C & ASM

Using the languages together
C and Assembler Together

- C & ASM need not be used separately.
- Within “C” you can insert assembly language using “asm(" nop");”
- ASM is generally used to speed up C code.
  - Typically a function is written in assembler, and called from C.
  - How is information passed between the two?
Let’s See how C does it

```c
// Add two numbers.
int add2_c(int a1, int a2) {
    int a3;
    a3 = a1 + a2;
    return(a3);
}

main() {
    volatile int i, j, k;
    i = 3;  j = 5;
    // Call "C" to add two numbers
    k = add2_c(i,j);
}
```

```
10       main() {
13       main:
0x03122:  8031 0006       SUB.W   #0x0006,SP
0x03126:  40B1 0003 0000  MOV.W   #0x0003,0x0000(SP)
0x0312C:  40B1 0005 0002  MOV.W   #0x0005,0x0002(SP)
    // Call "C" to add two numbers
15       k = add2_c(i,j);
0x03132:  412C            MOV.W   @SP,R12
0x03134:  411D 0002       MOV.W   0x0002(SP),R13
0x03138:  13B0 3158       CALLA   #add2_c
0x0313C:  4C81 0004       MOV.W   R12,0x0004(SP)
```

Performing the function call
- move 1st argument, i, into R12
- move 2nd arg, j, into R13
- call function
- move result from R12 to k

Remember, 1st arg in R12, 2nd in R13
What happens in function?

```c
int add2_c(int a1, int a2) {
    int a3;
    a3 = a1 + a2;
    return(a3);
}
```

Remember, 1st arg in R12, 2nd in R13

<table>
<thead>
<tr>
<th>Stack (w/ locals)</th>
<th>Stack (after call)</th>
<th>Stack (in func)</th>
<th>Stack (clean)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mem</td>
<td>Data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1FE</td>
<td>k</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1FC</td>
<td>j</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1FA</td>
<td>i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SP=1FA</td>
<td>i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1F8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1F4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1F2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1F0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1EF</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In function
- Set up stack
- Move args into locals
- Move a2 to R15
- Add a1 to R15 (i.e., a2)
- Move R15 to a3
- Move a3 (the result) into R12
- Clean up stack
- Return
Can we do better with ASM?

```c
main() {
    volatile int i, j, k;
    i = 3;   j = 5;
    // Call "ASM" to add two numbers
    k = add2_asm(i,j);
}
```

### ASM function
- Share header with "C"
- Declare function “global” so linker finds it.
- Put code in “text” section
- Label “add2_asm”
- Add 2\textsuperscript{nd} arg to 1\textsuperscript{st} and return this value
- Return
- End of ASM

To be fair, the compiler generates the same code from the function `add2_c` if optimizations are turned on.
Rules for passing arguments and using registers

- Registers R11 through R15 can be used without saving values (except when using ISR)
- Caller places first arguments in registers R12-R15, in that order. (If more arguments, put on stack - more about this below)
  - *ints* require one register (16 bits)
  - *chars* (8 bits) use one register
  - *long ints and floats* use two registers (32 bits)
  - a pointer to an array passes its address (16 bits) – (example later)
- Return values are placed in R12 (or R12, R13 if 32 bits).

```c
int manyVars(int a1, int a2, int a3, int a4, int a5, int a6) {
    return (a1 + a2 + a3 + a4 + a5 + a6);
}
main() {
    volatile int i1, i2, i3, i4, i5, i6, k;

    k = manyVars(i1, i2, i3, i4, i5, i6);
}
```

---

In function registers are:
- R12 = i1
- R13 = i2
- R14 = i3
- R15 = i4

Stack (in function)

<table>
<thead>
<tr>
<th>Mem (in arbitrary)</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1F0</td>
<td>??</td>
</tr>
<tr>
<td>1EE</td>
<td>i6</td>
</tr>
<tr>
<td>1EC</td>
<td>i5</td>
</tr>
<tr>
<td>1EA</td>
<td>old PC</td>
</tr>
<tr>
<td>1E8</td>
<td>a6</td>
</tr>
<tr>
<td>1E6</td>
<td>a5</td>
</tr>
<tr>
<td>1E4</td>
<td>a4</td>
</tr>
<tr>
<td>1F2</td>
<td>a3</td>
</tr>
<tr>
<td>1E0</td>
<td>a2</td>
</tr>
<tr>
<td>SP=1DE</td>
<td>a1</td>
</tr>
</tbody>
</table>
Global variables

```c
int n_g=4; // Global variable

// Add three numbers, one global
int add2_c(int a1, int a2) {
    volatile int a3;
    a3 = a1 + a2 + n_g;
    return(a3);
}

Main () {
    volatile int i, j, k;
    i=2; j=3;

    k = add2_c(i, j);

    k = add2_asm(i, j);
}
```

Example

• Weighted average (i.e., FIR filter)

• $y_i$ is output, $x_i$ is input: $y_i = \sum_{k=0}^{N-1} b_k x_{i-k}$

• Example weighted sum with middle point weighted twice as much as others:

  \[
  b = \{0.25, 0.50, 0.25\}
  \]
  \[
  y_i = \sum_{k=0}^{2} b_k x_{i-k} = 0.25x_{i-2} + 0.50x_{i-1} + 0.25x_i
  \]

• Very powerful technique at heart of digital signal processing

• Without loss of generality we can restate as:

  \[
  y_i = \sum_{k=0}^{N-1} b_k x_k \text{ where the } x_k = \text{ most recent inputs}
  \]
Implement FIR in C

```c
main() {
    const int n=3;
    const int b[]={0x2aaa, 0x2aaa, 0x2aaa}; // 1/3, 1/3, 1/3 (weights - Q15)
    int x[]={0x4000, 0x4000, 0x4000}; // last values of input
    int y; // Output

    y = fir_c (b, x, n);
}

int fir_c(const int *b, int *x, int n) {
    long sum=0;
    int i;
    for (i=0; i<n; i++)
        sum += (long) b[n] * x[n]; // Q15*Q15 is Q30
    return ((int) (sum>>15)); // Q30 to Q15
}
```

For 3 weights: 353 cycles or 118 cycles/weight
For 100 weights: 8792 cycles or 88 cycles/weight
16 bit multiplier/accumulator (MAC)

If OP1 and OP2 are Q15, RESHI:RESLO is Q30.
Implement FIR in ASM

```c
main() {
    const int n=3;
    const int b[]={0x2aaa, 0x2aaa, 0x2aaa};
    int x[]={0x4000, 0x4000, 0x4000};
    int y;
    y = fir_asm (b, x, n);
}
```

```
.global fir_asm
;
.text
;
.fir_asm
mov     #0, &RESHI
mov     #0, &RESLO

.fir_loop
mov     @r12+, &MACS
mov     @r13+, &OP2
dec     r14
jnz     fir_loop
rlc     &RESLO
rlc     &RESHI
mov     &RESHI, r12
ret

.end
```

**FIR in ASM**
- Clear result register
- Move b[0] into MACS, increment pointer
- Move x[0] into OP2, increment pointer (this performs mult/accum)
- Decrement n, check for zero, loop
- Left shift RESLO, MSB into Carry
- Left shift RESHI, Carry into LSB
- Mover RESHI into result and return

For 3 weights: 74 cycles or 25 cycles/weight (4.8 times faster than C)
For 100 weights: 1141 cycles or 11 cycles/weight (7.7 times faster than C)
Can we make C faster with pointers?

```c
int fir_c(const int *b, int *x, int n) {
    long sum=0;
    while (--n>=0)
        sum += (long) (*b++) * (*x++);
    return ((int) (sum>>15));
}
```

Yes – but only about 5%

Using ASM routines in practice

In practice: write and debug routine in C, then convert to assembler (perhaps using compiled C code as template)

References