- **Addition and Subtraction of Hexadecimal Numbers**
- **Simple assembly language programming**
  - A simple Assembly Language Program
  - Assembling an Assembly Language Program
  - Simple 9S12 programs
  - Hex code generated from a simple 9S12 program
  - Things you need to know for 9S12 assembly language programming

- **Introduction to Addressing Modes**
  - Most instructions operate on data in memory
  - Addressing mode used to find address of data in memory
  - MC9S12 Addressing modes: Inherent, Extended, Direct, Immediate, Indexed, and Relative Modes

A Simple MC9S12 Program

• All programs and data must be placed in memory between address **0x1000** and **0x3BFF**. For our programs we will put the first instruction at **0x2000**, and the first data byte at **0x1000**

• Consider the following program:

```
ldaa $1000      ; Put contents of memory at 0x1000 into A
inca            ; Add one to A
staa $1001      ; Store the result into memory at 0x1001
swi             ; End program
```

• If the first instruction is at address 0x2000, the following bytes in memory will tell the MC9S12 to execute the above program:

Address	Value	Instruction
0x2000	B6	ldaa $1000
0x2001	10	
0x2002	00	
0x2003	42	inca
0x2004	7A	staa $1001
0x2005	10	
0x2006	01	
0x2007	3F	swi

• If the contents of address 0x1000 were 0xA2, the program would put an 0xA3 into address 0x1001.

A Simple Assembly Language Program.

• It is difficult for humans to remember the numbers (op codes) for computer instructions. It is also hard for us to keep track of the addresses of numerous data values. Instead we use words called mnemonics to represent instructions, and labels to represent addresses, and let a computer programmer called an assembler to convert our program to binary numbers (machine code).

• Here is an assembly language program to implement the previous program:

```
prog   equ $2000        ; Start program at 0x2000
data   equ $1000        ; Data value at 0x1000

         org   prog

         ldaa   input
         inca
         staa   result
         swi

org     data
input: dc.b  $A2
result: ds.b  1
```

• We would put this code into a file and give it a name, such as **main.s**. (Assembly language programs usually have the extension .s or .asm.)

• Note that equ, org, dc.b and ds.b are not instructions for the MC9S12 but are **directives** to the assembler which make it possible for us to write assembly language programs. There are called assembler directives or psuedo-ops. For example the pseudo-op org tells the assembler that the starting address (origin) of our program should be 0x2000.

Assembling an Assembly Language Program

• A computer program called an assembler can convert an assembly language program into machine code.

• The assembler we use in class is a commercial compiler from Freescale called CodeWarrior (Eclipse IDE).

• The assembler will produce a file called **main.lst**, which shows the machine code generated.

```
Freescale HC12-Assembler
(c) Copyright Freescale 1987-2009

Abs.    Rel.    Loc     Obj. code       Source line
----     ----   ------   ---------       -----------
1       1
2       2       0000   2000             prog   equ     $2000 ; Start program at 0x2000
3       3       0000   1000             data    equ     $1000 ; Data value at 0x1000
4       4
5       5                               org      prog
6       6
7       7       a002000 B610 00         ldaa    input
8       8       a002003 42              inca
9       9       a002004 7A10 01         staa    result
10     10       a002007 3F              swi
11     11
12     12                               org      data
13     13       a001000 A2              input:  dc.b   $A2
14     14       a001001                 result: ds.b   1
```

This will produce a file called Project.abs.s19 which we load into the MC9S12.

```
S06B0000433A5C446F63756D656E747320616E642053657474696E6773
S1051000A20048
S10B2000B61000427A10013F02
S9030000FC
```

- The first line of the S19 file starts with a S0: the **S0** indicates that it **is the first line**.
  - The first line form CodeWarrior is too long for the DBug-12 monitor. You will need to delete it before loading the file into the MC9S12.
- The last line of the S19 file starts with a S9: the **S9** indicates that it **is the last line**.
- The other lines begin with a S1: the **S1** indicates these lines **are data** to be loaded into the MC9S12 memory.
- Here is the second line (with some spaces added):

  S1 0B 2000 B6 1000 42 7A 1001 3F 02

- On the second line, the S1 if followed by a **0B**. This tells the loader that there this line has 11 (0x0B) bytes of data follow.
- The count 0B is followed by **2000**. This tells the loader that the data (program) should be put into memory starting with address 0x2000.
- The next 16 hex numbers B61000427A10013F are the 8 bytes to be loaded into memory. You should be able to find these bytes in the **main.lst** file.
- The last two hex numbers, **0x02**, is a one byte checksum, which the loader can use to make sure the data was loaded correctly.

What will this program do?

C:\Users\Hector\Downloads\Chapter 03\miniide\example1_17.lst - generated by MGTEK Assembler ASM12 V1.26 Build 144 for WIN32 (x86) - Wed Jan 25 08:56:57 2017

```
    1:        =0000200B          prog:   equ $200B
    2:        =0000100F          data:   equ $100F
    3:
    4:        =0000200B                  org prog
    5:    200B B6 100F                   ldaa input
    6:    200E 42                        inca
    7:    200F 7A 1010                   staa result
    8:    2012 3F                        swi
    9:
   10:        =0000100F                  org data
   11:    100F EF                input:  dc.b $EF
   12:    1010 +0001             result: ds.b 1
```

```
Symbols:
data                    *0000100f
input                   *0000100f
prog                    *0000200b
result                  *00001010
```

Solution:

- ldaa input : Load contents of 0x100F into A (0xEF into A)
- inca: Increment A ($0xEF + 1 = 0xF0 \rightarrow A$)
- staa result : Store contents of A to address 0x1010 ($0xF0 \rightarrow$ adress 0x1010)
- swi : Software interrupt (Return control to DBug-12 Monitor)

Simple Programs for the MC9S12

A simple MC9S12 program fragment

```
org   $2000
ldaa  $1000
asra
staa  $1001
```

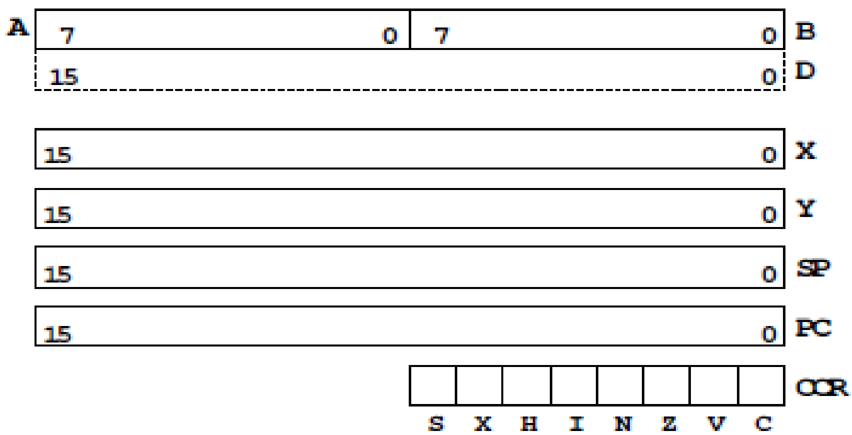
A simple MC9S12 program with assembler directives

```
prog:       equu  $2000
data:       equ   $1000

            org   prog
            ldaa  input
            asra
            staa  result
            swi

            org   data
input:      dc.b $07
result:     ds.b  1
```

MC9S12 Programming Model — The registers inside the MC9S12 CPU the programmer needs to know about



Things you need to know to write MC9S12 assembly language programs

HC12 Assembly Language Programming

Programming Model

MC9S12 Instructions

Addressing Modes

Assembler Directives

Addressing Modes for the MC9S12

• Almost all MC9S12 instructions operate on memory

• The address of the data an instruction operates on is called the effective address of that instruction.

• Each instruction has information which tells the MC9S12 the address of the data in memory it operates on.

• The addressing mode of the instruction tells the MC9S12 how to figure out the effective address for the instruction.

• Each MC9S12 instructions consists of a one or two byte op code which tells the HCS12 what to do and what addressing mode to use, followed, when necessary by one or more bytes which tell the HCS12 how to determine the effective address.

  – All two-byte op codes begin with an $18.

• For example, the LDAA instruction has 4 different op codes (86, 96, B6, A6), one for each of the 4 different addressing modes (IMM, DIR, EXT, IDX).

LDAA Load A LDAA

Operation

(M) ⇒ A
or
imm ⇒ A

Loads A with either the value in M or an immediate value.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise
Z: Set if result is $00; cleared otherwise
V: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LDAA #*opr8i*	IMM	86 ii	P
LDAA *opr8a*	DIR	96 dd	rPf
LDAA *opr16a*	EXT	B6 hh ll	rPO
LDAA *oprx0_xysppc*	IDX	A6 xb	rPf
LDAA *oprx9,xysppc*	IDX1	A6 xb ff	rPO
LDAA *oprx16,xysppc*	IDX2	A6 xb ee ff	frPP
LDAA [D,*xysppc*]	[D,IDX]	A6 xb	fIfrPf
LDAA [*oprx16,xysppc*]	[IDX2]	A6 xb ee ff	fIPrPf

The MC9S12 has 6 addressing modes

Most of the HC12's instructions access data in memory
There are several ways for the HC12 to determine which address to access

Effective address:
Memory address used by instruction

Addressing mode:
How the MC9S12 calculates the effective address

MC9S12 ADDRESSING MODES:

INH Inherent

IMM Immediate

DIR Direct

EXT Extended

REL Relative (used only with branch instructions)

IDX Indexed (won't study indirect indexed mode)

The Inherent (INH) addressing mode

Instructions which work only with registers inside ALU

ABA ; Add B to A (A) + (B) $\rightarrow$ A
18 06

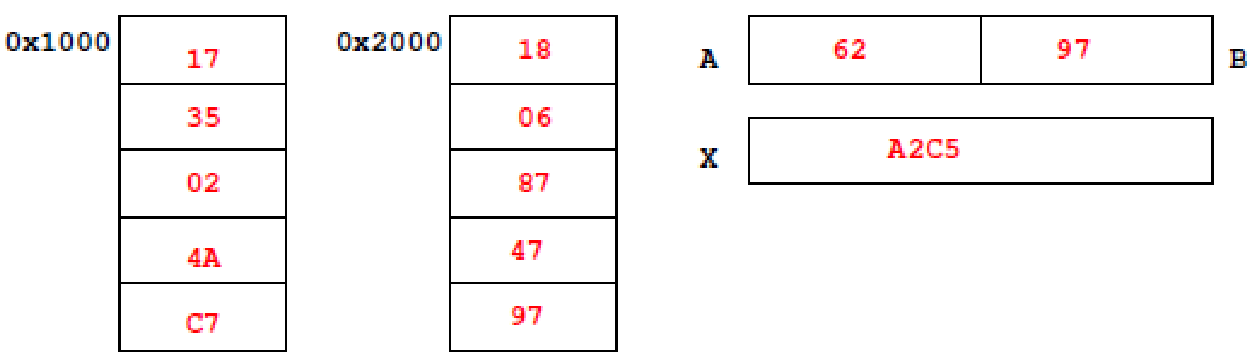
CLRA ; Clear A 0 $\rightarrow$ A
87

ASRA ; Arithmetic Shift Right A
47

TSTA ; Test A (A) − 0x00 Set CCR
97

The MC9S12 does not access memory

There is no effective address



The Extended (EXT) addressing mode

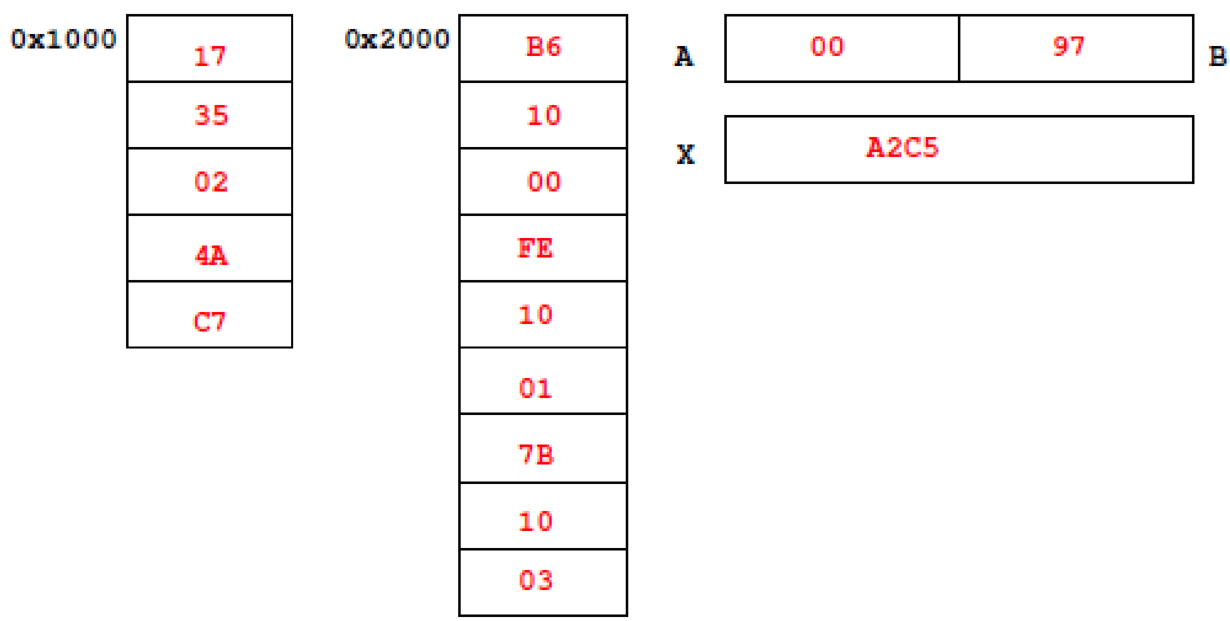
Instructions which give the 16−bit address to be accessed

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

LDX \$1001 ; (\$1001:\$1002) → X
FE 10 01 Effective Address: \$1001

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

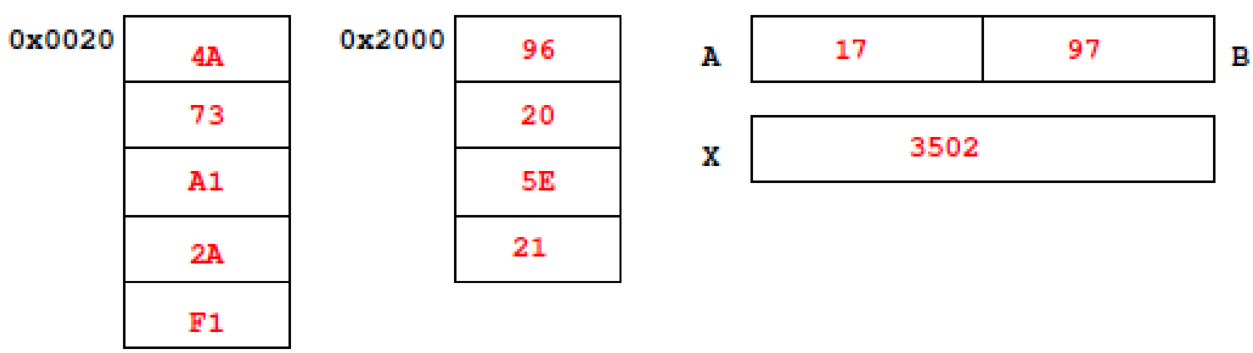
Direct (DIR) Addressing Mode

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



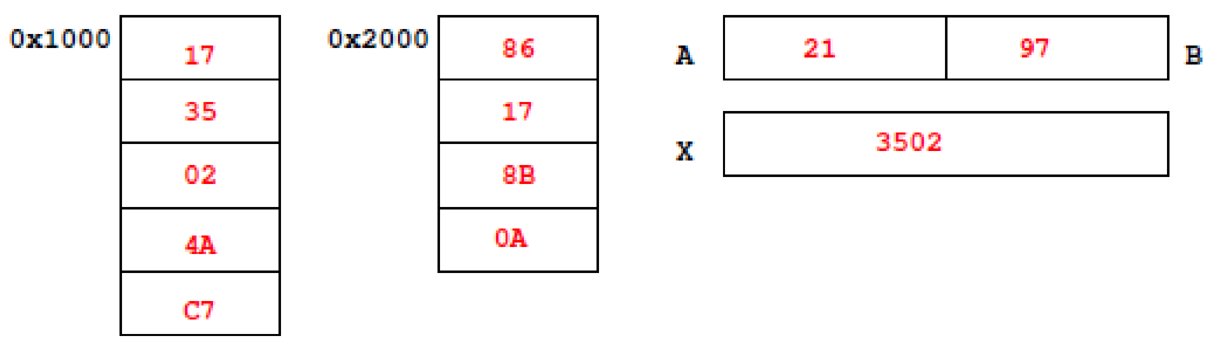
The Immediate (IMM) addressing mode

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)

Simple Forms

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

ADDA 5,Y ; Use (Y) + 5 as address to get value to add to
AB 45 Effective address: contents of Y + 5

More Complicated Forms

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

INC 4,+X ; Pre−increment Indexed
; Add 4 to X
; then increment the number at address (X)
62 23 Effective address: contents of X + 4

Different types of indexed addressing modes
(Note: We will not discuss indirect indexed mode)

INDEXED ADDRESSING MODES
(Does not include indirect modes)

	Example	Effective Address	Offset	Value in X After Done	Registers To Use
Constant Offset	LDAA n,X	(X)+n	0 to FFFF	(X)	X, Y, SP, PC
Constant Offset	LDAA −n,X	(X)−n	0 to FFFF	(X)	X, Y, SP, PC
Postincrement	LDAA n,X+	(X)	1 to 8	(X)+n	X, Y, SP
Preincrement	LDAA n,+X	(X)+n	1 to 8	(X)+n	X, Y, SP
Postdecrement	LDAA n,X−	(X)	1 to 8	(X)−n	X, Y, SP
Predecrement	LDAA n,−X	(X)−n	1 to 8	(X)−n	X, Y, SP

ACC Offset	LDAA A,X
	LDAA B,X
	LDAA D,X
		(X)+(A)
		(X)+(B)
		(X)+(D)
			0 to FF
			0 to FF
			0 to FFFF
				(X)	X, Y, SP, PC

Relative (REL) Addressing Mode

The relative addressing mode is used only in branch and long 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** PC + 2 + 0035 → PC

(BRA) **20 C7** PC + 2 + FFC7 → PC
PC + 2 − 0039 → PC

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

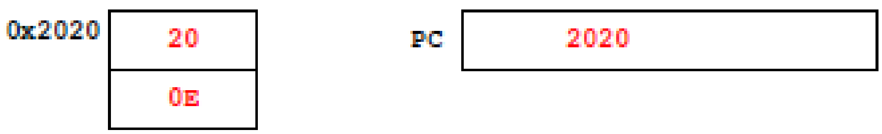
Treat the offset as an unsigned number; add the offset to the address following the current instruction to get the address of the instruction to branch to

(LBEQ) 18 27 02 1A If Z == 1 then PC + 4 + 021A → PC
If Z == 0 then PC + 4 → 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

```
0x2020      BRA       $200E          ; Branch to instruction at
                                     ; address $200E
```



Summary of HCS12 addressing modes

ADDRESSING MODES

Name		Example	Op Code	Effective Address
INH	Inherent	ABA	18 06	None
IMM	Immediate	LDAA #$35	86 35	PC + 1
DIR	Direct	LDAA $35	96 35	0x0035
EXT	Extended	LDAA $2035	B6 20 35	0x2035

IDX
IDX1
IDX2
	Indexed	LDAA 3,X
		LDAA 30,X
		LDAA 300,X
			A6 03
			A6 E0 13
			A6 E2 01 2C
				X + 3
				X + 30
				X + 300

IDX	Indexed Postincrement	LDAA 3,X+	A6 32	X (X+3 -> X)
IDX	Indexed Preincrement	LDAA 3,+X	A6 22	X+3 (X+3 -> X)
IDX	Indexed Postdecrement	LDAA 3,X-	A6 3D	X (X-3 -> X)
IDX	Indexed Predecrement	LDAA 3,-X	A6 2D	X-3 (X-3 -> X)
REL	Relative	BRA $1050 LBRA $1F00	20 23 18 20 0E CF	PC + 2 + Offset PC + 4 + Offset

A few instructions have two effective addresses:

- **MOVB $2000,$3000** ;move byte from address $2000 to ;$3000

- **MOVW 0,X,0,Y** ;move word from address pointed to ; by X to address pointed to by Y