Lecture 3

January 23, 2012

Assembly Language Programming

- Addition and Subtraction of Hexadecimal Numbers
- Simple Assembly Language Programming
  - A simple assembly language program
  - Assembling an assembly language program
  - Sample MC9S12 program
  - Hex code generated from sample MC9S12 program
  - Things you need to know for MC9212 assembly language programming
- Introduction to Addressing Modes
  - Most instrutions 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

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. The are called assembler directives or psuedo-ops. For example the psuedo-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 part of CodeWarrior
- 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          0000 2000   prog     equ      $2000  ; Start program at 0x2000
    2    2          0000 1000   data     equ      $1000  ; Data value at 0x1000
    3    3
    4    4                               org      prog
    5    5
    6    6  a002000 B610 00              ldaa     input
    7    7  a002003 42                   inca
    8    8  a002004 7A10 01              staa     result
    9    9  a002007 3F                   swi
   10   10
   11   11                               org      data
   12   12  a001000 A2          input:   dc.b     $A2
   13   13  a001001             result:  ds.b     1
```

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

```
S0630000433A5C446F63756D656E747320616E642053657474696E67735C7269736F6E5C4D7920446F63
S1051000A20048
S10B2000B61000427A10013F02
S9030000FC
```

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

```
S0630000433A5C446F63756D656E747320616E642053657474696E67735C7269736F6E5C4D7920446F63
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 from CodeWarrior is too long for the DBug-12 monitor. You will need to delete the first line 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 indcates 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` is 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 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 `test.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.

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

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

What will program do?

- `ldaa input` : Load contents of `0x1000` into **A**
  (`0xA2` into **A**)
- `inca` : Increment **A**
  (`0xA2 + 1 = 0xA3 -> ` **A**)
- `staa result` : Store contents of **A** to address `0x1001`
  (`0xA3 -> adress 0x1001`)
- `swi` : Software interrupt
  (Return control to DBug-12 Monitor)

Simple Programs for the MC9S12

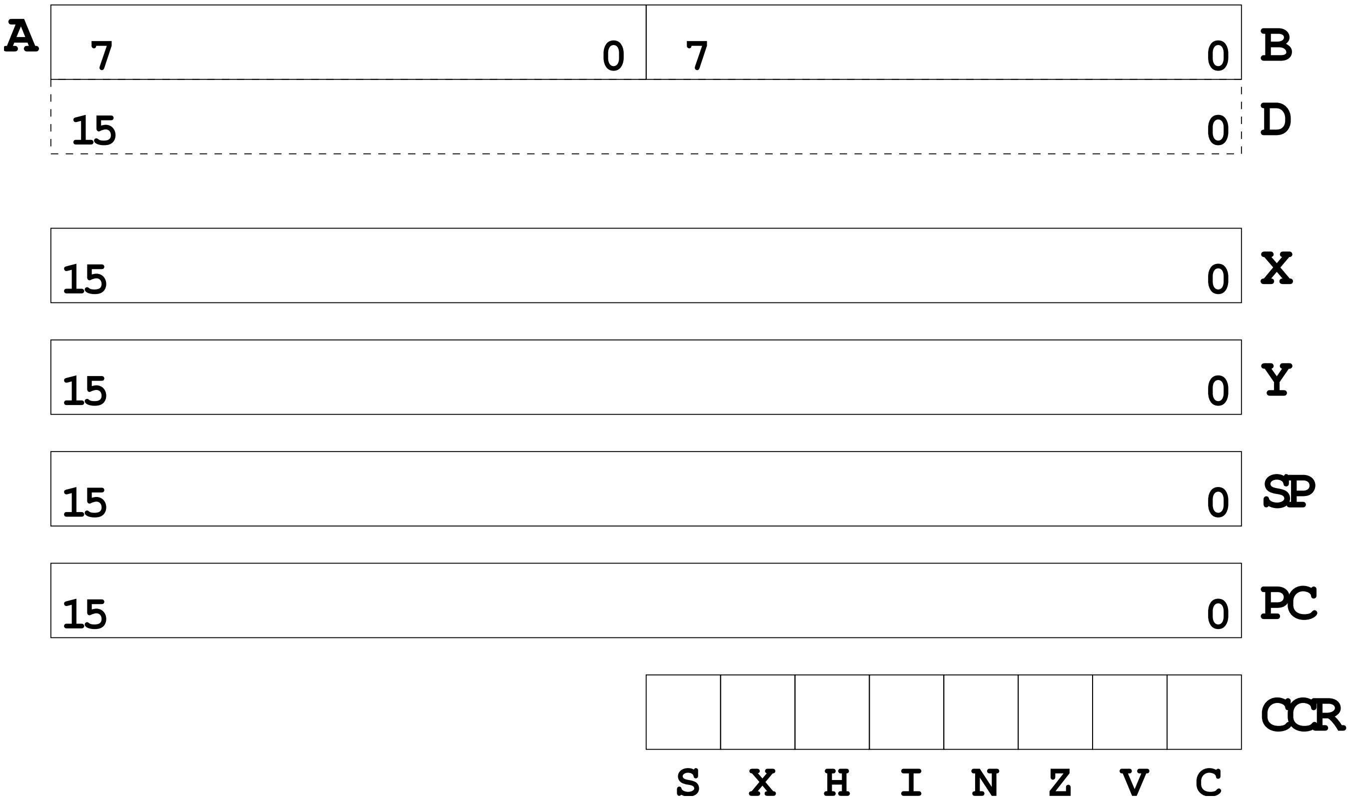
A simple MC9S12 program fragment

```
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 a



How the MC9S12 executes a simple program

EXECUTION OF SIMPLE HC12 PROGRAM

```
        LDAA $1013

        NEGA

        STAA $1014

0x2000   B6
0x2001   10
0x2002   13
0x2003   40
0x2004   7A
0x2005   10
0x2006   14

0x1013   6C
0x1014   5A
```

```
PC = 0x2000    Control unit reads B6
               Control decodes B6
PC = 0x2001    Control unit reads address MSB 10
PC = 0x2002    Control unit reads address LSB 13
               Control units tells memory to fetch
                      contents of address 0x1013
               Control units tells ALU to latch value
------------------------------------------------------------
PC = 0x2003    Control unit reads 40
               Control unit decodes 40
               Control unit tells ALU to negate ACCA
------------------------------------------------------------
PC = 0x2004    Control unit reads 7A
               Control decodes 7A
PC = 0x2005    Control unit reads address MSB 10
PC = 0x2006    Control unit reads address LSB 14
               Control units fetches value of ACCA from ALU
               Control units tells memory to store value
                      at address 0x1014
------------------------------------------------------------
PC = 0x2007
```

A

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

- Most 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 MC9S12 what to do and what addressing mode to use, followed, when necessary by one or more bytes which tell the MC9S12 how to deterime the effective address.
  - All two-byte op codes begin with an $18.
- For example, the `LDAA` instruction has 4 different op codes (`86, 96, A6, B6`), 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)**

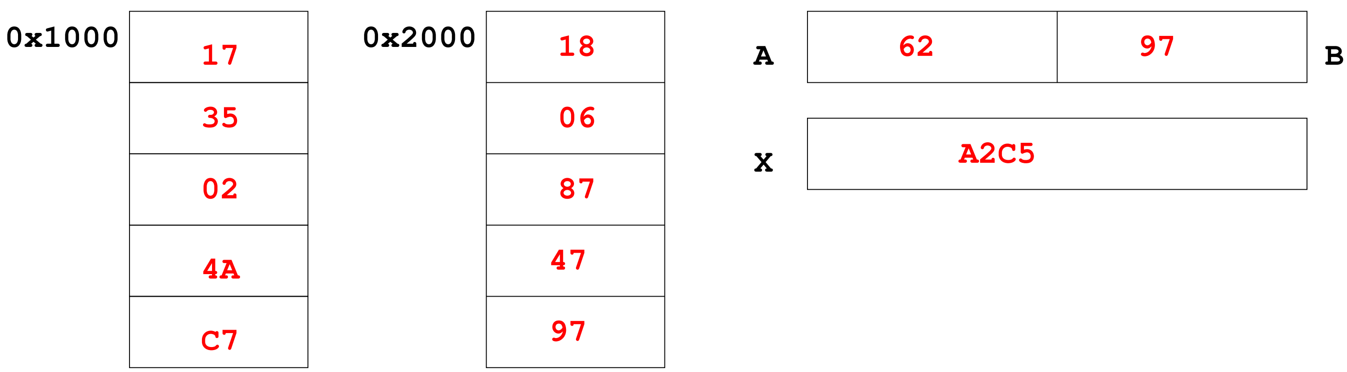
Inherent (INH) Addressing Mode

Instructions which work only with registers inside ALU

```
ABA          ; Add B to A    (A) + (B) -> A
  18 06

ASRA         ; Arithmetic Shift Right A
  87

CLRA         ; Clear A (0 -> A)
  47

TSTA         ; Test A     (A) - 0x00    Set CCR
  97
```

The MC9S12 does not access memory

There is no effective address



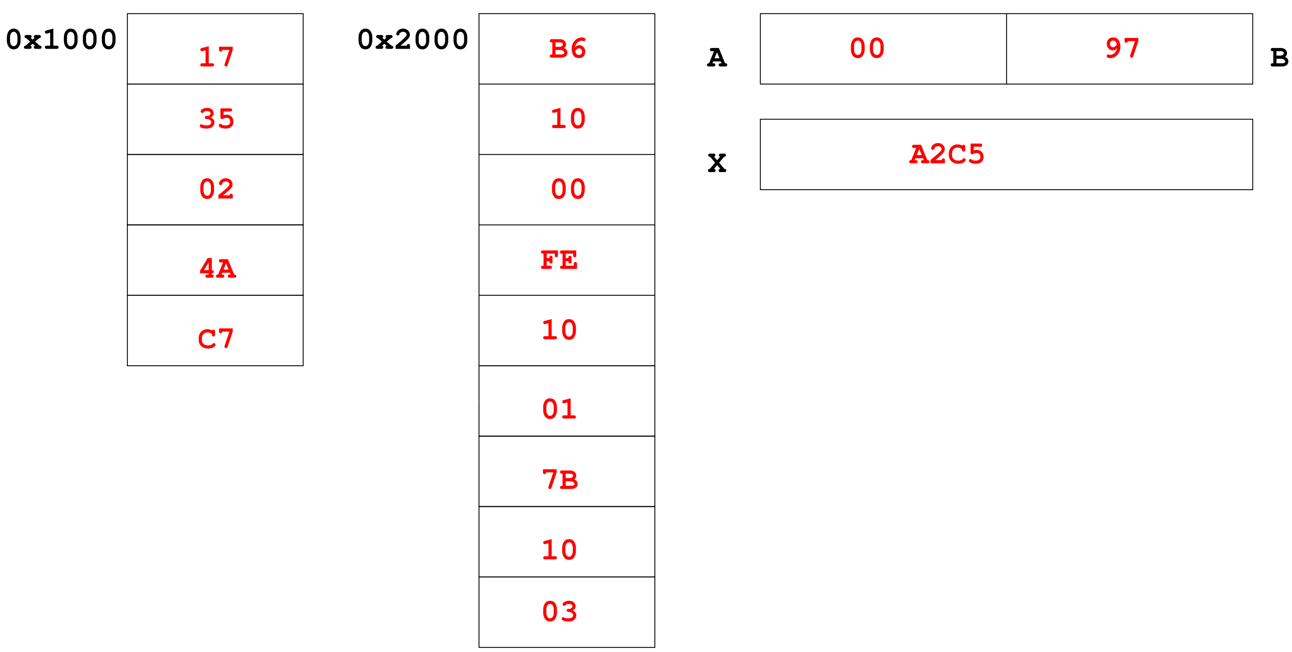
The *Extended* (EXT) addressing mode

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



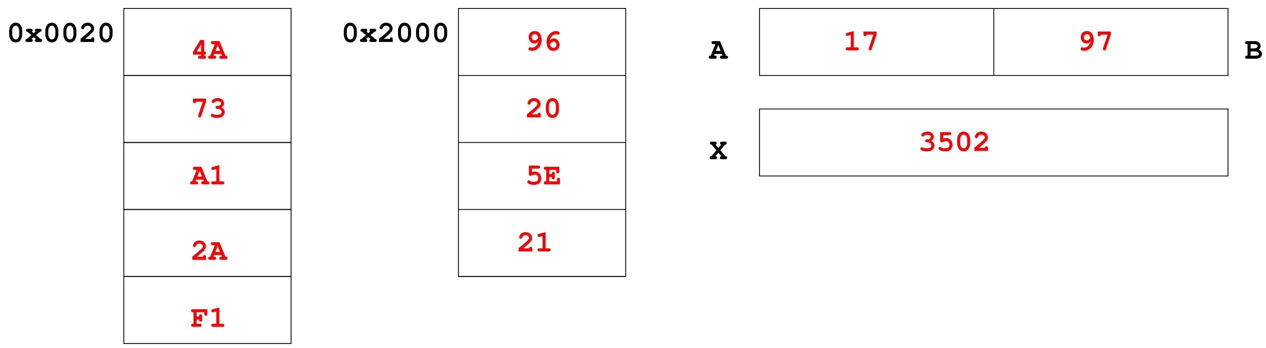
The *Direct* (DIR) addressing mode

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



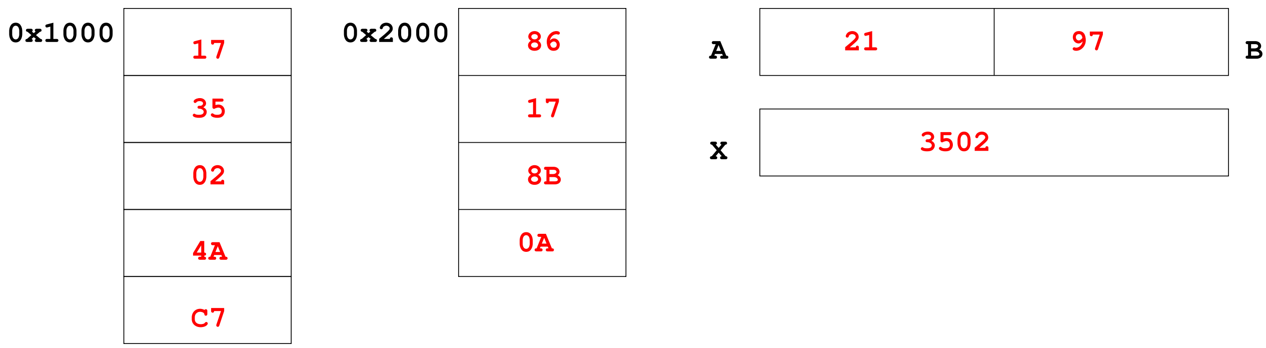
The *Immediate* (IMM) addressing mode

Immediate (IMM) Addressing Mode

Value to be used is part of instruction

```
LDAA   #$17         ; $17 -> A
  86 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

Indexed (IDX) 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
```

X		EFF ADDR	
Y		EFF ADDR	

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

The data books list three different types of indexed modes:

- Table 4.2 of the **Core Users Guide** shows details
- **IDX**: One byte used to specify address
  - Called the postbyte
  - Tells which register to use
  - Tells whether to use autoincrement or autodecrement
  - Tells offset to use
- **IDX1**: Two bytes used to specify address
  - First byte called the postbyte
  - Second byte called the extension
  - Postbyte tells which register to use, and sign of offset
  - Extension tells size of offset
- **IDX2**: Three bytes used to specify address
  - First byte called the postbyte
  - Next two bytes called the extension
  - Postbyte tells which register to use
  - Extension tells size of offset

Table 3-1. M68HC12 Addressing Mode Summary

Addressing Mode	Source Format	Abbreviation	Description
Inherent	**INST** (no externally supplied operands)	INH	Operands (if any) are in CPU registers
Immediate	**INST #***opr8i* or **INST #***opr16i*	IMM	Operand is included in instruction stream 8- or 16-bit size implied by context
Direct	**INST** *opr8a*	DIR	Operand is the lower 8 bits of an address in the range $0000–$00FF
Extended	**INST** *opr16a*	EXT	Operand is a 16-bit address
Relative	**INST** *rel8* or **INST** *rel16*	REL	An 8-bit or 16-bit relative offset from the current pc is supplied in the instruction
Indexed (5-bit offset)	**INST** *oprx5*,*xysp*	IDX	5-bit signed constant offset from X, Y, SP, or PC
Indexed (pre-decrement)	**INST** *oprx3*,**–***xys*	IDX	Auto pre-decrement x, y, or sp by 1 ~ 8
Indexed (pre-increment)	**INST** *oprx3*,**+***xys*	IDX	Auto pre-increment x, y, or sp by 1 ~ 8
Indexed (post-decrement)	**INST** *oprx3*,*xys***–**	IDX	Auto post-decrement x, y, or sp by 1 ~ 8
Indexed (post-increment)	**INST** *oprx3*,*xys***+**	IDX	Auto post-increment x, y, or sp by 1 ~ 8
Indexed (accumulator offset)	**INST** *abd*,*xysp*	IDX	Indexed with 8-bit (A or B) or 16-bit (D) accumulator offset from X, Y, SP, or PC
Indexed (9-bit offset)	**INST** *oprx9*,*xysp*	IDX1	9-bit signed constant offset from X, Y, SP, or PC (lower 8 bits of offset in one extension byte)
Indexed (16-bit offset)	**INST** *oprx16*,*xysp*	IDX2	16-bit constant offset from X, Y, SP, or PC (16-bit offset in two extension bytes)
Indexed-Indirect (16-bit offset)	**INST [***oprx16*,*xysp***]**	[IDX2]	Pointer to operand is found at... 16-bit constant offset from X, Y, SP, or PC (16-bit offset in two extension bytes)
Indexed-Indirect (D accumulator offset)	**INST [D,***xysp***]**	[D,IDX]	Pointer to operand is found at... X, Y, SP, or PC plus the value in D

The *Relative* (REL) addressing mode

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

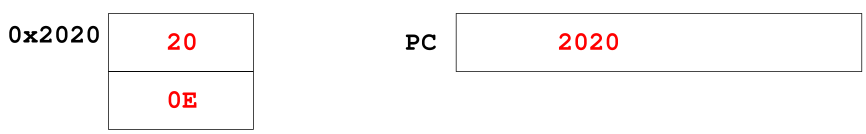
Treat the offset as an usigned number; add the offset to the address following the current instruction to get the address of the instruction to branch to

```
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

```
$2020           BRA    $2030    ; Branch to instruction at address $2030
```



Summary of MC9S12 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	0x0935

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`