Review Exam 1

- **Numbers**
  - Decimal to Hex (signed and unsigned)
  - Hex to Decimal (signed and unsigned)
  - Binary to Hex
  - Hex to Binary
  - Addition and subtraction of fixed-length hex numbers
  - Overflow, Carry, Zero, Negative bits of CCR

- **Programming Model**
  - Internal registers – A, B, (D = AB), X, Y, SP, PC, CCR

- **Addressing Modes and Effective Addresses**
  - INH, IMM, DIR, EXT, REL, IDX (Not Indexed Indirect)
  - How to determine effective address

- **Instructions**
  - What they do - Core Users Guide
  - What machine code is generated
  - How many cycles to execute
  - Effect on CCR
  - Branch instructions – which to use with signed and which with unsigned

- **Machine Code**
  - Reverse Assembly

- **Stack and Stack Pointer**
  - What happens to stack and SP for instructions (e.g., PSHX, JSR)
  - How the SP is used in getting to and leaving subroutines

- **Assembly Language**
  - Be able to read and write simple assembly language program
  - Know basic assembler directives – e.g., equ, dc.b, ds.w
  - Flow charts
Binary, Hex and Decimal Numbers (4-bit representation)

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>B</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>C</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>D</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>E</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>F</td>
<td>15</td>
</tr>
</tbody>
</table>

**Binary to Unsigned Decimal:**
Convert Binary to Unsigned Decimal

1110111₂

\[1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0\]
\[1 \times 64 + 1 \times 32 + 1 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1\]
123₁₀

**Hex to Unsigned Decimal**
Convert Hex to Unsigned Decimal

82D6₁₆

\[8 \times 16^3 + 2 \times 16^2 + 13 \times 16^1 + 6 \times 16^0\]
\[8 \times 4096 + 2 \times 256 + 13 \times 16 + 6 \times 1\]
33494₁₀
Unsigned Decimal to Hex
Convert Unsigned Decimal to Hex

| Division | Q | R
|----------|---|---
| 721/16   | 45| 1
| 45/16    | 2 | 13
| 2/16     | 0 | 2

\[721_{10} = 2D1_{16}\]

Signed Number Representation in 2’s Complement Form:

If the most significant bit (MSB) is 0 (most significant hex digit 0–7), then the number is positive.
Get decimal equivalent by converting number to decimal, and use the positive (+) sign.

Example for 8–bit number:

3A \(_{16}\) \(\rightarrow\) + \((3 \times 16^1 + 10 \times 16^0)_{10}\)
+ \((3 \times 16 + 10 \times 1)_{10}\)
+ 58 \(_{10}\)

If the most significant bit is 1 (most significant hex digit 8–F), then the number is negative.
Get decimal equivalent by taking 2’s complement of number, converting to decimal, and using negative (–) sign.

Example for 8–bit number:

A3 \(_{16}\) \(\rightarrow\) - \((5D)_{16}\)
- \((5 \times 16^1 + 13 \times 16^0)_{10}\)
- \((5 \times 16 + 13 \times 1)_{10}\)
- 93 \(_{10}\)
One’s complement table makes it simple to finding 2’s complements

<table>
<thead>
<tr>
<th></th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>E</td>
</tr>
<tr>
<td>2</td>
<td>D</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td>5</td>
<td>A</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

One’s complement

To take two’s complement, add one to one’s complement.

Take two’s complement of D0C3:

\[2F3C + 1 = 2F3D\]
Addition and Subtraction of Binary and Hexadecimal Numbers

Setting the C (Carry), V (Overflow), N (Negative) and Z (Zero) bits

How the C, V, N and Z bits of the CCR are changed

N bit is set if result of operation is negative (MSB = 1)

Z bit is set if result of operation is zero (All bits = 0)

V bit is set if operation produced an overflow

C bit is set if operation produced a carry (borrow on subtraction)

Note: Not all instructions change these bits of the CCR

Addition of Hexadecimal Numbers

ADDITION:

C bit set when result does not fit in word

V bit set when P + P = N or N + N = P

N bit set when MSB of result is 1

Z bit set when result is 0

\[
\begin{array}{ccc}
\text{AC} & +72 \\
7A & 1E \\
----- & ----- \\
C: 0 & C: 1 \\
V: 1 & V: 0 \\
N: 1 & N: 0 \\
Z: 0 & Z: 0
\end{array}
\]
Subtraction of Hexadecimal Numbers

SUBTRACTION:

C bit set on borrow (when the magnitude of the subtrahend is greater than the minuend)

V bit set when N - P = P or P - N = N

N bit set when MSB is 1

Z bit set when result is 0

\[
\begin{array}{c|c|c}
7A & 2C \\
-5C & -72 \\
---- & ------ \\
1E & BA \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
C: & 0 & C: & 1 \\
V: & 0 & V: & 0 \\
N: & 0 & N: & 1 \\
Z: & 0 & Z: & 0 \\
\end{array}
\]
Programming Model

<table>
<thead>
<tr>
<th>8-BIT ACCUMULATORS A AND B</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR 16-BIT DOUBLE ACCUMULATOR D</td>
</tr>
<tr>
<td>INDEX REGISTER X</td>
</tr>
<tr>
<td>INDEX REGISTER Y</td>
</tr>
<tr>
<td>STACK POINTER</td>
</tr>
<tr>
<td>PROGRAM COUNTER</td>
</tr>
<tr>
<td>CONDITION CODE REGISTER</td>
</tr>
</tbody>
</table>

Figure 2-1. Programming Model
Addressing Modes and Effective Addresses

The Inherent (INH) addressing mode

Instructions which work only with registers inside ALU

ABA ; Add B to A (A) + (B) → A
18 06

CLRA ; Clear A 0 → A
87

The HC12 does not access memory - There is no effective address

The Extended (EXT) addressing mode

Instructions which give the 16–bit address to be accessed

LDAA $1000 ; ($1000) → A
B6 10 00 Effective Address: $1000

STAB $1003 ; (B) → $1003
7B 10 03 Effective Address: $1003

Effective address is specified by the two bytes following op code
The Direct (DIR) addressing mode

Direct (DIR) Addressing Mode

Instructions which give 8 LSB of address (8 MSB all 0)

LDAA $20 ; ($0020) → A
96 20 Effective Address: $0020

STX $21 ; (X) → $0021:$0022
5E 21 Effective Address: $0021

8-LSB of effective address is specified by byte following op code

The Immediate (IMM) addressing mode

Value to be used is part of instruction

LDAA #$17 ; $17 → A
B6 17 Effective Address: PC + 1

ADDA #10 ; (A) + $0A → A
8B 0A Effective Address: PC + 1

Effective address is the address following the op code

The Indexed (IDX, IDX1, IDX2) addressing mode

Effective address is obtained from X or Y register (or SP or PC)

LDAA 0,X ; Use (X) as address to get value to put in A
A6 00 Effective address: contents of X

INC 2,X− ; Post−decrement Indexed
; Increment the number at address (X),
; then subtract 2 from X
62 3E Effective address: contents of X
## Indexed Addressing Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Example</th>
<th>Effective Address</th>
<th>Offset</th>
<th>Value in X After Done</th>
<th>Registers To Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant Offset</td>
<td>LDAA n,X</td>
<td>(X)+n</td>
<td>0 to FFFF</td>
<td>(X)</td>
<td>X, Y, SP, PC</td>
</tr>
<tr>
<td>Constant Offset</td>
<td>LDAA -n,X</td>
<td>(X)−n</td>
<td>0 to FFFF</td>
<td>(X)</td>
<td>X, Y, SP, PC</td>
</tr>
<tr>
<td>Postincrement</td>
<td>LDAA n,X+</td>
<td>(X)</td>
<td>1 to 8</td>
<td>(X)+n</td>
<td>X, Y, SP</td>
</tr>
<tr>
<td>Preincrement</td>
<td>LDAA n,+X</td>
<td>(X)+n</td>
<td>1 to 8</td>
<td>(X)+n</td>
<td>X, Y, SP</td>
</tr>
<tr>
<td>Postdecrement</td>
<td>LDAA n,X−</td>
<td>(X)</td>
<td>1 to 8</td>
<td>(X)−n</td>
<td>X, Y, SP</td>
</tr>
<tr>
<td>Predecrement</td>
<td>LDAA n,−X</td>
<td>(X)−n</td>
<td>1 to 8</td>
<td>(X)−n</td>
<td>X, Y, SP</td>
</tr>
<tr>
<td>ACC Offset</td>
<td>LDAA A,X</td>
<td>(X)+(A)</td>
<td>0 to FF</td>
<td>(X)</td>
<td>X, Y, SP, PC</td>
</tr>
<tr>
<td></td>
<td>LDAA B,X</td>
<td>(X)+(B)</td>
<td>0 to FF</td>
<td>(X)</td>
<td>X, Y, SP, PC</td>
</tr>
<tr>
<td></td>
<td>LDAA D,X</td>
<td>(X)+(D)</td>
<td>0 to FFFF</td>
<td>(X)</td>
<td>X, Y, SP, PC</td>
</tr>
</tbody>
</table>
Relative (REL) Addressing Mode

The relative addressing mode is used only in branch instructions.

Branch instruction: One byte following op code specifies how far to branch

Treat the offset as a signed number; add the offset to the address following the current instruction to get the address of the instruction to branch to

(BRA) 20 35 \( \text{PC} + 2 + 0035 \rightarrow \text{PC} \)

(BRA) 20 C7 \( \text{PC} + 2 + \text{FFC7} \rightarrow \text{PC} \)
\( \text{PC} + 2 - 0039 \rightarrow \text{PC} \)

Long branch instruction: Two bytes following op code specifies how far to branch

Treat the offset as an unsigned number; add the offset to the address following the current instruction to get the address of the instruction to branch to

(LBEQ) 18 27 02 1A If \( Z = 1 \) then \( \text{PC} + 4 + 021A \rightarrow \text{PC} \)
If \( Z = 0 \) then \( \text{PC} + 4 \rightarrow \text{PC} \)

When writing assembly language program, you don’t have to calculate offset
You indicate what address you want to go to, and the assembler calculates the offset
Instructions: Machine code, how many cycles to execute, effect on CCR, and branch instructions

CPU cycles of the 68HC12

- 68HC12 works on 48 MHz clock
- Each processor cycle takes 41.7 ns (1/24 MHz) to execute
- You can determine how many cycles an instruction takes by looking up the CPU cycles for that instruction in the Core Users Guide.

<table>
<thead>
<tr>
<th>Address</th>
<th>Inst</th>
<th>Mode</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>org $2000 ; Inst</td>
<td>Mode</td>
<td>Cycles</td>
</tr>
<tr>
<td>2000</td>
<td>C6 0A</td>
<td>ldab #10 ; LDAB</td>
<td>(IMM) 1</td>
</tr>
<tr>
<td>2002</td>
<td>87</td>
<td>clra ; CLRA</td>
<td>(INH) 1</td>
</tr>
<tr>
<td>2003</td>
<td>04 31 FC</td>
<td>dbne b,loop ; DBNE</td>
<td>(REL) 3</td>
</tr>
<tr>
<td>2006</td>
<td>3F</td>
<td>swi ; SWI</td>
<td>9</td>
</tr>
</tbody>
</table>

The program executes the ldab #10 instruction once (which takes one cycle). It then goes through loop 10 times (which has two instructions, one with one cycle and one with three cycles), and finishes with the swi instruction (which takes 9 cycles).

Total number of cycles:

\[ 1 + 10 \times (1 + 3) + 9 = 50 \]

50 cycles = 50 \times 41.7 \text{ ns/cycle} = 2.08 \mu\text{s}
Effects of instructions on CCR during execution of program.

Table A-1. Instruction Set Summary (Sheet 3 of 14)

<table>
<thead>
<tr>
<th>Source Form</th>
<th>Operation Description</th>
<th>Addr. Mode</th>
<th>Machine Coding (hex)</th>
<th>Access Detail</th>
<th>SXHI</th>
<th>NZVC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLT, SLT, SLT</td>
<td>Branch if lower than (F.C + Z - 1) (unsigned)</td>
<td>REL 23 tt</td>
<td>PP/P</td>
<td>PP/P</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>BMI, BMI, BMI</td>
<td>Branch if Minus (F.N + 1) (signed)</td>
<td>REL 26 tt</td>
<td>PP/P</td>
<td>PP/P</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>BNE, BNE, BNE</td>
<td>Branch if Not Equal (F.Z = 0)</td>
<td>REL 26 tt</td>
<td>PP/P</td>
<td>PP/P</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>BPL, BPL, BPL</td>
<td>Branch if Plus (F.N = 0)</td>
<td>REL 24 tt</td>
<td>PP/P</td>
<td>PP/P</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>BHI, BHI, BHI</td>
<td>Branch if Always (F.1 = 1)</td>
<td>REL 20 xx</td>
<td>PP/P</td>
<td>PP/P</td>
<td>----</td>
<td>----</td>
</tr>
</tbody>
</table>

6.3 Condition Code Changes

The following special characters are used to describe the effects of instruction execution on the status bits in the condition code register.

- - Status bit not affected by operation
0 - Status bit cleared by operation
1 - Status bit set by operation
\(\wedge\) - Status bit affected by operation
\(\downarrow\) - Status bit may be cleared or remain set, but is not set by operation.
\(\uparrow\) - Status bit may be set or remain cleared, but is not cleared by operation.
? - Status bit may be changed by operation, but the final state is not defined.
! - Status bit used for a special purpose
Which branch instruction should you use?

Branch if $A > B$

Is $0xFF > 0x00$?

If unsigned, $0xFF = 255$ and $0x00 = 0$,
so $0xFF > 0x00$

If signed, $0xFF = -1$ and $0x00 = 0$,
so $0xFF < 0x00$

Using unsigned numbers: BHI

Using signed numbers: BGT

For unsigned numbers, use branch instructions which checks C bit

For signed numbers, use branch instructions which checks V bit
Reverse Assembly

Disassembly of an HC12 Program

• It is sometimes useful to be able to convert HC12 op codes into mnemonics.

For example, consider the hex code:

<table>
<thead>
<tr>
<th>ADDR DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000 C6 05 CE 20 00 E6 01 18 06 04 35 EE 3F</td>
</tr>
</tbody>
</table>

• To determine the instructions, use:

  Table A-2 of the HCS12 Core Users Guide to start with.
  Table A-3 for indexed addressing mode
  Table A-5 for transfer and exchange instructions
  Table A-6 for postbyte encoding.

• Use up all the bytes for one instruction, then go on to the next instruction.

C6 05 ⇒ LDAA #$05 two-byte LDAA, IMM addressing mode
CE 20 00 ⇒ LDX #$2000 three-byte LDX, IMM addressing mode
E6 01 ⇒ LDAB 1,X two to four-byte LDAB, IDX addressing mode. Operand 01 => 1,X, a 5b constant offset which uses only one postbyte
18 06 ⇒ ABA two-byte ABA, INH addressing mode
04 35 EE ⇒ DBNE X,(-18) three-byte loop instruction Postbyte 35 indicates DBNE X, negative
3F ⇒ SWI one-byte SWI, INH addressing mode
Stack and Stack Pointer

THE STACK AND THE STACK POINTER

• When we use subroutines and interrupts it is essential to have such a storage region.

• Such a region is called a Stack.

• The Stack Pointer (SP) register is used to indicate the location of the last item put onto the stack.

• When you put something onto the stack (push onto the stack), the SP is decremented before the item is placed on the stack.

• When you take something off of the stack (pull from the stack), the SP is incremented after the item is pulled from the stack.

• Before you can use a stack you have to initialize the Stack Pointer to point to one value higher than the highest memory location in the stack.

• Use the LDS (Load Stack Pointer) instruction to initialize the stack point.

• The LDS instruction is usually the first instruction of a program which uses the stack.

• The stack pointer is initialized only one time in the program.
**PSHA**  
**Push A onto Stack**

**Operation:**  
$(SP) - 0001 \Rightarrow SP$  
$(A) \Rightarrow M_{(SP)}$

**Description:** Stacks the content of accumulator A. The stack pointer is decremented by one. The content of A is then stored at the address the SP points to. 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.

**CCR Details:**

<table>
<thead>
<tr>
<th>S</th>
<th>X</th>
<th>H</th>
<th>I</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Source Form** | **Address Mode** | **Object Code** | **Access Detail**
--- | --- | --- | ---
PSHA | INH | 3C | SP

---

**PSHX**  
**Push Index Register X onto Stack**

**Operation:**  
$(SP) - 0002 \Rightarrow SP$  
$(X_{hi}, X_{lo}) \Rightarrow M_{(SP)}, M_{(SP+1)}$

**Description:** Stacks the content of index register X. The stack pointer is decremented by two. The content of X is then stored at the address to which the SP points. After PSHX executes, the SP points to the stacked value of the high-order half of X. 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.

**CCR Details:**

<table>
<thead>
<tr>
<th>S</th>
<th>X</th>
<th>H</th>
<th>I</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Source Form** | **Address Mode** | **Object Code** | **Access Detail**
--- | --- | --- | ---
PSHX | INH | 34 | 02

---
**PULA**

**Pull A from Stack**

**Operation:**

\[
(M_{SP}) \Rightarrow A \\
(SP) + \$0001 \Rightarrow SP
\]

**Description:**

Accumulator A is loaded from the address indicated by the stack pointer. The SP is then incremented by one.

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.

**CCR Details:**

\[
\begin{array}{cccccccc}
S & X & H & I & N & Z & V & C \\
- & - & - & - & - & - & - & -
\end{array}
\]

<table>
<thead>
<tr>
<th>Source Form</th>
<th>Address Mode</th>
<th>Object Code</th>
<th>Access Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>PULA</td>
<td>INH</td>
<td>93</td>
<td>ufo</td>
</tr>
</tbody>
</table>

**PULX**

**Pull Index Register X from Stack**

**Operation:**

\[
(M_{SP}; M_{SP+1}) \Rightarrow X_H : X_L \\
(SP) + \$0002 \Rightarrow SP
\]

**Description:**

Index register X is loaded from the address indicated by the stack pointer. The SP is then incremented by two.

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.

**CCR Details:**

\[
\begin{array}{cccccccc}
S & X & H & I & N & Z & V & C \\
- & - & - & - & - & - & - & -
\end{array}
\]

<table>
<thead>
<tr>
<th>Source Form</th>
<th>Address Mode</th>
<th>Object Code</th>
<th>Access Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>PULX</td>
<td>INH</td>
<td>30</td>
<td>ufo</td>
</tr>
</tbody>
</table>
Subroutines

• A subroutine is a section of code which performs a specific task, usually a task which needs to be executed by different parts of a program.

• Example:
  – Math functions, such as square root

• Because a subroutine can be called from different places in a program, you cannot get out of a subroutine with an instruction such as jmp label because you would need to jump to different places depending upon which section of code called the subroutine.

• When you want to call the subroutine your code has to save the address where the subroutine should return to. It does this by saving the return address on the stack.
  – This is done automatically for you when you get to the subroutine by using the JSR (Jump to Subroutine) or BSR (Branch to Subroutine) instruction. This instruction pushes the address of the instruction following the JSR/BSR instruction on the stack.

• After the subroutine is done executing its code it needs to return to the address saved on the stack.
  – This is done automatically for you when you return from the subroutine by using the RTS (Return from Subroutine) instruction. This instruction pulls the return address off of the stack and loads it into the program counter, so the program resumes execution of the program with the instruction following that which called the subroutine.

The subroutine will probably need to use some MC9S12 registers to do its work. However, the calling code may be using its registers for some reason - the calling code may not work correctly if the subroutine changes the values of the HC12 registers.

  – To avoid this problem, the subroutine should save the MC9S12 registers before it uses them, and restore the MC9S12 registers after it is done with them.
BSR  

Branch to Subroutine

**Operation**

\[(SP) - \$0002 \Rightarrow SP\]
\[RTN_j; RTN_i \Rightarrow M_{gp}; M_{gp+1}\]
\[(PC) + \$0002 + \text{rel} \Rightarrow PC\]

Sets up conditions to return to normal program flow, then transfers control to a subroutine. Uses the address of the instruction after the BSR as a return address.

Decrements the SP by two, to allow the two bytes of the return address to be stacked.

Stacks the return address (the SP points to the high byte of the return address).

Branches to a location determined by the branch offset.

Subroutines are normally terminated with an RTS instruction, which restores the return address from the stack.

**CCR Effects**

<table>
<thead>
<tr>
<th>S</th>
<th>X</th>
<th>H</th>
<th>I</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Code and CPU Cycles**

<table>
<thead>
<tr>
<th>Source Form</th>
<th>Address Mode</th>
<th>Machine Code (Hex)</th>
<th>CPU Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSR rel$2$</td>
<td>REL</td>
<td>07 $rt$</td>
<td>5000</td>
</tr>
</tbody>
</table>

---

RTS  

Return from Subroutine

**Operation**

\[(M_{gp}; M_{gp+1}) \Rightarrow PC_H; PC_L\]
\[(SP) + \$0002 \Rightarrow SP\]

Restores the value of PC from the stack and increments SP by two. Program execution continues at the address restored from the stack.

**CCR Effects**

<table>
<thead>
<tr>
<th>S</th>
<th>X</th>
<th>H</th>
<th>I</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Code and CPU Cycles**

<table>
<thead>
<tr>
<th>Source Form</th>
<th>Address Mode</th>
<th>Machine Code (Hex)</th>
<th>CPU Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTS</td>
<td>INH</td>
<td>3D</td>
<td>5000</td>
</tr>
</tbody>
</table>
Example of a subroutine to delay for a certain amount of time

; Subroutine to wait for 100 ms

delay:   ldaa  #100       ; execute outer loop 100 times
loop2:   ldx  #8000      ; want inner loop to last 1ms
loop1:   dbne  x,loop1   ; inner loop – 3 cycles x 8000 times
         dbne  a,loop2
         rts

• Want inner loop to last for 1 ms. MC9S12 runs at 24,000,000 cycles/second, so 1 ms is 24,000 cycles.

• Inner loop should be 24,000 cycles/(3 cycles/loop) = 8,000 loops (times)

• Problem: The subroutine changes the values of registers A and X

• To solve this problem, save the values of A and X on the stack before using them, and restore them before returning.

; Subroutine to wait for 100 ms

delay:   psha
         pshx
         ldaa  #100       ; save registers
loop2:   ldx  #8000      ; execute outer loop 100 times
loop1:   dbne  x,loop1   ; want inner loop to last 1ms
         dbne  a,loop2   ; inner loop – 3 cycles x 8000 times
         pulx
         pula
         rts
         ; restore registers