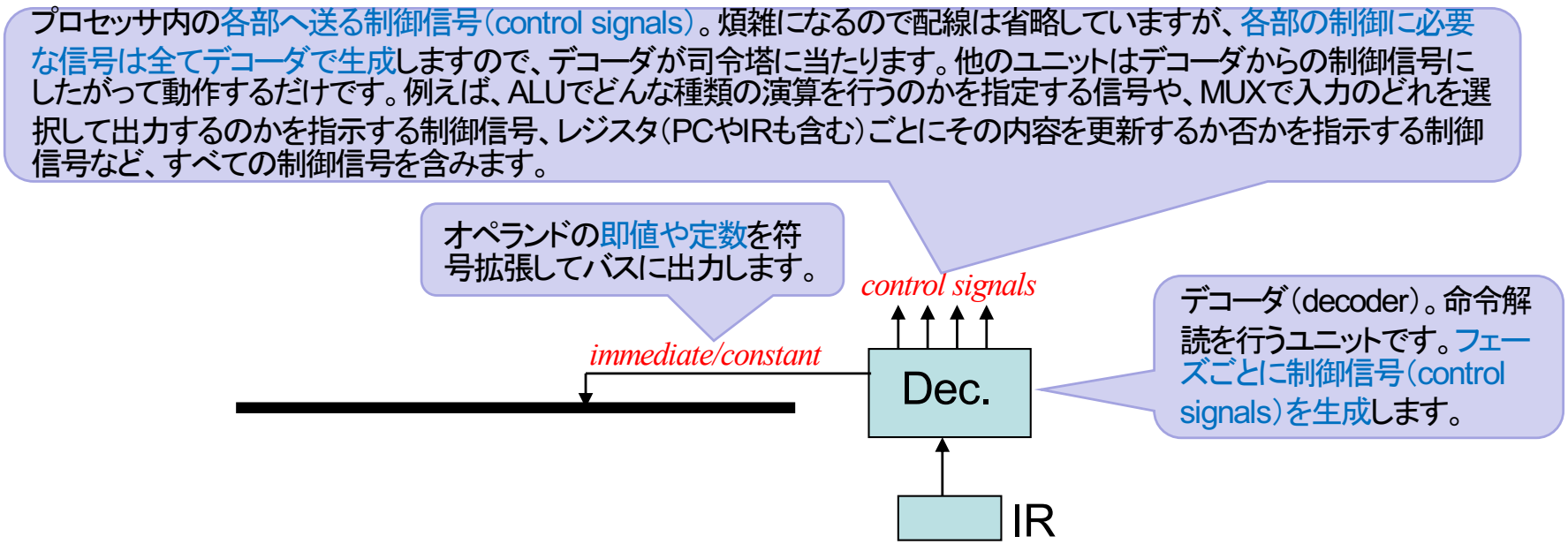
では、次の話題です。

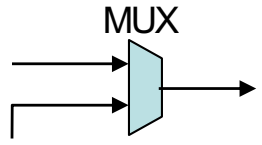
- 続いて、コンピュータの制御方式について考えます。

- まず、教科書の対応範囲は、以下の通りです。
  - 5章始め～5.1.4項（p.123～133）

- 再び、先ほどのハードウェア構成をもとに考えますが、その中で、デコーダが出力する信号に着目します。
  - 以下は、前のページのコピー（一部）ですが、強調したい部分を着色しています。

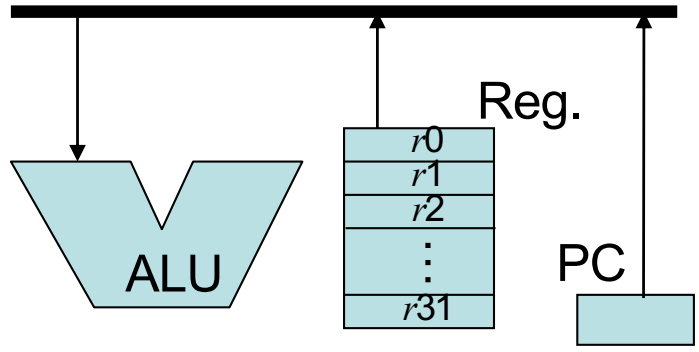


- 制御信号として、デコーダはどんな信号を出力する必要があるでしょうか？　以下に挙げていきます。
  - ALUに対する命令：　加算か減算かの2種類なら1ビットで指示できますが、他の演算機能を備えるALUなら、その種類に応じたビット数が必要になります。
    - なお、ALUを使用しないフェーズでも、なんらかの信号（命令）は送らなければなりません。もちろん、論理回路としては「Don't Care」で設計して構いません。
  - レジスタの書き込み指示信号：　各レジスタに対して、そのフェーズの最後のクロックの立ち上がりで、入力値を記憶するか否かを指示しなければなりません。
    - 先のハードウェア構成では、IR、MDR-r、MDR-w、MAR、PCと32本の汎用レジスタがありますので、合計37ビットの書き込み指示信号を出力しなければなりません。
  - マルチプレクサの選択信号
    - 先のハードウェア構成では、マルチプレクサが1個あります。上側と下側のどちらの入力を選択して出力するかを指示する1ビットの信号が必要です。

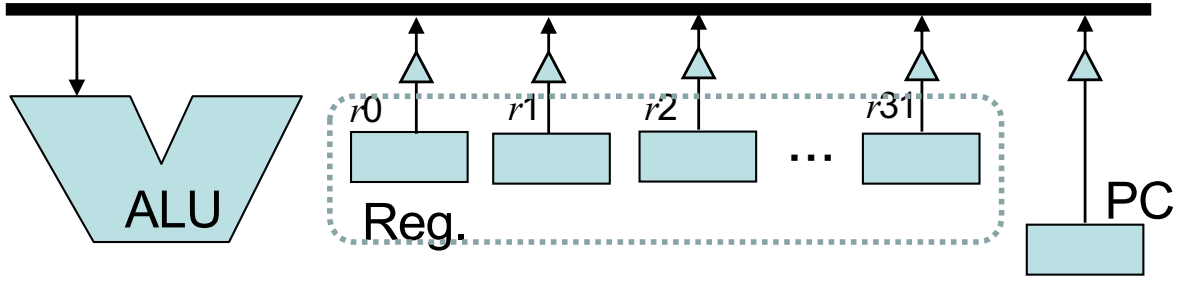


– バス制御信号： バスに出力するデータを選択するための信号。

- 例えば、ALUの左側入力のバスには、PCと32本の汎用レジスタからの出力が接続されています。



- 先（上）の図では32本の汎用レジスタを1まとめにして描いていましたが、実際のハードウェア構成は以下のようになります。



- 合計33本のレジスタのうち、どのレジスタの内容をバスに出力するかを選択しなければなりません。各レジスタとバスの間にゲートを設け、このゲートを開けることによってデータをバスに出力します（同時に2個以上のゲートを開けると故障します）。ゲートごとに開閉を支持する1ビットの制御信号が必要です（合計33ビット）。

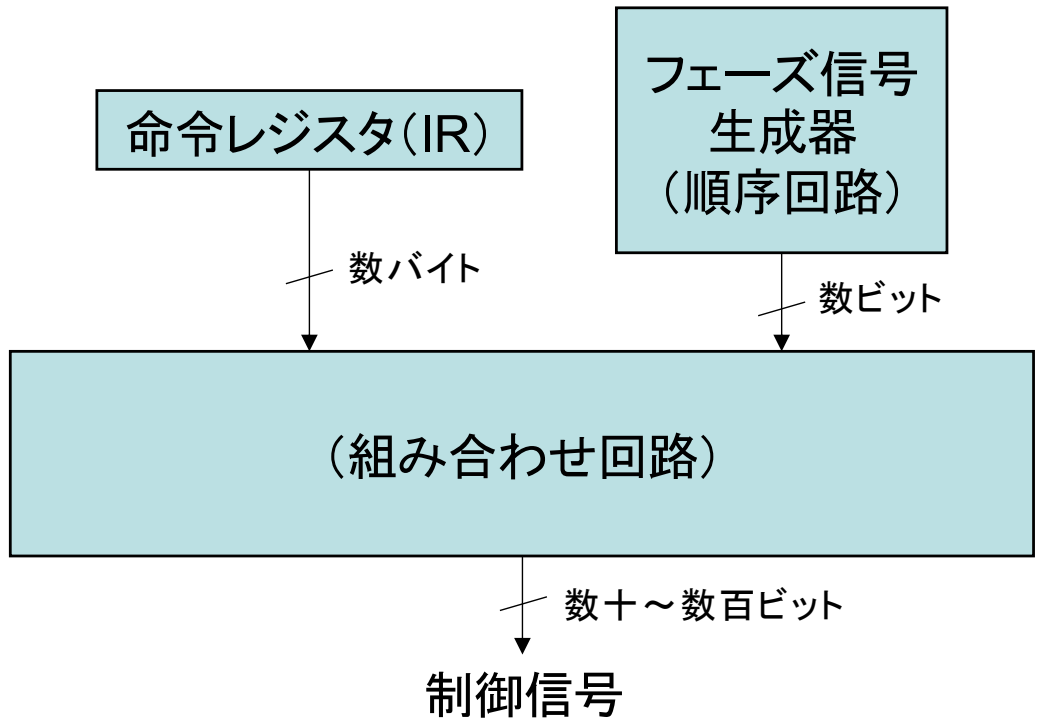
バス中のある1本の線に対して、一方が0（0v）を、他方が1（5v）を出力すると電気的に短絡（ショート）して壊れてしまいます。ゲートを閉じているときは、ゲートの出力をハイインピーダンス状態にして、入力が0でも1でも、また、他のゲートから0が出力されていても1が出力されていても、問題が生じないようにします。

    - 結局、このバスに対して、合計33ビットの制御信号が必要になります。
        - バス構成にせずに、33-to-1のマルチプレクサを設けても構いません。
    - 他のバスについても、同様に、出力数に応じた制御信号が必要になります。
- メモリアクセスのための制御信号： メモリに対して、少なくとも書き込みを行うのか否かを支持する1ビットの制御信号が必要です。
    - デフォルトで「読み出し」を指示します。利用しないときは、読み出したデータを捨てる（どこにも格納しない）だけです。
    - 実際には、「書き込みを行うのか否か」以外に、いくつかの制御信号が必要です。

- 以上のように、数十～数百ビットの制御信号を、毎フェーズ、生成しなければなりません。
    - 当然、これらの制御信号は、ハードウェアで、つまり、論理回路で生成します。
    - そのようにしてコンピュータを制御する方式を配線論理制御（wired-logic control）方式と言います。
    - しかし、あるとき、この「当然」が当然でなくなるアイデアが出てきました。

配線論理制御方式

- 命令の実行をハードウェアのみで制御
  - 組み合わせ回路で、動作フェーズ毎に、毎サイクル、制御信号を生成する。



- しかし、高機能で複雑なマシン命令を実装しようとすると、制御信号を生成する論理回路も複雑になります。
- 結局、「設計できない」「設計できても、制御信号の生成に時間がかかって、高速に動作できない」という結果に。
- そこで、発想の転換が必要に。。。
  - 制御信号は、命令の処理時に、毎回、論理回路で生成しなければいけませんか？
  - 生成すべき制御信号をメモリ（これを「制御メモリ（control memory）」と呼ぶ）に記憶しておき、それを読み出して使用すれは、毎回生成しなくても済みます。
- このようにして考え出されたのが、マイクロプログラム制御（microprogrammed control）方式です。

マイクロプログラム制御方式

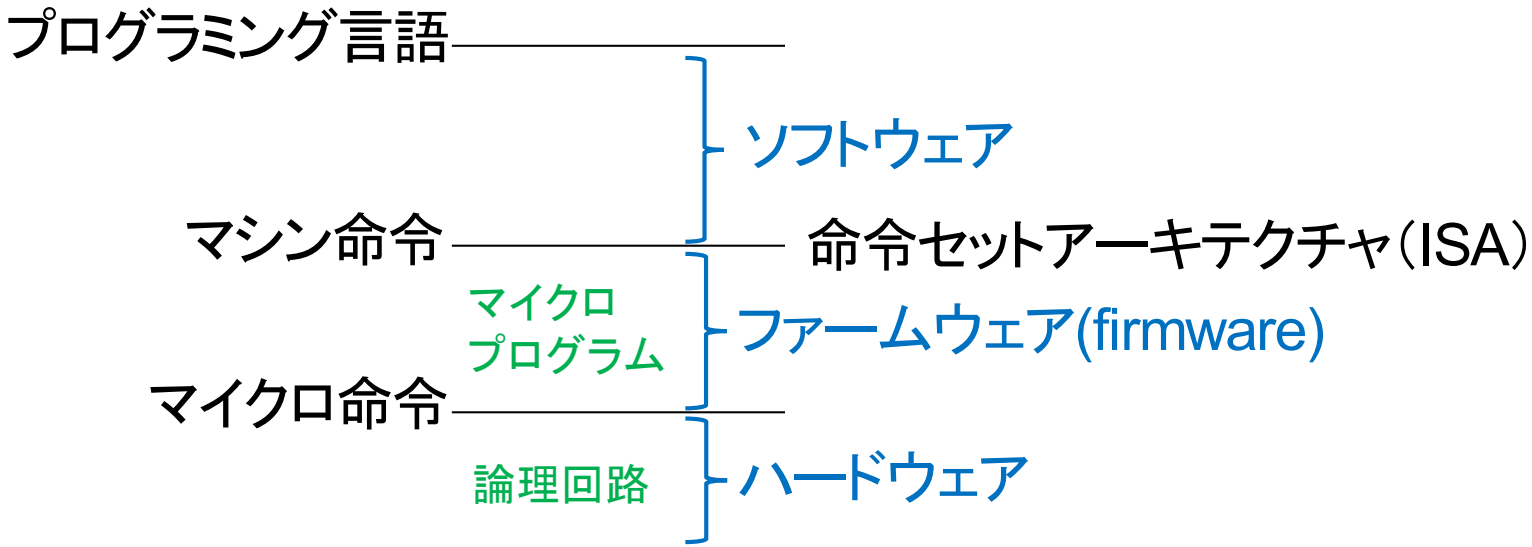
- これまで「制御信号」と呼んでいたものをマイクロ命令とみなす。
- フェーズ毎にマイクロ命令を実行する（制御信号そのままなので、実際には、実行というほどのことはしていないが）。1個のマシン命令を複数のマイクロ命令（列）を実行することで処理するので、これをマイクロプログラムという。
- マシン命令をマイクロプログラムで模擬実行（シミュレート; simulate）することを「エミュレート（emulate）する」という。

- マイクロプログラム制御機構
  - 要点のみ説明します。教科書の図5.6（p.130）を参照してください。
  - 制御メモリは、プロセッサ内に専用のメモリとして設けます。
    - この中にマイクロプログラムを格納します。
    - 一般ユーザは、制御メモリを読み書きすることはできません。

1. マシン命令のOPコードから、そのマシン命令をエミュレートするマイクロプログラムの先頭アドレス（制御メモリのアドレス）を求めます。
2. あとは、マシン命令のプログラムを実行するのと同様の仕組みで、マイクロプログラムを実行します。
3. そのマシン命令のエミュレーションを終えると、命令フェッチ用のマイクロプログラムに分岐します。
4. 次に実行するマシン命令をフェッチしたあとは、上記1.の処理から繰り返します。

マイクロプログラム制御方式の特徴

- ソフトウェアとハードウェアのインタフェース
  - 従来は命令セットアーキテクチャがソフトウェアとハードウェアの境界でした。
  - マイクロプログラム制御方式の登場により、ファームウエア（firmware）と呼ぶ領域ができました。
    - マイクロプログラムをファームウェアと呼んでいましたが、現在は、機器に組み込んだ制御プログラム（ユーザが書き換えることは不可）のことをファームウェアと呼ぶことがあります。



- 以下のように、ハードウェアとソフトウェアの間のトレードオフについて、変更の可能性が生まれました。
  - 複雑なマシン命令を設けることが可能に！
    - 「ソフトウェア機能のハードウェア化（高速化）」と捉えることができます。
    - 三角関数のマシン命令などは、マイクロプログラムを前提にして設けられました。
  - ハードウェアはそのままで、命令セットアーキテクチャを変更することが可能に！
    - 「ハードウェア機能のソフトウェア化（柔軟化）」と捉えることができます。
    - 汎用コンピュータのハードウェアでも、使用目的に合わせて専用命令を追加することで、性能を上げることができました。
  - ユニバーサルホストマシンの登場
    - ユーザによるマイクロプログラミングを可能にしたコンピュータをユニバーサルホスト（universal host）マシンと言います。
      - 命令セットの変更によって色々なマシンに変身できるので、「カメレオンマシン」とも呼ばれることがありました。
      - しかし、設計当初からサポートすることを考えていなかった命令セットをマイクロプログラムで実装することは、実際には容易ではなく、また、実装できても性能は出ませんでした。
    - 命令セットを変更するということは、ソフトウェア（プログラマ）に対して別のマシンであると見せかけること。このことより、「仮想マシン（virtual machine）」という術語が使われ始めました。
      - 現在は、一般には、「仮想マシン」はこれとは違う意味で使われています。

おわりに

- コンピュータが登場した当初は、命令セットも小規模で、コンピュータハードウェアも単純でした。これは半導体など実装技術の制約もあってのことでした。
- その後、実装技術の進歩に伴い、コンピュータも高機能化・複雑化の一途をたどります。ここで、マイクロプログラム制御方式は大きな貢献を果たしました。
- しかし、現在はマイクロプログラム制御方式は使われていません。その理由として挙げられるのは、
  - 制御メモリとして使用するメモリ素子の性能の優位性が薄れてきたから。
  - その後に開発されたプロセッサの高速化技術と合わないから。
- それでも、マイクロプログラム制御方式の考え方は、形を変えて現在のコンピュータシステム技術に活かされています。