HC11 Instruction Set
Instruction classes

1. Accumulator and Memory
2. Stack and Index Register
3. Condition Code Register
4. Program control instructions
Accumulator and memory instructions

- Loads, stores, and transfers (LST)
- Arithmetic operations
- Multiply and divide
- Logical operations
- Data testing and bit manipulation
- Shifts and rotates
Almost all MCU activities involve transferring data from memories or peripherals into the CPU or transferring results from the CPU into memory or I/O devices.

<table>
<thead>
<tr>
<th>Function</th>
<th>Mnemonic</th>
<th>IMM</th>
<th>DIR</th>
<th>EXT</th>
<th>INDX</th>
<th>INDY</th>
<th>INH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear Memory Byte</td>
<td>CLR</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Clear Accumulator A</td>
<td>CLRA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Clear Accumulator B</td>
<td>CLRB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Load Accumulator A</td>
<td>LDAA</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Load Accumulator B</td>
<td>LDAB</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Load Double Accumulator D</td>
<td>LDD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Pull A from Stack</td>
<td>PULA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Pull B from Stack</td>
<td>PULB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Push A onto Stack</td>
<td>PSHA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Push B onto Stack</td>
<td>PSHB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Store Accumulator A</td>
<td>STAA</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Store Accumulator B</td>
<td>STAB</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Store Double Accumulator D</td>
<td>STD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Transfer A to B</td>
<td>TAB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Transfer A to CCR</td>
<td>TAP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Transfer B to A</td>
<td>TBA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Transfer CCR to A</td>
<td>TPA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Exchange D with X</td>
<td>XGDX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Exchange D with Y</td>
<td>EGDY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
LST Example

AAA: .asciz "I love assembly language!"

ldx #AAA
xgdx
addd #5
xgdx
ldaa 0, X
jsr OUTCHAR
jsr OUTCRLF
dex
dex
ldab 0, X
tba
staa 2, X
ldaa #0
staa 3, X
ldx #AAA
jsr OUTSTRING
jsr OUTCRLF
Endian-ness

- Storing a 16-bit word in memory at address [ADDR, ADDR+1]
- Two choices:
  - Little-endian: at address ADDR store the LSB (the “little end”)
  - Big-endian: at address ADDR store the MSB (the “big end”)
- Example: storing 0xAABB

<table>
<thead>
<tr>
<th>ADDR+1</th>
<th>0xAA</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDR</td>
<td>0xBB</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ADDR+1</th>
<th>0xBB</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDR</td>
<td>0xAA</td>
<td></td>
</tr>
</tbody>
</table>
Arithmetic Operations

This group of instructions supports arithmetic operations on a variety of operands; 8- and 16-bit operations are supported directly and can easily be extended to support multiple-word operands. Two's-complement (signed) and binary (unsigned) operations are supported directly.

<table>
<thead>
<tr>
<th>Function</th>
<th>Mnemonic</th>
<th>IMM</th>
<th>DIR</th>
<th>EXT</th>
<th>INDX</th>
<th>INDY</th>
<th>INH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Accumulators</td>
<td>ABA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Add Accumulator B to X</td>
<td>ABX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Add Accumulator B to Y</td>
<td>ABY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Add with Carry to A</td>
<td>ADCA</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Add with Carry to B</td>
<td>ADCB</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Add Memory to A</td>
<td>ADDA</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Add Memory to B</td>
<td>ADDB</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Add Memory to D (16 Bit)</td>
<td>ADDD</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Compare A to B</td>
<td>CBA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Compare A to Memory</td>
<td>CMPA</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Compare B to Memory</td>
<td>CMPB</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Compare D to Memory (16 Bit)</td>
<td>CPD</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Decimal Adjust A (for BCD)</td>
<td>DAA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Decrement Memory Byte</td>
<td>DEC</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Decrement Accumulator A</td>
<td>DECA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Decrement Accumulator B</td>
<td>DECB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Increment Memory Byte</td>
<td>INC</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Increment Accumulator A</td>
<td>INCA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Increment Accumulator B</td>
<td>INCB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
Multi-Precision Addition

\[
\begin{array}{cccc}
\text{J3} & \text{J2} & \text{J1} & \text{J0} \\
\hline
\text{K3} & \text{K2} & \text{K1} & \text{K0} \\
\hline
\text{S3} & \text{S2} & \text{S1} & \text{S0}
\end{array}
\]

\[
\text{ldaa J0} \\
\text{ldab K0} \\
\text{aba} \\
\text{staa S0} \\
\text{ldaa J1} \\
\text{adca K1} \\
\text{staa S1} \\
\text{ldaa J2} \\
\text{adca K2} \\
\text{staa S2} \\
\text{ldaa J3} \\
\text{adca K3} \\
\text{staa S3}
\]

\[
\text{ldab S0} \\
\text{jsr print} \\
\text{ldab S1} \\
\text{jsr print} \\
\text{ldab S2} \\
\text{jsr print} \\
\text{ldab S3} \\
\text{jsr print}
\]

\text{print: clra}
\text{jsr CONSOLEINT}
\text{jsr OUTCRLF}
\text{rts}

\begin{verbatim}
.sect .data
J0: .byte 210 // 1234567890 =
J1: .byte 2  //  73  150  2  210
J2: .byte 150 // 01001001 10010110 00000010 11010010
J3: .byte 73  //
K0: .byte 177 // 887654321 =
K1: .byte 135 //  52  232  135  177
K2: .byte 232 // 00110100 11101000 10000111 10110001
K3: .byte 52  //
S0: .byte 0xFF // 1234567890 + 887654321 = 2122222211 =
S1: .byte 0xFF //  126  126  138  131
S2: .byte 0xFF // 01111110 01111110 10001010 10000011
S3: .byte 0xFF //
\end{verbatim}
Comparison Instructions

// example 1

AAA: .byte 0x11
BBB: .byte 0x22

ldaa AAA
ldab BBB
cba

// or

ldaa AAA
ldab BBB

cmpa BBB

// example 2: what does this code do?

AAA: .ascii "I love assembly!"
BBB: .byte 0x00

ldx #AAA
loop: ldaa 0, X
jsr OUTCHAR
inx
cpx #BBB
bne loop
More Arithmetic Operations

Compare instructions perform a subtract within the CPU to update the condition code bits without altering either operand. Although test instructions are provided, they are seldom needed since almost all other operations automatically update the condition code bits.
Test Instructions

- Set the CCR based on one single value, without changing the value tested

Example:

AAA: .ascii "I love assembly!"
BBB: .byte 0x00
END: .asciz "Done"

```
ldx #AAA
loop: ldaa 0, X
    jsr OUTCHAR
    inx
    tst 0, X
    bne loop
    jsr OUTCRLF
    ldx #END
    jsr OUTSTRING
```

What does this program do?
Multiply and Divide

Table 6-3. Multiply and Divide Instructions

<table>
<thead>
<tr>
<th>Function</th>
<th>Mnemonic</th>
<th>INH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply ( A \times B \rightarrow D )</td>
<td>MUL</td>
<td>X</td>
</tr>
<tr>
<td>Fractional Divide ( D + X \rightarrow X; r \rightarrow D )</td>
<td>FDIV</td>
<td>X</td>
</tr>
<tr>
<td>Integer Divide ( D + X \rightarrow X; r \rightarrow D )</td>
<td>IDIV</td>
<td>X</td>
</tr>
</tbody>
</table>

- 8 bits  8 bits = 16-bit result
- Integer divide (IDIV): 16 bits  16 bits = 16-bit result and 16-bit remainder
- Fractional divide (FDIV): 16 bits  16 bits = 16-bit result (a binary-weighted fraction between 0.000015 and 0.99998) and 16-bit remainder
Multiplication

J0 \times \mathbf{K0} = \mathbf{P1} \mathbf{P0}

AAA: \text{.byte} 0x03 \quad \text{// 3}
BBB: \text{.byte} 0x6F \quad \text{// 111}

\text{ldaa} \quad \text{AAA}
\text{ldab} \quad \text{BBB}
\text{mul}
\text{jsr} \quad \text{CONSOLEINT}
Integer Division

NNN: .word 100
DDD: .word 7
num: .asciz "Numerator:   
den: .asciz "Denominator: 
quo: .asciz "Quotient:    
rem: .asciz "Reminder:    

ldx #num
jsr OUTSTRING
ldd NNN
jsr CONSOLEINT
jsr OUTCRLF
ldx #den
jsr OUTSTRING
ldd DDD
jsr CONSOLEINT
jsr OUTCRLF
ldd NNN
ldx DDD
IDIV // X = D / X, D = D % X
xgdy // Y = reminder
xgdx // D = quotient
ldx #quo
jsr OUTSTRING
jsr CONSOLEINT
jsr OUTCRLF
xgdy // D = reminder
ldx #rem
jsr OUTSTRING
jsr CONSOLEINT
jsr OUTCRLF

Output:
Fractional Division

NNN:  .word  2
DDD:  .word  40
num:  .asciz "Numerator:   "
den:  .asciz "Denominator: "
quo:  .asciz "Quotient:    "
rem:  .asciz "Reminder:    "

ldx #num
jsr OUTSTRING
ldd NNN
jsr CONSOLEINT
jsr OUTCRLF
ldx #den
jsr OUTSTRING
ldd DDD
jsr CONSOLEINT
jsr OUTCRLF
ldd NNN
ldx DDD
FDIV           // X = D / X, D = D % X
xgdy           // Y = reminder
xgdx           // D = quotient
ldx #quo
jsr OUTSTRING
jsr CONSOLEINT
jsr OUTCRLF
xgdy           // D = reminder
ldx #rem
jsr OUTSTRING
jsr CONSOLEINT
jsr OUTCRLF

Tera Term Web 3.1 - COM1 YT

| Numerator: 2 |
| Denominator: 40 |
| Quotient: 3276 |
| Reminder: 32 |
Logical Operations

This group of instructions is used to perform the Boolean logical operations AND, inclusive OR, exclusive OR, and one’s complement.

<table>
<thead>
<tr>
<th>Function</th>
<th>Mnemonic</th>
<th>IMM</th>
<th>DIR</th>
<th>EXT</th>
<th>INDX</th>
<th>INDY</th>
<th>INH</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND A with Memory</td>
<td>ANDA</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>AND B with Memory</td>
<td>ANDB</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Bit(s) Test A with Memory</td>
<td>BITA</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Bit(s) Test B with Memory</td>
<td>BITB</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>One’s Complement Memory Byte</td>
<td>COM</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>One’s Complement A</td>
<td>COMA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>One’s Complement B</td>
<td>COMB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>OR A with Memory (Exclusive)</td>
<td>EORA</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>OR B with Memory (Exclusive)</td>
<td>EORB</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>OR A with Memory (Inclusive)</td>
<td>ORAA</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>OR B with Memory (Inclusive)</td>
<td>ORAB</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
Data Testing and Bit Manipulation

This group of instructions is used to operate on operands as small as a single bit, but these instructions can also operate on any combination of bits within any 8-bit location in the 64-Kbyte memory space.

Table 6-5. Data Testing and Bit Manipulation Instructions

<table>
<thead>
<tr>
<th>Function</th>
<th>Mnemonic</th>
<th>IMM</th>
<th>DIR</th>
<th>EXT</th>
<th>INDX</th>
<th>INDY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit(s) Test A with Memory</td>
<td>BITA</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Bit(s) Test B with Memory</td>
<td>BITB</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Clear Bit(s) in Memory</td>
<td>BCLR</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Set Bit(s) in Memory</td>
<td>BSET</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Branch if Bit(s) Clear</td>
<td>BRCLR</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Branch if Bit(s) Set</td>
<td>BRSET</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Setting Bits in Memory

AAA: .word 0x0002
bef: .asciz "Before: "
aft: .asciz "After:  

ldx #bef
jsr OUTSTRING
jsr OUTCRLF
ldd AAA
jsr CONSOLEINT
jsr OUTCRLF

ldx #AAA
bset 1, X, #0x09

ldx #aft
jsr OUTSTRING
jsr OUTCRLF
ldd AAA
jsr CONSOLEINT
jsr OUTCRLF

Tera Term Web 3.1 - COM1 VT

Before:
2
After:
11
Setting Bits In Memory: Endianness

AAA: .word 0x0002
bef: .asciz "Before: "
aft: .asciz "After: "

ldx #bef
jsr OUTSTRING
jsr OUTCRLF
ldd AAA
jsr CONSOLEINT
jsr OUTCRLF

ldx #AAA
bset 0, X, #0x09

ldx #aft
jsr OUTSTRING
jsr OUTCRLF
ldd AAA
jsr CONSOLEINT
jsr OUTCRLF

2306 = 00001001 00000010
### Shift and Rotate

All the shift and rotate functions in the M68HC11 CPU involve the carry bit in the CCR in addition to the 8- or 16-bit operand in the instruction, which permits easy extension to multiple-word operands.

<table>
<thead>
<tr>
<th>Function</th>
<th>Mnemonic</th>
<th>IMM</th>
<th>DM</th>
<th>EXT</th>
<th>INDX</th>
<th>INDY</th>
<th>INH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic Shift Left Memory</td>
<td>ASL</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arithmetic Shift Left A</td>
<td>ASLA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Arithmetic Shift Left B</td>
<td>ASLB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Arithmetic Shift Left Double</td>
<td>ASLD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Arithmetic Shift Right Memory</td>
<td>ASR</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arithmetic Shift Right A</td>
<td>ASRA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Arithmetic Shift Right B</td>
<td>ASRB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>(Logical Shift Left Memory)</td>
<td>(LSL)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Logical Shift Left A)</td>
<td>(LSLA)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>(Logical Shift Left B)</td>
<td>(LSLB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>(Logical Shift Left Double)</td>
<td>(LSLD)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Logical Shift Right Memory</td>
<td>LSR</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical Shift Right A</td>
<td>LSRA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Logical Shift Right B</td>
<td>LSRB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Logical Shift Right D</td>
<td>LSRD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Rotate Left Memory</td>
<td>ROL</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rotate Left A</td>
<td>ROLA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Rotate Left B</td>
<td>ROLB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Rotate Right Memory</td>
<td>ROR</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rotate Right A</td>
<td>RORAA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Rotate Right B</td>
<td>RORB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
Shift and Rotate

AAA: .byte 0x0F
BBB: .byte 0x8F
bef: .asciz "Before: 

ldx #bef
jsr OUTSTRING
jsr OUTCRLF
clra
ldab AAA
jsr CONSOLEINT
jsr OUTCRLF

ldaa AAA
ASRA

ldx #aft
jsr OUTSTRING
jsr OUTCRLF
tab
cира
jsr CONSOLEINT
jsr OUTCRLF

ldaa BBB
ASRA

// print A
Stack and index register instructions

- Remember that HC11 is *big endian*
- Stack instructions do not affect the condition codes
# Stack and Index Register Instructions

This table summarizes the instructions available for the 16-bit index registers (X and Y) and the 16-bit stack pointer.

<table>
<thead>
<tr>
<th>Function</th>
<th>Mnemonic</th>
<th>IMM</th>
<th>DIR</th>
<th>EXT</th>
<th>INDX</th>
<th>INDY</th>
<th>INH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Accumulator B to X</td>
<td>ABX</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Add Accumulator B to Y</td>
<td>ABY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compare X to Memory (16 Bit)</td>
<td>CPX</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Compare Y to Memory (16 Bit)</td>
<td>CPY</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Decrement Stack Pointer</td>
<td>DES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Decrement Index Register X</td>
<td>DEX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Decrement Index Register Y</td>
<td>DEY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Increment Stack Pointer</td>
<td>INS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Increment Index Register X</td>
<td>INX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Increment Index Register Y</td>
<td>INY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Load Index Register X</td>
<td>LDX</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Load Index Register Y</td>
<td>LDY</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Load Stack Pointer</td>
<td>LDS</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Pull X from Stack</td>
<td>PULX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Pull Y from Stack</td>
<td>PULY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Push X onto Stack</td>
<td>PSHX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Push Y onto Stack</td>
<td>PSHY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Store Index Register X</td>
<td>STX</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Store Index Register Y</td>
<td>STY</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Store Stack Pointer</td>
<td>STS</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Transfer SP to X</td>
<td>TSX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Transfer SP to Y</td>
<td>TSY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Transfer X to SP</td>
<td>TXS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Transfer Y to SP</td>
<td>TYS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Exchange D with X</td>
<td>XGDX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Exchange D with Y</td>
<td>XGDY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
**XGDX and XGDY**

- Bidirectional exchanges: the original content of D is not changed while using the double accumulator to process X or Y.

```
  D  <->  X

  D  <->  Y
```

The stack pointer \( SP \) always points to the \textit{next free location} on the stack.

- Automatically incremented on \( TSX \) and \( TSY \)
- Automatically decremented on \( TXS \) and \( TSY \).

Example:

\begin{align*}
\text{TSX} & \quad \text{INX} \\
\text{INX} & \quad \text{X} = 0x6FF \\
\text{TXS} & \quad \text{SP} = 0x6FF 5 \\
& \quad \text{SP} = 0x6FF 6 \\
& \quad \text{SP} = 0x6FF 7 \\
& \quad \text{SP} = 0x6FF 8 \\
& \quad \text{SP} = 0x6FF 9 \\
& \quad \text{SP} = 0x6FF A \\
& \quad \text{SP} = 0x6FF B \\
& \quad \text{SP} = 0x6FF C \\
\end{align*}
Condition Code Register Instructions

- These instructions allow a programmer to manipulate bits of the condition code register (CCR)

<table>
<thead>
<tr>
<th>Function</th>
<th>Mnemonic</th>
<th>INH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear Carry Bit</td>
<td>CLC</td>
<td>X</td>
</tr>
<tr>
<td>Clear Interrupt Mask Bit</td>
<td>CLI</td>
<td>X</td>
</tr>
<tr>
<td>Clear Overflow Bit</td>
<td>CLV</td>
<td>X</td>
</tr>
<tr>
<td>Set Carry Bit</td>
<td>SEC</td>
<td>X</td>
</tr>
<tr>
<td>Set Interrupt Mask Bit</td>
<td>SEI</td>
<td>X</td>
</tr>
<tr>
<td>Set Overflow Bit</td>
<td>SEV</td>
<td>X</td>
</tr>
<tr>
<td>Transfer A to CCR</td>
<td>TAP</td>
<td>X</td>
</tr>
<tr>
<td>Transfer CCR to A</td>
<td>TPA</td>
<td>X</td>
</tr>
</tbody>
</table>
Program Control Instructions

- Branches
- Jumps
- Subroutine calls and returns
- Interrupt handling
- Miscellaneous
Branches

These instructions allow the CPU to make decisions based on the contents of the condition code bits. All decision blocks in a flow chart would correspond to one of the conditional branch instructions summarized here.
Far branches

- If the branch destination is farther than 128 or +127 bytes, then must use a branch + jump combo

- Example:

  BHI TINBUK2  // Out of range

  Becomes

  BLS AROUND
  JMP TIMBUK2

  AROUND:
Jump

- The jump instruction allows control to be passed to any address in the 64-Kbyte memory map.

Table 6-10. Jump Instruction

<table>
<thead>
<tr>
<th>Function</th>
<th>Mnemonic</th>
<th>DIR</th>
<th>EXT</th>
<th>INDX</th>
<th>INDY</th>
<th>INH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jump</td>
<td>JMP</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Subroutine Calls and Returns

The CPU automates the process of remembering the address in the main program where processing should resume after the subroutine is finished. This address is automatically pushed onto the stack when the subroutine is called and is pulled off the stack during the **RTS** instruction that ends the subroutine.

### Table 6-11. Subroutine Call and Return Instructions

<table>
<thead>
<tr>
<th>Function</th>
<th>Mnemonic</th>
<th>REL</th>
<th>DIR</th>
<th>EXT</th>
<th>INDX</th>
<th>INDY</th>
<th>INH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch to Subroutine</td>
<td>BSR</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jump to Subroutine</td>
<td>JSR</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Return from Subroutine</td>
<td>RTS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

The table above lists the instructions used for subroutine calls and returns. BSR is used for branch to subroutine, JSR for jump to subroutine, and RTS for return from subroutine.
Interrupt Handling

- **SWI** is like **JSR** but pushes all registers onto the stack
- **RTI** pops them off the stack
- **WAI** pushes all registers and waits in low-power mode

Table 6-12. Interrupt Handling Instructions

<table>
<thead>
<tr>
<th>Function</th>
<th>Mnemonic</th>
<th>INH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return from Interrupt</td>
<td>RTI</td>
<td>X</td>
</tr>
<tr>
<td>Software Interrupt</td>
<td>SWI</td>
<td>X</td>
</tr>
<tr>
<td>Wait for Interrupt</td>
<td>WAI</td>
<td>X</td>
</tr>
</tbody>
</table>
Interrupt ex: ledClockInt (1/3)

/*********************************************************
* binary clock on the LED of the HC11 kit
* reset the clock with a flash when the interrupt is pressed
* andrea di blas
*********************************************************/
#include "v2_18g3.asm"

.sect .data
tick: .byte 0x00 // current second
/***********************************************************/
.sect .text
main:
/**** load ISR ****/
    ldd #CLOCKRESET
    ldx #ISR_JUMP15
    std 0, x
Interrupt vectors

// Ram isr image table jump addresses. Note the table is ordinal and that some // vectors are not available to the user. These are used or reserved for the // system and have not been mapped from the corresponding ROM table. // See Motorola documentation for discussion of these vectors.
//
// 1       unavailable to user      SCI
#define ISR_JUMP2 0x7bc5 // SPI
#define ISR_JUMP3 0x7bc8 // Pulse Accumulator Input
#define ISR_JUMP4 0x7bcb // Pulse Accumulator Overflow
// 5       unavailable to user      Timer Overflow
// 6       unavailable to user      Output Compare 5
#define ISR_JUMP7 0x7bd4 // Output Compare 4
#define ISR_JUMP8 0x7bd7 // Output Compare 3
#define ISR_JUMP9 0x7bda // Output Compare 2
#define ISR_JUMP10 0x7bde // Output Compare 1
#define ISR_JUMP11 0x7be3 // Input Capture 3
#define ISR_JUMP12 0x7be6 // Input Capture 2
#define ISR_JUMP13 0x7be9 // Input Capture 1
// 14      unavailable to user      Real Time Interrupt
#define ISR_JUMP15 0x7bec // IRQ
// 16      unavailable to user      XIRQ
// 17      unavailable to user      SWI
// 18      unavailable to user      Illegal Opcode
#define ISR_JUMP19 0x7bf8 // Cop fail
#define ISR_JUMP20 0x7bfb // Cop clock fail
// 21      unavailable to user      Reset (found at 0x8040)

(from v2_18g3.asm)
Figure 10-1. Main Timer System Block Diagram
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>Internal SRAM</td>
</tr>
<tr>
<td>0x00FF</td>
<td></td>
</tr>
<tr>
<td>0x0100</td>
<td>4k System Memory</td>
</tr>
<tr>
<td>0x0FFF</td>
<td></td>
</tr>
<tr>
<td>0x1000</td>
<td>User Area</td>
</tr>
<tr>
<td>0x7BC0</td>
<td></td>
</tr>
<tr>
<td>0x7BC1</td>
<td>User Interrupt vector jump table</td>
</tr>
<tr>
<td>0x7BFF</td>
<td></td>
</tr>
<tr>
<td>0x7C00</td>
<td>I/O ports</td>
</tr>
<tr>
<td>0x7FFF</td>
<td></td>
</tr>
<tr>
<td>0x8000</td>
<td>Internal 64-byte register block</td>
</tr>
<tr>
<td>0x803F</td>
<td></td>
</tr>
<tr>
<td>0x8040</td>
<td>External ROM (system code)</td>
</tr>
<tr>
<td>0xFFFF</td>
<td></td>
</tr>
</tbody>
</table>

#define ISR_JUMP15 0x7bec // IRQ
ledClockInt.asm (2/3)

/**** start program ****/

clra

ldx #LEDS

loop: ldaa tick

coma

staa 0, X

coma

inca

staa tick

ldd #250

jsr WAIT

jmp loop

/*****************************/
CLOCKRESET:
    pshx
    ldx #LEDS
    clra
    staa 0, X  // flash the LEDs
    cli       // re-enable interrupts (int disables)
    ldd #200  // keep LEDs on for a bit
    jsr WAIT
    clra
    staa tick  // clear seconds counter
    rti

/*********************************************************

/
Miscellaneous Instructions

- **NOP** is a do nothing instruction, just wastes an instruction cycle.
- **STOP** is used to turn off the oscillator and put the CPU into a low power mode. Only reset or interrupts can wake it up again.
- **TEST** is a reserved instruction only used at the factory when making the chips (illegal opcode).

<table>
<thead>
<tr>
<th>Function</th>
<th>Mnemonic</th>
<th>INH</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Operation (2-cycle delay)</td>
<td>NOP</td>
<td>X</td>
</tr>
<tr>
<td>Stop Clocks</td>
<td>STOP</td>
<td>X</td>
</tr>
<tr>
<td>Test</td>
<td>TEST</td>
<td>X</td>
</tr>
</tbody>
</table>