Chapter 4 - NUMB3RS

Choosing the right “types”
# Integer Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Bits</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>char, signed char</td>
<td>8</td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>unsigned char</td>
<td>8</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>short, signed short</td>
<td>16</td>
<td>-32768</td>
<td>32767</td>
</tr>
<tr>
<td>unsigned short</td>
<td>16</td>
<td>0</td>
<td>65535</td>
</tr>
<tr>
<td>int, signed int, long, signed long</td>
<td>32</td>
<td>$-2^{31}$</td>
<td>$2^{31}-1$</td>
</tr>
<tr>
<td>unsigned int, unsigned long</td>
<td>32</td>
<td>0</td>
<td>$2^{32}-1$</td>
</tr>
<tr>
<td>long long, signed long long</td>
<td>64</td>
<td>$-2^{63}$</td>
<td>$2^{63}-1$</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>64</td>
<td>0</td>
<td>$2^{64}-1$</td>
</tr>
</tbody>
</table>
Long Multiplication (32-bit)

main()
{
    int i, j, k;
    i = 1234; // assign an initial value to i
    j = 5678; // assign an initial value to j
    k = i * j; // multiply and store the result in k
}
// How does the assembly code work?
// What if multiplication gives overflow?
// Is this code time/space efficient?

32-bit and smaller data can mostly be handled in 1 clock cycle and with 1 assembly instruction. PIC32 contains a 32-bit MIPS core (what’s in core?), i.e. a RISC machine.
Long Long Multiplication
(64-bit x 64-bit = ?)

main()
{
    long long i, j, k;
    i = 1234; // assign an initial value to i
    j = 5678; // assign an initial value to j
    k = i * j; // multiply and store the result in k
}
// multu: 64-bit unsigned product in Hi | Lo
// mflo a0: a0 = Lo (reg Lo for mul/div result)
// mfhi a1: a1 = Hi (reg Hi for mul/div result)
// How many 32-bit multiplications are needed to perform a 64-bit by 64-bit multiplications?
// How many 32-bit additions are needed to perform a 64-bit by 64-bit multiplications?
// Does this code segment handle overflow?
// Use Watch Window to see register values.

Exploring The PIC32 - Lucio Di Jasio
main()
{
    int i, j, k;
    i = 1234;    // assign an initial value to i
    j = 5678;    // assign an initial value to j
    k = i / j;   // divide and store the result in k
}

// Compact, optimization-free instructions.
// 32-bit instruction div does the job.
// div v1,v0: Lo = quotient; Hi = remainder
// teq v0,zero: trap exception if v0 = 0 (div by 0)
// Time to put our experience in instruction set architectures to good use!

15:                      k = i/j;
    9D00001C 8FC30000 lw       v1,0(s8)
    9D000020 8FC20004 lw       v0,4(s8)
    9D000024 0062001A div       v1,v0
    9D000028 004001F4 teq       v0,zero
    9D00002C 00001012 mflo      v0
    9D000030 AFC20008 sw       v0,8(s8)
Long Long Integer Division

main () {
    long long  i, j, k;
    i = 1234;  // assign an initial value to i
    j = 5678;  // assign an initial value to j
    k = i / j;  // divide and store the result in k
} // a0 a1 store one of the 64-bit operands.
    // a2 a3 store the other 64-bit operand.
// jal <addr>: $ra = PC+4, go to <addr>, where
    <addr> is the location of the subroutine that
    performs the 64-bit integer division.
// Subroutines are provided for frequently needed
    operations to conserve overall code size.
// What is the nop (lazy for 1 clock cycle) for?
    (Hint: What is the current value of PC?)

Exploring The PIC32 - Lucio Di Jasio
Floating-Point Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>32</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
</tr>
<tr>
<td>long double</td>
<td>64</td>
</tr>
</tbody>
</table>

These types conform to IEEE 754 standard for floating-point formats. Without an FPU, PIC32 uses FP libraries that are much larger and complex than INT libraries.

Experiment with the code on P. 69 of Di Jasio and see.
The Simulator Stopwatch

Choosing the best fit data types for each application can be critical in embedded systems to balance performance while optimizing the use of (often limited) resources.
## Multiplication Test Results

<table>
<thead>
<tr>
<th>Multiplication Test</th>
<th>Width (Bits)</th>
<th>Cycle Count</th>
<th>Performance relative to Int</th>
<th>Performance relative to float</th>
</tr>
</thead>
<tbody>
<tr>
<td>Char Integer (char)</td>
<td>8</td>
<td>6</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Short Integer (short)</td>
<td>16</td>
<td>6</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Integer (int, long)</td>
<td>32</td>
<td>6</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Long Integer (long long)</td>
<td>64</td>
<td>21</td>
<td>3.5</td>
<td>-</td>
</tr>
<tr>
<td>Single Precision FP</td>
<td>32</td>
<td>71</td>
<td>11.8</td>
<td>1</td>
</tr>
<tr>
<td>(float, double)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double Precision FP</td>
<td>64</td>
<td>159</td>
<td>26.5</td>
<td>2.23</td>
</tr>
<tr>
<td>(long double)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See `NUMB3RS4.c` from the project code, and pp. 70~73 of Di Jasio for more details. (Be careful with docking!)
Simple Type Conversions

Implicit Integer Type Conversion

Example:

```c
short s;    // 16-bit
int i;      // 32-bit
i = s;      // OK!
```

Explicit Integer Type Conversion

Example:

```c
short s;  // 16-bit
int i;    // 32-bit
s = (short)i; // Must!
```
Bit Fields (< 1 Byte Size)

You can access each bit field using the “dot” notation, as in the following examples for SFR control bits:

```c
// Timer1 module control
T1CONbits.TON = 1;

// JTAG interface enable
DDPCONbits.JTAGEN = 0;
```
# Exact-Width Integer Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8_t</td>
<td>always an 8-bit signed type</td>
</tr>
<tr>
<td>uint8_t</td>
<td>always an 8-bit unsigned type</td>
</tr>
<tr>
<td>int16_t</td>
<td>always a 16-bit signed type</td>
</tr>
<tr>
<td>uint16_t</td>
<td>always a 16-bit unsigned type</td>
</tr>
<tr>
<td>int32_t</td>
<td>always a 32-bit signed type</td>
</tr>
<tr>
<td>uint32_t</td>
<td>always a 32-bit unsigned type</td>
</tr>
<tr>
<td>int64_t</td>
<td>always a 64-bit signed type</td>
</tr>
<tr>
<td>uint64_t</td>
<td>always a 64-bit unsigned type</td>
</tr>
</tbody>
</table>

Important when porting 32-bit code *down* to 16-bit or even lower-bit processors (for overall system cost purposes). These integer types are specified in `inttypes.h`. 
Some Math Libraries

The X/C32 compiler supports several ANSI C libraries:

- “limits.h”, contains macros that define implementation-dependent limits, such as the number of bits in the character type (CHAR_BIT), the largest integer value (INT_MAX), etc.

- “float.h”, contains similar implementation-dependent limits for floating-point data types, such as the largest exponent for a single-precision floating-point variable (FLT_MAX_EXP).

- “math.h”, contains trigonometric functions, rounding functions, logarithms, exponentials, and constants such as PI (M_PI).
The variable \( z \) so defined has now a \textit{real} and an \textit{imaginary} part that can be individually addressed using the following syntax:

\begin{verbatim}
__real__  z  
and
__imag__  z
\end{verbatim}

Similarly, this declaration produces a complex variable of 32-bit integer type:

\begin{verbatim}
__complex__ int x;
\end{verbatim}

Complex constants can then be created by adding the suffix “\textit{i}” or “\textit{j}”:

\begin{verbatim}
x = 2 + 3j;
z = 2.0f + 3.0fj;
\end{verbatim}

Notice the use of a \textbf{double underscore} before and after the keyword \texttt{complex}. 

Exploring The PIC32 - Lucio Di Jasio