Programming the Motorola MC68HC11 Microcontroller

COMMON PROGRAM INSTRUCTIONS WITH EXAMPLES

aba **Add register B to register A** *Similar commands are* `abx aby`

```
aba
```
add the value in register B to the value in register A and store in register A

anda **Logical And with register A** *Similar command is* `andb` Differs from `bita` in that the contents of register A is changed

```
anda #label
```
perform a logical AND between the value stored at memory location *label* and register A and store the result in register A

asr **Arithmetic Shift Right** *Similar commands are* `asra asrb asl asla aslb`

```
asr
```
Preserves signed numbers by retaining the leading bit. Use `lsr` and related commands with unsigned numbers. A right shift divides by 2, a left shift multiplies by 2.

bcc **Branch if C-bit is clear** *Similar command is* `bcs` *(branch if C-bit set)*

```
bcc
```
branches if the C-bit is clear. The C-bit indicates a carry or borrow.

bclr **Clear Bit(s)** *Similar command is* `bset`

```
bclr #label $F0
```
this example zeros the first four bits of the value stored at memory location *label*. `$F0` is the *mask*, in binary it is 11110000; the 1's correspond to the bits that will be cleared.

beq **Branch on Equal or Zero, i.e. if CCR Z-bit is 1**

```
cmpa #20
```
compares the value in register A to decimal 20 by subtracting 20 from A.

```
beq  label
```
if the last value in memory was a zero (checks the CCR Z-bit) then go to program location *label.*

```
tsta
```
test the value in register A

```
beq  label
```
if the value in register A is zero (i.e. the Z-bit is set) then we branch to the memory location *label*.

bita **Logical And with register A** *Similar commands are* `bita, bitb` Differs from `anda` in that register A remains unchanged. The result affects only the CCR.

```
bita #%10000000
```
this example checks bit 7 in register A and set the CCR accordingly. This could be follow with the beq or bne instruction to branch based on the result of the bit test. Another way to test bit 7 is to simply tsta and then branch based on the N-bit since bit 7 = 1 is characteristic of a signed negative number and will set the N-bit of the CCR.

ble **Branch if Lower or Equal**

Compares **signed** numbers. *Similar commands:* `blt` (branch if lower), `bgt` (branch if greater than), `bge` (branch if greater or equal). See `bls` for comparable *unsigned* number commands with examples.

bls **Branch if Lower or Same**

Compares **unsigned** numbers. *Similar commands:* `blo` (branch if lower), `bhi` (branch if higher), `bhs` (branch if higher or same). May not work properly if there is an overflow.

```
cba          first compare the value in register B to the value in register A  (A-B)
bls  label   branch to location label if A is less than or equal to B

ldd  Num1    16-Bit Version:  first load the value stored at Num1 into register D
cpd  #1000   compare the value in D to 1000
bls  label   branch to location label if D is less than or equal to 1000
```

bmi **Branch on Minus**

Similar command is `bpl` *(branch on positive)*

```
tsta         test the value in register A
bmi  label   if the value in register A is negative (i.e. the N-bit is set) then we branch to
             the memory location label.
```

bne **Branch if Not Equal or Zero**

Opposite of `beq`

bra **Branch**

```
bra  label   go to program location label and continue execution (don't return).
```

brclr **Branch if Bit(s) Clear**

```
brclr label1 #%11100000 label2   go to program location label2 if the first three bits of
                                 the value stored at label1 are zeros (clear).
```

bsr **Branch to Subroutine**

```
bsr  label   go to the subroutine at program location label and return here when done
```

bvs **Branch if Overflow bit is set**

```
bvs  label   go to the program location label if the v bit is set in the CCR.  The V-bit
             indicates a twos-complement overflow.
```

cba **Compare B to A**

Similar commands are `cmpa cmpb cpd cpx cpy`; see example at `bls`

```
cba          compare the value in register B to the value in register A by subtraction (A-B)
             and set the CCR accordingly.  If A=B then Z→1.  Can be used before beq,
             ble, blt, bgt, bge, bls, blo, bhi, bhs, etc.
```

clr **Replace Contents with Zeros**

Similar commands are `clra clrb`

```
clr  Ddrc    this example causes Port C to be an input port (all pins).  This would go
             near the beginning of the program after the lds command.
clra         this example places zeros in register A.
```

cmpa **Compare to A** *Similar commands are* `cba cmpb cpd cpx cpy`; see example at `bls`

```
cmpa #$04
```
this example compares the value in register A to $04 by subtracting $04 from register A. If the result is zero then they are equal and the CCR bit Z is set to 1. $04 is EOT or end of string. Often used before `beq.`

```
cmpa #EOT
```
this example compares the character in register A to the end of string character.

```
cmpa #end-3
```
"end" must be a constant, not a label. The subtraction of end-3 is performed and the value in register A is compared to the result.

```
cmpa 0,x
```
compare the value in register A to the value in the byte pointed to by register X.

Refer also to the ldaa command for discussion on the use of the # sign.

coma **Complement of A** *Similar commands are* `com comb`

```
coma
```
complement the value in register A and store the result in register A.

dec **Decrement by 1** *Similar commands are* `deca decb des dex dey`

```
dec  label
```
decrement the value stored at memory location *label* by 1.

```
deca
```
decrement the value stored in register A by 1. (Inherent addressing)

```
des
```
decrement the stack pointer; may be used to **allocate** stack space

```
dec  0,x
```
decrement the value stored at the top of the stack

end **End Program**

```
end
```
last program instruction

eora **Exclusive OR with reg A** *Similar command is* `eorb`

```
eora label
```
an exclusive OR is performed with the contents of register A and the value at address *label* with the result stored in register A.

equ **Equate a Label to a Value**

```
label equ  3
```
the assembler substitutes the value 3 wherever it sees *label* in the code. This does not use any memory space. The purpose is to facilitate code maintenance by permitting a single change of value here to result in multiple changes throughout the code wherever *label* appears. The line should be placed toward the beginning of the program or section of code before the first use of *label*.

fcb **Form Constant Byte** see SUBROUTINE LIBRARIES

fcc **Form Constant Character String** see SUBROUTINE LIBRARIES

fdb **Form Double Byte Constant**

```
fdb  main
```
This particular example is common to all our programs. By appearing after the `org $FFFE` instruction near the end of the program, this code loads the starting address of the program (represented by the label *main*) into the last two bytes of ROM. The cpu looks in the last two bytes of ROM to obtain the address for the beginning of the program when power is applied or in the event of a reset.

```
label fdb  5,8,465,17,89
```
5 is stored in a 2-byte block at mem location *label*, 8 is stored in a 2-byte block at location *label*+2, etc..

fdiv **Fractional Divide D/X** *Related commands are* `fdiv, mul`

```
ldd   #2
ldx   #3
fdiv
```

2 is loaded into register D (numerator)
3 is loaded into register X (denominator)
actually, the numerator is multiplied by 65536 before being divided by the denominator, quotient (43690) goes in register X, remainder (2) in register D, I think.

inc **Increment by 1** *Similar commands are* `inca incb ins inx iny`

```
inc   label
inca
ins
```

increment the value stored at memory location *label* by 1.
increment the value stored in register A by 1. (Inherent addressing)
increment the stack pointer; used to **deallocate** space on the stack

idiv **Integer Divide D/X** *Related commands are* `fdiv, mul`

```
ldd   #9
ldx   #4
idiv
```

9 is loaded into register D (numerator)
4 is loaded into register X (denominator)
division takes place, quotient (2) goes in register X, remainder (1) in register D

jmp **Jump to Another Location**

```
jmp   label
```

go to program location *label*. You can use this if you are not planning on returning to the current location.

jsr **Jump to Subroutine**

```
jsr   InString
```

go to a subroutine. This is used with the libraries because they are too far away to be accessed with the branch instructions which use relative addressing. Program execution returns to this point following the subroutine.

```
jsr   InitSCI
```

this example initializes the serial port (SCIWin on our simulator) and appears once in the program right after *main*. `InitSCI` is in our subroutine library.

ldaa **Load Register A** *Similar commands are* `ldab ldd lds ldx ldy`

```
ldd   10
ldaa  #10
ldaa  #$B
ldaa  #'B
ldaa  #%10011001
ldd   #label
ldaa  label
ldaa  Porte
const equ 2
ldaa  #const
ldaa  const,x
ldaa  4,X
```

load the value at address $000A into register D
load the decimal value 10 into register A
load the hex value B into register A
load the ASCII character code for B into register A
load the binary value 10011001 into register A
load the **address** value of *label* into register D
load the **data** value of *label* into register A
load the **data** from input Port E into register A
create a constant
load the **data** value 2 into register A
load the **data** that is 2 bytes past the address in register X into register A
load the **data** located 4 bytes past the location stored in register X into register A

Note the confusion we might have since `#10` and `label` and `#const` all denote data and `10` and `#label` denote addresses, and in the line `ldaa const,x` (*indexed addressing*), `const` is referring to data (2) again without the # sign. So although the # is significant in determining whether we are talking addresses or data, its meaning is not consistent in that regard. When the # sign is used it denotes the *immediate addressing mode* and this only occurs with load and compare commands (I think). So when we have the command `beq label`, label is an address even though the # sign is absent.

lds **Load Stack Pointer**

```
lds   #$00FF
```
this example initializes the stack pointer; required if the stack is to be used; same value is normally used; goes near the top of the program after `org $E000`

lsr **Logical Shift Right** *Similar commands are* `lsra lsrb lsrd` *and for left shift:* `lsl lsla` *etc.*

For use with unsigned numbers. See `asr` and related commands for use with signed values. A right shift divides by 2, a left shift multiplies by 2.

```
lsr   label
```
divide the value pointed to by *label* by 2.

```
lsra
```
the contents of register A are shifted to the right one bit and bit 7 becomes zero.

mul **Multiply A × B = D** *Related commands are* `idiv fdiv`

```
ldaa  #10
```
load 10 into register A

```
ldab  #5
```
load 5 into register B

```
mul
```
the values are multiplied, result goes in register D (unsigned values only, no overflow is possible).

org **Sets the Program Counter, which specifies the address of the next byte to be loaded**

```
org   0
```
first program instruction

```
org   $E000
```
follows global variables; moves to the beginning of the program area

```
org   $FFFE
```
third from last command; makes room for a 2-byte reset address. The address stored here tells the CPU where to look for the beginning of the program when it is powered up.

psha **Push Register A onto Stack** *Similar commands are* `pshb pshx pshy`

```
psha
```
put the contents of register A on the stack and decrement the stack pointer; used for saving the contents of a register at the start of a subroutine, the registers are restored near the end of the subroutine using `pula pulb pulx puly`

pula **Pull from Stack to Register A** *Similar commands are* `pulb pulx puly`

```
pula
```
pull the value from the top of the stack and store in register A; increment the stack pointer; used for restoring the contents of a register at the end of a subroutine, the registers are saved near the beginning of the subroutine using `psha pshb pshx pshy`

rmb **Reserve Memory Bytes**

```
label rmb   2
```
creates a global variable or array, goes near the top of the program after `org 0.` Consists of the label name to be used for the memory location followed by `rmb` following by the number of bytes

rts **Return from Interrupt** *Similar command is* `rts`

```
rti
```
goes at the end of an interrupt routine, pulls all registers and the return address from the stack.

rts **Return from Subroutine** *Similar command is* `rti`

```
rts
```
goes at the end of a subroutine, pulls the return address from the stack.

sev **Set the V-bit**

```
sev
```
sets the V-bit to 1 in the condition code register (CCR)

staa **Store the value that is in Register A into . . .** *Similar commands are* `stab std sts stx sty`

`staa label`	store the value that is in register A in the memory location *label*

stop **Stop Program Execution**

`stop`	stops the program at this point

suba **Subtract from register A** *Similar commands are* `subb subd`

`suba label`	subtract the value stored at *label* from register A and store in register A
`suba #12`	subtract decimal 12 from register A and store in register A

tab **Transfer A to B** transfers the value in register A to register B, leaving A intact

tcnt **Timer Counter Register** a 2-byte register that increments once with each program instruction during execution

tsx **Transfer Stack Pointer to Register X** *Similar command* `txs`

`tsx`	stores the address of the last value saved on the stack into register X. The stack pointer continues to point to the next empty byte, i.e. SP + 1 = X.

xgdx **Exchange D and X** exchanges values in registers D and X. Commonly used to permit 16-bit arithmetic to be done on a register address.