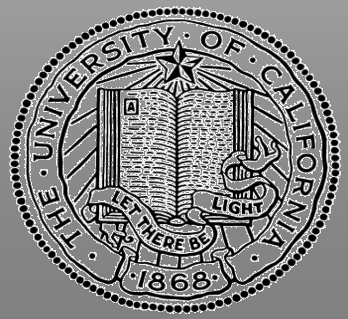
HC11 Instruction Set



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

LST

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 6-1. Load, Store, and Transfer Instructions

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

LST Example

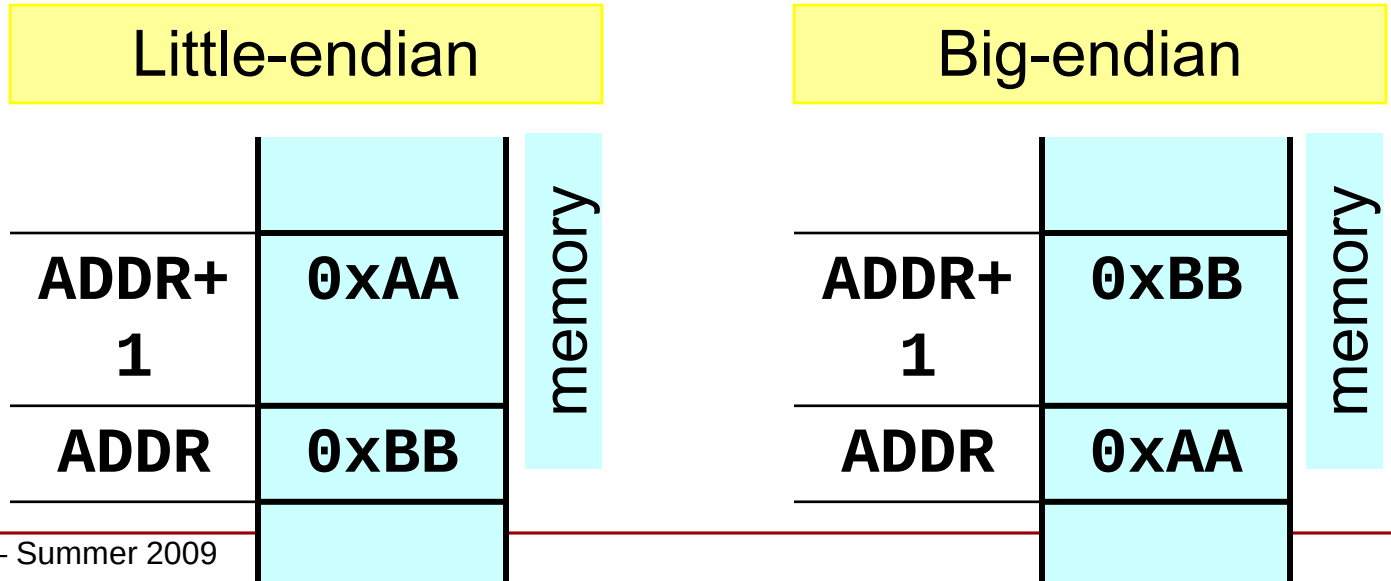
```
AAA:    .asciz "I love assembly language!“

        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
```

What does this program do?

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**



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. Twos-complement (signed) and binary (unsigned) operations are supported directly.

Table 6-2. Arithmetic Operation Instructions

Function	Mnemonic	IMM	DIR	EXT	INDX	INDY	INH
Add Accumulators	ABA						X
Add Accumulator B to X	ABX						X
Add Accumulator B to Y	ABY						X
Add with Carry to A	ADCA	X	X	X	X	X	
Add with Carry to B	ADCB	X	X	X	X	X	
Add Memory to A	ADDA	X	X	X	X	X	
Add Memory to B	ADDB	X	X	X	X	X	
Add Memory to D (16 Bit)	ADDD	X	X	X	X	X	
Compare A to B	CBA						X
Compare A to Memory	CMPA	X	X	X	X	X	
Compare B to Memory	CMPB	X	X	X	X	X	
Compare D to Memory (16 Bit)	CPD	X	X	X	X	X	
Decimal Adjust A (for BCD)	DAA						X
Decrement Memory Byte	DEC			X	X	X	
Decrement Accumulator A	DECA						X
Decrement Accumulator B	DECB						X
Increment Memory Byte	INC			X	X	X	
Increment Accumulator A	INCA						X
Increment Accumulator B	INCB						X

Multi-Precision Addition

J3	J2	J1	J0	+
K3	K2	K1	K0	=
S3	S2	S1	S0	

```
.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  //
```

```
        ldaa    J0
        ldab    K0
        aba
        staa    S0
        ldaa    J1
        adca    K1
        staa    S1
        ldaa    J2
        adca    K2
        staa    S2
        ldaa    J3
        adca    K3
        staa    S3

        ldab    S0
        jsr     print
        ldab    S1
        jsr     print
        ldab    S2
        jsr     print
        ldab    S3
        jsr     print

print:  clra
        jsr     CONSOLEINT
        jsr     OUTCRLF
        rts
```

Comparison Instructions

```
// example 1

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

   ldaa      AAA
   ldab      BBB
   cba

// or
   ldaa      AAA
   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.

Table 6-2. Arithmetic Operation Instructions (Continued)

Function	Mnemonic	IMM	DIR	EXT	INDX	INDY	INH
Twos Complement Memory Byte	NEG			X	X	X	
Twos Complement Accumulator A	NEGA						X
Twos Complement Accumulator B	NEGB						X
Subtract with Carry from A	SBCA	X	X	X	X	X	
Subtract with Carry from B	SBCB	X	X	X	X	X	
Subtract Memory from A	SUBA	X	X	X	X	X	
Subtract Memory from B	SUBB	X	X	X	X	X	
Subtract Memory from D (16 Bit)	SUBD	X	X	X	X	X	
Test for Zero or Minus	TST			X	X	X	
Test for Zero or Minus A	TSTA						X
Test for Zero or Minus B	TSTB						X

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

Function	Mnemonic	INH
Multiply (A × B ⇒ D)	MUL	X
Fractional Divide (D ÷ X ⇒ X; r ⇒ D)	FDIV	X
Integer Divide (D ÷ X ⇒ X; r ⇒ D)	IDIV	X

- 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	x
	K0	=
P1	P0	

```
AAA:    .byte  0x03   // 3
BBB:    .byte  0x6F   // 111


        ldaa   AAA
        ldab   BBB
        mul
        jsr            CONSOLEINT
```

Integer Division

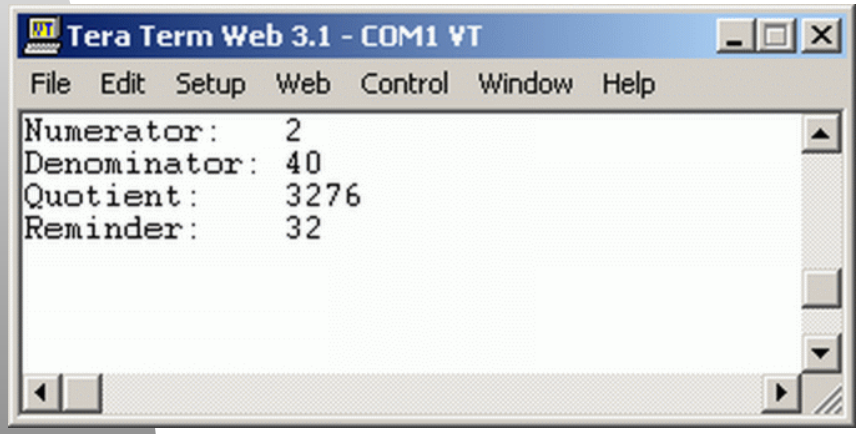
```
NNN:    .word   100
DDD:    .word   7
num:    .asciz  "Numerator:   "
den:    .asciz  "Denominator: "
quo:    .asciz  "Quotient:    "
rem:    .asciz  "Reminder:    "
```

Output:

```
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
```

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
```

Logical Operations

Table 6-4. Logical Operation Instructions

Function	Mnemonic	IMM	DIR	EXT	INDX	INDY	INH
AND A with Memory	ANDA	X	X	X	X	X	
AND B with Memory	ANDB	X	X	X	X	X	
Bit(s) Test A with Memory	BITA	X	X	X	X	X	
Bit(s) Test B with Memory	BITB	X	X	X	X	X	
One's Complement Memory Byte	COM			X	X	X	
One's Complement A	COMA						X
One's Complement B	COMB						X
OR A with Memory (Exclusive)	EORA	X	X	X	X	X	
OR B with Memory (Exclusive)	EORB	X	X	X	X	X	
OR A with Memory (Inclusive)	ORAA	X	X	X	X	X	
OR B with Memory (Inclusive)	ORAB	X	X	X	X	X	

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

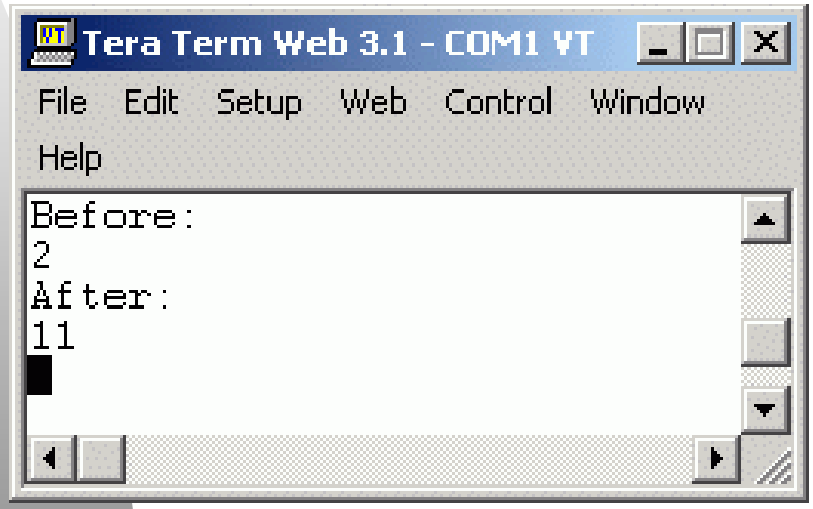
Data Testing and Bit Manipulation

Table 6-5. Data Testing and Bit Manipulation Instructions

Function	Mnemonic	IMM	DIR	EXT	INDX	INDY
Bit(s) Test A with Memory	BITA	X	X	X	X	X
Bit(s) Test B with Memory	BITB	X	X	X	X	X
Clear Bit(s) in Memory	BCLR		X		X	X
Set Bit(s) in Memory	BSET		X		X	X
Branch if Bit(s) Clear	BRCLR		X		X	X
Branch if Bit(s) Set	BRSET		X		X	X

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.

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
```

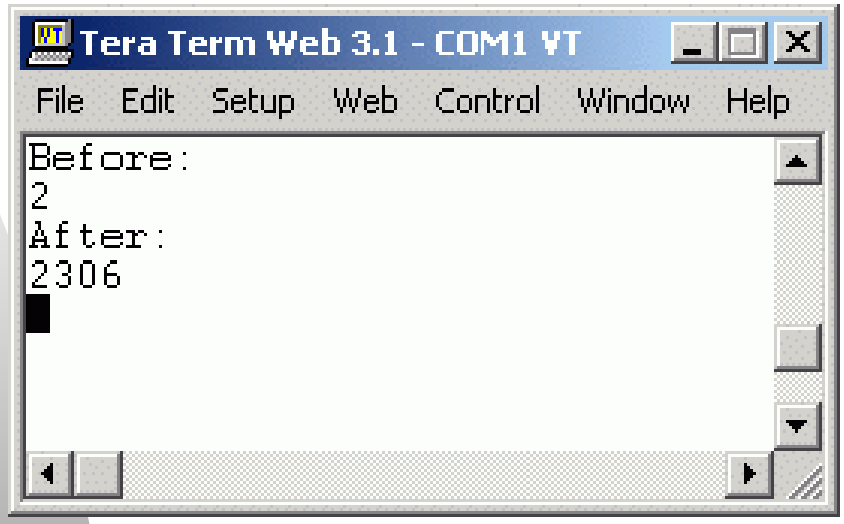
Setting Bits In Memory: Endianness

```
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

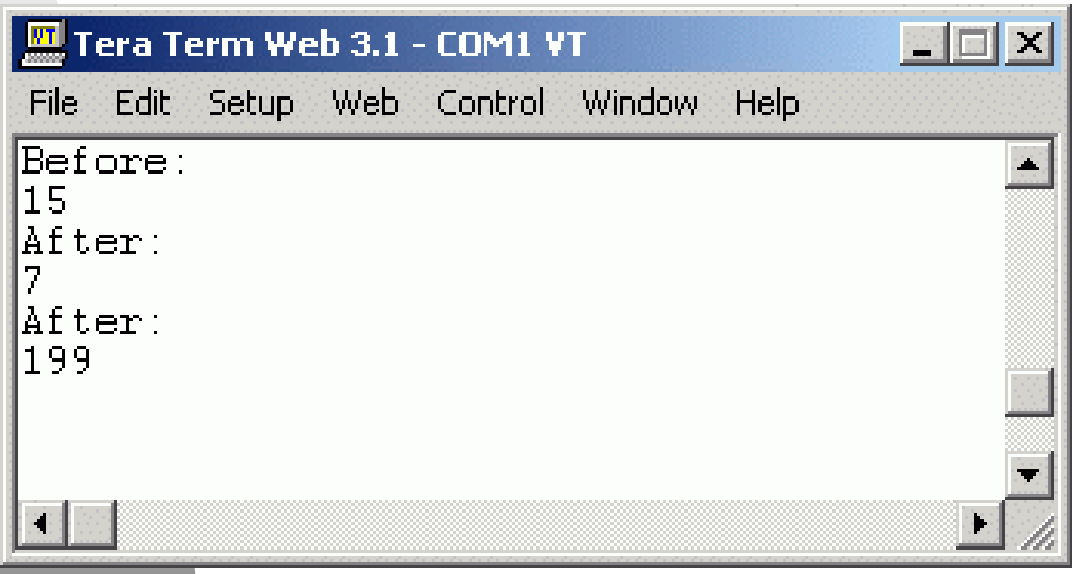
All the shift and rotate functions in the M68HC11 CPU involve the carry bit in the CCR in addition to the 8- or 16-bit operand in the instruction, which permits easy extension to multiple-word operands.

Table 6-6. Shift and Rotate Instructions

Function	Mnemonic	IMM	DM	EXT	INDX	INDY	INH
Arithmetic Shift Left Memory	ASL			X	X	X	
Arithmetic Shift Left A	ASLA						X
Arithmetic Shift Left B	ASLB						X
Arithmetic Shift Left Double	ASLD						X
Arithmetic Shift Right Memory	ASR			X	X	X	
Arithmetic Shift Right A	ASRA						X
Arithmetic Shift Right B	ASRB						X
(Logical Shift Left Memory)	(LSL)			X	X	X	
(Logical Shift Left A)	(LSLA)						X
(Logical Shift Left B)	(LSLB)						X
(Logical Shift Left Double)	(LSLD)						X
Logical Shift Right Memory	LSR			X	X	X	
Logical Shift Right A	LSRA						X
Logical Shift Right B	LSRB						X
Logical Shift Right D	LSRD						X
Rotate Left Memory	ROL			X	X	X	
Rotate Left A	ROLA						X
Rotate Left B	ROLB						X
Rotate Right Memory	ROR			X	X	X	
Rotate Right A	RORA						X
Rotate Right B	RORB						X

Shift and Rotate

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




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

ldaa AAA
ASRA

ldx  #aft
jsr  OUTSTRING
jsr  OUTCRLF
tab
clra
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

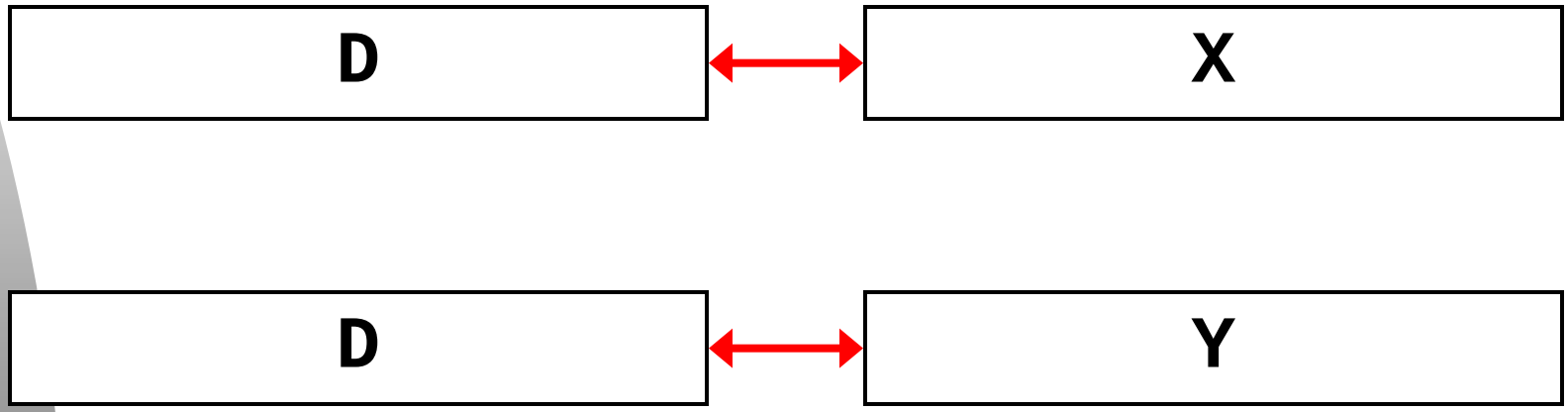
This table summarizes the instructions available for the 16-bit index registers (X and Y) and the 16-bit stack pointer.

Table 6-7. Stack and Index Register Instructions

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

XGDX and XGDY

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

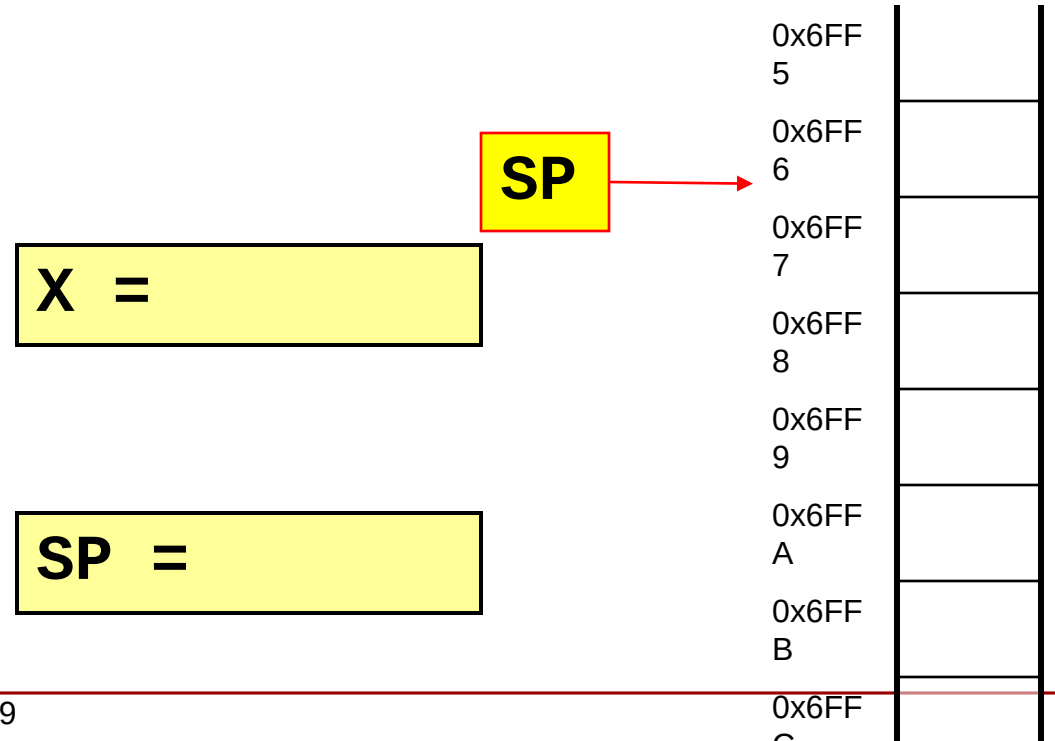


TSX, TSY, TXS, TYS

- The stack pointer **SP** always points to the ***next free location*** on the stack.
- Automatically incremented on **TSX** and **TSY**
- Automatically decremented on **TXS** and **TSY**.
- Example:

```
TSX
INX
INX

TXS
```



Condition Code Register Instructions

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

Table 6-8. Condition Code Register Instructions

Function	Mnemonic	INH
Clear Carry Bit	CLC	X
Clear Interrupt Mask Bit	CLI	X
Clear Overflow Bit	CLV	X
Set Carry Bit	SEC	X
Set Interrupt Mask Bit	SEI	X
Set Overflow Bit	SEV	X
Transfer A to CCR	TAP	X
Transfer CCR to A	TPA	X

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.

Table 6-9. Branch Instructions

Function	Mnemonic	REL	DIR	INDX	INDY	Comments
Branch if Carry Clear	BCC	X				C = 0 ?
Branch if Carry Set	BCS	X				C = 1 ?
Branch if Equal Zero	BEQ	X				Z = 1 ?
Branch if Greater Than or Equal	BGE	X				Signed ≥
Branch if Greater Than	BGT	X				Signed >
Branch if Higher	BHI	X				Unsigned >
Branch if Higher or Same (same as BCC)	BHS	X				Unsigned ≥
Branch if Less Than or Equal	BLE	X				Signed ≤
Branch if Lower (same as BCS)	BLO	X				Unsigned <
Branch if Lower or Same	BLS	X				Unsigned ≤
Branch if Less Than	BLT	X				Signed <
Branch if Minus	BMI	X				N = 1 ?
Branch if Not Equal	BNE	X				Z = 0 ?
Branch if Plus	BPL	X				N = 0 ?
Branch if Bit(s) Clear in Memory Byte	BRCLR		X	X	X	Bit Manipulation
Branch Never	BRN	X				3-cycle NOP
Branch if Bit(s) Set in Memory Byte	BRSET		X	X	X	Bit Manipulation
Branch if Overflow Clear	BVC	X				V = 0 ?
Branch if Overflow Set	BVS	X				V = 1 ?

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

Function	Mnemonic	DIR	EXT	INDX	INDY	INH
Jump	JMP	X	X	X	X	

Subroutine Calls and Returns

Table 6-11. Subroutine Call and Return Instructions

Function	Mnemonic	REL	DIR	EXT	INDX	INDY	INH
Branch to Subroutine	BSR	X					
Jump to Subroutine	JSR		X	X	X	X	
Return from Subroutine	RTS						X

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

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

Function	Mnemonic	INH
Return from Interrupt	RTI	X
Software Interrupt	SWI	X
Wait for Interrupt	WAI	X

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**)

Main timer and RTI diagram

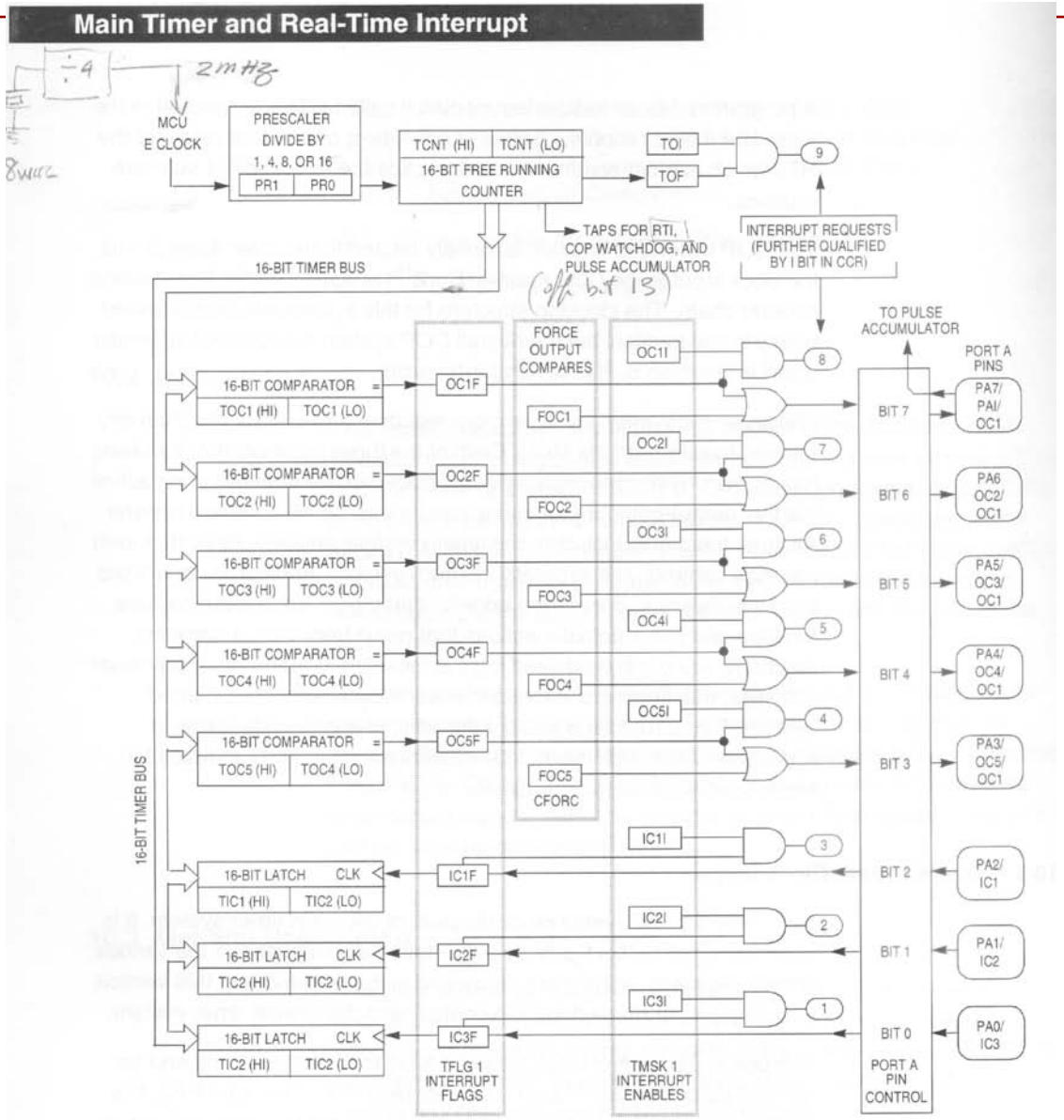
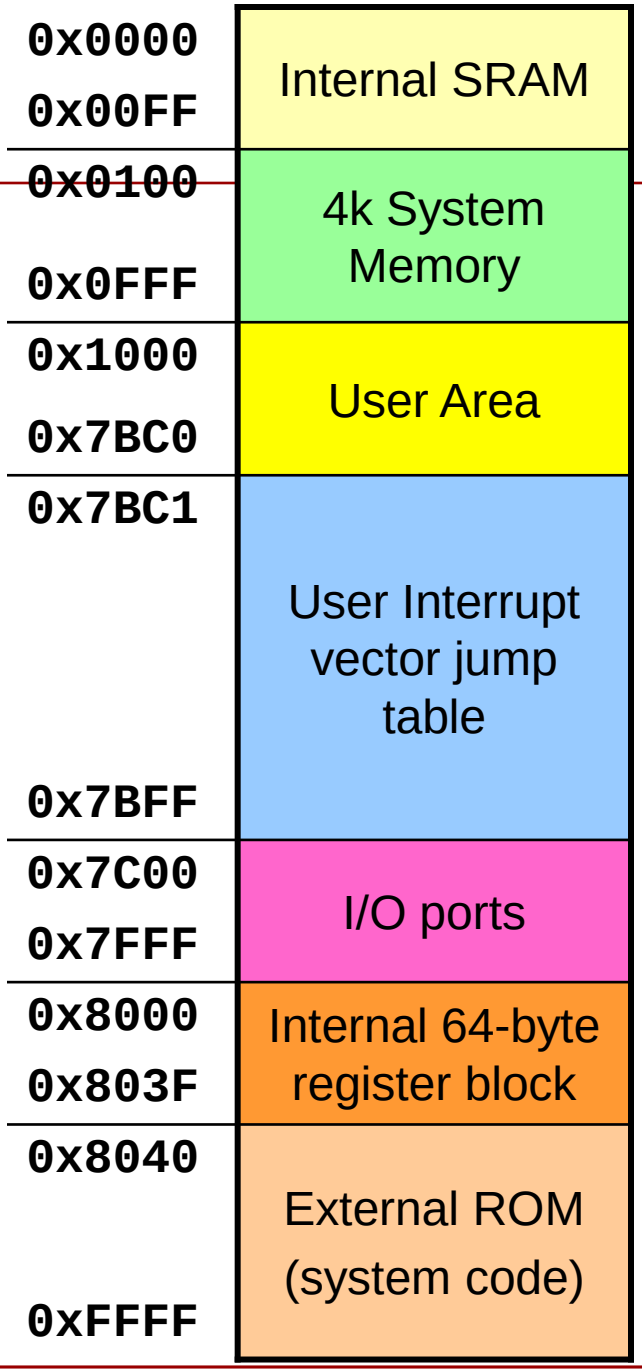


Figure 10-1. Main Timer System Block Diagram

HC11 Kit Memory map




```
#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
/*****************************************************/
```

ledClockInt.asm (3/3)

```
/* my ISR to reset the clock with a flash! */
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 6-13. Miscellaneous Instructions

Function	Mnemonic	INH
No Operation (2-cycle delay)	NOP	X
Stop Clocks	STOP	X
Test	TEST	X