# Experiment 3: Arithmetic Instructions, Scaling, Condition Code Register, and Branch Instructions

## Instructional Objectives:

- To properly implement arithmetic instructions to perform computations on the HC(S)12.
- To correctly scale numerical operations to obtain an accurate result for a computation.
- To understand the difference between signed and unsigned computations and how they affect the Condition Code Register.
- To create a program that will continuously run on the I/O board.

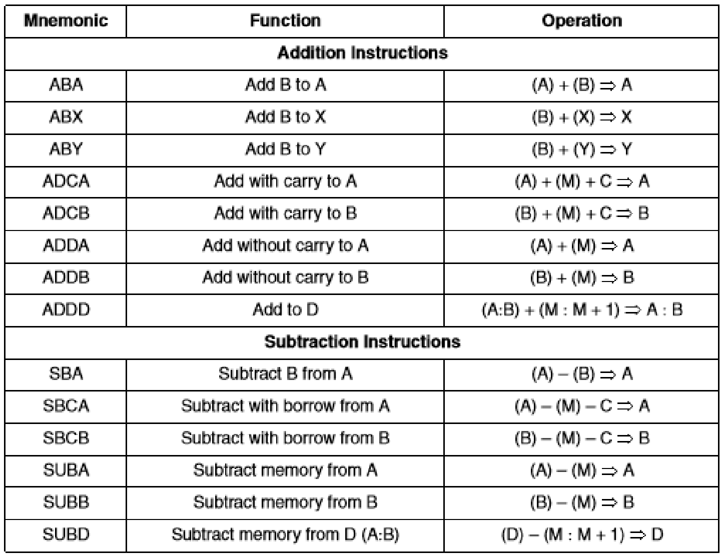
## Introduction:

The following procedure further instills important concepts for programming in assembly. These concepts include accurately implementing an alternative to floating point computations, understanding the Condition Code Register and the difference between signed and unsigned operations, understanding how the CCR affects branching and looping, and understanding how to write software that will continuously run on the I/O in lab.

## Arithmetic Instructions and Scaling:

The HC(S)12 is capable of performing addition, subtraction, comparing, incrementation, decrementation, multiplication, and division instructions. These operations can be performed either between two internal registers or between an internal register or a memory location. The result of an operation is typically placed in an internal register. These instructions are all listed in the Reference Manual document. For the purpose of convenience, some of these instructions are presented in the tables below.

**Table 5-4. Addition and Subtraction Instructions**



| Mnemonic | Function | Operation |
|---|---|---|
| **Addition Instructions** | | |
| ABA | Add B to A | $(A) + (B) \Rightarrow A$ |
| ABX | Add B to X | $(B) + (X) \Rightarrow X$ |
| ABY | Add B to Y | $(B) + (Y) \Rightarrow Y$ |
| ADCA | Add with carry to A | $(A) + (M) + C \Rightarrow A$ |
| ADCB | Add with carry to B | $(B) + (M) + C \Rightarrow B$ |
| ADDA | Add without carry to A | $(A) + (M) \Rightarrow A$ |
| ADDB | Add without carry to B | $(B) + (M) \Rightarrow B$ |
| ADDD | Add to D | $(A:B) + (M : M + 1) \Rightarrow A : B$ |
| **Subtraction Instructions** | | |
| SBA | Subtract B from A | $(A) - (B) \Rightarrow A$ |
| SBCA | Subtract with borrow from A | $(A) - (M) - C \Rightarrow A$ |
| SBCB | Subtract with borrow from B | $(B) - (M) - C \Rightarrow B$ |
| SUBA | Subtract memory from A | $(A) - (M) \Rightarrow A$ |
| SUBB | Subtract memory from B | $(B) - (M) \Rightarrow B$ |
| SUBD | Subtract memory from D (A:B) | $(D) - (M : M + 1) \Rightarrow D$ |

**Table 5-6. Decrement and Increment Instructions**

| Mnemonic | Function | Operation |
|---|---|---|
| **Decrement Instructions** | | |
| DEC | Decrement memory | $(M) - \$01 \Rightarrow M$ |
| DECA | Decrement A | $(A) - \$01 \Rightarrow A$ |
| DECB | Decrement B | $(B) - \$01 \Rightarrow B$ |
| DES | Decrement SP | $(SP) - \$0001 \Rightarrow SP$ |
| DEX | Decrement X | $(X) - \$0001 \Rightarrow X$ |
| DEY | Decrement Y | $(Y) - \$0001 \Rightarrow Y$ |
| **Increment Instructions** | | |
| INC | Increment memory | $(M) + \$01 \Rightarrow M$ |
| INCA | Increment A | $(A) + \$01 \Rightarrow A$ |
| INCB | Increment B | $(B) + \$01 \Rightarrow B$ |
| INS | Increment SP | $(SP) + \$0001 \Rightarrow SP$ |
| INX | Increment X | $(X) + \$0001 \Rightarrow X$ |
| INY | Increment Y | $(Y) + \$0001 \Rightarrow Y$ |

**Table 5-10. Multiplication and Division Instructions**

| Mnemonic | Function | Operation |
|---|---|---|
| **Multiplication Instructions** | | |
| EMUL | 16 by 16 multiply (unsigned) | $(D) \times (Y) \Rightarrow Y : D$ |
| EMULS | 16 by 16 multiply (signed) | $(D) \times (Y) \Rightarrow Y : D$ |
| MUL | 8 by 8 multiply (unsigned) | $(A) \times (B) \Rightarrow A : B$ |
| **Division Instructions** | | |
| EDIV | 32 by 16 divide (unsigned) | $(Y : D) \div (X) \Rightarrow Y$<br>Remainder $\Rightarrow D$ |
| EDIVS | 32 by 16 divide (signed) | $(Y : D) \div (X) \Rightarrow Y$<br>Remainder $\Rightarrow D$ |
| FDIV | 16 by 16 fractional divide | $(D) \div (X) \Rightarrow X$<br>Remainder $\Rightarrow D$ |
| IDIV | 16 by 16 integer divide (unsigned) | $(D) \div (X) \Rightarrow X$<br>Remainder $\Rightarrow D$ |
| IDIVS | 16 by 16 integer divide (signed) | $(D) \div (X) \Rightarrow X$<br>Remainder $\Rightarrow D$ |

Processors with an 8-bit data bus can perform 8-bit arithmetic operations and may be capable of 16-bit operations. 16-bit processors can typically perform 8-bit and 16-bit operations with a single instruction. The HC(S)12 can perform 16-bit computations and return a 32-bit result stored in two separate registers.

The HC(S)12 has no floating point computation unit. Therefore, in order to process decimal or fractional numbers, scaling must be used. Consider the following equation to convert a person's height from Inches to Centimeters.

$$Centimeters = 2.54 * Inches$$

This is a simple equation, yet as is, the HC(S)12 is not capable of performing it. To get past this problem, consider that the conversion factor above can be written as a fractional value.

$$Centimeters = \frac{254}{100} * Inches$$

The above operation can be performed by the HC(S)12 by first multiplying the input by the numerator and then dividing by the denominator.

One last thing to note is that the value to the right of the decimal point is truncated off. For example, if the result above is 178.8, the value returned with be 178. This is important when deciding how to order operations. Reconsider the above equation. Suppose that a person has a height of 70 inches. If the scaling is performed first, the conversion will become:

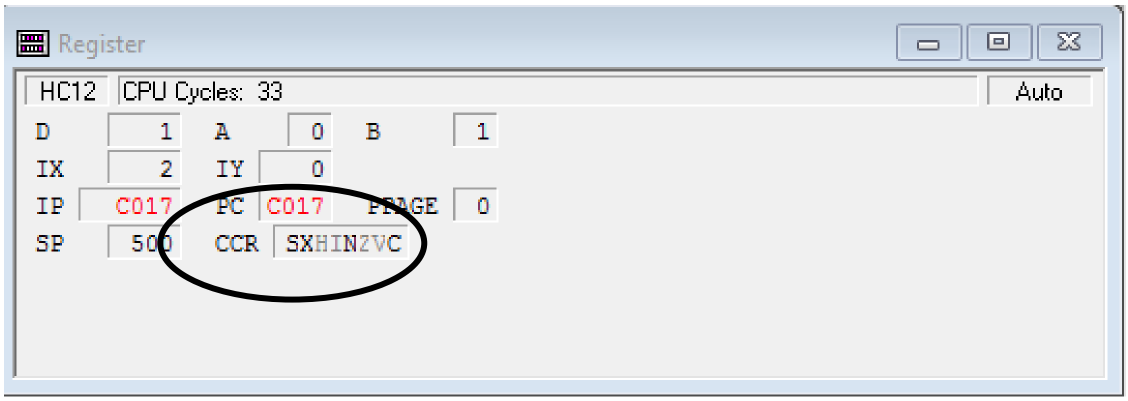
$$Centimeters = \left(\frac{254}{100}\right) * Inches = \left(\frac{254}{100}\right) * 70 \neq 2 * 70 = 140$$

Note that the above is not an accurate result. In order to return an accurate result using the HC(S)12, perform the multiplication first, then the division, like below.

$$Centimeters = \left(\frac{254}{100}\right) * Inches = \frac{254 * 70}{100} = \frac{17780}{100} \approx 177$$

**Condition Code Register:**

The Condition Code Register is another register in the HC(S)12. This register contains 5 status indicators, 2 interrupt masking bits, and a STOP instruction disable bit.



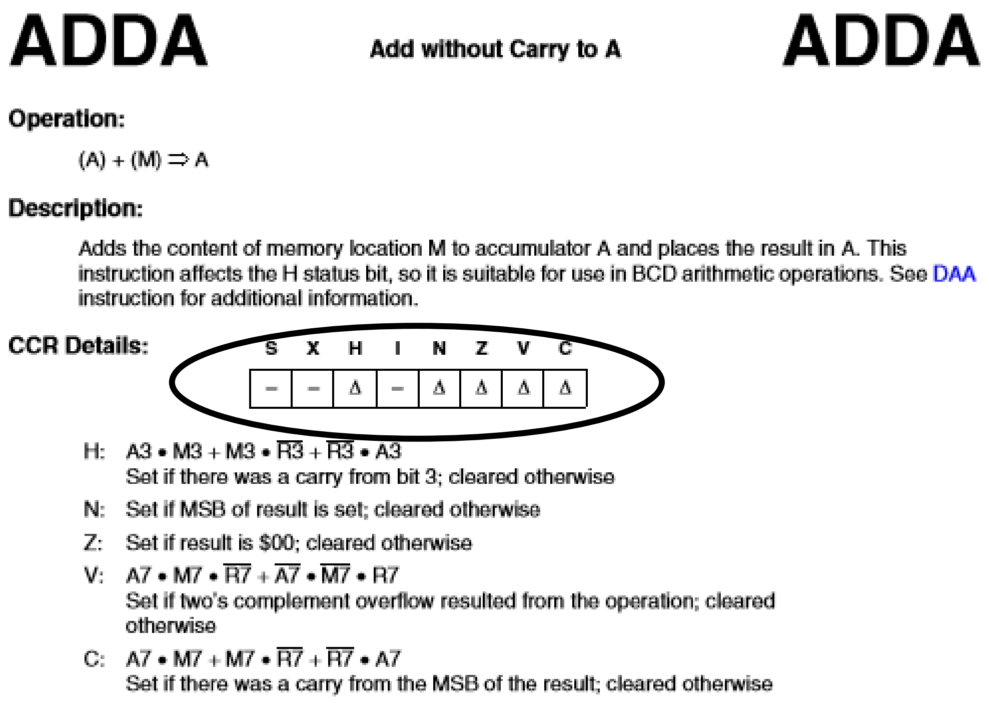
```
D    1   A    0   B     1
IX   2   IY   0
IP C017  PC C017  PPAGE 0
SP 500   CCR SXHINZVC
```

In the figure above, the bits that are bold (S, X, N, C) are set. The bits that are clear (H, I, Z, V) are clear. Each bit can be set or cleared by double-clicking on that bit in the register window. For a brief explanation of the function of each bit, please refer to the table below.

| | |
|---|---|
| **MASKING BITS:** | S – Disables STOP Instruction when set. |
| | X – Masks XIRQ request when set. |
| | I – Masks interrupt request from all |
| **ARITHMETIC BITS:** | H – Half Carry from bit 3 to bit 4 |
| | N – Negative (follows MS Bit of result) |
| | Z – Zero result |
| | V – 2’s complement overflow indication |
| | C – Carry/Borrow from MSB |

Masking bits won’t be discussed in this experiment. Arithmetic bits are relevant for conditional branching. In order to write code that uses conditional loops, it is important to know what conditions set and clear these bits.

In the instruction gallery in chapter 6, each instruction includes a table that shows what bits of the Condition Code Register it effects. A delta (Δ) means that this instruction is capable of changing the corresponding bit of the CCR. A dash (–) means that this instruction has no effect on the corresponding bit. A zero (0) means that this instruction will *always* clear the corresponding bit of the CCR. When a Δ appears in the table, the conditions necessary to set or clear that corresponding bit are also given. Please refer to the example on the next page.



**ADDA** Add without Carry to A **ADDA**

**Operation:**

$(A) + (M) \Rightarrow A$

**Description:**

Adds the content of memory location M to accumulator A and places the result in A. This instruction affects the H status bit, so it is suitable for use in BCD arithmetic operations. See DAA instruction for additional information.

**CCR Details:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| – | – | Δ | – | Δ | Δ | Δ | Δ |

H: $A3 \bullet M3 + M3 \bullet \overline{R3} + \overline{R3} \bullet A3$
Set if there was a carry from bit 3; cleared otherwise

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: $A7 \bullet M7 \bullet \overline{R7} + \overline{A7} \bullet \overline{M7} \bullet R7$
Set if two's complement overflow resulted from the operation; cleared otherwise

C: $A7 \bullet M7 + M7 \bullet \overline{R7} + \overline{R7} \bullet A7$
Set if there was a carry from the MSB of the result; cleared otherwise

The carry and half-carry bits are relevant for unsigned computation. Unsigned numbers have the following range:

$$\boldsymbol{8-bit\ unsigned\ numbers}:\quad \mathbf{0\ to\ 2^8 - 1 = 0\ to\ 255}$$

$$\boldsymbol{n-bit\ unsigned\ numbers}:\quad \mathbf{0\ to\ 2^n - 1}$$

There are two different ways to determine if the carry bit is set. One is to use the relevant formula provided in the lecture slides. The carry bit is set by the following operations, where $A_7$ is the MSB of the first number, $B_7$ is the MSB of the second number, and $R_7$ is the result.

$$C = A_7 \cdot B_7 + A_7 \cdot \overline{R_7} + B_7 \cdot \overline{R_7} \qquad (For\ Addition)$$

$$C = \overline{A_7} \cdot B_7 + \overline{A_7} \cdot R_7 + B_7 \cdot R_7 \qquad (For\ Subtraction)$$

The other is to use common sense. If the result of adding two 8-bit number won't fit in an 8-bit result, the carry bit is set. Additionally, if a subtrahend is larger than a minuend in a subtraction operation, the carry bit is also set.

The overflow and negative bits are relevant for signed computations. Signed numbers have the following range:

$$\boldsymbol{8-bit\ signed\ numbers:\quad -2^{8-1}\ to\ 2^{8-1}-1 = -128\ to\ 127}$$

$$\boldsymbol{n-bit\ signed\ numbers:\quad -2^{n-1}\ to\ 2^{n-1}-1}$$

There are two different ways to determine if the overflow bit is set. One is to use the relevant formula provided in the lecture slides. The overflow bit is set by the following operations, where $A_7$ is the MSB of the first number, $B_7$ is the MSB of the second number, and $R_7$ is the result.

$$V = A_7 \cdot B_7 \cdot \overline{R_7} + \overline{A_7} \cdot \overline{B_7} \cdot R_7 \qquad (For\ Addition)$$

$$V = A_7 \cdot \overline{B_7} \cdot \overline{R_7} + \overline{A_7} \cdot B_7 \cdot R_7 \qquad (For\ Subtraction)$$

The shortcut to the above is to use these two rules:

- If adding two positive numbers together results in a negative number, the overflow flag is set.
- If adding two negative numbers together results in a positive number, the overflow flag is set
- If subtracting a positive number from a negative number results in a positive number, the overflow flag is set.
- If subtracting a negative number from a positive number results in a negative number, the overflow flag is set.

**Note:** The carry bit and the overflow bits can be set at the same time. The processor simultaneously treats numbers as both signed and unsigned. For example, consider the following operation for two 8-bit numbers:

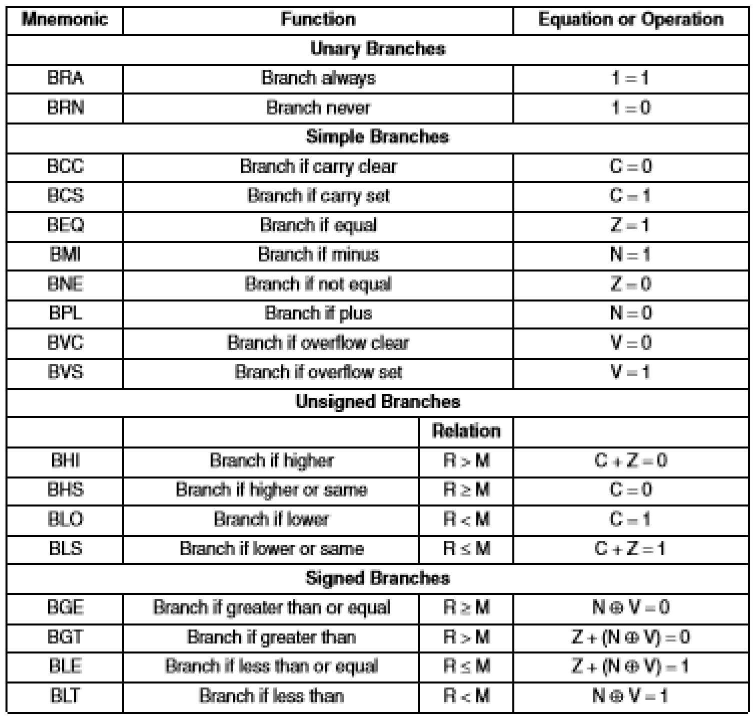
$$\$90 + \$90 = \$20$$

In the above, the carry flag is set, because the result won’t fit into an 8-bit number. The overflow flag is also set because adding two negative numbers results in a positive number. It is up to the programmer to determine how they want to consider these operations, usually through choice of conditional branch instructions.

**Branch Statements:**

Branch statements are statements that allow a program to go forward or backwards to a particular section of code. There are two types of branch instructions, unary branch instructions that always or never branch regardless of the values in the condition code register (CCR), and conditional branch instructions that branch based on bits that are set and clear in the CCR. Refer to the following table from the reference manual:

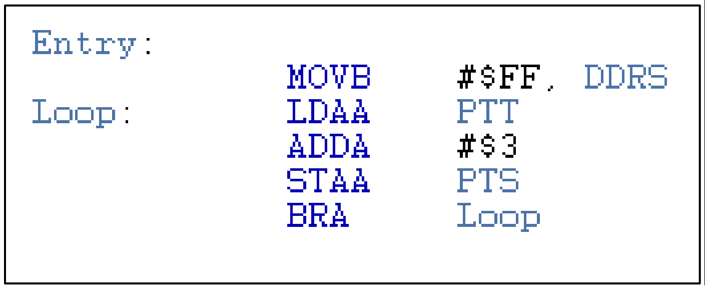
**Table 5-17. Short Branch Instructions**



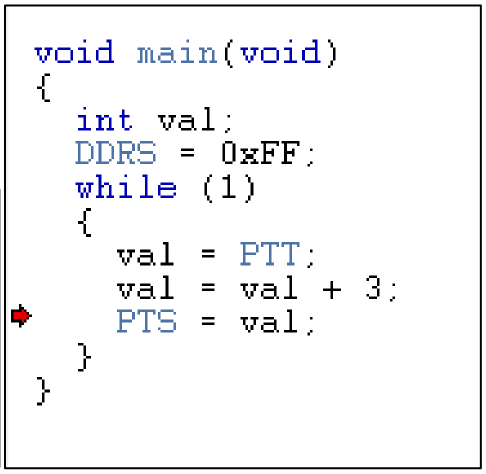
| Mnemonic | Function | | Equation or Operation |
|---|---|---|---|
| **Unary Branches** | | | |
| BRA | Branch always | | 1 = 1 |
| BRN | Branch never | | 1 = 0 |
| **Simple Branches** | | | |
| BCC | Branch if carry clear | | C = 0 |
| BCS | Branch if carry set | | C = 1 |
| BEQ | Branch if equal | | Z = 1 |
| BMI | Branch if minus | | N = 1 |
| BNE | Branch if not equal | | Z = 0 |
| BPL | Branch if plus | | N = 0 |
| BVC | Branch if overflow clear | | V = 0 |
| BVS | Branch if overflow set | | V = 1 |
| **Unsigned Branches** | | | |
| | | **Relation** | |
| BHI | Branch if higher | R > M | C + Z = 0 |
| BHS | Branch if higher or same | R ≥ M | C = 0 |
| BLO | Branch if lower | R < M | C = 1 |
| BLS | Branch if lower or same | R ≤ M | C + Z = 1 |
| **Signed Branches** | | | |
| BGE | Branch if greater than or equal | R ≥ M | N ⊕ V = 0 |
| BGT | Branch if greater than | R > M | Z + (N ⊕ V) = 0 |
| BLE | Branch if less than or equal | R ≤ M | Z + (N ⊕ V) = 1 |
| BLT | Branch if less than | R < M | N ⊕ V = 1 |

Branch statements are capable of altering the program counter in the range from -128 to +127 bytes from the current value of the program counter. In addition to the instructions above, each branch instruction has a corresponding long branch instruction that can be implemented by simply adding an L in front of the corresponding branch instruction (for example, `LBRA`). Long branch instructions are capable of changing the program counter to any location in memory (0 to 0xFFFF).

The only branch instructions that will be used for this experiment are unary branch instructions. Refer to the code below to see how a program can be continuously run. The following two excerpts are for Assembly and C respectively.



```
Entry:
                MOVB    #$FF, DDRS
Loop:           LDAA    PTT
                ADDA    #$3
                STAA    PTS
                BRA     Loop
```



```
void main(void)
{
  int val;
  DDRS = 0xFF;
  while (1)
  {
    val = PTT;
    val = val + 3;
    PTS = val;
  }
}
```

The above two programs do the same thing. A value is loaded from the dipswitches at Port T. Then 3 is added to this value and the result is stored in Port S. Rather than give labels to the addresses of these ports and data direction registers, predefined labels from the .inc and .h file are implemented. Both of the programs are capable of running continuously on the I/O board. Previous experiments had students step through the program and record the result. In future experiments, the programs that are written in lab will be continuously run, so it is important to become comfortable with using unary branch instructions now.

**Experimental Procedure:**

**Lab 3.1.1: Straight Line**

Create a new project for this lab. Write and test an assembly language program that solves the following straight line equation for y:

$$y = mx + b$$

Generate an integer result as close as possible that solves the following straight-line equation for y. Generate an integer result as close as possible to the actual answer. Assume variable x is always an unsigned 8-bit number.

The slope (m) is equal to 0.68 and the offset (b) is equal to 12. Leave the result in accumulator A. Assume value x is in the memory location labeled `Val`. Use assembler directives to define these values. Note that the result will always fit in 8 bits.

Test and confirm the program is working properly by testing the following values in `Val`. Modify `Val` using the data window in the debugger.

| | |
|---|---|
| `Val = 0` | Result: ____________________ |
| `Val = 10` | Result: ____________________ |
| `Val = 75` | Result: ____________________ |
| `Val = 200` | Result: ____________________ |
| `Val = 255` | Result: ____________________ |

**Extra Credit:** The number after the decimal point will always be truncated off of the result. The result will not be rounded up. For extra credit on the lab checkout, display the number to the left of the decimal point in register A and the result to the right of the decimal point in register B.

## Lab 3.1.2: Parabolic Line

Write and test an assembly language program that solves the following equation for y. Assume variable x is an unsigned 8-bit number.

$$y = mx^2 + b$$

The slope (m) is equal to 0.68 and the offset (b) is equal to 12. Leave the result in register D. Assume value x is in the memory location labeled `Val`. Use assembler directives to define these values. Note that the result will always fit in 16 bits.

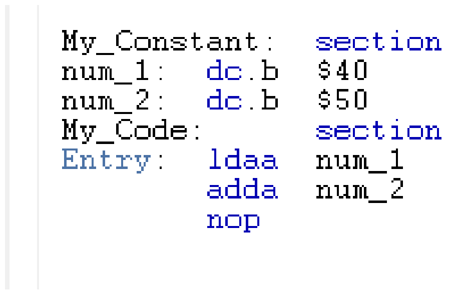
Test and confirm the program is working properly by testing the following values in `Val`. Modify `Val` using the data window in the debugger.

| | |
|---|---|
| `Val = 0` | Result: ____________________ |
| `Val = 10` | Result: ____________________ |
| `Val = 75` | Result: ____________________ |
| `Val = 200` | Result: ____________________ |
| `Val = 255` | Result: ____________________ |

**Hint:** Using the scaling approach outlined in this experiment will result in a number that is larger than the available registers of the HC(S)12. The above program can be written without having to write a complex algorithm to handle a large 32-bit number. Use the reference manual to select the correct arithmetic instruction. If done correctly, the program will return an accurate result.

**Laboratory 3.2.1: Condition Code Register**

The program below defines `num_1` as a byte containing `0x40` and `num_2` as a byte containing `0x50`. The program loads accumulator A with `num_1` and then adds `num_2` to the accumulator.



```
My_Constant:  section
num_1:  dc.b  $40
num_2:  dc.b  $50
My_Code:      section
Entry:  ldaa  num_1
        adda  num_2
        nop
```

Start a new project and enter the code above. Assemble and run your program. Step through the program (until the `nop` instruction is reached) and show the contents of Accumulator A after program execution.

A = ____________________

Show the states of the following bits of the Condition Code Register (CCR) after execution.

| | | |
|---|---|---|
| Carry Flag (C) | Set [ ] | Clear [ ] |
| Overflow Flag (V) | Set [ ] | Clear [ ] |
| Negative Flag (N) | Set [ ] | Clear [ ] |
| Zero Flag (Z) | Set [ ] | Clear [ ] |

Now change `num_1` to `0xF6` and `num_2` to `0xEC` and run the program again. The values for `num_1` and `num_2` will have to be changed in the program and the program will have to be reassembled and reloaded into memory. Run the program and fill in the blanks below.

What are the contents of Accumulator A after program execution? ____________________

Show the states of the following bits of the Condition Code Register (CCR) after execution.

| | | |
|---|---|---|
| Carry Flag (C) | Set [ ] | Clear [ ] |
| Overflow Flag (V) | Set [ ] | Clear [ ] |
| Negative Flag (N) | Set [ ] | Clear [ ] |
| Zero Flag (Z) | Set [ ] | Clear [ ] |

Change `num_2` to `0x57` and run the program again. Fill in the blanks below.

What are the contents of Accumulator A after program execution? ____________________

Show the states of the following bits of the Condition Code Register (CCR) after execution.

| | | |
|---|---|---|
| Carry Flag (C) | Set [ ] | Clear [ ] |
| Overflow Flag (V) | Set [ ] | Clear [ ] |
| Negative Flag (N) | Set [ ] | Clear [ ] |
| Zero Flag (Z) | Set [ ] | Clear [ ] |

Change `num_1` to `0xE9` and `num_2` to `0x89` and run the program again. Fill in the blanks below.

What are the contents of Accumulator A after program execution? ____________________

Show the states of the following bits of the Condition Code Register (CCR) after execution.

| | | |
|---|---|---|
| Carry Flag (C) | Set [ ] | Clear [ ] |
| Overflow Flag (V) | Set [ ] | Clear [ ] |
| Negative Flag (N) | Set [ ] | Clear [ ] |
| Zero Flag (Z) | Set [ ] | Clear [ ] |

Change the program so that both `num_1` and `num_2` are `0x3F` and change the `ADDA` instruction to a `SUBA` instruction. Reassemble and run the program. Fill in the blanks below.

What are the contents of Accumulator A after program execution? ____________________

Show the states of the following bits of the Condition Code Register (CCR) after execution.

| | | |
|---|---|---|
| Carry Flag (C) | Set [ ] | Clear [ ] |
| Overflow Flag (V) | Set [ ] | Clear [ ] |
| Negative Flag (N) | Set [ ] | Clear [ ] |
| Zero Flag (Z) | Set [ ] | Clear [ ] |

Change the above program so that `NUM_2` is `0x90`. Run the program and fill in the blanks below.

What are the contents of Accumulator A after program execution? ____________________

Show the states of the following bits of the Condition Code Register (CCR) after execution.

| | | |
|---|---|---|
| Carry Flag (C) | Set [ ] | Clear [ ] |
| Overflow Flag (V) | Set [ ] | Clear [ ] |
| Negative Flag (N) | Set [ ] | Clear [ ] |
| Zero Flag (Z) | Set [ ] | Clear [ ] |

## Laboratory 3.2.2:

Write a short program that loads accumulator A with `0x12` and defines a variable `VAR_1` as a byte of storage initialized with `0x30`. The code to define a byte is given below. The program should then use the `CMPA` instruction to compare the value in accumulator A to `VAR_1`. The `CMPA` instruction compares the two values by "subtracting" `VAR_1` from the value in accumulator A, but does not change either value. Run the program and fill in the blanks below.

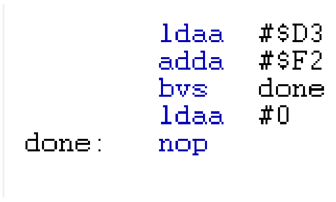
What are the contents of accumulator A after program execution? ____________________

Show the states of the following bits of the Condition Code Register (CCR) after execution.

| | | |
|---|---|---|
| Carry Flag (C) | Set [ ] | Clear [ ] |
| Overflow Flag (V) | Set [ ] | Clear [ ] |
| Negative Flag (N) | Set [ ] | Clear [ ] |
| Zero Flag (Z) | Set [ ] | Clear [ ] |

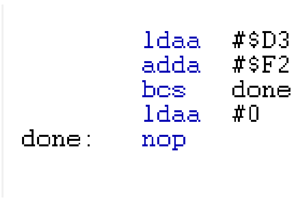
## Laboratory 3.2.3:

Enter, build, and run the following programs and determine the value in accumulator A when the `NOP` instruction is reached.



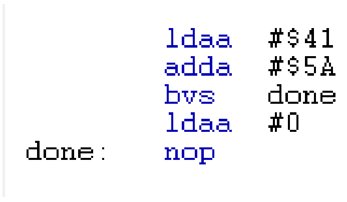
```
        ldaa   #$D3
        adda   #$F2
        bvs    done
        ldaa   #0
done:   nop
```

What are the contents of Accumulator A after program execution? ____________________



```
        ldaa   #$D3
        adda   #$F2
        bcs    done
        ldaa   #0
done:   nop
```

What are the contents of Accumulator A after program execution? ____________________



```
        ldaa   #$41
        adda   #$5A
        bvs    done
        ldaa   #0
done:   nop
```

What are the contents of Accumulator A after program execution? ____________________

## Laboratory 3.3: Table Lookup

Write a program that repeatedly reads the switches and writes the following values to the LEDs for the switch inputs (using a lookup table). The left value is the switch input and the right value is sent to the LEDs.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| \$0 | → | \$12 | \$1 | → | \$18 | \$2 | → | \$24 | \$3 | → | \$36 |
| \$4 | → | \$3A | \$5 | → | \$43 | \$6 | → | \$4B | \$7 | → | \$51 |
| \$8 | → | \$61 | \$9 | → | \$6F | \$A | → | \$7A | \$B | → | \$92 |
| \$C | → | \$B6 | \$D | → | \$C3 | \$E | → | \$D6 | \$F | → | \$F1 |

**Hint:** To successfully write an efficient version of the above program, think about the concepts learned in the first two labs, most importantly the different addressing modes and directives that are available for use.