B	$\$FF - B \rightarrow B$	B INH	53		1	2	-	-	-	-	↕	↕	0	1
CPD (opr)	Compare D to Memory 16 - Bit	$D - M:M + 1$	IMM DIR EXT IND,X IND,Y	1A 83 1A 93 1A B3 1A A3 CD A3	jj kk dd hh ll ff ff	4 3 4 3 3	5 6 7 7 7	-	-	-	-	↕	↕	↕	↕
DAA	Decimal Adjust A	Adjust Sum to BCD	INH	19		1	2	-	-	-	-	↕	↕	↕	↕
DEC (opr)	Decrement Memory Byte	$M - 1 \rightarrow M$	EXT IND,X IND,Y	7A 6A 18 6A	hh ll ff ff	3 2 3	6 6 7	-	-	-	-	↕	↕	↕	-
DECA	Decrement Accumulator A	$A - 1 \rightarrow A$	A INH	4A		1	2	-	-	-	-	↕	↕	↕	-
DECB	Decrement Accumulator B	$B - 1 \rightarrow B$	B INH	5A		1	2	-	-	-	-	↕	↕	↕	-
EORA (opr)	Exclusive OR A with Memory	$A \oplus M \rightarrow A$	A IMM A DIR A EXT A IND,X A IND,Y	88 98 B8 A8 18 A8	ii dd hh ll ff ff	2 2 3 2 3	2 3 4 4 5	-	-	-	-	↕	↕	0	-
EORB (opr)	Exclusive OR B with Memory	$B \oplus M \rightarrow B$	B IMM B DIR B EXT B IND,X B IND,Y	C8 D8 F8 E8 18 E8	ii dd hh ll ff ff	2 2 3 2 3	2 3 4 4 5	-	-	-	-	↕	↕	0	-
INC	Increment Memory Byte	$M + 1 \rightarrow M$	EXT IND,X IND,Y	7C 6C 18 6C	hh ll ff ff	3 2 3	6 6 7	-	-	-	-	↕	↕	↕	-
INCA	Increment Accumulator A	$A + 1 \rightarrow A$	A INH	4C		1	2	-	-	-	-	↕	↕	↕	-
INCB	Increment Accumulator B	$B + 1 \rightarrow B$	B INH	5C		1	2	-	-	-	-	↕	↕	↕	-
LDAA (opr)	Load Accumulator A	$M \rightarrow A$	A IMM A DIR A EXT A IND,X A IND,Y	86 96 B6 A6 18 A6	ii dd hh ll ff ff	2 2 3 2 3	2 3 4 4 5	-	-	-	-	↕	↕	0	-
LDAB (opr)	Load Accumulator B	$M \rightarrow B$	B IMM B DIR B EXT B IND,X B IND,Y	C6 D6 F6 E6 18 E6	ii dd hh ll ff ff	2 2 3 2 3	2 3 4 4 5	-	-	-	-	↕	↕	0	-
LDD (opr)	Load Double Accumulator D	$M \rightarrow A,$ $M + 1 \rightarrow B$	IMM DIR EXT IND,X IND,Y	CC DC FC EC 18 EC	jj kk dd hh ll ff ff	3 2 3 2 3	3 4 5 5 6	-	-	-	-	↕	↕	0	-

Mnemonic	Operation	Boolean expression	Addressing mode for operand	Machine coding (hexadecimal)		Bytes	Cycles	Condition codes							
				Opcode	Operand(s)			S	X	H	I	N	Z	V	C
LSL (opr)	Logical Shift Left		EXT IND,X IND,Y A INH B INH	78 68 18 68 48 58	hh ll ff ff	3 2 3 1 1	6 6 7 2 2	-	-	-	-	↕	↕	↕	↕
LSLD	Logical Shift Left Double		INH	05		1	3	-	-	-	-	↕	↕	↕	↕
LSR (opr)	Logical Shift Right		EXT IND,X IND,Y A INH B INH	74 64 18 64 44 54	hh ll ff ff	3 2 3 1 1	6 6 7 2 2	-	-	-	-	0	↕	↕	↕
LSRD	Logical Shift Right Double		INH	04		1	3	-	-	-	-	0	↕	↕	↕
MUL	Multiply 8 by 8	$A \times B \rightarrow D$	INH	3D		1	10	-	-	-	-	-	-	-	↕
NEG (opr)	2's Complement Memory Byte	$0 - M \rightarrow M$	EXT IND,X IND,Y	70 60 18 60	hh ll ff ff	3 2 3	6 6 7	-	-	-	-	↕	↕	↕	↕
NEGA	2's Complement A	$0 - A \rightarrow A$	A INH	40		1	2	-	-	-	-	↕	↕	↕	↕
NEGB	2's Complement B	$0 - B \rightarrow B$	B INH	50		1	2	-	-	-	-	↕	↕	↕	↕
ORAA (opr)	OR Accumulator A (inclusive)	$A + M \rightarrow A$	A IMM A DIR A EXT A IND,X A IND,Y	8A 9A BA AA 18 AA	ii dd hh ll ff ff	2 2 3 3 3	2 3 4 4 5	-	-	-	-	↕	↕	0	-
ORAB (opr)	OR Accumulator B (inclusive)	$B + M \rightarrow B$	B IMM B DIR B EXT B IND,X B IND,Y	CA DA FA EA 18 EA	ii dd hh ll ff ff	2 2 3 3 3	2 3 4 4 5	-	-	-	-	↕	↕	0	-
PSHA	Push A onto Stack	$A \uparrow \text{Stk}, SP = SP - 1$	A INH	36		1	3	-	-	-	-	-	-	-	-
PSHB	Push B onto Stack	$B \uparrow \text{Stk}, SP = SP - 1$	B INH	37		1	3	-	-	-	-	-	-	-	-
PULA	Pull A from Stack	$SP = SP + 1, A \downarrow \text{Stk}$	A INH	32		1	4	-	-	-	-	-	-	-	-
PULB	Pull B from Stack	$SP = SP + 1, B \downarrow \text{Stk}$	B INH	33		1	4	-	-	-	-	-	-	-	-
ROL (opr)	Rotate Left		EXT IND,X IND,Y A INH B INH	79 69 18 69 49 59	hh ll ff ff	3 2 3 1 1	6 6 7 2 2	-	-	-	-	↕	↕	↕	↕
ROR (opr)	Rotate Right		EXT IND,X IND,Y A INH B INH	76 66 18 66 46 56	hh ll ff ff	3 2 3 1 1	6 6 7 2 2	-	-	-	-	↕	↕	↕	↕
SBA	Subtract B from A	$A - B \rightarrow A$	INH	10		1	2	-	-	-	-	↕	↕	↕	↕
SBCA (opr)	Subtract with Carry from A	$A - M - C \rightarrow A$	A IMM A DIR A EXT A IND,X A IND,Y	82 92 B2 A2 18 A2	ii dd hh ll ff ff	2 2 3 2 3	2 3 4 4 5	-	-	-	-	↕	↕	↕	↕
SBCB (opr)	Subtract with Carry from B	$B - M - C \rightarrow B$	B IMM B DIR B EXT B IND,X B IND,Y	C2 D2 F2 E2 18 E2	ii dd hh ll ff ff	2 2 3 2 3	2 3 4 4 5	-	-	-	-	↕	↕	↕	↕
STAA (opr)	Store Accumulator A	$A \rightarrow M$	A DIR A EXT A IND,X A IND,Y	97 B7 A7 18 A7	dd hh ll ff ff	2 3 2 3	3 4 4 5	-	-	-	-	↕	↕	0	-
STAB (opr)	Store Accumulator B	$B \rightarrow M$	B DIR B EXT B IND,X B IND,Y	D7 F7 E7 18 E7	dd hh ll ff ff	2 3 2 3	3 4 4 5	-	-	-	-	↕	↕	0	-
STD (opr)	Store Accumulator D	$A \rightarrow M, B \rightarrow M + 1$	DIR EXT IND,X IND,Y	DD FD ED 18 ED	dd hh ll ff ff	2 3 2 3	4 5 5 6	-	-	-	-	↕	↕	0	-

Mnemonic	Operation	Boolean expression	Addressing mode for operand	Machine coding (hexadecimal)		Bytes	Cycles	Condition codes							
				Opcode	Operand(s)			S	X	H	I	N	Z	V	C
SUBA (opr)	Subtract Memory from A	$A - M \rightarrow A$	A IMM	80	ii	2	2	-	-	-	-	↕	↕	↕	↕
			A DIR	90	dd	2	3								
			A EXT	B0	hh ll	3	4								
			A IND,X	A0	ff	2	4								
			A IND,Y	18 A0	ff	3	5								
SUBB (opr)	Subtract Memory from B	$B - M \rightarrow B$	B IMM	C0	ii	2	2	-	-	-	-	↕	↕	↕	↕
			B DIR	D0	dd	2	3								
			B EXT	F0	hh ll	3	4								
			B IND,X	E0	ff	2	4								
			B IND,Y	18 E0	ff	3	5								
SUBD (opr)	Subtract Memory from D	$D - M:M + 1 \rightarrow D$	IMM	83	jj kk	3	4	-	-	-	-	↕	↕	↕	↕
			DIR	93	dd	2	5								
			EXT	B3	hh ll	3	6								
			IND,X	A3	ff	2	6								
			IND,Y	18 A3	ff	3	7								
TAB	Transfer A to B	$A \rightarrow B$	INH	16		1	2	-	-	-	-	↕	↕	0	-
TBA	Transfer B to A	$B \rightarrow A$	INH	17		1	2	-	-	-	-	↕	↕	0	-
TEST	TEST (Only in test Modes)	Address Bus Counts	INH	00		1	·	-	-	-	-	-	-	-	-
TST (opr)	Test for Zero or Minus	$M - 0$	EXT	7D	hh ll	3	6	-	-	-	-	↕	↕	0	0
			IND,X	6D	ff	2	6								
			IND,Y	18 6D	ff	3	7								
		$A - 0$	A INH	4D		1	2	-	-	-	-	↕	↕	0	0
TSTA															
TSTB		$B - 0$	B INH	5D		1	2	-	-	-	-	↕	↕	0	0
XGDX	Exchange D with X	$IX \rightarrow D, D \rightarrow IX$	INH	8F		1	3	-	-	-	-	-	-	-	-
XGDY	Exchange D with Y	$IY \rightarrow D, D \rightarrow IY$	INH	18 8F		2	4	-	-	-	-	-	-	-	-

Notes:

Cycle:

- = Infinity or until reset occurs.
- .. = 12 cycles are used beginning with the opcode fetch. A wait state is entered which remains in effect for an integer number of MCU E-clock cycles (n) until an interrupt is recognized. Finally, two additional cycles are used to fetch the appropriate interrupt vector (total = 14 + n).

Operands:

- dd = 8-bit direct address \$0000-\$00FF. (High byte assumed to be \$00)
- ff = 8-bit positive offset \$00 (0) to \$FF (255) added to index.
- hh = High order byte of 16-bit extended address.
- ii = One byte of immediate data.
- jj = High order byte of 16-bit immediate data.
- kk = Low order byte of 16-bit immediate data.
- ll = Low order byte of 16-bit extended address.
- mm = 8-bit mask (set bits to be affected).
- rr = Signed relative offset \$80 (-128) to \$7F (+127). Offset relative to the address following the machine code offset byte.

Condition codes:

- = Bit not changed
- 0 = Always cleared (logic 0)
- 1 = Always set (logic 1)
- ↕ = Bit cleared or set depending on operation.
- ↓ = Bit may be cleared, cannot become set.

(Source: Motorola 1993, *M68HC11 E Series, Programming reference guide*, pp. 19–35.)

Table 11.2: Index registers and stack manipulation instructions

Mnemonic	Operation	Boolean expression	Addressing mode for operand	Machine coding (hexadecimal)		Bytes	Cycles	Condition codes							
				Opcode	Operand(s)			S	X	H	I	N	Z	V	C
CPX (opr)	Compare X to Memory 16-Bit	$IX - M:M + 1$	IMM DIR EXT IND,X IND,Y	8C	jj kk	3	4	-	-	-	-	↕	↕	↕	↕
				9C	dd	2	5								
				BC	hh ll	3	6								
				AC	ff	2	6								
				CD AC	ff	3	7								
CPY (opr)	Compare Y to Memory 16-Bit	$IY - M:M + 1$	IMM DIR EXT IND,X IND,Y	18 8C	jj kk	4	5	-	-	-	-	↕	↕	↕	↕
				18 9C	dd	3	6								
				18 BC	hh ll	4	7								
				1A AC	ff	3	7								
				18 AC	ff	3	7								
DES	Decrement Stack Pointer	$SP - 1 \rightarrow SP$	INH	34		1	3	-	-	-	-	-	-	-	-
DEX	Decrement Index Register X	$IX - 1 \rightarrow IX$	INH	09		1	3	-	-	-	-	-	↕	-	-
DEY	Decrement Index Register Y	$IY - 1 \rightarrow IY$	INH	18 09		2	4	-	-	-	-	-	↕	-	-
FDIV	Fractional Divide 16 by 16	$D/IX \rightarrow IX; r \rightarrow D$	INH	03		1	41	-	-	-	-	-	↕	↕	↕
IDIV	Integer Divide 16 by 16	$D/IX \rightarrow IX; r \rightarrow D$	INH	02		1	41	-	-	-	-	-	↕	0	↕
INS	Increment Stack Pointer	$SP + 1 \rightarrow SP$	INH	31		1	3	-	-	-	-	-	-	-	-
INX	Increment Index Register X	$IX + 1 \rightarrow IX$	INH	08		1	3	-	-	-	-	-	↕	-	-
INY	Increment Index Register Y	$IY + 1 \rightarrow IY$	INH	18 08		2	4	-	-	-	-	-	↕	-	-
LDS (opr)	Load Stack Pointer	$M:M + 1 \rightarrow SP$	IMM DIR EXT IND,X IND,Y	8E	jj kk	3	3	-	-	-	-	↕	↕	0	-
				9E	dd	2	4								
				BE	hh ll	3	5								
				AE	ff	2	5								
				18 AE	ff	3	6								
LDX (opr)	Load Index Register X	$M:M + 1 \rightarrow IX$	IMM DIR EXT IND,X IND,Y	CE	jj kk	3	3	-	-	-	-	↕	↕	0	-
				DE	dd	2	4								
				FE	hh ll	3	5								
				EE	ff	2	5								
				CD EE	ff	3	6								
LDY (opr)	Load Index Register Y	$M:M + 1 \rightarrow IY$	IMM DIR EXT IND,X IND,Y	18 CE	jj kk	3	4	-	-	-	-	↕	↕	0	-
				18 DE	dd	2	5								
				18 FE	hh ll	3	6								
				1A AE	ff	2	6								
				18 EE	ff	3	6								
PSHX	Push X onto Stack (Lo First)	$IX \uparrow \text{Stk}, SP = SP - 2$	INH	3C		1	4	-	-	-	-	-	-	-	-
PSHY	Push Y onto Stack (Lo First)	$IY \uparrow \text{Stk}, SP = SP - 2$	INH	18 3C		2	5	-	-	-	-	-	-	-	-
PULX	Pull X from Stack (Hi First)	$SP = SP + 1, IX \downarrow \text{Stk}$	INH	38		1	5	-	-	-	-	-	-	-	-
PULY	Pull Y from Stack (Hi First)	$SP = SP + 1, IY \downarrow \text{Stk}$	INH	18 38		2	6	-	-	-	-	-	-	-	-
STS (opr)	Store Stack Pointer	$SP \rightarrow M:M + 1$	DIR EXT IND,X IND,Y	9F	dd	2	4	-	-	-	-	↕	↕	0	-
				BF	hh ll	3	5								
				AF	ff	2	5								
				18 AF	ff	3	6								
STX (opr)	Store Index Register X	$IX \rightarrow M:M + 1$	DIR EXT IND,X IND,Y	DF	dd	2	4	-	-	-	-	↕	↕	0	-
				FF	hh ll	3	5								
				EF	ff	2	5								
				CD EF	ff	3	6								
STY (opr)	Store Index Register Y	$IY \rightarrow M:M + 1$	DIR EXT IND,X IND,Y	18 DF	dd	3	5	-	-	-	-	↕	↕	0	-
				18 FF	hh ll	4	6								
				1A EF	ff	3	6								
				18 EF	ff	3	6								
TSX	Transfer Stack Pointer to X	$SP + 1 \rightarrow IX$	INH	30		1	3	-	-	-	-	-	-	-	-
TSY	Transfer Stack Pointer to Y	$SP + 1 \rightarrow IY$	INH	18 30		2	4	-	-	-	-	-	-	-	-
TXS	Transfer X to Stack Pointer	$IX - 1 \rightarrow SP$	INH	35		1	3	-	-	-	-	-	-	-	-

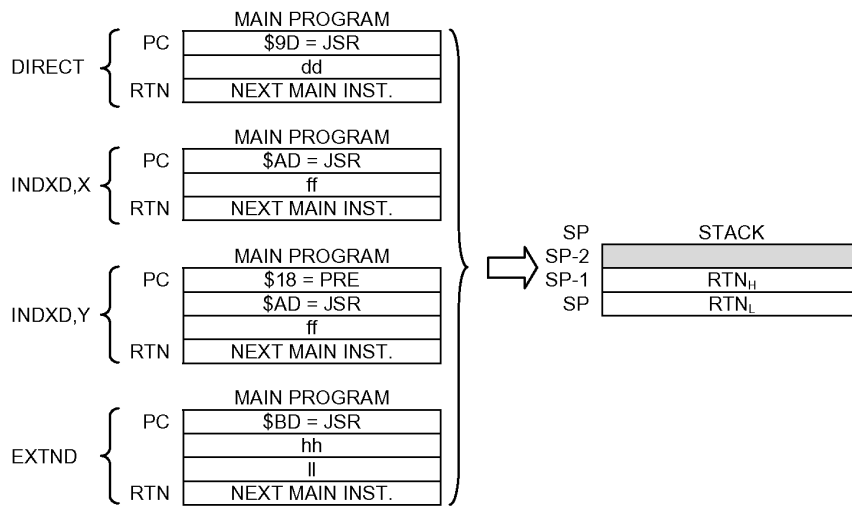
Mnemonic	Operation	Boolean expression	Addressing mode for operand	Machine coding (hexadecimal)		Bytes	Cycles	Condition codes							
				Opcode	Operand(s)			S	X	H	I	N	Z	V	C
TYS	Transfer Y to Stack Pointer	$Y - 1 \rightarrow SP$	INH	18 35		2	4	-	-	-	-	-	-	-	-

Table 11.3: Jump and branch instructions

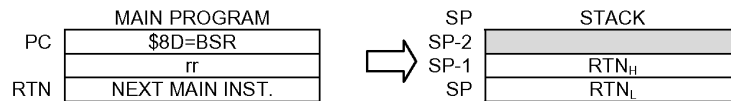
Mnemonic	Operation	Boolean expression	Addressing mode for operand	Machine coding (hexadecimal)		Bytes	Cycles	Condition codes							
				Opcode	Operand(s)			S	X	H	I	N	Z	V	C
BCC (rel)	Branch if Carry Clear	$? C = 0$	REL	24	rr	2	3	-	-	-	-	-	-	-	-
BCS (rel)	Branch if Carry Set	$? C = 1$	REL	25	rr	2	3	-	-	-	-	-	-	-	-
BEQ (rel)	Branch if = Zero	$? Z = 1$	REL	27	rr	2	3	-	-	-	-	-	-	-	-
BGE (rel)	Branch if \geq Zero	$? N \oplus V = 0$	REL	2C	rr	2	3	-	-	-	-	-	-	-	-
BGT (rel)	Branch if $>$ Zero	$? Z + (N \oplus V) = 0$	REL	2E	rr	2	3	-	-	-	-	-	-	-	-
BHI (rel)	Branch if Higher	$? C + Z = 0$	REL	22	rr	2	3	-	-	-	-	-	-	-	-
BHS (rel)	Branch if Higher or Same	$? C = 0$	REL	24	rr	2	3	-	-	-	-	-	-	-	-
BLE (rel)	Branch if \leq Zero	$? Z + (N \oplus V) = 1$	REL	2F	rr	2	3	-	-	-	-	-	-	-	-
BLO (rel)	Branch if Lower	$? C = 1$	REL	25	rr	2	3	-	-	-	-	-	-	-	-
BLS (rel)	Branch if Lower or Same	$? C + Z = 1$	REL	23	rr	2	3	-	-	-	-	-	-	-	-
BLT (rel)	Branch if $<$ Zero	$? N \oplus V = 1$	REL	2D	rr	2	3	-	-	-	-	-	-	-	-
BMI (rel)	Branch if Minus	$? N = 1$	REL	2B	rr	2	3	-	-	-	-	-	-	-	-
BNE (rel)	Branch if Not = Zero	$? Z = 0$	REL	26	rr	2	3	-	-	-	-	-	-	-	-
BPL (rel)	Branch if Plus	$? N = 0$	REL	2A	rr	2	3	-	-	-	-	-	-	-	-
BRA (rel)	Branch Always	$? 1 = 1$	REL	20	rr	2	3	-	-	-	-	-	-	-	-
BRCLR (opr) (msk) (rel)	Branch if Bit(s) Clear	$? M \bullet mm = 0$	DIR IND,X IND,Y	13 1F 18 1F	dd mm rr ff mm rr ff mm rr	4 4 5	6 7 8	-	-	-	-	-	-	-	-
BRN (rel)	Branch Never	$? 1 = 0$	REL	21	rr	2	3	-	-	-	-	-	-	-	-
BRSET (opr) (msk) (rel)	Branch if Bit(s) Set	$? (M) \bullet mm = 0$	DIR IND,X IND,Y	12 1E 18 1E	dd mm rr ff mm rr ff mm rr	4 4 5	6 7 8	-	-	-	-	-	-	-	-
BSR (rel)	Branch to Subroutine	See Special Ops	REL	8D	rr	2	6	-	-	-	-	-	-	-	-
BVC (rel)	Branch if Overflow Clear	$? V = 0$	REL	28	rr	2	3	-	-	-	-	-	-	-	-
BVS (rel)	Branch if Overflow Set	$? V = 1$	REL	29	rr	2	3	-	-	-	-	-	-	-	-
JMP (opr)	Jump	See Special Ops	EXT IND,X IND,Y	7E 6E 18 6E	hh ll ff ff	3 2 3	3 3 4	-	-	-	-	-	-	-	-
JSR (opr)	Jump to Subroutine	See Special Ops	DIR EXT IND,X IND,Y	9D BD AD 18 AD	dd hh ll ff ff	2 3 2 3	5 6 6 7	-	-	-	-	-	-	-	-
NOP	No Operation	No Operation	INH	01		1	2	-	-	-	-	-	-	-	-
RTI	Return from Interrupt	See Special Ops	INH	3B		1	12	↕	↕	↕	↕	↕	↕	↕	↕
RTS	Return from Subroutine	See Special Ops	INH	39		1	5	-	-	-	-	-	-	-	-
STOP	Stop Internal Clocks		INH	CF		1	2	-	-	-	-	-	-	-	-
SWI	Software Interrupt	See Special Ops	INH	3F		1	14	-	-	-	1	-	-	-	-
WAI	Wait for Interrupt	Stack Regs and WAIT	INH	3E		1	..	-	-	-	-	-	-	-	-

(Source: Motorola 1993, *M68HC11 E Series, Programming reference guide*, pp. 19–35.)

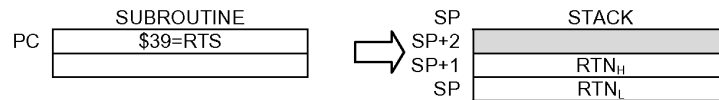
Special operations



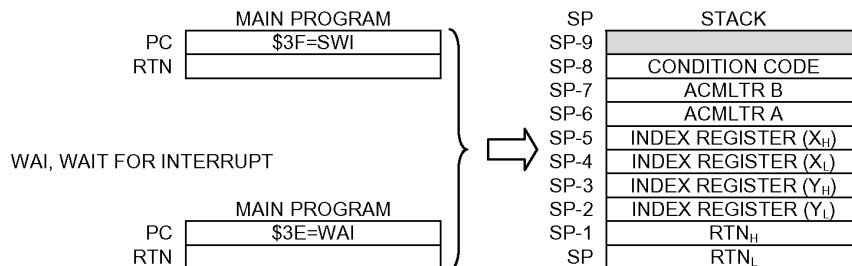
BSR, BRANCH TO SUBROUTINE



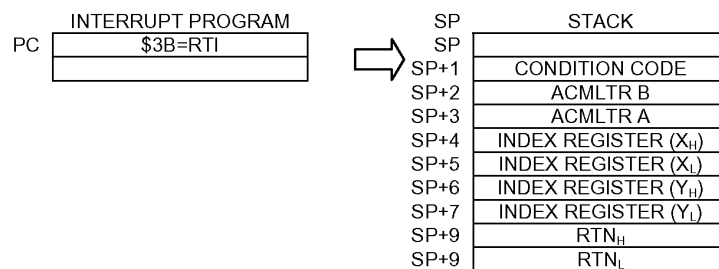
RTS, RETURN FROM SUBROUTINE



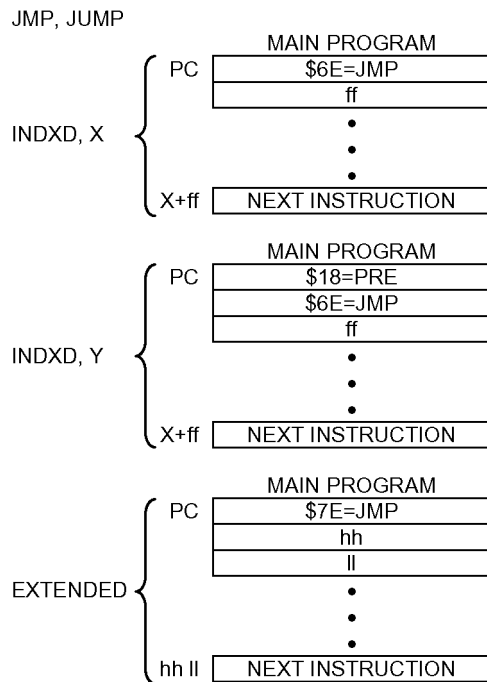
SWI, SOFTWARE INTERRUPT




RTI, RETURN FROM INTERRUPT



Special operations, cont.



LEGEND

RTN	=	Address of next instruction in main program to be executed upon return from subroutine.
RTN _H	=	Most significant byte of return address
RTN _L	=	Least significant byte of return address
	=	Stack pointer location after execution
dd	=	8-bit direct address \$0000-\$00FF. (High byte assumed to be \$00)
ff	=	8-bit positive offset \$00 (0) to \$FF (255) added to index.
hh	=	High order byte of 16-bit extended address.
ll	=	Low order byte of 16-bit extended address.
rr	=	Signed relative offset \$80 (-128) to \$7F (+127). (Offset relative to the address following the machine code offset byte)

(Source: Motorola 1993, *M68HC11 E Series, Programming reference guide*, pp. 36–7.)

Table 11.4: Condition code register manipulation instructions

Mnemonic	Operation	Boolean expression	Addressing mode for operand	Machine coding (hexadecimal) Opcode Operand(s)	Bytes	Cycles	Condition codes							
							S	X	H	I	N	Z	V	C
CLC	Clear Carry Bit	0 → C	INH	0C		1 2	-	-	-	-	-	-	-	0
CLI	Clear Interrupt Mask	0 → I	INH	0E		1 2	-	-	-	0	-	-	-	-
CLV	Clear Overflow Flag	0 → V	INH	0A		1 2	-	-	-	-	-	-	0	-
SEC	Set Carry	1 → C	INH	0D		1 2	-	-	-	-	-	-	-	1
SEI	Set Interrupt Mask	1 → I	INH	0F		1 2	-	-	-	1	-	-	-	-
SEV	Set Overflow Flag	1 → V	INH	0B		1 2	-	-	-	-	-	-	1	-
TAP	Transfer A to CCR	A → CCR	INH	06		1 2	↕	↓	↕	↕	↕	↕	↕	↕
TPA	Transfer CC Register to A	CCR → A	INH	07		1 2	-	-	-	-	-	-	-	-

(Source: Motorola 1993, *M68HC11 E Series, Programming reference guide*, pp. 19–35.)

General points to note are that within each table the instructions are listed alphabetically and that each table covers a specific area.

Table 11.1 gives a list of instructions that:

- include all instructions which involve either Accumulators A, B or D in their execution. e.g. 'ADDA' – add the operand addressed by the instruction into Accumulator A
- includes all instructions which operate directly on the contents of a memory location. e.g. CLR 'M' – clear the memory location with address 'M'.

Table 11.2 includes all instruction that operates on the contents of either the **index registers** or the **'stack pointer'**.

Table 11.3 includes the complete list of **'Branch'** and **'Jump'** instructions.

Table 11.4 includes the instructions that can be used to manipulate the contents of the condition code register.

To appreciate the full details provided for each instruction consider the first few instructions from table 11.1 and the legend of symbols used.

Table 11.5: Portion of the accumulator and memory instruction

Mnemonic	Operation	Boolean expression	Addressing mode for operand	Machine coding (hexadecimal)		Bytes	Cycles	Condition codes							
				Opcode	Operand(s)			S	X	H	I	N	Z	V	C
ADDA (opr)	Add Memory to A	$A + M \rightarrow A$	A IMM	8B	ii	2	2	-	-	↕	-	↕	↕	↕	↕
			A DIR	9B	dd	2	3								
			A EXT	BB	hh ll	3	4								
			A IND,X	AB	ff	2	4								
			A IND,Y	18 AB	ff	3	5								
ADDB (opr)	Add Memory to B	$B + M \rightarrow B$	B IMM	CB	ii	2	2	-	-	↕	-	↕	↕	↕	↕
			B DIR	DB	dd	2	3								
			B EXT	FB	hh ll	3	4								
			B IND,X	EB	ff	2	4								
			B IND,Y	18 EB	ff	3	5								

Legend:

Opcode Operation Code (Hexadecimal)
 Cycles Number of machine cycles
 Bytes Number of program bytes
 + Arithmetic plus
 - Arithmetic minus
 • Boolean AND
 ↑Stk Memory location pointed to by the Stack Pointer

+ Boolean Inclusive OR
 ⊕ Boolean Exclusive OR
 → Transfer into
 0 Bit = Zero
 00 Byte = Zero

Condition code symbols:

H Half carry from bit 3
 I Interrupt mask
 N Negative (sign bit)
 Z Zero (byte)
 V Overflow 2's complement
 C Carry from bit 7
 R Reset always
 S Set always
 ↕ Bit cleared or set depending on operation
 - Not affected

Starting from the left most column of table 11.5 we have the following:

- Mnemonic** – The mnemonics used for the Add instruction for the 68HC11 are
 - ADDA – for addition into accumulator A
 - ADDB – for addition into accumulator B
- Operation** – A brief explanation of the instruction.
- Boolean expression** – For all tables, this column provides a quick reference as to the operation performed by the instruction.

- **Addressing mode for operand** – There are 6 addressing modes possible. i.e. Inherent (INH), Immediate (IMM), Direct (DIR), Extended (EXT), Indexed (IND,X or IND,Y), and Relative (REL).

Consider the instruction, written in assembly code

ADDA # \$45

We understand this to mean ‘add the hexadecimal number 45 into Accumulator A’

(i.e. immediate addressing). N.B. ‘\$’ indicates ‘hexadecimal’, ‘#’ indicates immediate addressing.

or $(\text{Acc A}) + 45_{16} \rightarrow \text{Acc A}$

From our discussions to date this would be stored in memory as

Byte 1 (Opcode)	Opcode (ADD)	Register field (A)	Addressing mode # (Immediate)
Byte 2 (ii)	8 bit Operand (45_{16})		

In fact the instruction is stored as two 8-bit bytes. The information contained in the **Machine Coding Column (Opcode)** allows us to select the appropriate binary code for the first byte.

The information supplied under ‘Opcode’ is the Operation Code or machine code, specified as a hexadecimal number i.e. $\text{OPCODE} = 8B_{16}$.

The actual binary digits stored for this instruction are:

Byte 1 $8B_{16}$	1 0 0 0 1 0 1 1
Byte 2 45_{16}	0 1 0 0 0 1 0 1



Note

The hexadecimal number given in the ‘Opcode column’ is the machine code that goes in the first byte of the instruction.

Additional information given about ‘ADDA using immediate addressing’ appears as follows:

- In the ‘**Cycles column**’ there is the entry ‘2’.

A check against the legend indicates this is the ‘number of MPU cycles’ it takes to fetch and execute this instruction.

This information is provided so that at the end of a program the programmer can calculate how long the program will take to execute. This may be important because:

- i. the program may be controlling some external process and strict time limits may have to be met for safety reasons. To ensure that the program can be executed within the limits the time for execution must be calculated.
 - ii. the programmer may have to program a ‘delay’ e.g. between successive outputs to a terminal. This can be done by writing a program to count down an accumulator to zero. The time taken gives the delay and is determined by the execution times of the instructions involved.
- In the ‘Bytes column’ there is an entry 2.

Consulting the legend this column indicates the number of program bytes required to store this instruction in memory.

i.e. ‘ADDA #\$45’ is a 2 byte instruction.

Byte 1 8B ₁₆	1 0 0 0 1 0 1 1
Byte 2 45 ₁₆	0 1 0 0 0 1 0 1

A similar analysis also applies for the other addressing modes.

Example 1

What is the machine code for the ADDB \$45, X and ADDB \$45, Y instructions.

Solution

The entries for the ‘ADDB’ row of table 11.1 are:

Mnemonic	Operation	Boolean expression	Addressing mode for operand	Machine coding (hexadecimal)		Bytes	Cycles	Condition codes							
				Opcode	Operand(s)			S	X	H	I	N	Z	V	C
ADDB (opr)	Add Memory to B	B + M → B	B IMM	CB	ii	2	2	-	-	↓	-	↓	↓	↓	↓
			B DIR	DB	dd	2	3								
			B EXT	FB	hh ll	3	4								
			B IND,X	EB	ff	2	4								
			B IND,Y	18 EB	ff	3	5								

From our studies of indexed addressing we know the instruction requires at least two bytes and this is confirmed by the entry in the ‘**Bytes** column’ for addressing modes ‘**IND,X** and **IND,Y**’.

For IND,X the two bytes will contain the following:

Byte 1 (EB ₁₆)	Opcode (ADD)	Register field (B)	Addressing mode (Indexed)
Byte 2 (ff)	A positive 8 bit offset (45 ₁₆)		

For IND,Y the three bytes will contain the following:

Byte 1	Pre-byte
---------------	----------

(18 ₁₆)			
Byte 2 (EB ₁₆)	Opcode (ADD)	Register field (B)	Addressing mode (Indexed)
Byte 3 (ff)	A positive 8 bit offset (45 ₁₆)		

The machine code for these instruction is therefore:

ADDB \$45, X

Byte 2 EB ₁₆	1 1 1 0 1 0 1 1
Byte 3 45 ₁₆	0 1 0 0 0 1 0 1

ADDB \$45, Y

Byte 1 18 ₁₆	0 0 0 1 1 0 0 0
Byte 2 EB ₁₆	1 1 1 0 1 0 1 1
Byte 3 45 ₁₆	0 1 0 0 0 1 0 1

The offset of 45₁₆ is the value specified by the programmer in the instruction.

Example 2

What is the machine code for the instruction?

ADDA VALUE

where 'VALUE' is the name given to a memory location (00B6₁₆) that contains the number to be added to the accumulator.

Solution

This instruction may be written in an alternative form as:

ADDA \$00B6

The concept of using a 'name' for a memory location rather than specifying the 'actual address' is one that you are encouraged to adopt. This method is used extensively when writing assembly language programs because it makes the program easier to follow and debug, particularly if someone else has written it.

When using ‘names for locations’ it is essential you adopt the procedure used for writing assembly language programs, where at the beginning of each program all names are declared. These ‘names for locations’ are referred to as **labels**.

e.g. VALUE EQU \$00B6

The ‘EQU’ directive is a signal to the assembler that wherever ‘VALUE’ appears in the program the address ‘00B6₁₆’ is intended. ‘VALUE’ is a label.

The solution to this problem then is to find the machine code for

ADDA \$00B6

As there is no ‘addressing symbol’ used (such as ‘#’ or ‘, X’) and the instruction is not a ‘**branch instruction**’ then the addressing mode intended is either

‘**direct addressing**’ or ‘**extended addressing**’

Because the address of the memory location is

00B6₁₆

direct addressing can be used since the ‘high-byte’ of the address is ‘00’. The instruction is then

ADDA \$B6

Previous work shows the information supplied in this 2-byte instruction is:

Byte 1 (9B ₁₆)	Opcode (ADD)	Register field (A)	Addressing mode (Direct)
Byte 2 (dd)	Low byte of the operand address (B6 ₁₆)		

The execution of this instruction will result in the contents of memory location 00B6₁₆ being added to Accumulator A.

Consulting the ‘Machine Coding’ column of table 11.1 gives the actual machine code as:

Byte 1 9B ₁₆	1 0 0 1 1 0 1 1
Byte 2 B6 ₁₆	1 0 1 1 0 1 1 0

Checking the ‘Bytes’ column confirms it is a 2 byte instruction and the ‘Cycles’ column indicates that it takes 3 machine cycles to execute.

This same instruction could be coded into machine code using ‘Extended addressing’ in which case

ADDA \$00B6

would be coded as a 3 byte instruction as follows:

Byte 1 BB ₁₆	1 0 1 1 1 0 1 1
Byte 2 00 ₁₆	0 0 0 0 0 0 0 0
Byte 3 B6 ₁₆	1 0 1 1 0 1 1 0

Note the execution time is now 4 machine cycles since an extra byte has to be fetched from memory.

Although the program would still run and would provide the correct operation it would use an additional memory location and would take longer to execute, i.e. it is less efficient.

Example 3

Why is there no INH (inherent) entry in the 'Addressing mode' column for the 'ADDA' and 'ADDB' instructions?

Solution

Inherent addressing covers both 'implied addressing' and 'accumulator addressing'. In each case the 'addressing mode' is inherent in the OPCODE.

e.g. CLRA – clear Accumulator A (accumulator addressing)
INX – increment the index register (implied addressing)

In each case no operand need be specified as the operation involves only the 'Accumulator A' in 'CLRA' and the 'index register' in 'INX'.

For the 'ADDA' and 'ADDB' instructions obviously some 'operand' has to be specified, i.e. some memory contents or data have to be specified to be added to the appropriate accumulator. Therefore 'implied addressing' and 'accumulator addressing' have no significance for these instructions.

Example 4

It has been stated several times throughout this section, that the hexadecimal digits provided in the 'Machine Code' column of the instruction set, is the machine code for the 'first byte' of the instruction and represents

Opcode	Register field	Addressing mode
--------	----------------	-----------------

By comparing these 'Machine Code' column values for all the 'ADDA' and 'ADDB' addressing options, determine which bits of the machine code represents the information in the '**first byte**'.

Solution

Instruction	Machine code								
	Hex	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ADDA Immediate	8B ₁₆	1	0	0	0	1	0	1	1
ADDB Immediate	CB ₁₆	1	1	0	0	1	0	1	1
ADDA Direct	9B ₁₆	1	0	0	1	1	0	1	1
ADDB Direct	DB ₁₆	1	1	0	1	1	0	1	1
ADDA Indexed	AB ₁₆	1	0	1	0	1	0	1	1
ADDB Indexed	EB ₁₆	1	1	1	0	1	0	1	1
ADDA Extended	BB ₁₆	1		1	1	1	0	1	1
ADDB Extended	FB ₁₆	1	1	1	1	1	0	1	1

- By comparing the machine code bits for ADDA and ADDB with the same addressing mode it can be seen that they differ in the 'Bit 6' position.

The 'register field bit' is bit 6 of the machine code.

For Accumulator A, Bit 6 is [0].

For Accumulator B, Bit 6 is [1].

- By comparing the machine code bits for the ADDA instructions with the different addressing modes it can be seen that Bits 4 and 5 indicate the addressing mode.

i.e.

Bit 5	Bit 4	Address mode
0	0	Immediate
0	1	Direct
1	0	Indexed
1	1	Extended

- The remaining bits specify the OPCODE.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	Register field	Address mode field					
OPCODE				OPCODE			

- Boolean/arithmetic operation.** For all tables, this column provides a quick reference as to the operation performed by the instruction.

Most of the symbols used are specified in the legend given in table 11.1. Those omitted are as follows:

A – Contents of Accumulator A

B	–	Contents OF Accumulator B
M	–	Contents of the memory location (or data for immediate addressing) as addressed by the instruction
C	–	Carry bit
X	–	Contents of index register X
Y	–	Contents of index register Y
XH	–	Contents of the high byte of the index register
XL	–	Contents of the low byte of the index register
SP	–	Contents of the Stack Pointer



Note

The symbol '+' is intended as the 'arithmetic plus' for all instructions except the INCLUSIVE-OR and the **branch instructions** where it represents **logical 'OR'**.

Boolean expression in table 11.3. This column indicates **exactly** the conditions under which the conditional branch will occur. All expressions are logical or Boolean expressions.

e.g. The entry for 'BMI' instruction'. Note that it can be used with 'relative addressing only', the machine code is $2B_{16}$, it takes 3 machine cycles to execute, it is a 2 byte instruction and the condition that must exist for the branch to occur is the 'negative bit', N, must equal 1.

Some of the branch tests are quite complicated and these will be discussed later.

Warning: Ensure you check and understand the 'branch test' when selecting the appropriate branch instruction.

e.g. BMI (Branch if minus) and BLT (Branch if less than zero) may appear to be the same.

BMI will cause a branch if $N = 1$ i.e. if the MSB of the result of the last operation is [1].

BLT will cause a branch if $N = 1$ providing the **overflow is not set** i.e. $(N \oplus V) = 1$, it will also branch if $N = 0$ providing the overflow is set.

Boolean operation in Table 11.4. This indicates a quick reference to the instructions that can be performed on the condition code register.

CCR – Condition code register

Condition code register. This section of all the tables is split into 8 columns, one for each condition code bit, and indicates the affect the instruction execution has on each bit in the condition code register.

The possible conditions that can apply for the setting of the condition code bits are indicated for each. The symbols used are briefly explained in the ‘Legend’ below table 11.1 and are as follows:

- This symbol against an instruction indicates that the value of the condition code bit is unaffected, i.e. if set, it remains set, if reset it remains reset.

Note that none of the branch instructions affects the condition code bits.

↕ This symbol indicates that the condition code bit will be set or reset depending on the operation.

e.g. if the result of the operation is zero then the ‘Z-bit’ will be [1]. Obviously if the ‘Z-bit’ is [1] then the result is not negative and hence the ‘N-bit’ will be zero.

Details for setting the condition code bits in the status register have been outlined in previous modules.

- 1 This symbol indicates that it is ‘set’ as a result of the instruction without any test conditions applying.
- 0 This symbol indicates it is ‘Reset’ without any test conditions applying.
- ↓ This symbol indicates that it may be reset, but not set.

Points to note

- The M68HC11 can perform certain operations directly on the contents of a memory location without first bringing them into an accumulator.

e.g. ‘CLR’ instruction
 ‘DEC’ ”
 ‘COM’ ”
 ‘INC’ ”
 etc.

These instructions may be identified by the entries in the **Boolean expression** column. If an accumulator is used then ‘A’ or ‘B’ will appear. If the instruction applies directly to memory contents then ‘M’ appears by itself.

i.e. For ‘DEC’ the entry is ‘M – 1 → M’

- Certain instructions have restricted ‘addressing modes’.
 e.g. The COM, NEG, DEC, ROL, ROR, ASL, ASR, etc. instructions only allow ‘Indexed and Extended Addressing’ when referencing a location in memory.
- All ‘**compare**’ and ‘**test**’ instructions merely set the flags according to the result obtained from the operation performed, but do not store the result.

e.g. CMPA \$1F3B

This instruction subtracts the contents of location 1F3B₁₆ from the accumulator and sets all the condition codes according to the result. The actual contents of the accumulator remain unchanged.

A ‘**compare**’ or ‘**test**’ instruction is always followed by a conditional branch

i.e. BEQ – Branch if equal to zero

Or BNE – Branch if not equal to zero
etc.

- **Conditional Branch Instructions**

Refer to table 11.3

To give an indication of the significance of the branch conditions, it is best to consider them in pairs.

BCC/BCS instructions – These allow the program to test the status of the ‘carry bit’. The ‘carry bit’ may have been set as a result of an arithmetic operation or as a result of one of the Shift/Rotate instructions.

BEQ/BNE instructions – The condition for branching is based exclusively on the ‘Z-bit’ and tests for the ‘Result = 0’.

BVS/BVC instructions – These test the **overflow bit** and provide a means of ensuring that the binary data does not exceed the limits of 8-bit 2’s complement arithmetic.

BPL/BMI instructions – These test the MSB of the result, i.e. the ‘sign bit’, and should be used only with ‘signed numbers’, i.e. 2’s complement numbers. Note that they do not test for the possibility of an **overflow**.

BLT/BGE instructions – These are very similar to the BPL/BMI instructions but in addition to checking the ‘sign-bit’ they also check for a 2’s complement overflow.

BGT/BLE instructions – These are very similar in test to the BLT/BGE instructions but with the added test to put the decision point above the zero value instead of below it.

BHI/BLS instructions – These are special instructions in that they apply for **unsigned numbers**, i.e. the MSB does not indicate the ‘sign’ and 2’s complement numbers are not being used.

Example 5

Let Accumulator A contain $E7_{16}$ and memory location 0001_{16} contains $7E_{16}$.

- What instructions would be used to cause a program to **branch** if the contents of the accumulator are less than the contents of the memory location. Assume a ‘signed number’ system, i.e. the numbers are 2’s complement values.
- Repeat a. for an ‘unsigned’ number system.

Solution

- To determine if the number in Accumulator A is less than that in memory location 0001_{16} then first a ‘compare’ instruction is required.

i.e. CMPA \$ 01 (direct addressing)

This operation results in the condition codes being set according to the following calculation:

i.e. $(ACCA) - (\text{location } 0001_{16})$

or $(E7 - 7E)$ which is equivalent to $(-25_{10} - 126_{10})$ for a ‘signed 2’s complement system’.

A close examination of table 11.3 indicates possible suitable branch instructions are:

BMI – Branch if minus (tests for $N = 1$)

BLT – Branch if less than zero (test is $N \oplus V = 1$)

as it is ‘signed numbers’ we are considering, these are the only appropriate choices.

To select the appropriate instruction consider the actual operation performed by the ALU.

$E7_{16}$ is 1 1 1 0 0 1 1 1

$7E_{16}$ is 0 1 1 1 1 1 1 0

2’s complement of $7E_{16}$ is 1 0 0 0 0 0 1 0

$E7_{16} - 7E_{16}$ results in

	1	1	1	0	0	1	1	1	E7
+	1	0	0	0	0	0	1	0	2’s complement of 7E
$\times \rightarrow$	0	1	1	0	1	0	0	1	

discard
carry

From this result the condition code values would be as follows:

- N = 0 – MSB = 0
- Z = 0 – result is not zero
- V = 1 – overflow has occurred
- i.e. the 2 numbers added are both negative but the result is positive.
- C = 0 – Since the CPU subtracts using the 2’s complement method, we need to invert the carry result to get the correct status.



Note

In this example the carry generated by performing the subtraction using 2’s complement arithmetic is discarded and is **not the borrow** resulting from the subtraction. This can be identified by performing normal subtraction as follows:

$E7_{16} - 7E_{16}$ is

	1	1	1	0	0	1	1	1	= E7
–	0	1	1	1	1	1	1	0	= 7E
C = 0	0	1	1	0	1	0	0	1	Normal subtraction

i.e. there is **no borrow** therefore the ‘C-bit’ of the condition code register is [0].

The result of a branch should indicate the $E7_{16}$ (-25_{10}) is less than $7E_{16}$ ($+126_{10}$).

The ‘BMI instruction checks for $N = 1$. But $N = 0$ therefore this instruction would not branch and would give the incorrect result.

The BLT instruction performs the test $(N \oplus V)$ which is [1] and would cause the correct branching operation.

The appropriate instructions are then

CMPA \$ 01
BLT (offset distance)

- b. For an unsigned number system $E7_{16}$ is 231_{10} and $7E_{16}$ is 126_{10} and as such no branch should occur.

The ‘compare instruction’ is still required and the condition code bits will be set identical to that in (a). From the summary given in Conditional Branch Instructions above the only instructions that can be used for unsigned numbers are

BLS or BHI

BLS results in a branch if the contents of 0001_{16} are ‘same or less than’ the contents of Accumulator A. This example requires that ‘it be less than’ that of Accumulator A.

This can be provided as follows:

CMPA \$ 01
BEQ 02
BLS (offset)

If the numbers are the same then the ‘BEQ’ instruction will jump forward over the BLS instruction. (N.B. Firstly branch instructions do not alter the condition codes and secondly, the branch ‘02’ is required since the BLS instruction is 2 bytes long.) This ensures that the BLS instruction will only branch if the contents of 0001_{16} are less than Accumulator A.

- **Special instructions** in table 11.3, include instructions which do not fall under any general category.
 - NOP or ‘No Operation’ instruction. This is a single byte instruction which ‘does nothing’ except increment the program counter to the next instruction. It is very useful since the inclusion of ‘NOP’ instructions in a program allow extra instructions to be added without altering the locations of all the program. Similarly if an instruction is deleted during the ‘debugging phase’ it can be replaced with ‘NOP’s’.
 - WAI This instruction is used when the timing of an input/output is controlled by an external interrupt. Its use will be demonstrated in a later module.
 - JSR Allows the program to **‘Jump to Subroutine’** and note that ‘indexed’ and ‘extended’ addressing can be used to specify the start address of the sub routine.

e.g. JSR \$ 101F

This will cause the program to jump to the subroutine whose first instruction is located at memory location $101F_{16}$.

(See the illustration following table 11.3.)

On a ‘JSR’ the value on the ‘program counter’ (which points to the next instruction address in the main program) is pushed on the stack and the address specified by the JSR instruction (e.g. $101F_{16}$) is put in the PC.

RTS The end of the subroutine is indicated by an RTS instruction and causes the original value of the PC to be returned from the **stack**.

(See the illustration following table 11.3.)

BSR A branch to subroutine uses relative addressing to specify the subroutine address. This allows the subroutine to start at a location which must be within the range +127 to – 128 locations of the current value of the PC.

SWI In summary,

- When the SWI is executed the PC, XREG, Acc A, Acc B and Condition Code register contents are saved on the **stack**. (N.B. The PC contains the address of the instruction following the SWI.)
- The new PC value is taken from memory locations $FFFA_{16}$ and $FFFB_{16}$. i.e. These supply the address of the ‘service routine for the Software Interrupt’.
- The Service Routine is then run.

The effect of the SWI is identical to an ‘external hardware interrupt’ which alerts the processor by activating an ‘Interrupt Request’ line.

RTI The final instruction of an Interrupt Service Routine must be this instruction. When executed it causes all the register contents to be returned from the **stack**. This returns the original value of the PC which is the address of the next instruction and allows the main program to continue.

(See the illustration following table 11.3.)



Activity 11.1

All questions refer to the MC68HC11 series microprocessors.

1. What is meant by the shorthand notation: $A + B > A$?
2. How is the C flag affected by the 'add' and 'add with carry' instructions?
3. Is the C flag changed when the AND instruction is executed?
4. Explain the difference between the NEG instruction and the COM instruction.
5. Explain the difference between the ANDA instruction and the BITA instruction.
6. The decimal adjust instruction is associated with which accumulator?
7. When the RORA instruction is executed the LSB of Accumulator A is shifted into the _____ register.
8. List eleven operations that can be performed directly on an operand in memory without first loading it into one of the MPU registers.
9. Explain the difference between the SUBB instruction and the CMPB instruction.
10. List the four types of logic operations that the '68HC11 MPU can perform.
11. When the LDX instruction is executed, from where is the index register loaded?
12. List four conditional branch instructions that are commonly used after a compare or subtract instruction to compare two's complement numbers.
13. Explain the difference between the BGT and BHI instructions.
14. Which instruction is often used to fill in a hole left in a program after an unwanted byte is removed?
15. Which of the condition codes can be individually set or cleared?
16. What is the difference between an unconditional branch instruction and a conditional branch instruction?
17. What condition is tested by the branch if minus (BMI) instruction?
18. When is the N flag set?
19. When is the Z flag set?
20. During an add operation, the C flag is set. What does this represent?
21. During a subtract operation, the C flag is set. What does this indicate?
22. Often, when two positive 2's complement numbers are added, the sign bit of the answer will indicate a negative sum. This 'error' can be spotted by checking which flag?
23. Under, what condition will the BEQ instruction cause a branch to occur?
24. Under what condition will the BPL instruction cause a branch to occur?
25. When subtracting unsigned binary numbers, which flag indicates that the difference is a negative number?

26. How is the ADC instruction different from the ADD instruction?
27. How is the SBC instruction different from the SUB instruction?
28. A primary use of the ADC and SBC instructions is in _____ arithmetic.
29. The accumulator contains the number 7_{10} . If two ASLA instructions are executed, what number will be in the accumulator?
30. The BRA instruction will cause a branch to occur:
 - a. Anytime that it is executed.
 - b. Only if the Z flag is set.
 - c. Only if the N flag is set.
 - d. Only if the C flag is set.
31. The 'operand address field' that follows the opcode of an unconditional branch instruction is:
 - a. The address of the operand.
 - b. The address of the next opcode to be executed.
 - c. Added to the program count to form the address of the next opcode to be executed.
 - d. Added to the program count to form the address of the operand that is to be tested to see if a branch operation is required.
32. The opcode for an unconditional branch instruction is at address AF_{16} . The relative address is $0F_{16}$. From what address will the next opcode be fetched?
 - a. $A0_{16}$
 - b. $C0_{16}$
 - c. BE_{16}
 - d. $B1_{16}$
33. The opcode for an unconditional branch instruction is at address 30_{16} . The relative address is EF_{16} . From what address will the next opcode be fetched?
 - a. 21_{16}
 - b. EF_{16}
 - c. 32_{16}
 - d. 19_{16}
34. The carry register:
 - a. Acts like the ninth bit of the accumulator.
 - b. Is set when a 'borrow' for bit 7 of the accumulator occurs.
 - c. Is set when a carry from bit 7 occurs.
 - d. All the above.

- 35 The numbers 0101 10002, and 0110 00112, are added using the ADD instruction.

Immediately after the ADD instruction is executed, the condition code registers will indicate the following:

- a. C = 1, N = 1, V = 1, Z = 0
 - b. C = 0, N = 1, V = 1, Z = 0
 - c. C = 0, N = 1, V = 0, Z = 0
 - d. C = 0, N = 0, V = 1, Z = 1
36. When you are adding multiple-precision binary numbers, all bytes except the least significant ones must be:
- a. Added using the ADD instruction.
 - b. Added using the DAA instruction.
 - c. Added using the ADC instruction.
 - d. Decimal adjusted before addition takes place.

Solutions are provided at the end of the module.

11.2 Program formatting

We now have all the information necessary to write programs in machine language to solve any problem. It should be noted, however, that all programs must be written in a strict format, which all other programmers can understand. Even a program written by you a week ago would now be difficult to recall if you don't document it in a standard format. This fact cannot be emphasised enough and be warned – 'programs submitted in some other format will not be accepted as assignments in this course.'

The format is simple and logical and it uses a fixed set of columns known as 'fields'. The technique used to solve problems often differs from programmer to programmer however in this course we will again follow a 'standard technique', which will at least guarantee you success in solving problems.

11.2.1 Problem solving techniques

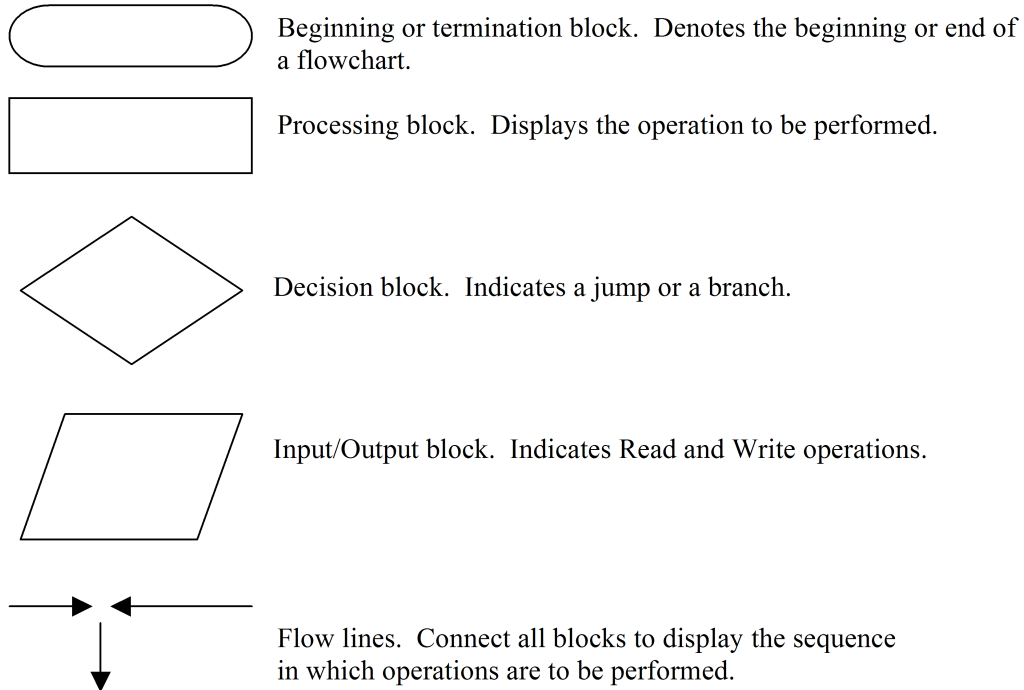
There are 4 basic steps in solving a problem. These are:

Step 1 – a flowchart

The flowchart is a block diagram used to graphically illustrate the technique used by the programmer to solve the problem. The words used in each block (or box) are standard English language and are **never** abbreviated computer jargon for a particular computer. The flowchart should be an explanation only of how the problem can be solved on any computer.

Flowchart symbols:

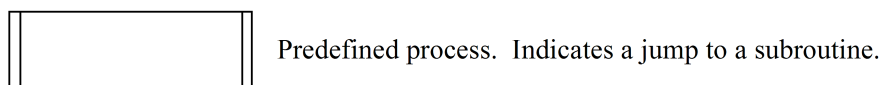
There are five basic symbols used to draw flowcharts. Each corresponds to one of the five basic functions of a computer, as follows:



Flow lines used to indicate loops must be labeled as in the assembly language section of the program so that each loop can be easily identified.

Flow lines used to indicate loops and jumps must enter the main program flow above or below a process block.

Flow lines exiting from a decision block must be labeled with a (Y)es or (N)o to indicate the direction of flow based on the outcome of the decision.



A subroutine can be considered as a separate predefined process. When the program reaches this block, the main program is halted and flow passes to the subroutine. At the end of the subroutine flow returns to this block and the main program continues. Subroutines must be shown as a separate flowchart with a Start block at the beginning and a Return block at the end.

Step 2 – a memory map

The memory map is a diagram showing the memory addresses used for data, I/O devices, the program itself and any unused memory. It will vary from processor to processor according to the available memory addresses.

The next two steps describe the program itself, which is written in two sections as shown.

Step 3 – the assembly language section

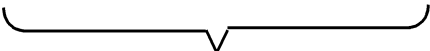
This section contains everything (labels, mnemonics, operands and comments) except the actual codes used by the processor. We will see later that by describing this section to an ‘assembler program’ on another computer we can automatically generate machine codes and so avoid the process described in Step 4.

Step 4 – the machine coded section


This section of the program lists the actual addresses used by a particular processor together with the machine codes relevant for that processor. It is the only part of the whole programming process, which the computer itself understands.

As mentioned previously when writing and coding programs it is essential to present them in a format, which is consistent with that given by most assemblers. All programs you write should follow this format:

Memory address	Machine code	Label field	Mnemonics field	Operand field	Comment field



Machine code section



Assembly language section

Label field – It is convenient to give labels to certain instructions in the program. This makes it easy to follow.

e.g. START and END (or STOP)

Its major use however is to indicate **branch destinations**.

e.g. BPL LOOP

indicates the branch destination, i.e. the instruction to be executed immediately after a ‘**branch**’, is that against the label LOOP, i.e. ADDA instruction.

Mnemonics field – Contains the mnemonic of the instructions only.

Operand field – Contain the ‘address mode symbol’ and the ‘operand’.

Note: Use labels and names for operands rather than hexadecimal address.

Comment field – A brief comment on the function of the instruction.

Programming example 1

The foregoing process is best illustrated by an example. We will reprogram the previous example given in section 8.5.

Write a complete program using the stored program concept to perform the following numerical exercise: $9 - 5 + 2$ on a Motorola '68HC11 series processor. The result is to be stored in a memory location labelled ANS.

Step 1: Flowchart

Step 2: Memory map

Step 3: Assembly language program

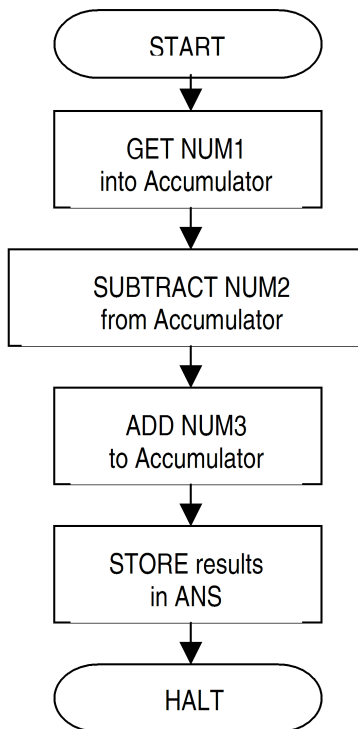
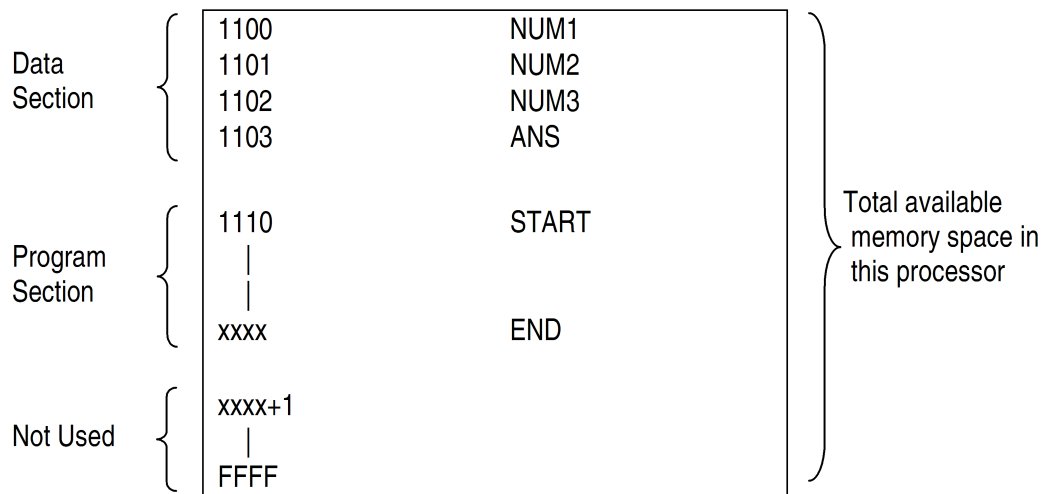
Step 4: Machine code program

Notes:

1. This program must be designed to operate on the M68HC11 series processor. Its memory space (allocated by the manufacturer) available to the programmer is 1100 to B600.
2. We will also illustrate the use of **labels**, which were mentioned in section 11.1.2. In this problem we will allocate labels to the data in this problem. For instance:

Data	Label
9	NUM1
5	NUM2
2	NUM3
Result	ANS

Remember labels are not recognized by the computer, however, they are used extensively in designing programs as they assist in layout and understanding of the programming techniques used.

Step 1 – Flowchart**Step 2 – Memory map**

Steps 3 and 4 – Program sections

Note the use of extended addressing modes because the memory address locations are 16 bits in length.

Machine Code Section (Step 4)		Assembly Language Section (Step 3)			
Address	Operation code	Label field	Mnemonics field	Operand field	Comments field
1100		NUM1		9	Number 1 =9
1101		NUM2		5	Number 2 = 5
1102		NUM3		2	Number 3 = 2
1103		ANS			Location reserved for answer
1110	B6	START	LDAA	NUM1	Loads Accumulator A with Number 1 from location 1100
1111	11				
1112	00				
1113	B0		SUBA	NUM2	Subtracts Number 2 from the Accumulator
1114	11				
1115	01				
1116	BB		ADDA	NUM3	Adds Number 3 to the Accumulator
1117	11				
1118	02				
1119	B7		STAA	ANS	Stores the result in Accumulator A into location 1103.
111A	11				
111B	03				
111C	7E	HALT	JMP	HALT	Halts the process at this point
111D	11				Not an ideal way to end a program, but acceptable in this course.
111E	1C				

Note: This program is not designed to run on the 68HC11 microcontroller.

This program would be started at location \$1110 after it has been typed into the CPU RAM.

Let us now consider another example where the assembly language program has been developed by someone else and we are to describe its operation and code it for a '68HC11 series processor with memory addresses as given.

Programming example 2

Consider the following Assembly Language Program:

Label field	Mnemonic field	Operand field	Comment field
START	LDX	#\$0004	Set the XREG = 4
	CLRA		Clear Accumulator A
LOOP	ADDA	\$00,X	Add (M) into Accumulator A
			Where $M = (XREG) + 00$
	DEX		Decrement XREG for next value
	BPL	LOOP	For $M > 0000$ go back to loop
END	JMP	END	Stop the program

Describe the function of the program and code it into machine code with the program starting at location 0010_{16} .

Solution

Before solving the problem note the function of each of the columns.

The function of this program is to sum the contents of the memory locations 0000_6 to 0004_{16} , i.e. a table, leaving the answer in the accumulator. Note how indexed addressing is used to point to the next entry in the table each 'pass through the loop'.

Let the contents of 0000_{16} to 0004_{16} be as follows:

Location	Contents
0000	01
0001	02
0002	03
0003	04
0004	05

The register contents as the program is run would then be as follows:

- After the program has just executed the CLRA instruction
Acc A = 0 XREG = 0004
- When the program is about to execute BPL LOOP for the first time Acc A = 05_{16} (i.e. contents of 0004_{16} have been added to it)
XREG = 0003_{16}
As the 'DEX' instruction set the 'condition codes' and the result is 0003_{16} then the BPL instruction will cause the program to branch back to LOOP and repeat at the 'ADDA' instruction.
- When the program is about to execute BPL LOOP for the second time

Acc A = 09_{16}
 XREG = 0002_{16}

- When program is about to execute BPL LOOP for the third time

Acc A = $0C_{16}$ (i.e. $5_{10} + 4_{10} + 3_{10} = 12_{10} = C_{16}$)
 XREG = 0001_{16}

- After the fourth time

Acc A = $0E_{16}$
 XREG = 0000_{16}

Note this is still positive (i.e. N bit = 0) so it will branch one more time.

- After the fifth time

Acc A = $0F_{16}$ (i.e. 15_{10})
 XREG = $FFFF_{16}$ (i.e. $0000_{16}-1$)

which means the N bit = 1 and the **branch** will not occur therefore the next instruction will be the JMP END instruction which is an endless loop that effectively stops the program. Resetting the processor with terminate the program properly.

The fully coded program is as follows:

Address	Operation code	Label field	Mnemonic field	Operand field	Comment field
0010	CE	START	LDX	#\$0004	Set the XREG = 4
0011	00				
0012	04				
0013	4F		CLRA		Clear Acc A
0014	AB	LOOP	ADDA	\$00,X	Add (M) into Acc A Where M = (XREG) + 00
0015	00				
0016	09		DEX		Decrement XREG for next M
0017	2A		BPL	LOOP	For M > 0000 go back to loop
0018	FB				
0019	7E	END	JMP	END	STOP
001A	00				
001B	19				

The machine language program is written by selecting the appropriate machine code from the instruction set.

e.g. LDX #\$0004

is coded into three bytes as follows:

Byte 1	CE	LDX (Immediate)
Byte 2	00	} Operand = 0004_{16}
Byte 3	04	

N.B. In some references the assembly code for this instruction is written as

```
LDX    #$04
```

This is valid, however, care should be exercised to check that it is in fact a 3-byte instruction. This can be found in the ‘bytes column’ of the instruction details.

Coding for the remainder of the instructions should be straight forward except possibly the ‘BPL LOOP’ instruction.

The machine code for ‘BPL LOOP’ is

```
Byte 1    2A
Byte 2    OFFSET
```

The value of the OFFSET is that value which when added to the current value of the PC will cause the next instruction to be fetched from location 0014_{16} (i.e. the ‘ADDA’ instruction). This is **relative addressing**.

When the ‘BPL LOOP’ instruction is executed the PC will already have been incremented to point at the next instruction.

i.e. $PC = 0019_{16}$

Therefore, the address calculation will be as follows:

```
001916 + OFFSET = 001416
or    OFFSET = -5
```

This OFFSET is held as a 2’s complement value, which for -5_{16} is FB_{16} .



Activity 11.2

Label field	Mnemonic field	Operand field	Comment field
START	CLRA		Clear Accumulator A
LOOP	TST	\$0020	Test the Multiplier
	BEQ	STOP	If it is zero STOP
	DEC	\$0020	Otherwise decrement the multiplier
	ADDA	\$21	Add Multiplicand to the Product
	BRA	LOOP	Repeat the loop
STOP	JMP	STOP	

Figure 11.1:

This program performs a multiplication by repeated addition. The memory locations reserved for variables are as follows:

Multiplier = Address 0020_{16}

Multiplicand = Address 0021_{16}

Let the Multiplier = 05_{16} , the Multiplicand = 04_{16} and the program when coded into machine code is to start at Address 0010_{16} .

In relation to the program in figure 11.1 answer the following questions.

1. Code the program into machine code with starting address 0010_{16} .
2. What addressing mode does the TST instruction use?
 - a. Immediate
 - b. Direct
 - c. Extended
 - d. Indexed
3. The BEQ instruction checks to see if the TST instruction set the:
 - a. Z flag
 - b. C flag
 - c. H flag
 - d. V flag
4. The DEC instruction decrements the number in:
 - a. Accumulator A
 - b. Memory location 0020
 - c. Accumulator B
 - d. The index register

5. Which instruction is executed immediately after the BRA instruction?
 - a. WAI
 - b. BEQ
 - c. CLRA
 - d. TST
6. With the values given for the multiplier and multiplicand, how many times will the main program loop be repeated?
 - a. Four times
 - b. Five times
 - c. Twenty times
 - d. Twice
7. After the program has been executed, memory location \$0020 will contain:
 - a. 05_{16}
 - b. 04_{16}
 - c. 20_{16}
 - d. 00_{16}
8. After the program has been executed, the product will appear in:
 - a. Memory location \$0020
 - b. Memory location \$0021
 - c. Accumulator A
 - d. Accumulator B

Label field	Mnemonic field	Operand field	Comment
START	LDX	#\$0005	
LOOP	LDAA	\$20,X	
	ADDA	\$30,X	
	STAA	\$40,X	
	INX		
	CPX	#\$0015	
	BNE	LOOP	
STOP	JMP	STOP	

Figure 11.2:

For questions 9–14 refer to the program in figure 11.2.

9. Analyse the program and determine what it does.
10. Code the program into machine code with starting address 0010_{16} and fill in the appropriate comments in the 'comment field'.

11. On the first pass through the main program loop, the 'LDAA \$20,X' instruction takes into operand from memory
 - a. \$0005
 - b. \$0020
 - c. \$0025
 - d. \$0014
12. On the first pass, the 'ADDA \$30,X' adds the contents of what memory location to Accumulator A?
 - a. \$0005
 - b. \$0030
 - c. \$0035
 - d. \$0016
13. On the second pass through the program loop, the contents of memory location:
 - a. \$0021 are added to the contents of \$0031 and the result is stored in \$0041.
 - b. \$0026 are added to the contents of \$0036 and the result is stored in \$0046.
 - c. \$0025 are added to the contents of \$0035 and the result is stored in \$0045.
 - d. \$0020 are added to the contents of \$0030 and the result is stored in \$0040.
14. How many times is the main program loop repeated?
 - a. 10_{16} times
 - b. 05_{16} times
 - c. 30_{16} times
 - d. 15_{16} times



Activity 11.1 solutions

1. Add the contents of Accumulator A to the contents of Accumulator B; transfer the result to Accumulator A.
2. The C flag is set if a carry occurs; it is cleared otherwise.
3. No, the C flag is unaffected by the AND instruction.
4. The COM instruction replaces the operand with its 1's complement. The NEG instruction replaces the operand with its 2's complement.
5. With the ANDA instruction, the result of the AND operation is placed in Accumulator A. With the BITA instruction, the condition code registers are set according to the result but the result is not retained.
6. The decimal adjust instruction works only with the Accumulator A.
7. Carry (C).
8. A byte in memory can be: cleared, incremented, decremented, complemented, negated, rotated left, rotated right, shifted left arithmetically, shifted right arithmetically, shifted right logically, and tested.
9. With the SUBB instruction, a difference is produced and placed in Accumulator B. With CMPB, the flags are set as if a difference were produced, but the difference is not retained.
10. Complement, AND, inclusive OR, and exclusive OR.
11. The upper half of the index register is loaded from the specified memory location; the lower half from the byte following the specified memory location.
12. BGE, BGT, BLE, BLT.
13. BGT is used to test the result of subtracting two's complement numbers. BHI is used to test the result of subtracting unsigned numbers.
14. NOP
15. C, I and V
16. An unconditional branch instruction always causes a branch operation to occur. On the other hand, the conditional branch instruction implements a branch operation only if some specified condition is met.
17. The BMI instruction tests the Negative (N) bit of the Condition Code register to see if it is set.
18. Generally speaking, the N flag is set if the previous instruction left a 1 in the MSB of the accumulator.
19. Generally, the Z flag is set if the previous instruction left all zeros in the accumulator.
20. During an add operation, the carry bit is set if there is a carry from bit 7 of the accumulator.
21. During a subtract operation, the carry bit is set if bit 7 had to 'borrow' a bit to complete the subtraction.
22. This condition results from a two's complement overflow. Thus, the V flag will be set if this condition occurs.

23. The BEQ instruction causes a branch to occur only if the Z bit is set.
24. The BPL instruction causes a branch to occur only if the N bit is clear.
25. The carry flag.
26. When the ADC instruction is executed, an additional 1 is added to the sum if the carry flag is set.
27. When the SBC instruction is executed, an additional 1 is subtracted from the difference if the carry flag is set.
28. Multiple-precision.
29. The first ASLA instruction multiplies the number by two, giving 14_{10} . The second ASLA doubles this number, giving 28_{10} .
30. a. – The BRA instruction causes a branch anytime that it is executed.
31. c. – The address that follows a branch opcode is added to the program count to form the address of the next opcode to be executed.
32. b. – During the execution of the branch instruction, the program counter is advanced twice to $B1_{16}$. Thus, when the relative address ($0F_{16}$) is added, the new address becomes $C0_{16}$.
33. a. – The execution of the branch instruction increments the program counter to 32_{16} . When the relative address (EF_{16}) is added, the new address becomes

		0011		0010 ₂		32_{16}
		1110		1111 ₂		EF_{16}
	1	0010		0001 ₂	1	21_{16}
Ignore Carry	↑		Ignore Carry	↑		

34. d. – The carry register performs all the functions listed.
35. b. – The result of this addition is

0101	1000 ₂
0110	0011 ₂
-----	-----
1011	1011 ₂

The C flag is cleared to 0 because there was no carry from bit 7. The N flag is set to 1 because bit 7 is 1. The V flag is set to 1 because, if you consider the numbers to be signed binary, the addition of two positive numbers resulted in a negative answer. The Z flag is cleared to 0 because the result is not zero.

36. c. – The ADC instruction must be used when you are adding multiple-precision numbers.



Activity 11.2 solutions

Address	Operation code	Label field	Mnemonic field	Operand field	Comment field
0010	4F	START	CLRA		Clear Accumulator A
0011	7D	LOOP	TST	\$0020	Test the Multiplier
0012	00				
0013	20				
0014	27		BEQ	STOP	If it is zero STOP
0015	07				
0016	7A		DEC	\$0020	Otherwise decrement the multiplier
0017	00				
0018	20				
0019	9B		ADDA	\$21	Add the Multiplicand to the product
001A	21				
001B	20		BRA	LOOP	Repeat the loop
001C	F4				
001D	7E	STOP	JMP	STOP	STOP
001E	00				
001E	1D				

2. c. – Extended.
3. a. – The BEQ instruction tests the Z flag.
4. b. – The DEC instruction decrements the number in memory location \$0020.
5. d. – The relative address (F4) directs the program back to the TST instruction.
6. b. – The multiplier (05) is decremented on each pass until it reaches 00. Thus, the loop will be repeated five times.
7. d. – The multiplier is reduced to 00 as the program is executed.
8. c. – The product appears in Accumulator A.
9. This program adds together the corresponding values of two equal sized tables and stores a result in a third table. The tables are stored as follows:

Table 1 is in locations 0025₁₆ to 0034₁₆

Table 2 is in locations 0035₁₆ to 0044₁₆

Table 3 (answers) is in locations 0045₁₆ to 0054₁₆

LDAA \$20,X takes a value from table 1 into ACCA

ADDA \$30,X adds the corresponding value from table 2 into Acc A

STAA \$40,X stores this value in table 3

This procedure is then repeated 16₁₀ or 10₁₆ times.

10.

Address	Operation code	Label field	Mnemonic field	Operand field	Comment field	
0010	CE	START	LDX	#\$0005	Set Index Register to \$0005	
0011	00	LOOP	LDAA	\$20,X	Load ACCA from Address M where M = \$20 + XREG Add the contents of location (\$30 + XREG) into Acc A	
0012	05					
0013	A6					
0014	20		ADDA	\$30,X		
0015	AB					
0016	30		STAA	\$40,X	Store Acc A into the location (\$40 + XREG) increment XREG for next value Compare the index register to \$0015	
0017	A7					
0018	40					
0019	08		INX	#\$0015		
001A	8C		CPX			
001B	00	END	BNE	LOOP	If not equal to \$0015. Go back to loop Yes – STOP	
001C	15					
001D	26					
001E	F4		JMP	END		
001F	7E					
0020	00					
0021	1F					

11. c. – The Offset address (\$20) is added to the number in the index register (\$0005) to form an operand address of \$0025.
12. c. – The contents of location \$0035 are added to Accumulator A on the first pass through the loop.
13. b. – On the second pass, the contents of location \$0026 are added to the contents of location \$0036. The result is stored at \$0046.
14. a. – The index register starts at \$0005 and is incremented to \$0015. Therefore, the loop is repeated 10_{16} times.