■6 群（コンピュータ･基礎理論とハードウェア）- 4 編（コンピュータアーキテクチャ（I）標準的）

3 章 コンピュータの構成

【本章の構成】

本章では，コンピュータの構成要素（3-1 節），レジスタとメモリ（3-2 節），命令の機能（3-3 節），命令形式（3-4 節），及びアドレス方式（3-5 節）について，それぞれ述べる．

■6 群 - 4 編 - 3 章

3-1 コンピュータの構成要素

（執筆者：三輪　忍）[2011 年 9 月 受領]

本節では，まず，コンピュータの全体的な構成を概観する．次いで，コンピュータを構成する主要なモジュールについて，その内部構成とその処理について簡単に述べる．また，コンピュータの性能を上げるためには，これらのモジュール間の通信性能がボトルネックとなる．最後にそれについて述べる．

3-1-1 全体構成

コンピュータは，その用途により，様々なモジュールを組み合わせて構成される．それぞれのモジュールは機能や内部構成は異なるが，大きく以下の 3 つに分類できる．

1. プロセッサ
2. 主記憶
3. 入出力装置

プロセッサは，プログラムに従い，命令を実行する装置である．プログラムの実行において中心的な役割を果たすことから，プロセッサはコンピュータの心臓部に例えられることが多い．汎用コンピュータのプロセッサは，特に，CPU（Central Processing Unit）と呼ばれる．例えば，Intel 社の Core シリーズ，AMD 社の Opteron シリーズなどがその代表である．

主記憶は命令やデータを格納する領域である．物理的には DRAM（9 群 3 編参照）によって実現される．プロセッサは，プログラムの実行に必要な命令やデータを主記憶から取り出し，処理を行う．そうして処理された結果は，必要に応じて主記憶に書き込まれる．

入出力装置は，ユーザをはじめとする，外界とのコミュニケーション手段を提供する．入力装置からの信号/データがプロセッサ/主記憶に送られ，プロセッサ/主記憶からの信号/データが出力装置へと送られる．入力装置の代表はキーボードやマウス，出力装置の代表はディスプレイである．HDD や SSD（8 群 2 編参照），CD や DVD などの補助記憶装置，ネットワークインタフェースカードも入出力の一種である．

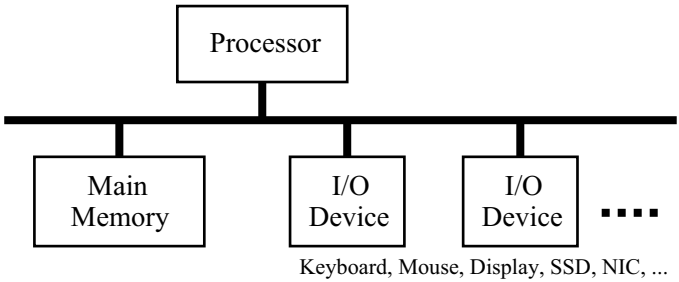


図 3·1　コンピュータの構成

コンピュータの全体構成を**図 3･1** に示す．各々のモジュールはシステムバス*を用いて接続される．システムバスを介して，各モジュールは他のモジュールと信号やデータをやり取りする．

図はモジュール間をバス方式で接続しているが，近年では，P2P 方式で接続するケースも増えてきている[1, 2]．これは，一つには，コンピュータの高機能化が進み，接続する入出力装置の数が増加したことによって，バス方式では高い処理性能と拡張性を実現するのが難しくなってきているためである．また，近年のプロセッサデザインはチップ上に数個程度のコア（プロセッサ）を集積するマルチコアが主流となっている[3, 4]．コア数は，今後は数十～数百個程度に増加すると言われており[5]，このような流れも P2P 方式を後押ししている[6]．

以下では，主記憶，及び，プロセッサについて簡単に説明する．入出力装置については 8 群で詳しく述べられているので，そちらを参照していただきたい．

3-1-2 主記憶

主記憶は，その実装は様々なものが存在するが，論理的にはいずれも巨大な 1 次元の配列が存在するものと考えてよい．

図 3･2 は主記憶の概念を表した図である．主記憶は，図に示すように，複数のエントリがシーケンシャルに並んだ構造をしている．一つひとつのデータは，これらのエントリー，場合によってはエントリーの集合によって保持される．個々のエントリーのサイズは 1 バイトであることが多い．

各エントリーはそれと一意に対応するインデクスを用いて参照される．このインデクスをメモリアドレスと呼ぶ．図の例では，アドレス 4 を用いて主記憶を参照すると，対応するエントリーに格納されたデータ *X* が取得できる．

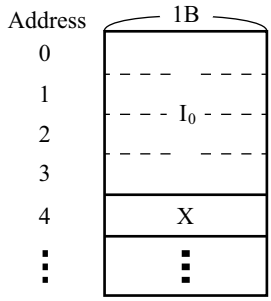


図 3･2　主記憶の構成

主記憶に格納する命令やデータは，1 つのエントリに収まらないこともある．近年の汎用プロセッサでは，1 命令につき 4 バイト程度を必要とするのが普通である．また，データも，その型にもよるが，1 つにつき数バイトを必要とすることが多い．このような場合は，複数の連続するエントリーを用いて格納する．図では，アドレス 0 からアドレス 3 までの 4 つのエントリーを用いて，1 つの 4 バイトの命令 I_0 を保持している．

* 一般的な意味のバスであり，バス方式のバスに限定しているのではない．

以上，主記憶について簡単に説明したが，コンピュータのおおまかな構成を理解するためには，上述の理解で取り敢えず十分である．主記憶について更に詳しく知りたい方は次項及び本編 4 章を，主記憶の物理構成について知りたい方は 9 群 3 編を参照していただきたい．

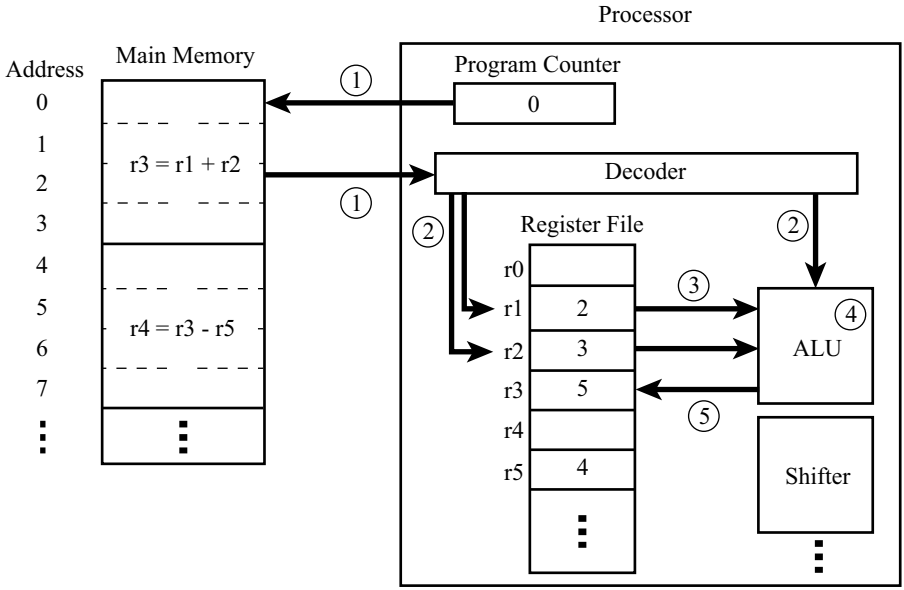


図 3･3　プロセッサの構成と動作

3-1-3 プロセッサ

(RISC) プロセッサの基本的な構成を**図 3･3** に示す．プロセッサは，以下の 4 つのモジュールを用いて命令を実行する．

1. プログラムカウンタ
2. デコーダ
3. レジスタファイル
4. 演算器

プログラムカウンタは，プロセッサが現在実行中の命令のアドレスを保持するレジスタである（本編 1 章参照）．プロセッサはこのレジスタの値を用いて主記憶を参照し，該当するアドレスに格納された命令を取得する．命令の実行が完了すると，実行結果に応じて，次に実行すべき命令のアドレスがプログラムカウンタにセットされる．

デコーダでは，命令のビット列を見て，命令の種類や演算に必要なオペランド（本章 4 節

参照）を解析する．解析結果に応じて，必要なオペランドをレジスタファイルから読み出す．また，適切な演算器を選択し，レジスタファイルから読み出された信号が選択された演算器へと送られるよう，データパスを切り替える．

レジスタファイルは，演算に必要なデータを一時的に格納する領域である．複数のレジスタから成り，1 つのデータは基本的には 1 つのレジスタに格納される．データは，ロード命令（本章 3 節参照）が実行されると，主記憶からレジスタファイルに書き込まれる．また，ストア命令（同節参照）によって主記憶に書き戻される．

演算は，演算器（実行ユニットとも呼ばれる）によって行われる．プロセッサには，演算の種類に応じて，ALU（Arithmetic Logic Unit）やシフタ，浮動小数点演算器など様々な演算器が用意されている．それらを適切に選択し，演算が行われる．演算結果は，該当するレジスタへと書き込まれる．

図 3･3 を例に，プロセッサの動作を説明する．$r0 \sim r5$ の記号はレジスタ番号を表している．また，各エントリーの中の数字は，そのレジスタが保持する値である．すなわち，レジスタ $r1$ は値 2 を保持している．プログラムカウンタの初期値を 0 とする．

プロセッサが 1 命令を処理する流れは以下のようになる．

1. まず，プログラムカウンタの値を用いて主記憶を参照し，該当する命令を取得（フェッチ）する．プログラムカウンタの値 0 を用いて主記憶を参照すると，アドレス 0 から始まる命令は「r3=r1+r2」であるため，これを取得する．フェッチされた命令はデコーダへと送られる．

2. デコーダにより，命令の種別と必要なオペランドを解析し，使用するレジスタと演算器を決定する．命令「r3=r1+r2」は，加算命令であるため ALU が選択される．また，この命令は，r1, r2 という 2 つのソースオペランド，及び，r3 という 1 つのデスティネーションオペランドを有する．そこで，2 つのソースオペランドの値を ALU へと送るよう，レジスタファイルに要求する．

3. レジスタ読み出しが行われ，指定したソースオペランドの値が選択された演算器へと送られる．r1 の値である 2，及び，r2 の値である 3 が ALU へと送られる．

4. 演算が行われる．r1+r2，すなわち，2 + 3 が行われ，その結果，5 を得る．

5. 演算結果がレジスタへと書き込まれる．この命令のデスティネーションオペランドは r3 であるため，先ほどの演算結果 5 が r3 に書き込まれる．結果がレジスタへと書き込まれると，プログラムカウンタの値を更新（この場合は +4）し，1 の処理に戻る．

プロセッサは，上述のようにして命令を次々に実行していく．

上述のように，レジスタファイルは演算に必要なデータを格納するメモリである．命令が実行される際，レジスタファイルは，基本的には，常に参照される．したがって，レジスタファイルは，高速に（多くの場合 1 クロックサイクルで）アクセスできることが求められる．そのため，レジスタファイルは SRAM（9 群 3 編参照）によって構成される．

レジスタファイルは，汎用プロセッサでは，整数演算用と浮動小数点演算用，それぞれ別に

設けられる．これは，整数演算と浮動小数点演算は，全くと言ってよいほど別の命令流だからである．整数演算の結果を用いて浮動小数点演算を行うケース，あるいは，その逆のケースは極めて稀である．すなわち，性能の観点からはデータパスを分けるデメリットはほとんどないと言ってよい．

また，近年のハイエンドプロセッサでは，メディア系のプログラムに多く含まれるベクトル演算を高速に行うため，1 命令で複数データに対する演算を行う SIMD 演算器を設け，これを利用できるように命令セットを拡張している．この拡張を Intel 社は SSE（Streaming SIMD Extensions）と呼んでいる[7]．SIMD 演算器を有するプロセッサでは，上述した 2 つのレジスタファイル以外に，SIMD 演算用のレジスタファイルを持つ．

3-1-4 フォン・ノイマン・ボトルネック

このように主記憶とプロセッサとを分けたことは，フォン・ノイマン・ボトルネックと呼ばれる問題を引き起こした．主記憶，及び，システムバスのスループットによって，プロセッサの性能が律速されてしまうという問題である．プロセッサ内の命令の処理能力は，主記憶がシステムバスを通じて命令やデータを供給する能力よりもはるかに高い．

この問題を解決する最も単純な方法は，命令とデータとでメモリを分けてしまうことである．メモリを 2 つに分けたうえで，それぞれのメモリ用にシステムバスも分割する．このようにすればスループットは 2 倍になる．このようなアーキテクチャをハーバードアーキテクチャと呼ぶ*．ハーバードアーキテクチャは，DSP（Digital Signal Processor）などで多く採用されている．

一方，汎用プロセッサをはじめとする多くのプロセッサでは，キャッシュを用いることでこの問題を緩和している．

キャッシュとは，主記憶上のデータの一部を保持する，高速アクセス可能な小容量のメモリである．プロセッサ内部に設けられる．

主記憶へのすべての参照は，まず，キャッシュに対して行われる．参照するデータ（命令）のアドレスによってキャッシュを検索し，該当するデータが存在するか否かをチェックする．該当するデータがキャッシュに存在した場合は，キャッシュヒットである．ヒットした場合は，該当データをキャッシュから直接取得できる．すなわち，そのような場合は主記憶を参照する必要はない．

このようにキャッシュヒットした場合は主記憶は参照されないため，そのようなケースでは主記憶やシステムバスを利用せずに済む．すなわち，主記憶やバスのスループットに対する要求を緩和することができる．

キャッシュは階層化されることもある．近年の汎用プロセッサでは，キャッシュは 2 ~ 3 程度の階層構造をなすことが多い．プロセッサに近い方から順に，レベル 1 キャッシュ，レベル 2 キャッシュ，…，のように呼ばれる．

このような記憶階層を持つプロセッサでは，レベル 1 キャッシュは，命令用とデータ用とに分離するのが一般的となっている．これは，形を変えたハーバードアーキテクチャと言え

* ハーバードアーキテクチャに対して，命令とデータが主記憶を共有する方式をプリンストンアーキテクチャと呼ぶ．

る．

■参考文献

1) http://www.hypertransport.org/
2) Intel Corporation, An Introduction to the Intel QuickPath Interconnect (white paper), 2009.
3) T. Fischer et al., Design solutions for the bulldozer 32nm SOI 2-core processor module in an 8-core CPU, In *Proceedings of the International Solid-State Circuits Conference*, pp.78–80, 2011.
4) N. A. Kurd et al., Westmere: A family of 32nm IA processors. In *Proceedings of the International Solid-State Circuits Conference*, pp.96–97, 2010.
5) S. Vangal et al., An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS, In *Proceedings of the International Solid-State Circuits Conference*, pp.98–99, 2007.
6) W. J. Dally and B. Towles, Route Packets, Not Wires: On-Chip Interconnection Networks, In *Proceedings of the 38th Design Automation Conference*, pp.684–689, 2001.
7) J. Coke et al., Improvements in the Intel Core2 Penryn Processor Family Architecture and Microarchitecture, *Intel Technology Journal*, vol.12, no.3, pp.179–192, 2008.

■6 群 - 4 編 - 3 章

3-2 レジスタとメモリ

■6 群 - 4 編 - 3 章

3-3 命令の機能

（執筆者：中田　尚）[2011 年 8 月 受領]

コンピュータは主記憶や外部から与えられる命令とデータに従い動作する．そして，ある目的のために作成された命令とデータの集合がプログラムである．プログラムは HDD や DVD-ROM のような外部記憶装置や，あらかじめシステムに組み込まれた ROM などに保存されている．プログラムの保存場所に関わらず，すべてのプログラムはいったん主記憶に展開される．また，データは記憶装置以外にもコンピュータに接続された，キーボードやカメラといった入力装置からも供給される．主記憶から順に読み込んだ命令による指示と供給されるデータに従い動作する．このとき，作業領域としてレジスタを利用する．

命令の機能に注目して大まかに分類すると，各種の算術論理演算を行う演算命令，プログラムの実行の流れを制御する分岐命令やジャンプ命令，レジスタと主記憶の間でデータの転送を行うロード/ストア命令，外部機器の制御などを行うその他の命令に分けることができる．

3-3-1 演算命令

(1) 算術論理演算命令

演算命令は最も基本的な命令である．演算命令は更に算術演算命令と論理演算命令に分けられる．このことから併せて算術論理演算命令と呼ばれることもある．算術演算命令は，加減乗除が含まれ，整数を対象とした演算と浮動小数点を対象とした演算がある．一方，論理演算命令は論理積（AND）や論理和（OR）といった命令が含まれる．**図 3･1** に加算（ADD）命令の例を示す．

```
ADD r1, r2, r3 ;  r1 = r2 + r3
```

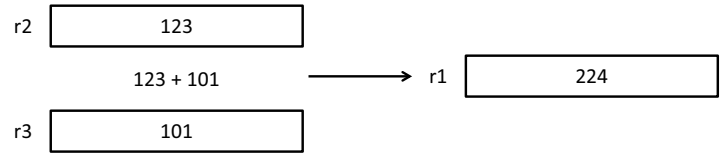


図 3･1　加算命令

典型的な命令は図 3･1 のように，2 つのレジスタからの値を入力として演算を行い，結果を 1 つのレジスタに格納する．このとき，明示的な制限がなければ同一のレジスタを複数回使用してよく，例えば `ADD r1, r1, r1` のような指定も可能である．また，演算によっては演算対象がレジスタではなく定数を指定できるものもある．この定数を即値（Immediate）と呼ぶ．これにより，A=B+10 のような演算が可能となる．

(2) データ変換命令

2-1 節で述べたとおり，整数と浮動小数点数ではデータの表現形式が違う．また，それぞれに独立したレジスタファイルが搭載されているプロセッサも多い．したがって，整数と浮動小数点数を演算するためにはデータ形式の変換とレジスタファイル間のデータ転送が必要

である．

データ変換には整数と浮動小数点数間の変換と，浮動小数点数間の変換がある．整数から浮動小数点数への変換では有効数字の喪失による誤差が発生する可能性があり，浮動小数点数から整数への変換では誤差に加えオーバーフローの可能性がある．浮動小数点数間では単精度と倍精度のように精度が異なる浮動小数点数間の変換がある．こちらも倍精度から単精度のように，表現範囲が狭くなる方向の変換ではオーバーフローの可能性に注意が必要である．

(3) フラグの利用

特に算術演算命令には符号の有無や各種フラグの参照・更新方法に応じて，同一種類の演算であっても幾つかのバリエーションがある．

例えば，レジスタのビット幅を超える値の加算はキャリーフラグを用いることで**図 3･2** のように実現することができる．この例では ADD 命令と ADC（ADd with Carry）命令を用いて，r1, r2 と r3, r4 にそれぞれ連続して格納された値を加算し，結果を r1, r2 に格納する．ここでは r1, r3 に下位 32 bit，r2, r4 に上位 32 bit が格納されている．

```
ADD r1, r1, r3 ;  r1 = r1 + r3
ADC r2, r2, r4 ;  r2 = r2 + r4 + carry
```

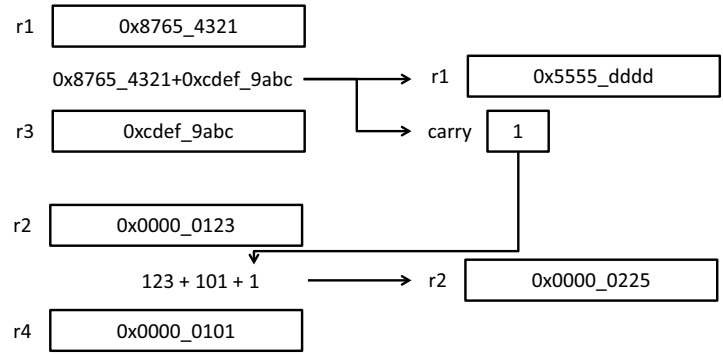


図 3･2　大きな数の加算

(4) 条件実行

実行に対して条件を付けることができる命令のことを条件実行命令と呼ぶ．条件判定にはフラグの値を用いることが多く，指定された条件を満たす場合には通常命令と同様に実行されるが，それ以外の場合には何も行わない．

条件実行を効果的に用いることができれば，プログラム中の条件分岐の頻度を減らし実行の流れをスムーズにすることができる．一方，条件が成立しなかった場合には有効な演算を行わないので，使い分けには注意が必要である．つまり，条件が成立する確率が十分高いか条件実行を行う命令が十分少なければそのメリットを得られる可能性が高いが，条件が成立せずに無効になる命令が多くなるにつれてそのメリットは小さくなり，通常の条件分岐を用いた方が高速に実行できる可能性が発生する．具体的にどの程度の頻度で使い分けを行うべ

きかは，対象プロセッサや実行するプログラムに大きく依存する．

(5) マルチメディア命令

基本的な演算を拡張し特定の処理に特化した命令としてマルチメディア命令がある．

現在では 64 bit プロセッサも広く使われるようになり，このようなプロセッサではレジスタのビット幅も同様に拡大している．その一方，画像処理を含む各種信号処理において，処理対象データのビット幅はそれほど必要とはされていない．例えば，256 階調の画像処理であれば 8 ビットで十分である．そこで，1 つのレジスタに複数のデータを格納し，一括処理する命令が SIMD（Single Instruction Multiple Data）命令である．

例えば，64 ビットのレジスタを使い，8 ビットの演算 8 個を一括して実行する．また，このような処理ではオーバフローの扱いが問題になるが，この問題に対しては最大値または最小値を超えた場合には最大値または最小値を出力することが多い．これを飽和演算と呼ぶ．演算中に飽和処理が発生した場合にフラグなどを用いた検出が可能であるかどうかはプロセッサにより様々である．

また，SIMD 命令以外にも，データ中の最上位の 1 のビット位置を得る CLZ（Count Leading Zero）命令や，絶対値総和を求める SAD（Sum of Absolute Difference）などもマルチメディア命令の例である．**図 3·3** に CLZ 命令，**図 3·4** に SAD 命令の例を示す．

```
CLZ r1, r2 ;  r1 = CountLeadingZero(r2)
```

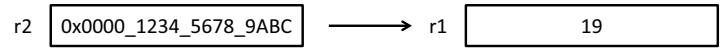


図 3·3 CLZ **命令**

```
SAD r1, r2, r3 ;  r1 = |r2[0]-r3[0]| + |r2[1]-r3[1]|+
                  ;       |r2[2]-r3[2]| + |r2[3]-r3[3]|
```

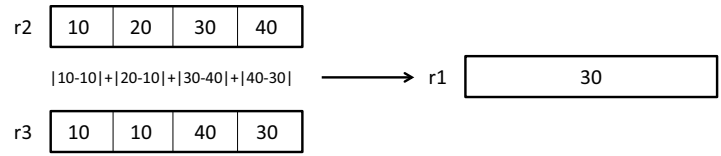


図 3·4 SAD **命令**

このように，利用頻度が高いが通常命令の組合せでは命令数が多くなってしまうような処理を 1 命令で実行可能にすることにより，飛躍的な性能向上を実現することができる．

(6) レイテンシー

演算の実行開始から完了までの所要時間（レイテンシー）は，より効率の良いコンピュータを実現するうえで重要な要素である．種々の命令はその演算内容によりそのレイテンシーは異なる．一方，コンピュータは一定の周期を刻むクロックに従って動作しており，演算の

種類によってクロックの速度を変化させることは不可能である．

最も単純な解決策は，レイテンシーが最も長い演算命令にクロックを合わせることであるが，これではレイテンシーの短い命令を実行するときに，無駄な待ち時間が発生する．レイテンシーが長い命令の例としては，除算命令が挙げられる．除算命令は加減算と比較して 10 倍以上のレイテンシーであるが，その出現頻度は比較的小さいので，除算命令のレイテンシーにクロックを合わせることは許容できない．そこで，考えられるのが演算のマルチサイクル動作である．レイテンシーの長い命令には複数サイクルかけて実行を行うことで，クロックの高速化を実現する．

ただし，マルチサイクル動作にはその機構が複雑になるという欠点もあるため，除算命令をサポートしないプロセッサも存在する．このようなプロセッサで除算を行いたい場合には，複数の命令を組み合わせることにより実現する．

3-3-2 制御移譲命令

制御移譲命令はプログラムの実行の流れを変化させる命令である．プログラムの実行はプログラムカウンタによって制御されているため，プログラムカウンタの値を変更することにより実行の流れを制御する．プロセッサによっては演算命令を用いてプログラムカウンタを変更することができるものもあるが，ここでは説明しない．

プログラムカウンタを変更する命令を，分岐命令またはジャンプ命令と呼ぶ．両者の使い分けはプロセッサや文献によって様々であるが，以降では分岐命令と呼ぶ．分岐命令にはフラグレジスタの内容によってプログラムカウンタを変更するかどうかを切り替えるものがあり，これを条件分岐と呼ぶ．条件によらず変更する命令は無条件分岐と呼ぶ．

分岐先の指定方法には，現在のプログラムカウンタからの相対アドレスで指定，レジスタの値を利用して指定，絶対値で指定の 3 種類がある．

(a)

```
if(r1==r2) then
  真の場合の処理
else
  偽の場合の処理
endif
```

(b)

```
1000: CMP r1, r2        ; r1 と r2を比較
1004: BEQ 14            ; 等しければ PC+14 へ飛ぶ
1008:  ...              ; else 節開始
100c:  ...              ;
1010:  ...              ; else 節開始
1014: BRA 10            ; PC+10 へ飛ぶ
1018:  ...              ; then 節開始
101c:  ...              ;
1020:  ...              ; then 節終了
1024:
```

図 3･5　if 文の実現

相対アドレスによる分岐は if 文や for 文のような制御構造を実現するために用いられる．例えば，**図 3･5** (a) のような if 文は図 (b) のように実現される．この例では，まず CMP（CoMPare）命令で r1 と r2 を比較し，両者の値が等しければ BEQ（Branch on EQual）命令により Then 節へ飛び，等しくなければそのまま Else 節を実行する．Else 節の終端では Then 節を飛ばすために，BRA（BRanch Always）命令を使用している．

レジスタの値を利用した分岐をレジスタ間接分岐と呼び，関数呼び出しや関数からの復帰時に利用される．関数への復帰のためには呼び出し元のプログラムカウンタを保存しておく必要があるが，分岐とプログラムカウンタの保存を同時に行う命令が用意されていることが多い．**図 3･6** に関数呼び出しの実現例を示す．この例では関数呼び出しに BAL（Branch And Link）命令，復帰に BR（Branch to Register）命令，プログラムカウンタの保存に LR（Link Register）レジスタを利用している．

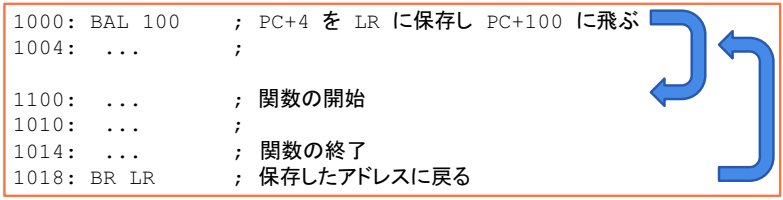


図 3･6　関数呼び出し

また，switch-case 文は case ごとに用意した分岐先アドレスの配列とレジスタ間接分岐の組合せで実現される．

3-3-3 ロード/ストア命令

主記憶の指定されたアドレスからレジスタへデータを転送する命令がロード命令であり，その逆がストア命令である．アドレスの指定方法については 3-5 節で説明する．

ロード/ストア命令では，操作対象のデータサイズに応じて複数種類の命令が用意されていることがある．レジスタのサイズよりも小さなデータをロードするときには，上位のビットをどのように扱うのかを指示することができる．実際の処理は上位を 0 とするか符号ビットで埋める（符号拡張）かのどちらかである．一方，1 つの命令でレジスタのサイズよりも大きなデータを複数のレジスタにロード/ストアする命令も存在する．

3-3-4 その他の命令

これまでに述べた命令は計算の実行に必要な命令であったが，コンピュータの動作にはこれらに加えてシステム全体の制御に関する命令が必要である．

(1) システムコール

システムコールはユーザプログラムと OS プログラムのインタフェースである．システムコール命令を実行すると，あらかじめ決められた手順で OS へと制御が移る．具体的な命令形式や，利用手順はシステムや OS によって様々である．

(2) 専用レジスタ

演算に用いられる汎用レジスタ以外に，システムを制御するための専用レジスタが用いられるが，通常は演算命令から専用レジスタにアクセスすることはできず，専用レジスタへアクセスするために専用の転送命令が用意されている．

(3) モード切替え

幾つかのプロセッサは，1 つのプロセッサでありながら複数の動作モードを持ち，必要に応じて動的に切り替えることができる．例えば，ARM プロセッサ[1)]には通常の 32 bit で動作するモードのほかに Thumb と呼ばれる 16 bit で動作するモードがある．Thumb モードでは実行できる命令に制限があるが，命令長が通常の半分であるので，コード密度が高いという利点がある．これらのモードの変更には分岐命令の一種である BX（Branch and eXchange）命令を用いる．

■参考文献

1) ARM Limited, http://www.arm.com/

■6 群 - 4 編 - 3 章

3-4 命令形式

（執筆者：小林良太郎）[2013 年 12 月 受領]

プロセッサはメモリ上に保持されたプログラムを処理する．このメモリ上のプログラムは，2 進数で表現される機械語命令によって構成されている．機械語命令は，2 進数で表現されることにより，効率的にハードウェア上で処理することが可能となる．命令形式とは，あらかじめ定められた機械語命令の形式である．

命令形式を分類する基準は数多く存在する．以下にその基準を示す．

- 命令長
- 命令フィールド
- op コード
- オペランド

3-4-1 命令長

命令長は，命令を構成するビットの数である．命令長の単位として，ビット，バイト，ワードが用いられる．命令長がどの命令においても固定である場合，固定長命令と呼ぶ．一方，命令によって可変である場合，可変長命令と呼ぶ．

命令長を増加させると，命令の種類やオペランドの数などを増やすことができる．オペランドは，命令が処理するデータである．また，可変長命令は固定長命令よりも命令の種類やオペランドの数などを増やすことができる．その一方で，命令の種類やオペランドの数などが増加すると，命令を処理するためのハードウェアが複雑になる．

命令はメモリ上に保持するため，命令長はバイトの整数倍となる．固定長命令の場合，命令とデータを同一の長さで扱うことができると，メモリ上の命令やデータの扱いを簡単化しやすいため，データの長さ（語長）と命令長を同一とする場合が多い．

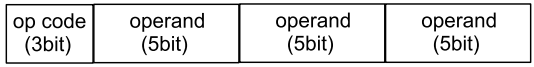
半導体技術の進歩に伴い，プロセッサ上に保持できるデータの量やメモリ上に保持できるデータの量は増加し続けている．これに合わせて語長も増加し続けており，より多くのデータ，より大きなデータ，より精度の高いデータを扱うことができるようになってきている．その一方，命令長増加によるメリットとデメリットを考慮すると，近年では，語長に合わせて命令長を増加させる必要性は低い．

3-4-2 命令フィールド

命令は命令長が n ビットである機械語命令にエンコードされる．機械語命令には最大で 2 の n 乗通りの命令をエンコードすることが可能である．エンコードにおいて，ハードウェアでの処理に適した規則性を持たせることで，ハードウェアを簡単化することができる．その一方で，エンコードに規則性を持たせると，使用されないビット列のパターンが生まれるため，エンコード可能な命令の組合せが減り，ビット列の利用効率は下がる．

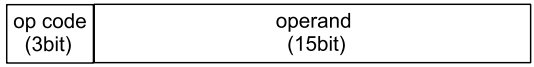
エンコードに規則性を持たせる方法として，命令を幾つかの領域（命令フィールド）に分割するという方法が使われる．命令フィールドを用いる場合，フィールドの数，フィールドの長さ，フィールドに保持する情報の種類がパラメータとなる．フィールドに保持する情報は，op コード，機能コード，オペランドに分類することができる．op コードと機能コードは命令の種類を示し，オペランドはデータを示す．これらについては，次節以降で述べる．エンコードの際に使用するパラメータの組合せの数を少なくするほど，また，パラメータの一部を共通化するほど，ハードウェアを簡単化することができる．その一方で，上記の形式を使用することにより，各フィールドの長さは固定されてしまうため，op コードや operand に使用するビット数に過不足が生まれる可能性が増える．

説明のため，命令フィールドに関する命令形式の例を**図 3･7** に示す．左から順に，フィールドに番号 *n* を付け，それぞれ第 *n* フィールドと呼ぶこととする．



a) 形式1(opコード1つ、オペランド3つ)

b) 形式1(opコード1つ、オペランド2つ)



c) 形式1(opコード1つ、オペランド1つ)

図 3･7　命令フィールドと命令形式

図 3･7 に示した 3 つの命令形式のみを使用する場合，すべての命令について，最も左側の 3 bit が第 1 フィールドとなる．op コードをデコードするためには，第 1 フィールドのみを参照すればよい．一方，命令からオペランドをデコードするためには，op コードに従って，図 3･7 に示した形式のいずれかでビットを切り出せば良い．これらより，命令をデコードするハードウェアは簡単に実現できる．しかし，op コードは 3 ビットに固定され，また，operand は 5, 10, 15 ビットのいずれかに固定されるため，フィールドに無駄，あるいは，不足が生じる可能性がある．例えば，op コードの種類が少なく，2 ビットで表現可能であれば，第 1 フィールドの 1 ビットが無駄になる．逆に，op コードの種類が多く，3 ビットで表現できないのであれば，第 1 フィールドの長さが足りないことになる．

3-4-3 op コード op コードと機能コードは，命令の種類を示す．命令の種類には，演算命令，制御命令（分岐命令，ジャンプ命令），ロード/ストア命令などが存在する．命令は，op コードのみで表現される命令と op コードと機能コードの 2 つによって表現される命令に分けることができる．命令を op コードと機能コードによって表現する場合，複数種類の命令を同

一の op コードで表現し，それらの種類の違いを機能コードで表現する．なお，機能コードは命令形式に必須のものではなく，すべての命令を op コードのみで表現するという選択も可能である．

加算命令や減算命令のように，プロセッサ内での処理内容が，ALU 上での演算を除き，同一となる命令が数多く存在する．こうした命令は，ALU 上での演算は異なる処理となるが，それ以外（例えば命令のデコードやオペランドに関する操作など）は同一の処理となる．機能コードを用いる命令形式においては，上記の性質を利用し，ALU 上での演算処理を機能コードで表現し，それ以外の処理を op コードで表現する．これにより，複数種類の命令を同一の op コードで表現できるようになるため，op コードに必要とされるビット数が減少し，op コードに割り当てるフィールドの長さを短くすることができる．これにより，命令フィールドの利用効率を上げることができる．

説明のため，op コードと機能コードを使用する命令の例を**図 3･8** に示す．第 1 フィールドが 6 ビットの op コード，第 2 フィールドから第 4 フィールドがオペランド，第 5 フィールドが 6 ビットの機能コードである．各フィールド内においては，説明に必要となる op コードと機能コードのビット列のみ示す．

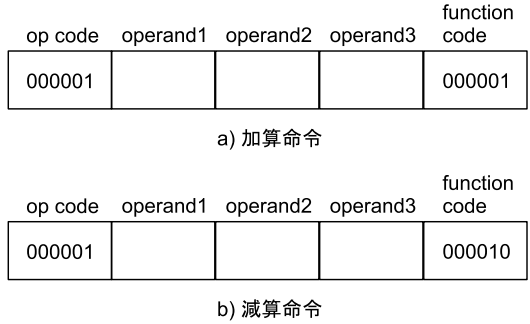


図 3･8　op コードと機能コードを使用する命令

図 3･8(a),(b) は，それぞれ，加算命令，減算命令である．加算命令と減算命令は op コードが同一であるが，機能コードが異なる．ALU 上では機能コードに基づいて加算か減算のいずれかの処理を行うが，ALU 以外の回路（デコーダやレジスタファイルなど）では op コードが同じであるため同じ処理を行う．

3-4-4 オペランド

オペランドは命令の処理するデータであり，種類は以下のとおりである：演算の入力として使われるデータやメモリアドレスの計算に使われるデータ（即値），レジスタ番号を指定するデータ，メモリアドレスを指定するデータなど．オペランドがレジスタ番号の場合，当該レジスタは，ALU の入出力データの保持，メモリアドレスの保持，メモリアドレスの計算に使われるデータの保持などに使用される．オペランドがメモリアドレスの場合，当該メモリアドレスは，ALU の入出力データの保持，別のメモリアドレスの保持などに使用される．上

記のメモリアドレスは，ロード/ストア命令のアクセスするメモリアドレスとして，あるいは，制御命令の分岐先/ジャンプ先となるメモリアドレスとして使用される．

オペランドの数に基づいて命令形式を分類することができる．命令内のフィールドに割り当てられているオペランドの数が n の場合，当該命令は n アドレス命令と呼ばれ，当該命令形式は n アドレス方式と呼ばれる．例えば，図 3･7 において，(a) は 3 アドレス命令，(b) は 2 アドレス命令，(b) は 1 アドレス命令となる．図 3･8 において，(a) と (b) は 3 アドレス命令，(b) は 2 アドレス命令，(b) は 1 アドレス命令となる．

以下に，命令が使用できるアドレス方式によって，計算の処理の仕方がどのように変化するかを説明する．

まず，加算命令が使用できるアドレス方式を 3 アドレス方式とした場合を考える．この場合，1 つのオペランドを使って出力データを保持するレジスタ番号を 1 つ指定し，2 つのオペランドを使って入力データを保持するレジスタ番号を 2 つ指定することによって，2 入力 1 出力の加算を行うことができる．例えば，レジスタ番号 A〜D を用いて表現された $D = A + B + C$ という計算は，A と B の加算を行う命令を 1 つ実行した後，その結果と C の加算を行う命令を 1 つ実行することで実現できる．

次に，加算命令が使用できるアドレス方式を 2 アドレス方式とした場合を考える．この場合，使用できるオペランドが 2 つに制限されるが，入力データの一方を保持するレジスタ番号と出力データを保持するレジスタ番号を同一にすることによって，2 入力 1 出力の加算を行うことができる．したがって，2 アドレス方式においても，$D = A + B + C$ という計算を 2 つの加算命令で実現することができる．ただし，加算命令を実行するたびに，入力データの一方が出力データによって上書きされてしまうため，入力データがともに別の命令によって使用される場合には，入力データを別の領域に移動させる命令などを実行する必要がある．

このように，命令の使用できるアドレス方式が制限されていくにつれて，データを移動させる回数が増えていくため，実行しなければならない命令数が増えていく．その一方で，使用できるオペランドの数が減っていくため，プロセッサ内でデータを保持するレジスタやスタックなどの記憶領域は減っていく．

■6 群 - 4 編 - 3 章

3-5 アドレス方式

（執筆者：小林良太郎）[2013 年 12 月 受領]

アドレス方式（アドレッシングモードとも呼ばれる）は，命令からメモリアドレスを指定する方式である．指定するメモリアドレスは，命令を保持するメモリアドレス（命令アドレス）とデータを保持するメモリアドレス（データアドレス）に分けることができる．制御命令は命令アドレスを指定し，それを PC に書き込む．メモリ命令はデータアドレスを指定し，そのアドレスにアクセスする．

アドレス方式においては，以下 3 つの要素が重要となる．オペランドの数，各オペランドの種類，オペランドからメモリアドレスを導出する方法である．オペランドの数は 0 以上である．各オペランドの種類は，即値，レジスタ番号，メモリアドレスである．オペランドからメモリアドレスを導出する方法においては，オペランドの数や種類に応じて，レジスタへのアクセス，加算，連結などの操作を行う．

以下では，実際に用いられるアドレス方式の例を示す．

図 3･9 に直接アドレス方式を示す．この方式では，命令のオペランドがメモリアドレスを直接示す．オペランドの長さで指定できるアドレス空間が制限される．

これに対し，疑似直接アドレス方式では，PC の上位ビットと命令のオペランドを連結して得られた値でメモリアドレスを示す．PC の上位ビットを用いることで，直接アドレス方式よりもアドレス空間を広げることができる．

図 3･10 に PC 相対アドレス方式を示す．この方式では，PC の上位ビットと命令のオペランドを加算して得られた値でメモリアドレスを示す．分岐命令はこの方式を用いて分岐先となるメモリアドレスを指定する．

図 3･11 にレジスタ間接アドレス方式を示す．この方式では，命令のオペランドがレジスタ番号を示しており，このレジスタ番号でレジスタファイルにアクセスして得られた値でメモリアドレスを示す．分岐命令はこの方式を用いて分岐先となるメモリアドレスを指定する．

図 3･12 にレジスタ間接アドレス方式を示す．この方式では，命令のオペランドがレジスタ番号と即値を示しており，このレジスタ番号でレジスタファイルにアクセスして得られた値と即値を加算して得られた値でメモリアドレスを示す．ロード/ストア命令はこの方式を用いてアクセスするメモリアドレスを指定する．

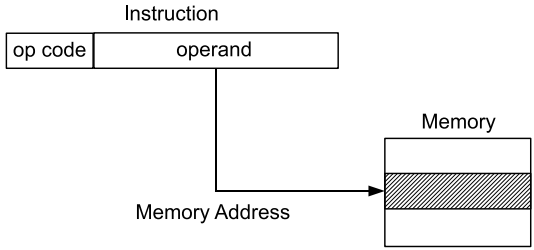


図 3･9　直接アドレス方式

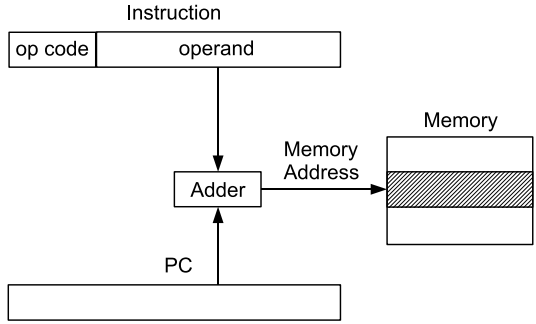


図 3･10　PC 相対アドレス方式

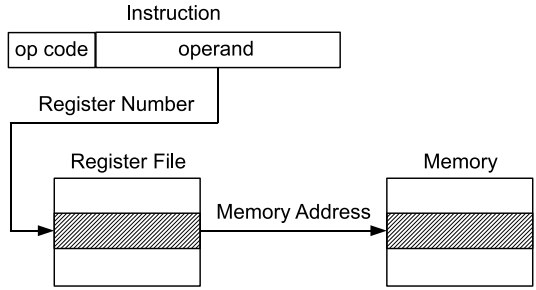


図 3･11　レジスタ間接アドレス方式

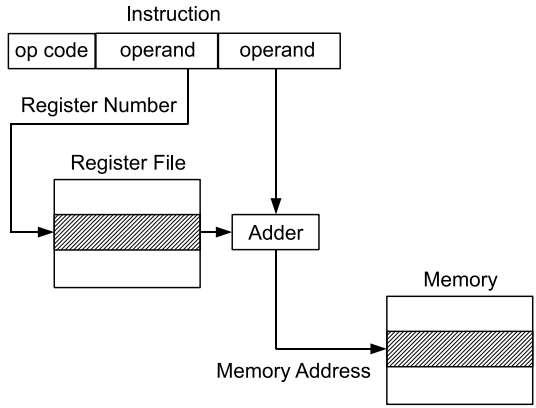


図 3･12　レジスタ相対アドレス方式