【組込みシステム編】第５回 組込みシステム

こんにちは、南角 です。
前回書いたように、浮動小数点数に関して、まだ round や 符号、 メモリ上の形式などの話などが残っていますので、今回はまず符号の話から始めたいと思います。

皆さんご存じのとおり、多くの CPU では、符合有りの整数において負数を表現する場合、2 の補数を用いています。
2 の補数に関する正確な定義はともかくとして、私は実用上はその値の正の整数と足すと 0 となる数が 2 の補数表現におけるマイナス表現だと考えています。
例えば整数が 16 ビットの場合、10 進数の 5 は 2 進数では 0000 0000 0000 0101 となり、
1111 1111 1111 1011 と加えると 1 0000 0000 0000 0000 となり、オーバーフローを無視すると
0000 0000 0000 0000 と 10 進数でいうと 0 になります。
そこで 1111 1111 1111 1011 が-5 というわけです。
ここで MSB (Most Significant Bit - 最上位ビット)が符号ビットだと説明される方もおられますが、それは違うと思います。
確かに MSB を見れば、その数が正か負かは判断できますが、MSB で符号を決めているわけではありません。
もし MSB で符号を決めているのであれば、16 ビットの 2 進数でいうと 1000 0000 0000 0000 が-0 となり、2 の補数では+0 と-0 が存在することになりますがそうではありません。
1000 0000 0000 0000 は 10 進数でいうと-32768、負数のもっとも小さな数になります。

しかし浮動小数点における仮数部の表現は、MSB は本当に符号ビットです。
ですから浮動小数点においては＋0.0 と－0.0 が存在します。
ちなみに浮動小数点における指数部の負数の表現形式は、指数部にゲタ（バイアス）を加えて常に正の値で表したものとしています。たとえば指数部が 8 ビットであれば、常に 127 を加えて正の数（0 と 255 は特殊な意味で用いるため 1 から 254）となるようにしています。
以上の意味では、浮動小数点表現における仮数部も指数部も符号なし 2 進数ということになります。

なお、浮動小数点の広く採用されている規格である IEEE 754 においては、4 倍精度という 128 ビット（16 バイト）のものも存在して、この形式では仮数部も 112 ビットあるようなので、long long の値を格納した場合の精度落ちも発生しないと思いますが、まだ私は使用したことがないため、これ以上は何とも言えません。

ところで、私が浮動小数点プロセッサ(FPU)というものを最初に使用したのは、確か入社 2、3 年の頃で、インテルの i8087 でした。
当時は i8086 のコンパイラを使用していましたが、むろん当初はコンパイラのサポートはなく、C から呼び出

せる関数をアセンブラで作成しました。
80 ビットのレジスタ FP0 から FP7 がありましたが、これがスタック構造だったため、コメント部分に、命令ごとにスタック上のデータ位置を記述したことを覚えています。
FPU を最初に使用したのは i8086+i8087 で、はじめは C から呼び出せる関数をアセンブラで作成して演算時間を短くするのに利用していました。
やがて新しい機種を開発することになり、その機種ではさらなる性能、機能の向上のため
MC68020+MC68881 を使用することになりました。
MC68881 というのが FPU です。
i8087 は i8086 と同時に命令をフェッチしており、命令によって i8086 の命令なのか i8087 の命令なのかが区別されており、それぞれが自分の命令を実行する方式でした。
それに対して MC68881 はいわば演算用デバイスで、必要に応じて MC68020 から呼び出される方式だったと思います。
当時の FPU はいわば贅沢品で、C 言語における double が 8 バイト(64 ビット)であるにも関わらず、
内部的には i8087 で 10 バイト(80 ビット)、MC68881 で 12 バイト(96 ビット)のレジスタを持ち演算を行っています。
RISC などでは内部的にも 8 バイト(64 ビット)のレジスタしか持たず、浮動小数点演算を行う場合も多く、その結果、ソースコードが同じであっても、PC などで行うシミュレータと実機での動作が異なる場合もあることは、以前もお伝えした通りです。
また PC などを使う場合に注意することは、内部的には 10 バイトのデータをメモリに保存する時は、途中に 2 バイトの詰め物(padding)をして 12 バイトになることです。
これはもちろん 32 ビット CPU なので、メモリアクセスの効率化（data alignment）のためだと思いますが、サイズを 4 の整数倍にするためだと思います。
現状の Intel の 64 ビット CPU だと効率化のために 8 の整数倍になるのでしょうか？
残念ながら手元に 64 ビット対応のコンパイラがないため確認できません。

すこし驚いたのは浮動小数点の値を整数化する時です。
確か fmove だったと思いますが、MC68881 には FPU 内部のレジスタを形式を指定してメモリに書き込むだけで、FPU の丸めモードにしたがって自動的に型変換してくれたため、丸めモードを round にしておけば四捨五入だろうと考えていたところ、実は偶数への丸めだったことです。
浮動小数点の値を整数に変換する場合、最も近くの表現できる整数へ丸める。さらに 0.5 のように表現可能な 2 つの値の中間の値であったら、一番低い仮数ビットが 0 になるほうに丸めるという動作です。
これは世の中の四捨五入とは別の動作になります。たとえば 1.5 を丸めた場合 2 になります。これはいいのですが 2.5 を丸めた場合も 2 になります。
最初に使用した FPU(MC68881)対応コンパイラでは、double の変数を long に（明示的にせよ、暗黙的にせよ）変換する場合に単に fmove 命令を使用していました。
従来、FPU を使用せずソフトウェアで処理していた時は、四捨五入の動作をしていたものが、FPU に対応するバージョンに置き換えた時に微妙に計算結果が異なり、客先からクレームが来ました。
初期のころはコンパイラ屋さんもよくわかっていなかったようで、FPU のモードを利用した整数変換になっており、この問題が発生しました。
この時は対策として、C からアセンブラに変換して、そのアセンブラのソースコードをきちんと四捨五入する

アセンブラに置き換えるというツールを作成して、変換を行った後、アセンブルして機械語を生成するということを行いました。

何か昔話のようになってしまいましたが、言いたかったことは、コンパイラも 100%信用してはいけないということです。
このようなこともあったため、今でも時に応じて C や C++のソースコードから生成されたアセンブラ、そしてアセンブラであってもそのまま機械語になるとは限らないため、実際に生成されたオブジェクトを逆アセンブルして、確認するということが習慣になっています。
それなりに長い間使われているコンパイラであれば、不具合も少ないとは思いますが、とはいえコンパイラもあくまでソフトウェアです。
最適化オプションや状況によっては不具合が残っていたかもしれません。
機械語で見ないとわからない不具合もあります、みなさんもアセンブラだからと言って敬遠せず、是非アセンブラも好きになってください。
アセンブラを知ることにより、使用している CPU も、わかってくるはずです。

もう一つ言っておきたいことがあります。
CPU はハードウェア屋さんの領域だと思っていませんか？私は CPU もソフトウェアだと思っています。
CPU の電気特性や信号のタイミング、接続などはハードウェア屋さんの領域だと思いますが、CPU の動作に関してはソフトウェア屋さんの仕事です。
むしろソフトウェア屋さんがハードウェア屋さんに教えてあげないといけません。
もうひとつ、当たり前ですが、ハードウェア屋さんとは仲良くしてください。
ハードウェア屋さんはハードウェア屋さんで、ソフトウェア屋に気を使って、自分の判断でソフトウェアに関係ないと思った場合にその情報を伝えてこない場合があります。例えば、ある製品で製造中止に伴い EEPROM が変更になったことがありました。
EEPROM 自体は変更になりましたが、ソフトウェアからみた、書き込み方法などその使用方法に変更がなかったため、特に変更の連絡はありませんでした。
おそらく書類や会議で「EEPROM が変更になったが、互換品なのでソフトウェアの変更は不要」くらいの連絡はあったと思います。しかしこちらも上記のような連絡があれば特に気にはしません。
しかし、実は EEPROM の特性に変更があり、書き込み速度がかなり遅くなっていました。この EEPROM は製品の型番、パラメータなども保持しており、こちらは書き込み速度が遅くなっても大した問題はありません。
しかし、重要な用途があり、それは、電源断割り込みにより保持しておかなければならないデータを保存することでした。
もう少し詳しく説明しますと、製品の電源を切っても、一瞬で 200V や 100V が 0 になるわけではなく、電圧は徐々に低下します。そしてある電圧より低下した時に、電源低下割り込みを発生させ、ハードウェアが動作可能な電圧を保っている間に、保存データの EEPROM への書き込みを行っていました。
これは時間との勝負であり、製品によってはコンデンサーなどを付加して、低下する時間を延ばしたりもしていました。
もうお分かりだと思いますが、EEPROM の変更により、電源断の時に、従来は書き込みが間に合っていたデータが、失われるようになりました。
これも一種のリアルタイム性であり、ハードウェアの変更により、処理が間に合わなくなった例です。

その他、温度によってデバイスの書き込み速度が遅くなるために生じる不具合など、ハードウェアとソフトウェアが一体となって成立する組み込みシステムならではのものだと思います。
使用される環境という意味では、紫外線の影響が大きい宇宙空間や、ノイズが激しいうえに、規制上？ハードウェア的なノイズ対策ができにくいパチンコ台などにも、一定周期でリセット信号を入れるなどの、独自の対策が必要だったりします。
デバイスに関しては、フラッシュメモリなどは書き込み回数の上限（寿命）もあり、今までちゃんと書き込めていたものが、ある時から書き込めなくなる場合もあります。
このような場合は、試験を行えば行うほど寿命を縮めてしまうというジレンマもあり、何かと難しいところもあります。

以上のように、組み込みシステムにおいては、実験室で正しく動作するソフトウェアを作成しておけばよいわけではなく、ハードウェア屋さんと一緒になって、（保障している条件内の）さまざまな環境で製品を動作させる必要があることを、忘れないでください。

次回は組込みシステム編の最終回です。
繰り返しますが、組込みシステムの最大の特徴は、現実世界の変化に対応して、制限時間以内に結果を返すことです。
現実世界の変化を知るためには、各種センサーを使用します。そしてセンサーが情報を CPU に伝える場合には割り込みを使用することが一般的です。
RTOS にとっても、タスクから RTOS に制御を移すために、定期的なタイマー割り込みを使用します。
このように、割り込みおよび割り込み処理が、組み込みシステムにとって重要な役目を果たします。
最終回は、この重要な割り込みに関してまとめたいと思います。

組込みシステム編　第 5 回　おわり