# Module Introduction

**PURPOSE:**

The intent of this module is to present all CPU12 instructions to better prepare you to write short routines in assembly language.

**OBJECTIVES:**

- Discuss all CPU12 instruction sets.
- Describe data handling instructions.
- Identify arithmetic instructions.
- Describe logic instructions.
- Identify data test instructions.
- Describe branch instructions.
- Discuss jump and subroutine calls.
- Identify the HCS12 CPU features that support high-level language programs.

**CONTENT:**

- 31 pages
- 6 questions

**LEARNING TIME:**

- 60 minutes

The intent of this module is to present all CPU12 instructions to better prepare you to write short routines in assembly language. You will become more familiar with Data Handling, Arithmetic, Logic, Data Test, Branch, and finally, Jump and Subroutine Calls. You will learn about the HCS12 CPU features that support high-level language programs and you will be better prepared to write these programs.

## HCS12 Instruction set

- Superset of the M68HC11 instruction set
- CPU12 implementations:
  - the original M68HC12
  - and the newer HCS12
- All memory and I/O are mapped in a common 64-Kbyte address space
- Full set of 8-bit and 16-bit mathematical instructions

Let's start with the HCS12 instruction set. CPU12 instructions are a superset of the M68HC11 instruction set. Code written for an M68HC11 can be reassembled and runs on a CPU12 with no changes. The CPU12 provides expanded functionality and increased code efficiency.

There are two implementations of the CPU12, the original M68HC12 and the newer HCS12. Both implementations have the same instruction set, although there are small differences in cycle-by-cycle access details. For example: the order of some bus cycles changed to accommodate differences in the way the instruction queue was implemented. These minor differences are transparent for most users.

In the M68HC12 and HCS12 architecture, all memory and input/output (I/O) are mapped in a common 64-Kbyte address space (memory-mapped I/O). This allows the same set of instructions to be used to access memory, I/O, and control registers. General-purpose load, store, transfer, exchange, and move instructions facilitate movement of data to and from memory and peripherals.

The CPU12 has a full set of 8-bit and 16-bit mathematical instructions. There are instructions for signed and unsigned arithmetic, division, and multiplication with 8-bit, 16-bit, and some larger operands. Special arithmetic and logic instructions aid stacking operations, indexing, binary-coded decimal (BCD) calculation, and condition code register manipulation. There is also dedicated instruction for multiply and accumulate operation, table interpolation, and specialized fuzzy logic operations that involve mathematical calculations.

# Instruction Set

- Data Handling
- Arithmetic
- Logic
- Data Test
- Branch
- Jump & Subroutine Calls

To reference the information in the CPU12 Users Manual,
see the document CPU12RM/AD or go to www.Freescale.com.

Most instructions are standard 68xx, but some new and efficient opcodes have been added. Here, we will briefly cover the CPU12 instruction set to help you better understand the program examples during this module.

Let's look at the various instruction types that the CPU12 supports. The data handling instruction type includes Loads, Stores, Pulls, Push, Transfers, INC, DEC, Rotates, and Shifts.

The arithmetic instructions include ADD, SUB, and MULT.

The logic instructions include AND, OR, and EOR.

The data test instructions include Bit Test and Compare.

The branch instructions include conditional branches such as Branch if not equal,(BNE), Branch if higher, (BHI) and many more conditional branches.

The HCS12 allows the program to jump to any memory address during program execution, and the jump to subroutine instruction may be used to call a routine that needs to be executed on periodic bases.

To reference the information in the CPU12 Users Manual, see the document CPU12RM/AD or go to the web site listed here.

## Load and Store Instructions

| Mnemonic | Function | Operation |
|---|---|---|
| | **Load Instructions** | |
| LDAA | Load A | (M) ⇒ A |
| LDAB | Load B | (M) ⇒ B |
| LDD | Load D | (M : M + 1) ⇒ (A:B) |
| LDS | Load SP | (M : M + 1) ⇒ $SP_H$:$SP_L$ |
| LDX | Load index register X | (M : M + 1) ⇒ $X_H$:$X_L$ |
| LDY | Load index register Y | (M : M + 1) ⇒ $Y_H$:$Y_L$ |
| LEAS | Load effective address into SP | Effective address ⇒ SP |
| LEAX | Load effective address into X | Effective address ⇒ X |
| LEAY | Load effective address into Y | Effective address ⇒ Y |
| | **Store Instructions** | |
| STAA | Store A | (A) ⇒ M |
| STAB | Store B | (B) ⇒ M |
| STD | Store D | (A) ⇒ M, (B) ⇒ M + 1 |
| STS | Store SP | ($SP_H$:$SP_L$) ⇒ M : M + 1 |
| STX | Store X | ($X_H$:$X_L$) ⇒ M : M + 1 |
| STY | Store Y | ($Y_H$:$Y_L$) ⇒ M : M + 1 |

Load instructions copy memory content into an accumulator or register. The memory content is not changed by the operation. Load instructions (excluding LEA_ instructions) affect condition code bits. As a result, no separate test instructions are needed to check the loaded values for negative or 0 conditions. The LEA instruction calculates an effective address and puts it into a destination register.

For example: LEAX B and Y adds register B to register Y and puts the sum into register X. Neither register B or Y contents are affected by this operation. If register Y equals $1000 and register B equals $25, the resulting value in register X will be $1025.

Store instructions copy the content of a CPU register to memory. Register and accumulator content is not changed by the operation. Store instructions automatically update the N and Z condition code bits, which can eliminate the need for a separate test instruction in some programs.

## Data Move Instructions

- **Moves the content of one memory location to another memory location.**
- **The content of the source memory location is not changed.**

**EXAMPLE:**

```
MOVW   2,X+ , 2,-Y
MOVB   1,Y+, 1,X+
MOVB   2,X+, 2,Y-
MOVW   2,X+, 2,-SP
MOVW   2,X+, 2,Y+
```

The function of the data move instructions is to move the content of one memory location to another memory location. As a result, the content of the source memory location is not changed.

Move instructions use separate addressing modes to access the source and destination of a move operation. The following combinations of addressing modes are supported: IMM–EXT, IMM–IDX, EXT–EXT, EXT–IDX,IDX–EXT, and IDX–IDX. IDX operands allow indexed addressing mode specifications that fit in a single post byte including 5-bit constant, accumulator offsets, and auto increment/decrement modes. Nine-bit and 16-bit constant offsets would require additional extension bytes and are not allowed.

The Move instructions support Auto increment, Pre-Decrement, Post Increment and post Decrement addressing modes as shown in the example above

# Data Movement

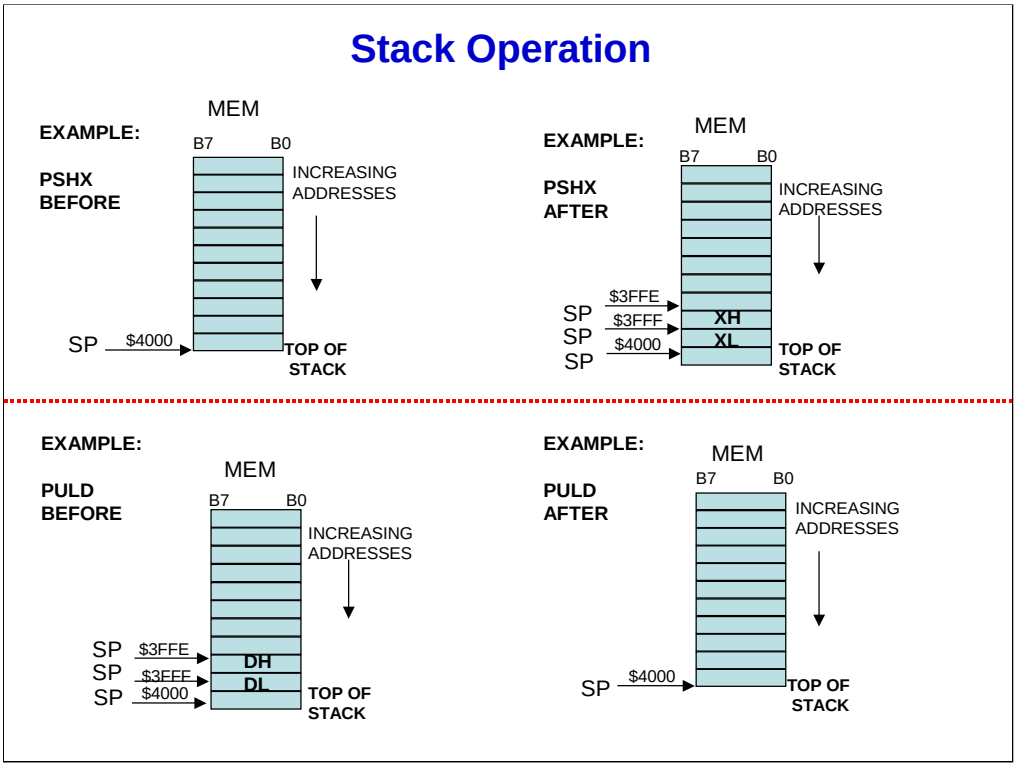
### Data Handling Instructions

| Mnemonic | Function | Operation |
|---|---|---|
| PSHA | Push A | $(SP) - 1 \Rightarrow SP; (A) \Rightarrow M_{(SP)}$ |
| PSHB | Push B | $(SP) - 1 \Rightarrow SP; (B) \Rightarrow M_{(SP)}$ |
| PSHC | Push CCR | $(SP) - 1 \Rightarrow SP; (A) \Rightarrow M_{(SP)}$ |
| PSHD | Push D | $(SP) - 2 \Rightarrow SP; (A : B) \Rightarrow M_{(SP)} : M_{(SP+1)}$ |
| PSHX | Push X | $(SP) - 2 \Rightarrow SP; (X) \Rightarrow M_{(SP)} : M_{(SP+1)}$ |
| PSHY | Push Y | $(SP) - 2 \Rightarrow SP; (Y) \Rightarrow M_{(SP)} : M_{(SP+1)}$ |
| PULA | Pull A | $(M_{(SP)}) \Rightarrow A; (SP) + 1 \Rightarrow SP$ |
| PULB | Pull B | $(M_{(SP)}) \Rightarrow B; (SP) + 1 \Rightarrow SP$ |
| PULC | Pull CCR | $(M_{(SP)}) \Rightarrow CCR; (SP) + 1 \Rightarrow SP$ |
| PULD | Pull D | $(M_{(SP)} : M_{(SP+1)}) \Rightarrow A : B; (SP) + 2 \Rightarrow SP$ |
| PULX | Pull X | $(M_{(SP)} : M_{(SP+1)}) \Rightarrow X; (SP) + 2 \Rightarrow SP$ |
| PULY | Pull Y | $(M_{(SP)} : M_{(SP+1)}) \Rightarrow Y; (SP) + 2 \Rightarrow SP$ |

Now, let’s look at Push and Pull instructions.

Push instructions save the content of a register onto the stack area. The Stack Pointer (SP) is decremented by one or two depending on whether the size of the register is 8 or 16 bits, respectively. The content of the register is then stored at the address where the SP points. Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine. The Push operation does not affect the Condition Code Register (CCR).

Pull instructions are commonly used at the end of a subroutine to restore the contents of CPU registers that were pushed onto the stack before the subroutine execution. The Pull instructions do not affect the CCR unless it is pulled.




Here, you can see the Push operation depicted. Note that the SP is decremented by two and then the X register is pushed onto the stack.

The Pull operation works exactly the opposite of Push. On the HCS12DP256, the RAM ends at location $3FFF. Putting the SP to $4000 is usually a good choice since the stack grows toward lower memory addresses.

## Question

**What type of addressing should you use with PSHA? Select the correct response and then click Done.**

**a. Extended Addressing**
**b. Inherent Addressing**
**c. Indexed Addressing**

You have just learned about push and pull instructions. See what you remember by answering this question.

The type of addressing you should you use with PSHA is (SP) – 1 => SP; (A) => SP M(SP).

## Question

**When are Pull instructions commonly used? Select the correct response and then click Done.**

a. Pull instructions are commonly used at the end of a subroutine to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

b. Pull instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine.

c. Pull instructions are commonly used to move the contents of one or more CPU registers at the start of a subroutine.

d. Pull instructions are commonly used at the beginning of a subroutine to restore the contents of CPU registers that were pulled onto the stack before subroutine execution.

Here is another question to check your understanding of the Push and Pull instructions.

Pull instructions are commonly used at the end of a subroutine to restore the contents of CPU registers that were pushed onto the stack before subroutine execution. The Pull instructions do not affect the Condition Code Register unless the CCR is pulled.

## Transfer and Exchange

**Data Handling Instructions**

| FUNCTION | MNEMONIC | OPERATION |
|---|---|---|
| TRANSFER DATA | TBA<br>TAB | B → A<br>A → B |
| | TXS<br>TYS | R → SP |
| | TSY<br>TSX | SP → R |
| **TRANSFER REG TO REG** | **TFR** | **A, B, CCR, D, X, Y, SP**<br>**→ A, B, CCR, D, X, Y, SP** |
| **EXCHANGE** | **EXG** | **A, B, CCR, D, X, Y, SP**<br>**◄► A, B, CCR, D, X, Y, SP** |
| EXCHANGE DATA | XGDX<br>XGDY | D ◄► X<br>D ◄► Y |

**EXAMPLE 1: TFR X ,A**

**EXAMPLE 2: EXG Y ,B**

Now, let's examine transfer and exchange instructions. Transfer instructions copy the content of a register or accumulator into another register or accumulator. The source content is not changed by this operation. Transfer register to register (TFR) is a universal transfer instruction, but other mnemonics are accepted for compatibility with the M68HC11.

The transfer A to B (TAB) and transfer B to A (TBA) instructions affect the N, Z, and V condition code bits in the same way as M68HC11 instructions.

The TFR instruction does not affect the condition code bits. The sign extend 8-bit operand (SEX) instruction is a special case of the universal transfer instruction. It is used to sign extend 8-bit 2's complement numbers so that they can be used in 16-bit operations. The 8-bit number is copied from accumulator A, accumulator B, or the condition code register to accumulator D, the X index register, the Y index register, or the SP. All the bits in the upper byte of the 16-bit result are given the value of the Most-significant Bit (MSB) of the 8-bit number.

Exchange (EXG) instructions exchange the contents of pairs of registers or accumulators. When the first operand in an EXG instruction is 8-bits and the second operand is 16-bits, a zero-extend operation is performed on the 8-bit register as it is copied into the 16-bit register.

# Alter Data

### Data Handling Instructions

| FUNCTION | MNEMONIC | OPERATION |
|---|---|---|
| DECREMENT | DEC<br>DECA<br>DECB | (M)-1 → (M)<br>A-1 → A<br>B-1 → B |
| | DEX<br>DEY<br>DES | X-1 → X<br>Y-1 → Y<br>S-1 → S |
| INCREMENT | INC<br>INCA<br>INCB | (M)+1 → (M)<br>A+1 → A<br>B+1 → B |
| | INX<br>INY<br>INS | X+1 → X<br>Y+1 → Y<br>S+1 → S |

The increment and decrement instructions are optimized 8- and 16-bit addition and subtraction operations. They are generally used to implement loop counters. These instructions add one or subtract one respectively to the content of the memory location or a register. The N, Z and V status bits are set or cleared according to the results of the operation.

The C status bit is not affected by the operation. This allows the INC and DEC instructions to be used as a loop counter in multiple-precision computations.

When operating on unsigned values, only BEQ, BNE, LBEQ, and LBNE branches can be expected to perform consistently. When operating on 2’s complement values, all signed branches are available.

# Clear, Complement, and Negate

### Data Handling Instructions

| FUNCTION | MNEMONIC | OPERATION |
|---|---|---|
| COMPLEMENT, 2'S (NEGATE) | NEG<br>NEGA<br>NEGB | 0-(M) → (M)<br>0-A → A<br>0-B → B |
| COMPLEMENT, 1'S | COM<br>COMA<br>COMB | $(\overline{M})$ → (M)<br>$\overline{A}$ → A<br>$\overline{B}$ → B |
| CLEAR | CLR<br>CLRA<br>CLRB | 0 → (M)<br>0 → A<br>0 → B |
| BIT(S) CLEAR | BCLR | (M) • $\overline{MASK}$ → (M) |
| BIT(S) SET | BSET | (M) + MASK → (M) |

- Bit Manipulation Example: BSET OFFSET,X, #MASK

Now let's examine Clear, Complement and Negate instructions. Each of these instructions performs a specific binary operation on a value in an accumulator or in memory.

Clear operations (CLR) clear the value to 0 and complement operations replace the value with its 1's complement. Negate operations (NEG) replace the value with its 2's complement.

NEG affects N, Z, V, and C. It replaces the contents of Accx or Memory (M) with its 2's complement. However, the value $80 is left unchanged.

Complement (COM) affects only N and Z. It clears V and sets C. It replaces the contents of Accx or Memory (M) with its 1's complement.

CLR (write 0's to operand) clears N, V, C, and sets Z. The contents of Accx or M are replaced with 0's.

Bits Clear (BCLR) and Bits Set (BSET) are used so SET and CLEAR memory operand bits given by 1's that are set in the instruction's operand mask. For BCLR (clear multiple bits in location M). The bits to be cleared are specified by 1's in the mask byte. For BSET (set multiple bits in location M). The bits to be set are specified by 1's in the mask byte. The BSET and BCLR instructions affect N and Z, clears V, and leaves C unaffected.

Users are tempted to use bit manipulation to clear timer system status flags. Note that timer flags are cleared by writing a 1 to the flag after having read it while it was a 1. If you use BSET instruction, you may clear more flags than intended. This is because you would clear any flag in the register that happened to be set during the operand read cycle of the BSET instruction, and not just the bits that were set in the mask of the BSET instruction. You could use BCLR with a mask that has 1's in bits to be cleared. BCLR instruction ANDs operand with the inverse of the mask.

## MAX and MIN Instructions

**Data Handling Instructions**

| Mnemonic | Function | Operation |
|---|---|---|
| | **Minimum Instructions** | |
| EMIND | MIN of two unsigned 16-bit values result to accumulator | MIN ((D), (M : M + 1)) ⇒ D |
| EMINM | MIN of two unsigned 16-bit values result to memory | MIN ((D), (M : M + 1)) ⇒ M : M+1 |
| MINA | MIN of two unsigned 8-bit values result to accumulator | MIN ((A), (M)) ⇒ A |
| MINM | MIN of two unsigned 8-bit values result to memory | MIN ((A), (M)) ⇒ M |
| | **Maximum Instructions** | |
| EMAXD | MAX of two unsigned 16-bit values result to accumulator | MAX ((D), (M : M + 1)) ⇒ D |
| EMAXM | MAX of two unsigned 16-bit values result to memory | MAX ((D), (M : M + 1)) ⇒ M : M + 1 |
| MAXA | MAX of two unsigned 8-bit values result to accumulator | MAX ((A), (M)) ⇒ A |
| MAXM | MAX of two unsigned 8-bit values result to memory | MAX ((A), (M)) ⇒ M |

The maximum (MAX) and minimum (MIN) instructions are used to make comparisons between an accumulator and a memory location. These instructions can be used for linear programming operations, such as simplex-method optimization, or for fuzzification.

MAX and MIN instructions use accumulator A to perform 8-bit comparisons, while EMAX and EMIN instructions use accumulator D to

perform 16-bit comparisons. The result, which is a maximum or minimum value, can be stored in the accumulator (EMAXD, EMIND, MAXA, MINA) or the memory address (EMAXM, EMINM, MAXM, MINM). Click “Example A” to see a comparison between register A and memory locations pointed by X register.

When an operand is found in memory lower than the value in A register, the loop ends and the lower value is loaded from memory to A register. The MIN and MAX instructions assume that the operands are unsigned. N, Z, V and C flags of CCR are updated according to the result of the operation.

## Example A

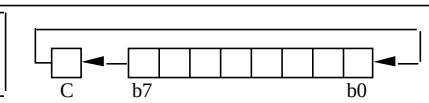
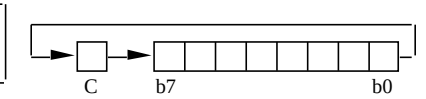
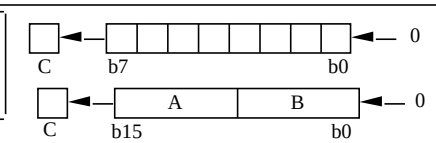
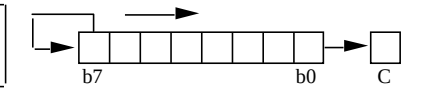
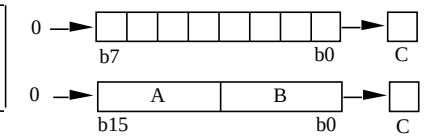
- Example:
```
LOOP    MINA    1,X+
        BHS     LOOP
```

- This example compares register A and memory locations pointed by X register.

**[The content on this page is the “Example A” ]**

# Shift and Rotate

## DATA HANDLING INSTRUCTIONS

| FUNCTION | MNEMONIC | OPERATION |
|---|---|---|
| ROTATE LEFT | ROL<br>ROLA<br>ROLB | M<br>A<br>B<br>  |
| ROTATE RIGHT | ROR<br>RORA<br>RORB | M<br>A<br>B<br>  |
| SHIFT LEFT,<br>ARITHMETIC<br>(LOGICAL) | ASL(LSL)<br>ASLA(LSLA)<br>ASLB(LSLB)<br>ASLD(LSLD) | M<br>A<br>B<br>D<br>  |
| SHIFT RIGHT,<br>ARITHMETIC | ASR<br>ASRA<br>ASRB | M<br>A<br>B<br>  |
| SHIFT RIGHT,<br>LOGICAL | LSR<br>LSRA<br>LSRB<br>LSRD | M<br>A<br>B<br>D<br>  |

Now let’s look at shift and rotate instructions.

There are shifts and rotates for all accumulators and for memory bytes. They all pass the shifted-out bit through the C status bit to facilitate multiple-byte operations. Because logical and arithmetic left shifts are identical, there are no separate logical left shift operations. Logic shift left (LSL) mnemonics are assembled as arithmetic shift left memory (ASL) operations.

Rotate Left (ROL) and Rotate Right (ROR) instructions rotate the operand in a register or a memory location through carry bit. ROL shifts all bits of Accx or M one place to the left. ROR shifts all bits of Accx or M one place to the right. There is no 16-bit rotate for ACCD.

Logical and arithmetic Shifts  all affect condition code register bits N, Z, V, and C .

LSL and ASL are identical in operation and use same opcode. ASL shifts all bits in Accx or M one place to the left. Note that Left Shifts is an efficient way to multiply by powers of two.

Arithmetic Shift Right (ASR) shifts all of Accx or M one bit to the right. BIT 7 is shifted into bit 6 but bit 7 is held constant since this is the operand sign bit. Likewise bit 6 is shifted into bit 5 and so on, and finally bit 0 is shifted into the Carry bit (C-Bit).

Right Shifts is an efficient way to divide by powers of two.

## Question

**Match each instruction type to the statement that describes it by dragging the letters on the left to their corresponding items on the right. Click "Done" when you are finished.**

| | | | |
|---|---|---|---|
| A | Transfer instructions | B | Are optimized 8- and 16-bit addition and subtraction operations |
| B | Increment and decrement instructions | D | Are used to make comparisons between an accumulator and a memory location |
| C | Clear, complement, and negate instructions | A | Copy the content of a register or accumulator into another register or accumulator |
| D | MAX and MIN instructions | C | Perform a specific binary operation on a value in an accumulator or in memory |
| E | Shift and rotate instructions | E | Pass the shifted-out bit into the C Flag bit allowing the software to test each shifted bit, if necessary |

Done Reset Show Solution

Let's review some characteristics of the data handling instructions we have just looked at.

Transfer instructions copy the content of a register or accumulator into another register or accumulator. The Increment and decrement instructions are optimized 8- and 16-bit addition and subtraction operations. The clear, complement, and negate instructions perform a specific binary operation on a value in an accumulator or in memory. The MAX and MIN instructions are used to make comparisons between an accumulator and a memory location. Finally, all shift and rotate instructions pass the shifted-out bit into the C Flag bit allowing the software to test each shifted bit, if necessary.

# Data Test Instructions

| FUNCTION | MNEMONIC | TEST |
|---|---|---|
| BIT TEST | BITA<br>BITB | A • (M)<br>B • (M) |
| COMPARE | CBA<br>CMPA<br>CMPB | A-B<br>A-(M)<br>B-(M) |
| | CPD<br>CPX<br>CPY | $R_L$-(M+1)<br>$R_H$-(M)-C |
| **COMPARE STACK** | **CPS** | **SP - ( M :M +1)** |
| TEST, ZERO OR MINUS | TST<br>TSTA<br>TSTB | (M)-0<br>A-0<br>B-0 |

Compare and test instructions perform subtraction between a pair of registers or between a register and memory. The result is not stored, but condition codes are set by the operation. These instructions are generally used to establish conditions for branch instructions. In this architecture, most instructions update condition code bits automatically, so it is often unnecessary to include separate test or compare instructions.

# Conditional Branch Instructions

| MNEMONIC | CONDITION | CCR TEST | INDICATION |
|---|---|---|---|
| (**L**) BMI | MINUS | N=1 | r=NEGATIVE |
| (**L**) BPL | PLUS | N=0 | r=POSITIVE |
| *(**L**) BVS | OVERFLOW | V=1 | r=SIGN ERROR |
| *(**L**) BVC | NO OVERFLOW | V=0 | r=SIGN OK |
| *(**L**) BLT | LESS | [N⊕V]=1 | A < M |
| *(**L**) BGE | GREATER OR EQUAL | [N⊕V]=0 | A >= M |
| *(**L**) BLE | LESS OR EQUAL | [Z+(N⊕V)]=1 | A <= M |
| *(**L**) BGT | GREATER | [Z+(N⊕V)]=0 | A > M |
| (**L**) BEQ | EQUAL | Z=1 | A=M |
| (**L**) BNE | NOT EQUAL | Z=0 | A <> M |
| (**L**) BHI | HIGHER | [C+Z]=0 | A > M |
| (**L**) BLS | LOWER OR SAME | [C+Z]=1 | A <= M |
| (**L**) BCC (BHS) | CARRY CLEAR | C=0 | A >= M |
| (**L**) BCS (BLO) | CARRY SET | C=1 | A < M |

Indication refers to the use of a CMPA M instruction immediately before the branch

*Use for signed arithmetic only

Branch instructions cause a program sequence to change when specific conditions are met. The CPU12 uses three kinds of branch instructions. These are short branches, long branches, and bit condition branches. Branch instructions can also be classified by the type of condition that must be satisfied in order for a branch to be taken.

Some branch instructions belong to more than one classification. The list of branch instructions above support algebraic and non-algebraic comparison operations.

For example, a non-algebraic 8-bit operand can be in the range of hex ‘0’ to hex ‘FF’, which can represent a value between 0 to 255 decimal. For algebraic 8-bit operand, the maximum positive value that can be represented is from ‘0’ to hex ‘7F, which is equal to plus 127 decimal. Where as the value of hex 80 – hex FF represent the values of minus128 to minus 1 decimal.

Click “Example B” to see an example that compares register A with memory locations pointed by X register.

CPU12 supports two type of branches, short and long. Short branches use an 8-bit signed offset which is added to the value of the Program Counter (PC) when the condition is met. The branch ranges from +127 to -128 locations. Long branches use 16-bit signed offset allowing the CPU to branch anywhere in the 64K memory map.

## Example B

- Example:
- Next: CMP A,X+ ; Compare A to memory location pointed to by (X)
- BNE Next: If not equal, loop back to next
- The example above compares register A with memory locations pointed by X register. The program will loop until a value in memory found that compares to register A.
- Note:  Register x increment with every iteration.

**[The content on this page is the “Example B” ]**

## Conditional Branch Instructions

| FUNCTION | MNEMONIC | OPERATION |
|---|---|---|
| DECREMENT & BRANCH | DBEQ | COUNTER - $01 → COUNTER<br>IF COUNTER =0, THEN (PC)+$0003 +REL → PC |
| | DBNE | COUNTER - $01, → COUNTER<br>IF COUNTER <>0, THEN (PC)+$0003 +REL → PC |
| INCREMENT & BRANCH | IBEQ | COUNTER + $01 → COUNTER<br>IF COUNTER =0, THEN (PC)+$0003 +REL → PC |
| | IBNE | COUNTER + $01 → COUNTER<br>IF COUNTER <>0, THEN (PC)+$0003 +REL → PC |
| TEST & BRANCH | TBEQ | IF COUNTER = 0, THEN PC+$0003 + REL → PC |
| | TBNE | IF COUNTER <>0, THEN PC+$0003 + REL → PC |

With loop primitive instructions, the loop primitives can also be thought of as counter branches. The instructions test a counter value in a register or accumulator (A, B, D, X, Y, or SP) for zero or non-zero value as a branch condition. There are pre-decrement, pre-increment, and test-only versions of these instructions. The numeric range of 8-bit offset values is $80 or (–128) to $7F (+127) from the address of the next memory location after the offset value. Click “Example C” to see an example that moves a block of data from memory location pointed by Y register to memory location pointed by X register.

The BDNE instruction subtracts a ‘1’ from the loop counter register D and if the loop counter has not yet decremented to ‘$0000’, the program will continues to loop.

## Example C

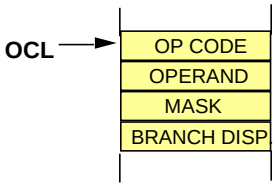
```
EXAMPLE:
          -
 LOOP     MOVW  2,Y+, 2,X+
          DBNE  D,LOOP
            -
```

- The example above moves a block of data from memory location pointed by Y register to memory location pointed by X register. The X and Y registers are incremented by two to point to the next word in memory.

**[The content on this page is the “Example C” ]**

# Branch if Bits Set or Clear

BRSET
BRCLR
(M) MASK SERVICE




The Mask value is used to test bit or bits in the memory operand.

Example:
A value of $80 in the mask, tests bit 7 of the operand in memory.
A value of $61 in the mask, tests bits 6 and 1.

- Addressing modes allowed are: DIR, EXT, IDX, IDX1 & IDX2.

Let's look at bit condition branch instructions. The bit condition branches are taken when a bit or bits in a memory location are in a specific state. A mask operand is used to test the byte location. If all bits in the location that corresponds to the ones in the mask are set using the BRSET instruction, the branch is taken. The numeric range of 8-bit offset values is $80 (–128) to $7F (127) from the address of the next memory location after the offset value. The Branch if bit or bits cleared, BRCLR instruction test for zero or zeros in the memory location specified, depending on the value in the mask field.

Bit condition branch instructions are useful for polling interrupt status flags and for making program decisions based on bit values. Each branch is taken from the next instruction address (OCL +4, 5, OR 6 ). Addressing modes that are allowed include DIR, EXT, IDX, IDX1, and IDX2.

Click "Example D" to see an example of Port D bit 7 instruction tested.

## Example D

EXAMPLE:

```
WAIT  BRCLR    PORTD,X $80  WAIT
```

- The example above instruction tests Port D bit 7 and stays in this short loop until the bit is set. This instruction is useful for polling the status of a port pin or even an I/O device flag.

**[The content on this page is the “Example D” ]**

## Question

**Which of following statements about condition branch instructions are true? Select all options that apply.**

- Branch instructions cause a program sequence to change when specific conditions are met.
- All instructions belong to more than one classification.
- Branch instructions can be classified by the type of condition that must be satisfied in order for a branch to be taken.
- Some instructions belong to more than one classification.

Consider this question regarding condition branch instructions.

The CPU12 uses three kinds of branch instructions; short branches, long branches, and bit condition branches. Branch instructions can be classified by the type of condition that must be satisfied in order for a branch to be taken. Only some instructions belong to more than one classification.

# Add/Decimal Adjust Instructions

| FUNCTION | MNEMONIC | OPERATION |
|---|---|---|
| ADD | ADDA<br>ADDB<br>ADDD | A + (M) ► A<br>B + (M) ► B<br>$D_L$ + (M+1) ► $D_L$; $D_H$ + M + C ► $D_H$ |
| ADD ACCUMULATORS | ABA<br>ABX<br>ABY | A + B ► A<br>X + B ► X<br>Y + B ► Y |
| ADD WITH CARRY | ADCA<br>ADCB | A + M + C ► A<br>B + M + C ► B |
| DECIMAL ADJUST | DAA | CONVERTS BINARY ADDITION OF BCD CHARS INTO BCD FORMAT |

Next, we will examine add and decimal adjust instructions. First, let's look at ABY. This is one of the rare places the instruction set isn't entirely general. You can add B to X or Y but you can't add A to X or Y. ABY is useful for calculating offsetts into multi-dimensional arrays.

If you want to do 16-bit arithmetic on X or Y, just do XGDX then 16-bit arithmetic such as ADDD, then XGDX again. Note that X acts as temp for D in this sequence.

All Add instructions update Condition code register bits N, Z, C, H and V.

ABX and ABY (adds ACCB to index register X or Y) and it does not affect condition code, CCR register field. ABX and ABY are useful for pointing index register to a new (calculated) address.

Add with Carry,ADDC instruction includes the carry bit in the addition operation to support multi-precision addition.

DAA is only of use immediately after executing ADDA to transform the accumulator's hexadecimal results to decimal, using the half carry bit.

# Subtract & Multiply Instructions

| FUNCTION | MNEMONIC | OPERATION |
|---|---|---|
| SUBTRACT | SUBA<br>SUBB<br>SUBD | A – (M) → A<br>B – (M) → B<br>$D_L$ – (M+1) → $D_L$; $D_H$ – (M) – C → $D_H$ |
| SUBTRACT ACCUMULATORS | SBA | A – B → A |
| SUBTRACT WITH CARRY | SBCA<br>SBCB | A – (M) – C → A<br>B – (M) – C → B |
| MULTIPLY | MUL | A * B → D |
| **EXTENDED MULTIPLY** | **EMUL** | **D * Y → Y : D** |
| **EXTENDED MULTIPLY SIGNED** | **EMULS** | **D * Y → Y : D** |

Now, let's discuss subtract and multiply instructions.

SUBA instruction Subtracts Memory location specified from A and places the difference in accumulator A. SBA instruction Subtracts B from A and places the difference in accumulator A.

SUBD instruction Subtracts 16-bits from memory location specified and places the difference in accumulator D. All other instruction that includes C in the operation, will subtract with Carry.

The subtract operation always affect N, Z, V, and C.

MUL is 8x8 unsigned multiply giving a 16-bit result. MUL takes 3 clocks & affects C bit according to bit 7 (ACCB bit 7) of 16-bit result.

# Divide Instructions

**INTGER DIVIDE 16/16 UNSIGNED OR SIGNED (IDIV/IDIVS)**

| | |
|---|---|
| **OPERATION** | D REG / X REG |
| **RESULT** | QUOTIENT IS IN X<br>REMAINDER IS IN D |
| **INTEGER DIVIDE** | IDIV/IDIVS |

RADIX POINT OF THE RESULT IS TO THE RIGHT OF THE LSB

**EXTENDED DIVIDE 32-BIT BY 16-BIT UNSIGNED OR SIGNED (EDIV/EDIVS)**

**OPERATION** (Y:D)/ (X) → Y; REMAINDER → D

V = 1, IF RESULT > $FFFF FOR UNSIGNED, UNDEFINED IF DIVISOR IS $0000
V = 1, IF RESULT > $7FFF FOR SIGNED, UNDEFINED IF DIVISOR IS $0000
C = 1, IF DIVISOR WAS $0000

Let's look at the description of Integer Divide (IDIV) 16/16 Unsigned or Signed. This divides an unsigned/signed 16-bit dividend in double accumulator D by an unsigned/signed 16-bit divisor in index register X.

This produces an unsigned/signed 16-bit quotient in X, and an unsigned/signed 16-bit remainder in D.

The signed 16-bit dividend depends on whether the IDIV or IDIVS is used. If both the divisor and the dividend are assumed to have radix points in the same positions, the radix point of the quotient is to the right of bit 0. In the case of division by zero, C is set, the quotient is set to $FFFF, and the remainder is indeterminate.

The condition codes for IDIV are: Z equals 1 if the quotient equals 0; C equals 1 if the divisor equals 0. The condition codes for Fractional Divide (FDIV) are: Z equals 1 if the quotient equals 0; V equals 1 if the dividend is greater than the divisor; C equals 1 if the divisor equals 0.

Now let's examine extend divide 32/16 unsigned or signed. This divides a 32-bit unsigned/signed dividend by a 16-bit divisor, depending on whether the EDIV or EDIVS is used. This produces a 16-bit unsigned/signed quotient and an unsigned/signed 16-bit remainder. All operands and results are located in CPU registers.

If an attempt to divide by zero is made, the contents of double accumulator D and index register Y do not change. C is set and the states of the N, Z, and V bits in the CCR are undefined.

EDIV/EDIVS are useful for A/D and D/A calculations. Their result values can be compared to ratiometric A/D results. Also, the results are in correct form to drive a weighted D/A. Examine the condition codes in the grey box to see how.

## Fractional Divide Instruction

Fractional Divide (FDIV)

•The radix point of the result is to the left of the MSB

•If the numerator is greater than or equal to the denominator, then V Flag is set.

Result Examples:

A result of 1 is 1/$10000 which is .0001

A result of $C000 is $C000/$10000 which is .75

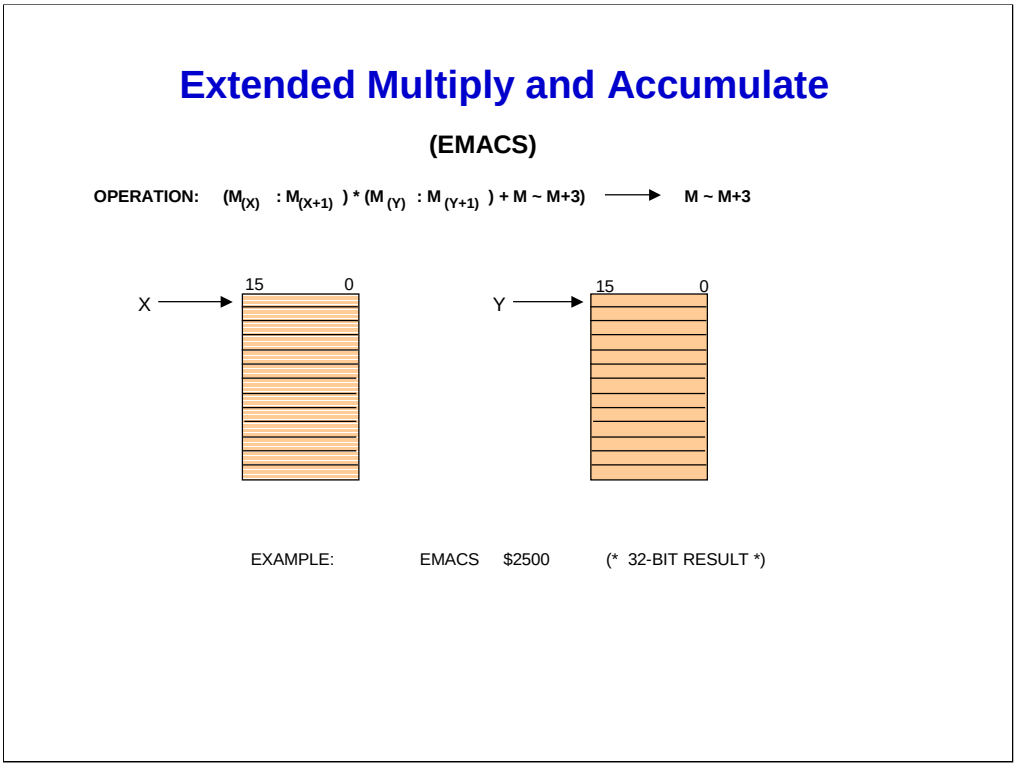
A result of $FFFF IS $FFFF/$10000 which is .9999

Here are the Fractional Divide (FDIV) instructions. FDIV may be executed after IDIV to resolve the remainder. In fact, FDIV can follow another FDIV to resolve more bits past the radix point. If the numerator is greater than or equal to the denominator, then V Flag is set which indicates an overflow condition is generated by this operation.

Ratiometric A/D and D/A values are also weighted binary fractions. These express the analog value as a fractional portion of the analog reference. For example $C000 means 3/4 or 0.75 (base 10) of the reference value. This idea also extends to percentage calculations.

FDIV is intended for use when the dividend is less than the divisor allowing for fractional results to be obtained. If the FDIV is successful, then the quotient (in X register) equals 0 and the remainder is a 16 bit fractional value in ACCD.

If FDIV fails, when the dividend is greater than the divisor, then the quotient (in X register) equals $FFFF and the remainder is undefined.




Extended Multiply and Accumulate (EMACS) Instruction allows for multiply and accumulate operations by multiplying two 16-bit operands to produce a 32-bit intermediate result. This 32-bit intermediate result is then added to the content of a 32-bit accumulator in memory. EMACS is a signed integer operation. All operands and results are located in memory.

When the EMACS instruction is executed, the first source operand is fetched from an address pointed to by X, and the second source operand is fetched from an address pointed to by index register Y. Before the instruction is executed, the X and Y index registers must contain values that point to the most significant bytes of the source operands. The most significant byte of the 32-bit result is specified by an extended address supplied with the instruction.

The EMACS instruction may be useful for DSP applications that do require extremely high speed since the instruction takes 13 E clocks to execute.

# Logic Instructions

| FUNCTION | MNEMONIC | OPERATION |
|---|---|---|
| AND | ANDA<br>AND B<br>AND CC | A • (M) ► A<br>B • (M) ► B<br>CCR • MASK → CCR |
| EXCLUSIVE OR | EORA<br>EORB | A ⊕ (M) ► A<br>B ⊕ (M) ► B |
| INCLUSIVE OR | ORAA<br>ORAB<br>ORCC | A + (M) ► A<br>B + (M) ► B<br>CCR + MASK → CCR |

The Boolean logic instructions perform a logic operation between an 8-bit accumulator, or the CCR, and a memory value. This supports AND, OR, and EXCLUSIVE OR functions.

The AND instruction may be used to mask out unwanted operand bits.

The OR instruction may be used to set operand bits.

The EXCLUSIVE OR instruction may be used to toggle operand bits.

## Question

**What type of instruction allows for multiply and accumulate operations by multiplying two 16-bit operands to produce a 32-bit intermediate result?**

a. Add and decimal adjust instructions
b. Subtract and multiply instructions
c. Divide instructions
d. Logic instructions
e. Extended multiply and accumulate instructions
f. Fractional Divide Instructions

Here is a question to check your understanding of the latest instructions you have just learned.

Extended Multiply and Accumulate (EMACS) Instruction allows for multiply and accumulate operations by multiplying two 16-bit operands to produce a 32-bit intermediate result. This 32-bit intermediate result is then added to the content of a 32-bit accumulator in memory. EMACS is a signed integer operation. All operands and results are located in memory.

# Jump and Subroutine Instructions

| Mnemonic | Function | Operation |
|---|---|---|
| BSR | Branch to subroutine | $SP - 2 \Rightarrow SP$<br>$RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$<br>Subroutine address $\Rightarrow$ PC |
| CALL | Call subroutine in expanded memory | $SP - 2 \Rightarrow SP$<br>$RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$<br>$SP - 1 \Rightarrow SP$<br>$(PPAGE) \Rightarrow M_{(SP)}$<br>Page $\Rightarrow$ PPAGE<br>Subroutine address $\Rightarrow$ PC |
| JMP | Jump | Address $\Rightarrow$ PC |
| JSR | Jump to subroutine | $SP - 2 \Rightarrow SP$<br>$RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$<br>Subroutine address $\Rightarrow$ PC |
| RTC | Return from call | $M_{(SP)} \Rightarrow$ PPAGE<br>$SP + 1 \Rightarrow SP$<br>$M_{(SP)} : M_{(SP+1)} \Rightarrow PC_H : PC_L$<br>$SP + 2 \Rightarrow SP$ |
| RTS | Return from subroutine | $M_{(SP)} : M_{(SP+1)} \Rightarrow PC_H : PC_L$<br>$SP + 2 \Rightarrow SP$ |

Jump (JMP) instructions cause immediate changes in a program sequence. The JMP instruction loads the PC with an address in the 64-Kbyte memory map, and program execution continues at that address. The address can be provided as an absolute 16-bit address or determined by various forms of indexed addressing.

Subroutine instructions optimize the process of transferring control to a code segment that performs a particular task. A Short Branch (BSR), a Jump to Subroutine (JSR), or an expanded-memory call (CALL) can be used to initiate subroutines. There is no Long Branch to Subroutine (LBSR) instruction, but a PC-relative JSR performs the same function. A return address is stacked, then execution begins at the subroutine address.

Subroutines in the normal 64-Kbyte address space are terminated with a Return-from-Subroutine (RTS) instruction. RTS unstacks the return address so that execution resumes with the instruction after BSR or JSR.

The call subroutine in expanded memory (CALL) instruction is intended for use with expanded memory. CALL stacks the value in the PPAGE register and the return address, then writes a new value to PPAGE to select the memory page where the subroutine resides. The page value is an immediate operand in all addressing modes, except indexed indirect modes. In these modes, an operand points to locations in memory where the new page value and subroutine address are stored.

The return from call (RTC) instruction is used to terminate subroutines in expanded memory. RTC unstacks the PPAGE value and the return address so that execution resumes with the next instruction after CALL. For software compatibility, CALL and RTC execute correctly on devices that do not have expanded addressing capability.

# CCR Instructions

| FUNCTION | MNEMONIC | OPERATION |
|---|---|---|
| CLEAR CARRY | CLC | 0 ► C |
| CLEAR INTERRUPT MASK | CLI | 0 ► I |
| CLEAR OVERFLOW | CLV | 0 ► V |
| SET CARRY | SEC | 1 ► C |
| SET INTERRUPT MASK | SEI | 1 ► I |
| SET OVERFLOW | SEV | 1 ► V |
| ACCUMULATOR A ► CCR | TAP | A ► CCR |
| CCR ► ACCUMULATOR A | TPA | CCR ► A |
| **OR CONDITION CODE** | **ORCC** | **CCR + OPERAND** |
| **AND CONDITION CODE** | **ANDCC** | **CCR ^ OPERAND** |

Condition Code Register (CCR) instructions allow the user to manipulate a particular bit or bits in the CCR. You can set or clear a particular bit, move accumulator A register to CCR, or AND and OR an operand with the CCR.

A good example would be to re-enable interrupts by executing the Clear Interrupt Mask (CLI) instruction or disable interrupts by executing the Set Interrupt Mask (SEI) instruction.

Click “Block Move” and “Clear RAM” to attempt to finish these programs by filling the blanks.

# Block Move Routine

**Write a block move routine. The routine copies data from memory location $1000 to memory location $1100. The routine will end when A data byte with a value of zero is moved.**

**WRITE YOUR PROGRAM HERE**

```
          ORG  $1000
SOURCE    FCC  'DATA TO MOVE'
          FCB  0
          ORG  $4000
LOOP
          BEQ  DONE
          BRA  LOOP
DONE      BRA  DONE
```

**SUGGESTED PROGRAM STEPS**

ORIGINATE DATA AT ADDRESS $1000.

FORM TABLE OF DATA TO BE MOVED
FORM CONSTANT BYTE OF '0'.

PROGRAM BEGINS @$4000.

1. INIT SOURCE POINTER T0 $1000.
2. INIT DESTINATION POINTER TO $1100.
3. GET DATA FROM SOURCE ADDRESS.
4. WRITE DATA TO DESTINATION ADDRESS,
5. IF DATA MOVED = 0, GO TO STEP 9, ELSE GO TO 6.
6. INCREMENT SOURCE POINTER.
7. INCREMENT DESTINATION POINTER.
8. GO TO STEP 3.
9. STAY HERE.

**[The content on this page is the "Black Move" ]**

# Clear RAM Routine

**Write a routine to clear the HCS12 RAM memory, assume RAM begins at $1000**
**and ends at $3FFF.**

```
CLRRAM_RTN:
                        ; Initialize X pointer to start of RAM ($1000)
LOOP:                   ; Clear memory pointed to by X register, Inc X
                        ; Compare pointer with $4000
                        ; If pointer not equal, Branch to LOOP
Done    BRA   Done      ;  End program here
```

**[The content on this page is the “Clear RAM” ]**

## Question

**Match each instruction type to the statement that describes it by dragging the letters on the left to their corresponding items on the right. Click "Done" when you are finished.**

| | | | |
|---|---|---|---|
| A | JMP | B | This instruction is used to terminate subroutines in expanded memory. |
| B | RTC | A | These instructions cause immediate changes in program sequence. |
| C | RTS | D | These instructions update the condition that was tested due to a compare, math or logical operation. The program then, can evaluate the condition and take the appropriate action by either branching or not. |
| D | CCR | C | This unstacks the return address so that execution resumes with the instruction after BSR or JSR. |

Done Reset Show Solution

Let's review jump and subroutine call instructions and condition code register instructions.

Jump (JMP) instructions cause immediate changes in program sequence. The return from call (RTC) instruction is used to terminate subroutines in expanded memory. The return-from-subroutine (RTS) instruction unstacks the return address so that execution resumes with the instruction after BSR or JSR. The Condition Code Register (CCR) instructions update the condition that was tested due to a compare, math or logical operation. The program then, can evaluate the condition and take the appropriate action by either branching or not.

## Module Summary

- Data Handling
- Arithmetic
- Logic
- Data Test
- Branch
- Jump & Subroutine Calls

Now that you have completed this module, you should be able discuss all the CPU12 instruction sets, describe data handling instructions, and identify arithmetic instructions. You should also be better prepared to describe logic instructions, identify data test instructions, describe the branch instructions, and discuss the jump and subroutine calls. Finally, you should be able to identify the HCS12 CPU features that support high-level language programs.