Programming the Motorola MC68HC11 Microcontroller

CONTENTS:

COMMON PROGRAM INSTRUCTIONS WITH EXAMPLES
MEMORY LOCATIONS
PORTS
SUBROUTINE LIBRARIES
PARALLEL I/O CONTROL REGISTER (PIOC)

COMMON PROGRAM INSTRUCTIONS WITH EXAMPLES

aba  Add register B to register A

    \textit{Similar commands are} \ aba \ abx \ aby

    \textit{add the value in register B to the value in register A and store in register A}

anda  Logical And with register A

    \textit{Similar command is} \ andb \ Differs from bita in that the contents of
    register A is changed

    \textit{perform a logical AND between the value stored at memory location label}
    \textit{and register A and store the result in register A}

asr  Arithmetic Shift Right

    \textit{Similar commands are} \ asra \ asrb \ asl \ asla \ aslb

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

bcc  Branch if C-bit is clear

    \textit{Similar command is} \ bcs \ (branch if C-bit set)

    \textit{branches if the C-bit is clear. The C-bit indicates a carry or borrow.}

bclr  Clear Bit(s)

    \textit{Similar command is} \ bset

    \textit{this example zeros the first four bits of the value stored at memory location}
    \textit{label}. $\$F0$ \textit{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

    \textit{Similar commands are} \ bne \ \textit{branch based on the result of the bit test.}

    \textit{this could be follow with the beq or bne instruction to branch based on the}
    \textit{result of the bit test. Another way to test bit 7 is to simply tsta and then}
    \textit{branch based on the N-bit since bit 7 = 1 is characteristic of a signed}
    \textit{negative number and will set the N-bit of the CCR.}
**ble**  Branch if Lower or Equal  
Compares signed numbers. Similar commands: 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
- **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, bgt, bge, bls, blo, bhi, bhs, etc.

**clr**  Replace Contents with Zeros
- **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 bis

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. "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** #end-3

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

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

dec a

dec 0,x

decrement the value stored at memory location label by 1.

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

decrement the stack pointer; may be used to allocate stack space

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

This particular example is common to all our programs. By appearing after the org $FFFF 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**

  2 is loaded into register D (numerator)
  3 is loaded into register X (denominator)

**fdiv**

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 in register A by 1. (Inherent addressing)

**ldd #2**

2 is loaded into register D (numerator)

**ldx #3**

3 is loaded into register X (denominator)

**fdiv**

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.

- **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**
- **ldda Porte**
- **const equ 2**
- **ldaa #const**
- **ldaa const,x**
- **ldaa 4,X**

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 $FFFFE  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)
<table>
<thead>
<tr>
<th><strong>sta</strong></th>
<th><strong>Store the value that is in Register A into . . .</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sta</strong></td>
<td><strong>Similar commands are</strong> sta, std, sts, stx, sty</td>
</tr>
<tr>
<td><strong>sta</strong></td>
<td><strong>sta label</strong></td>
</tr>
<tr>
<td></td>
<td>store the value that is in register A in the memory location <strong>label</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>stop</strong></th>
<th><strong>Stop Program Execution</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>stop</strong></td>
<td><strong>stop</strong></td>
</tr>
<tr>
<td></td>
<td>stops the program at this point</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>suba</strong></th>
<th><strong>Subtract from register A</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>suba</strong></td>
<td><strong>Similar commands are</strong> subb, subd</td>
</tr>
<tr>
<td><strong>suba</strong></td>
<td><strong>suba label</strong></td>
</tr>
<tr>
<td><strong>suba</strong></td>
<td><strong>suba #12</strong></td>
</tr>
<tr>
<td></td>
<td>subtract the value stored at <strong>label</strong> from register A and store in register A</td>
</tr>
<tr>
<td></td>
<td>subtract decimal 12 from register A and store in register A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>tab</strong></th>
<th><strong>Transfer A to B</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>transfers the value in register A to register B, leaving A intact</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>tcnt</strong></th>
<th><strong>Timer Counter Register</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>tcnt</strong></td>
<td><strong>a 2-byte register that increments once with each program instruction during execution</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>tsx</strong></th>
<th><strong>Transfer Stack Pointer to Register X</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>tsx</strong></td>
<td><strong>Similar command</strong> txs</td>
</tr>
<tr>
<td><strong>tsx</strong></td>
<td><strong>tsx</strong></td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>xgdx</strong></th>
<th><strong>Exchange D and X</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>xgdx</strong></td>
<td><strong>exchanges values in registers D and X. Commonly used to permit 16-bit arithmetic to be done on a register address.</strong></td>
</tr>
</tbody>
</table>
MEMORY LOCATIONS

$0000 - $00FF
RAM, 256 bytes. Globals and the Stack reside here. Globals are loaded at the low end, i.e. $0000, $0001, etc. The Stack is builds from the high end also referred to as the bottom of the stack, i.e. $00FF. The stack pointer points to the top of the stack, which is the first unused stack location, i.e. if $00FC contains the last stored data then the stack pointer points to $00FB.

$1000 - $103F
Control registers, status registers and ports are mapped to this area

$1040 - $B5FF
Unused addresses.

$B600 - $B7FF
Internal EEPROM, 512 bytes for short programs

$B800 - $DFFF
Unused addresses.

$E000 - $FFFF
ROM, 8192 bytes. Object code and libraries go here. Libraries begin at $FE00 (a convention in this class) and interrupt vectors begin at $FFF6. Interrupt vectors are pointers to code to be implemented in the event of an interrupt. For example, the reset interrupt consists of the bytes $FFFE and $FFFF so this is where we put the address that tells the CU where to begin program execution on power-up.

PORTS

PortA ($1000) Pins 0, 1, 2 are inputs. Pins 3, 4, 5, 6 are outputs. Pin 7 is selectable by bit 7 of the pulse accumulator control register (Pact1) 0 = input, 1 = output.

PortB ($1004) Output.

PortC ($1003) Configurable as

- input - Ddrc = 0
- latched input - Ddrc = 0, HNDS = 0
- output - Ddrc = 1, CWOM = 0
- open collector output - Ddrc = 1, CWOM = 1

The Ddrc is an 8-bit register so each bit of PortC can be set independently as either input or output.

PortD ($1008) Serial I/O - inputs and outputs are set by data direction register Ddrc.

PortE ($100A) Input - ADPU bit = 1 - A/D conversion.
ADPU bit = 0 - parallel input
SUBROUTINE LIBRARIES

**InCh** accepts input from the keyboard. When a key is pressed, the ASCII value is loaded in register A and the routine returns to the main program.

**InDec** accepts input from the keyboard. Returns an unsigned 16-bit number in register D. V-bit is set if illegal character is typed.

**InHex** accepts input from the keyboard as hex (must be 0-9, A-F, or a-f; ignored are space, tab and $). Result goes in register D.

**InitSCI** initializes the serial communications interface. Gets executed once near the beginning of the program. Watch the capitalization; instructor often gets this wrong on assignments.

```plaintext
jsr InitSCI Initialize the SCI port
```

**InString** accepts input from the keyboard. Before calling this routine, register X must point to a memory array and register B must contain the maximum length of the string. The string is inputted to the SCI window and stored in the array pointed to by register X; the last character is EOT=$04. The string array may be created as a global as follows:

```plaintext
Str rmb 20 String variable
```

**OutCh** places the character in register A on the screen in the SCI window.

**OutDec** outputs a 16-bit number in register D to the SCI window.

**OutHex** outputs the contents of register D to the SCI window in unsigned hex format.

**OutString** displays a string in the SCI window. Before calling this routine, register X must point to a memory array containing the string, ending with EOT=$04. The following code may be used to produce the string. It is placed at the end of the program just before the last org command.

```plaintext
Label fcb CR
    fcc "My message here ">
    fcb EOT
```

Note that the quotation marks in "My message here "> is not the only delimiter that can be used to mark the string. Any printable ASCII character other than ";" will work; whatever the first character is becomes the delimiter. EOT is defined as $04; it is the end of string character. The instructions above load values into ROM beginning at the current program location.
PARALLEL I/O CONTROL REGISTER (PIOC)

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>STAF</td>
<td>STAI</td>
<td>CWOM</td>
<td>HNDS</td>
<td>OIN</td>
<td>PLS</td>
<td>EGA</td>
<td>INVB</td>
</tr>
</tbody>
</table>

STAF - Strobe A Flag
Set in response to STRA pin change of state, selectable rising or falling edge according to how the EGA bit is set. How the STAF is cleared depends on the selected handshake mode:

Simple Strobe Mode (HNDS=0) - clears when PORTCL is read
Full-Input Handshake Mode (HNDS=1, OIN=0) - clears when PORTCL is read
Full-Output Handshake Mode (HNDS=1, OIN=1) - clears when PORTCL is written to

STAI - Strobe A Interrupt Enable
0 - STAF interrupts are inhibited.
1 - A hardware interrupt request is generated whenever the STAF bit is set.

CWOM - Port C Wired-OR Mode
0 - Port C outputs are active push-pull drivers.
1 - Port C outputs are open-drain drivers.

HNDS - Handshake/Simple Strobe Mode Select
0 - Simple Strobe mode is selected; STRB is pulsed for 2 clock cycles after each write to port B (no handshaking).
1 - either full-input or full-output handshake mode is selected; Port C is used.

OIN - Output/Input Handshake Select
HNDS must be 1 or there is no effect.
0 - full-input handshake is selected.
1 - full-output handshake is selected.

PLS - Strobe B Pulse Mode Select
Controls the configuration of the STRB pin. HNDS must be 1 or else STRB will default to pulsed mode.
0 - Interlocked mode; (HNDS must be 1) STRB will remain active until an edge is detected at the STRA pin.
1 - Pulsed mode; STRB is active for 2 clock cycles when triggered.

EGA - Edge Select for Strobe A
0 - falling edges are detected at the STRA input pin.
1 - rising edges are detected at the STRA input pin.

INVB - Invert Strobe B
0 - Negative logic; STRB signals are active low.
1 - Positive logic; STRB signals are active high.