アナログ・デバイセズの DSP、プロセッサ、開発ツール用テクニカル・ノート
http://www.analog.com/jp/ee-notes、http://www.analog.com/jp/processorsにはさまざまな情報を掲載しています。

VisualDSP++®ツールを使用した Blackfin®プロセッサのトラブルシューティングのこつ

著者: Jorge Manguane

Rev 1 – 2006 年 12 月 11 日

はじめに

このドキュメントでは、Blackfin®プロセッサと VisualDSP++®開発ツールのデバッグ機能について説明します。プログラマは、以下に説明する内容を応用して、遭遇した問題の原因を狭めた後に、アナログ・デバイセズの組込型プロセッサ・サポート・チームに報告することができます。これは、迅速な問題解決に役立ちます。

デバッグのこつ

次の内容を説明します。

- エミュレータを使ったアプリケーションの実行とアプリケーションのブート。特に、SDRAM 初期化時の考慮事項の説明。
- デュアル・コア・プロセッサのコア B のアンロック。エミュレータはこれを自動的に行いますが、コードをブートするとき、コア B を手動でアンロックする必要があります。
- ハードウェア・エラーとソフトウェア例外。
- Blackfin プロセッサのデバッグ機能とツール
  - トレース・バッファ
  - ブレークポイント(ソフトウェア、組込、ハードウェア)
  - VDK デバッグ機能(VDK Status ウインドウ、VDK State History ウインドウ)
- キャッシュをイネーブルしたときのデバッグ問題
- 割り込み

この EE ノートでは、トラブルシューティング・ペリフェラルに関する問題はカバーしていません。

エミュレーション対スタンドアロン・ブート

エミュレータ・ソフトウェアは、`.xml` ファイルを使って EZ-KIT Lite®評価ボード上の SDRAM タイミングなどのリソースを設定します。評価プラットフォーム(すなわち EZ-KIT Lite ボード)が存在するすべての Blackfin プロセッサに対してデフォルトの`.xml` ファイルがあります。このファイルでは、エミュレータが接続されたとき所定のレジスタを初期化する定義が生成されます。

たとえば、次の例は ADSP-BF537 Blackfin プロセッサの `ADSP-BF537-proc.xml` ファイルからの抜粋です。

```
<register-reset-definitions>
<register name="EBIU_SDRRC" reset-
value="0x03A0" core="Common" />
<register name="EBIU_SDBCTL" reset-
value="0x25" core="Common" />
<register name="EBIU_SDGCTL" reset-
value="0x0091998d" core="Common" />
<register name="EBIU_AMGCTL" reset-
value="0xff" core="Common" />
</register-reset-definitions>
```

したがって、ADSP-BF537 EZ-KIT Lite ボードのアプリケーションを開発するときは、エミュレータ・ソフトウェアが起動されると、SDRAM が自動的に初期化されます。

ただし、スタンドアロン・アプリケーションになると(すなわち、エミュレータを使ってダウンロードするのではなくアプリケーションをブートさせます)、システム内で SDRAM メモリを使う場合、ユーザの責任で SDRAM コントローラをイネーブルする必要があります。これは、ローダ・ファイルを生成する際に `Project Options` ダイアログ・ボックス・ページのローダ・ページを使って初期化ファイルをインクルードすることにより行われます。「*ADSP-BF533*

Blackfin Booting Process (EE-240)[1]」を参照してください。

エミュレータ・セッションからスタンドアロン・ブートへアプリケーションを移行させる際に問題を生ずる原因になるもう 1 つの違いは、ADSP-BF561 Blackfin デュアル・コア・システムに関係しています。デフォルトとして、エミュレータ・ソフトウェアがコア B を“アンロック”するため、コア B は L1 命令メモリの先頭から実行できるようになります。両コアを使用する場合、コア B は、システム・リセット・コンフィギュレーション・レジスタ(SICA_SYSCR)のビット 5 をクリアすることにより、コア A によってアンロックされる必要があります。

エミュレータがコア B をアンロックすることにより発生するもう 1 つの問題は、動作モードまたはクロック周波数の変更に関する問題です。特に、PLL または電圧レギュレータを変更するときは、コア B はアイドル状態(単にブレーク・ポイントではなく)にある必要があります。たとえば、ブレーク・ポイントがコア B に設定されていて、かつ PLL 周波数を変更するコードがコア A で実行されている場合には、問題が発生します。PLL 周波数を変更する前に、コア B をアイドル状態にするコードを実行するように注意してください。これは、サプリメンタル割り込みまたは GPIO ピンを使って行うことができます。

ハードウェア・エラーとソフトウェア例外

ハードウェア・エラーとソフトウェア例外は、Blackfin プロセッサで起こる 2 つの特別なタイプのイベントです。これらの各イベントは、イベント・ベクタ・テーブル(EVT)に別々のエントリを持っています。このドキュメントの後半で説明するブレーク・ポイント方法を使ってトラップできるように、これらの各イベントに対してハンドラをインストールしておく必要があります。その時点で、プロセッサの状態を調べて何によってそのイベントが発生したかを知ることができます。

シーケンサ・ステータス・レジスタ(SEQSTAT)には、その違反状態をさらに詳しく知るときに使うことができる 2 つのフィールドがあります。HWERRCAUSE フィールドはハードウェア・エラーが発生したときの状態を識別するときに使い、EXCAUSE フィールドは例外が発生したときの状態を識別するときに使います。

ハードウェア・エラーは、MMR を不正なワード・サイズでアクセスしたとき(たとえば、16 ビット MMR を 32 ビットとしてアクセス、または逆)や、またはコアまたは DMA コントローラが予約済みまたは未初期化メモリ・スペースをアクセスしようとしたときなどのように、さまざまな理由から発生します。RETI アドレスに、違反ロケーションの 10 ロケーションまでのアドレスが格納されます。ハードウェア・エラーがイネーブルされてイベントが処理されると、状態はクリアされますが、ハードウェア・エラー原因は直前のエラー状態のままになります。

ADSP-BF561 Blackfin デュアル・コア・プロセッサの場合、特定のコアで発生したハードウェア・エラーは、そのコアでのエラーのみ発生します。DMA コントローラがハードウェア・エラーを発生すると、エラーは両コアに送られます。

各ハンドラ(ハードウェア・エラーまたは例外)は、HWERRCAUSE フィールドと EXCAUSE フィールドを読み出して、イベントの原因を特定することができます。あるいは、エミュレータを使ってデバッグするとき、emuxcept のようなハンドラ内にトラップ命令を配置して、ハードウェア・エラーおよび/または例外が発生したときプロセッサが実行を停止できるようにすることができます。次に SEQSTAT レジスタ内の該当するフィールドを調べて、イベントの原因を知ることができます。

イベントの原因を知ることができるので、違反命令のアドレスを調べてプログラム内でいつ問題が発生したかを知ることができます。例外の場合は、例外レジスタ(RETX)からの戻り値に、“違反”命令のアドレスまたは次に実行される命令のアドレスが返されます。RETX内のアドレスは、例外タイプ(サービス(S)またはエラー(E))に依存します。例外を発生させるイベントとそのタイプ(サービスまたはエラー)は、「*ADSP-BF53x/BF56x Blackfin Processor Programming Reference*[2]」に記載されています。便利のために、アペンディックスAにもこの表を記載してあります。

エラー・タイプの例外の場合、RETX は違反命令のアドレスを保持します。サービス・タイプの例外の場合、RETX は違反命令の次の命令のアドレスを保持します。

この時点で、違反命令を調べて問題のさらに詳しい原因を探すことができます。命令が有効な CPLB 定義を持たないメモリをアクセスしていないか？命令が不正なロケーションに対してメモリ・ロード/ストアを実行していないか？ポインタまたはインデックス・レジスタが無効なメモリ領域を指していないか？ を調べることができます。

ブレーク・ポイントをハードウェア・エラーまたは例外を発生させた命令の周辺に設定して、アドレス・レジスタ(Ix または Px)を監視しながらコードを 1 ステップずつ実行することができます。ブレーク・ポイントの設定および/または問題の命令の前での 1 ステップずつのコード実行により、問題の状況が変化することがあります(すなわち、これらの状態で問題が発生しなくなってしまうことがあります)。このようなケースでは、問題の命令の後ろにブレーク・ポイントを設定して、ブレーク・ポイントに遭遇したときのプロセッサの状態を調べることができます。プロセッサはイベント・ハンドラへ分岐するため、ブレーク・ポイントはイベント・ハンドラ(例外ハンドラまたはハードウェア・エラー・ハンドラ)の先頭命令に配置されるようになることに注意してください。

トレース・バッファの使用

Blackfin プロセッサ上にある 16 スロット・トレース・バッファを使うと、直前の 16 個のフロー不連続変化(ゼロ・オーバーヘッド・ハードウェア・ループは除く)を記録することができます。トレース・バッファ内の情報は、問題の原因を調べる際に、または、さらに重要なことですが、予期しない動作を安定して示す小さいテスト・ケースを導出することにより問題の範囲を狭める際に役立ちます。前のセクションでは、特定のイベントを発生させる命令を特定する方法を説明しましたが、多くの場合、特定された同じ命令が問題を示さないことがあります。命令がフェッチされ実行される前に発生する場合は(実行ステージまで進まないケースもあります)、根本原因をゼロにしようとする努力にとって致命的です。

たとえば、P2 レジスタを使ってメモリ・ロードを実行する命令のケースを考えます。この命令の実行の直前に、割り込みが受理されものとします。この割り込み処理では、プログラミングとしては悪い例ですが、使用しているレジスタを待避/復旧していないとします。ISR コードはポインタ・レジスタ P2 を変更し、割り込みサービス・ルーチンからリターンした後に、元のメモリ・ロード命令が実行されます。このとき、P2 は非同期イベント時に上書きされているため、P2 は所望のメモリ・ロケーションを指さなくなっています。このために前述のいずれかのイベントが発生してしまいます。さらに紛らわしいことに、データが誤ったメモリに対して読み書きされてしまいます。後者は一般に検出が困難です。

トレース・バッファを使うと、問題発生前に生じたフロー変化を容易にウインドウ内に表示することができます。不連続について直前の 16 対を記録しています。対内の先頭エントリが不連続(call 命令)のソースで、2 番目のエントリが不連続のディステネーションすなわちターゲット(呼び出される関数の先頭命令)です。上記 P2 の例では、与えられたトレース対の先頭命令は割り込みからのリターン命令(RTI)で、対の 2 番目のエントリはロード命令またはその前の命令になります。トレース・バッファは不連続のアドレスも表示するため、ISR 内で RTI 命令のアドレスを調べて、P2 が変更されかつ ISR が終了する前に復旧されていなかったことを見つけることができます。この ISR は、アプリケーションが使用する RTOS のスケジューラの一部として使用することができます。もちろん、ここで説明した例は非常に単純化されたものです。これは、ISR が既知の問題に対する対策を行っていなかったことを意味しています。

何も明確にならないときもあります(すなわち、この解析をすべて行っても問題発生の理由を説明できない場合)。問題に遭遇する前に発生した変化を知ると、小さなテスト・ケースの作成に役立ちます。このテスト・ケースは、サポート・チームが迅速に問題の調査と解決を行う際に非常に役立ちます。

図 1 に、トレース・バッファ内のエントリの構造を示します。最も左の列は 0～31 のサイクルを表しています。サイクル 0 と 1 はトレース・バッファ内に記録された不連続の最後の対で、サイクル 2 と 3 は最後から 2 番目の対で、以下同様です。左から 2 つ目の列は対のグルーピングを表しています。たとえば、サイクル 0 と 1 は 15 番目の対(0xf)に、サイクル 2 と 3 は 14 番目の対(0xe)に、サイクル 0x1eと 0x1fは 0 番目の対(0x0)に、それぞれなります。対の先頭命令は不連続のソースで、2 番目の命令は不連続のディステネーションです。0xf対の場合、サイクル 0 がソース・アドレス(RTS命令)で、サイクル 1 がディステネーション命令(CALL Initialize__3VDKFv)になります。すなわち、この命令がアドレス 0xffa086beで終了するサブルーチンからリターンした後に最初に実行されます。

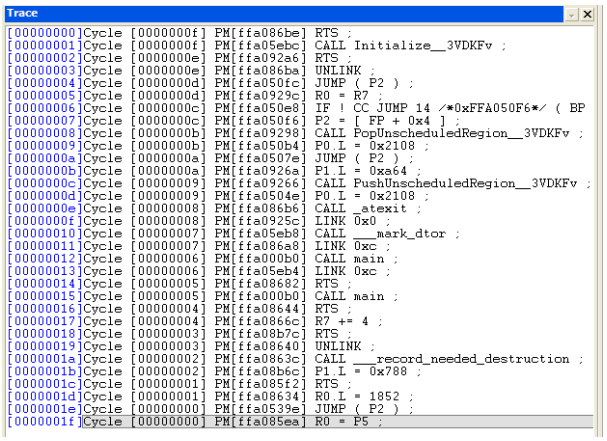

図1.トレース・バッファの例

ブレークポイントの使用

このセクションでは、ソフトウェア、組込、ハードウェアの各ブレーク・ポイントの違いを説明し、その使い方を説明します。

ソフトウェア・ブレークポイント

ソフトウェア・ブレーク・ポイントは便利で使い易くなっています。エディタ(ソース)ウインドウまたはIDDEのDisassemblyウインドウ内で命令をダブルクリックするだけで、ブレーク・ポイントを設定し、コードのラインが一致したとき実行を停止させることができます。ただし、この背景では、ブレーク・ポイントが設定されたロケーションの値はエミュレータ内部に"キャッシュ"されています。

エミュレータはブレーク・ポイント・ロケーションでメモリを読み出し、それをエミュレータの内部ブレーク・ポイント・リストに保存します。アプリケーションが実行されると、そのロケーションにトラップ命令を配置します。ブレーク・ポイントが一致すると、または停止イベントが発生すると、ブレーク・ポイント・ロケーションにあるトラップ命令が前に"キャッシュ"された命令で置き換えられます。明らかに、この動作からソフトウェア・ブレーク・ポイントは侵害的であることが分かります。このため、ソフトウェア・ブレーク・ポイントを使って問題を診断しようとすると、ソフトウェア・ブレーク・ポイントによってアプリケーションのタイミングが変化してしまうので、遭遇する多くの問題は逃げて行ってしまうように見えます。図 2に、VisualDSP++ IDDEセッションでソフトウェア・ブレーク・ポイントがどのように見えるかを示します。

```
//-------------------------------
void main(void)
{
    Init_Flags();
    Init_Timers();
    Init_Interrupts();

    while(1);
}
```

図2.ソフトウェア・ブレークポイントの例

組込ブレークポイント

組込ブレーク・ポイントは、アプリケーション・コード自体の一部です。ソフトウェア・ブレーク・ポイントと似ていますが、デバッガがブレーク・ポイント・テーブル・リストを参照する必要がない点、または"停止"オペコードをアプリケーションへ挿入する必要がない点が異なっています。このため、このクラスのブレーク・ポイントは準非侵害的です。emuexcpt 命令は実行されるとプロセッサを停止させます。この命令はエミュレータが接続された場合にのみ意味があります。その他の場合は、NOP として機能します。イベント・ハンドラ内部で組込ブレーク・ポイントを使用することは望ましいことです。

これによって、コード・スペースを使う以外は、アプリケーションのタイミングに影響を与えないで、イベントが発生した直後にプロセッサの状態を確実に観測できるためです。トレース・バッファ情報と組み合わせて使用すると、プロセッサの状態を観測することができるようになり、イベント発生直前の変化も観測で

きるようになります。図 3に、組込ブレーク・ポイントの例を示します。

```
EX_INTERRUPT_HANDLER(Timer0_ISR)
{
    asm("EMUEXCPT;");  // embedded breakpoint

    // confirm interrupt handling
    *pTIMER_STATUS = 0x0001;
    brkptcounter++; // hw breakpoint counter
    // shift old LED pattern by one
    if(sLight_Move_Direction)
    {
        if((ucActive_LED = ucActive_LED >> 1) <= 0x0020) ucActive_LED = 0x1000;
    }
    else
    {
        if((ucActive_LED = ucActive_LED << 1) == 0x1000) ucActive_LED = 0x0020;
    }

    // write new LED pattern to PORTF LEDs
    *pPORTFIO_TOGGLE = ucActive_LED;

}
```

図3.組込ブレークポイントの例

ハードウェア・ブレークポイント

一方、ハードウェア・ブレーク・ポイントはアプリケーション・コードを変更することがないため完全に非侵害的です。その代わり、ハードウェア・ブレーク・ポイントではチップ上の物理的なハードウェア・ロジックを使用します。このロジックが、命令バスとデータ・バスを監視します。Blackfin プロセッサでは、ハードウェア・ブレーク・ポイントはウォッチポイント・レジスタ・ユニットにより実現されています。6 個の命令ウォッチポイント・レジスタと 2 個のデータウォッチポイント・レジスタがあります。命令ハードウェア・ブレーク・ポイントは、6 個の特定の命令アドレスまたは 3 個の命令アドレス範囲に設定することができます。データ・ハードウェア・ブレーク・ポイントは 2 個の特定データ・アドレスまたは 1 個のデータ・アドレス範囲に設定することができます。ハードウェア・ブレーク・ポイントは、RAM または ROM タイプのメモリで使用することができます。

VisualDSP++ IDDE内からハードウェア・ブレーク・ポイントをイネーブルするときは、`Settings`に行き、`Hardware Breakpoints`を選択します。図 4に、`Hardware Breakpoints`ウインドウの`Instruction`ページの 1 つを示します。

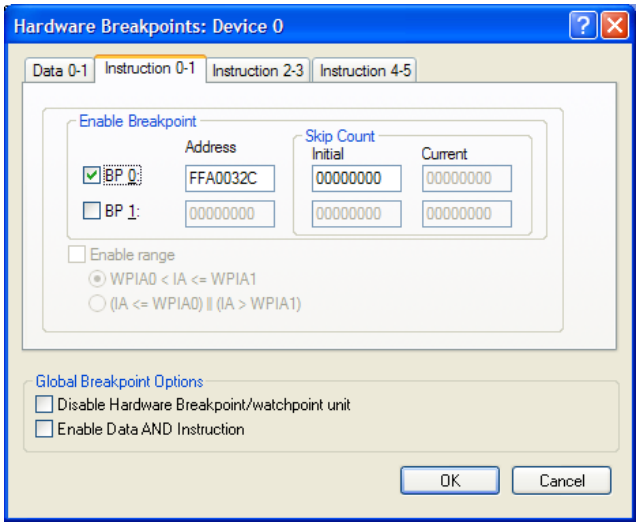

図4.ハードウェア・ブレークポイント(命令)

次に、これらの命令が実行されようとする時に、プロセッサを停止させる命令アドレスまたはアドレス範囲を指定することができます。

データ・アクセスの場合、アクセスのタイプを指定し(読み出し、書き込み、または両方)、エミュレーション停止をトリガーする必要があります。図 5に、`Hardware Breakpoints`ウインドウの`Data`ページを示します。

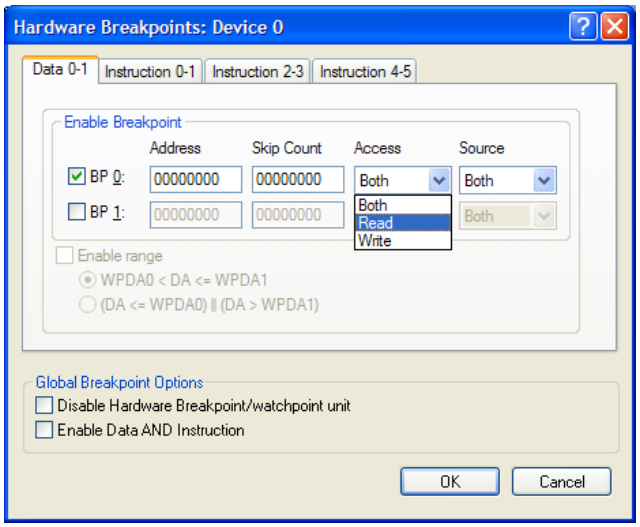

図5.ハードウェア・ブレークポイント(データ)

次にコードを実行すると、内部命令/データ・アドレス・バスとハードウェア・ブレーク・ポイント・レジスタで指定されたアドレスが一致したとき、プロセッサが停止します。

ハードウェア・ブレーク・ポイントはスキップ・カウント機能を提供します。この機能は、プロセッサが停止するまでに、指定された領域に対して行う無視すべきアクセス回数を指定するときに使います。たとえば、スキップ・カウントを 0xA に設定すると、プロセッサは 10 番目のアドレス一致で停止します。

VisualDSP++カーネル(VDK)

VDK は、複数のタスクを持つプロジェクトの管理を簡素化するリアルタイム・カーネルですが、アプリケーションにある程度の抽象化を追加します。このため、すべての RTOS と同様に、システム内のバクをピンポイントで見つけることが困難になります。

VisualDSP++はシステム性能を詳しく表示できるカーネルを意識したデバッガを持っているため、アプリケーションのチューニングとRTOS採用システムのデバッグに役立ちます。この機能を使うと、種々のスレッドを何時でも表示することができます(実行時、停止時、レディ時など)。多くのデバッグ・ニーズのなかで、この機能は特定のスレッドが実行されない理由を特定するときに役立ちます。 図 6に、VDK State Historyウインドウを示します。

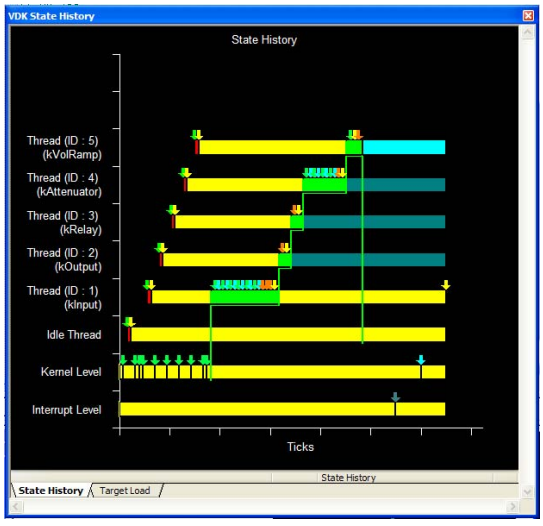

図6.VDK State History ウインドウ

スレッド優先順位を正しく設定してください。各タスクのニーズを実行時間で把握しておく必要があります。VDK State History ウインドウは全体のスレッド・タイム・バランスを求めるのに役立ちます。

もう 1 つの便利なデバッグ・ウインドウであるVDK Statusウインドウは、カーネル・パニック・エラーの原因を表示します。図 7に、VDK Statusウインドウを示します。

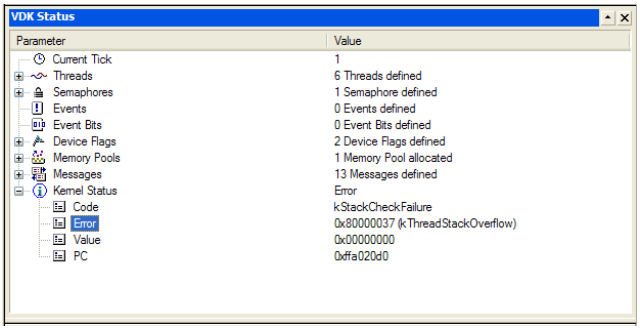

図7.VDK Status ウインドウ

図 7の例には、カーネル・パニックを発生させるスタック・オーバーフローが示してあります。表示された`Value`は、スタック・サイズが不足しているスレッド(IDLEスレッド)を示しています。

キャッシュ関連の問題

キャッシュ問題が疑われる場合は、先ず該当するプロセッサのアノーマリ・リストを調べて、観測される特定の動作がそこに記載されているか否かを確認してください。

予期しない動作が既知の問題に関係ないように見えるときは、注目の領域を L1 メモリへ移動することにより、キャッシュ・コントローラを原因として除外してみてください。前のセクションで、この領域の特定方法を説明しました。キャッシュをターンオンしてアプリケーションを実行し、次にキャッシュをターンオフして再度実行します。動作の違いを観測します。キャッシュをターンオフしても問題が消えない場合には、ソフトウェア・アプリケーションに競合状態があることを示しています。

キャッシュをターンオフすると、アプリケーションの残りの部分のタイミングが変化して、停止に至ることもあります。このため、注目の領域を L1 へ移動してキャッシュをターンオンのままにしてみてください。問題が消えずに、かつ L1 内にあるコードの領域で例外および/またはハードウェア・エラーが発生する場合には、キャッシュ・インテグリティの問題ではありません。

例外が発生する場合には、上の ハードウェア・エラーとソフトウェア例外のセクションを参照してください。

キャッシュのコヒーレンシ

Blackfin プロセッサは、キャッシュ・メモリとメイン・メモリとの間でコヒーレンシを維持していません。一般に、コヒーレンシはペリフェラル DMA チャンネルがキャッシュ可能と定義された外部メモリの領域をアクセスするシステムで問題になります。キャッシュ・コントローラはこれらのアクセスを認識できないため、古いデータを計算に使用して予期しない結果を発生させます。ソフトウェアは、DMA コントローラからアクセスされた可能性のあるラインを無効にすることにより、コヒーレンシを維持する必要があります。

割り込み関連の問題

ISR では、リソースのプッシュとポップを正しい順序で行う必要があります。また、RETI のプッシュとポップの重要性にも注意してください。RETI がスタックにプッシュされると、割り込みのネスティングがイネーブルされ、逆に RETI をポップすると、割り込みのネスティングがディスエーブルされます。したがって、高い優先順位の割り込みを割り込みサービス・ルーチンに割り込ませない場合は、RETI をスタックにプッシュしないでください。C/C++でのプログラミングでは、次の非ネストの割り込みハンドラを使ってください。

```
EX_INTERRUPT_HANDLER(Timer_handler)
```

特定の ISR に対して割り込みネスティングをイネーブルするときは、次の割り込みハンドラを使ってください。

```
EX_REENTRANT_HANDLER(Timer_handler)
```

このリエントラントなハンドラは、ISR の開始で RETI をプッシュし、RTI が実行される直前の終わりでポップします。

同じ ISR への分岐の繰り返しを防止するため、割り込み原因をクリアしてから ISR を終了してください。たとえば、コア・タイマの場合、コア・タイマ・コントロール・レジスタ内の TINT (タイマ割り込み)ビットをクリアすると、割り込みがクリアされます。

割り込みのネストを使用する場合は、共用リソースの使用から発生する問題を回避してください。ISR の実行時間を短くすると、低い優先順位の ISR もタイムリにサービスされます。ISR を短くすると ISR での使用リソース数を減らすことができるため、スタック使用率を軽減することができます。割り込みのネストで発生する別のタイプの問題は、スタック・オーバーフローです。ネストされた割り込みで(深くネストされたサブルーチンでも)、スタック・オーバーフローを検出する 1 つの方法は、各 ISR の開始でスタック・ポインタ(SP)を読み出して、ポインタがスタックの終わりに近いか否かをチェックすることです。

まとめ

この EE ノートでは、問題範囲を狭める際に使用できる VDK ツールと Blackfin プロセッサ機能について説明しました。

最初に、必ず使用するプロセッサのシリコン・レビジョンのアノーマリ・リストをチェックして、問題は既知であるか否かを確認してください。既知の場合には、指定された対策を適用してください。既知のシリコン・エラッタに対する自動ソフトウェア・サポートを得る為には、最新ツールを使用していることと、シリコン・ワークアラウンドがイネーブルされていることを確認してください。

メイン・アプリケーションを実行する前に、アプリケーションにイベント・ハンドラ(例外ハンドラと割り込みハンドラ)をインストールして、必要な場合にイベントをトラップできるようにする必要があります。

正常に動作しないのは何か？ 例外/ハードウェア・エラーが発生しているか？ 発生しているなら、何の例外および/またはハードウェア・エラーか？ などの動作を確認します。ペリフェラルがオーバーフロー/アンダーフローしているか？ DMA エラーが発生しているか？などを調べるときは、アペンディックス A の表が役立ちます。

再現性を良くする方法を見つけます。常に可能とはかぎりませんが、問題が発生する頻度を上げると問題を解決するチャンスが大きくなります。再現性が良くなると、ループの繰り返しを長くまたは短くすること、コア電圧を変更すること、コアおよび/またはシステム周波数を調整することなどが意味を持つようになります。1 回に 1 つの変数を変化させることに注意してください。変数を変えてもバグに影響がない場合には、変数を新しい変更の前の状態に戻します。

ソフトウェア・ブレーク・ポイントを使って、問題が発生する前のプロセッサの状態を観測します。ソフトウェア・ブレーク・ポイントを挿入したとき、停止に至る場合は、組込ブレーク・ポイントまたは、最終的

にはハードウェア・ブレーク・ポイントを使ってください。

ハードウェア・エラー/例外が発生する場合は、シーケンサのステータス・レジスタからそれぞれの原因を見つけて、アペンディックス A の表をチェックして、これらのイベントが何から発生しているか調べます。

組込ブレーク・ポイントまたはハードウェア・ブレーク・ポイントを使って、それぞれの例外ハンドラでイベントをトラップしてください。

Trace ウインドウを使って、問題が発生する前のプロセッサの変化を観測します。

後の解析のために、VisualDSP++で`Register->Save Registers`を選択して、すべてのレジスタを保存します(図 8参照)。

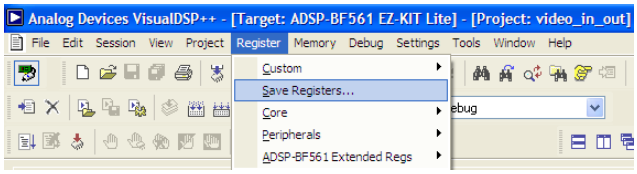

図8.レジスタ保存機能

上に説明するステップの後でも、問題を直すことができない場合には、予期しない動作を発生させるイベントのシーケンスを知ると、小さいテスト・ケースを発生させることができます。テスト・ケースが得られたら、それを使った結果をテスト・ケースも含めて組込型プロセッサ・サポート・チーム用にまとめてください。これにより迅速な問題の再現が可能になるので、最終的な問題の解決に役立ちます。

アペンディックスA

表1.例外を発生させるイベント

Exception	EXCAUSE [5:0]	Type: Error (E) Service (S)	Notes/Examples
Force Exception instruction EXCPT with 4-bit m field	m field	S	Instruction provides 4 bits of `EXCAUSE`.
Single step	0x10	S	When the processor is in single step mode, every instruction generates an exception. Primarily used for debugging.
Exception caused by a trace buffer full condition	0x11	S	The processor takes this exception when the trace buffer overflows (only when enabled by the Trace Unit Control register).
Undefined instruction	0x21	E	May be used to emulate instructions that are not defined for a particular processor implementation.
Illegal instruction combination	0x22	E	See section for multi-issue rules in the *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*.
Data access CPLB protection violation	0x23	E	Attempted read or write to Supervisor resource, or illegal data memory access. Supervisor resources are registers and instructions that are reserved for Supervisor use: Supervisor only registers, all MMRs, and Supervisor only instructions. (A simultaneous, dual access to two MMRs using the data address generators generates this type of exception.) In addition, this entry is used to signal a protection violation caused by disallowed memory access, and it is defined by the Memory Management Unit (MMU) cacheability protection lookaside buffer (CPLB).
Data access mis-aligned address violation	0x24	E	Attempted misaligned data memory or data cache access.
Unrecoverable event	0x25	E	For example, an exception generated while processing a previous exception.
Data access CPLB miss	0x26	E	Used by the MMU to signal a CPLB miss on a data access.
Data access multiple CPLB hits	0x27	E	More than one CPLB entry matches data fetch address.
Exception caused by an emulation watch-point match	0x28	E	There is a watchpoint match, and one of the `EMUSW` bits in the Watchpoint Instruction Address Control (`WPIACTL`) register is set.
Instruction fetch misaligned address violation	0x2A	E	Attempted misaligned instruction cache fetch. On a misaligned instruction fetch exception, the return address provided in `RETX` is the destination address which is misaligned, rather than the address of the offending instruction. For example, if an indirect branch to a misaligned address held in P0 is attempted, the return address in `RETX` is equal to P0, rather than to the address of the branch instruction. (Note this exception can never be generated from PC-relative branches, only from indirect branches.)

Instruction fetch CPLB protection violation	0x2B	E	Illegal instruction fetch access (memory protection violation).
Instruction fetch CPLB miss	0x2C	E	CPLB miss on an instruction fetch.
Instruction fetch multiple CPLB hits	0x2D	E	More than one CPLB entry matches instruction fetch address.
Illegal use of supervisor resource	0x2E	E	Attempted to use a Supervisor register or instruction from User mode. Supervisor resources are registers and instructions that are reserved for Supervisor use: Supervisor only registers, all MMRs, and Supervisor only instructions.

表2.ハードウェア・エラー割り込みを発生させるハードウェア状態

Hardware Condition	HWERRCAUSE(Hexadecimal)	Notes / Examples
System MMR Error	0x02	An error can occur if an invalid System MMR location is accessed, if a 32-bit register is accessed with a 16-bit instruction, or if a 16-bit register is accessed with a 32-bit instruction.
External Memory Addressing Error	0x03	
Performance Monitor Overflow	0x12	
RAISE 5 instruction	0x18	Software issued a `RAISE 5` instruction to invoke the Hardware Error Interrupt (`IVHW`).
Reserved	All other values.	

参考

[1] *ADSP-BF533 Blackfin Booting Process (EE-240).* Rev 3. January 2005. Analog Devices, Inc.

[2] *ADSP-BF53x/ADSP-BF56x Programming Reference.* Rev 1. May 2005. Analog Devices, Inc.

ドキュメント改訂履歴

Revision	Description
Rev 1 – December 11, 2006 by J. Manguane	Initial Release.