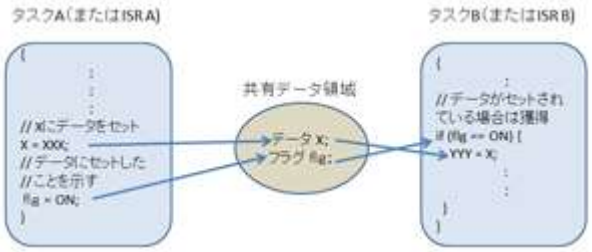
【組込みシステム編】第 2 回 CPU に不具合なし？

こんにちは、南角です。

今回は、前回(第 1 回　パイプラインストール)の課題を解説します。



上のようなケースで、動作がおかしくなる場合があるが、その原因を考えてくださいという課題でした。
おかしくなるのはタスク B でデータがセットされていないにも関わらず、データを読みに行く場合があり、さらに調べてみるとデータがセットされていないにも関わらず、フラグがセットされていました。
実は前回あえて書かなかったのですが、命令の流れが変わる命令には、ジャンプ命令、サブルーチン呼び出し命令のほかにも、サブルーチンや割り込みからの復帰命令があります。
サブルーチンから、呼び出し元に復帰する場合、スタックや専用のレジスタに保存している再開アドレスを、PC や IP と呼ばれる次に実行する命令のアドレスを保持するレジスタに格納する必要がありますが、これも命令実行の通常の流れを乱す命令です。
今回のケースでは、フラグを ON する命令の直後が関数からの復帰命令だったため、コンパイラの最適化処理により、そのまた前にあったデータセットをする命令が、関数からの復帰命令の後に再配置されたために発生しました。
つまりデータをセット→フラグをセットのはずが、コンパイラの最適化処理のため、フラグをセット→データをセットの順番になってしまい、その間でタスク B に処理が切り替わったため、タスク B はフラグがセットされているので、データもセットされていると思い、データを読み込んだが、まだデータはセットされていなかったことが、不具合の原因でした。
当時はまだ volatile も知らず、ずっと CISC を使っていたため、命令の再配置も知らなかったため、最初は戸惑いましたが、テストして状況から判断するとそれしかありえなかったため、実機にローディングした機械語オブジェクトを逆アセンブルして原因がわかりました。
これは C 言語をアセンブルした、機械語ニーモニックを見てもわかりません。
命令の再配置は C 言語がアセンブラに変換された後のアセンブラから機械語オブジェクトへの変換時に実施されるからです。つまり、この命令の再配置は、命令をアセンブラで記述しても発生します。

そのため最終的に生成された機械語オブジェクトを見ないと原因はわかりませんでした。
原因が判明した後は、対策を立てるだけでしたが、これは容易でした。
関数を超えての再配置はしないことは、すぐに確認できたので、flg を ON する処理を flg のアドレスを与えて処理を行う関数に変更して修正しました。
あるいは、バリア一命令ではないのですが、データをセットする処理とフラグをセットする処理の間に関数を置けば、その関数の前後の処理が入れ替わることはないこともすぐに確認できたので、それでもよかったと思います。この関数は特に何らかの処理をしなくても大丈夫でした。
もちろん今ならそんなことをしなくても、変数 x、flg に volatile 宣言を追加するだけで済ませます。
ちなみにアセンブラによっても異なりますが、アセンブラで再配置を禁止するには、アセンブラ疑似命令で例えば.noreorder とすれば同じ効果が得られます。
とにかく、C などで書いた命令が、必ずしも書いた順番に実行されない場合があるということには、注意してください。

今回、コラムが多くなったので本文は解説だけにしました。
今回のタイトルの意味はコラムを参照してください。来月は処理が何も存在しないときの、RTOS のスケジューラについて考えてみたいと思います。

組込みシステム編　第 2 回おわり

コラム１　-　パイプラインと割り込み

命令の流れが変わる命令の中には、call や jsr などのサブルーチン呼び出し命令があると書きました。
それではある意味、現実世界と組み込みソフトウェアをつなぐ存在でもある外部割込みはどうでしょう？
まず割り込み自体もある意味、関数呼び出しです。
ソフトウェア割り込みに関しては、もちろんソフトウェアによる、サブルーチン（割り込みハンドラ、ISR)の呼び出しですので、本文に書いたコンパイラの最適化の対象ですが、ハードウェアからのサブルーチン呼び出しといえる、外部割込みはどうでしょうか？
外部割込みはソフトウェアの実行とは関係ないため、その発生するタイミングはソフトウェアの実行とは全く関係ありません。
別の言い方をすると、ソフトウェアのどの部分が実行されているタイミングで割り込みが発生するかわからないので、コンパイラはどうしようもない気がしますが、どうなのでしょうか？
しかし、問題なのは関数呼び出し命令の、あとの命令が実行されるのはまずいのですが、この場合は関数呼び出し命令そのものが存在しないため、実行されるとまずい命令そのものが存在しないため問題はないと思います。
ただ厳密にいうと、外部割込み発生時、発生時に実行していた命令の実行終了後に、ISR に制御が移るのではなく、割り込み発生時に実行していた命令とそれに続く数命令を実行した後に、ISR に制御が移るといわなければならないのかもしれませんね。
このあたりに興味がある方は、自分の使用している CPU のハードウェアマニュアルを眺めてみてください。
何か新しい発見があるかも知れません。

コラム２　-　CPU に不具合なし

直前のコラムに書いたように、割り込み発生時の CPU の動きは興味深いものがあります。
CPU は状態マシン FSM であり、さまざまな状態を持っており、割り込み発生時も割り込みという状態に切り替わるのが普通です。
そして CPU も割り込み時に固有の動作をします。キャッシュミスの時も同じです。
昔使用した CPU は、キャッシュミス時にキャッシュラインを満たすために、バーストモードに移行しますが、その最中に割り込みが発生して割り込みハンドラに制御が移り、その割り込みハンドラ（ISR）の中である命令を実行すると、CPU がダウンしました。
もちろん、マニュアルにはそんなことは全く書いて無く、それを突き止めるのには大変苦労しました。
原因を突き止め、その CPU メーカーに問い合わせと CPU の修正依頼をしたのですが、最終的には制約事項、つまりその CPU 仕様になりました。そうなると当然ソフトウェアで対処しなければなりません。
また、別の CPU でクロックを落とした省電力モード中にある命令を実行すると、CPU がダウンする場合もありました。
現在では CPU も様々な機能が追加され、今やハードウェア屋さんも、すべてのケースの試験を行うことは難しくなっています。しかも CPU に何らかの不具合があったとしても、簡単には直せないのが普通だと思います。
製品開発時、製品が何か不具合を起こして、原因がソフトウェアっぽいと、まずアプリケーションソフトウェア屋さんが調べ、原因が不明の場合は OS 屋さんにお鉢が回って来ます。
その当時私は OS 屋さんを率いていたので、私のチームがソフトウェアの最後の砦だったわけです。
OS 屋が、不具合の原因がソフトウェアなのかハードウェアなのかの切り分けも行わねばなりませんでした。
よくハードウェア屋さんのチームとは協力して、時には議論（喧嘩）しながら、不具合の原因を調べたものです。
しかし原因がソフトウェアでもなく、ハードウェアでもない場合、 CPU の不具合である場合もかなりありました。製品の性格上、最新の CPU を使う機会が多かったせいかもしれませんが。
みなさんも、もし何らかの不具合に悩まされていて、その原因が不明（ソフトウェアでもハードウェアでもない）の場合は、CPU 自体の不具合かもしれないということは心の片隅にでも置いておいてください。
ソフトウェアでも不具合は何らかの境界で発生することが多いため、境界条件での試験を多く行うと思います。
CPU でも一緒なのではないかと想像しています。例えば、今回のコラムに書いた、キャッシュミスをして、CPU がバーストモードに移行して、キャッシュラインを埋めようとした時に、外部割込みが発生した場合などがまさしくそれで、さまざまな条件の組み合わせ時における、CPU の検証を行うエンジニアの方も大変だろうと思います。
このコラムは昔の CPU 屋さんへの恨みを思い出して、過剰な表現になっているかもしれません。CPU 屋さんの苦労もわかります。CPU 屋さんごめんなさい。

コラム３　-　オブジェクト指向とキャッシュミス

実は前のコラムには、もう一つの展開があります。
はっきりと検証したわけではなく状況証拠しかないため、今回はここまででとどめておきますが、実は個人的

には、ある状況証拠から、少なくとも当時のコンパイラのレベルでは、同じ機能を従来方式の機能ベースの C で作成した場合と、オブジェクト指向ベースで C++で作成した場合を比較すると、 C++で作成したほうが、従来方式よりもキャッシュミスが約 800 倍発生したのではないかという疑いを持っています。
これ以上は、興味を持った方がおられれば、調べていただければ幸いです。
その時は連絡いただければ、その時の状況証拠もお話しします。