

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 GNU C compiler, one parameter is passed in the registers. The other parameters are passed on the stack.
 - The rightmost parameter is pushed onto the stack first, then the next-to-right, etc.
 - The left-most parameter is passed in the registers
 - * An 8-bit parameter is put into the B register
 - * A 16-bit parameter is put in the D register
 - * A 32-bit parameter is put into the {X:D} register.
- 8-bit characters are passed as 16-bit integers
- The GNU C compiler for the HC12 seems to have a bug. When the leftmost parameter is a character held in an array, the compiler passes a zero.
 - There are no problems passing integers
- The return value is passed in registers.
 - 8-bit return values are passed in B
 - 16-bit return values are passed in D
 - 32-bit return values are passed in {X:D}
- To link with the Gnu C compiler, need to write code in Gnu AS format
www.cse.unsw.edu.au/~cs3221/labs/assembler-intro.pdf
- In the assembly language program, declare things the C program has to know about as `.global`:

```
.global    foo
```

- In the C program, declare things in the assembly program as `extern`

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

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

Bug in HC12 Gnu C compiler

```
#include "DBug12.h"

typedef char myvar;

int foo(myvar a, char b);
char x[]={1,2,3,4,5};

main()
{
  int i;
  int d;

  for (i=0;i<4;i++) {
    d = foo(x[i],x[i+1]);
    DB12FNP->printf("d = %d, x[i] = %d\r\n",d,x[i]);
  }
  asm(" swi");
}

int foo(myvar a, char b)
{
  return a;
}

/*
  Result when myvar typedef'd as int
>g 2000

d = 1, x[i] = 1
d = 2, x[i] = 2
d = 3, x[i] = 3
d = 4, x[i] = 4
User Bkpt Encountered
*/

/*
  Result when myvar typedef'd as char
>g 2000

d = 0, x[i] = 1
d = 0, x[i] = 2
d = 0, x[i] = 3
d = 0, x[i] = 4
User Bkpt Encountered
*/
```

The C program `example.c`

```
#include "DBug12.h"

extern int foo(int i1, int i2, char c);
extern unsigned int z;

main()
{
    int i1=10, i2=20;
    char c=30;
    int d;

    d = foo(i1,i2,c);
    DB12FNP->printf("d = %d, z = %d\n",d,z);
}
```

The assembly program `foo.s`

```
.global foo
.global z

.sect    .data

z: .byte    2        ; Two bytes for global variable z

.sect    .text

; Stack frame: will allocate four bytes for two 16-bit local variables
; Then will be the return address
; Then the two parameters, x and y
; This is what the stack frame will look like after
; the four bytes are allocated
;   SP      -> v1 (high byte)
;           -> v1 (low byte)
;   SP - 2 -> v2 (high byte)
;           -> v2 (low byte)
;   SP - 4 -> return address high
;           -> return address low
;   SP - 6 -> i2 (high byte)
;           -> i2 (low byte)
;   SP - 8 -> Nothing (c promoted to int)
;           -> c
;
; The parameter i1 is passed in the D register

foo: leas    -4,sp      ; Four bytes on stack for local variables

      std     0,sp      ; Save i1 in local variable
      ldx     -6,sp      ; Put i2 into X register
      movw    #0xaabb,z ; Put 0xaabb into global variable z

      ldd     #0x1234    ; Value to be returned
      leas    4,sp      ; Deallocate stack
      rts
```

C program which calls fuzzy logic assembly function

```
#include "DBug12.h"

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

int main ()
{
    char dPWM;
    int i;

    for (i=0;i<6;i++) {
        dPWM = fuzzy(ERROR[i],d_ERROR[i]);
        DB12FNP->printf("ERROR = %3d, d_ERROR = %3d, ",
        ERROR[i],d_ERROR[i]);
        DB12FNP->printf("dPWM: %4d\r\n", dPWM);
    }
    asm(" swi");
}
```

The Assembly program

```
.global    fuzzy
.sect      .data

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

```

; Fuzzy input membership function definitions for speed error
E_Pos_Medium:    .byte    170, 255,  6,  0
E_Pos_Small:     .byte    128, 208,  6,  6
E_Zero:          .byte    90, 170,  6,  6
E_Neg_Small:     .byte    48, 128,  6,  6
E_Neg_Medium:    .byte    0,  80,  0,  6

; Fuzzy input membership function definitions for differential speed error
dE_Pos_Medium:   .byte    170, 255,  6,  0
dE_Pos_Small:    .byte    128, 208,  6,  6
dE_Zero:         .byte    90, 170,  6,  6
dE_Neg_Small:    .byte    48, 128,  6,  6
dE_Neg_Medium:   .byte    0,  80,  0,  6

; Fuzzy output membership function definition
PM_Output:       .byte    192
PS_Output:       .byte    160
ZE_Output:       .byte    128
NS_Output:       .byte    96
NM_Output:       .byte    64

; Locations for the fuzzy input membership values
I_E_PM:          .byte    0
I_E_PS:          .byte    0
I_E_ZE:          .byte    0
I_E_NS:          .byte    0
I_E_NM:          .byte    0

I_dE_PM:         .byte    0
I_dE_PS:         .byte    0
I_dE_ZE:         .byte    0
I_dE_NS:         .byte    0
I_dE_NM:         .byte    0

; Output fuzzy membership values - initialize to zero
M_PM:            .byte    0
M_PS:            .byte    0
M_ZE:            .byte    0
M_NS:            .byte    0
M_NM:            .byte    0

; Rule Definitions
Rule_Start:      .byte    E_PM,dE_PM,MARKER,O_NM,MARKER
                  .byte    E_PM,dE_PS,MARKER,O_NM,MARKER
                  .byte    E_PM,dE_ZE,MARKER,O_NM,MARKER
                  .byte    E_PM,dE_NS,MARKER,O_NS,MARKER

```

```

        .byte      E_PM,dE_NM,MARKER,O_ZE,MARKER

        .byte      E_PS,dE_PM,MARKER,O_NM,MARKER
        .byte      E_PS,dE_PS,MARKER,O_NM,MARKER
        .byte      E_PS,dE_ZE,MARKER,O_NS,MARKER
        .byte      E_PS,dE_NS,MARKER,O_ZE,MARKER
        .byte      E_PS,dE_NM,MARKER,O_PS,MARKER

        .byte      E_ZE,dE_PM,MARKER,O_NM,MARKER
        .byte      E_ZE,dE_PS,MARKER,O_NS,MARKER
        .byte      E_ZE,dE_ZE,MARKER,O_ZE,MARKER
        .byte      E_ZE,dE_NS,MARKER,O_PS,MARKER
        .byte      E_ZE,dE_NM,MARKER,O_PM,MARKER

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

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

; Main program
        .sect      .text

; Stack frame:
;               SP      -> ERROR
;               SP + 1  -> d_ERROR
;               SP + 2  -> Return address high
;               SP + 3  -> Return address low
;               SP + 4  -> 2'nd param of function high byte
;               SP + 5  -> 2'nd param of function low byte
fuzzy:
        leas      -2,sp      ; Room on stack for ERROR and d_ERROR
        stab      0,sp      ; ERROR passed in B register
        ldab      5,sp      ; d_ERROR passed on stack
        stab      1,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    #0xFF        ; 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
                                SEX      B,D         ; Return answer on stack
                                ; Returning a signed int, so need to
                                ; sign extend B into D
                                leas     2,sp        ; Return stack frame to entry value

                                RTS

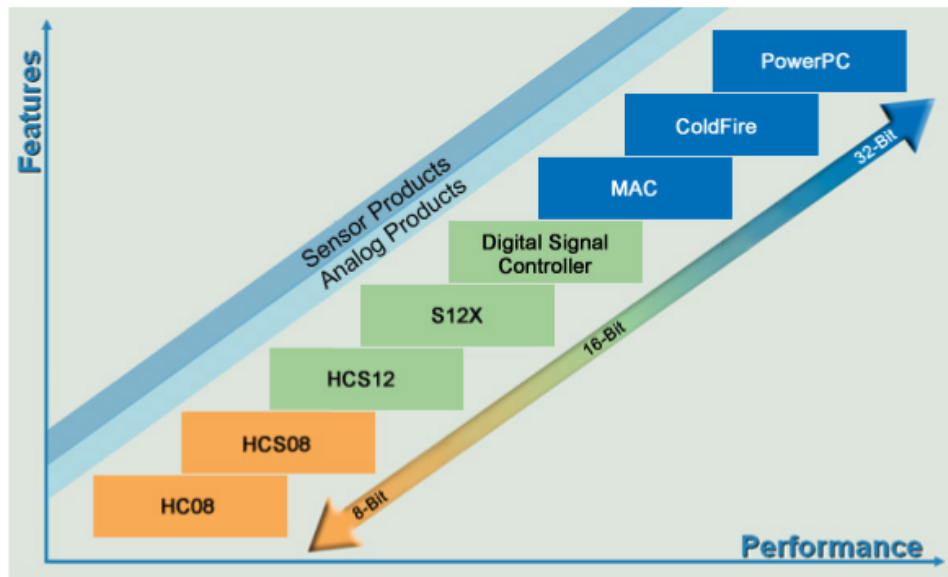
```


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, CISC or SISC?
 - * 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 I²C)
 - 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)