WP473 (v1.0.1) 2016 年 4 月 11 日

64 ビット ARM ヘテロジニアス プラットフォームへの ソフトウェアの移行

Zynq UltraScale+ MPSoC は、既存のソフトウェア インフラへの投資を無駄にすることなく最新技術を取り入れることができる画期的なプラットフォームを提供します。

概要

Zynq® UltraScale+™ MPSoC で採用されている ARM®v8 アーキテクチャは、最小限の書き換えで既存の ARMv7 コードを素早く実行可能にします。この優れたアーキテクチャ互換によって、設計者の生産性が向上し、最小限の開発コストとエンジニアリング投資で製品をいち早く市場投入できます。

はじめに

次世代 ARMv8 アーキテクチャをベースとするザイリンクス Zynq UltraScale+ MPSoC を利用することで、既存のソフトウェアへの投資を無駄にすることなく、常に最先端技術を取り入れることができます。

米国のコンピューター学会 (ACM) が発行する月刊会誌『Communications of the ACM』([参照 1]) の最近の記事で、Steve Furber 氏が Zynq UltraScale+ MPSoC のような 32 ビット ARMv7 SoC は史上最高の生産性をもたらす画期的な製品であると評価しています。この大規模なインストール ベースの ARMv7 SoC は、シンプルなライブラリやツールから完全な OS に至るまで、より大規模なインストール ベース ソフトウェアとしての役割を果たします。近年、64 ビット プラットフォームへの移行が加速する中で、システム設計者は既存のソフトウェア インフラを新しいアーキテクチャへ移行するという大変な作業を行うべきか、新しいプラットフォームでゼロから構築すべきかという選択に直面しています ([参照 2])。

あらゆる中規模ソフトウェア プロジェクトには、人件費とソフトウェア インフラの両方で莫大な投資が行われています。これらの投資を無駄にすることは、プロジェクトの短期的および長期的将来のいずれにおいても非常に大きな損失をもたらします。つまり、SoC プラットフォーム間でソフトウェアを移植することは、不確実性を生み出す原因となります。この移植プロセスで問題が生じると、機能と性能を最大限に引き出せなくなります。設計コストを抑えて迅速に製品を市場投入することが求められる現代では、不確実性に対応する余裕はありません。

ザイリンクスの Zynq ポートフォリオに追加された最新製品 Zynq UltraScale+ MPSoC は、この課題にデザインで対処する次世代 ARMv8 アーキテクチャをベースとしています。ARMv8 では、同じような CPU アーキテクチャとソフトウェア開発プロセスを使用できるため、既存のソフトウェア インフラへの投資損失を最小限に抑えた移植プロジェクトが可能です。

ARMv8 はデザインでソフトウェアの移植を容易にする

Zynq UltraScale+ MPSoC で採用されている ARMv8 アーキテクチャは、前世代の ARMv7 アーキテクチャと互換性があるため、ソフトウェア移植作業が非常にシンプルになる上、最新の技術および機能へ対応できる拡張性にも優れています。ソフトウェア開発者には使い慣れた開発環境が提供されるため、移植作業を開始すると同時にコードの記述に専念できます。ARMv8 アーキテクチャは、主に次の 2 つの互換性を提供します。

- ARMv8 アーキテクチャは、次の 2 つの実行ステートをサポートします。
  - o AArch32、ネイティブ 32 ビット ステート
  - o AArch64、ネイティブ 64 ビット ステート
- AArch64 ステートの 64 ビット ARMv8 命令セットは、ARMv7 の命令セットから拡張されたものであり、AArch32 ステートの 32 ビット命令セットは、ARMv7 の命令セットと完全互換です。

AArch64 実行ステートは、ネイティブな ARMv8 実行環境であり、ARMv8 アーキテクチャのすべての機能セットおよび技術をサポートします。一方、AArch32 実行ステートは、ARMv7 で採用されている環境とのネイティブな後方互換機能を提供します。ソフトウェアの実行という視点で考えると、このステートはネイティブな ARMv7 実行環境です。コンパイル時にコードが実行すべき実行ステートは、開発者が選択します。

通常の動作時、ソフトウェアで求められる実行環境に応じて CPU が AArch64 実行ステートと AArch32 実行ステートをオンザフライで切り替えます。これらのコンテキスト変更は例外の境界で実行され、ユーザーやソフトウェア設計者による入力は必要ありません。つまり、32 ビット コードと 64 ビット コードは効率的にサイドバイサイド実行されます。

AArch64 ステート動作の場合、フル 64 ビットの ARMv8 命令セットが提供されます。おおむね、64 ビットの ARMv8 命令セットは ARMv7 命令セットの拡張版です。命令セットの構文やビヘイビアーにわずかな変更が加えられ、32 ビット命令や混合 (32 ビットと 64 ビット) レジスタをサポートします。AArch32 ステート動作の場合、プロセッサは標準の 32 ビット命令と 32 ビット レジスタを使用します。つまり、ソフトウェア設計者は命令セットをほかの ARMv7 プロセッサと同じように扱うことができます。

コード ベースの活用

低レベルのベアメタルや RTOS コード開発者は、ベースとなるプロセッサ アーキテクチャの違いによる影響を最も高い頻度で受けます。ソフトウェアを移植する際のベスト プラクティスとして、コンパイラのすべての警告メッセージおよびエラー メッセージを有効に設定し、ARMv8 コンパイラを使用して修正を加えずに再コンパイルする方法があります。ソフトウェア開発者は、このプロセスでの警告またはエラーを解析することで、無視しても問題ない箇所と、移植時に注意すべき箇所を判断できます。

ARMv8 コンパイラは、C および C++ などの一般的な高レベル コードをネイティブ サポートします。つまり、適切な ARMv8 ボード サポート パッケージ (BSP) の準備が整った後に、これらのコードがコンパイルされて動作します。一方、アセンブリ コードはその使用方法に注意が必要です。ARMv7 の多くのアセンブリ命令が ARMv8 に存在していますが、それらの構文やビヘイビアーは微妙に異なる場合があります。ARMv7 と比べて予想どおりにコンパイルまたは動作しないコーディング コンストラクトとは、ハードコードされたメモリ ロケーション (すべてのソフトウェア プロジェクトに該当)、ARMv7 コプロセッサ (CP15 など) やレジスタへのアクセス、およびデータ アライメントなどです。移植時における注意事項の詳細は、ARM 社の『ARM® Cortex™-A Pogrammer's Guide for ARMv8-A』(DEN0024A) を参照してください。

アーキテクチャ間の相違点を概念化には、容易に移植できるコードと注意が必要なコードの例を参照してみると有効です。たとえば、ADD 命令は ARMv7 構文から ARMv8 構文への変換に最小限の作業のみ必要となる代表的な例です。ARMv7 のレジスタには **Rn** という命名規則があり、n はレジスタの番号を表します。ARMv7 ではすべてのレジスタが 32 ビットであるため、命名規則はシンプルです。一方、ARMv8 のレジスタは 32 ビットまたは 64 ビットのいずれかになります。32 ビット レジスタは **Wn** という命名規則に従い、64 ビット レジスタは **Xn** という命名規則に従います。このため、ソフトウェア設計者は、既存のコードを確認して、当該オペランドのサイズに応じて適切に変更する必要があります。表 1 を参照してください。

Table 1: ADD 命令の命名規則 (環境別)

ARMv7	ARMv8 A32	ARMv8 A64 (32 ビット)	ARMv8 A64 (64 ビット)
`ADD Rd、Rn`	`ADD Rd、Rn`	`ADD Wd、Wn`	`ADD Xd、Xn`

ADD のような移植命令は、コードの目的を確認してレジスタ名を必要に応じて変更するだけです。この移植プロセスは、コードベース全体の複雑レベルにもよりますが、自動化できます。

ARMv7 コプロセッサへのアクセスなどの状況では、より綿密なコード解析が必要になる場合があります。これらのコプロセッサはプロセッサ全体のシステム レベルの属性や機能を制御するため、解析がベスト プラクティスと考えられ、それの操作は一般に ARMv7 の動作です。

AArch64 ステートで動作している場合、ARMv8 アーキテクチャには CP15 レジスタが存在しないため、このレジスタへは直接アクセスできません。AArch32 ステートで動作している場合は、概念コプロセッサを介してこのレジスタへアクセスできます。これは、レガシ 32 ビット コードに対して一貫した実行環境を提供するために AArch32 実行ステートで使用される概念です。

たとえば、プログラムが Main ID レジスタのコンテンツを必要としている場合を想定します。このコンテンツは、慣例的にコプロセッサ CP15 の `c0` レジスタの一部となります。この値へアクセスするには、ARMv7 用の特殊な MRC 命令と ARMv8 の AArch32 実行ステートを使用します。

```
MRC p15, 0, r1, c0, c0, 0
```

AArch32 ステートでそのまま実行されるレガシ コードは修正が不要ですが、AArch64 ステートへ移植されたコードは、新しいアーキテクチャの要件を満たすように修正する必要があります。

ネイティブ AArch64 実行ステートの場合、Main ID レジスタの値は MIDR_EL*n* という新しい専用レジスタにあります。このレジスタへは、システム レジスタのアクセス命令 MRS を使用して簡単にアクセスできます。

```
MRS x1, MIDR_EL1
```

したがって、開発者は必要に応じてアセンブリ命令を変更できるように、このような特殊な操作および呼び出しを把握しておく必要があります。なお、これは周辺のソフトウェア コード全体に影響を与えないシングル ライン変更です。ソース レジスタとターゲット レジスタは、64 ビット アーキテクチャ要件に従って、両方ともネイティブの 64 ビット レジスタです。

既存のソフト ウェアを未修正で使用

Linux オペレーティング システムをベースとするソフトウェアの移植プロセスは、RTOS やベアメタル コード開発者に比べてさらにシンプルです。ベースとなる Linux オペレーティング システムは、内在するハードウェアの違いの多くを無視します。ほとんどの場合、64 ビット Linux コンパイラで再コンパイルするだけで、コードを 64 ビット Linux 上で実行できます。ザイリンクスの PetaLinux ツールでは、64 ビット Linux 環境用に当該コードをネイティブ統合して再コンパイルすることで、このプロセスを自動化できます。

別の場合では、開発者が 32 ビット バイナリ形式でコードを保持する必要があるかもしれません。ソフトウェアは、32 ビット形式でのみ有効な外部ライブラリにリンクされることがあります。あるいは、変換に時間を要する 32 ビットのアセンブリ コードを含むこともあります。このようなソフトウェア移植は、最新の Linux ディストリビューション (Multiarch または Multib) の機能を使用することで、大幅に時間を短縮できます。これらのターム (Multiarch/Multib) は厳密には同じではありませんが、ほとんど区別しないで使用されます。

Multiarch は、システムやファイル システムのレイアウトを設定する手段であり、32 ビットと 64 ビットの両方が動的にリンクされたプログラムを実行する機能を提供します。レガシ 32 ビット ARMv7 Linux プログラムを未修正のまま使用でき、新しい 64 ビット プログラムと共存させることが可能です。この機能を利用することで、特定の 32 ビット アプリケーションを移植して 64 ビット アーキテクチャに対して修正する一方で、システムを素早く起動して動作させることが可能になります。

Multiarch 対応のディストリビューションは、32 ビット ライブラリと 64 ビット ライブラリを同じ Linux ファイル システムに統合することで、この互換性を達成しています。64 ビット アプリケーションが実行されるときには 64 ビット ライブラリが使用され、32 ビット アプリケーションでは 32 ビット ライブラリが使用されます。

Multiarch オペレーティング システムを独力で作成して管理するにはかなりの時間が必要です。幸いにも、Yocto Project、Canonical、および Linaro のオープン ソース製品には、簡単に統合できるオプションがあります。Yocto Project は、完全なエンドツーエンドのビルド システムを提供し、Canonical と Linaro は、すぐに使用できるドロップインのリファレンス ファイル システムを提供しているため、開発者は迅速に開発を進めることができます。ザイリンクス アンサー 66636 では、ZCU102 評価プラットフォームにおける Ubuntu Core 14.04 ファイル システムの統合について詳しく説明しています。

これらの Multiarch ファイル システムは、デフォルトで 64 ビット アプリケーションのみを使用するように設定されています。これらは dpkg 管理ツールに基づいているため、シンプルなコマンド インターフェイスを用いてアーキテクチャを追加できます。

```
# dpkg -add-architecture armhf
```

32 ビット アーキテクチャ (例 : **armhf**) をサポートするように dpkg ツールを設定した後、Intel x86 互換バージョンの Linux 上と同じように apt-get などの高レベル パッケージ管理ツールを使用できます。

一部のシステムでは、Multiarch を使用した場合、64 ビット ホスト上で既存の 32 ビット アプリケーションを実行するだけで十分となり、その他の作業は必要ありません。それ以外のシステムでは、時間を節約するために Multiarch が使用されて、基盤となる 64 ビットのアーキテクチャおよびオペレーティング システム要件を満たすようにソフトウェア スタックのその他のコンポーネントが調整および最適化されます。

まとめ

ARMv8 は、性能、セキュリティ、アーキテクチャの点で新しい機能を数多く提供しますが、ソフトウェア開発者が既存のソフトウェア インフラを無駄にすることに不安を抱き、適用することに消極的になるのでは、これらの新機能の価値が失われます。ARMv8 は、ハードウェアとソフトウェア両方インターフェイスからプロセッサ アーキテクチャに適宜拡張性をもたらすことで、これらの不安を軽減します。低レベル コードの移植は、開発者が慣れ親しんだ命令セットを必要に応じて拡張させることで効率化されます。その上、AArch32 ステートへのフォールバック機能により、既存の 32 ビット コードを素早く実行できるため、最小限の時間と労力で新しいハードウェア上でデザインを評価できます。最後に、Multiarch Linux では、一部のコードを完全に無修正で実行できるため、全体的なシステム統合を大幅に軽減できます。Zynq UltraScale+ MPSoC などの ARMv8 プロセッサでは、ソフトウェアへの投資を無駄にすることなく、次世代機能を実現できます。Zynq UltraScale+ MPSoC の詳細は、

xilinx.com を参照してください。また、ARM 社が提供するマイグレーション ガイドも参照してください (https://community.arm.com/docs/DOC-8453)。

参考資料

1. Jason Fitzpatrick、『An Interview with Steve Furber』Communications of the ACM、Vol. 54 No. 5 : http://cacm.acm.org/magazines/2011/5/107684-an-interview-with-steve-furber/fulltext
2. David Brash、『ARMv8-A Architecture Evolution』ARM Connected Community、2016 年 1 月 5 日 : https://community.arm.com/groups/processors/blog/2016/01/05/armv8-a-architecture-evolution

改訂履歴

次の表に、この文書の改訂履歴を示します。

日付	バージョン	内容
2016 年 4 月 11 日	1.0.1	タイトルの変更
2016 年 3 月 31 日	1.0	初版

免責事項

自動車用のアプリケーションの免責条項

この資料に関するフィードバックおよびリンクなどの問題につきましては、jpn_trans_feedback@xilinx.com まで、または各ページの右下にある [フィードバック送信] ボタンをクリックすると表示されるフォームからお知らせください。いただきましたご意見を参考に早急に対応させていただきます。なお、このメール アドレスへのお問い合わせは受け付けておりません。あらかじめご了承ください。