

Exam 1  
Feb. 22, 24, 26?

- You will be able to use (and will need) all of the handouts from class
- No calculators will be allowed for the exam.
- 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

## Programming the MC9S12 in C

- A comparison of some assembly language and C constructs

Assembly		C
-----		-----
; Use a name instead of a num		/* Use a name instead of a num */
COUNT: EQU 5		#define COUNT 5
;		/*-----*/
;start a program		/* To start a program */
org    \$2000		main()
lds    #0x2000		{
		}
;		/*-----*/

- Note that in C, the starting location of the program is defined when you compile the program, not in the program itself.
- Note that C always uses the stack, so C automatically loads the stack pointer for you.

Assembly		C
-----		-----
;allocate two bytes for		/* Allocate two bytes for
;a signed number		* a signed number */
org    \$1000		
i:    ds.w    1		int i;
j:    dc.w    \$1A00		int j = 0x1a00;
;		/*-----*/
;allocate two bytes for		/* Allocate two bytes for
;an unsigned number		* an unsigned number */
i:    ds.w    1		unsigned int i;
j:    dc.w    \$1A00		unsigned int j = 0x1a00;
;		/*-----*/
;allocate one byte for		/* Allocate one byte for
;an signed number		* an signed number */
i:    ds.b    1		signed char i;
j:    dc.b    \$1F		signed char j = 0x1f;

<pre> Assembly ;----- ;Get a value from an address ; Put contents of address ; \$E000 into variable i  i: ds.b 1      ldaa \$E000     staa i  ;----- </pre>	<pre> C /*-----*/ /* Get a value from an address */ /* Put contents of address */ /* 0xE000 into variable i */  unsigned char i;  i = * (unsigned char *) 0xE000;  /*-----*/ /* Use a variable as a pointer    (address) */  unsigned char *ptr, i;  ptr = (unsigned char *) 0xE000; i = *ptr; *ptr = 0x55;  /*-----*/ </pre>
---	---

- In C, the construct `*(num)` says to treat `num` as an address, and to work with the contents of that address.
- Because C does not know how many bytes from that address you want to work with, you need to tell C how many bytes you want to work with. You also have to tell C whether you want to treat the data as signed or unsigned.
  - `i = * (unsigned char *) 0xE000;` tells C to take one byte from address `0xE000`, treat it as unsigned, and store that value in variable `i`.
  - `j = * (int *) 0xE000;` tells C to take two bytes from address `0xE000`, treat it as signed, and store that value in variable `j`.
  - `* (char *) 0xE000 = 0xaa;` tells C to write the number `0xaa` to a single byte at address `0xE000`.
  - `* (int *) 0xE000 = 0xaa;` tells C to write the number `0x00aa` to two bytes starting at address `0xE000`.

```

Assembly      | C
;-----| /*-----*/
;To call a subroutine | /* To call a function */
  ldaa i      | sqrt(i);
  jsr  sqrt   |
;-----| /*-----*/
;To return from a subroutine | /* To return from a function */
  ldaa j      | return j;
  rts         |

;-----| /*-----*/
;Flow control      | /* Flow control */
  blo         | if (i < j)
  blt         | if (i < j)
              |
  bhs         | if (i >= j)
  bge         | if (i >= j)
;-----| /*-----*/

```

- Here is a simple program written in C and assembly. It simply divides 16 by 2. It does the division in a function.

```

ASSEMBLY      | C
-----|-----
          |
          | signed char i;
i:  org     $1000 |
     ds.b   1     |
          |
          | signed char div(signed char j);
          | main()
          | {
          |   i = div(16);
          | }
          |
          |
div:  asra   | signed char div(signed char j)
     rts    | {
          |   return j >> 1;
          | }

```

## A simple C program and how to compile it

Here is a simple C program

```
#define COUNT 5

unsigned int i;

main()
{
    i = COUNT;
}
```

1. Start CodeWarrior and create a new project.
2. On the **Project Parameters** menu, leave the **C** box checked, give the project a name, and **Set** an appropriate directory.
3. On the **C/C++ Options** menu, select **ANSI startup code**, **Small memory model**, and **None** for floating point format. Then select **Finish**. This will open a new project for a C program.
4. Select **Edit – Standard Settings**. Select **Target – Compiler for HC12**, then click on **Options**. Click on the **Output** tab, and select the **Generate Listing File** option. Click **OK**, the **OK**.
5. C does not use an `org` statement to tell the compiler where to put code or data. CodeWarrior uses a linker file called `Project.prm`. You will have to edit this file to tell the compiler where to put your program and data. CodeWarrior has been set up to put your program into Flash EEPROM starting at address `0xC000`. In this class, you will put your program into RAM starting at address `0x2000`, or into EEPROM starting at address `0x0400`. In the window which lists the project files, select **Project Settings – Linker Files – Project.prm**. Find the following line:

```
RAM          = READ_WRITE    0x1000 TO    0x3FFF;
```

and change it to this:

```
RAM          = READ_WRITE    0x1000 TO    0x2000;
PROG         = READ_ONLY     0x2000 TO    0x3FFF;
```

Next, find the line

```
INTO ROM_C000/*, ROM_4000*/;
```

and change it to

```
INTO PROG/*, ROM_4000*/;
```

Save and close `Project.prm`.

6. In the window which lists the project files, double-click on `main.c`. Modify the file to look like this:

```
#include <hidef.h>      /* common defines and macros */
#include "derivative.h" /* derivative-specific definitions */

void main(void) {

}
```

7. Enter your C program.
8. Select **Project – Make**. This will create a `Project.abs.s19` file and a listing file `main.lst` in the `bin` directory. You will need to delete the first line (which starts with `S0`) from the `Project.abs.s19` file.
9. If all went well, you should be able to download the `Project.abs.sa9` file into the MC9S12.

In the `bin` directory there will be several files with the `.lst` extension. The file `Start12.lst` contains C startup code. The file `main.lst` shows the assembly language which was produced by the C compiler.

The `Start12.lst` is fairly long, because it contains uncompiled code for a lot of things we do not use. Here are the portions of `Start12.lst` which we use. It just loads the stack pointer, initializes any needed global data, zeros out the rest of the global data, and calls the `main.c` code.

```

131: static void Init(void)
134: /* purpose:      1) zero out RAM-areas where data is allocated  */
135: /*                2) copy initialization data from ROM to RAM    */
139: ZeroOut:
0000 fe0000      [3]    LDX   _startupData:2
0003 fd0000      [3]    LDY   _startupData
0006 270e        [3/1]  BEQ   CopyDown ;abs = 0016
148: NextZeroOut:
0008 35          [2]    PSHY
000b ec31        [3]    LDD   2,X+
185: NextWord:
000d 6970        [2]    CLR   1,Y+
000f 0434fb      [3]    DBNE  D,NextWord ;abs = 000d
0012 31          [3]    PULY
0013 03          [1]    DEY
0014 26f2        [3/1]  BNE   NextZeroOut ;abs = 0008
206: CopyDown:
0016 fe0000      [3]    LDX   _startupData:4
216: NextBlock:
0019 ec31        [3]    LDD   2,X+
001b 270b        [3/1]  BEQ   funcInits ;abs = 0028
257: Copy:
001f 180a3070    [5]    MOVB  1,X+,1,Y+
0023 0434f9      [3]    DBNE  D,Copy ;abs = 001f
0026 20f1        [3]    BRA   NextBlock ;abs = 0019
271: funcInits:                                     ; call of global constructors is only i
0028 3d          [5]    RTS

```

Function: `_Startup`

```

399: /* purpose:      1) initialize the stack
400:                  2) initialize the RAM, copy down init data etc (Init)
401:                  3) call main;
405:
406: /* initialize the stack pointer */
0000 cf0000      [2]    LDS   #__SEG_END_SSTACK
460: Init(); /* zero out, copy down, call constructors */
0003 0700        [4]    BSR   Init
469: main();
0005 060000      [3]    JMP   main
470: }

```

Here is the main.lst file.

ANSI-C/cC++ Compiler for HC12 V-5.0.38 Build 9056, Feb 26 2009

```
1: #include <hidef.h>      /* common defines and macros */
2: #include "derivative.h" /* derivative-specific definitions */
3: #define COUNT 5
4:
5: unsigned int i;
6:
7: void main(void) {
```

Function: main

```
8:      i = COUNT;
0000 c605      [1]    LDAB #5
0002 87       [1]    CLRA
0003 7c0000   [3]    STD  i
9:  }
0006 3d       [5]    RTS
10:
```

The file `Project.map` shows where various things will be put in memory. It is fairly long. Here are the relevant parts:

```
*****
STARTUP SECTION
-----
Entry point: 0x2029 (_Startup)

*****
SECTION-ALLOCATION SECTION
Section Name          Size  Type   From      To        Segment
-----
.init                 49    R      0x2000    0x2030    PROG
.startData            10    R      0x2031    0x203A    PROG
.text                 7     R      0x203B    0x2041    PROG
.copy                 2     R      0x2042    0x2043    PROG
.stack                256   R/W    0x1000    0x10FF    RAM
MODULE:               -- main.c.o --
- PROCEDURES:
  main                203B   7      7         1   .text
- VARIABLES:
  i                   1100   2      2         1   .common
MODULE:               -- Start12.c.o --
- PROCEDURES:
  Init                2000   29     41        1   .init
  _Startup            2029   8      8         0   .init
- VARIABLES:
  _startupData        2031   6      6         3   .startData
- LABELS:
  __SEG_END_SSTACK    1100   0      0         1
```

This shows that the total program occupies addresses from 0x2000 to 0x2043. The stack occupies addresses from 0x1000 to 0x10FF. Our variable `i` is located at address 0x1100. The entry point to the program is at 0x2029. This means that, to run the program, you need to tell DBug-12 to run the program from 0x2029, not from 0x2000:

```
g 2029
```

## Pointers in C

- To access a memory location using a pointer in C, you might think the following would work:

```
*address
```

- You need to tell compiler whether you want to access 8-bit or 16 bit number, signed or unsigned:

```
*(type *)address
```

- To read from an eight-bit unsigned number at memory location 0x1000:

```
x = *(unsigned char *)0x1000;
```

- To write an 0xaa55 to a sixteen-bit signed number at memory locations 0x1010 and 0x1011:

```
*(signed int *)0x1010 = 0xaa55;
```

- If there is an address which is used a lot:

```
#define PORTB (* (unsigned char *) 0x0001)
```

```
x = PORTB;      /* Read from address 0x0001 */
```

```
PORTB = 0x55;  /* Write to address 0x0001 */
```

- To access consecutive locations in memory, use a variable as a pointer:

```
unsigned char *ptr;
```

```
ptr = (unsigned char *)0x1000;
```

```
*ptr = 0xaa;      /* Put 0xaa into address 0x1000 */
```

```
ptr = ptr+2;     /* Point two further into table */
```

```
x = *ptr;        /* Read from address 0x1002 */
```

- To set aside ten locations for a table:

```
unsigned char table[10];
```

- Can access the third element in the table as:

```
table[2]
```

or as

```
*(table+2)
```

- To set up a table of constant data:

```
const unsigned char table[] = {0x00,0x01,0x03,0x07,0x0f};
```

This will tell the compiler to place the table of constant data with the program (which might be placed in EEPROM) instead of with regular data (which must be placed in RAM).

- There are a lot of registers (such as `PORTA` and `DDRA`) which you will use when programming in C. CodeWarrior includes the header file `mc9s12dp256.h` which has the registers predefined.

## Setting and Clearing Bits in C

- You often need to set or clear bits of a hardware register.
  - The easiest way to set bits in C is to use the bitwise OR (`|`) operator:

```
DDRB = DDRB | 0x0F; /* Make 4 LSB of Port B outputs */
```
  - The easiest way to clear bits in C is to use the bitwise AND (`&`) operator:

```
DDRP = DDRP & ~0xF0; /* Make 4 MSB of Port J inputs */
```

Program to make LEDs on Dragon12-Plus board count

```

#include "derivative.inc"
    lds    #$2000      ; Load stack pointer
    bset   DDRP,$$0F   ; Make PPO-PP3 outputs
    bset   PTP,$$0F   ; Turn off seven-seg LEDs
    bset   DDRJ,$$02   ; Make PJ1 output
    bclr   PRJ,$$02   ; Turn on individual LEDs
    movb   $$FF,DDRB  ; Activate control lines for LEDs
loop:   inc    PORTB   ;
        bsr    delay   ; Wait a bit
        bra    loop    ; Repeat
delay:  ldy    #100
l1:     ldx    #8000
l2:     dbne   x,l2
        dbne   y,l1
        rts

-----
#include <hidef.h>      /* common defines and macros */
#include "derivative.h" /* derivative-specific definitions */
#define D_1MS (24000/3) // Inner loop is 3 cycles
#define TRUE 1
void delay(unsigned int ms);

int main() {
    DDRP = DDRP | 0x0F; /* Make PPO-PP3 outputs */
    PTP = PTP | 0x0F; /* Turn off seven-seg LEDs */
    DDRJ = DDRJ | 0x02; /* Make PJ1 output */
    PTJ = PTJ & ~0x02; /* Turn on individual LEDs */
    DDRB = 0xFF; /* Activate control lines for LEDs */
    while (TRUE) { /* Repeat forever */
        PORTB = PORTB + 1; /* Increment LEDs; */
        delay(100); /* Wait a bit */
    }
}

void delay (unsigned int ms) {
    unsigned int i;
    while (ms > 0) {
        i = D_1MS;
        while (i > 0) {
            i = i - 1;
        }
        ms = ms - 1;
    }
}

```

Here is the `main.lst` file. Note that the inner loop of the `delay()` function ( $i = i - 1$ ) takes 3 cycles to execute.

```

    9:      DDRP = DDRP | 0x0F;      /* Make PP0-PP3 outputs      */
0000 1c00000f      [4]      BSET  _DDRP,#15
    10:     PTP = PTP | 0x0F;      /* Turn off seven-seg LEDs  */
0004 1c00000f      [4]      BSET  _PTP,#15
    11:     DDRJ = DDRJ | 0x02;     /* Make PJ1 output          */
0008 1c000002      [4]      BSET  _DDRJ,#2
    12:     PTJ = PTJ & ~0x02;     /* Turn on individual LEDs  */
000c 1d000002      [4]      BCLR  _PTJ,#2
    13:     DDRB = 0xFF;           /* Activate control lines for LEDs */
0010 c6ff         [1]      LDAB  #255
0012 5b00         [2]      STAB  _DDRAB:1
    14:     while (TRUE) {         /* Repeat forever          */
    15:         PORTB = PORTB + 1; /* Increment LEDs;        */
0014 720000      [4]      INC   _PORTAB:1
    16:         delay(100);        /* Wait a bit              */
0017 c664         [1]      LDAB  #100
0019 87           [1]      CLRA
001a 160000      [4]      JSR   delay
001d 20f5        [3]      BRA   *-9 ;abs = 0014
    17:     }
    18:     }
    19:     void delay (unsigned int ms)
    20:     {
0000 3b           [2]      PSHD
    21:         unsigned int i;
    22:         while (ms > 0) {
0001 200b        [3]      BRA   *+13 ;abs = 000e
    23:             i = D_1MS;
0003 ce1f40      [2]      LDX   #8000
    24:             while (i > 0) {
0006 0435fd      [3]      DBNE  X,*+0 ;abs = 0006
    25:                 i = i - 1;
    26:             }
    27:             ms = ms - 1;
0009 ee80        [3]      LDX   0,SP
000b 09          [1]      DEX
000c 6e80        [2]      STX   0,SP
000e ec80        [3]      LDD   0,SP
0010 26f1        [3/1]   BNE   *-13 ;abs = 0003
    28:         }
    29:     }
0012 3a          [3]      PULD
0013 3d          [5]      RTS

```