Linking Assembly Subroutine with a C Program

- To link an assembly subroutine to a C program, you have to understand how parameters are passed.
- For the CodeWarrior C compiler, one parameter is passed in the registers. The other parameters are passed on the stack.
  - The left-most parameter is pushed onto the stack first, then the next-to-left, etc.
  - The right-most parameter is passed in the registers
    - An 8-bit parameter is passed in the B regisiter
    - A 16-bit parameter is passed in the D register
    - A 32-bit parameter is passed in the {X:D} register combination.
- A value returned from the assembly language program is returned in the registers:
  - An 8-bit parameter is returned in the B regisiter
  - A 16-bit parameter is returned in the D register
  - A 32-bit parameter is returned in the {X:D} register combination.
- In the assembly language program, declare things the C program has to know about as `XDEF`:

```
XDEF     foo
```

- In the C program, declare things in the assembly program as you would an other functions:

```
int foo(int x);
```

- In the assembly language program, use the stack to store local variables
  - Need to keep close track of stack frame

Consider an assembly-language function `fuzzy` which uses two 8-bit arguments `arg1` and `arg2`, and returns an 8-bit argument `result`.

- Declare the function in the C program as

```
char fuzzy(char arg1, char arg2);
```

- Here is how the function may be called in the C program:

```
char x,y,result;

result = fuzzy(x, y);
```

- When the program is compiled, the value of the variable `x` is pushed onto the stack, the value of the variable `y` is loaded into the B register, and the function is called with a `JSR` instruction:

```
      15:     result = fuzzy(x,y); /* call the assembly function */
  000b f60000       [3]     LDAB  x
  000e 37           [2]     PSHB
  000f f60000       [3]     LDAB  y
  0012 160000       [4]     JSR   fuzzy
```

- In the assembly language function, you may need to use some local variables, which need to be allocated on the stack. If the `fuzzy` function needs two local 8-bit variables `var1` and `var2`, you will need to allocate two bytes on the stack for them. Here's what the start of the assembly language program will look like:

```
fuzzy:
      leas      -2,sp        ; Room on stack for var1 and var2

; Stack frame after leas -2,sp
;
;     SP      -> var1
;     SP + 1  -> var2
;     SP + 2  -> Return address high
;     SP + 3  -> Return address low
;     SP + 4  -> 1st parameter of function (arg1)
;
;     2nd paramter (arg2) passed in B register
```

- In the assembly language program, you access `arg1`, `var1` and `var2` with indexed addressing mode:

```
stab   1,SP     ; Save arg2 into var2
ldaa   4,SP     ; Put arg1 into ACCA
staa   0,SP     ; Save arg1 into var1
```

- When you return from the assembly language function, put the value you want to return into B, add two to the stack (to deallocate `var1` and `var2`), and return with an RTS. For example, it you want to return the value of the variable `var2`, you would do the following:

```
ldab   1,SP       ; Put var2 into B
leas   2,SP       ; Deallocate local variables
rts               ; Return to calling program
```

- Any global variables used by the program should be declared in a separate section:

```
; section for global variables
FUZZY_RAM: SECTION
; Locations for the fuzzy input membership values
I_E_PM:          ds.b     1
I_E_PS:          ds.b     1
```

- Any global constants used by the program should be declared in a separate section:

```
; section for global variables
FUZZY_RAM: SECTION
; Locations for the fuzzy input membership values
I_E_PM:          ds.b     1
I_E_PS:          ds.b     1

FUZZY_CONST: SECTION
; Fuzzy input membership function definitions for speed error
E_Pos_Medium:    dc.b    170, 255,   6,   0
E_Pos_Small:     dc.b    128, 208,   6,   6
```

- The assembly language code should be put in its own section:

```
; code section
MyCode:     SECTION
; this assembly routine is called by the C/C++ application
fuzzy:
            leas      -2,sp        ; Room on stack for ERROR and d_ERROR
```

C program which calls fuzzy logic assembly function

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

int fuzzy(unsigned char e, unsigned char de);
unsigned char ERROR[]  =  {  2, 54,106,158,206,255};
unsigned char d_ERROR[] = {101,102,103,104,105,106};

void main (void)
{
    char dPWM;
    int i;

/* Set up SCI for using printf() */
    SCI0BDH = 0x00;           /* 9600 Baud */
    SCI0BDL = 0x9C;
    SCI0CR1 = 0x00;
    SCI0CR2 = 0x0C;           /* Enable transmit, receive */

    for (i=0;i<6;i++) {
        dPWM = fuzzy(ERROR[i],d_ERROR[i]);
        (void) printf("ERROR = %3d, d_ERROR = %3d, ", ERROR[i],d_ERROR[i]);
        (void) printf("dPWM: %4d\r\n", dPWM);
    }
    asm(" swi");
}
```

The Assembly program

```
;**************************************************************
;* This stationery serves as the framework for a              *
;* user application. For a more comprehensive program that    *
;* demonstrates the more advanced functionality of this       *
;* processor, please see the demonstration applications       *
;* located in the examples subdirectory of the                *
;* Freescale CodeWarrior for the HC12 Program directory       *
;**************************************************************

; export symbols
            XDEF fuzzy
            ; we use export 'Entry' as symbol. This allows us to
            ; reference 'Entry' either in the linker .prm file
            ; or from C/C++ later on

; Include derivative-specific definitions
        INCLUDE 'derivative.inc'

; Offset values for input and output membership functions
E_PM            equu         0  ; Positive medium error
E_PS            equ          1  ; Positive small error
E_ZE            equ          2  ; Zero error
E_NS            equ          3  ; Negative small error
E_NM            equ          4  ; Negative medium error
dE_PM           equ          5  ; Positive medium differential error
dE_PS           equ          6  ; Positive small differential error
dE_ZE           equ          7  ; Zero differential error
dE_NS           equ          8  ; Negative small differential error
dE_NM           equ          9  ; Negative medium differential error
O_PM            equ         10  ; Positive medium output
O_PS            equ         11  ; Positive small
O_ZE            equ         12  ; Zero output
O_NS            equ         13  ; Negative small
O_NM            equ         14  ; Negative medium output
MARKER          equ        $FE  ; Rule separator
END_MARKER      equ        $FF  ; End of Rule marker

; variable/data section
FUZZY_RAM: SECTION
; Locations for the fuzzy input membership values for speed error
I_E_PM:         ds.b     1
I_E_PS:         ds.b     1
I_E_ZE:         ds.b     1
I_E_NS:         ds.b     1
I_E_NM:         ds.b     1
```

```
; Locations for the fuzzy input membership values for speed error diff
I_dE_PM:          ds.b     1
I_dE_PS:          ds.b     1
I_dE_ZE:          ds.b     1
I_dE_NS:          ds.b     1
I_dE_NM:          ds.b     1

; Output fuzzy membership values - initialize to zero
M_PM:             ds.b     1
M_PS:             ds.b     1
M_ZE:             ds.b     1
M_NS:             ds.b     1
M_NM:             ds.b     1


FUZZY_CONST: SECTION
; Fuzzy input membership function definitions for speed error
E_Pos_Medium:     dc.b    170, 255,   6,   0
E_Pos_Small:      dc.b    128, 208,   6,   6
E_Zero:           dc.b     88, 168,   6,   6
E_Neg_Small:      dc.b     48, 128,   6,   6
E_Neg_Medium:     dc.b      0,  80,   0,   6
; Fuzzy input membership function definitions for speed error
dE_Pos_Medium:    dc.b    170, 255,   6,   0
dE_Pos_Small:     dc.b    128, 208,   6,   6
dE_Zero:          dc.b     88, 168,   6,   6
dE_Neg_Small:     dc.b     48, 128,   6,   6
dE_Neg_Medium:    dc.b      0,  80,   0,   6

; Fuzzy output memership function definition
PM_Output:        dc.b    192
PS_Output:        dc.b    160
ZE_Output:        dc.b    128
NS_Output:        dc.b     96
NM_Output:        dc.b     64
; Rule Definitions
Rule_Start:       dc.b      E_PM,dE_PM,MARKER,O_NM,MARKER
                  dc.b      E_PM,dE_PS,MARKER,O_NM,MARKER
                  dc.b      E_PM,dE_ZE,MARKER,O_NM,MARKER
                  dc.b      E_PM,dE_NS,MARKER,O_NS,MARKER
                  dc.b      E_PM,dE_NM,MARKER,O_ZE,MARKER

                  dc.b      E_PS,dE_PM,MARKER,O_NM,MARKER
                  dc.b      E_PS,dE_PS,MARKER,O_NM,MARKER
                  dc.b      E_PS,dE_ZE,MARKER,O_NS,MARKER
                  dc.b      E_PS,dE_NS,MARKER,O_ZE,MARKER
                  dc.b      E_PS,dE_NM,MARKER,O_PS,MARKER
```

```
                dc.b      E_ZE,dE_PM,MARKER,O_NM,MARKER
                dc.b      E_ZE,dE_PS,MARKER,O_NS,MARKER
                dc.b      E_ZE,dE_ZE,MARKER,O_ZE,MARKER
                dc.b      E_ZE,dE_NS,MARKER,O_PS,MARKER
                dc.b      E_ZE,dE_NM,MARKER,O_PM,MARKER

                dc.b      E_NS,dE_PM,MARKER,O_NS,MARKER
                dc.b      E_NS,dE_PS,MARKER,O_ZE,MARKER
                dc.b      E_NS,dE_ZE,MARKER,O_PS,MARKER
                dc.b      E_NS,dE_NS,MARKER,O_PM,MARKER
                dc.b      E_NS,dE_NM,MARKER,O_PM,MARKER

                dc.b      E_NM,dE_PM,MARKER,O_ZE,MARKER
                dc.b      E_NM,dE_PS,MARKER,O_PS,MARKER
                dc.b      E_NM,dE_ZE,MARKER,O_PM,MARKER
                dc.b      E_NM,dE_NS,MARKER,O_PM,MARKER
                dc.b      E_NM,dE_NM,MARKER,O_PM,END_MARKER

; code section
MyCode:     SECTION
; this assembly routine is called by the C/C++ application

; Stack frame after leas -2,sp
;
;               SP      -> ERROR
;               SP + 1  -> d_ERROR
;               SP + 2  -> Return address high
;               SP + 3  -> Return address low
;               SP + 4  -> 1st parameter of function (d_ERROR)
;
;               2nd paramter (ERROR) passed in B register
fuzzy:
                leas      -2,sp         ; Room on stack for ERROR and d_ERROR
                stab      1,sp          ; d_ERROR passed in B register
                ldab      4,sp          ; ERROR passed on stack
                stab      0,sp          ; Save in space reserved on stack

; Fuzzification
                LDX       #E_Pos_Medium ; Start of Input Mem func
                LDY       #I_E_PM        ; Start of Fuzzy Mem values
                LDAA      0,SP           ; Get ERROR value
                LDAB      #5             ; Number of iterations
Loop_E:         MEM                      ; Assign mem value
                DBNE      B,Loop_E       ; Do all five iterations
                LDAA      1,SP           ; Get d_ERROR value
                LDAB      #5             ; Number of iterations
Loop_dE:        MEM                      ; Assign mem value
```

```
                DBNE      B,Loop_dE     ; Do all five iterations

; Process rules
                LDX       #M_PM         ; Clear output membership values
                LDAB      #5
Loopc:          CLR       1,X+
                DBNE      B,Loopc

                LDX       #Rule_Start   ; Address of rule list -> X
                LDY       #I_E_PM       ; Address of input membership list -> Y
                LDAA      #$FF          ; FF -> A, clear V bit of CCR
                REV                     ; Rule evaluation

; Defuzzification
                LDX       #PM_Output    ; Address of output functions -> X
                LDY       #M_PM         ; Address of output membership values -> Y
                LDAB      #5            ; Number of iterations
                WAV                     ; Defuzzify
                EDIV                    ; Divide
                TFR       Y,D           ; Quotient to D; B now from 0 to 255
                SUBB      #128          ; Subtract offset from d_PWM
                                        ; dPWM returned in B; already there
                leas      2,sp          ; Return stack frame to entry value

                RTS
```

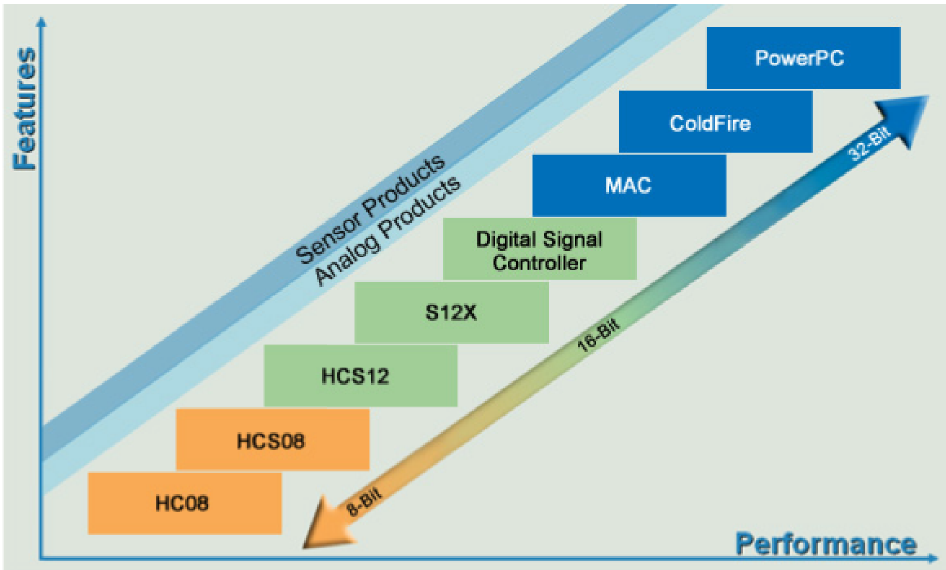
Microcontroller Architectures
Things to Consider

- Performance vs. Cost
  - Speed (instructions/second)
  - Precision (8, 16, 32 or 64 bits, fixed or floating point)
  - Princeton or Harvard Architecture
  - RISC or CISC?
    - RISC: Reduced Instruction Set Computer
      - Very few instructions (8-bit PIC uses 33 instructions)
      - Each instruction takes one cycle to execute
      - Each instruction takes one word of memory
      - Reduces hardware size, increases software size
      - Easier to implement pipelines, etc.
    - CISC: Complex Instruction Set Computer
      - Larger number of more specialized instructions
      - Increases hardware size, reduces software size
- Voltage
- Peripherals
  - A/D converter (number of bits)
  - COM ports (how many, what type – SCI, SPI I^2C)
  - USB
  - Ethernet
  - Timers
  - Specialized items
    - PWM
    - Media control (Compact Flash, Secure Digital cards)
    - Many others
- Memory
  - Address bus size
  - RAM
  - EEPROM
  - Flash EEPROM
- Special Requirements
  - Low power for battery applications
  - Radiation hardened for space applications
  - Temperature range

- Development Tools
  - Software Tools
    - Assembler
    - C Compiler
    - IDE
  - Hardware tools
    - Evaluation boards
    - In Circuit Emulators
    - Background Debug Mode
- Familiarity
  - Different lines from same manufacturer often have similar programming models and instruction forms
  - For example, consider writing the byte $AA to address held in the X register:
    - `Motorola:  movb  #$AA, 0,X`
    - `Intel:     mov   [ECX] 0AAH`
  - Consider the way the 16-bit $1234 number is stored in memory location $2000

```
1. Motorola:  $12 is stored in address $2000,
              $34 is stored in address $2001
2. Intel:     $34 is stored in address $2000,
              $12 is stored in address $2001
```

Freescale (Motorola) Microcontrollers



- HC08 (8 bit)
  - \$1.00 each
  - 8 pins to 80 pins
  - 128 bytes to 2 KB RAM
  - 1.5 KB to 7680 KB Flash EEPROM
  - 2 MHz to 8 MHz clock
  - Lots of different peripherals
- HCS08 (8 bit)
  - \$2.00 each (and higher)
  - 8 pins to 64 pins
  - 512 bytes to 4 KB RAM
  - 4 KB to 60 KB Flash EEPROM
  - 8 MHz or 20 MHz clock
  - Lots of different peripherals
- HCS12 (16 bit)
  - \$10.00 each (and higher)
  - 48 pins to 112 pins
  - 2 KB to 12 KB RAM
  - 1 KB to 4 KB EEPROM
  - 32 KB to 512 KB Flash EEPROM
  - 25 MHz to 50 MHz clock

  - Lots of different peripherals

- S12X (16 bit)
  - $20.00 each (and higher)
  - 48 pins to 112 pins
  - 4 KB to 12 KB RAM
  - 1 KB to 4 KB EEPROM
  - 32 KB to 512 KB Flash EEPROM
  - 25 MHz clock
  - Lots of different peripherals

- 56800 DSP (32 bit)
  - $7.00 each (and higher)
  - 48 pins to 112 pins
  - 4 KB to 32 KB RAM
  - 16 KB to 512 KB Flash EEPROM
  - 32 MHz to 120 MHz clock
  - Specialized for such things as audio processing

- MAC (32 bit)
  - $20.00 each (and higher)
  - 32-bit upgrade of 9S12 line for automotive applications
  - 112 pins to 208 pins
  - 16 KB to 48 KB RAM
  - 384 KB to 1024 KB Flash EEPROM
  - 40 MHz to 50 MHz clock
  - Specialized for such things as audio processing

- ColdFire (32 bit)
  - $40.00 each (and higher)
  - 144 pins to 256 pins
  - 16 MHz to 266 MHz clock

- Power PC (32 bit)
  - $40.00 each (and higher)
  - 272 pins to 388 pins
  - 26 KB to 32 KB RAM
  - 448 KB to 1024 KB Flash EEPROM
  - 40 MHz to 66 MHz clock

Other Manufacturers

- Low end (8 bit)
  - PIC from Microchip
    - Very inexpensive ($0.50)
    - Low pin count (6 to 100)
    - Often small memory (16 bytes RAM, 128 bytes ROM)
    - RISC
  - 8051 (Originally Intel, now National, TI)
  - Z8 (Zilog – similar to 8051)
- Mid-Range (16 bits)
  - Z80 and Z180 from Rabbit
- High End (32 bit)
  - ARM - licensed to Intel, TI, many others
  - MIPS - licensed to Hitachi
- Soft Core
  - Altera NIOS
    - Can customize to meet needs
    - Speed vs. size (number of logic gates)
    - 16-bit or 32-bit
    - Fixed point or floating point
    - Memory management or no memory management
    - Can build specialized instructions to increase performance
  - Xilinx ARM (soft core or hard core)