LA TROBE UNIVERSITY
DEPARTMENT OF ELECTRONIC ENGINEERING

ELECTRONICS-2

Microprocessors: Lecture 3 & 4
Introduction to the 68HC11
Memory Address Map, Registers
and 68HC11 Instruction Set
Addressing Modes
Conditional Jumps

Contents:

16 Bit Memory Address Map

Address:	Value (Binary)	(Hex)
FFFF	0 0 0 0 0 0 0 0	00
FFFE	1 1 1 0 0 0 0 0	E0
FFFD	1 1 1 1 1 1 0 1	FD
FFFC	0 0 0 0 0 0 0 0	00
FFFB	1 1 1 1 1 0 1 0	FA
FFFA	0 0 0 0 0 0 0 0	00
FFF9	1 1 1 1 0 1 1 1	F7
FFF8	0 0 0 0 0 0 0 0	00
⋮		
0007	0 0 0 0 0 1 1 1	07
0006	0 0 0 0 0 1 1 0	06
0005	0 0 0 0 0 1 0 1	05
0004	0 0 0 0 0 1 0 0	04
0003	0 0 0 0 0 0 1 1	03
0002	0 0 0 0 0 0 1 0	02
0001	0 0 0 0 0 0 0 1	01
0000	0 0 0 0 0 0 0 0	00

Figure 1 -Memory Address map. 2^{16} Addresses

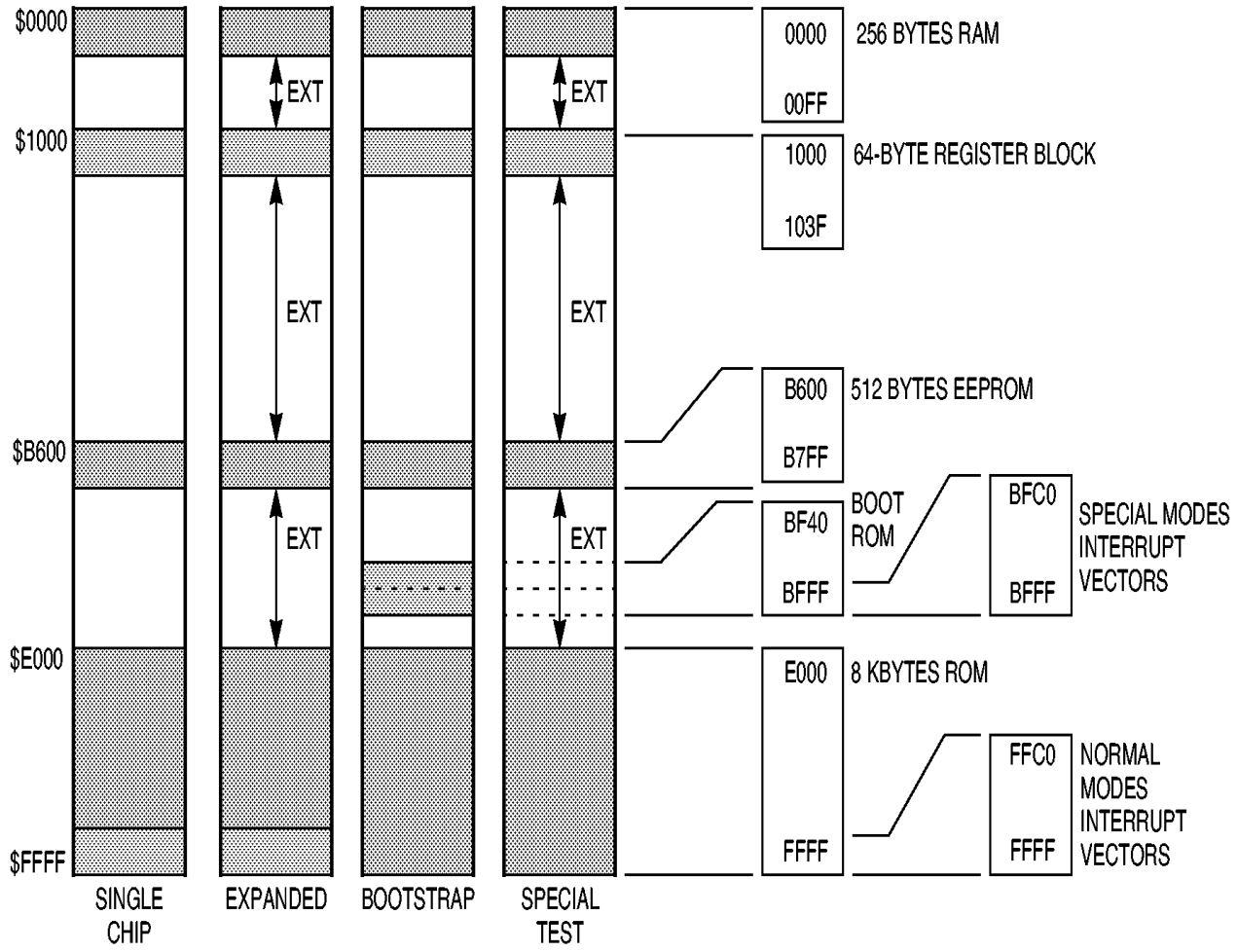


Figure 2- 68HC11 Memory Map.

Which mode this chip starts in is determined by the
MODA & MODB pins.
We will limit the majority of our study to Expanded mode as this is the mode used by HCCOM.

The 68HC11 Registers / Accumulators

Example Register Usage:

```
        LDAA #$12       ; Load Accumulator A with the number $12 (=18 decimal)
        LDAB #$34       ; Load Accumulator B with the number $34 (=52 decimal)
        ABA             ; Add accumulator B to A
        STAA RESULT     ; Store Accumulator A in an address labeled RESULT
        SWI             ; Stop the program execution - Exit to BUFFALO Monitor
                        ; Result should contain $46 (=70 decimal)
RESULT  RMB 0           ; RMB = Reserve Memory Byte
```

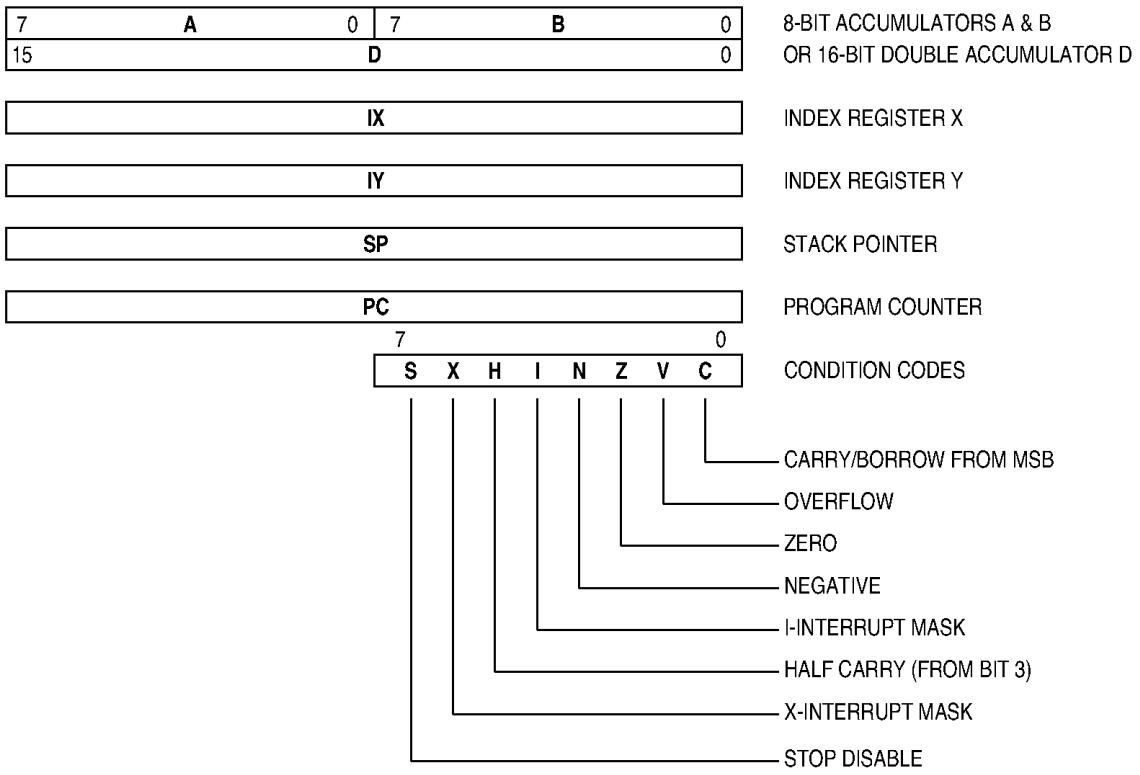


Figure 1-2 M68HC11 Programmer's Model

Figure 3 - 68HC11 Register Set

Pointer & Index Registers
SP - Stack Pointer
IX - Index X
IY - Index Y
PC Program Counter

Stack

The stack is a section of memory set aside for, temporary storage during the execution of a program.

The stack "grows downwards" through memory as it fills (PUSH), and recedes up through memory as it shrinks (PULL). i.e. When a value is **push**ed onto the stack, value is inserted in memory at the location pointed to by the stack pointer (SP), then the stack pointer automatically decrements.

Stack Pointer

The stack pointer is a 16-bit register that points to the next free location on the stack.
When a value is **pul**led from the stack, the stack pointer (SP) automatically increments, and the value at that location is then copied into the register.

Stack Operation : PUSH

When a value located in an accumulator or register is pushed onto the stack, it is copied from the accumulator or register and stored to the location pointed to by the stack pointer.
The stack pointer then decrements to point to the next free location.

Examples: psha * push the accumulator A
pshb * push the accumulator B
pshy * push the index register Y

Stack Operation : PULL

When a value is pulled from the stack into an accumulator or register, the stack pointer increments to point to last used location on the stack. The value at that location is then copied into the accumulator or register. (equivalent instruction on 80x86 - **POP**). Pull is the reverse operation to push.

Stack Pointer

Not normally altered directly except once on power on reset (eg: LDS #STACK ;initialize start of stack) The stack is used to hold parameters passed to subroutines and to save values temporarily

```
>asm 2100
2100 PSHA           >
2101 PSHB           >
2102 PSHX           >
2103 PSHY           >
2105 PULX           >
2106 PULY           >
2108 PULA           >
2109 PULB           >
210A RTS            >
210B TEST           >♥
```

Example: Using the Stack Pointer hold data

```
; Assemble code at 2100 Hex.
; Push Accumulator A
; Push Accumulator B
; Push Register X
; Push Register Y
; Pull Register  X        (= pop)
; Pull Pegister  Y
; Pull Accumulator A
; Pul Accumulator B
; Return from subroutine (PC is on stacktop)
```

Press Control-C to exit the assembler

Calling the subroutine at $2100 saves the calling program's return address, A, B, X then Y. Normally the reverse order is used to restore the original values to the original registers, however, for this illustration we have swapped the order of X with Y and B with A. The result of swapping registers can be seen using the BUFFALO command **rm** (Register Modify) as follows:

```
>rm p
P-E328 Y-BAAA X-F000 A-00 B-FF C-D0 S-0041
P-E328 2100

>call 2100

P-2100 Y-F000 X-BAAA A-FF B-00 C-D0 S-0041
>
```

BUFFALO

BUFFALO is the monitor program that controls HC-COM configuration, booting and provides some user interface functions like print a character, print a number, read a character, assemble instructions....

BUFFALO'S INTERACTIVE COMMANDS:

ASM [<addr>] Line assember/disassembler
[/,=] Same addr,
[^,-] Prev addr,
[+,CTLJ] Next addr
[CR] Next opcode,
[CTLA,.] Quit
BF <addr1> <addr2> [<data>] Block fill memory
BR [-][<addr>] Set up bkpt table
BULK Erase EEPROM,
BULKALL Erase EEPROM and CONFIG
CALL [<addr>] Call subroutine
GO [<addr>] Execute code at addr,
PROCEED Continue execution
EEMOD [<addr> [<addr>]] Modify EEPROM range
LOAD, VERIFY [T] <host dwnld command>
Load or verify S-records
MD [<addr1> [<addr2>]] Memory dump
MM [<addr>] or [<addr>]/ Memory Modify
[/,=] Same addr,
[^,-,CTLH] Prev addr,
[+,CTLJ,SPACE] Next addr
<addr>O Compute offset,
[CR] Quit
MOVE <s1> <s2> [<d>] Block move
OFFSET [-]<arg> Offset for download
RM [P,Y,X,A,B,C,S] Register modify
STOPAT <addr> Trace until addr
T [<n>] Trace n instructions
TM Transparent mode (CTLA = exit, CTLB = send brk)
[CTLW] Wait,
[CTLX,DEL] Abort
[CR] Repeat last cmd

An MSDOS cross-assembler for HCCOM is called AS11.

AS11 - 68HC11 Assembler.

The syntax between AS11 and Buffalo varies slightly. We will mostlty work with the BUFFALO assembler, but code snippets that are provided will use the AS11 syntax. The AS11 assembler accepts assembly language files and produces loadable machine code as output.

AS11 Assembler Data Types

Expressions used by assembler to allocate initialised space (similar to constants in C)

FCB - Form Constant Byte - Creates initialised space for byte sized objects

FCC - Form Constant Character String

FCW - Define Word

Lecture Code Examples

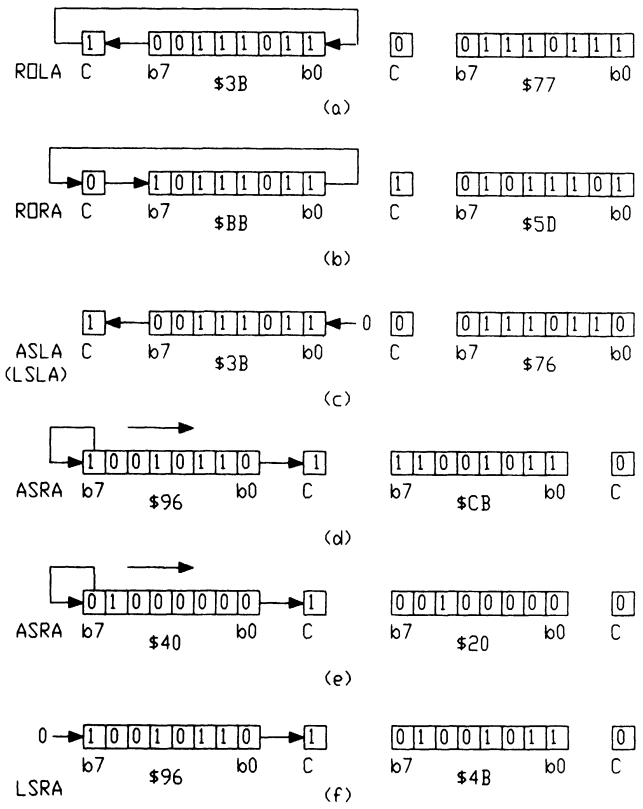
Shift Left & Right

Binary Multiplication

Binary Division

Extending for larger numbers

Shift & Rotate - Left & Right



The six 68HC11 Addressing Modes

Immediate
Direct
Extended
Indexed
Inherent
Relative

Immediate Addressing Mode (Syntax)

The data for the operation **immediate**ly follows the instruction
The **# symbol must be used to select immediate addressing**

Examples:

```
LDAA #$42    ; Load Acc.A with the value $42                -> Machine Codes: 86 42
LDX #$DEEE ; Load Index Register X with hex value DEEE  -> Machine Codes CE DE EE
```

Direct Addressing Modes

Direct Addressing can access an operand located in the first 256 bytes of memory, $00..$FF
An 8-bit memory address specifies where - data is read from, or where data is written to.
Examples:

```
LDAA $42      ; AccA = Memory[$42] -> Machine Code 96 42
STAA  $FF     ; Memory[FF] = AccA
LDX    $12    ; IX = Memory[$12]..[$13]
```

Extended Addressing Modes

Extended Addressing uses 16-bit address to specify
A 16-bit memory address specifys where: data is read from, or data is written to.

Examples:

```
LDAA $4321  ; AccA = Memory[$4321] -> Machine Code B6 43 21
STAA  $2000  ; Memory[$2000] = AccA
LDX    $1234  ; IX = Memory[$1234]..[$1235]
```

Indexed Addressing Modes

Indexed addressing always uses one of two index registers **X** or **Y**.
Instruction Format: `Operation Offset, Index_Register`

Offset is an unsigned 8-bit value (0..255) added to the contents of the 16 bit index register
The value of the index register is not altered. Addition is modulo 65536

EXAMPLE:

LDAA 42, X ; AccA = Memory[42+X]
The Assembler Generates -> Machine Code A6 42

Inherent Addressing Modes

All data for the instruction is within the CPU registers already.

ABA	; ADD B to A	A = A + B	-> Machine code $1B
MUL	; MULTIPLY A x B	D = A * B	-> Machine code $3D
NEGA	; NEGATE A	A = -A	-> Machine code $40
ABX	; ADD B to X	X = X + B	-> Machine code $3A
INX	; Invrement X	X = X + 1	-> Machine code $08
SEC	; Set Carry	CarryFlag = 1	-> Machine code $0D
TAB	; Transfer A to B	B = A	-> Machine code $16

Addressing Modes Review

Immediate, Direct, Extended, Indexed, Inherent, Relative

LDAA 4231 -> B6 42 31
LDAA 42 -> 96 42
LDAA #42 -> 86 42
LDAA 42, X -> A6 42
BRA 2020 -> 20 1E (at $2000)
ABA -> 1B

Note - there are no memory to memory moves without an interim register

Tha 68HC11 Instruction set

Instruction Set Categories

ARITHMETIC
BRANCH, JUMP & SUBROUTINE CALL/RETURN
COMPARE
DATA MOVEMENT
LOGICAL

ARITHMETIC Instructions
ADDITION,
SUBTRACTION,
TWOS COMPLEMENT (NEGATION),
DECREMENT & INCREMENT
MULTIPLY & DIVIDE
ARITHMETIC/LOGICAL SHIFT/ROTATE
BINARY CODED DECIMAL
CLEAR (bit(s) = 0) & SET (bit(s) = 1)

COMPARE & TEST
CONDITION CODE MANIPULATION
CONDITIONAL Branches
SIGNED / UN-SIGNED NUMERIC INTERPRETATION

DATA MOVEMENT
Push/Pull
Load/Store Register
Transfer Registers
Exchange Registers
INTERRUPT HANDLING:

LOGICAL
LOGICAL AND
LOGICAL EXCLUSIVE OR
LOGICAL OR
ONES-COMPLEMENT (NOT)
MISCELLANEOUS

ADDITION

ABA	A = A + B
ABX	IX = IX + B
ABY	IY = IY + B
ADCA	A = A + M + CarryFlag
ADCB	B = B + M + CarryFlag
ADDA	A = A + M
ADDB	B = B + M
ADDD	D = D + M

SUBTRACTION

SBA	A = A - B
SBCA	A = A - M - CarryFla
SBCB	B = B - M - CarryFlag
SUBA	A = A - M
SUBB	B = B - M
SUBD	D = D - M

NEGATION

NEG	M = -M
NEGA	A = -A
BEGB	B = -B

TWOS COMPLEMENT

DECREMENT

DEC	M = M - 1
DECA	A = A - 1
DECB	B = B - 1
DES	SP = SP - 1
DEX	IX = IX - 1

INCREMENT

INC	M = M + 1
INCA	A = A + 1
INCB	B = B + 1
INS	SP = SP + 1
INX	IX = IX + 1
INY	IY = IY + 1

MULTIPLY / DIVIDE

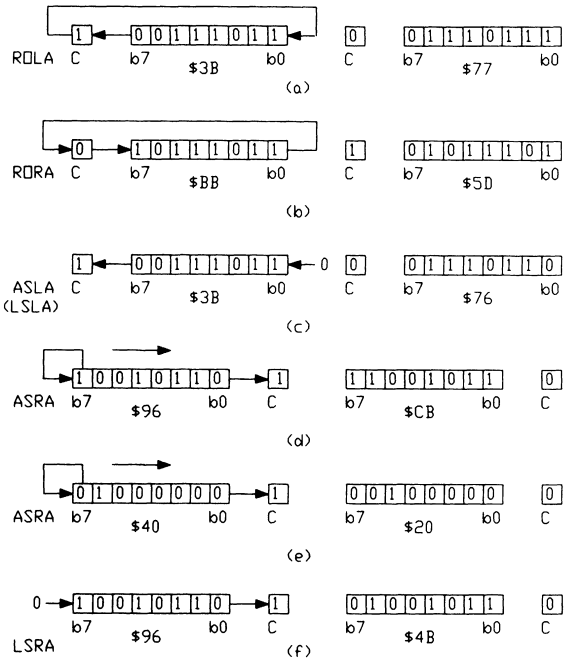
MULTIPLY

MUL	D = A * B

DIVIDE

IDIV	IX = D / IX, D = D % IX (Unsigned Integer Divide)
FDIV	IX = D / IX, D = D % IX (Unsigned fractional divide - Radix point assumed left of bit 15 of both operands)

Arithmetic/Logical Shift & Rotate



ARITHMETIC SHIFT

LEFT: (Multiply by 2)

ASL	Arithmetic Shift Left (M)
ASLA	Arithmetic Shift Left (A)
ASLB	Arithmetic Shift Left (B)
ASLD	Arithmetic Shift Left (D)

RIGHT: (Divide By 2, Sign-Extend MSBit)

ASR	Arithmetic Shift Right (M)
ASRA	Arithmetic Shift Right (A)
ASRB	Arithmetic Shift Right (B)

LOGICAL SHIFT

LEFT: (Same as ASL)

LSL	Logical Shift Left (M)
LSLA	Logical Shift Left (A)
LSLB	Logical Shift Left (B)
LSLD	Logical Shift Left (D)

SHIFT RIGHT: (Zero Fill MSBit)

LSR	Logical Shift Right (M)
LSRA	Logical Shift Right (A)
LSRB	Logical Shift Right (B)
LSRD	Logical Shift Right (D)

ROTATE

LEFT: (extended multiply)

ROL	ROtate Left (M)
ROLA	ROtate Left (A)
ROLB	ROtate Left (B)

RIGHT: (extended divide)

ROR	ROtate Right (M)
RORA	ROtate Right (A)
RORB	ROtate Right (B)

CLEAR (bit(s) = 0) & SET (bit(s) = 1)

CLR	M = 0
CLRA	A = 0
CLRB	B = 0
BCLR	Clear Bits (M)
BSET	Set Bits (M)

Jump Instructions

JUMP address (0..FFFF)

JMP	Jump to Address
JSR	Jump to Subroutine
RTS	Return from Subroutine
NOP	No OPeration ; i.e do nothing but fetch next instruction

Branch & Jump

BRA	Branch Always
BRN	Branch Never

COMPARE & TEST

CONDITION CODE MANIPULATION

CLC	CarryFlag = 0	Clear Carry Flag
CLV	OVerflowFlag = 0	Clear Overflow Flag
SEC	CarryFlag = 1	Set Carry Flag
SEV	OVerflowFlag = 1	Set Overflow Flag
TAP	CCR = A	Transfer A to Condition Codes Register (CCR)
TPA	A = CCR	Transfer CCR to A

CONDITIONAL Branches

BEQ	Branch if EQual	
BNE	Branch if Not Equal	
BCC	Branch if CarryFlag is Clear	
BCS	Branch if CarryFlag is Set	
BRCLR	Branch if bits clear	
BRSET	Branch if bits set	

SIGNED COMPARISON

BMI	Branch if Minus	
BPL	Branch if Plus	
BVS	Branch if oVerflow Set	
BVC	Branch if oVerflow Clear	
BGT	Branch if Greater-Than	>
BGE	Branch if Greater-Than or Equal-to	>=
BLE	Branch if Less-Than or Equal-to	<=
BLT	Branch if Less Than	<

UN-SIGNED COMPARISON

BHI	Branch if HIgher than	>
BHS	Branch if Higher or Same	>=
BLS	Branch if Lower or Same	<=
BLO	Branch if Lower	<

Push - Push register value onto stack

PSHA ; M[SP--] = A

The contents of the A register is pushed onto the stack - i.e. stored into memory at the address contained in the stack pointer; then the stack pointer is decremented.

PSHB ; M[SP--] = B

PSHX ; M[SP--] = IX.LOW ; M[SP--] = IX.HIG

PSHY ; M[SP--] = IY.LOW ; M[SP--] = IY.HIGH

Pull - Pull (POP) value from stack to Register

PULA ; A = M[++Sp]

The stack pointer is first incremented, and then Accumulator A is popped off the stack - i.e. loaded from memory where the stack pointer is pointing.

PULB ; B = M[++SP]

PULX ; X.HIGH = M[++SP], X.LOW = M[++SP]

PULY ; Y.HIGH = M[++SP], Y.LOW = M[++SP]

Load Register

```
LDAA ; A = M
LDAB        ; B = M
LDD         ; D = M
LDS         ; SP = M
LDX         ; X = M
LDY         ; Y = M
```

Store Register

```
STAA        ; M = A
STAB        ; M = B
STS         ; M = SP
STD         ; M = D
STX         ; M = IX
STY         ; M = IY
```

Transfer Registers

```
TAB         ; A = B
TBA         ; B = A
TSX         ; IX = SP + 1
TSY         ; IY = SP + 1
TXS         ; SP = IX - 1
TXY         ; SP = IY - 1
```

Exchange Registers

```
XGDX        ; D <=> IX
XGDY        ; D <=> IY
```

INTERRUPT HANDLING:

```
CLI         ; Clear interrupt Mask
SEI         ; Set interrupt Mask
SWI         ; Software Interrupt
RTI         ; Return from Interrupt
WAI         ; Wait for interrupt
```

AS11 Constant Definition

' ASCII character
$ hexadecimal constant
@ octal constant
% binary constant
digit decimal constant

AS11 Assembler Directives

EQU - EQUATE

Used to define constants for example:
OUTSTR EQU $E3D9
ORG xxxx - ORIGINATE
Locate the program at the specified address, xxxx

AS11 Assembler Expressions
Expressions may consist of symbols, constants or the character "*" (denoting the current value of the program counter) joined together by one of the operators: + - * / % & | ^.
The operators are same as C:

+	add
-	subtract
*	multiply
/	divide
%	modulo
&	bitwise AND
|	bitwise OR
^	bitwise EXCLUSIVE-OR

Command line assembler Invocation & command line options
To invoke the cross-assembler AS11 in a dos box enter:
AS11 file1 (file2...) -option1 -option2
AS11 Options:

l	enable output listing.
nol	disable output listing (default).
cre	generate cross reference table.
s	generate a symbol table.
c	enable cycle count.
noc	disable cycle count.

Example: AS11 hex2A.asm -l > hex2A.lst
(If successful creates a file hex2A.S19 - this can be **load**ed directly into HCCOM for testing)

Example: Single Hex Number Conversion to ASCII using C Language

```
// The algorithm to convert Hex2Ascii in the C language:
// We know ASCII characters
//          0..9 are $30..$39 - So the function is Add $30, and for
//          A..F are $41..47   - Add ($41 - $0A) = $37
char hex2ascii (char hex_in) // returns an ASCII character
{ char result;
  result = (hex_in & 0x0F);     // Mask input to be in the range $0..$F
  result += 0x30;               // assume range 0..9, so add 0x30
  if (result > 0x39)            // check above assumption
   {                            // result is 0x3A or more -> incorrect assumption so fix it
     result += 7;               // add 7 to get the final result in the range $41..$47
```

```
    }
  return (result);                  // return the result - is returned in Accumulator A
}
```

Hex2ASCII using AS11 (the M68HC11 assembler)

```
; This subroutine converts a single hex number (0 to F) to ASCII
; ON ENTRY:  ACCA =  hex digit to be converted
; ON RETURN: ACCA = ASCII character code of hex digit
HEX2ASCII
        ANDA    #$0F  ; Ensure number is in range $0..$F
        ADDA    #$30   ; 30 Hex = ASCII character '0'
        CMPA    #$39  ; check if digit > '9'
        BMI     DONTADD
        ADDA    #$07  ; ACCA > '9' so add another 7
DONTADD
        RTS           ; return to calling routine
                      ; Accumulator A holds the returned result
```

Entering Hex2ASCII using BUFFALO Program

```
ASM 2000
ANDA #0F
ADDA #30
CMPA #39
BMI 200A  - Buffalo's inbuilt assembler cannot use symbols
ADDA #07
RTS
```

As you can see, BUFFALO assumes all numbers are hexadecimal.

```
La Trobe University HC-COM.
BUFFALO 3.41 (ext) - Bit User Fast Friendly Aid to Logical Operation
by Tony Fourcroy. Ported to HC-COM by jtc.

Free memory range: 2000 - 7FFF
>ASM 2000

2000 TEST            >ANDA #0F
     84 0F
2002 TEST            >ADDA #30
     8B 30
2004 TEST            >CMPA #39
     81 39
2006 TEST            >BMI 200A
     2B 02
2008 TEST            >ADDA #07
     8B 07
200A TEST            >RTS
     39
200B TEST            >

200C TEST            >♥
>_
```

Byte2ASCII in C

```
; This subroutine converts a Byte to 2 ASCII digits
; ON ENTRY:    ACCA =   byte to be converted
; ON RETURN:  2 ASCII character codes
int byte2ascii (char byte_in)              // returns an ASCII character
{ int result;                              // integer = 2 bytes
   result = (hex2ascii (byte_in >> 4);     // convert high hex digit to character
   result = (result  << 8);                // and store in upper 8 bits
   result += hex2ascii (byte_in);          // convert low hex digit to character
   return (result);                        // result is returned in A & B register
  // AccA=Most Significant Character, AccB=Least Significant Character
}
```

Byte2ASCII - AS11 (68HC11 assembler)

```
; This subroutine converts a Byte to 2 ASCII digits
; ON ENTRY:    ACCA =   byte to be converted
; ON RETURN: ACCA:ACCB = 2 ASCII character codes
BYTE2ASCII
        PSHA                    ; save Accumulator A on stack
        ANDA        #$0F        ; mask low byte
        JSR         HexToAscii
        TAB                     ; Transfer Character in AccA to AccB
        PULA                    ; restore Accumulator A from stack
        LSRA                    ; shift Accumulator A right four times
        LSRA                    ; to transfer high nybble to low nybble
        LSRA
        LSRA
        JSR         HexToAscii  ; most significant character in AccA
        RTS                     ; return to calling routine
                                ; Accumulator A holds the returned result
```