Linux を搭載した M32R アーキテクチャ研究開発用プラットフォーム

高田 浩和、 作川 守、 坂本 圭、 山本 整†、 稲岡 一弘††、 近藤 弘郁、 清水 徹

三菱電機株式会社 システム LSI 事業化推進センター、 †三菱電機株式会社 情報技術総合研究所、
††三菱電機セミコンダクタ・アプリケーション・エンジニアリング株式会社

あらまし 組み込み応用システムのためのアーキテクチャ研究開発用プラットフォームとして、Linux を搭載したソフトウェア／ハードウェア開発環境を構築した。組み込み用 32 ビット RISC マイコンである M32R プロセッサをターゲットとしている。今回開発した Linux プラットフォームでは、M32R ソフトマクロ・コアを用いることにより、FPGA 上で Linux を動作させた。2002 年 5 月現在、FPGA モデル版の M32R プロセッサを搭載したターゲットボードにおいて、2.4 系の SMP 対応カーネル上で Debian ベースのディスクレス・システムが稼働し始めている。

本論文では、Linux を新たなアーキテクチャに移植する際に何が問題となるのかについて述べ、今回のフィージビリティ・スタディを通して苦労した点についてふれながら、M32R 向け Linux 移植を実際どのように進めたのかについて具体的に説明する。

キーワード: M32R アーキテクチャ、M32R ソフトマクロ・コア、FPGA プロトタイピング、Linux 移植、SMP カーネル、組み込み用 Linux ソフトウェア・プラットフォーム、クロス開発

1. はじめに

組み込み応用システムのためのアーキテクチャ研究開発用プラットフォームとして、M32R プロセッサ向けに、Linux を搭載したソフトウェア／ハードウェア開発環境を構築した。

M32R は三菱電機オリジナルの組み込み(embedded)用 32 ビット RISC マイコンである。M32R は、デジタルスチルカメラ(DSC)、デジタルビデオカメラ(DVC)、携帯電話(PDC)など、デジタルコンシューマ機器を中心とした多くの機器に組み込んで広く使用されている組み込みプロセッサである。組み込みプロセッサの場合、チップ単体で使用されるよりも、システム LSI としてチップの内部に文字通り“埋め込まれ”て使用される場合が多いため、残念ながら M32R という名前が前面に出ることは少ない。

本論文では、まず最初に、今回のターゲットである M32R プロセッサ・コアと M32R ソフトマクロについて説明する。そして、Linux カーネルを M32R という新たなアーキテクチャへ移植する際にいったい何が問題となったかについて述べるとともに、M32R 固有の問題や実装について詳細に説明する。また、今回のフィージビリティ・スタディを通して苦労した点についても簡単にふれ、M32R 向け Linux 移植を実際どのようにして進めたのかについて具体的に説明する。

2. M32R プロセッサ用組み込みシステム・プラットフォーム

微細加工技術や設計 CAD 技術の進展により、最近ではシステム全体を1チップに集積することが可能となってきた。マイクロプロセッサ、周辺 IO 機能、メモリ、そしてユーザロジックを同一チップ上に集積した SoC(System on Chip)では、広い内部バスの利用などによって汎用チップの組み合わせでは実現できない性能・機能が実現できる。また、SoC では目的とするシステムに最適なチップ機能や回路構成を実現することができ、システムの性能とコストを最適化することができる。このようなシステム LSI においては、その中に組み込まれるプロセッサ・コアが LSI の性能を左右するキーパーツともなってくる。そのため、SoC 向けには、よりコンパクトで高性能なプロセッサ・コアが必要とされてきている。M32R コアはこのようなシステム LSI 向けに開発された、コンパクトで高性能な 32 ビット RISC プロセッサ・コアである。

M32R プロセッサ・コアのロードマップを図1に示す。今後、次世代携帯情報端末などのデジタルコンシューマ機器やホームサーバなどのネットワーク情報機器向けには、組み込み用途においても、1000MIPS を越えるデータ処理性能が要求されてくることが予想されている。M32R アーキテクチャをもつプロセッサ・コアとしても、今後、次世代 M32R コアとして、これらの用途にも十分適用可能な性能を持つコアを開発してゆく計画である。

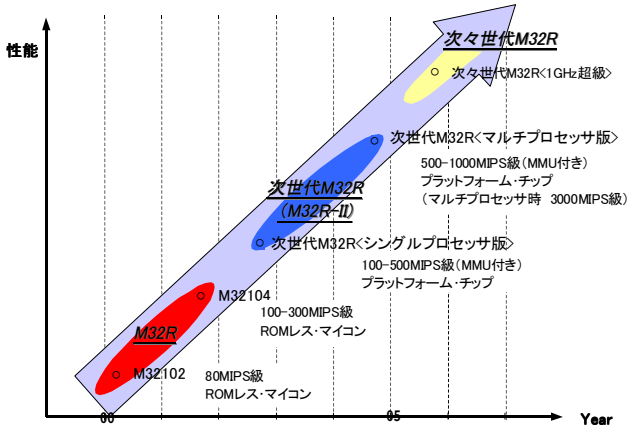

図1 M32R プロセッサ・コアの開発ロードマップ

そのような高性能なプロセッサ・コアを実現するためには、回路技術・プロセス技術だけでなく、アーキテクチャの面でも何らかのブレークスルーが不可欠となる。中でも、マルチプロセッサ技術は、プロセッサ数を増やすことにより、クロック周波数を上げずにスケーラブルな性能向上を期待できるのに加え、データ処理の機能分散を図ってシステム性能を大幅に向上できる可能性をもつことから、組み込み向けプロセッサにおいても非常に有用な技術と考えられる。

また、マイクロプロセッサを内蔵したシステムLSIを動作させるためには当然ながらソフトウェアが不可欠となる。ますます高機能化・複雑化するソフトウェアを効率よく作成できる環境を実現するためには、ソフトウェア・プラットフォームの整備が必須になってくる。そうした中、最近では、PC サーバとしての利用が進むLinux を組み込み用途にも適用する動きが盛んとなってきた。Linuxプラットフォームを使えば、ホストPC/EWSで開発したCプログラムを組み込みターゲットに容易に移植できるようになる。

現在のところははまだ、MMU を持たないプロセッサが使われる小規模な組み込み応用に特化したLinuxディストリビューションを組み込みLinuxと呼ぶ向きが多いが、今後、ますます高い処理性能が組み込みシステムに求められる状況を鑑みると、組み込み用途においても、これからは PC で動作するようなフル仕様のLinuxシステムをダウンサイジングして使用することが主流になってくると考えられる。

次世代M32Rプロセッサは、このようなLinuxプラットフォームを動かすためのエンジンとなる、プラットフォーム・チップとして位置付けられる。

組み込みシステム開発においては、ソフトウェア／ハードウェア両面からみてシステム性能を最適化することが重要であり、プロトタイピング時にはソフトウェア、ハードウェアの細かな部分まで変更・チューニングできるようになっていることが望ましい。

そこで今回の開発では、ハードウェアとして、既存のマイクロプロセッサ・チップを用いるのではなく、マルチプロセッサ対応のMMU付きM32Rソフトマクロ・コアを新規に開発、これをマッピングしたFPGAを搭載したターゲットボードを開発した。

プラットフォームOSとしては、前述のとおり、オープンソースとして全てが公開されているLinuxを採用することとした。対称型マルチプロセッサ(SMP; Symmetrical Multi Processor)が標準でサポートされていることもLinuxを採用した大きな理由の一つである。すでに述べたように、近い将来、組み込み分野にマルチプロセッサが使用される可能性は高い。M32R の組み込みLinuxプラットフォームでは、SMP対応のLinux環境を組み込みシステムで利用することを想定している。今回の開発ではSMP環境を実現することまでをねらって、プラットフォーム開発を進めた。

まずはプラットフォーム構築の第一歩として、M32R アーキテクチャへのLinuxシステムのポーティングを行い、そのシステム開発を通してハードウェアのシステム・アーキテクチャの検証を行うとともに、ソフトウェア開発とチューニングを行える環境を整備することを目指し、開発を進めた。

2.1 M32R アーキテクチャ

M32Rは32ビットRISCマイクロプロセッサであり、他のRISCプロセッサと同様、ロード・ストア・アーキテクチャを採用している。したがって、メモリアクセスはロード命令とストア命令のみにより行い、各種の演算はレジスタ間演算命令で実行する。乗除算命令を除くすべての演算命令は1クロックで実行され、命令の完了は発行順序によらない(out-of-order completion 方式)。また、積和演算・飽和演算といった DSP 機能命令を標準でサポートしている。

M32Rのレジスタ構成を図2に示す。M32Rは汎用レジスタとして32ビットレジスタを16本(R0～R15)、積和演算用に56ビットのアキュムレータを持っている。R14 はサブルーチン・コールの際の戻りアドレスを格納するためのリンクレジスタ(LR)としても使用される。また、スタックポインタにはSPI(割り込み用)とSPU(ユーザ用)の2本があり、プロセッサ状態語レジスタ(PSW)の SM ビットの設定により、R15(SP)としてどちらのスタックポインタを使用するかが選択される。

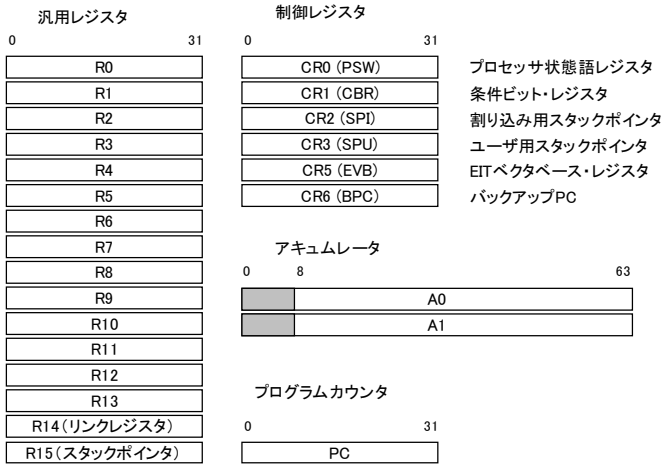

図2 M32Rのレジスタ構成

M32Rプロセッサには、命令長が16ビットと32ビットの2種類の命令がある。16 ビットの命令には使用頻度の高い演算が割り付けられており、コンパクトで効率のよい命令コード体系となっている。16ビット命令は2個を32ビットにパックすることができ、これら2 個の命令は、右側 16 ビット命令の MSB(パラレルビット)が 0 の場合はシーケンシャルに実行(逐次実行)、1 の場合はパラレルに実行(並列実行)される(図3)。

このように、M32R ではパラレルビットを設定することにより、静的にスケジューリングされた2個の16ビット長命令を並列に実行することが可能である。命令ペアを並列実行する場合、ロード・ストアおよび分岐命令は左側、積和演算命令は右側のみに配置可能となっている。

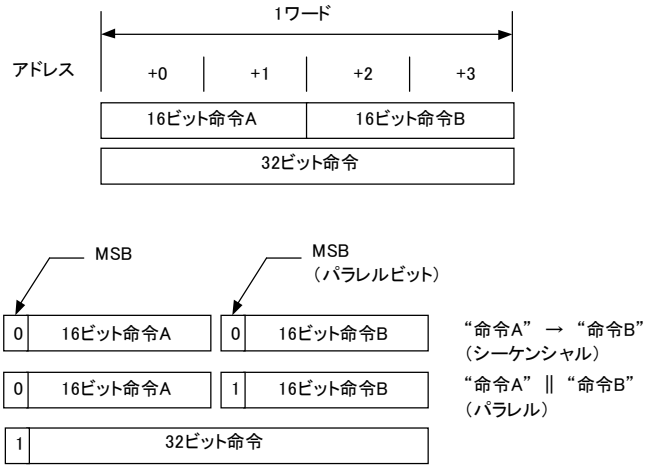

図 3 命令フォーマット

M32R の命令セットアーキテクチャ (ISA; Instruction Set Architecture)は、大まかに M32R ISA と M32R-II ISA の2つに分類できる。M32R ISA では、並列実行命令として右側 16 ビットに配置可能な命令は NOP 命令のみとなっているが、M32R-II ISA では論理・算術・シフト・比較の各演算命令が並列実行可能な右側命令として利用可能となっている。その他、M32R-II ISA では、プロセッサモードがサポートされ、いくつかの命令がスーパバイザモードでのみ使用可能な特権命令となっているほか、アキュムレータが 1 本追加されて2本となるなど、命令セットの強化が行われている。今回の Linux 移植では、M32R-II ISA 仕様の M32R ソフトマクロ・コアを用いた。

2.2 M32R ソフトマクロ

M32R ソフトマクロは、SoC に内蔵するためのコンパクトなマイコンとして開発されたもので、ハードウェア記述言語 Verilog HDL で記述されている。これは論理合成可能な機能記述モデルであり、特定のプロセス・テクノロジに依存していない。1相エッジ、同期型の記述であるため、EDA ツールとの親和性も高いという特長がある。そのため、他の IP(Intellectual Property)やユーザロジックと組み合わせて、ASIC や FPGA(Field Programmable Gate Array)上にインプリメントすることができ、フレキシビリティの高いシステムを容易に構築することができる。

M32R はコンパクトな 32 ビットマイクロプロセッサであり、周辺 IO を含めても、一個の FPGA にマッピングすることができる。そのため、FPGA 化して実際の回路動作を検証することも可能である。M32R ソフトマクロの HDL 記述を CAD ツールによって FPGA にマッピングすることで、実際にハードウェアを動作させながらソフトウェア開発を行うことができ、ハードウェアとソフトウェアの協調設計(コ・デザイン)および並行開発という開発スタイルをとることができる。このように FPGA を活用して容易にプラットフォームを構成できることから、ソフトマクロ・コアはシステム・プロトタイピングにも非常に有用なものとなっている。

M32R ソフトマクロの構成を図 4 に示す。CPU コアは CPU バスおよび、バスインタフェースを介して外部バスに接続されている。キャッシュメモリとは別に、CPU バスにはワークメモリとして使用可能な内蔵 SRAM を接続できる。また、ICU,UART 等の各種内蔵周辺 IO は周辺 IO バスに接続され、追加・削除などのカスタマイズが容易な構成となっている。CPU からは周辺 IO バスを介してこれらの周辺 IO 機能を利用することができる。

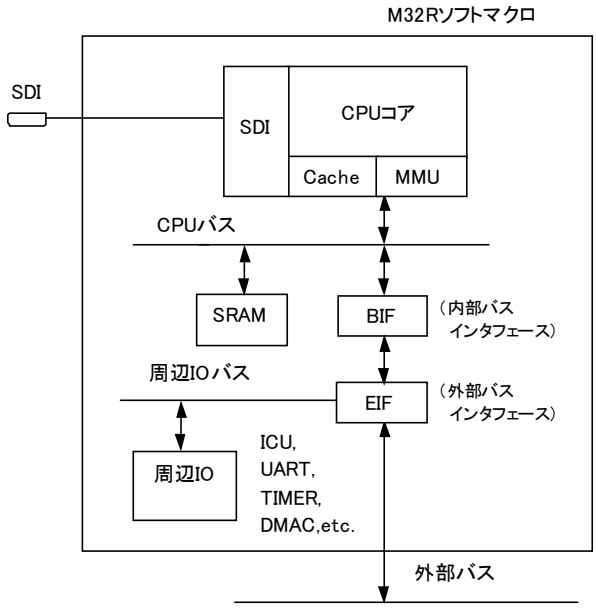

図 4 M32R ソストマクロの構成

Linux を移植するにあたり、既存の M32R ソフトマクロ・コアに対していくつかの機能拡張を行った。プロセッサ・モード(ユーザモードとスーパバイザモード)と新規命令を追加したのに加え、ハードウェアとして MMU(Memory Management Unit)モジュールを新たにサポートした。

- アドレス変換バッファ(TLB; Translation Lookaside Buffer)の構成は、命令 TLB／データ TLB、それぞれ 32 エントリ、フルアソシアティブ。ただし、FPGA 版では、回路規模の制約から各 16 エントリとした。
- ページサイズ: 4k/16k/64k バイト(ユーザページ)、
  4M バイト(ラージページ)。

2.3 内蔵デバッグ機能と SDI

M32R プロセッサ・ファミリの大きな特長として、標準でデバッグ機能を内蔵していることが挙げられる。SDI(Scalable Debug Interface)は、M32R ファミリに共通な内蔵デバッグ機能のインタフェース仕様であり、チップに内蔵されるデバッグ機能は、JTAG の5本のデバッグ専用ピンを使用して制御することができる。

SDI には以下の機能と特徴がある。

- JTAG インタフェースを使用したデバッグ制御。
  IEEE1149.1 で規定された JTAG インタフェースを使用して、内臓デバッグ機能を制御する。モニタプログラムの実行も JTAG インタフェースを使用して行うため、チップやボード上にモニタプログラムを格納するためのメモリが不要。

・スケーラブルな構成。
必須機能とオプション機能が提供され、内蔵ハードウェアや端子数を必要最小限に抑えることが可能。
・リアルタイムデバッグのサポート。
リアルタイムデバッグに必要な機能(デバッグ用 DMA 機能、リアルタイムトレースやイベント出力機能)をオプションで用意。

3. Linux/M32R の開発

Linux は、バージョンを重ねるごとに多くの機能追加がなされ、同時にさまざまなアーキテクチャのマシンに移植されて、いまや組み込みマイコンからサーバ、メインフレームまでの多岐にわたり動作する安定かつ強力な OS となっている。

しかし、Linux を動作させるには、当然のことながら、カーネルだけではなく、その上で動作させるアプリケーション・プログラムを構築するためのツール／ライブラリが不可欠である。また、実際に Linux を動作させるためのターゲットボードや実機デバッグの環境も必要となる。

そこで、具体的に、次の項目について開発を進めた。実際にはこれらの項目について、時系列に重点を移しながら並行して開発を進めてゆくというアプローチをとった。

・カーネル移植
・プラットフォーム開発(FPGA 版 M32R 搭載ボード)
・GNU ツール拡張(m32r-linux ツールチェイン)
・ライブラリ移植(GNU C library, etc.)

3.1 M32R への Linux カーネル移植

よく知られているように、Linux ではマシンアーキテクチャ依存部分が完全に切り分けられており、柔軟に種々のアーキテクチャに対応可能な構成となっている。新規のマシンアーキテクチャに Linux を移植する場合でも、新たに作成すべきコードは OS 全体からすればごくわずかでよい。

新規アーキテクチャに Linux カーネルを移植する場合、すでに存在する他のアーキテクチャの実装を参照しながら、アーキテクチャ依存部分のコードを実装することになる。M32R アーキテクチャへの移植の場合、具体的には、include/asm-m32r, arch/m32r ディレクトリ以下のファイルを作成することになる。

コードの書き換えに際して、とりわけ、ヘッダ・ファイル群の書き換えについては留意が必要である。アセンブリ言語や asm 文で書かれたコードを異なるアーキテクチャ向けに書き直す場合、機械的に一対一に置きかえることができない命令も多く、とりあえずコンパイルを通すために適当にコードを書き換えてしまうと、後になって非常にデバッグの困難な不具合を混入してしまう場合がある。inline 関数が多用されているという Linux カーネルのコーディング上の特徴のために、不具合箇所の特定はより困難になっている。したがって、移植に際しては、書き換え対象のファイルに関するカーネルの内部処理についてある程度理解を進めながら、段階をおって作業を進めることが望ましい。

そこで、ファイルを切り出した最小構成のカーネルをビルドし、その動作を確認しながら、include/asm-m32r と arch/m32r にファイルを順次追加して、コンパイルに必要なファイルを整備していく方法をとった。最初に stub ルーチンを用意し、kernel, mm, fs, その他といった順でファイルを付け加えながら、カーネルのビルドを徐々に進めていった。

カーネルの開発当初はボードもまだ無く、実機デバッグできる十分なターゲット環境が無かったため、GNU のシミュレータを使用した。これは MMU に対応していないものであったが、シミュレータなので実機環境のようにダウンロードで待つ必要もなく、C ソースレベルでデバッグできたことから、初期のデバッグには大いに威力を発揮した。実際、カーネルのブート時、init()ルーチンの最後に execve()の処理で外部コマンド/sbin/init が実行されるまではデマンドローディング機構は使用されない。外部コマンドの起動は、各種の初期化処理やカーネルデーモンの起動処理が終わった後であるため、romfs のルートイメージを initrd として使用することにより、スケジューラを含めたカーネルの基本動作の確認とデバッグをシミュレータを用いて行うことができた。

移植は最初、2.2 系カーネル(2.2.16)について進めた。その後、stock カーネルのバージョンアップに追従する形でいくつかのバージョンを経て、現在は 2.4 系カーネル(2.4.18)に移行している。

3.1.1 システムコール I/F

システムコールは TRAP 呼び出しを採用している。システムコールの TRAP インタフェース部分では、汎用レジスタおよびアキュムレータの退避と復帰を行う。ptrace によるレジスタ参照が行えるよう、汎用レジスタは全てのレジスタをコンテキストとして退避している。

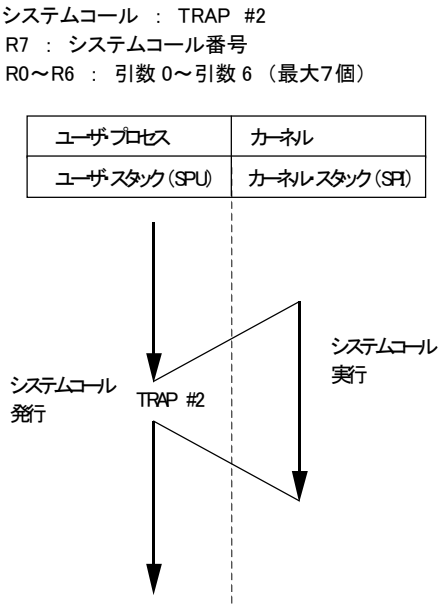

図 5 システムコール I/F

スタックは、Linux カーネル用に SPI を、ユーザプロセス用に SPU を使用する。M32R では、TRAP 命令の実行の際にスタックポインタは切り替わらない仕様であるため、TRAP インタフェースの入り口部分では、CLRPSW 命令により PSW の SM ビットを 0 クリアし、明示的にスタックポインタを SPI に切り替える。CLRPSW 命令によるスタックポインタ切り替えでは作業用のレジスタが不要であるため、レジスタ退避による多重例外は発生しない。

システムコール発行時のスタック・フレームの構成を図 6 に示す。カーネルのシステムコール関数の中には、sys_clone()などのように、スタックトップを引数にもつような特殊なインタフェースを持つものがいくつか存在する。そこで、SH アーキテクチャの実装を参考に、スタックトップの値(pt_regs)を暗黙の引数として、常にスタックに積んでおく方式を採用した。M32R 用gccの C 言語 ABI(Application Binary Interface)では、関数引数のうち、最初の 4 個までは R0～R3 を使用したレジスタ渡しにより渡されるが、それを超える引数はスタック渡しとなってスタックに積んで渡される。pt_regs をスタックに積んでおくことにより、8個目の引数として、スタックトップ・アドレスを暗黙的に渡すことができる。

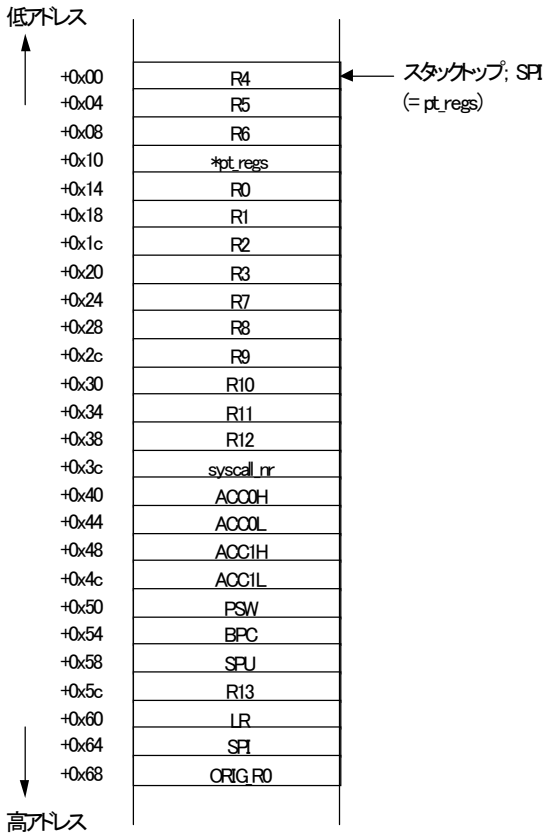

図 6 システムコール発行時のスタック・フレーム

syscall_nr および ORIG_R0 のフィールドの値は、シグナル処理において使用している。システムコールの発行時には、システムコール番号が R7 にセットされ TRAP が実行されるが、signal ハンドラがシステムコールから呼び出されたものかどうかを判定するために、スタック上の syscall_nr というフィールドにシステムコール番号を R7 とは別に保持している。また、R0 のフィールドはシステムコールの返り値がセットされて値が変更されるため、シグナル処理後にシステムコールを再実行する場合に備えて、ORIG_R0 というフィールドに元の R0 の値を保持している。

3.1.2 メモリ管理

Linux ではページングによりメモリを管理する。M32R 用カーネルでは、他のアーキテクチャと同様に、4kバイトのページサイズを採用した。デマンドローディングやコピー・オン・ライトの処理では、メモリアクセス例外(ページフォールト)が起きることによって初めて物理ページが割り当てられる。M32R 用カーネルでは、TLB ミス例外とアクセス例外のハンドラの処理を明確に分け、ページフォールトの処理を両者のハンドラを実行することにより処理する実装を採用した。

M32R 用カーネルにおけるデマンドローディング(demand loading)処理の流れについて図 7 に示す。ページテーブルに存在しないコード領域への命令アクセスもしくはデータ領域へのオペランドアクセスが発生すると、MMU 例外が発生し、例外ハンドラが起動される。TLB ミスによる TLB ミス例外が発生すると、TLB ミス・ハンドラが実行され、TLB エントリの設定が行われる。TLB ミス・ハンドラでの処理内容をできるだけ軽くするため、TLB ミス・ハンドラでは TLB エントリの設定以上の処理は行わない。ページのマップ処理やページテーブルの設定等の処理はアクセス例外ハンドラで処理を行う。

すなわち、ページテーブルにエントリが存在しないページについては、TLB ミス・ハンドラにてページ属性としてアクセス禁止属性を設定しておき、アクセス例外が発生するようにしておく。こうすることにより、TLB ミス・ハンドラからの復帰時に命令を再実行した際、引き続きアクセス例外が発生し、アクセス例外ハンドラが起動されてデマンドローディングの処理が行われることになる。

また、Linux では、fork によるプロセスの複製や、リード処理で読み込んできたページは、コピー・オン・ライト・モードで処理され、無駄なコピー処理を極力減らしている。このようなページのコピー・オン・ライト処理に関しても、最初にページの属性を書き込み不可の状態にしておき、書き込み処理によりアクセス例外が発生して初めて、ページのコピー処理を行うようにしている。

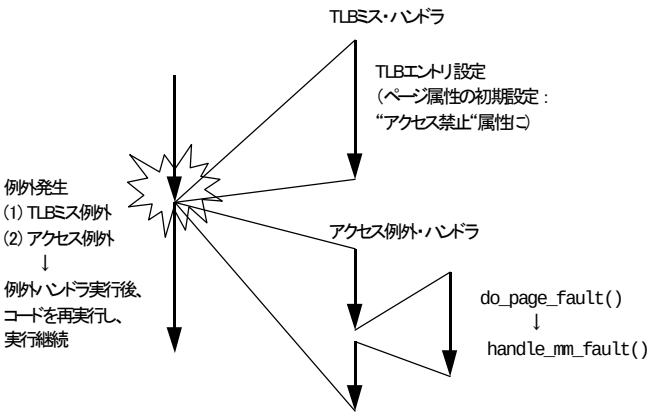

図 7 例外発生とデマンドローディング処理

3.1.3 SMP 対応

Linux2.4 では、資源をアクセスする場合の排他制御がカーネル・オブジェクトごとに細かく行えるようになり、マルチプロセッサ環境での性能向上が図られている。カーネル全体をロックするカーネル・ロック(ジャイアント・ロック)では、同時には一つのプロセスしかカーネル部分を実行することができず、マルチプロセッサでの処理効率が低下するためである。

このような、SMP 環境でのプロセッサ間の排他制御にはスピンロックが用いられる。M32R 用カーネルでは、M32R の LOCK／UNLOCK 命令を用いてスピンロックを実装した。LOCK 命令は排他制御用のロード命令であり、LOCK 命令を実行すると、UNLOCK 命令によるストアを実行するまでの間、プロセッサは CPU バスのバス権を獲得することができる。ただし、バス権を獲得できない場合にはプロセッサの LOCK 命令の実行はハードウェア的に待たされる。割り込みを禁止した状態で LOCK および UNLOCK 命令を用いて lock 変数にアクセスすることにより、lock 変数に対するアトミックなアクセスが可能となるため、これを利用してスピンロック操作を実現している。

また、SMP 環境では、システム全体の時計とは別に、プロセッサごとにローカルな時計(ローカルタイマ)を動作させる必要がある。M32R 用カーネルでは、ローカルタイマはプロセッサ間割り込みを用いてクロック割り込みを入れることにより実現した。

3.2 プラットフォームの開発

Linux では、メモリ保護機能やデマンドローディング機構を備えた OS であり、プログラムの実行には MMU が必須となる。そこで、Linux 開発プラットフォームとして、MMU 付き M32R ソフトマクロ・コアをマッピングした FPGA を搭載したターゲットボード Mappi(写真1)を開発した。

ソフトマクロ・コアと FPGA を使用しているため、CPU コアや内蔵 IO のモデル修正が必要な場合でも HDL を変更してただちに不具合を修正することができ、開発効率をあげることができた。

Mappi は上下に拡張ボードを接続可能な構造となっている。Ethernet、PC カード I/F を持つ拡張ボードを追加し、ネットワークブートすることができるようになったのは、2001 年 12 月頃のことであった。

M32R コアでは、すでに述べたように、SDI を使用して内蔵デバッグ機能を利用することができ、JTAG ポート経由でのプログラムのダウンロード、ステップ実行、ハードウェアブレークポイントの設定等が可能となっている。

一般に、開発途中の不安定なハードウェア上でのソフトウェア開発はデバッグが非常に難しい。しかし、SDI 機能が利用できたことでデバッグの作業効率はずいぶん向上した。加えて、ダウンロード・実行のために ROM モニタを用意する必要がないこと、Ethernet が利用可能となる以前から高速なダウンロードができたことも、ソフトウェア開発に際しては大きなメリットとなった。

最終的には、2個の FPGA にそれぞれ M32R プロセッサをマッピングしたマルチプロセッサ環境を構築し、SMP カーネル移植のプラットフォームも実現した。具体的には、図 8 の FPGA#1 のユーザロジック部分をもう一つの CPU コアで置き換え、FPGA#0 にバス・アービタを追加、さらに ICU をマルチプロセッサ対応に変更し、外部割込みを2個の CPU にブロードキャストするとともに、プロセッサ間割り込みに対応した。

Linux/M32R 自体は主としてユニプロセッサ環境で開発を進めたが、カーネルの SMP 化等の変更は、マルチプロセッサ環境を用いて並行して開発を行った。FPGA を利用しているため、ユニプロセッサとマルチプロセッサの環境の切り替えは、FPGA にダウンロードするモデルを変更するだけで済み、容易に環境を切り替えて検証を行うことができ、効率よく開発を進めることができた。

3.3 GNU ツール／ライブラリの整備

カーネルも含め、Linux 上のプログラム開発には GNU のツールチェインが必須である。しかし、開発を始めた時点では、クロスツールとして Cygnus Solution(現 RedHat)社製の組み込み用ツールキット GNUPro が使える状態にあったにすぎない。これは m32r-elf ターゲットのツールであり、カーネルの開発には利用できたが、そのままでは linux 上のアプリケーション・プログラムやライブラリ開発には利用できないという問題があった。

本格的な Linux システムを構築するためには、PIC(Position Independent Code)のコード生成を可能にして、ツールを shared library 対応にする必要がある。GNUPro の gcc はバージョンが 2.8 系と古く、いくつか不具合も見つかったため、FSF 版の GNU ツールに GNUPro 版との差分パッチを適用し、これをベースに m32r-linux ターゲットの GCC(2.95.4 系、3.0 系)、GNU Binutils(2.11.92 系)を開発した。i386 アーキテクチャの実装を参考に、bfd ライブラリを変更して ELF のダイナミックリンク機能に対応することで、shared library に対応した。

また、カーネルのデバッグを容易にするため、SDI によるリモート接続の行えるgdbを開発した。m32rsdi というリモートターゲットをサポートしており、/dev/m32rsdi というデバイスを用いて、ターゲットとのリモート接続が可能になっている。これにより、リモート環境を用いた C のソースレベル・デバッグが実現でき、MMU を使用するプログラムのデバッグが可能となった。/dev/m32rsdi デバイスへのアクセスにはデバイスドライバを使用するため、root 権限は不要であり、ユーザレベルでgdbを起動できる。

さらに、アプリケーション開発のためには GNU C library (glibc)を使えるようにしておく方が望ましいと思われたため、カーネルのスケジューラが動き出したあたりから、glibc(2.2.3)のライブラリ移植を並行して進めた。

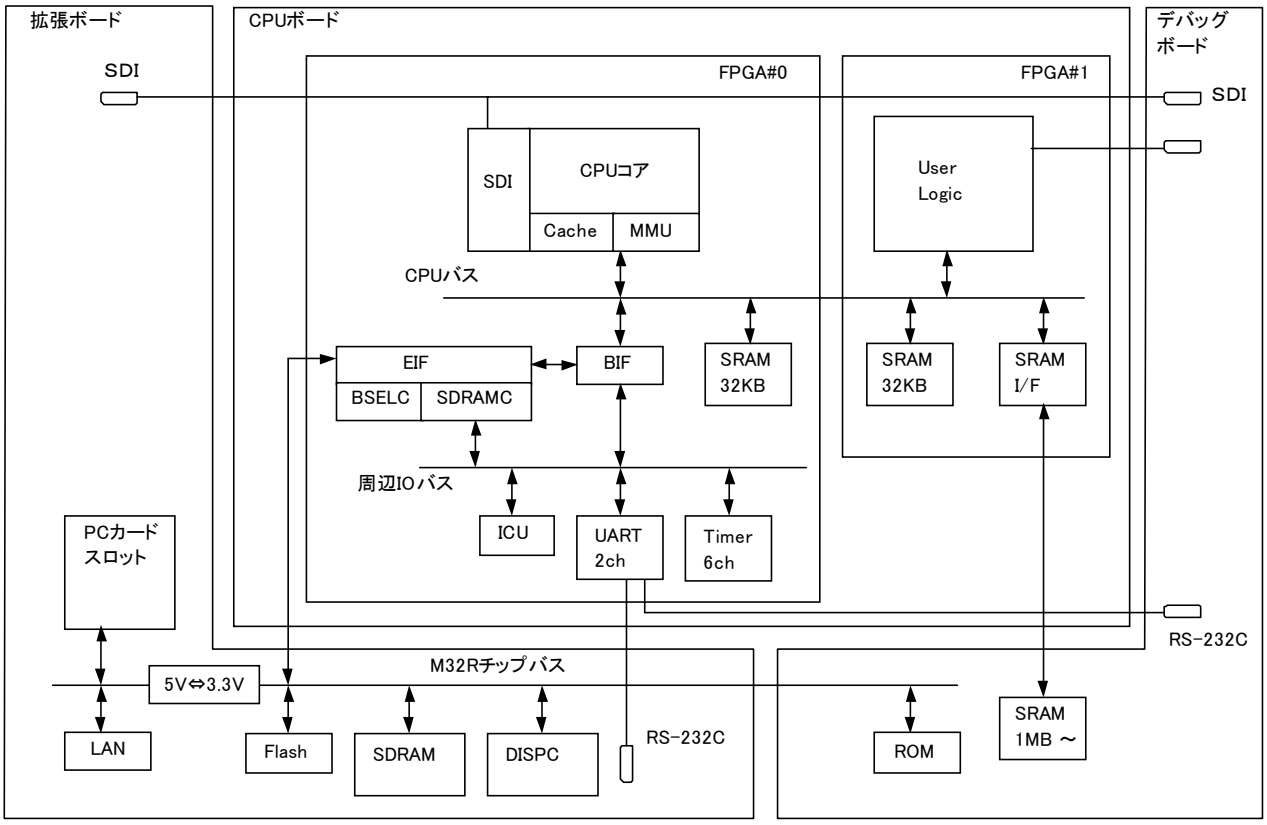

図 8　評価用ターゲットボード(Mappi)の構成
FPGA 版 M32R （M32R ソフトマクロ・コア: MMU, Cache, SDI, SDRAMC, UART, Timer)、
FPGA Xilinx XCV2000E x 2, SDRAM(64MB), Flash, 10BaseT Ethernet, Serial 2ch,
PCMCIA, Display I/F(VGA).

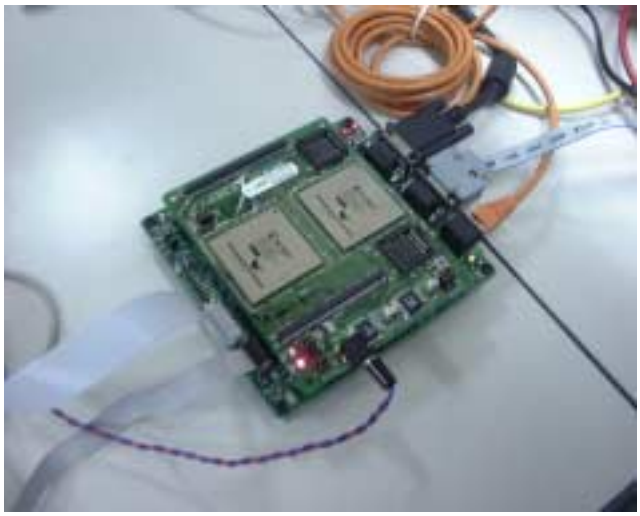

写真1．ターゲットボード: Mappi
CPU ボード(上段:109.2㎜×134.7㎜)と拡張ボード(下段)の
2段重ねで使用している状態。
CPU ボードには2組の FPGA とコンフィギュレーション ROM
ソケットが実装されている。コンフィギュレーション ROM を使用
すると、電源投入後、FPGA にモデル・データが自動的に
ダウンロードされ、FPGA にマッピングした M32R ソフトマクロ
の機能モデルをただちに動作させることができる。

写真2．Linux/M32R ブート時のコンソール表示の様子。
SDI 機能を利用して M32R 用 Linux カーネルをダウンロード・
実行している。左のボードが簡易エミュレータ。

glibc が用意できたことで、bash を始めとする、さまざまなプログラムがビルドできるようになった。とはいえ、実際にアプリケーション・プログラムが動作するようになるまでには数々の複合バグを修正する必要があり、以下のデバッグには合わせて数カ月を要した。

・システムコール I/F の見直し
・kernel の user_copy ルーチンの fixup 処理
・ダイナミックリンカ ld-linux.so によるリロケーション解決のデバッグ
・

中でも最もデバッグが難しいと思われたのは、ダイナミックリンク処理のデバッグである。しかし、シミュレータでのデバッグと、SDI によるリモート接続をサポートした gdb を駆使することで、なんとかデバッグを進めることができた。

M32R 固有の問題としては、asm 文やアセンブリ言語コードにけるテンポラリラベル(0:～9:)の記述には注意が必要である。テンポラリ・ラベル以外のラベルではワード・アライメントが必ずとられるのに対して、テンポラリ・ラベルではアライメントがとられない。そのため、分岐先を示すのに使用する場合には、必ず明示的に .fillinsn擬似命令を分岐先命令の前に挿入してワード・アライメントをとるようにする必要がある(リスト1)。

また、アセンブラによる最適化によって、16 ビット命令が並列実行されるよう最適化された場合に、テンポラリ・ラベルの示すロケーションが変化せず、分岐先が不正となるツールの不具合がある。これはアセンブラ内部でのテンポラリ・ラベルのロケーション情報の持ち方に起因する不具合であり、その改修には大幅な変更が必要と思われたため、現状、コーディングによって問題を回避している。リスト2の例では、.fillinsn 擬似命令を挿入してワード・アライメントをとり、nop 命令の並列実行コードを明示的に記述することにより、テンポラリ・ラベルの張られたst命令のロケーションを固定している。

```
        bra     1f
          :
        .fillinsn
    1:  addi    r0, #4
        ld      r1, @r0
```

リスト1. 明示的なアライメント指定

```
        .text
        addi    r0, #1
        .fillinsn
    1:  st      r5, @+r6    ||  nop

        .section __ex_table, “a”
        .long   1b
        .previous
```

リスト2. コーディングによる最適化時の不具合回避

3.4 開発環境

Linux/M32R の開発環境について図 9 に示す。プログラムのバイナリ・コードは、パラレル(または USB)ポートに接続された ICE(または簡易エミュレータ)を使用することにより、SDI 機能を用いて JTAG ポートを用いてホスト PC からダウンロード・実行することができる。

前述の M32R 用のgdbを用いると、パラレルポートに接続された簡易エミュレータからターゲットボードに JTAG ケーブルを接続するだけで、ブレークポイント設定、ダウンロード・実行、C ソースレベル・デバッグを行うことができる。

また、評価ボードの FPGA モデルを頻繁に変更する場合には、コンフィギュレーション ROM の代わりに FPGA アダプタを接続して FPGA にモデルをダウンロードすることになる。

また、シリアル・ケーブルを接続することで、シリアル・コンソール用が使用できる。

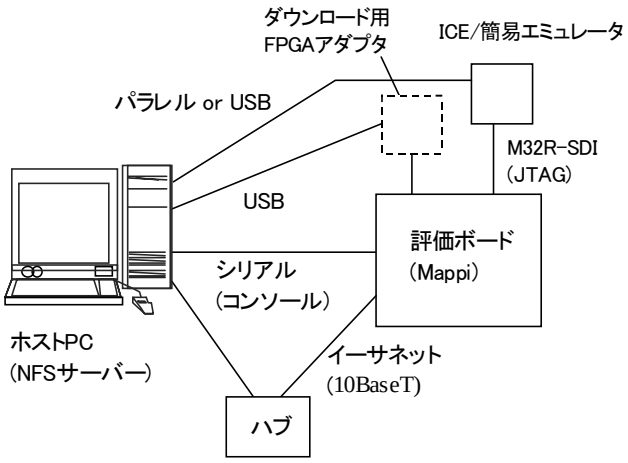

図 9 ソフトウェア開発・デバッグ環境

Linux/M32R の開発では、ルートイメージをホスト PC 上におき、これを NFS マウントすることで、ディスクレスな環境での開発を行った。NFSroot 機能を用いることで、Mappi をネットワークブートさせる。ファイルをホスト上に置いているため、開発中のカーネルがハングアップしてもファイルの内容が壊れることがなく、安全に開発を進めることができた。また、クロス開発したユーティリティやアプリケーション・プログラムは、ホスト上のファイルとしてただちにインストール／更新できるので、非常に効率的に開発を行うことができた。

カーネルのデバッグでは、上述のような SDI 対応のgdbを主として使用したが、それ以外にもkgdb、strace コマンドを併用した。kgdb を利用したデバッグでは、カーネル・ソースにkgdbパッチを適用してソースを変更することにより、シリアル経由でカーネルの動作をデバッグ可能である。M32R カーネル用のkgdbパッチでは、TRAP#1 を埋め込むことにより、ブレークポイントおよびステップ実行の処理を実現した。

また、strace コマンドによるシステムコール・トレースのログ取得は、システムコール引数や返り値の値のチェックに活用でき、

ライブラリのデバッグにも有用であった。しかし、今回の M32R 用カーネル移植では、ptrace 機能を最後に実装したため、strace コマンドをカーネルの基本動作のデバッグに活用することができなかった。カーネル移植に際しては、もっと早い段階で ptrace やシステムコール・トレースの機能を実装するのが望ましいと思われる。

4. ソフトウェア・パッケージのクロス開発

Linux/M32R のソフトウェア・パッケージ整備とディストリビューション化への試みとして、カーネル以外のソフトウェア・パッケージの構築を試みた。多様なソフトウェア・パッケージをコンパイルし、実機での動作確認を行うことで、ツールやハードウェアの不具合を発見することも目的である。

ソースがオープンであり、きちんとパッケージ管理されていることから、Debian/GNU Linux[4]をベースとなるディストリビューションとして選択した。Debian の場合、dpkgや apt といったパッケージ管理ツールが用意されており、パッケージのバージョン管理を行いやすいためである。

M32R 用のソフトウェア・パッケージは、Debian のソースパッケージからプログラム・ソースを展開し、dpkg-buildpackage コマンドを用いて、M32R 用の deb パッケージ(ローカルパッケージ)を作成した。

セルフツール群やシェル等の基本コマンド、ユーティリティ、Web サーバ(boa)、deb パッケージ管理ツール dpkg/apt など、現在、M32R 用 deb パッケージとして 99 個のパッケージが作成済みである。

主な M32R 用のパッケージについて以下に列挙する:

adduser, anacron, apt, base-files, bash, bc, binutils, bison, boa, bsdgames, bsdutils, devianutils, devfsd, diff, dpkg, elvis-tiny, fileutils, findutils, flex, ftp, grep, gzip, hostname, klogd, less, libc6, locales, login, lynx, make, mawk, modutils, mount, net-tools, netbase, netkit-inetd, netkit-ping, passwd, portmap, procps, rsh-client, rsh-server, samba, sash, strace, sysklogd, tar, tcpd, tcsh, telnet, util-linux, wu-ftpd

これらパッケージ(セルフ環境用の deb パッケージ)の大部分はクロス開発により作成している。dpkg-buildpackage コマンドのオプションにて m32r をターゲットと指定することで、クロスコンパイルを行うことができる (dpkg-buildpackage -a"m32r" -t"m32r-linux"-uc -us)。

perl については configure と外部モジュールの作成にセルフ環境での実行が必要なため、セルフツールを用いて手作業でパッケージをビルドした。なお、セルフ用の gccについてはまだパッケージ化できていないため、クロスコンパイルしたものを手作業でインストールしている。その他のセルフツール(binutils など)については、クロス開発した deb パッケージを dpkg コマンドを用いてインストールした。

クロス開発ではアプリケーション・プログラムをコンパイルするための、クロスツール用のヘッダ・ファイルおよびライブラリが必要となる。これらはクロスツールとは別に作成・インストールしなければならない。そこで、図 10 に示すように、dpkg-cross コマンドを用いてセルフ環境用のパッケージをクロス環境用のパッケージに変換し、変換したそれらのパッケージをホストマシンにインストールするようにした。こうすることで、ホストマシン上の deb パッケージ・データベースが利用でき、クロス開発用のファイル管理が容易となった。また、セルフ用とクロス開発用でソースを別個に管理する必要がなくなった。

5. 今後の展開

Linux/M32R としては、今後、組み込み向けのシステム・チューニングが必要と考えられる。組み込み向けに、カーネルサイズのシュリンクや、性能チューニング、リアルタイム性の改善を行っていく必要がある。 また、ディストリビューション開発に向けては、さらにソフトウェア・パッケージの充実と安定化を図り、M32R の機能を生かしたミドルウェアやドライバの開発・拡充に今後注力していく予定である。

ハードウェアについても、今後、システム性能の評価を行ってキャッシュ構成やバスプロトコルについての検討や、モデルのタイミング改善など、M32R ソフトマクロ・コアのチューニングを継続

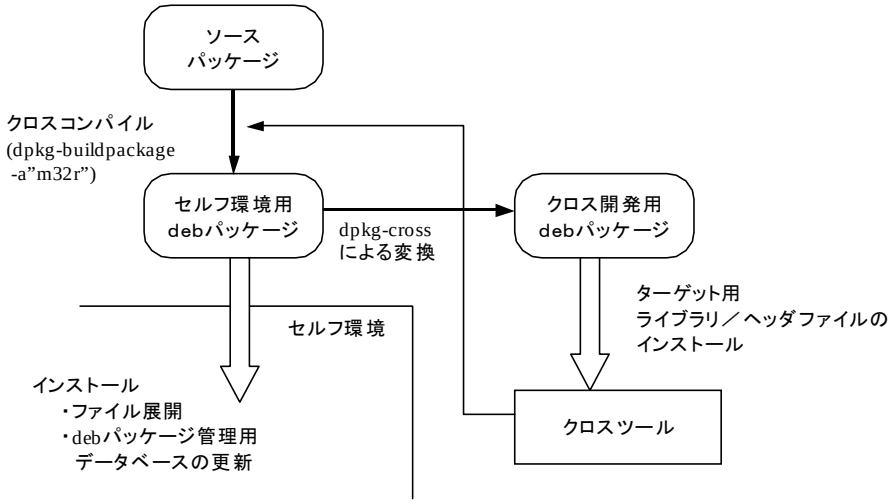

図 10 deb パッケージを利用したクロス開発の流れ

して行きたいと考えている。

SMP 対応についても、コヒーレンシ・プロトコルや割り込み処理の分配方法についても検討の余地がありそうであり、システム性能の評価を行いながら、ソフトウェアとハードウェアの両面から性能のチューニングを進めて行くことを考えている。

2.5 系の Linux カーネルでは、O(1)スケジューラが採用されたことで、マルチプロセッサ向けのスケジューリング処理が強化された。Linux カーネルについては、M32R 用 2.4 系カーネルのさらなる安定化を図るとともに、開発版である 2.5 系のカーネルについても stock カーネルの開発動向についてウォッチしながら、M32R 用カーネルのアップグレードを図ってゆく予定である。

6. おわりに

GNU/Linux というオープンソースの優れたリソースを活用することで、非常にフレキシブルに効率良く開発ができる環境が利用できるようになってきた。今後、組み込みシステム開発においても、オープンソースの活用が大きなインパクトを与える場面は増えてくると予想される。

M32R プロセッサへの Linux の移植では、カーネルの移植、GNU ツールの拡張／ライブラリの整備など、多岐にわたるソフトウェア開発を行った。ターゲットボード等のハードウェアをも同時開発する中での作業は困難とも思えたが、1年半たらずでこのようなシステムの立ち上げを行うことができたのは、ひとえにオープンソースの賜物といっても過言ではないであろう。Linux/M32R の開発成果についても、機会をみて積極的に公開していきたいと考えている。

M32R ソフトマクロと FPGA によるフレキシブルなハードウェア環境を用いてソフトウェアとハードウェアの並行開発ができたことは画期的なことであった。フィードバックや不具合の修正が容易なソフトマクロ・コアを用いたことは、プロジェクトを加速する上で非常に有効であった。

このことは同時に、M32R ソフトマクロ・コアと FPGA を用いることで、誰もが FPGA 上で Linux を動かせる環境が実現できることを意味している。Linux を載せたカスタム・コンピュータを我々が自分自身の手で作れる世界はもうすぐそこまで来ている！！

今回の M32R アーキテクチャへの Linux 移植は非常に興味深く貴重でエキサイティングな体験であった。Linux ユーザとして、このようなプロジェクトに携わることができたのは非常に好運だったと思っている。最後になったが、今回の開発にご協力いただいた関係各位にこの場を借りてお礼を申し上げるとともに、オープンソースの偉大な成果を生み出してきた GNU/Linux コミュニティの方々の努力に感謝と敬意を表したい。

参考文献

1) 中島雅美、近藤弘郁、高田浩和、作川守、樋口崇、大谷素賀子、山本整、稲坂朋義、白井健治、清水徹、「FPGA プロトタイピングを活用した高性能マイクロコンピュータ“M32R”の開発」、信学技報、CAS2002-36, VLD2002-50, DSP2002-76 (2002-06), pp. 61-66.

2) 高橋浩和、三好和人、「図解 Linux カーネル 2.4 の設計と実装」、Linux Japan、Nov. 2000 – Nov. 2001.

3) 高橋浩和、「Linux V2.4 カーネル内部解析報告、ドラフト第 4 版」、
http://www03.u-page.so-net.ne.jp/da2/h-takaha/internal24/

4) Debian/GNU Linux
http://www.debian.org/

5) 東京大学大規模集積システム設計教育研究センター(VDEC)ホームページ、M32R ソフトコア提供プログラム
http://www.vdec.u-tokyo.ac.jp/CHIP/M32R/M32R.html
※ MMU なしの製品版 M32R ソフトマクロ(M32102)については、テストベンチ、テストケースなどを含む開発環境が、すでにオープン化され、VDEC から無償公開されている。秘密保持契約の下で、研究および教育を目的とした利用(改変を含む)が可能となっている。