C vs Assembler

Background

Assembler is very powerful. In assembler, you have complete control over where your code compiles and where your variables are stored. Assembler is also very fast and efficient. Assembler is very difficult to read, however, and tends to be throw-away code. It's easier to start from scratch and ignore already written assembler code and start over than to modify existing code.

C is a little constraining. However, it is easier to write C code which is understandable and can be reused with relative ease.

The following examples of C code and its assembler counterpart

1. Binary Outputs

Write a program which sends the numbers {1, 2, 3, 4} to PORTA..D

C-Code

```
// Subroutine Declarations
#include <pic18.h>

// Subroutines

// Main Routine

void main(void)
{

   TRISA = 0;
   TRISB = 0;
   TRISC = 0;
   TRISD = 0;
   TRISE = 0;
   ADCON1 = 0x0F;

   PORTA = 1;
   PORTB = 2;
   PORTC = 3;
   PORTD = 4;

   while(1);

}
```

Compilation Results:

```
Memory Summary:
    Program space        used    3Ah (    58) of 10000h bytes   (  0.1%)
    Data space           used     1h (     1) of   F80h bytes   (  0.0%)
    EEPROM space         used     0h (     0) of   400h bytes   (  0.0%)
    ID Location space    used     0h (     0) of     8h nibbles (  0.0%)
    Configuration bits   used     0h (     0) of     7h words   (  0.0%)
```

This C code compiles into 29 lines of assembler (58 bytes: each instruction is 16 bits or two bytes)

When you download this code to your PIC board, you should see this:

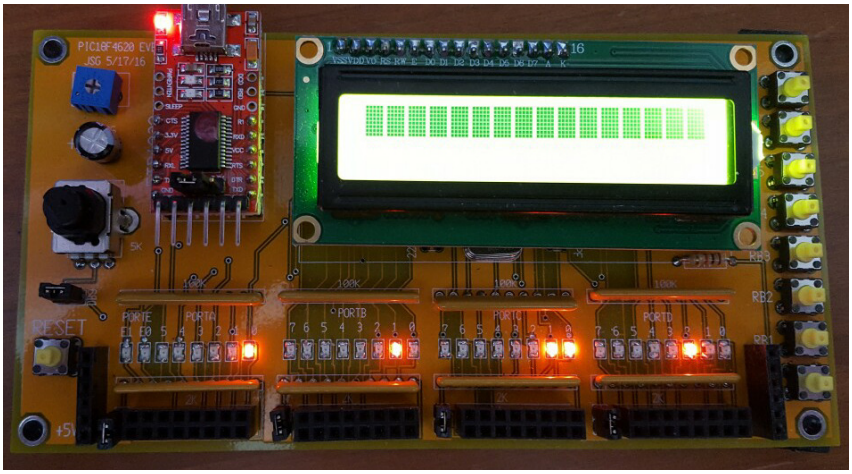

Figure 1: PORTA = 1, PORTB = 2, PORTC = 3, PORTD = 4

The conversion from C to assembler shows up in the .LST file

18F4620\Clock\Test.C

```
// Subroutine Declara
#include <pic18.h>

// Subroutines

// Main Routine

void main(void)
{

   TRISA = 0;
   TRISB = 0;
   TRISC = 0;
   TRISD = 0;
   TRISE = 0;
   ADCON1 = 0x0F;

   PORTA = 1;
   PORTB = 2;
   PORTC = 3;
   PORTD = 4;

   while(1);

}
```

C:\ECE376_18F4620\Clock\Clock.lst

```
159        151  00FFE8  51FF            movf    (??_main+0+0)&0ffh,w
160        152                          line    20
161        153                        ;Test.C: 20: PORTA = 1;
162        154  00FFEA  0E01            movlw   low(01h)
163        155  00FFEC  6E80            movwf   ((c:3968)),c    ;volatile
164        156                          line    21
165        157                        ;Test.C: 21: PORTB = 2;
166        158  00FFEE  0E02            movlw   low(02h)
167        159  00FFF0  6E81            movwf   ((c:3969)),c    ;volatile
168        160                          line    22
169        161                        ;Test.C: 22: PORTC = 3;
170        162  00FFF2  0E03            movlw   low(03h)
171        163  00FFF4  6E82            movwf   ((c:3970)),c    ;volatile
172        164                          line    23
173        165                        ;Test.C: 23: PORTD = 4;
174        166  00FFF6  0E04            movlw   low(04h)
175        167  00FFF8  6E83            movwf   ((c:3971)),c    ;volatile
176        168                          line    25
177        169                        ;Test.C: 25: while(1);
178        170
179        171  00FFFA                  l1111:          ; BSR set to: ?
180        172
181        173
182        174  00FFFA                  l1110:          ; BSR set to: ?
183        175
```

Figure 2: Resulting program memory assignment from the dot-lst file

Note

- Each line of C is converted to several lines of assembler
- The assembler code is pretty much the way I would code this by hand - only with the actual address of PORTA..D being used rather than the name.

Also note that the code at the end

```
while(1);
```

is a "stop" command. The program is always running. To make it stop, put it in an infinite loop.

Example 2: Counting (making a light blink)

```
// Subroutine Declarations
#include <pic18.h>

void main(void)
{
   unsigned char X;

   TRISA = 0;
   TRISB = 0;
   TRISC = 0;
   TRISD = 0;
   TRISE = 0;
   ADCON1 = 0x0F;

   while(1) {
      X = X + 1;
      PORTD = X;
      }
   }
```

The compilation results are:

```
Memory Summary:
    Program space        used    3Ah (    58) of 10000h bytes   (  0.1%)
    Data space           used     2h (     2) of   F80h bytes   (  0.1%)
    EEPROM space         used     0h (     0) of   400h bytes   (  0.0%)
    ID Location space    used     0h (     0) of     8h nibbles (  0.0%)
    Configuration bits   used     0h (     0) of     7h words   (  0.0%)
```

This code compiles into 29 lines of assembler. The conversion from C to assembler shows up again in the .LST file:

620\Clock\Clock.lst

```
153  00FFE8  51FF                movf    (??_main+1+0)&0ffh,w
154                              line    21
155                           ;Test.C: 21: while(1) {
156
157  00FFEA                     l111:           ; BSR set to: ?
158
159                              line    22
160                           ;Test.C: 22: X = X + 1;
161  00FFEA  0100                movlb   0   ; () banked
162  00FFEC  0100                movlb   0   ; () banked
163  00FFEE  29FE                incf    ((main@X))&0ffh,w
164  00FFF0  0100                movlb   0   ; () banked
165  00FFF2  0100                movlb   0   ; () banked
166  00FFF4  6FFE                movwf   ((main@X))&0ffh
167                              line    23
168                           ;Test.C: 23: PORTD = X;
169  00FFF6  C0FE  FF83          movff   (main@X),(c:3971)   ;volatile
170                              line    24
171
172  00FFFA                     l1110:          ; BSR set to: ?
173
174                              line    21
175  00FFFA  D7F7                goto    l111
176                              global  start
177  00FFFC  EF00  F000          goto    start
178                              opt stack 0
179                           GLOBAL __end_of_main
180  010000                      __end_of_main:
181                              FNSIZE  _main,2,0
182                           ; ================ function  main ends ===
```

Figure 3: Contents of program memory for the blinking-light program

This isn't exactly how I would code it by hand. To increment variable X, for example, I would define where X is in memory:

```
X EQU 1;
```

then increment it in the main routine

```
incf   X,F
```

You can also see that the main loop compiles into 8 lines of assembler, taking 9 clocks to execute. This routine counts every 9 clocks (900ns).

Example 3: 32-Bit Counter

One nice feature of C is that you have more than just 8-bit variables. You can also use

- 8-bit variables (char)
- 16-bit variables (integer)
- 32-bit variables (long integer)
- 32-bit floating point numbers (float)
- 64-bit floating point numbers (double)

When you copy a 32-bit variable into an 8-bit variable

- The low 8-bits are copied
- The other bits are ignored

Suppose you want to count using a 32-bit variable. This code could be

```
// Subroutine Declarations
#include <pic18.h>

// Subroutines

// Main Routine

void main(void)
{
   unsigned long int X;

   TRISA = 0;
   TRISB = 0;
   TRISC = 0;
   TRISD = 0;
   TRISE = 0;
   ADCON1 = 0x0F;

   X = 0;

   while(1) {
      X = X + 1;
      PORTD = X;
      PORTC = X >> 8;
      PORTB = X >> 16;
      PORTA = X >> 32;
      }
   }
```

Note: The command X >> 8 shifts X right eight times. This results in

- Bits 0..7 going to PORTD
- Bits 8..15 going to PORTC
- Bits 16..23 going to PORTB, and
- Bits 24..31 going to PORTA

The compilation results are:

```
Memory Summary:
    Program space        used    86h (   134) of 10000h bytes   (  0.2%)
    Data space           used     5h (     5) of   F80h bytes   (  0.1%)
    EEPROM space         used     0h (     0) of   400h bytes   (  0.0%)
    ID Location space    used     0h (     0) of     8h nibbles (  0.0%)
    Configuration bits   used     0h (     0) of     7h words   (  0.0%)
```

The code compiles into 67 lines of assembler (134/2).

When downloaded to your PIC board, it counts as a 32-bit counter

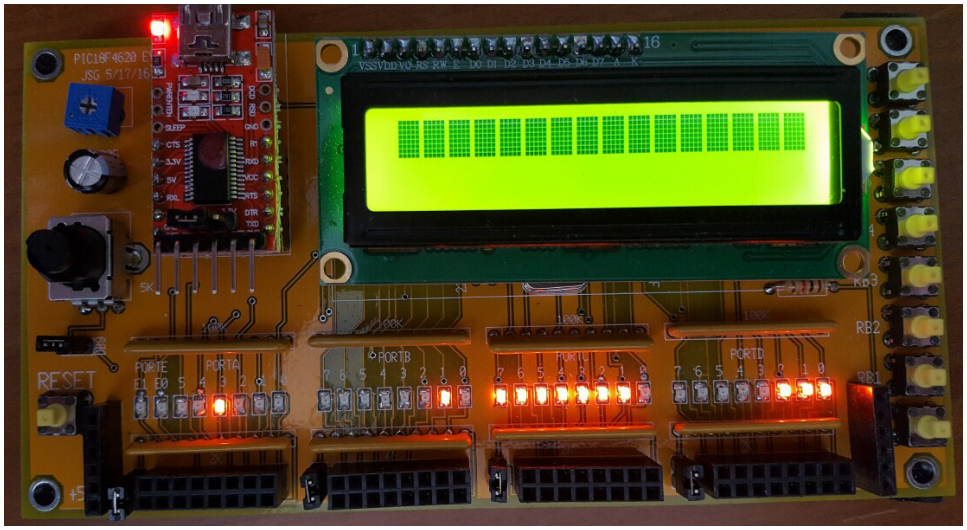

Figure 4: Counting as a 32-bit counter: PORTA ; PORTB : PORTC : PORTD

Now, the conversion to assembler isn't so easy to follow. The single line fo C code

```
X = X + 1
```

compiles into 25 lines of assembler as follows:

```
;Test.C: 24: X = X + 1;
    movlw   01h
    movlb   0    ; () banked
    movlb   0    ; () banked
    addwf   ((main@X))&0ffh,w
    movlb   0    ; () banked
    movlb   0    ; () banked
    movwf   ((main@X))&0ffh
    movlw   0
    movlb   0    ; () banked
    movlb   0    ; () banked
    addwfc  ((main@X+1))&0ffh,w
    movlb   0    ; () banked
    movwf   1+((main@X))&0ffh
    movlw   0
    movlb   0    ; () banked
    movlb   0    ; () banked
    addwfc  ((main@X+2))&0ffh,w
    movlb   0    ; () banked
    movwf   2+((main@X))&0ffh
    movlw   0
    movlb   0    ; () banked
    movlb   0    ; () banked
    addwfc  ((main@X+3))&0ffh,w
    movlb   0    ; () banked
    movwf   3+((main@X))&0ffh
    line    25
```

Figure 5: The dot-lst shows how initialization in C is converted to assembler

The compiler can be pretty tricky as well. Shifting the data to PORTA..D is as follows:

```
;Test.C: 25:  PORTD = X;
   movff   (main@X),(c:3971)    ;volatile
   line    26
;Test.C: 26:  PORTC = X >> 8;
   movff   0+1+(main@X),(c:3970)    ;volatile
   line    27
;Test.C: 27:  PORTB = X >> 16;
   movff   0+2+(main@X),(c:3969)    ;volatile
   line    28
;Test.C: 28:  PORTA = X >> 32;
   movff   (main@X),(c:3968)    ;volatile
   line    29
```

Figure 6: The dot-lst file shows how theC code is convertered to assembler

X is a 32-bit variables, stored in four bytes in RAM. The C compiler places the low byte at memory location main@x. The next byte is at main@x+1, and so on.

Example 4: Subroutines and Wait loops

Another nice feature of C is you have access to subroutines. Suppose you want to write a routine which waits X milliseconds. One way to do this is

```
// Subroutine Declarations
#include <pic18.h>

// Subroutines
void Wait(unsigned int X)
{
   unsigned int i, j;
   for (i=0; i<X; i++)
      for (j=0; j<185; j++);
   }


// Main Routine

void main(void)
{
   unsigned char X;

   TRISA = 0;
   TRISB = 0;
   TRISC = 0;
   TRISD = 0;
   TRISE = 0;
   ADCON1 = 0x0F;

   X = 0;

   while(1) {
      X = X + 1;
      Wait(1000);
      }
   }
```

The subroutine **Wait()** can be called whenever you want to wait so many milliseconds. The number 500 is a guess: whatever it takes to burn one millisecond (10,000 clocks). If you look, the code

```
for (j=0; j<500; j++);
```

compiles into 54 lines of assembler (taking 5.4us). Executing this 185 times should result in about 10,000 clocks (1ms) - meaning - change the 500 to 185 and it *should* take 1ms.

Another way to determine this number '500' is to write a test routine (i.e. test your code)

```
while(1) {
   PORTC = PORTC + 1;
   Wait(10);
   }
```

Using an oscilloscope, check the timing on RC0: the period should be 20ms (10ms low, 10ms high). If not, adjust the constant 500 to make it 10ms.

```
;Test.C: 11: for (j=0; j<500; j++);
   movlw   low(0)
   movlb   0    ; () banked
   movlb   0    ; () banked
   movwf   ((Wait@j))&0ffh
   movlw   high(0)
   movlb   0    ; () banked
   movwf   ((Wait@j+1))&0ffh
   movlw   0F4h
   movlb   0    ; () banked
   movlb   0    ; () banked
   subwf   ((Wait@j))&0ffh,w
   movlw   01h
   movlb   0    ; () banked
   subwfb  ((Wait@j+1))&0ffh,w
   btfss   status,0
   goto    u41
   goto    u40
  u41:
   goto    l1114
  u40:
   goto    l1115

  l1114:        ; BSR set to: ?

   movlb   0    ; () banked
```

Figure 7: Part of the *for (j=0; j<500; j++);* conversion to assembler

Example 5: Binary Clock (take 2)

Once the wait routine works, you can write the code for a binary clock much easier in C:

```
// Global Variables

// Subroutine Declarations
#include <pic18.h>

// Subroutines
void Wait(unsigned int X)
{
   unsigned int i, j;
   for (i=0; i<X; i++)
      for (j=0; j<500; j++);
   }
```

```
// Main Routine

void main(void)
{
   unsigned char SEC, MIN, HR;

   TRISA = 0;
   TRISB = 0;
   TRISC = 0;
   TRISD = 0;
   TRISE = 0;
   ADCON1 = 0x0F;

   SEC = 0;
   MIN = 0;
   HR = 0;

   while(1) {
      SEC = SEC + 1;
      if (SEC > 59) {
         SEC = 0;
         MIN = MIN + 1;
         }
      if (MIN > 59) {
         HR = HR + 1;
         MIN = 0;
         }
      if (HR > 12) HR = 1;

      PORTD = SEC;
      PORTC = MIN;
      PORTB = HR;

      Wait(1000);

      }
   }
```

The compilation results are:

```
Memory Summary:
    Program space        used   174h (   372) of 10000h bytes   (  0.6%)
    Data space           used     Ah (    10) of   F80h bytes   (  0.3%)
    EEPROM space         used     0h (     0) of   400h bytes   (  0.0%)
    ID Location space    used     0h (     0) of     8h nibbles (  0.0%)
    Configuration bits   used     0h (     0) of     7h words   (  0.0%)
```

This program compiles into 186 lines of assembler (372/2).

- That's 2.16 times more than what we came up with when coding directly in assembler (86 lines). Typically, C is 3-10 times less efficient than assembler.
- The coding was a lot easier, and
- If() statements can do almost anything.

In-Line Assembler:

In almost every C language, there is a way to insert assembler code into your C code. With Hi-Tech C, the commands are

```
asm(" nop ");
```

This inserts the assembler command "nop" into the code at this point in your C code.

You can also insert several lines of assembler with the compiler directives #asm and #endasm

```
#asm
   nop
   nop
   nop
#endasm
```

Normally you don't want to do this:

- assembler is much harder to understand and debug
- assembler is much harder to maintain

but

- assembler is 3-10 times smaller and faster than C

The times you would do this are

- When your compiled C code doesn't fit in your processor. You need to make the code smaller somehow.
- When your compiled C code takes too long to execute. You need to speed it up somehow.

For the latter, what's often done is monitor how much time is spend in each routine. The routine which is eating up most of your time is then hand-coded into assembler (biggest bang for the buck). This results in code which is much harder to maintain, but it is faster.