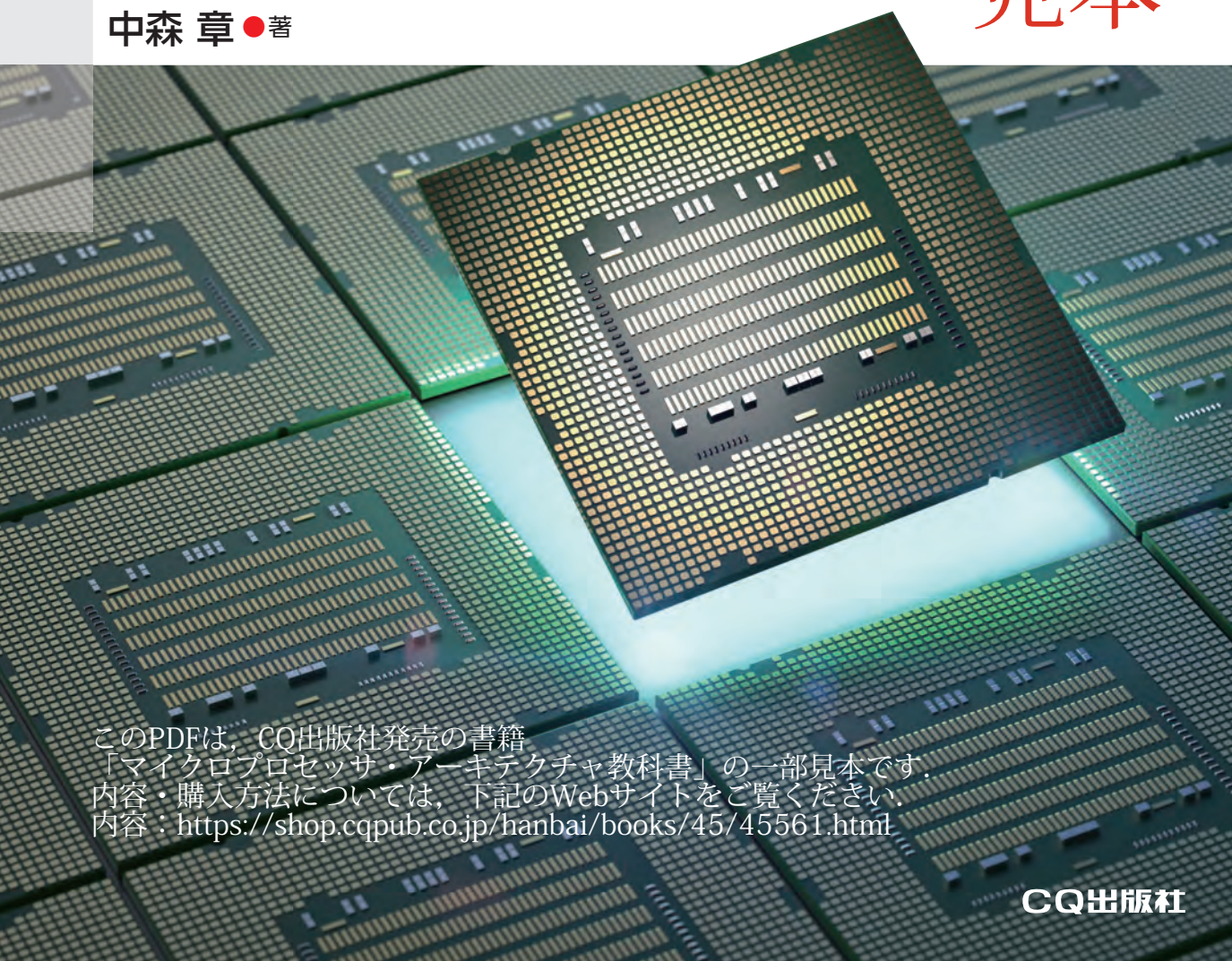
TECHI Vol.57

マイクロプロセッサ・アーキテクチャ教科書

MPUの基本からAIプロセッサ, TrustZone, RISC-Vまで

見本

中森 章 ●著



CQ出版社

Chapter 1

プロセッサの構成要素と動作の基本

プロセッサの基礎知識

プロセッサとは何だろう．専門書や教科書を読んでも難しそうだ．しかしMPUとは，そんなに複雑なものだろうか．じつは，その背景にある考え方は単純なのではないだろうか．直感で理解できたら嬉しい．それが本書のテーマである．まずはMPUの基本的な動作について解説する．

1 コンピュータができること

● 数値計算

大規模数値計算，高度な意志決定支援，人工知能など，コンピュータの応用分野は無限にある．しかし，コンピュータ自体ができることは単純である．データの「転送」，「加減乗除」，「論理演算」，「シフト」，「分岐」といった基本操作だけである．その計算結果を，人間 が意味付けすること，たとえば，「結果がある値になったらある事象が成立したとみなす」とすることで，さまざまな結論を導き出す．つまり，ある一般的な問題を数値計算の操作に代表させ（これを問題のモデル化という），問題解決を行う．

極端な一例を示す．たとえば「命」と名付けられた記憶領域があり，そこには0または1の値が入るものとする．0は死んだこと，1は生きていることと意味付ける．コンピュータでの操作としたら，起動時に「命」に1を格納し，ある時間が経過したら「命」に0を格納するものとする．これは人の一生を計算していることになる．定期的に「命」の値を調べていれば，「あっ，死んだ」とか「あっ，生き返った」とか言うことができる．

あるいは，論理的な思考を実現するためには，仮定と結論の組み合わせや，ある項目から連想される別の項目を多数記憶しておき，最初の仮定から始まって，そこから得られる結論を次々と連想される項目に置き換えていくことで，最終的な結論を得られる．この操作を実現する基本操作は比較処理である．比較処理は，コンピュータでは排他的論理和という論理演算で実現される．また，連想結果の置き換えは転送処理に他ならない．結局，複雑に見える処理も基本操作の積み重ねで実現されるのである．その意味では，スーパーコンピュータのMPUも，サーバのMPUも，PCのMPUもできることに大差はない．違いは，データの処理速度くらいであろう．

● プログラム内蔵方式

さて，数値計算は定型的な処理であるので，ある程度自動化できる．たとえば，自動数表作成機や微積分装置などは，数値的な定型処理を機械化したものである．これらはコンピュータと呼べるだろうか．答えは否である．コンピュータをコンピュータたらしめる属性はプログラム内蔵方式と条件分岐にあるといわれている．

プログラム内蔵方式とは，コンピュータの動作を規定するプログラム（命令列）をシステムの記憶装置に内蔵していることをいう．命令とデータに区別はなく，命令の実行によって記憶装置にある命令を変更してそれを実行できるのが大きな特長である．このような命令の自己書き換えに関しては，デバッグの難しさや保護の難しさから推奨されない．しかし，これは粒度（書き換えてから実行するまでの時間的空間的距離）の小さい場合である．粒度の大きい場合は，ハードディスクなどの補助記憶装置から記憶装置に命令やデータをロードして，その部分を実行することに等しく，これはコンピュータシステムにおいてごく普通に行われる．

条件分岐とは，途中の計算結果に応じて処理を切り分けることができる機能を指す．プログラムを見ただけでは分岐が発生するか否かは不明である．その条件分岐を処理する段階になって初めて分岐するか否かが決定されるのである．条件分岐のおかげで，繰り返しや複雑な制御構造が実現できる．

以上のコンピュータの属性を一言でいえば，記憶装置にある命令を実行する有限オートマトンである．有

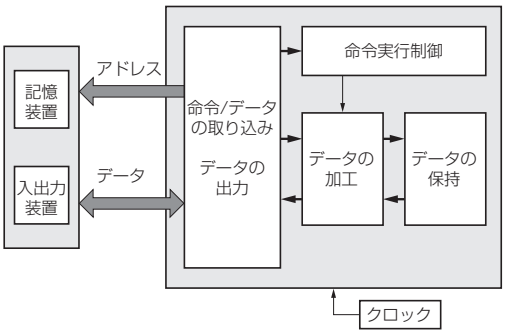


図1　MPUの構成要素

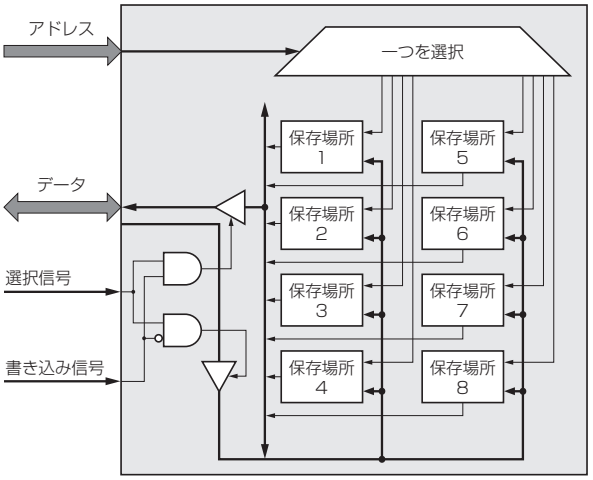


図2　メモリの概念図

限オートマトンの意味に関しては，抽象的で難しい概念なので，ここでは省略する．

そして，マイクロプロセッサ，あるいは，MPUとはコンピュータを1チップに集積したものである．本章ではMPUの具体的な動作原理について説明する．本章では主として「MPU」という単語を使用するが，ここでの議論は，それを「コンピュータ」に置き換えても，そのまま当てはまる．

最近では「MPU」と言われてもピンとこないかもしれない．「プロセッサ」という言葉の方が通りがいい．また，大規模なプロセッサに対して小規模なものは「コントローラ」とか「マイコン」などと呼ばれる．コントローラやマイコンは「MCU」とも呼ばれる．本書では，これらを総合してMPUと呼ぶ．MPUとMCUでは微妙な違いもあるが，それは必要に応じて説明する．

2 MPUの構成要素

現代のMPUは**フォン・ノイマン型**と呼ばれる．これは**プログラム内蔵方式**のことであり，フォン・ノイマンが提唱したということになっているが，最近ではそれは誤りとされている．フォン・ノイマン型とかフォン・ノイマンボトルネックという言葉はやがて消滅するかもしれないが，ここでは慣例にしたがっておこう．

典型的なMPUは，「記憶装置」，「命令やデータを取り込むしくみ」，「命令実行を制御するしくみ」，「データを加工（処理）するしくみ」を基本的な構成要素とする．また周辺機器とデータをやりとりするための「入出力処理」というものもある．**図1**に典型的なMPUの構成要素を示す．

● プログラム内蔵方式には記憶装置が必須

プログラム内蔵方式には，プログラムを内蔵するための記憶装置が必須である．ほかの構成要素はMPUに内蔵されるが，記憶装置は，基本的にMPUの外部にある．この記憶装置は**メモリ**と呼ばれ，その構成によって**ROM**（Read Only Memory）と**RAM**（Random Access Memory）に大別される．

メモリとは複数の保存場所の集合で，各保存場所には位置を特定するための**アドレス**が付けられている．そして，あるアドレスを指定すると，それに対応する保存場所の内容が外部に読み出されるという装置である．RAMでは，アドレスと新しいデータを与えることで，アドレスで指定される保存場所の内容を変更することもできる．**図2**にメモリの概念図を示す．

メモリ内の保存場所の大きさ（ビット数）はいくつでもかまわない．しかし，最近のメモリは一つの保存場所の大きさを8ビット（＝1バイト）とする**バイトアドレス**方式が主流である．つまり，一つのアドレスを与えると1バイトのデータを得ることができる[注1]．

しかし，MPUの扱うデータは1バイトのみとは限らない．ハーフワード（16ビット），ワード（32ビット），ダブルワード（64ビット）といったデータ[注2]も

注1：メモリによっては保存場所の大きさが16ビット（×16という），32ビット（×32という）のものも存在する．どちらかといえば，16ビットが一般的かもしれない．だたし，その場合もバイト単位での書き込みは可能になっている．

注2：インテルやモトローラに代表されるCISC時代からMPUを使ってきている人は，16ビットをワード，32ビットをダブルワード，64ビットをクオドワードと呼ぶ．現在は32ビットMPUが主流なので32ビットをワードと呼ぶのが自然だが，16ビットMPU時代の慣習も根強く残っている．CISCメーカーは16ビットをワード，RISCメーカーは32ビットをワードと呼ぶ傾向が強い．

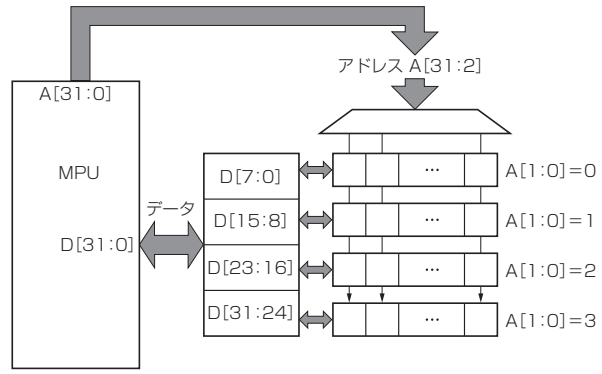


図3
32ビットデータバス

扱う．おそらく，もっとも多用されるデータ長はワード（16ビットまたは32ビット）であろう．「○○ワード」という表現は，ワードが基準になっていることを示す証拠である．このため，MPUはある程度まとまったビット数（あるいはワード単位）のデータをメモリから取り出すほうが効率的である．このため，MPUとメモリ間のデータのインターフェース（データバス）は16ビット幅または32ビット幅であることが多い（アドレスのビット幅はまちまち）．たとえば，データバスが32ビット幅の場合は，1バイト出力のメモリを4個並列につなげて32ビットのデータ供給を実現できる（**図3**）．

アドレスはバイト位置を示すものである[注3]のに，一つのアドレスから4バイト（32ビット）のデータを取り出そうとすると「バイト並び（エンディアン）」の問題が生じる．複数バイトから構成されるデータに対し，バイトアドレスのより小さい保存場所にデータの上位の値を置くか，データの下位の値を置くかで2通りの方法がある．アドレスの小さい場所にデータの上位を置くのが**ビッグエンディアン**，逆にデータの下位を置くのが**リトルエンディアン**である．メモリでのバイト並びは異なるが，ビッグエンディアンでもリトルエンディアンでも，データバス上では同じイメージになる（**図4**）．

MPU内の演算器などはデータの下位側から計算をしていくので，その意味で，すべてはリトルエンディアンに集約されるといえなくもない．エンディアンとは，あくまでもメモリ上にデータがどの順序で格納されているかを示しているにすぎず，MPUの内部処理とは直接は関係ない．また，ビッグエンディアンの場合，ビット番号の名付け方がリトルエンディアンと逆順になっていることが多く，惑わされやすいが，実質（バイト内のビットイメージ）は同じである．

注3：昔の大型計算機などではアドレスの割り付けがワード単位になっているものもある．つまり，すべてのデータはワード単位でしか扱わないのが基本である．このような場合にはエンディアンの問題はない．

● 命令やデータを取り込むしくみ

MPUがまず行わなければならないことは，メモリに格納された命令やデータを内部に取り込むことであ

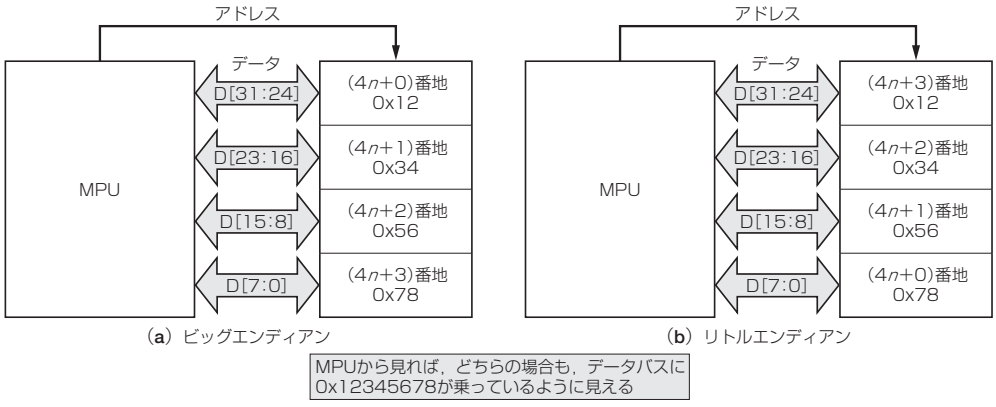


図4　エンディアンとデータバスの関係

Column エンディアンの由来

エンディアンという言葉はコンピュータ用語の中ではそれほど古くない．1980年にDanny Cohenが『On Holy Wars and a Plea for Peace』という論文の中で初めて使用したというのが定説である．その語源はジョナサン・スイフトの『ガリバー旅行記』にある．小人国（リリパット）の中に出てくる，ゆで玉子を小さい端から食べる（割る?）主義の人々と大きい端から食べる主義の人々が由来である．「端」を表す「エンド」という単語に，「主義者」を表す「イアン」（例としてベジタリアンなどがある）が合成されてできた．

昔は，リトルエンディアンをインテル形式，ビッグエンディアンをモトローラ形式と呼んでいた．本来，リトルエンディアンはDECが，ビッグエンディアンはIBMが提唱したものらしいが．エンディアンのことを，ちょっと前のMIPSのドキュメントではSEX（性別）と書かれていた時期もある．なぜか，ビッグエンディアンが男（male）で，リトルエンディアンが女（female）である．

エンディアンという表現は，日本では坂村健氏が広めたような覚えがある．

る．そのためには，メモリに与えるアドレスを生成し，メモリから出力されたデータを取り込めばよい．

まず，命令について考えよう．MPUは**PC**（Program Counter：プログラムカウンタ）という記憶機構（レジスタ）を備えている．これは，命令を取り込むメモリのアドレスを保持する．PCの値はアドレスバスを通じてMPUの外部に出力され，これがメモリに入力される．アドレスバスに値を出力すると，データバスを入力状態にしてメモリから出力された値（命令）を取り込む．これを**命令フェッチ**という．

MPUがリセットされると，ある特定の値がPCの初期値として設定される．そしてPCは，通常はメモリから取り込んだ命令のバイト数だけ値が増加していく．メモリから取り込んだ命令が分岐命令だった場合は，分岐先のアドレスが新たなPCとして設定される[注4]．

データバスから取り込まれた命令は，一般に，命令レジスタと呼ばれる記憶機構に保持される．それ以降の命令処理は，命令レジスタの内容にしたがって行われる．

さて，命令が扱うデータについて考えよう．メモリには命令のほかにデータも格納されている．命令によっては，その実行のためにメモリのデータが必要である．このため，命令の実行にともなって，メモリに格納された値が必要になった場合に活性化される機構を備えている．メモリを参照するアドレスはロード命令やストア命令などを実行（正確にはデコード）することで生成される．このアドレスはオペランドアドレスレジスタ（とりあえず，そう命名する）という記憶機構に保持される．**オペランドアドレスレジスタ**の値はアドレスバスを通じてMPUの外部に出力され，これがメモリに入力される．アドレスバスに値を出力すると，データバスを入力状態にしてメモリから出力された値（データ）を取り込む．これを**オペランドフェッチ**，または，**オペランドリード**という．

図5にMPUの命令やデータを取り込むしくみを示す．

● 命令実行を制御するしくみ

MPUは，メモリから取り込んだ命令を解釈し実行する．メモリから取り込まれた命令は**命令レジスタ**に保持され，それが**命令デコーダ**によって解釈（デコー

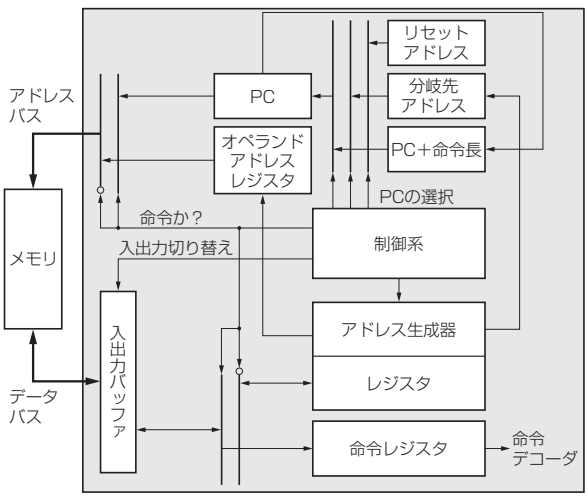


図5　命令やデータを取り込むしくみ

注4：分岐先のアドレスは，通常，分岐命令の実行によって決定される．つまり，PCを分岐先に設定し直すタイミングは，分岐命令の実行後である．しかし，最近のMPUでは命令のフェッチと実行部分が切り離されている場合もあり，この場合は命令フェッチ部が自律的に分岐命令を処理する．

Chapter 2

もっとも基本的なプロセッサ高速化技法

パイプライン処理の概念と実際

パイプラインとはMPUの命令実行を高速化する手法の一つであり，現在では，ほとんどすべてのMPUで採用されている．RISCのパイプライン処理は，見事なまでにヘネシー＆パターソンが提唱した5ステージのパイプラインにしたがっている（『コンピュータアーキテクチャ』，通称ヘネパタ本を参照のこと）．前半では一度に1命令を実行する通常のパイプライン（シングルパイプライン）についての基本概念を説明する．2命令以上を同時実行するものはスーパースカラと呼ぶが，これと対比する場合はユニスカラパイプライン，あるいは単にスカラパイプラインとも呼ばれることもあるようだ．後半ではシングルパイプラインの代表とも言えるMPUやMCUのパイプライン構造を解説する．

パイプライン処理の概念

1 パイプラインとは

● 流れ作業＝パイプライン

コンピュータの性能を向上させる方法については，いろいろ考案されている．**パイプライン**とは，ハードウェアを並列化して性能を向上させるための一般的な手法である．その基本的な考え方は，プログラム内蔵方式を提唱したフォン・ノイマンによってすでに提案されていたという．たとえば，MPUの命令実行に比べて10倍以上も遅いメモリアクセスが存在する状況下で効率的に命令の処理を行うために，命令の実行とメモリアクセスをオーバラップして処理することが考えられた．これが，パイプライン処理の原型である．

パイプラインの基本的な考え方はごく自然なものである．なにもコンピュータの技術に固有なものではない．自動車の製造ラインや電子部品工場などで行われている流れ作業は，パイプラインそのものである．一つの製品が数分ごとに完成していくようすを思い浮かべてほしい．実際，パイプラインの呼び名は，石油が次々とパイプを通過していく石油化学パイプラインと動作が似ていることに由来している．

各工程が1単位時間かかるN工程からなる処理を考える．単純に考えると，この処理を終了するためにはN時間を要する［**図1**（**a**）］．これをN人の人が流れ作業によって各工程を分担し，前の工程から受け継いだ製品に1単位時間で加工を施して，後の工程に引き継ぐようにする［**図1**（**b**）］．この場合，もともとの処理ではN時間に一つしか製品が完成しないが，流れ作業では見かけ上，1単位時間に一つの製品が完成することになる．つまり，処理速度はN倍に改善される．これがパイプラインの原理である．

● ステージ，段数，ハザード

ここで，各工程をパイプラインの**ステージ**という．「段」という表現も使われ，N工程から構成されるパイプラインはNステージパイプライン，またはN段パイプラインと呼ばれる．また，あるステージを分担する人が手間取って，そこでの処理を1単位時間以内に終わらせることができないような場合は，パイプライン処理に乱れが生じ，処理性能が低下する．パイプラインステージでの処理を単位時間内に終わらせることを阻害する要因を**ハザード**という．

パイプライン処理をコンピュータに適用する場合は，各ステージが並列に処理できることが前提である．ハードウェア資源を共有するステージがあると，ハザードが生じ，待ち合わせが必要になる（これを**ストール**という）．逆にいうと，ハードウェア資源が競合しないようにパイプラインステージを分割するのがプロセッサ設計者の腕の見せどころである．

パイプライン処理は，まず大型計算機で採用され

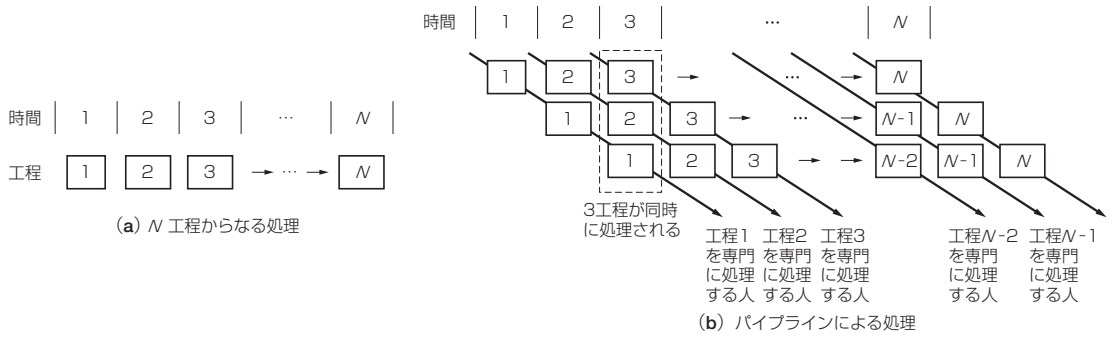


図1　パイプライン処理の概念

た．その後，半導体の集積技術が進み，MPUでも大量のトランジスタが利用可能になると，MPUにも採用されるようになる．パイプライン処理の採用を大々的に表明したMPUは，NECのV60が最初ではないかと筆者は記憶している．それ以前のIntelの8086でもオペランドフェッチと実行をパイプライン化していたが，Intelがパイプラインを明言したのは80386以後（最近のIntelの発言ではPentium以降）となっている．一方，68000系のMPUも古くからバスサイクル同期のパイプライン処理をしていたようである．しかし，こちらもパイプラインを明言したのは68060が初めてだったと思う．68060はすでにスーパースカラ構造になっていたので，シングルパイプライン時代の68000系のパイプライン構造は不明である．

2 パイプラインの理論

● パイプラインステージ

CISC初期においてもパイプライン構造を採用しているものがあった．しかし，それらのMPUにおいてパイプライン処理は有効に機能していたとはいえない．各MPUメーカーがパイプラインを強調しなかったのは，それが性能に寄与していなかったからではないかと考えられる．しかし，RISCの登場によってパイプライン処理はにわかに脚光を浴びる．RISCのパイプラインは，CISCとは異なり，全命令でパイプラインのステージ数は固定であることが多い．筆者だけの感覚かもしれないが，命令フェッチ，命令デコード，実行という処理の流れも，その区切りが明確になっているように感じる．

RISCの存在意義は，パイプライン処理をいかに効率的に実現できるかにかかっているといっても過言ではない．このため，RISCでは命令やオペランドをキャッシュからフェッチすることを前提としている．通常のメモリはアクセス時間が遅いので，メモリアクセス・ステージの処理時間が他のステージに比べて長くなり，効率的なパイプライン処理を行うことはできない．ステージの処理時間を均一化するため，キャッシュの導入は必然だったといえる．キャッシュの導入により，メモリアクセスステージが1または2クロックという固定クロック数で処理できるようになった．

RISCのパイプラインは，コンピュータアーキテクチャの有名な教科書で学ぶことができる．それが，ヘネシーとパターソンによる『コンピュータアーキテクチャ』（通称ヘネパタ本）である．この教科書では，仮想的なMPUとしてDLX（デラックスと発音する）というMPUを定義し，そのパイプラインとして次の5ステージの処理が提案されている．もっともDLXはMIPSのR2000/R3000と非常に近い（同じ?）構造をしており，以下はR3000のパイプラインそのものということもできる．ただし，ヘネパタ本ではメモリがキャッシュであることを特に強調してはいない．

● RISCのパイプライン処理

RISCのパイプライン処理を**図2**に示す．パイプラインがスムーズに動作する場合は，全ステージ数と同じ数の命令が（理論的には）同時実行できる．

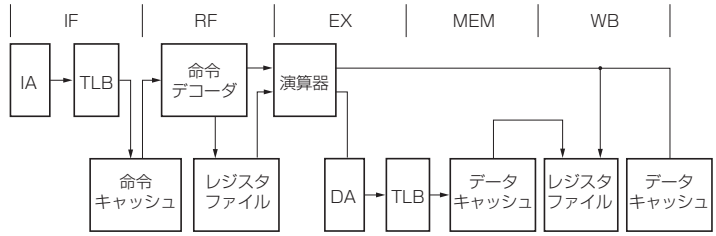
(1) 命令フェッチ（IF）
命令キャッシュから命令を取り出す．

(2) 命令デコード（RF）
フェッチした命令をデコードする．同時にレジスタオペランドをフェッチする．

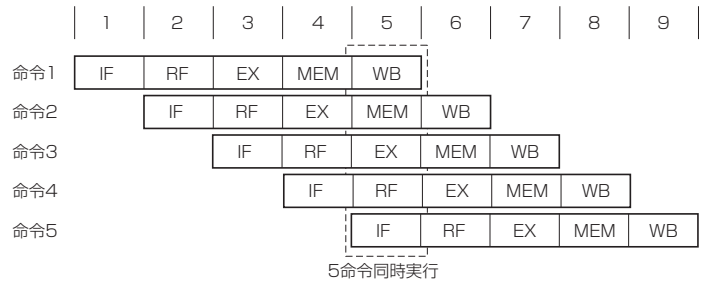
(3) 命令実行（EX）
デコード結果とフェッチしたレジスタの値を基に命令を実行する．ロード/ストア命令の場合は実効アドレスの計算を行う．分岐命令の場合は分岐先アドレスを計算する．

(4) オペランドフェッチ（MEM）
EXステージで計算したアドレスに対応するメモリの値をデータキャッシュからリードする．

(5) ライトバック（WB）
EXステージで計算した結果，またはMEMステー



（**a**）ステージと機能ブロックの関係



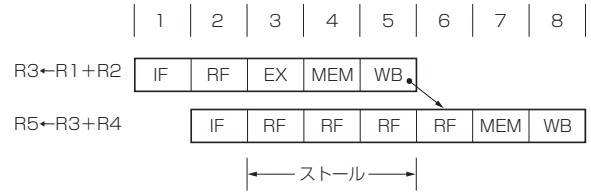
（**b**）スムーズなパイプラインの流れ

図2
RISCのパイプライン処理

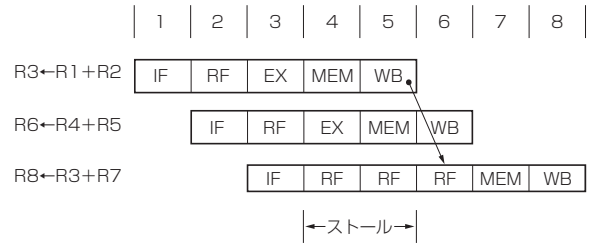
ジでフェッチしたオペランドをレジスタに格納する．ストア命令の場合はデータキャッシュにライトする．

上のパイプラインではアドレス変換のステージがないが，これはIFまたはMEMステージに先立って行われる．この詳しい説明は後半で解説するR3000のパイプラインの実際の項に譲る．

RISCのパイプラインの特徴は，アドレス計算をする専用のステージがなく，EXステージで代用している点である．このため，アドレス計算用の演算器と命令実行用の演算器（実際は加算器）をそれぞれ別個に用意する必要はない．これはRISCの「ロード/ストアアーキテクチャ」という特徴に由来する．つまり，一つの命令では2回加算を行うことがない，1命令で1回だけ演算器を使用するという制限の下で，レジスタとレジスタ間の加算，または，アドレス計算（ロード/ストア命令）は別の命令に分かれて定義されている．



（**a**）前の命令の結果（R3）を直後の命令で使用する場合



（**b**）前の命令の結果（R3）を2命令後の命令で使用する場合

図3　RAW（Read After Write）**ハザード**

● データハザードとフォワーディング

パイプラインの処理が乱れるハザードは，RISCのパイプラインでも発生する．それを詳しく見ていこう．まずはレジスタの依存関係に起因するハザードである．レジスタ間のリード/ライトの前後関係で，次の4種類が考えられる．

(1) RAW (Read After Write) ハザード

これは，レジスタライトの完了前に後続命令によって同一のレジスタをリードしようとした場合に生じる（**図3**）．

(2) WAR (Write After Read) ハザード

これは，レジスタから値をリードする前に後続命令によって同一のレジスタにライトをしようとした場合に生じる．

(3) WAW (Write After Write) ハザード

これは，同一レジスタへのライト順序が狂う場合に生じる．

(4) RAR (Read After Read) ハザード

一応挙げたが，レジスタへの変更がともなわないので，このようなハザードは存在しない．

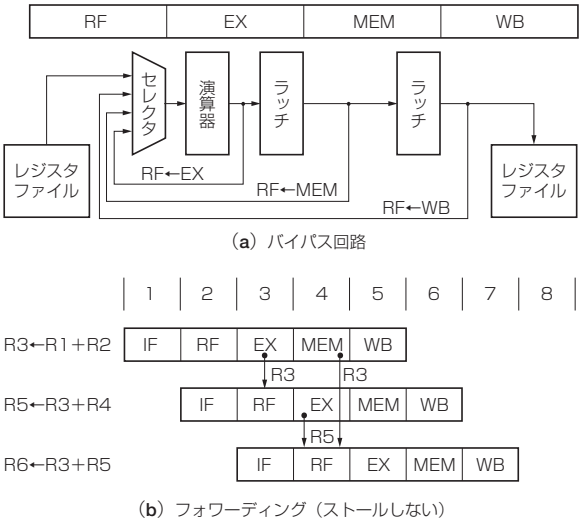


図4　バイパス回路とフォワーディング

以上はデータに起因するハザードなので，総称してデータハザードと呼ばれる．しかし，(2) および (3) のハザードは命令の実行順序が狂わない限り発生しない．通常のパイプラインでは発生しないが，スーパースカラ構造では発生することがある．これは後半で説明する．

当面の課題は (1) のRAWハザードである．これは，フォワーディング，バイパス，または，ショートサーキットと呼ばれる手法で解決可能である．つまり，EX，MEM，WBステージからRFステージへのバイパス回路を設けることで解決できる（**図4**）．RISCでは，パイプライン処理を乱さないために，フォワーディングはなかば常識である．

しかし，パイプラインのステージ数が多い場合，具体的には，レジスタをフェッチするステージ（RF）とレジスタへの書き込みステージ（WB）の間の段数が多いと，各ステージからRFステージへのバイパス経路がその段数分必要なので，実行ステージ（EX）へ与えるデータのセレクタが巨大になってしまう．これはもちろん動作周波数にも影響を与える．どの程度フォワーディングを行うかは悩むところである．

● ロード遅延と遅延ロード

ロードした値を直後の命令で使用する場合を考える．この場合，MEMステージで値が初めて確定する．このとき後続命令はEXステージにあるのでフォワーディングは不可能である［**図5**（**a**）］．なにも対処しないと変更前のレジスタの値をフェッチしてしまう．この待ち時間を**ロード遅延**という．

このため，プログラムの意図どおりに命令を処理するには，パイプラインの**インターロック**が必要となる．インターロックとはハザードの有無をテストし，ハザードがある場合はハザード原因が解決するまでパイプラインを停止する機構である．

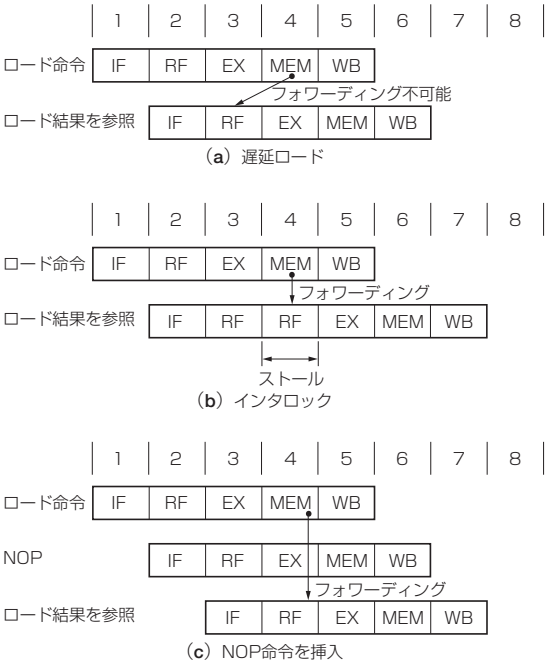


図5　遅延ロード

また，停止しているサイクルを**パイプラインストール**（パイプラインバブル）と呼ぶ．**図2**で示す5ステージ構成のパイプラインなら1クロックストールさせればよい［**図5**（**b**）］．

パイプラインのストールは，処理性能の低下を意味する．それを回避する手法の一つは，プログラムの意図を損わない範囲で命令の順序を入れ替えることである．いまの場合，1クロック分（1命令分）待ち合わせればいいので，ロードした値を参照する命令と後続の無関係な命令を入れ替えればよい．

入れ替えるべき適当な命令がない場合は，NOP命令を挿入することになる［**図5**（**c**）］．この手法はデータハザードの回避にも有効である．このような命令入れ替えや命令挿入を，**命令スケジュール**と呼ぶ．

RISCのアセンブラは命令スケジュールを当然のように行っている（禁止の設定も可能）．つまり，アセンブラが「勝手に」最適化するので，プログラマが書いたとおりの順序でコード生成が行われるとは限らないのである．この事実を知ったとき，筆者は少々衝撃を受けたが，今では慣れてしまった．

RISCは制御構造の単純化を目標としているから，インターロックは歓迎すべきものではない．ロード遅延をそのまま許し，アセンブラによる命令スケジュールによってのみストールを回避しようという考えがあ

Chapter 5

仮想記憶 / メモリ保護機能を実現するために

MMU の基礎と実際

ここではWindowsやLinuxなど，仮想記憶を使う場合に必須となるMMUについて解説する．通常は仮想記憶を使わないことの多い組み込み用途であっても，信頼性の高いシステムを構築するためにMMUのメモリ保護機能を使う場合もある．ここでは，アドレス変換，TLB（Translation Look-aside Buffer），PTE（Page Table Entry），メモリ保護機能について解説したあと，現代のMMUの基礎である680x0系やx86系，MIPSやPowerPCのMMUについて解説する．

MMUとはMemory Management Unitの略語である．つまり，メモリ管理ユニットのことで，MPUの外部または内部にあって仮想記憶機能を実現する．単にC言語などでプログラミングするだけなら，仮想記憶の知識などはほとんど必要ない．しかし，プログラムサイズが一昔前に比べてはるかに巨大化しており，またマルチタスクが当然のように行われている昨今，その裏方にはMMUという「働き者」がいることを心に留めておいてほしい．

1 仮想記憶とは

● 仮想的に広大なメモリを用意する

その昔，まだメモリが高価だった頃，コンピュータに実装できるメモリ容量はわずかなものだった．時としてアプリケーションプログラムの容量は実際の物理メモリの容量を超え，そのようなプログラムを動作させるためにはアプリケーションプログラム側で細工する必要があった．

プログラムの性質として見ると，ある瞬間瞬間に実行されているのは全体の一部分にすぎない．そこで，プログラムをいくつかのブロックに分割し，必要な部分だけをメモリにロードして実行させ，不要になったらそのブロックを補助記憶装置（多くの場合，ハードディスク）に退避し，代わりにほかの必要なブロックを補助記憶装置から取り出して，新しいブロックと入れ替えるしくみが必要になる（**図1**）．しかし，実装されている物理メモリの容量を考慮しながらプログラミングをするのは効率的でないし，物理メモリの容量が変化すると，同じプログラムが使用できなくなってしまう．

そこで，このようなメモリ管理をOSに任せるしく

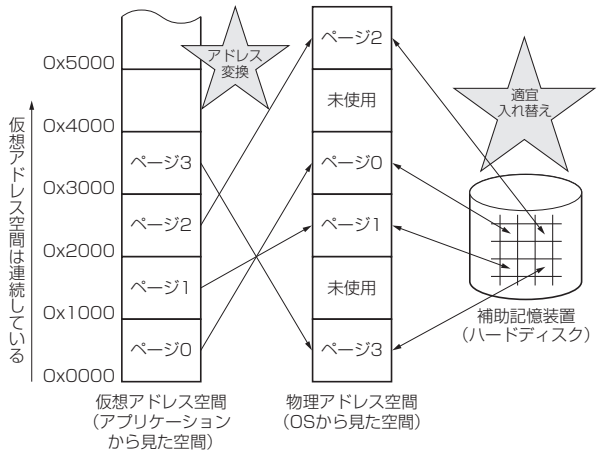


図1
仮想記憶のイメージ

みが考案された．これが仮想記憶の原点である．仮想記憶を利用すると，ユーザーは物理メモリを意識することなく，物理メモリの容量を超えるような巨大なプログラムを実行できる．

● マルチタスクを実現する

PCはもとより，現在では規模の大きな組み込み機器は，そのほとんどがマルチタスクで動作している．マルチタスクとは，複数のタスク（プログラム）を同時に物理メモリに置き，ある決められた順番に少しずつ（その多くは時分割で）実行していくものである．この場合，各タスクが必要とする全部の領域を物理メモリに割り当てようとすると，メモリに入りきらなくなってしまう．物理的に限られた容量しかないメモリを，多くのタスク間で分割して使用する手段が必要である．この場合も仮想記憶が有効である．そのため，仮想記憶といえば，現在ではマルチタスクを実現する手法として紹介されることが多い．

マルチタスクも，物理メモリを複数のブロックに分けて，そのブロックを各タスクに割り当てて実行させることで実現される．このような仮想記憶を行う場合，各タスクが自身に割り当てられた物理メモリのブロック以外をアクセスしないように保護する機能も必要になってくる．

● 現在ではページング方式が主流になっている

仮想記憶の方式としては，大きく分けて，**セグメント方式**と**ページング方式**がある．現在ではページング方式が主流なので，ここではページング方式を主体に話を進める．セグメント方式についてはコラム2で言及する．

ページング方式の場合，タスクのアドレス空間を分割したブロックを**ページ**と呼ぶ．また，不要になったページを補助記憶装置に退避したり，必要なページを補助記憶装置から復元する作業を**ページスワップ**と呼ぶ．

メモリのアクセス速度に比べてハードディスクのアクセス速度は非常に遅いので，ページスワップが頻繁に発生すると，プログラムの実行速度は低下する．しかし，プログラムとデータにはある程度局所性があるため，ページスワップがあまり発生しないことを期待して仮想記憶が実現されている．ところが，頻繁にページの範囲を超えて分岐が発生するプログラムや不連続な大量のデータを参照するプログラムでは，ページスワップの発生する確率が高くなる．このような場合は，そのタスクのページサイズを大きくすることで，ある程度ページスワップを回避できる．このため，MPUによってはタスクごとにページサイズを可変にできるようになっている．

2 アドレス変換

● アドレス変換とは？

PCにおけるプログラミングにおいて「このプログラムは物理アドレスの何番地に割り当てられるから」などと考えてプログラムを作る人は（OS屋などを除き），まずいない．誰もが，自分の書いたプログラムは，たとえば「0番地から配置され無限の容量をもっている」と考える．つまり，プログラムはそれぞれ固有のアドレス空間をもっている．マルチタスクを行うということは，重複するアドレス空間をもつ複数のプログラム（タスク）を同時に物理メモリに割り当てて実行するということである．このような操作を可能にするためには，プログラムの中で想定されているアドレスを，実際の物理メモリに配置するためのアドレスに読み替えるしくみが必要になる．これが，**アドレス変換**である．

プログラムが想定しているアドレスは**仮想アドレス**（論理アドレスともいう）と呼ばれ，物理メモリに割り当てられるアドレスを**物理アドレス**（実アドレスともいう）と呼ぶ．アドレス変換とは，仮想アドレスを物理アドレスに変換する作業のことである．プログラムの仮想アドレス空間は，一定の容量をもつページに分割される．このページ単位に，仮想アドレスから物理アドレスの変換が行われる（**図1**）．ページのサイズ（容量）はOSによってまちまちである．昔は1ページのサイズが2Kバイトのものが多かったが，現在は4Kバイトのものが多いようだ．4Kバイトは16進数で表現すれば1000バイトである．私見ではあるが，人間にとってなんとなくきりのいい数値なので，OS屋さんに好まれるのであろう．

それはともかく，仮想アドレスと物理アドレスの対応は，物理メモリ上に置かれたアドレス変換テーブルによる．この変換テーブルはページテーブルと呼ばれ，通常4バイトまたは8バイト長のエントリの集まりである．これをとくにページテーブルエントリ（PTE）と呼ぶ．32ビットOSの場合，アドレスは32ビットで表現されるので，PTEには最低でも1ページあたり32ビット（4バイト）の領域が必要である．

もっとも，仮想アドレスと物理アドレスは，同一ページ内のオフセット（1ページが4Kバイトの場合はアドレスの下位12ビット）は一致するので，必要なビット数はもう少し少なくてよい．しかし実際には，そのページの保護情報のための情報やページスワップのための情報も必要になるし，ワード長（4バイト）またはダブルワード長（8バイト）のほうが（OSの）プログラムで扱いやすいので，一つの仮想アドレスに対して4バイトまたは8バイトのPTEが用いられるのが普

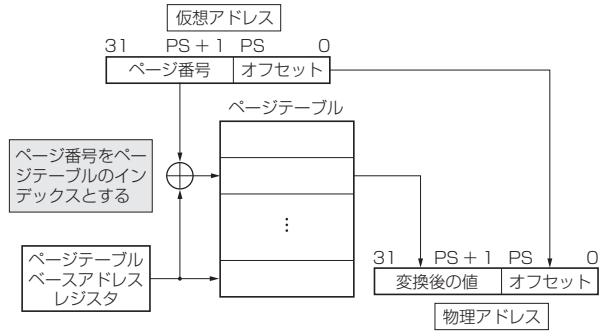


図2
1レベルのアドレス変換

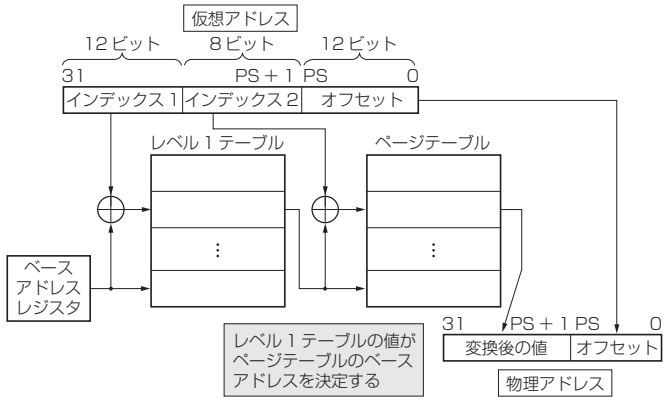


図3
2レベルのアドレス変換

通である．

● アドレス変換のレベル

さて，仮想アドレスが32ビット，1ページが4Kバイトの場合を考えよう．この場合，仮想アドレスの下位12ビットがページ内オフセット，上位20ビットがページ番号になる．このページ番号をインデックスとしてページテーブルを参照すれば，そのページに対応する物理アドレスを取り出すことができる．

なお，ページテーブルのベースアドレスはタスクごとに固有な値をもっていて，コンテキスト（タスクを性格づける情報）の一部である特権レジスタに格納されている．**図2**に仮想アドレスから物理アドレスを得る変換作業の概念図を示す．この図では20ビットのインデックスでページテーブルを参照するので，PTEの数は1M個必要である．PTEの容量は4バイトまたは8バイトなので，1タスクあたり4Mバイトまたは8Mバイトの物理メモリの容量がページテーブルのために必要になる．

しかし，タスクのもつアドレス空間は32ビット（4Gバイト）のすべての領域を使っているわけではなく，命令，データ，スタックなど，性質の異なる領域ごとにある程度塊になって存在している．このような場合，1M個のページテーブルエントリをすべて用意するのは不経済である．へたをしたら物理メモリがページテーブルだけであふれてしまうという状況も起こりかねない．そこで，ページテーブルを多段階に分けて参照する方法が考えられている．

この方式では，仮想アドレスのページ番号をさらにいくつかの領域に分ける．たとえば，20ビットのページ番号を上位12ビットと下位8ビットに分ける．この場合，上位12ビットをインデックスとして1段目のテーブルを参照し，2段目のテーブル（これがページテーブル）へのベースアドレスを獲得する．そして，下位8ビットをインデックスとして2段目のテーブルを参照し，物理アドレスを獲得する．この概念図を**図3**に示す．**図2**ではページテーブルを直接参照しているので1レベルのページング，**図3**では2回目でページテーブルを参照しているので2レベルのページングと呼ばれる．

最近のMPUでは，2レベルのページングでアドレス変換を行うことが主流だが，MC68030や68040では3レベルのページングを行うこともできる．

3 TLB

● TLBとは？

MPUが仮想記憶モードで動作している場合，仮想

アドレスから物理アドレスへの変換を，いちいち物理メモリ上のページテーブルを参照しにいっていたのでは，その処理が命令実行のボトルネックになってしまう．それを避けるために，MPUは内部に**TLB**（Translation Look-aside Buffer）と呼ばれる変換テーブルをもっている．日本語ではアドレス変換緩衝機構と訳されることが多い．モトローラは**ATC**（Address Translation Cache），つまり，アドレス変換キャッシュと呼んでいる．その名のとおり，TLBとは，PTEをチップ内にキャッシュしたものである．

MPUはアドレス変換を行うとき，まずTLBを参照し，そこに目的の仮想アドレスと物理アドレスのペアが格納されていれば（TLBヒット），その物理アドレスを用いて命令を処理する．もし該当する仮想アドレスがTLB内になければ（TLBミス），物理メモリ上のページテーブルを参照しに行き，その値をTLBに登録する．また，TLBにはPTEと同様にメモリ保護などの情報が格納されており，TLB参照の際に不正アクセスがないかどうかチェックする．もし不正なアクセスである場合は，メモリ保護例外を発生する．以上がMMUの機能である．

ただし，最近のRISCチップでは，TLBを参照したとき，仮想アドレスが登録されていないと直ちに例外を発生して，TLBの内容を入れ替える処理をOSのプログラムに任せる．何度もメモリ上のテーブルを参照してTLBの内容を更新する処理は，実現が複雑であり，メモリアクセスはロード/ストア命令だけというRISCのポリシにも反する．何よりもパイプライン動作が妨げられてしまう．このためRISCでは，TLBの機能そのものがMMUの機能ということもできる．

● TLBの構造（連想方式）

TLBとは，仮想アドレスをタグとして内容を参照し，一致するタグがあれば対応するデータを物理アドレスとして出力する一種のキャッシュメモリである．その構造は参照の仕方により，次の3種類に分類できる．

- フルアソシアティブ方式
- ダイレクトマップ方式
- *n*ウェイセットアソシアティブ方式（$n \geqq 2$）

▶フルアソシアティブ方式

フルアソシアティブ（Full Associative）方式は，TLBのエントリ数の数だけ異なる仮想アドレスを格納できる方式である．ほかの方式とは異なり，各エントリに格納される仮想アドレスに制限はない．通常は連想メモリという特殊なメモリで構成されるため，LRU処理（詳細は後述）が複雑になるため，多くのエントリをもたせることができない．現在の技術では50エントリ程度が限界と思われる．ただし，実装されているエントリをむだなく使用することができるので，少ないエントリ数でも高いヒット率（仮想アドレスを参照したとき，TLB内に存在する確率）を得ることができる．**図4**にフルアソシアティブ方式のTLBの構成を示す．

▶ダイレクトマップ方式

ダイレクトマップ（Direct Mapped）方式は，もっとも単純な方式である．仮想アドレスが決まると，その仮想アドレスで参照するエントリが一意に決まってしまう．たとえば，256エントリのダイレクトマップ方式のTLBを参照する方法として，仮想アドレスのビット19～12（8ビット）を使用してエントリをインデックスする方法が考えられる．これは，ページサイズが4Kバイトの場合である．仮想アドレスのビット31～12がページ番号を表し，その下位8ビットである．

8ビットのデータは256種類を識別できるので，仮想アドレスとTLBのエントリを1対1に対応させることができる．

ただし，この場合，下位8ビットが一致する仮想アドレスは異なるアドレスであっても同一のTLBエントリが参照されてしまう．プログラムの仮想アドレスが256通りでまんべんなく変化することは稀なので，場合によっては一度も参照されないエントリが存在する．逆に同じエントリが何度も参照され，前のデータを書き潰してしまうおそれもある．

ダイレクトマップ方式は，構造は単純でエントリ数を多くもたせることができるが，エントリ数を多くしないと高いヒット率は期待できない．**図5**にダイレクトマップ方式のTLBの構成を示す．

▶*n*ウェイセットアソシアティブ方式

***n*ウェイセットアソシアティブ**（*n*-way Set Associative）方式とは，ダイレクトマップ方式の改良版である．ダイレクトマップ方式のエントリを*n*系統用いて構成する．この方式も，LRU処理の制限から*n*の値は2また

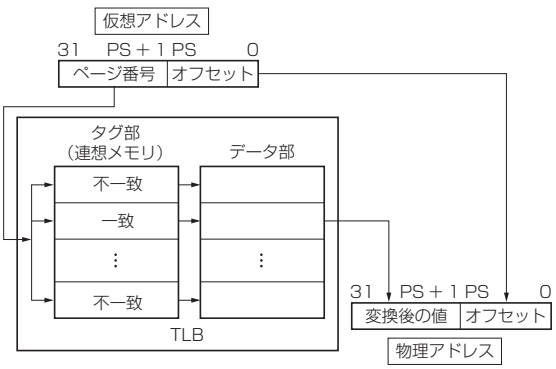


図4　フルアソシアティブ方式

Chapter 6

外的要因と内的要因，ハードウェア割り込みとソフトウェア割り込みの違いを理解する

割り込みと例外の概念とその違い

割り込みには，MPUの動作とはまったく非同期に外部のデバイスが要求するハードウェア割り込みと，プログラム中に明示的に分岐命令を記述するソフトウェア割り込みがある．また，プログラムの実行結果によって発生する予期しない事象を例外と呼ぶ．ハードウェア割り込みは外的要因で発生するが，ソフトウェア割り込みと例外はMPUの内的要因で発生する．例外と割り込みの区別はそれぞれのMPUアーキテクチャ上の決め事であり，その本質は同じと考えられる．

以前，筆者は割り込みというものの概念がよくわからなかった．

MPUは与えられた処理を順次こなしていく．その処理に割り込んで，いったい何をするのか．処理Aをこなしながら処理Bも行う必要があるのなら，AとBを同時に実行するようにプログラムすればよいではないか．

例外についても同様に，言っていることは固定アドレスに分岐して戻ってくることである．それはサブルーチンコールと何が違うのか．

以降の解説は，経験を積んで筆者が感じ取った割り込みと例外の意義やしくみである．

1 MPUにおける割り込みと例外

● 割り込みとは何か

割り込みとは，一連の仕事をしているときにその仕事を中断させて別の仕事をさせることである．割り込みされる側からすると，予期しないタイミングで発生するのが特徴である．

MPUでアプリケーションプログラムを実行する場合，通常は割り込みを意識しない．割り込みが発生するとそれまでの処理は中断され，特定の割り込み処理を行って元の処理に復帰する．アプリケーションプログラム側は割り込まれたことに気付かない（**図1**）．

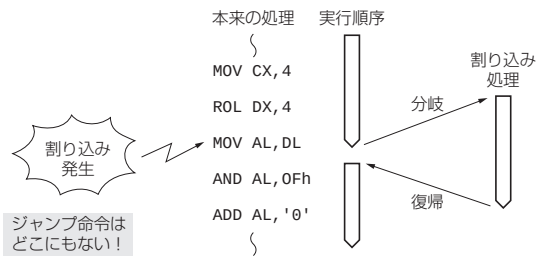


図1　割り込み処理の概念（ハードウェア割り込み）

MPUのプログラム実行順序としては，**図1**のように一筆書き状態の順番でプログラムを実行しているにすぎないが，人間の時間感覚で見ると，本来の処理と割り込み処理が平行に実行されたように見える．本来のプログラムが気付かないうちに並行動作が行われる…ここに割り込みの本質がある．

● ハードウェア割り込みとソフトウェア割り込み

割り込みは大きく分けて，MPUに接続された外部のデバイスが要求する**ハードウェア割り込み**と，プログラムで明示的に要求する**ソフトウェア割り込み**の二つがある．

ハードウェア割り込みとは，**図1**のように外部からの要因でジャンプ命令などを使わずにプログラムの実行を分岐することである．ハードウェア割り込みはアプリケーションプログラムには見えない．外部のハードウェアの状態が変わったことを検出し，それにしたがって処理が必要な場合に利用する．一般的に，外部割り込みはMPUの処理とは非同期に行われる．

一方，ソフトウェア割り込みは，割り込み処理へ切り替える命令をアプリケーションプログラム中に明示的に記述する（**図2**）．この意味で，ソフトウェア割り込みはサブルーチンコールのようにもみえる．たいていのMPUには，ソフトウェア割り込みを発生させるためのトラップ命令やシステムコール命令が用意されている．

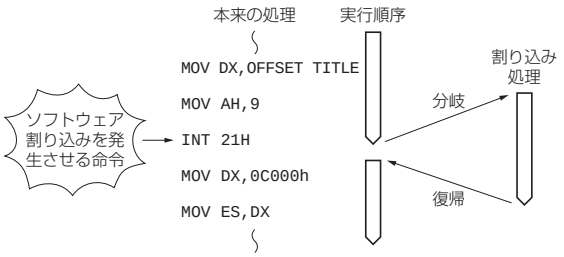


図2　ソフトウェア割り込み

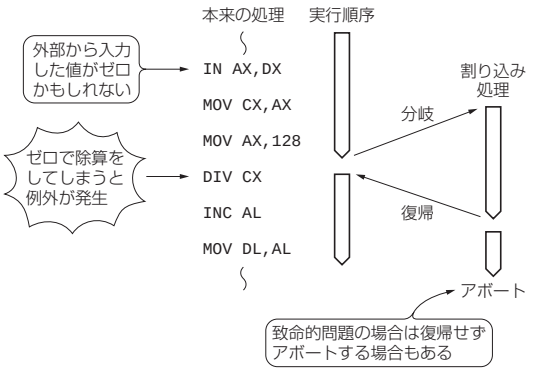


図3　例外の概念

● 例外とは何か

一般的に割り込みは，プログラムの実行とは無関係（非同期）に発生するが，プログラムの実行結果によって発生する予期しない事象がある．たとえば，ゼロ除算，オーバフロー，アドレスエラー，ページフォルト（TLBミス）などである．これらの発生によってもプログラムの処理は中断され，それら予期しない状態を処理するプログラムが実行される（**図3**）．

これらは，要因がプログラムの実行そのものにあり，外部からの要因によって割り込まれたわけではないので，とくに**例外**と呼ぶ．

「例外」を辞書で引くと，「通例の原則にあてはまらないこと．一般の原則の適用を受けないこと．また，そのもの.」とある．コンピュータの世界でもイメージは同じだが，命令の処理が通常と同じようには終了しない事象を表す．

どのような事象が発生したときに例外となるのかは，MPUのアーキテクチャによって異なる．たとえば，定義されていない命令コードを実行すると，あるアーキテクチャでは例外となるが，あるアーキテクチャでは**NOP**と同じ動作となり，そのままプログラムを実行し続ける．

● 割り込みと例外の区別

要因発生後の動作，つまり割り込み処理へ分岐する動作は，割り込みも例外も共通である．しかし，割り込みの場合は元のプログラムに復帰するのが前提であるが，例外は場合によっては，致命的な事象と判断してプログラム処理を中止（アボート）することもある．

事象発生後の挙動が同じという点で，割り込みと例外は言葉のうえでの区別のようにも思える．実際，割り込みと例外を同一視するアーキテクチャのMPUも多い．そのような場合，外的要因によるハードウェア割り込みを**外部割り込み**，内的要因による例外とソフトウェア割り込みを**内部割り込み**と呼んで区別する．

割り込みと呼ぶか例外と呼ぶかは，そのMPUのアーキテクチャ上の決め事である．ここでは原則として，外的要因によるものを割り込み，内的要因によるものを例外として話を進める（とはいえ，「ソフトウェア例外」とはあまり呼ばないが…）．

● ベクタとハンドラ

割り込みが発生すると割り込み処理へ分岐するわけだが，どこへ分岐するかを示すものを**割り込みベクタ**と呼ぶ．そして，割り込み処理ルーチンのことを，**割り込みハンドラ**と呼ぶ．また，割り込みと呼ぶか例外と呼ぶかに対応して，ベクタとハンドラも，割り込みベクタ，割り込みハンドラ，例外ベクタ，例外ハンドラと呼ばれる．

● 割り込みの受け付け，NMIとリセット

割り込みとは，本来の処理の途中で別の処理を行わせることだが，処理の内容によっては，実際に連続して実行しないと意味をなさない場合や，途中で割り込み処理が実行されては都合の悪い場合もある．そのような場合は，割り込みの受け付けを禁止することもできる．

しかし，外的要因の中には非常に緊急性を有する事象もある．もしそれが発生した場合は，割り込まれると都合の悪い処理であっても，その緊急の割り込み処理を実行する必要があるだろう．このような重要な割り込みは，割り込み受け付け禁止ができない割り込みとして**ノンマスカブル割り込み**（Non Maskable Interrupt，NMI）を使う．通常，割り込みと呼ぶ場合は，ソフトウェアで割り込みの受け付けを禁止することができる**マスカブル割り込み**のことをいう．

MPUのアーキテクチャによっては，リセットも割り込みや例外に分類するものがある．割り込みベクタがプログラマブルなMPUであっても，さすがにリセット時は特定のアドレスから実行を開始したり，特定アドレスのメモリを読み込み，その値をアドレスとして実行を開始したりする（リセットベクタ）．また，ノンマスカブル割り込みという意味では，リセットも

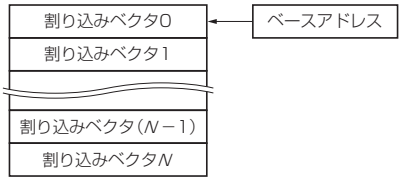


図4　割り込みベクタテーブル

ノンマスカブルな割り込みといえる．しかもNMIよりも優先度が高く，MPUの中ではもっとも優先度の高い割り込みといえる．

● 割り込みベクタテーブル

CISC系MPUの多くは，割り込みや例外に対する割り込みベクタの値，つまり割り込みハンドラのアドレスを自由に設定することができる．その割り込みベクタをある決められた順序でメモリ上に並べたものを**割り込みベクタテーブル**と呼ぶ．

多くの場合，割り込みベクタテーブルのベースアドレス，つまり先頭の割り込みベクタが格納されているアドレスは物理アドレスの0番地である．MMUをサポートするMPUでは，この割り込みベクタのベースアドレス（物理アドレスで指定する）を変更可能な場合が多い．そのため，割り込みベクタのベースアドレスを保持する特別なレジスタが用意されている．このベースアドレスレジスタの値を変更することで，割り込みベクタテーブルを任意のアドレスに配置することができる（**図4**）．

一方，RISC系MPUの多くは，割り込みベクタの値がアーキテクチャで一意に決められているので，割り込みベクタテーブルというものは存在しないことが多い．さらに，割り込みベクタの値は仮想アドレスだが，対応する物理アドレスは1対1で決まっている（たとえば，アドレス変換されない）ことが多い．

2 外部割り込みと例外の動作の概要

ここでは，ハードウェア（外部）割り込みと例外の動作について解説する．以降ではとくに明記しない限り，ハードウェア割り込みを単に「割り込み」と示す

Column1　ソフトウェア割り込みとサブルーチンコール

ソフトウェア割り込みは，トラップ命令やシステムコール命令などのように，プログラムで明示的に記述して積極的に発生させる割り込みである．ソフトウェア割り込みは，OSが提供するサービスを得るためのシステムコールのインターフェースとして利用される．意味的にはサブルーチンコールと大差はない．それではなぜ，ソフトウェア割り込みというわずらわしい（わけでもないが）手順を踏むのであろうか．それには少なくとも二つの理由がある．

一つは実行レベルの問題である．WindowsやLinuxでは，ユーザープログラムはMPUの提供するユーザーモードで実行されている．それに対して，OS内部はカーネルモードで実行される．通常のサブルーチンコールでは現在の実行レベルを保持するので，ユーザープログラムからコールしたサブルーチンでは特権命令を実行できない．ソフトウェア割り込みによって，実行レベルを特権レベルに上げることができる．

二つ目はコールするアドレスの問題である．WindowsやLinux上のユーザープログラムは，基本的にすべて仮想アドレス上で動作する．一方，OSのサービスルーチンの先頭アドレスは一意に決まっている．その先頭アドレスを明示的にユーザープログラムで指定するには，仮想アドレスがどの物理アドレスに変換されるのかを知る手段がない以上，一般には不可能である．割り込みベクタテーブルは，通常，システムに一つだけ存在するので，OSのサービスを割り込みハンドラで指定するようにすれば，すべてのタスクから同じOSのサービスルーチンを実行できてむだがない．

歴史的にながめれば，保護やアドレス変換がない昔のMPUでは，システムコールがサブルーチンコールによって行われていた．これは仕方のないことである（というか，それ以外の方法はなかった）．しかし，比較的新しいところでは，OS/2でもシステムコールをサブルーチンコールで実現していた．その当時，すでにMS-DOSではシステムコールに`INT`命令を使用していたので，OS/2は先祖返りといえなくもない．なぜ，そのようなしくみを採用したのか，IBMの見解を聞いてみたいものである．OS/2を動作させるMPUが，アドレス変換がまだ洗練されてなかった80286だったことが一因かもしれない．

面白いところでは，Windows CEや一部のLinuxのシステムでは，システムコールにアドレスエラーを利用している．MPUにはトラップ命令やシステムコール命令が用意されているのに，なぜこうなっているのかは謎である．

ことにする．ソフトウェア割り込みについては，コラム1を参照してほしい．

● 割り込まれるプログラムの影響

割り込みや例外は，割り込まれるプログラム側からすれば，意図しない場所で秘密裏に処理される．このときの動作はどうなっているのだろうか．まず，プログラムの実行を規定する要因を考えよう．ある瞬間のプログラムを完全に再現するには，

- プログラムの命令コードとデータ
- プログラムでアクセス可能なすべてのレジスタの値
- PC（プログラムカウンタ）の値
- SR（ステータスレジスタ）の値

といったデータが一意に定まっていればよい．これらの情報をコンテキストと呼ぶ．ここでPCは，いうまでもなく，現在実行している命令コードのアドレスである．SRは，PSW（Program Status Word）やPSR（Program Status Register）とも呼ばれ，条件分岐用の条件フラグや実行レベルが含まれる（x86でいうところのFLAGレジスタ）．

これらのうち，プログラムの命令コードとデータは，そのプログラムの実行が終了するまで物理メモリまたは補助記憶上に存在しているので，とくに気にする必要はない．レジスタの値は壊されると困るので，割り込みハンドラでは，そこで使用するレジスタの値をスタックなどに退避しておき，例外ハンドラを抜けるときに退避しておいた値を書き戻してやればよい．レジスタは割り込みハンドラで使用しないこともあるが，PCとSRの値は必ず変更される．つまり，PCとSRがプログラムの挙動を性格付ける．

結論として，各レジスタやPCとSRを割り込み処理の前に保存し，割り込み処理を終了した後で元に戻してやれば，割り込まれたプログラムは何も知らずに処理を継続することができる．

● 割り込み/例外発生時の動作

実際に，割り込みや例外が発生したときのMPU内の動きについて見てみよう（**図5**）．多くのMPUでは，割り込みや例外が発生すると，PCとSRを自動的に特定の場所に退避するようになっている．また，外部から割り込みアクノリッジ（ベクタ）を読み込むMPUもある（詳細は，実際のMPUでの動作の項目で説明）．

CISC系MPUでは，割り込みや例外が発生するとPCとSRを（割り込み用）スタックに退避し，割り込みからの復帰を指示する命令（**RETI**など）を実行すると，スタックからPCとSRの元の値を取り出して，新たにPCとSRに設定し直す．

RISC系MPUでは，スタックアクセス（＝メモリアクセス）を行うと処理速度が低下してしまうので，退避専用の特殊レジスタに値を格納する．割り込みハンドラの終了を指示する命令は，PCとSRの値をこの特殊レジスタから取り出す．これらのレジスタは1組しか用意されていないのが普通で，多重に割り込みや例外が発生すると値が上書きされてしまう．多重に割り込みが発生する可能性がある場合は，スタックなりメモリなりに内容を退避する必要がある（RISCにもスタックという概念はある）．

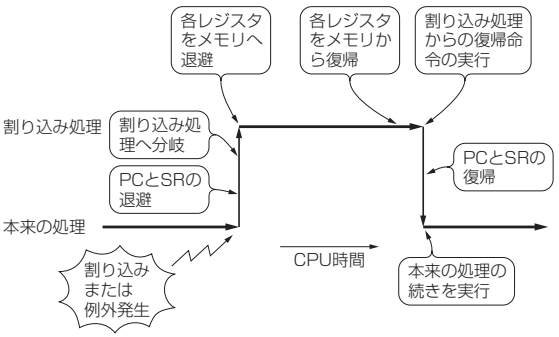


図5　割り込み/例外処理の動作の概要

割り込み発生前と割り込みハンドラからの復帰後で，プログラムが使用しているレジスタの値は保存されなければならない．このレジスタの退避/回復処理は大量のメモリアクセスを伴うので，性能低下につながる．それを避けるため，アーキテクチャによっては割り込みハンドラのみがアクセスできる，通常のレジスタとは独立したレジスタを提供していることもある．このような構造をレジスタバンクと呼ぶ．Armなどのアーキテクチャは，例外の種類ごとに数種類のレジスタバンクを備える．

また，割り込みからの復帰命令はMPUによって異なるが，だいたい次のような名称で呼ばれる．

RETI（RETurn from Interrupt）
RETE（RETurn form Exception）
IRET（Interrupt RETurn）
ERET（Exception RETurn）

この名称によって，そのMPUのアーキテクチャが割り込み/例外のことを，割り込み（Interrupt）と呼んでいるか，例外（Exception）と呼んでいるかを知ることができる．

● 割り込み発生と割り込みマスク

一般的なMPUでは，一度に一つの割り込み要求しか受け付けないようにするため，割り込み発生時には新たな割り込みの受け付けができなくなる．ソフトウェアによる割り込みや例外処理中に発生する割り込みは，割り込み処理が終了するまで待たされる．具体

Chapter 8

処理性能を上げるための最後の切り札

マルチプロセッサの基礎

MPU単体での性能向上に限界が見え始め，さらなる処理性能向上のために，複数のプロセッサを使って並列実行する方法（マルチプロセッサ）が考え出された．マルチプロセッサには，いわゆるプロセッサを複数個並べたものから，1チップの中にコアを複数個実装したものなどがある．マルチプロセッサ時のメモリ共有やキャッシュについても考察している．

2001年，PCの世界ではAMDのAthlonMPやIntelのXeonが発表され，これ以降，デュアルプロセッサ構成がにわかに脚光を浴びてきた．これらは二つのプロセッサをSMP形態で構成するマルチプロセッサである．マルチプロセッサは，従来はワークステーションやサーバなどのハイエンド専用だったが，現在ではデスクトップ用の可能性も見えてきている．つまり，AMDやIntelの製品系列にはデュアルプロセッサだけではなく，4CPU以上のマルチプロセッサも標準的に組み込まれている．

MPUの高速化技術はもはや出尽くした感があり，マルチスレッディングやマルチプロセッサが最後の手段と考えられる．

本章ではマルチプロセッサ構成の基礎について説明する．マルチプロセッサ構成自体には多くの研究がなされており，最新の成果を説明することは難しい．しかし，その基本となる知識はそれほど多くはない．それらについてみていってみよう．

なお，ここではスーパーコンピュータのような超並列構造には言及しない．

1 マルチプロセッサの基礎

● マルチプロセッサ

MPUの実行速度を上げる手法としてパイプラインやスーパースカラが考案されてきたが，それらはプロセッサ自体の高速化であり，それ以上に性能を追求する場合，単一プロセッサでは限界がある．そこで，タスクをいくつかのプロセスに分割し，それぞれを別個のプロセッサで並列に実行するという方式が考えられた．これがマルチプロセッサである．

RISCの産みの親であるStanford大学のHennessy総長は次のように言っている．「個別の技術でMPUの性能を向上する手法は行き詰まった．これからは，マルチプロセッサ構成のMPUを効率的に利用するソフトウェアの開発が性能向上のキーポイントである」と．もっとも，HennessyはR10000を発表した時点（1992年）で，性能向上の次の技術は1チップに複数のプロセッサを統合することだと主張していた．諦めるのが早すぎる感もあるが，それ以降登場してきた技術はチップマルチプロセッサとマルチスレッディング（ハイパースレッディング）のみ（その意味でHennessyの予言は的中している）であり，ここ10年以上飛躍的な技術革新がないのも事実である．

● マルチプロセッサの構成方式

一般に，タスクが異なれば，データ領域も異なる．特別な場合を除き，同一のメモリ領域をアクセスする場合の競合（順序の保証など）を考慮する必要はない．しかし，一つのタスクを分割しているプロセス（正確にはスレッド）間では共通のメモリ領域（変数領域）をアクセスすることは珍しくない．というより，メモリ領域はほとんどが共通といってもよい．

マルチプロセッサ構成では，複数のプロセッサが互いにアクセス可能な共有メモリを用いるのが普通である．このような構成を共有メモリ型マルチプロセッサ（Shared Memory Multi-processor）という．この共有メモリの構成方法によって，**図1**のように，マルチプロセッサの構成は2種類に分類できる．より一般的には，プロセッサと共有メモリを結合する方法は，共有バスとは限らず，相互結合網と呼ばれる．しかし，ここでは簡単のために，共有バスの場合を考える．

本書では共有メモリのあり方に注目して分類する．またマルチプロセッサには，密結合マルチプロセッサ

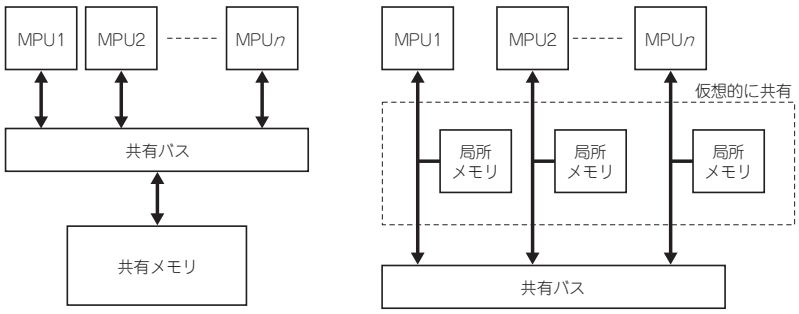


図1
マルチプロセッサの構成　（**a**）集中共有メモリ方式（SMP）　（**b**）分散共有メモリ方式

（Tightly Coupled Multi-Processor = TCMP）と疎結合マルチプロセッサ（Loosely Coupled Multi- Processor = LCMP）という分類もある．複数のプロセッサが一つのOSと一つのメモリを共有する形態がTCMP，個々のプロセッサがそれぞれのメモリとOSをもち（つまり独立したコンピュータシステムを形成し），入出力ポートなどを通じて通信（ネットワークや高速バスで結合）を行うのがLCMPである．大型計算機では，密結合マルチプロセッサを疎結合することで性能を向上させている．以下に説明するSMPはTCMPと同一視されることが多い．

▶集中共有メモリ方式

これは，プロセッサから共有メモリへのアクセス時間が一定という構成である．すべてのプロセッサが時間的空間的に対称なので，対称型マルチプロセッサ（SMP=Symmetric Multi-Processor）とも呼ばれる．

SMPにおいて，複数のプロセッサは同等なものとして扱われる．また，共有メモリはすべてのプロセッサから同一のアドレスでアクセスできる．共有メモリが一つしかないので，SMPはUMA（Uniform Memory Architecture）とも呼ばれる．UMAとは，本来は異なる属性のメモリを同一のメモリ空間上に配置するアーキテクチャである．現在では，グラフィックス用のフレームバッファをメインメモリの一部として確保する方式を指すことが多い．

SMPの利点としては，各CPUが簡単にデータを共有でき，並列のプログラムが簡単に書けること，逐次処理のプログラムやプロセスを処理するのに優れている点が挙げられる．逆に弱点は，同時に一つのプロセッサしか共有メモリにアクセスできないので，CPUの数が増えるにしたがいアクセス権の調停が難しくなることである．しかし，その簡易性から，ほとんどすべてのマルチプロセッサではこの方式を採用している．このため，多数のCPUをマルチプロセッサ構成にすることが難しく，現実には4CPU程度が限界となっている．

キャッシュ内蔵のCPUをマルチプロセッサ構成で使用する場合，コヒーレンシの問題が発生する．後で説明するが，UMAでは，各CPUがアドレスバスを常に監視（スヌープ）して，自身のもつキャッシュ内のデータにそのアドレスが一致するかどうかチェックして，そこを無効化またはデータ更新することでコヒーレンシを維持する．

SMPの反意語として非対称型マルチプロセッサ（ASMP=Asymmetric Multi-Processor）というものもある．これは，次に説明する分散共有メモリ方式を示す場合もあるが，OSのカーネル用，ユーザープログラム用というようにCPUの役割を区別する方式を示す場合もある．SMPとは異なり，ASMPの定義ははっきりしていない．

▶分散共有メモリ方式

これは，プロセッサごとに局所メモリをもつ構成である．プログラムの実行には参照という局所性があるので，この構成なら共有バスの転送量を低減できる．プログラムを容易にするために，局所メモリに連続したアドレスを割り付け，論理的に単一の共有メモリとしてアクセスできるようにする．UMAと対応して，NUMA（Non-Uniform Memory Architecture）と呼ばれる．

バス速度，アクセス競合の点で，高並列マシンの集中共有メモリ方式は現実的でない．この場合，分散メ

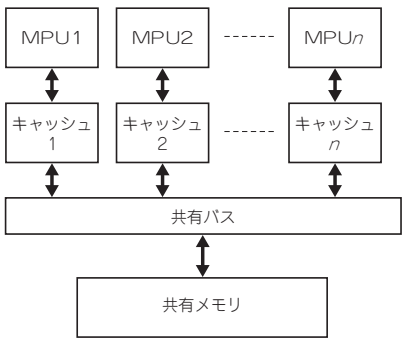


図2　マルチプロセッサとキャッシュ

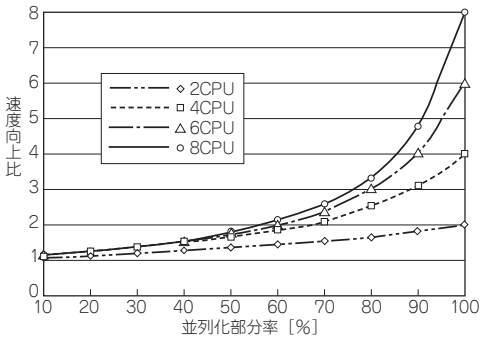


図3　アムダールの法則

モリ方式を採らざるをえない．現実としては，**図2**のように，SMP構成で各プロセッサにキャッシュをもたせる方式が一般的である．キャッシュはプログラムから意識されないため，SMPでありながら，バス速度，アクセスの競合の問題を解決できる．

NUMAにおいて，キャッシュのコヒーレンシを保証するシステムはccNUMA（Cache Coherent Non-Uniform Memory Architecture）と呼ばれる．たとえば，SunのEWSであるSunFire15Kでは，18組の4CPUからなるSMPモジュールをccNUMAで結合して性能を上げている．

NUMAでは，自分のメモリへのアクセス時間に比べて，他のCPUのメモリへのアクセスが遅くなる傾向があり，それが採用の妨げとなっていた．しかし，現在では技術革新により3～5倍かかっていたアクセス時間が平均で1.5倍，最悪でも3倍程度に縮まり，たいした問題ではなくなっている．

● 並列処理の割り当て

マルチプロセッサはOSの助けを借りて並列動作を行うことができる．CPUを複数並べただけでは性能向上は望めない．OSがプログラムをプロセスなりスレッドなりに分解して，各CPUに処理を割り当てて実行させるのである．

さて，SMPに対してOSがそれぞれのプロセッサに処理を割り当てる方法には2通りの方法がある．一つはプロセス単位での割り当て，もう一つはスレッド単位での割り当てである．

プロセスというのは一つのアプリケーションプログラムそのものであるから，これはアプリケーションプログラムごとに一つのプロセッサを割り当てるというイメージであろう．OSとしてはとくに特殊な処理は必要ない．この場合，各アプリケーションの実行が終了する時間は，それぞれのプログラムが一つのCPUで実行される場合と同じである．アプリケーションが複数ある場合には，それらが同時に実行されるので，すべてのアプリケーションの実行を終了する時間は早くなる．当然，アプリケーションが一つの場合には効果がない．この意味で，あるプロセスだけに限ってみれば，CPUが一つのときと実行速度になんら変わりはない．

スレッドというのは一つのプロセスを並列実行可能な部分に分割したもので，プロセスはいくつかのスレッドを寄せ集めたものである．たとえればこれは，複数のアプリケーションプログラムが複数のCPUで時分割に実行されるイメージである．スレッドの割り当ての運がよければ(?)，一つのアプリケーションの各部分が別個のCPUに割り当てられて実行されることになる．つまり，あるプロセスに限ってみれば，CPUがひとつの場合よりも高速に実行できることになる．

このように，マルチプロセッサ（SMP）において「処理が速い」というのには2通りの意味があるので注意が必要である．マルチプロセッサで速度を向上させようとしたらスレッド分割にせざるをえない．ところで，マルチプロセッサシステムでCPUの数をどんどん増やしていけばそれだけ処理速度が速くなるのか，という疑問がある．じつはそうならないことがアムダールの法則（Amdahl's Law）によって示唆されている（**図3**）．これは，

改良後の実行時間＝影響されない部分の実行時間
　　＋（影響される部分の実行時間／改良の度合）

というもので，上の式で「影響」を「並列実行」，「改良」を「プロセッサの個数」に当てはめればよい．つまり，どのようなアプリケーションプログラムでも並列に実行できない部分が存在する（前後関係の依存性がある）ので，並列化できる部分しか高速化できないというものである．並列処理を念頭に置く場合，性能向上は次の式で表される．

速度向上比＝$1/((1-P)+P/N)$
　　＝旧実行時間／新実行時間
P：並列化部分率（元の実行時間のうち，並列処理可能な実行時間の割合）
N：並列化度（プロセッサの個数）

2 マルチプロセッサのキャッシュ制御

● マルチプロセッサのキャッシュ

マルチプロセッサにおけるキャッシュは単一プロセッサと比べると複雑である．共有メモリのコピーがそれぞれのキャッシュに存在するが，共有メモリと各キャッシュ間で一貫性（コヒーレンシ，またはコンシステンシという）を保証する必要があるからである．システムにおいてコヒーレンシが保たれている状態をコヒーレントという．コヒーレントにおいては，ある

Chapter 10

MPUの新たな用途

AIチップの概要

AI専用チップに注目が集まる背景

AIチップの歴史

● ディープラーニングの元祖は1980年代から

歴史をたどると，1980年代には既に（論理学の観点ではなく），脳の動きをまねるハードウェアを構築するという動きがあった．それが「パーセプトロン」である（コラム1）．人工的に脳内のニューロンやニューラルネットワークを構築しようとするものだ．パーセプトロンは，使用しているAIのモデルを見ても明らかなように，現在のディープラーニングの基礎となる技術である．当時もパーセプトロンで深いニューラルネットワークを構築すれば人間の推論や学習の仕組みを計算機上に構築できるという予測はあった．しかし，いかんせん計算機の能力が低くて論理をハードウェアで実行できなかった．現在のディープラーニングの隆盛は計算機能力の多大な向上と無縁ではない．

● 計算の仕方は確立しているのだから専用AIチップを作りたくなる

現在，AIと呼ばれているもののほとんど全てはディープラーニング技術であり，（パーセプトロンをはじめとする）約60年の技術の蓄積を高速計算機で実現できるようになり，実用化にこぎつけている．その利用方法は確立しており，高速化のためにはソフトウェアよりもハードウェアでという動きが当然起こってきている．半導体メーカがディープラーニングを処理する，いわゆるAIチップの開発に乗り出すのも自然の流れである．

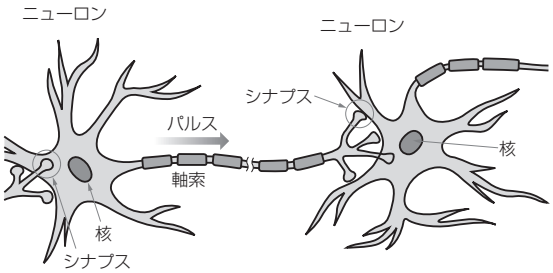


図1　脳の基本構造…脳神経細胞「ニューロン」とネットワーク結合「シナプス」

AIチップで行う基本計算

AIチップとは脳の動きをまねるチップである．これは，ニューロンとシナプスを模式化したモデルを使って人間の知能を再現することを目的とする．

● 脳の動き

脳には数多くの神経細胞が存在していて，その結び付きによって情報が伝達されたり，記憶が定着したりする．この役割を持つ神経細胞が「ニューロン」である．また，ニューロン同士の結合間に電気信号が送られることで，情報を伝達するが，この接合部にあるのが「シナプス」である．つまり，脳の神経細胞がニューロンで，そのつなぎ目がシナプスである（**図1**）．

● 脳神経細胞「ニューロン」の信号処理

このままでは概念に過ぎない．ニューロンをソフトウェアなりハードウェアで扱いやすくする必要がある．そのためにニューロンを模式化したものが「形式ニューロン」[**図2**（**a**）]である．その入出力の関係が**図2**（**b**）である．

w_kはシナプスの特性を示す「重み」である．重みが正の場合は，シナプスは興奮性を示し，重みが負の場合はシナプスが抑制性であることを示す．

x_kは他のニューロンからの出力である．θはニューロンが活性化（発火：出力を後段に伝えること）する

ための閾値である．また，関数$f(x)$には**図3**(**a**)～(**c**)のようなものが考えられている．**図3**(**c**)のシグモイド関数は以前からよく使われてきたのであるが，バック・プロパゲーション(単純にいうと学習)が難しいため，最近では**図3**(**a**)，(**b**)，(**d**)，(**e**)の線形関数が好まれるようである．特に**図3**(**d**)のReLU関数(Rectified Linear Units；整流化線形ユニット)は有名である．

● 脳の動きを模擬した「ニューラルネットワーク」で行う計算

形式ニューロンをつないだものが神経ネットワーク(ニューラルネットワーク)である(**図4**)．AIチップは，ニューラルネットワークの入力から出力までの計算を高速に行うチップである．ここで重要なのは**図4**(**b**)の関数をどうやって高速化するかである．着目点は，

$\Sigma\, w_k \cdot x_k$

の部分である．これを書き下せば，

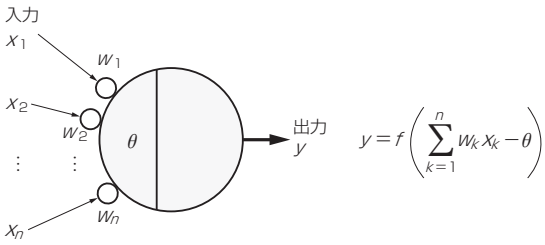
$w_0 \times x_0 + w_1 \times x_1 + w_2 \times x_2 + \cdots\cdots + w_n \times x_n$

という積和算になる．積和とくれば従来，DSP(Digital Signal Processor)やArmプロセッサのNEONなどのSIMD命令で行うと高速計算ができた．

上述の計算式は，以下の2つのベクトルの内積とみなすこともできる．

$(w_0,\ w_1,\ w_2,\ \cdots,\ w_n)$
$(x_0,\ x_1,\ x_2,\ \cdots,\ x_n)$

ベクトルを集めたものは行列である(テンソルともいう)．つまり，行列と行列の乗算，行列とベクトルの乗算が高速に処理できれば，多くの積和算を同時に行えることになる．



(**a**) 模式図　　(**b**) 入力と出力の関係式

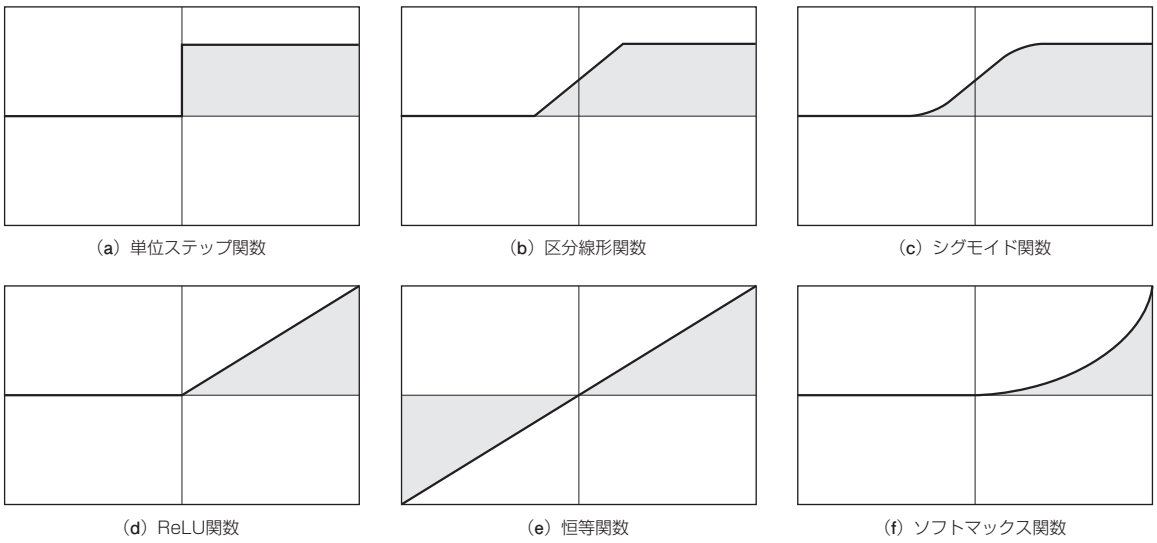
図2　脳神経細胞ニューロンのモデル

● 基本的に並列計算できる

行列と行列の乗算や行列とベクトルの乗算は，行ごと列ごとに並列に計算が可能であるから，マルチプロセッサによる並列処理に適した計算といえる．さらに，ニューラルネットワークにおいて，特定の形式ニューロンの計算には，1つ前の層の結果(ニューロン値)のみが関係し，同層のニューロン計算には依存しないので，その意味でもある層におけるニューロンの値を並列に求めるという処理が可能である．これらから，AIチップの神髄は「テンソル処理の高速処理と並列処理」ということになる．

● GPUが有利な理由

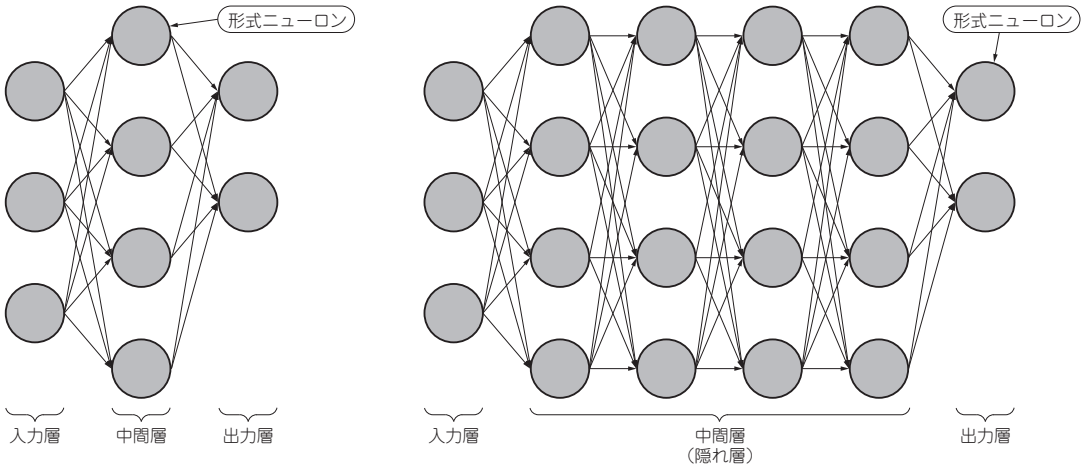
行列と行列の乗算，行列とベクトルの乗算が得意なのはGPU(Graphics Processing Unit)である．GPUでは3次元座標の変換にこれらの乗算を多用する．GPUは数百から数千の積和コアを備えており，テン



(**a**) 単位ステップ関数　(**b**) 区分線形関数　(**c**) シグモイド関数

(**d**) ReLU関数　(**e**) 恒等関数　(**f**) ソフトマックス関数

図3　脳神経細胞のON/OFFを表す活性化関数あれこれ



（a）単純なネットワーク　　（b）深層なネットワーク

図4　脳神経ネットワークの基本構造

ソル処理には適している．NVIDIAのGPUがAI処理チップとして一躍有名になったのはそういう経緯があると思われる．もちろん，DSPやSIMD命令を備えたプロセッサでも同様な処理は可能であるが，GPUに匹敵するコア数や並列処理数を備えたものはほとんど存在しないと思う．

● GPUではないAIチップが開発される理由

ただし，本来GPUはグラフィックス処理を目的としたプロセッサであるから，テンソル処理には無駄な機能も含んでいる．これは消費電力の高騰につながる．そこで，テンソル処理機能のみに特化した構造のチップが開発されるようになった．これがAIチップである．

● 低消費電力なアナログ的アプローチ

また，AIチップには，ディジタル的ではなくアナログ的に人間の脳の動きをより正確にまねようという動きもある（実際はディジタル回路で実装することが多い）．このようなアプローチで開発されたAIチップを「ニューロモーフィック」チップという．ニューロモーフィックチップでは，基本的にクロックが存在せず，イベント駆動型で動作するので，DSP，SIMDやテンソル処理と比べて消費電力が極端に少ないのが特徴である．

AIチップの利点の考察…端末側にメリットがある気がする

さて，AIチップの利点は何であろうか．それは従来複雑な論理で実現していた機能を，比較的単純な構成のディープニューラルネットワークで実現できてしまうことではないかと思っている．超解像や音声認識などがそれに当たる．例えば，正弦（sin）は，一般的には，テーラー展開などでその値を計算する．次のような対応表を作成しておけば，0から9に対する正弦の値は一瞬でその値が求まる（ニューラルネットワークの学習とは，単純にいえば，このような対応表を作ることに等しいといえる）．

```
sin（0）= 0.000000
sin（1）= 0.841471
sin（2）= 0.909297
sin（3）= 0.141120
sin（4）= − 0.756802
sin（5）= − 0.958924
sin（6）= − 0.279415
sin（7）= 0.656987
sin（8）= 0.989358
sin（9）= 0.412118
```

ニューラルネットワーク化によって，複雑な論理で計算するよりも回路規模を小さくできるから，消費電力は小さくなるし，結果は，何層かの，ネットワークを通過するだけで得られるので，スループットの改善にもなる．層の追加で精度も簡単に高品質化できる．しかも，このディープニューラルネットワークは「重み」を変えることでさまざまな機能に対応できる．つまり，1つで何度でもおいしい回路になっていると思う．近年，インターネットのエッジ側でデータ量を減らしてインターネットの負荷を低減するエッジコンピューティングが注目を集めている．AIチップはこのようなエッジコンピューティングに適すると思われる．

Chapter 12

現在の最新 MPU

Arm と RISC-V の命令セットアーキテクチャ

これまで，やや古いMPUの命令セットアーキテクチャを紹介してきたが，新しいところとして，ArmとRISC-Vの命令セットアーキテクチャを機能別に比較しながら説明する．

ここではArm32とかArm64という表現を使用しているが，実はそれらは正式名称ではない．Arm32はArmv7-Aアーキテクチャのネイティブ（Thumbでない方の）モードの命令セットまたはArmv8-AアーキテクチャのAArch32モードの命令セット，Arm64はArmv8-AアーキテクチャのAArch64モードの命令セットを示す．本稿では直感的な理解のために，これらの表現をあえて混同して使用している．

さらに，以下の説明では，アルファベットの大文字と小文字を適宜区別して使っているが，アセンブラ記述では大文字と小文字は，ラベル名以外は区別されない．たとえば，レジスタ名がX0とかx0とかになっているが，同じものとして見てほしい．

レジスタセット

まずは，データの保持や演算を行うレジスタの仕様を知っておくことが必要である．少なくとも使えるレジスタの本数を知らないと，存在しないレジスタを指定する恐れがある．使えるレジスタの本数を知ることは命令セットアーキテクチャを理解する第一歩である．

● Arm32（AArch32）の場合

図1にAArch32（Armv7-A）のレジスタセットを示す．通常のプログラミングにおいては，「アプリケーションレベルビュー」のレジスタを見ておけば十分である．

R0からR15までの16本のレジスタがある．このうち，R15はPC（プログラムカウンタ）である．R15に値を書き込むと，そのアドレスにジャンプできる．また，R15をロード命令のアドレッシングでベースレジスタに使用すればPC相対アドレッシングを行うことができる．

R15以外は，基本的に汎用のレジスタである．「基本的に」の意味は，R14はサブルーチンコールで戻りアドレスが自動的に設定されたり，R13はスタックポインタとして用途が限定されたりしているので，CPUの都合で値が勝手に変化する可能性がある．

APSRはCPUの動作状態を決定するステータスレジスタである．APSRへのアクセスは特殊な命令（MSR/MRS）を使って行う．

● Arm64（AArch64）の場合

図2にAArch64（Armv8-A）のレジスタセットを示す．R0からR30までの31本の汎用レジスタが存在し，それに加えて，SP（スタックポインタ），ZR（ゼロレジスタ），PC（プログラムカウンタ）が存在する．AArch32とは異なり，PCにプログラムで直接アクセスはできない．分岐命令の実行時に，その分岐先が書き込まれ，そこから命令フェッチが開始されるレジスタがCPU内部に存在するという以上の意味はない．SPとZRは，命令のエンコード的にはR31に相当する．それが使用される場面に応じて，SPとして機能したり，ZRとして機能したりする不思議なレジスタである．SPも，普通のMOVE命令や演算のソースやデスティネーションに指定できるので，汎用レジスタとみなすこともできる．つまり，AArch64の汎用レジスタの本数は32本であるといっても差し支えない．

なお，Rn（n = 0，…，30）という名称は仮想的なものだ．アセンブリ言語の記述では，XnまたはWnを使用する．Xnは64ビットレジスタ，Wnは32ビットレジスタを示す．物理的にはXnの下位32ビットがWnとなる．また，SPやZRは，64ビット長と32ビット長で，それぞれ，XSP，WSP，XZR，WZRと記述する．また，サブルーチンコールで戻りアドレスが格納されるレジスタはX30/W30である．

ところで，Armv8-Aでは，AArch64の他にも，AArch32をサポートする．この場合，AArch64のW0からW14がAArch32のR0からR14とみなされる．それ以外のAarch64の汎用レジスタから

アプリケーションレベルビュー	システムレベルビュー								
	User	System	Hyp	Supervisor	Abort	Undefined	Monitor	IRQ	FIQ
R0	R0_usr								
R1	R1_usr								
R2	R2_usr								
R3	R3_usr								
R4	R4_usr								
R5	R5_usr								
R6	R6_usr								
R7	R7_usr								
R8	R8_usr								R8_fig
R9	R9_usr								R9_fig
R10	R10_usr								R10_fig
R11	R11_usr								R1_fig
R12	R12_usr								R12_fig
SP (R13)	SP_usr		SP_hyp	SP_svc	SP_abt	SP_und	SP_mon	SP_irq	SP_fig
LR (R14)	LR_usr			LR_svc	LR_abt	LR_und	LR_mon	LR_irq	LR_fig
PC (R15)	PC								
APSR	CPSR								
			SPSR_hyp	SPSR_svc	SPSR_abt	SPSR_und	SPSR_mon	SPSR_irq	SPSR_fiq
			ELR_hyp						

図1[(1)] **Armv7-Aのレジスタセット**

AArch32の汎用レジスタの割り当ては**図3**のようになっている．**図3**では，R15（PC）にはレジスタの割り当てが存在していないように見える．これは，どこのドキュメントにも書かれていないので想像であるが，AArch32のR15はAArch64のPCに割り当てられていると考えられる．そうでないと，任意のAArch32のプログラムを動作させることができないからである．

● RISC-V（RV32）の場合

図4にRISC-Vのレジスタセットを示す．RISC-Vは，X0からX31の32本の汎用レジスタを持つ．X0が，値が常にゼロの，ゼロレジスタであるほかは，残りの31本のレジスタは真に汎用である．サブルーチンコールの戻りアドレスは，プログラムで指定する．戻りアドレスを格納するレジスタを省略すると，アセンブラはX1を戻りアドレスの格納先として指定する．

図4でXLENというのはアーキテクチャのビット長を示す．32ビットアーキテクチャ（RV32）なら32，64ビットアーキテクチャ（RV64）なら64となる．本稿では，RISC-Vは32ビットアーキテクチャを取り上げる．

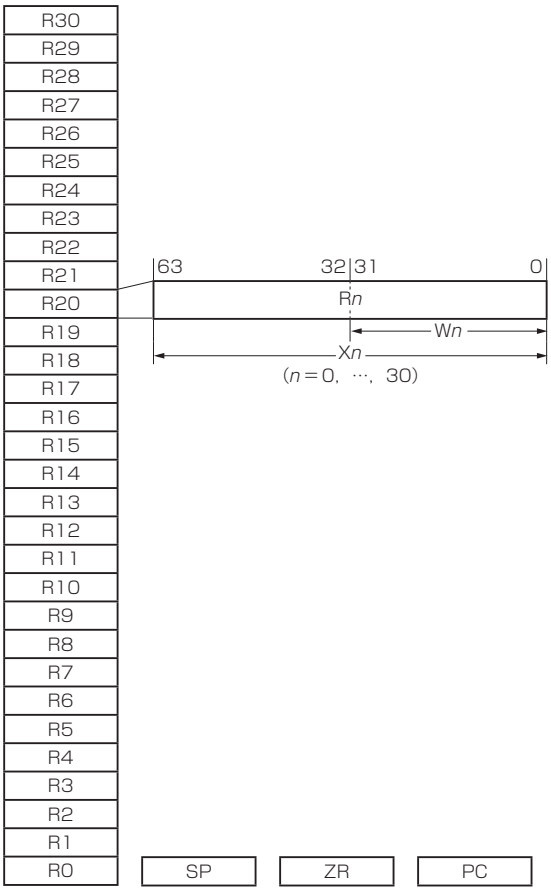


図2[(2)] **Armv8-Aのレジスタセット**

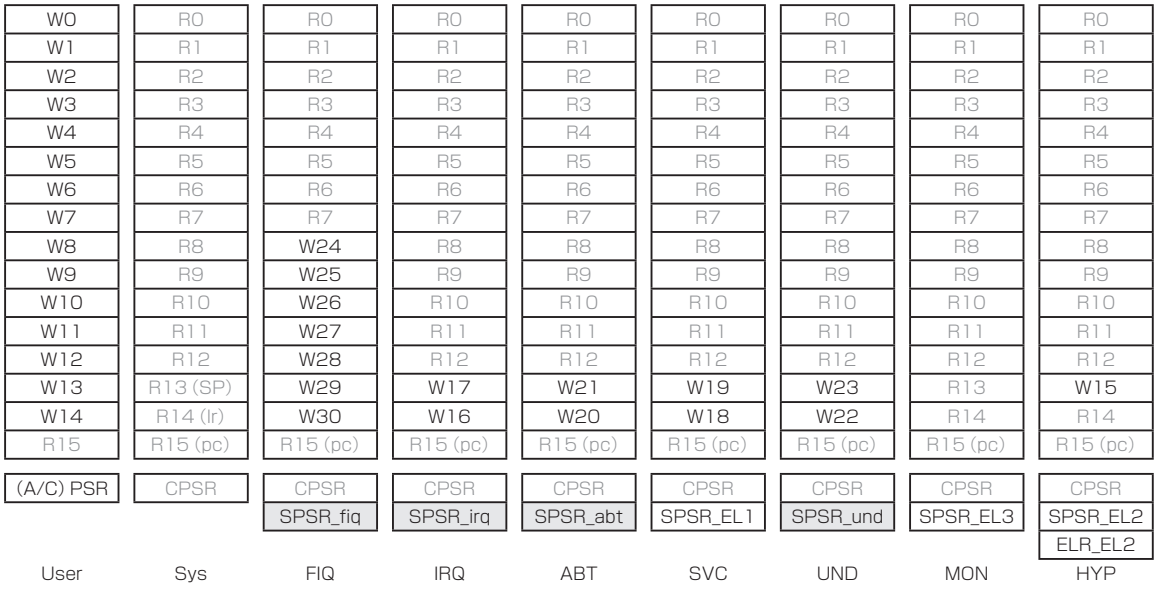


図3(3) **Armv8-AのAArch32時のレジスタセット**

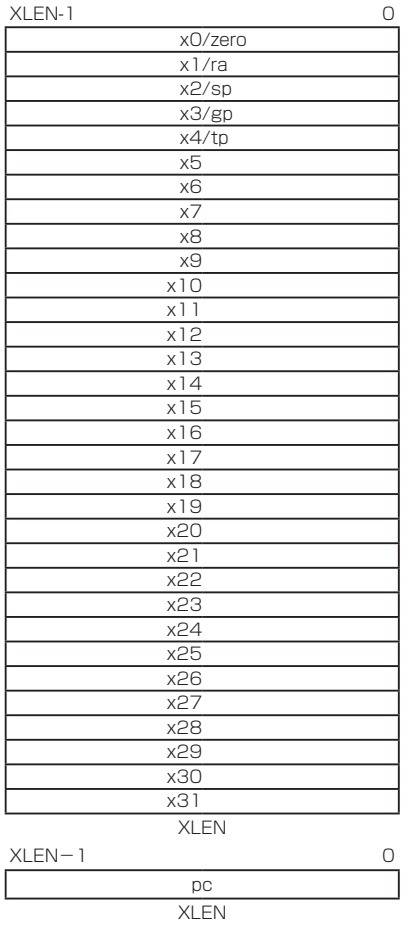


図4(4) **RISC-Vのレジスタセット**

MOVE（移動）命令

この節で登場する命令を**表1**にまとめる．

● Arm32（AArch32）の場合

Arch32において，MOVE命令のニーモニックは`MOV`である．ビット反転した値をMOVEする時は`MVN`命令を使用する．`MOV`命令の最大の用途は，あるレジスタから別のレジスタに値をコピーすることである．たとえば，

```
MOV R0, R1
```

はレジスタR1の内容をレジスタR0にコピー（移動）する．Armの場合，右側のオペランドから左側のオペランドに値を移動するように記述する．

表1　MOVE命令

機　能	AArch32	AArch64	RISC-V
定数値設定（特殊）	`MOV`	`ORR XRZ`	`ORI X0`
16ビット定数の設定	`MOVW`	`MOVZ`	`ORI X0` `(ADDI X0)`
32ビット定数の設定（その1）	`MOV` `ORR`	`MOVZ` `MOVK`	`LUI+ADD`
32ビット定数の設定（その2）	`MOVW` `MOVT`		
32ビット定数の設定（その3）	`LDR(PC)`	`LDR(PC)`	
ラベルのアドレスを設定	`ADR`	`ADR`	`AUIPC` `ADD`

▶定数値をレジスタに格納する方法

さて，問題は定数値（イミディエート値）のレジスタへのMOVEである．Armの命令長は32ビットである．この32ビット内には，オペレーションコードやレジスタ番号が含まれるので，32ビット長の命令の中に32ビットの定数値を埋め込むのは不可能である．

Armアーキテクチャでは，伝統的に，8ビット定数と，偶数ビットの右ローテート（回転）により32ビットの定数を表現する．ということは，8ビットのローテートで表現できない定数は指定できないということになる．でも，安心してほしい，Armには任意の32ビット定数をレジスタに転送する必殺技がある．これについては後述する．

8ビットのローテートに話を戻す．まず，8ビット長以内の定数はそのままレジスタに格納できる．その記法は低数値の前に「#」を付けて，

```
MOV r0,#0x12
```

などのようになる．nビットの右ローテートは（32 − n）ビットの左ローテートに等しいので，上述の定数を，24ビット，16ビット，8ビットだけ右ローテートした，

```
MOV r0,#0x1200
MOV r0,#0x120000
MOV r0,#0x12000000
```

という定数も使用可能である．逆に考えれば，0xFFという8ビットのデータからは，2ビット，4ビット，6ビット，8ビット，…，の右ローテートにより，

```
0xC000003F
0xF000000F
0xFC000003
0xFF000000
……
```

といった定数が作れることになる．さて，こんな定数を使う場面があるのかは，よく分からない．ただし，8ビットの定数を24ビット，16ビット，8ビットで右ローテートすることを考えれば，ORR（論理和）命令と組み合わせることで，任意の32ビット定数を生成することが可能である．つまり，32ビット低数値である0x12345678をR0に格納するためには，

```
MOV r0,#0x12000000
ORR r0,r0,#0x00340000
ORR r0,r0,#0x00005600
ORR r0,r0,#0x00000078
```

という4命令を実行すればいいことになりる．しかし，32ビット定数のレジスタ代入のために4命令も使うのはコードサイズが増えてもったいない．

▶Armv7-Aでの32ビット定数のレジスタ格納

そんな考えをArmのアーキテクトが持ったのかどうかは定かではないが，Armv7-Aでは，MOVW，MOVTという新規のMOVE命令が導入された．MOVWの「W」はWide Immediate（8ビットよりビット幅が広い16ビット定数）を意味し，MOVTの「T」はTop（上位）を意味する．MOVW命令はレジスタの下位16ビットに16ビット定数を直接格納する命令である．MOVTはレジスタの上位16ビットに16ビット定数を直接格納する命令である．このとき，レジスタの下位16ビットは保存される．

まず，MOVW命令を使えば，任意の16ビット定数をレジスタに格納できる．たとえば，

```
MOVW r0,#0x1234
MOVW r0,#0x5aa5
MOVW r0,#0xbeef
MOVW r0,#0xcafe
```

といった具合である．さらに，MOVT命令と組み合わせれば

```
MOVW r0,#0x5678
MOVT r0,#0x1234
```

というように，2命令で0x12345678という32ビット定数をR0に格納することができる．

ここでの注意点はMOVW命令を先に実行することである．MOVW命令は格納先のレジスタの上位16ビットを0にしてしまうので，MOVW命令とMOVT命令の実行順序が逆の場合は，下位16ビットしか格納されないことになる．

▶さらに便利な32ビット定数のレジスタ代入方法

さて，ここで必殺技の紹介だ．実は，1命令で32ビット定数をレジスタに格納可能な命令がある．正確にはArmの命令ではなく，アセンブラの疑似命令である．それは，真のArm命令であるロード命令と同じニーモニックを使うのだが，第2オペランドが定数値であることが異なる．この場合，定数値の頭には「#」ではなく「＝」をつける．つまり，

```
LDR r0,＝0x12345678
```

といった具合である．通常は，32ビット定数をレジスタに格納する場合は，この疑似命令を使用する．この疑似命令はアセンブラにより，PC相対ロード命令（ロード命令に関しては後述するが，メモリからデータをレジスタに読み込む命令と思ってほしい）に変換される．つまり，上述の命令は

```
LDR r0,[pc, #disp]
```

というArmの命令に変換される．ここでディスプレースメントである`disp`は0x12345678というデータが格納されているデータのアドレスを示すように決定される．たとえば，上述のLDR（疑似）命令が0x100番地にあり，0x1000番地に0x12345678という値が格納されているとすると，`disp`の値は0x1000番地からPCの値（ArmでのPCは2命令先を指しているので，0x100番地を実行時のPCは0x108番地になる）を引い

Chapter 14

MPUによる情報保護機能の実装

セキュリティ機能

情報保護への必要性の高まりから，最近のMPUにはセキュリティ機能の搭載が進んでいる．本章では，Armのセキュリティ機能であるTrustZoneと，MIPS及びRISC-Vのセキュリティ機能について解説する

Armのセキュリティ機能であるTrustZone

Armv7-A（Armv8-A）でのTrustZone

● TrustZoneとは何か

TrustZoneとはTrust（信頼できる）なZone（領域）である．つまり，アドレス空間の中にセキュア（秘密）な空間を設ける機構を意味する．TrustZoneの歴史は古く，既に20年以上前のArm11の時代には実装が始まっている．現在提供が行われているCortex-Aファミリでは標準機能の1つとなっており，Cortex-A15以降の世代ではリソース共有機能も強化され，より実用的なものとなっている．

TrustZoneの特徴は，実行環境を2つに分離することである．その1つは通常の実行環境であるノーマルワールド（Normal World）あるいはノンセキュアワールド（NonSecure World）という領域で，ここでAndroidなどの汎用OSが動作する．この領域では，アプリケーションのインストールやデータの読み書き，インターネットへの接続など，通常の動作を行う．

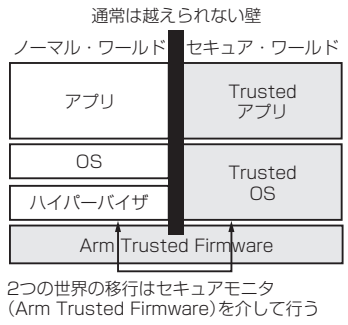


図1　セキュアワールドとノーマルワールド

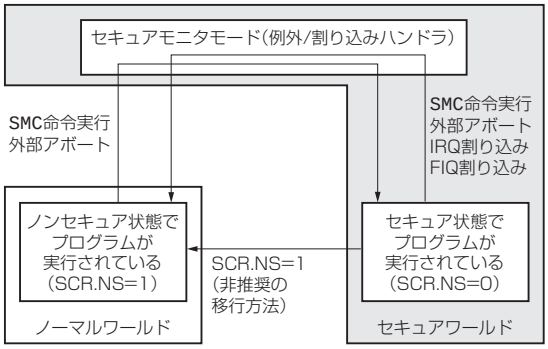
もう1つの環境はセキュアワールド（Secure World）と呼ばれる．この領域は，いわゆる秘密の領域である．ある特定の特権モードでしかアクセスできない．セキュアワールドではTrusted OSと呼ばれる専用OSが動作する．そして，システムのバックエンドでセキュアな情報管理を行う．セキュアワールド側から通常の汎用OSの動作は見える一方，ノーマルワールドにある汎用OS側からはセキュア領域を認識できない．この仕組みにより，マルウェアなどの外部からの攻撃や，Jailbreak（ソフトウェアを改造するアプリケーション）のように意図的にセキュアな仕組みを解除して仕掛けを施そうとする行為を防ぐことが可能になる．この様子を**図1**に示す．

TrustZoneの構築にはセキュアワールドで動作するTrusted OSを開発することが必要である．このOS開発がTrustZone普及のネックになっていたといわれている．そんな時，GlobalPlatformがTEE（Trusted Execution Environment）を定義したことにより，TrustZoneがより注目されるようになった．TEEは，大々的に普及が始まっているモバイルOSなどのセキュア領域をいかに活用するかということを目指したものである．これにより，Trusted OSの開発が容易になった．

● 領域保護を行う具体的な仕組み

▶セキュア状態とノンセキュア状態

TrustZoneはCPUに新たな状態を定義する．それがセキュア状態とノンセキュア状態である．また，メモリ空間もセキュアワールドとノンセキュアワールド（ノーマルワールド）に分離する．そして，セキュア状態ではセキュアワールドとノーマルワールドで動作



スーパーバイザ，FIQ，IRQ，未定義，アボート，システムは，CPUの動作モードを示している

図2　セキュアワールドとノーマルワールドの切り替え

できるが，ノンセキュア状態ではノーマルワールドでしか動作できないようにする．同じように，セキュア状態では，セキュアワールドとノーマルワールドの資源にアクセスできるが，ノンセキュア状態ではノーマルワールドの資源にしかアクセスできない．このような動作環境を構築するのがTrustZoneである．

▶セキュアワールドからノーマルワールドに移行する方法

ノーマルワールドとセキュアワールドの切り替えはSCR(セキュア構成レジスタ)のNSビット(ビット0)で行う．リセット直後はSCR.NS = 0なので，セキュアワールドにいる．2つのワールド間の切り替えは，**SMC**(セキュアモニタコール)命令またはIRQ，FIQ，外部アボートを用いてセキュアモニタモードに移行して行う．なお，セキュアワールドでSCRのNSビットを書き換えてノーマルワールドに切り替えることは非推奨である．この様子を**図2**に示す．

具体的には，**SMC**命令を実行するとCPUの実行モードがセキュアモニタモードに移行する．セキュアモニタモードでは，SCRのNSビットの値にかかわらず，セキュアワールドにいることになる．セキュアモニタモードは，セキュアワールドとノーマルワードを橋渡しするモードである．セキュアモニタモードからノーマルモードに移行する場合は，**SMC**命令例外からのリターンの前に，ソフトウェアでSCRのNSビットを1(ノンセキュア)に設定する．

なお，**SMC**命令の実行以外にも，IRQ割り込みやFIQ割り込み，あるいは外部アボートでモニタモードに移行することができる．割り込みや例外によるモニタモードへの移行が可能かどうかはSCRの設定によって行う．モニタモードに移行する場合のSCRの設定を**表1**に示す．

モニタモード(それぞれの要因の例外/割り込みハンドラ)では次のような処理が行われる．これらは全てソフトウェアで実現される．

1，セキュア状態で使っていたレジスタの値をセキュア領域のメモリに退避
2，レジスタに適当な値を格納して，セキュア状態での値を隠す(ノーマルワールドのレジスタを退避している場合は，それを回復する)
3，分岐先の例外レベルに対応したELRにノーマルワールドのアドレスを設定する
4，SCRレジスタのNSビットを1に設定する
5，**ERET**命令を実行して，ELRが示すアドレスに分岐する

表1　IRQ，FIQ，外部アボートでモニタモードに移る場合のSCRの設定

ビット	名　称	意　味
3	EA	0：外部アボートをアボートモードで実行 1：外部アボートをモニタモードで実行
2	FIQ	0：FIQ割り込みをFIQモードで処理 1：FIQ割り込みをモニタモードで処理
1	IRQ	0：IRQ割り込みをFIQモードで処理 1：IRQ割り込みをモニタモードで処理

▶ノーマルワールドからセキュアワールドに移行する方法

ノーマルワールドからセキュアワールドに移行するためには**SMC**命令の実行一択である…といいたいところだが，SCRレジスタのEAビットが1の場合は外部アボートでもモニタモードに遷移できる．IRQ割り込みやFIQ割り込みではモニタモードには遷移しない．これが，セキュアワールドからノーマルワールドへ移行する場合との違いである．

ここで，セキュアモニタ(例外/割り込みハンドラ)では次のような処理が行われる．これらは全てソフトウェアで実現される．

1，ノーマルワールドで使っていたレジスタの値をノーマルワールドのメモリに退避
2，分岐先の例外レベルに対応したELRにセキュアワールドのアドレスを設定する
3，SCRレジスタのNSビットを0に設定する
4，**ERET**命令を実行して，ELRが示すアドレスに分岐する

また，セキュアモニタモードで例外が発生すると自動的にSCRのNSビットが0になる．つまり，自動的にセキュアワールドに移行する．

▶アドレス空間の分離

TrustZoneの本質は，アドレス空間をセキュアワールドとノーマルワールドに分離することである．つまり，メモリ空間をセキュアとノーマル(ノンセキュア)に分離することが必要となる．

Cortex-AにはMMU(Memory Management Unit：

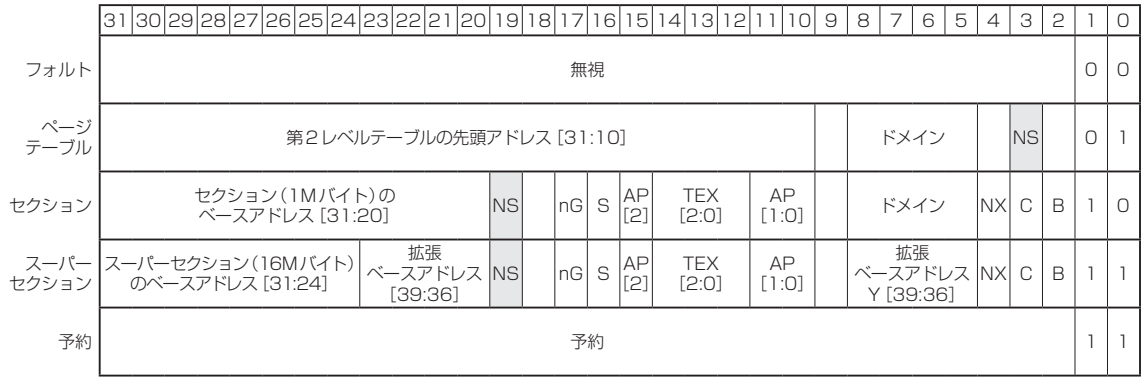


図3　セキュアとノーマルな空間はアドレス変換テーブルのNSビットで区別する

この図は，Armv7-Aのアドレス変換デスクリプタである．Armv8-Aでは形式が少し異なるが，デスクリプタの中にNSビットを含むことは同じである

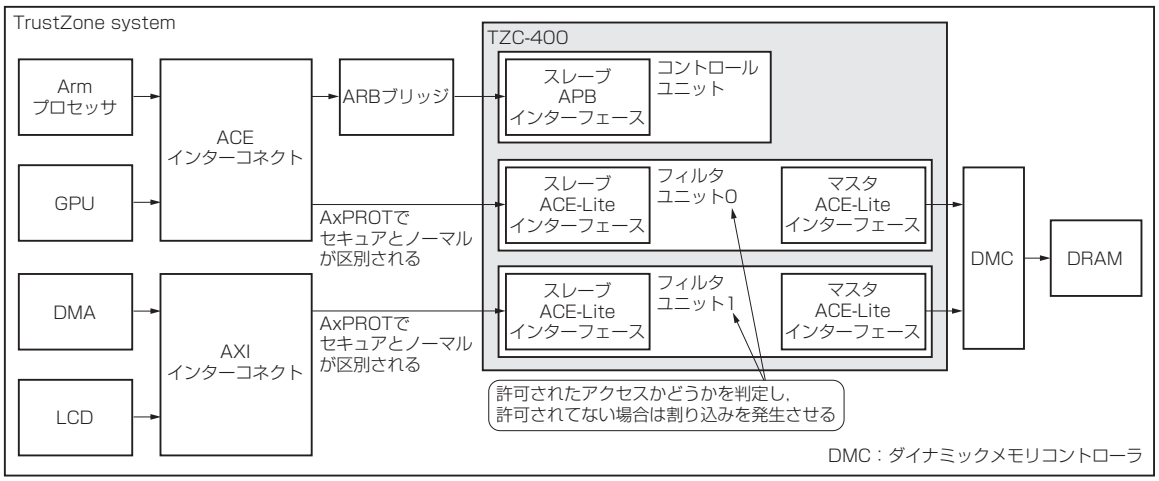


図4　TrustZoneでのメモリ分離

メモリ管理ユニット）が内蔵されている．メモリ空間がセキュアであるかノーマル（ノンセキュア）であるかはMMUで指定する．正確にいえば，MMUが管理するアドレス変換テーブルの中にNS（NonSecure）ビットが存在（**図3**）し，このNSビットを1に設定してあるメモリ空間はノーマル（ノンセキュア）ということになる．逆に，NSビットを0に設定してあるメモリ空間はセキュアである．このため，Cortex-Aでは，最小では，ページ単位（4Kバイト単位）で，そのメモリ領域がセキュアワールドであるかノーマルワールドであるかを指定できる．

CPUの外部のアドレス空間の分離は，CPUから出力されるACEまたはAXIバスの制御信号であるAxPROT［1］（リード用がARPROT，ライト用がAWPROT）信号とTZPC（トラストゾーンプロテクションコントローラ：TZC-400など）というIPを使って実現される（**図4**）．AxPROT［1］はアドレス変換テーブルのNSビットの値が反映されている．

TZC-400は，ACE-LiteまたはAXI信号によるリードまたはライト要求を受けて，それらがTZC-400内のレジスタ設定で許可されたアクセスならばアクセスをそのまま通過させ，許可されてないアクセスならばエラー応答または割り込みを発生させる．そして，その旨をマスタ（CPU）に通知する．**図4**でいうフィルタユニットがその役割を果たす．フィルタユニットは，TZC-400に入力されるアドレスを監視し，特定の領域で指定されたアドレス範囲に対しセキュアなアクセス（AxPROT［1］= 0）であれば許可するという単純な構成になっている．なお，1つのTZC-400の中にフィルタユニットは4個存在する．

ところで，ノーマルワールドのメモリ空間とセキュアワールドのメモリ空間は異なる空間である．例えば，セキュアワールドのメモリ空間の0x8000番地とノーマルワールドのメモリ空間の0x8000番地は，別

ISBN978-4-7898-4556-4
C3055 ￥4000E
CQ出版社

定価 4,400円（本体4,000円）⑩