第1章
ディジタル回路設計の全体的な流れ
HDLによる設計の概要と本書の構成

この章では，HDLを使ったディジタル回路設計の全体像をつかんでいただきながら，本書の構成を紹介しておきたいと思います．

1-1 本書のねらい

● ねらいは現在の回路設計のセンスをつかむこと

本書のゴールは「プロの卵として第一歩を踏み出せる」というレベルです．難しいと思われるかもしれませんが，適切な内容を選んで学習を進めれば，さほどでもありません．

本書の文章を第1章から読むだけでは退屈なはずですので，出てくる回路を実際にCAD(メーカのWebサイトから入手できる)にかけたりして体で覚えていけば，きっと飽きることなくスムーズに学習が進むでしょう．もしわからない箇所にぶつかったときは，そこで止まらずに，飛ばしてどんどん先に進んでください．

● 従来の入門書と本書とのスタンスの違い

本書は，従来のディジタル回路の入門書と比べて，次の①②で大きくスタンスを変えました．

▶ その①：ゲート・レベルでの回路作成法の説明は必要最低限に抑えました．

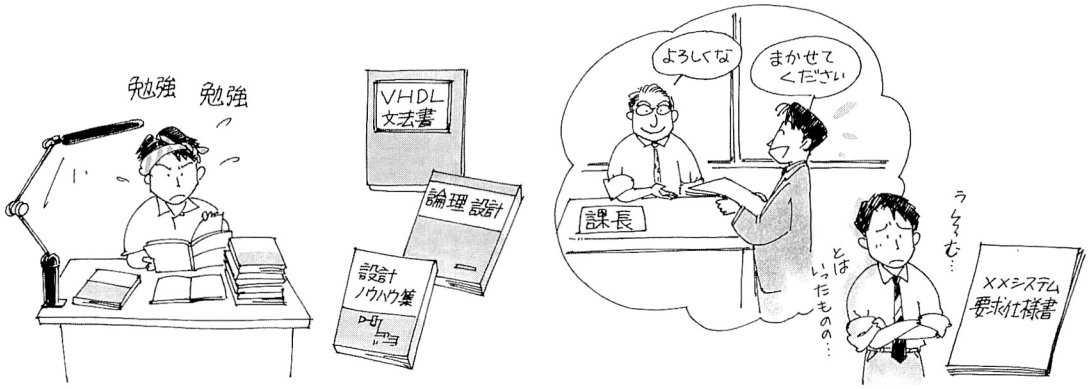
具体的には，カルノー・マップやクワイン・マクラスキー法などの論理圧縮法や，ステート・マシンをゲート・レベルの回路に直す方法などを省略し，その代わりハザードや非同期入力の扱いなどにページを割きました．

また，初学者がゲート・レベル設計から学習を始めてしまうと，現在実際に行われている回路設計の説明に到達するまで非常に長い道のりとなってしまいます．本書のスタンスは，たとえばJavaプログラミングをマスタしようとしたとき，2進数→アセンブラ→FORTRAN→C→…というように歴史に従って下から勉強を進めるのではなく，まずJavaから入って必要に応じて下へ降りる，というものです．

▶ その②：良いハードウェアをどのような考えかたで作るかを説明します．

HDL(ハードウェア記述言語，詳しくは1-2節)を使って，どのような考えかたをして回路を作っていくのか，良い回路に仕上げるためにはどのようなアイデアを用いるのか，といった回路設計のセンスを説明します(図1-1)．

〈図1-1〉ディジタル回路設計で使う考えかたをつかもう



（a）細かいノウハウはあれば
あるほど良いのだが…

（b）ノウハウの根本にあるものの考えかたを
知らないと実設計には手がつけられない

1-2　現在のディジタル回路の設計現場では

● 回路で行う処理が高度になってきた

設計者に要求されるのは，性能の優れた回路，つまり速くて小規模で消費電力の少ない回路を，スピーディにバグなしで作ることです．これは昔も今もまったく変わりません．

もっとも変わったのは，作らなければならない処理の内容が昔とは段違いに複雑になり，多様化してきたことです．とくに，ちょっとしたソフトウェアなみのデータ処理を回路で実現することが，ごく普通になってきました．たとえば，CPUコア，画像/音声処理，プロトコルの上位レイヤ処理，誤り訂正符号処理，暗号処理などです．FPGAメーカのホームページなどを見ると多くのIPコア(いわゆる回路部品のこと)が販売されていますが，それらの多くはこうしたデータ処理を行う回路です．いろいろな機械の制御シーケンサなど，昔から作られているような回路も多くありますが，それらもけっこう高度な処理をするようになってきています．

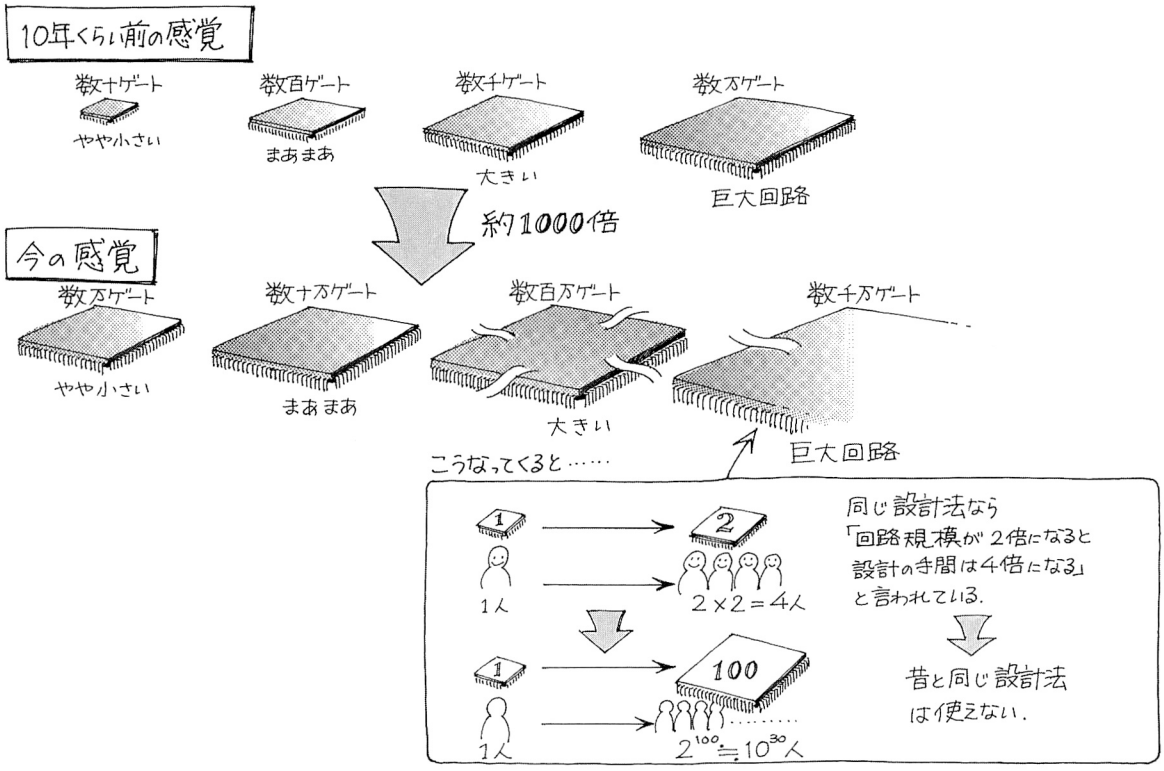
このような変化が起こったのは，デバイスの規模と速度が格段に進歩したからです．データ処理回路は，一つの処理を行うだけでも数万～数十万ゲートをあっという間に消費してしまうので，昔は作りたくても作れませんでした．今ではそれが可能になったわけです．10年くらい昔と比べると，ゲート数の感覚が1000倍くらい変わりました(**図1-2**)．デバイスのスピードも速くなって，ASICなら数百MHz～1 GHz，FPGAなどでも数十MHz～400 MHzは珍しくありません．

● HDLと自動論理合成CADの進歩

このような複雑な処理を行う回路を設計できるようにするために，HDLと設計自動化CAD(正確にはEDAツールという)が開発されてきました．多少の細かい無駄には目をつぶり，回路の大部分はコンピュータに設計させてしまうのです．人は概略設計だけをすればよく，あとはコンピュータが苦手な箇所(1-4節も参照)だけに手を入れるようにします．

今でもアセンブラがなくならないのと同じく，人が直接ゲートをいじる機会も残っています(将来も残るでしょう)．しかし，すべてをゲート・レベルで組まなくてもよくなったということは大きな進歩です．

〈図1-2〉10年前と現在のディジタル・システムの規模に対する感覚の相違



● 動作が複雑な回路を作るならHDLやCADは必須

このような動向のなかで，現在，HDLやCADをまったく使わずに回路設計をするのは，現場では考えられない状況になっています．

先述のような処理内容の複雑な回路を作るときは，たとえその回路規模が大きくなくても，適切な抽象度でCADを使って設計をしないと，とてもデバッグしきれません．もし複雑なプログラミングで(たとえば「Linuxを作る」とき)，コンパイラがなくてアセンブラしかなかったら，いつバグのないプログラムが完成するでしょうか?　回路でも同じです．

また，同じ処理でも速度などの性能が異なる回路をいろいろ取り揃えたい，少し機能を追加/削除したい，ざっと性能の見積もりをとりたい，といった局面も多くあります．CADを使わずに毎回毎回ゲート・レベルで回路を手設計していては，こうした要求にまったく応えられません．

● HDLと実際にできる回路との対応をつかむのが最大のポイント

その一方で，現場の回路設計では，なんでもいいから回路を作れば終わりということはまずなくて，回路速度や回路サイズに対する要求が必ずついてまわります．また，他の回路モジュールやチップとうまくつながるように調整する必要などもあります．実際に，CADの自動論理合成で得られる回路では性能が足りず，苦労させられる場合が多々あります．

回路の性能や信頼性を上げる作業は，単にHDLの文法やCADの使いかたを知っているだけではまったく不足です．「性能を上げるためには回路をどういじるべきか」「どうHDLを書くとどんな回路になるか」

というイメージが，自分の頭の中にないとできません．HDL設計を学ぶうえでは，これが最大のポイントです．おもに第3章と第6章で説明します．

● C言語とRTL

残念ですが，現在のところ，C言語のようなソフトウェアの記述から回路を自動で作るのは一般的ではありません．1-3節で説明しますが，RTL(Register Transfer Level)といって，もう少しハードウェアに近いHDLコードの書きかたをする必要があります．

近い将来，C/C＋＋言語の拡張版がHDLの代わりに使われるようになるかもしれません．実際に，現在System CやSpec Cなどの策定が進んでいます．ただし，これらはC/C＋＋と同じ文法で回路動作やRTLを書けるということであって，C＋＋/Cプログラミングとまったく同じセンスでハードウェアを設計できるようになるわけではありません．良い設計をしたり確実に動く設計をするにはハードウェアのセンスが必要なままなので，そこは誤解しないでおく必要があります．むしろ，ソフトウェアで良いアルゴリズムほど，ハードウェアに向かない傾向があります．

1-3　回路設計作業の全体的な流れ

それでは，以下，現在の回路設計の作業の流れと注意点について，順を追って紹介していきます．

各項目が本書の何章で詳しく説明されるかも示します．本書の内容は，「とりあえず動く回路を作るための知識」(第5章まで)と「回路性能を上げるための知識」(それ以降)に分けられます．

● いきなり回路を作り始めない

たとえばディジタル時計を作る場合，「60進カウンタを使って秒や分を数えればいいかな」と処理の進めかたを考えつくまでに，時間はかからないと思います．カウンタを組み合わせながら回路を考えていく，という方もいるでしょう．しかし，もう少し仕様(スペック)が複雑になってくると，ゲート・レベルで回路をいじりながら仕様を実現していく方法は，すぐに通用しなくなります．

そこで，以下のような基本手順に従い，処理アルゴリズムといった大枠の決定から始めて，必要に応じて細部を詰めていきます．

● 作業1…実現したい処理の回路仕様を決める

どのような処理をする回路を設計するのか，回路の仕様をきちんと決めます．ここをいい加減にすませると，後でいろいろな誤解やトラブルが起きるもとになります．

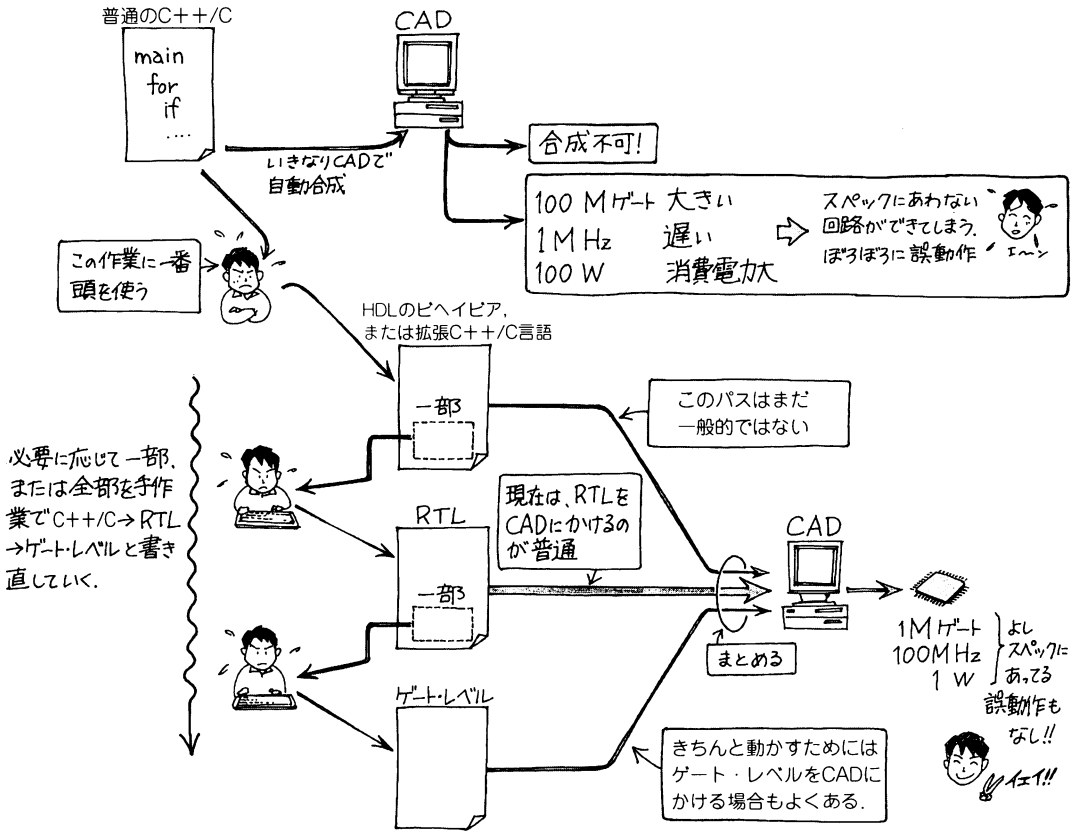
多くの場合，回路入出力のタイムチャートや状態遷移図，要求性能，回路内部のブロック図などをあわせたものが仕様です．

● 作業2…モジュールの構成を考える

通常，複雑な回路の設計では，全体の処理をいくつかの機能単位で分割し，それぞれを独立した回路モジュール(いわゆるIPコア)として設計/テストしてから，最後に各モジュールを結合して全体のテストをします．

このとき，各モジュールの性能をどの程度にして，各モジュールをどうつなぎ(全体のバス・アーキテクチャをどうするか)，どのような手順でデータの交換をするか，ということを検討します．それによっ

〈図1-3〉現在のディジタル回路設計作業の流れ



て，各モジュールの仕様が決まってきます．モジュールの接続については，第4章で説明します．

以下，1個のモジュールの設計手順です(**図1-3**)．

● 作業3…モジュールのビヘイビアを考える

どのようなアルゴリズムで処理を進めていくかを決めます．このことをハードウェアの世界では「ビヘイビア」といいます．

先述のように，現在はビヘイビアをC++/C/HDLで書いてもCADにかからないのが普通ですが，C言語などでアルゴリズムをあらかじめチェックする，ということはよくやります*1．これを第2章の前半で説明します．

● 作業4…クロックごとにレジスタに何を代入するかを決めてRTLソースを記述する

ビヘイビアをもう少し煮詰めて，回路で行う処理をクロック単位で正確に決めます．このようにクロ

*1：このCプログラムは，回路のテスト・ベクトル生成に使ったりすることもできるので，無駄にはならない．

ック単位の処理が決まっている設計レベルを，RTL(Register Transfer Level)といいます．

現在のところ，ここまで人手でやれば，あとの作業はCADで自動化されていますので，要所だけ人手を加えるようにすればよいです．

以下，この作業4の内訳についてもう少し詳しく説明します．

▶ **作業4-1**…ビヘイビアを解析して1クロックでどれだけの処理をするか決める(スケジューリング)

ビヘイビアでは，普通はループ実行が使われていますし，クロックという概念もありません．そこで，ビヘイビアを解析して，1クロックぶんの処理を区切っていきます．これをスケジューリングといいます．第2章の後半で説明します．

また，回路の性能向上のために，処理をパイプライン化したり並列化したりすることもあります．おもに第7章で説明します．

COLUMN　用語解説

基本的ないくつかの用語について，その意味をまとめます．実は技術者であっても，人によって違った定義をしていたりすることが頻繁にあります．普通，細かいことを言わなくても十分話は通じますが，思わぬ誤解のもとになる場合もありますので，あまりいい加減にはしないほうがよいかもしれません．

● CAD(Computer Aided Design) と EDA (Electronic Design Automation)

本書で「設計自動化CAD」と言っているものは，正しくはEDAツールといいます．厳密には，EDAは設計そのものをコンピュータがやること，CADは設計作業をコンピュータが補助(支援)することです．本書ではあまり区別せずに使います．

● ASICとFPGAとPLD

ASICというのは「ユーザが自分専用に作ったIC」ということで，広い意味では，FPGAもPLDもASICです．

▶ ASIC

しかし普通は，ASICはカスタム，スタンダードセル，ゲートアレイの3者を指します．これらは，いわゆる普通の(V)LSIのことで，シリコン・チップの上にゲートを作って配線し，回路を作り込みます(3者の違いはここでは省略．作りかたと性能が違う)．

チップ・ベンダにRTLを持ち込むと作ってくれます．できる回路の性能は高いですが，コストも1億円やそれ以上のオーダでかかります．いったん起こしたチップ上の回路は修正できないので，バグがあると取り返しがつきません．これがハードウェア開発とソフトウェア開発の大きな違いの一つです．

▶ FPGA，PLD(CPLD，SPLD)

FPGAやPLDは，内部のロジックをユーザが何度も自由に変更できるデバイスです．あらかじめチップ上に基本的なゲート(会社によってロジック・セルやスライスなど，いろいろ呼びかたがある)を組み込んでおき，それらの結線情報をSRAMやEEPROMなどのメモリに記憶しておくものです．ユーザがメモリの内容を変更すると，内部の結線を変更できます．

性能を上記のASICと比べれば，回路速度は数分の1～10分の1以下しか出ませんし，容量もあまりありません．デバイスのデータシートにはたとえば100万ゲート相当と書いてあっても，実際にASICへもってくると10分の1～100分の1くらいにしかなりません(回路にもよる)．値段は10万円オーダのものから百円オーダのものまで，さまざまです．

FPGAとPLDは内部構造が少し違うので設計テクニックも異なりますが，それよりは容量の違いのほうが目につきます．FPGAは大～中規模の回路，PLDは中～小規模の回路向けです．PLDには，その規模によってさらにCPLDとSPLDの違いがあります．SPLDは10年以上昔のPALやGALなどです．

Altera社やXilinx社のものがよく使われます．FPGAやPLDはボードやチップの試作に使う場合が多いですが，製品に使うこともよくあります．最近，かなり実用に耐えるCADが無料で配布されるよう

▶ **作業4-2**…モジュールの入出力部の回路を検討する

モジュールの入出力部分の回路にかなり注意しないといけない場合があります．もし作っている回路が複数のクロック信号をもっていたり，あるいはチップ外部とのインターフェースであるなら，入出力の信号タイミングやハザードに注意を払って設計しないと誤動作する危険があります．第5章で説明します．

▶ **作業4-3**…HDLでRTLソースを書く

上記の作業を踏まえて，HDLでRTLソースを書きます．VHDLでRTLを書く方法を第3章の前半で説明します．他のHDLによる記述法についてはAppendix Bを参照してください．

▶ **作業4-4**…回路の無駄をとって性能を上げる

できたRTLソースをCADで合成すると，基本的にはHDL記述の内容を「そのまま」反映した回路が

になったり，デバイスも安価になったりして，とても面白い世界になってきました．アマチュアにも普及してきています．

● **ビヘイビアとRTLとゲート・レベル**

▶ ビヘイビア

クロックの考えかたがなく，プログラムと同じように処理手順を記述したレベルです．言語はC＋＋，C，HDLなどを使います．ただし，普通のソフトウェアのプログラムと違って，いくつかのプロセスが並列に動作するのがビヘイビアです．これは，複数の回路モジュールが並列に動作することに対応しています．

これと基本的に同じ内容ですが，以下のような定義もあります．

(1) イベント駆動型で回路動作を記述したレベル
(2) どの信号が物理レジスタになるかがアサインされていないレベル
(3) 処理アルゴリズムは決まっているが実際の演算タイミング(スケジューリング)が決まっていないレベル

▶ RTL

クロックごとの回路動作を明確に決めたレベルです．RTLと言っても，さらにいくつかレベルの差があります．つまり，

(a) 回路アーキテクチャ，つまりデータ・パスのブロック図が概要しか決まっていない(部品の共有などをしていない)．
(b) ブロック図が完全に決まっている．
(c) さらに，ブロック図中の部品の内部ロジックが決まっている．

というぐあいです．(c)はいわゆるゲート・レベルと重なります．

基本的に同じ内容ですが，以下のような定義もあります．

(1) 物理レジスタを決めて演算スケジューリングをしたレベル
(2) クロック駆動型で回路動作を記述したレベル

なお，シンセサイザブルという意味でRTLという言葉を使うこともよくありますが，厳密には違います．本文で述べたとおり，RTLでもシンセサイザブルだとは限りません．

▶ ゲート・レベルとは

個々のデバイスに合わせてテクノロジ・マッピングをし，どのようなプリミティブ(ゲート)をどう接続するかが決まったレベルです．ネット・リストとも言います．

ただし，テクノロジ・マッピングする前であっても，回路が論理式で書かれていれば，それをゲート・レベルと呼ぶこともあります．論理レベルという場合もあります．

● **論理合成とビヘイビア合成**

通常，ビヘイビアからRTLを自動合成することをビヘイビア合成(なぜかRTL合成とはあまり言わない)，RTLからゲート・レベルの回路を自動合成することを論理合成と呼んでいます．合成のことをシンセシス(synthesis)とも言います．

できます．このことを考えて，演算子(演算回路)を使い回したり，演算回路やセレクタなどの接続順序を工夫したりします．このようにすることで，CADは良い回路を出してくれます．

HDLによるRTLソースと回路との対応について第3章の後半で説明します．また，回路性能が何で決まるかについて第6章で説明します．これら二つは必須の知識なので，ぜひ読んでください．具体的な性能向上テクニックを第7章と第8章で説明します．

● 作業5…必要に応じてゲート・レベルで回路を設計する

おもに演算回路では，CADによる自動論理合成では良い回路が出なかったり時間がかかりすぎたりするために，論理ゲートを手で組み合わせて回路を作る必要が出てくることがよくあります．また，CADがサポートしている演算は限られているので，サポートされていない演算回路は自前で作らざるをえません．こうしたとき，HDLでゲートの組み合わせかたを直接書きます．

演算回路の定石や，演算回路を組むうえでのヒントを第9章で説明します．

● 作業6…チップを作る

ここからあとは，ほぼCAD任せになります．ロジック設計者が積極的に手を出すことは多くないので本書からは外しましたが，ここで簡単に説明します．作業内容はデバイスによってかなり違います．

▶ 作業6-1…テクノロジ・マッピング

テクノロジ・マッピングとは，ANDやORといった論理演算をどの素子(プリミティブ)でどのように組み合わせて実現するかを決めることです(**図1-4**)．たとえば論理式で，

(A　and　B) or (C　and　D)

などと書かれていたとき，それをANDプリミティブ2個とORプリミティブ1個で作るのか，それとも，AO(AND-OR)プリミティブ1個で作るのか，というようなことです．

ASICの場合には，ANDひとつ取っても，入力数や出力ドライブ能力(ストレングス)の異なる*2いくつかのプリミティブが用意されていて，回路のどこにどれを使うかで回路性能が大きく変わります．FPGAやPLDの場合には，プリミティブはロジック・セル(PLA＋フリップフロップ)などもっと大きな単位になるのが普通です．

テクノロジ・マッピングのときに回路の遅延などを見積もって，どの程度の性能が出るか予測を立てます．このときの予測を，仮配線結果と呼ぶこともあります．

▶ 作業6-2…テスト設計の導入

ASICでは，作ったシリコン・チップが不良品か正常品か，製造時にテスト(ロジックのシミュレーションとは違う)を行って調べます．各ゲートがうまく製造できているかを調べるのですが，このための補助回路をテクノロジ・マッピング後に追加するのが普通です．何パーセントのゲートがチェックできるかをテスト・カバレッジといって，これが規定水準に達しないと製造できないので，かなり大変な作業になることがあります．本書では省略します．

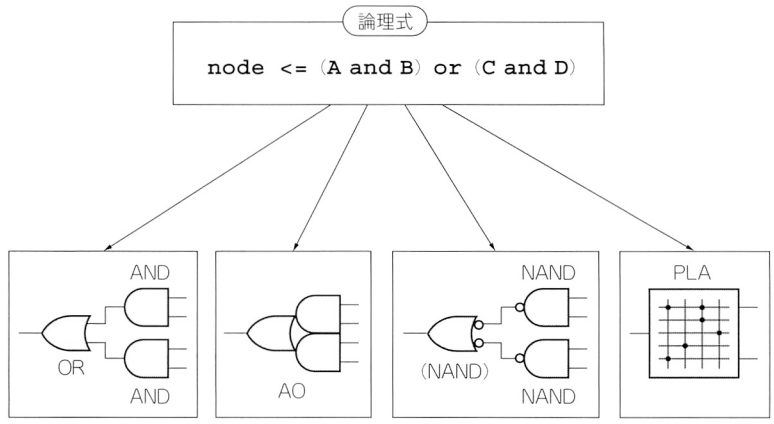
▶ 作業6-3…プレース・アンド・ラウト(配置配線)，ないしフィジカル・デザイン

プリミティブをチップのどの位置に置いてどう配線するかを決めます．フィジカル・デザイン(PD)といいますが，FPGAではフィッティングということもあります．ここまできたら，回路の遅延などの最終的な性能見積もりを立ててチェックします．実配線結果と呼ぶこともあります．

---

*2：ドライブ能力が高いと速いが，面積が大きくなる．

〈図1-4〉
テクノロジ・マッピング



▶ 作業6-4…チップ製造

ASICではマスクを起こして実際にチップを作ります．FPGAなどでは，チップ・データをデバイスに送り込んでやれば使えるようになります．

● 実際には各作業をいったりきたりする

実際には作業1から作業6への一方通行ではなくて，設計中やデバッグ中に行ったり来たりします．また，ビヘイビアを考えているとき，回路速度や回路規模などでボトルネックになりそうな部分だけを先に回路化して性能を確認することはよくあります．

1-4　ディジタル回路設計での注意点あれこれ

ここまで述べた以外に，現在の回路設計でどんな注意点があるかを，いくつか記します．

● すべてのRTLソースがシンセサイザブル(合成可能)なわけではない

次の理由から，RTLソースを書いてもCADで回路を作れない(合成可能でない)ものがあります．

▶ 合成可能でない文法を使って記述した場合

HDLの文法書に載っている記述がすべて合成可能だとは限りません．どのようなCADでも合成可能なようにするため，決まったスタイルのもとで記述を行います．第3章で説明します．

▶ RTLソース中に記述したデータ型と演算をCADがサポートしていない場合

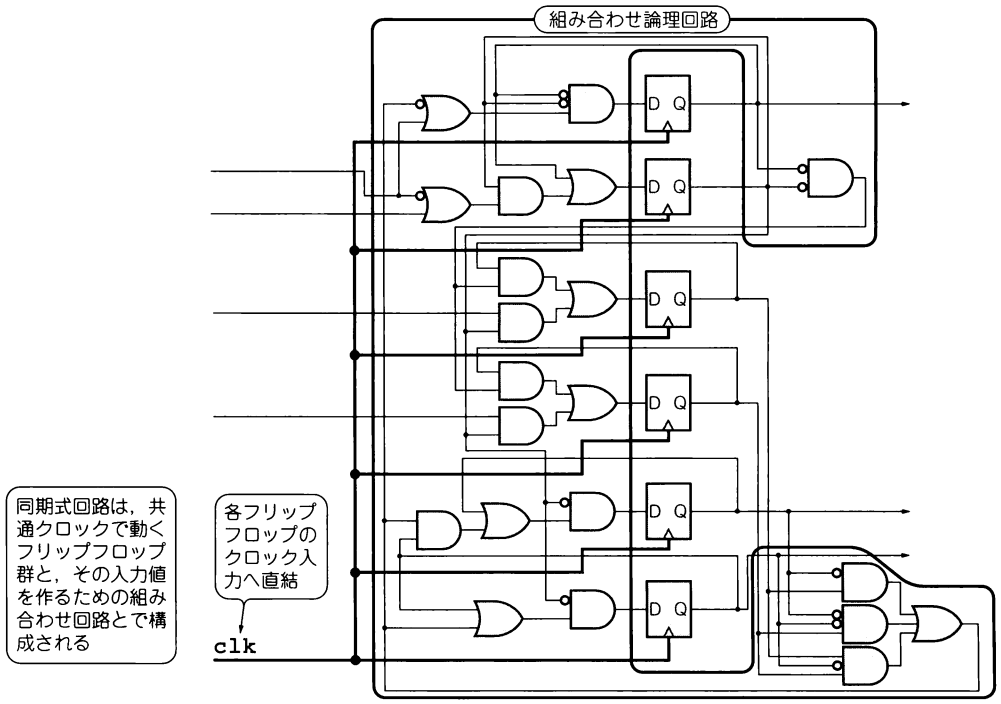
ソフトウェアと違ってハードウェアでは，データ型やクラスを工夫して効果的に処理を組む，というようなセンスはあまりありません．データ型を工夫しても，CADが限られたデータ型しかサポートしていないのでどのみち回路に落ちません．

現在，CADに関わらずシンセサイザブルなデータ型は，

(1) 範囲つきの`integer`型(整数)
(2) `std_logic_vector`または`std_logic`型(いわゆるブール型)
(3) 列挙型(C言語の列挙型と同じ)

くらいしかありません．配列はたかだか2次元，普通は1次元のものしか使いません．構造体のような複

〈図1-5〉
同期式回路の一例



雑なデータ型はまず合成できません．使える演算も，自分で作らない限りは，
(1) 整数の四則演算
(2) 論理演算
くらいしかありません．

また，ハードウェアでは，ポインタ，メモリ管理，再帰呼び出しなどが本質的にありません(CPUのメモリに相当するものが存在しないから)．こうした点からも，ソフトウェア的な処理の組みかたや最適化はできないのです．

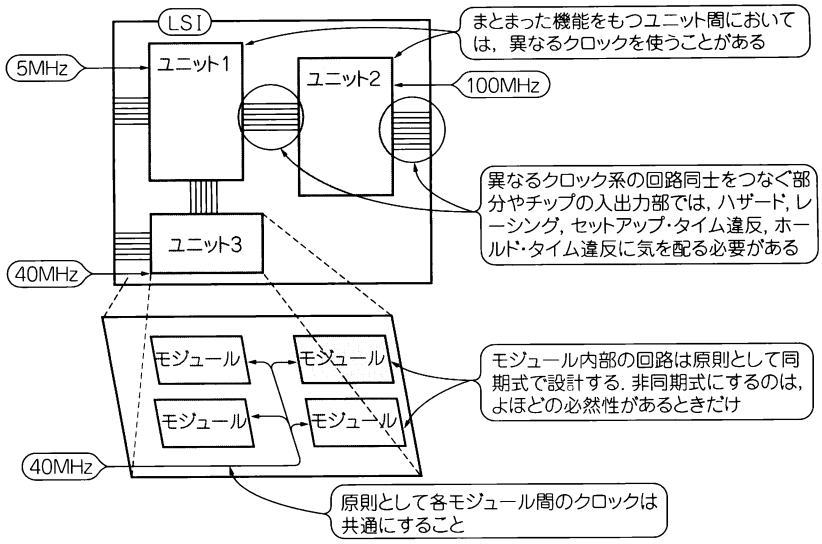
● 原則として全レジスタのクロックを共通にする

個々のモジュールの内部回路は，原則として同期式順序回路(Synchronous Sequential Circuit)として設計します．これは，**図1-5**に示すように，すべてのレジスタ(フリップフロップ)のクロック入力がクロック信号に直結しているものです．すべてのレジスタはクロック立ち上がり動作か，ないしはすべてのレジスタが立ち下がり動作でなければなりません(混在してはいけない)．また，レジスタのクロック入力以外の箇所には，クロック信号を接続してはいけません．

第2章で説明する方法のもとでビヘイビアからRTLを作成していけば，おのずと同期式の回路になるので，安心してください．同期式にする理由はいろいろあるのですが，CADでうまく回路を合成できるようにし，しかも回路を安定して確実に動かすようにするためだと考えてください．

なお，図1-6に示すように，まとまった機能をもつモジュール間では，複数のクロックを用いたりしてクロックが共通でなくなる場合があります．

〈図1-6〉
回路は原則として同期式で設計する



● CADが手伝ってくれないことは何かを知っておく

現在，CADにRTL記述を与えても，以下の作業はしてくれないのが普通です．

▶ ビヘイビアや状態遷移の最適化はしない

ソフトウェアのコンパイラと異なり，ビヘイビアや状態遷移の最適化はしません．つまり，1クロックの動作を速く小さい回路で行えるような最適化はするのですが，複数クロックにまたがる動作を簡単化するようなことはしてくれません．正確に言うと，組み合わせ回路の最適化はそれなりにするが，順序回路の動的性質を利用した最適化やデータ表現形式の最適化はできない，ということです．

▶ ハザードやレーシングの対策はしない

CADによる自動化が難しいテーマです．第5章で述べます．

▶ 小規模/高速な演算回路の自動合成は難しい

これもCADによる自動化が難しいテーマです．第9章で述べます．

▶ 非同期式順序回路の合成は難しい

同期式でないすべての回路を非同期式順序回路といいます．非同期式で設計する場合は，論理合成CADのサポートは期待できません．むしろ，CADがロジックを意図しない形に変更してしまい，設計を妨害するかもしれません．一般には，非同期式順序回路を確実に動かすには，レイアウトまで考慮に入れてゲート・レベルで設計するしかありません．

● いろいろな設計レベルで最適化を図る

回路性能は一つの設計レベルだけでは決まらないので，ビヘイビア(アルゴリズム)～ゲート・レベル～レイアウトにいたるまで，さまざまな視点から性能向上を図る必要があります．たとえばビヘイビアや全体的なモジュール構成に問題があると，それをゲート・レベルの回路設計テクニックで解決することはできません．一方，ビヘイビアをいくらいじっても，乗算器や加算器など個々の回路要素の速度が上がるわけではありません．それにはゲート・レベルの知識がいります．

第2章
単独モジュールのRTL設計法
CRC計算回路と簡易CPUを例にして

この章では，1個の回路モジュール(機能ブロック)をRTLで設計する方法を説明します．次の二つの話が出てきます．

(1) 回路を作るまえに，どのような処理を回路に実行させるか固める
　このとき，アルゴリズムのチェックにC言語などで書いたソフトウェアを使うとよいです．

(2) 次にRTLを作る
　これは，クロック信号がくるたびに1ステップずつ処理を進めていくようにすることです．

2-1 例題としてとりあげるテーマ

● 1個のモジュールはどんな処理をするのか

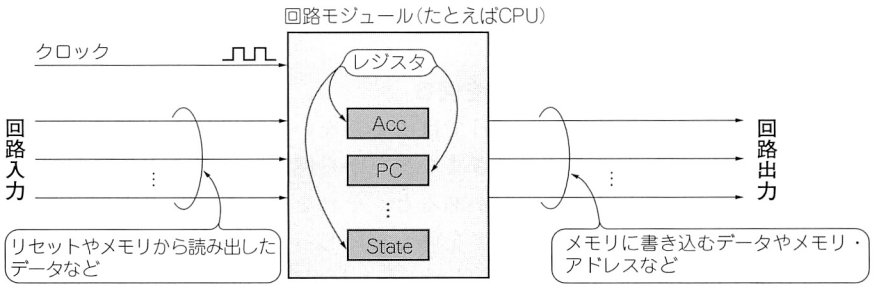
ディジタル回路の1個ぶんのモジュールは，目安として，C言語のプログラムでたかだか300行程度の関数1個にあたる処理をします．処理の内容は，ごくおおざっぱに分類して，次の2種類があります(**図2-1**)．

(1) 外からの入力信号を受けて，適切なタイミングで出力信号を出すこと
　たとえばバスやメモリ，機械などのコントローラ(シーケンサ)などがこのような処理をします．

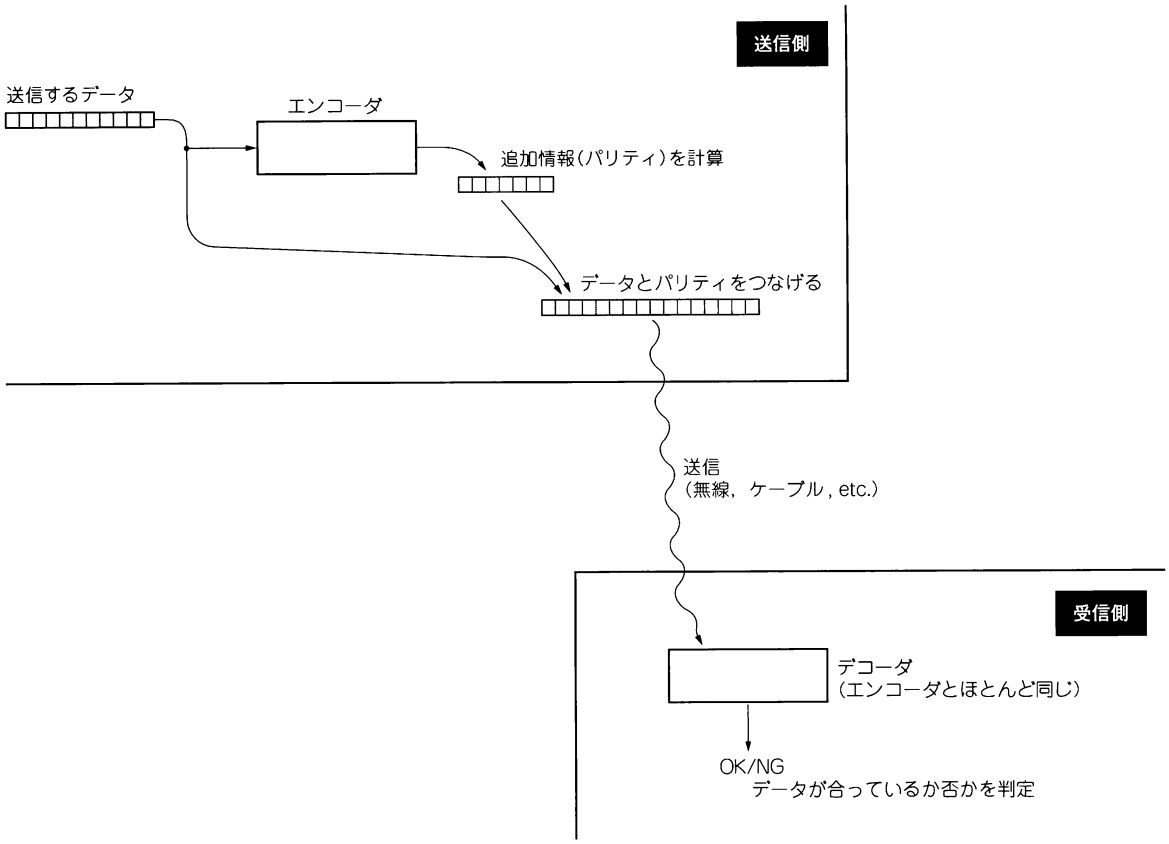
(2) 計算などのデータ加工
　たとえばプロセッサや画像/音声の処理回路などがこのような処理をします．

多くのモジュールでは，(1)と(2)の両方の処理があるのが普通です．(2)では，外からの入力信号に従ってモジュール内にあるレジスタ(ソフトウェアでの変数に相当．後述)の値を変更していきます．つま

〈図2-1〉
モジュールの概念



〈図2-2〉CRCの基本概念



り，多くのモジュールの設計では，入力信号を受けて，内部のレジスタと出力信号の両方を適切に制御する回路を組みます．

本章では，簡単な例題と少し複雑な例題の二つを並行して扱います．どちらも典型的な回路設計手順に従って作っていくことができます．

● 例題①：CRC計算回路

一つ目の例題はCRC計算回路です．CRC(Cyclic Redundancy Check)とは誤り検出符号の一つです．**図2-2**に概略を示します．これは，データをケーブルや無線などで送る際に，送信側で余分なデータを少し追加しておくことで，受信側で受けたデータが正しいどうかを判定できるようにするしかけです．

本章では，CRCの標準の一つであるCRC-CCITTを回路化してみます．ただし，なぜCRCがうまく動くのかというアルゴリズムの原理については，ここでは理解していなくてもかまいません．

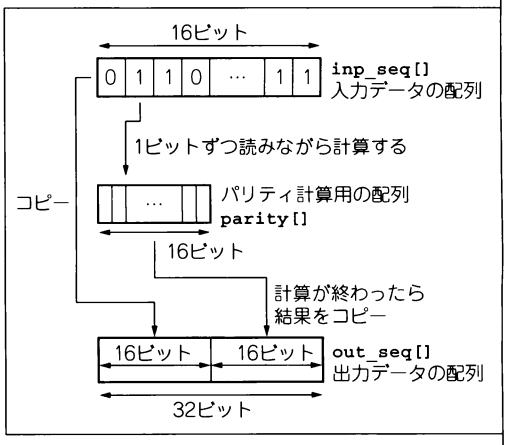
● 例題②：基本的なCPU

二つ目の例題は，昔の8ビットCPU程度の簡単なCPUです．なお，本書の最後では，現在組み込み用途でよく使われている8ビットCPUのPICマイコンを作ってみます．

〈図2-3〉C言語で書いたCRCエンコーダのアルゴリズム(crcenc.c)

```
#include <stdio.h>
#include <string.h>


char xor(char in1, char in2)
{
    if (in1 == in2)
        return '0';
    else
        return '1';
}


void crcenc(char inp_seq[16], char out_seq[32])
{
    char parity[16], prev_parity[16];
    int  datacnt;

    // initialize
    parity[15] = '0';
    parity[14] = '0';
    parity[13] = '0';
    parity[12] = inp_seq[0];
    parity[11] = '0';
    parity[10] = '0';
    parity[9]  = '0';
    parity[8]  = '0';
    parity[7]  = '0';
    parity[6]  = '0';
    parity[5]  = inp_seq[0];
    parity[4]  = '0';
    parity[3]  = '0';
    parity[2]  = '0';
    parity[1]  = '0';
    parity[0]  = inp_seq[0];

    out_seq[0] = inp_seq[0];

    // read input data and calculate CRC
    for (datacnt = 1; datacnt <= 15; datacnt++) {
        // back up current 'parity'
        strncpy(prev_parity, parity, 16);

        // compute next parity
        parity[15] = prev_parity[14];
        parity[14] = prev_parity[13];
        parity[13] = prev_parity[12];
        parity[12] = xor(prev_parity[11], xor(prev_parity[15], inp_seq[datacnt]));
        parity[11] = prev_parity[10];
        parity[10] = prev_parity[9];
        parity[9]  = prev_parity[8];
        parity[8]  = prev_parity[7];
        parity[7]  = prev_parity[6];
        parity[6]  = prev_parity[5];
        parity[5]  = xor(prev_parity[4], xor(prev_parity[15], inp_seq[datacnt]));
        parity[4]  = prev_parity[3];
        parity[3]  = prev_parity[2];
        parity[2]  = prev_parity[1];
        parity[1]  = prev_parity[0];
        parity[0]  = xor(prev_parity[15], inp_seq[datacnt]);

        // write output sequence
        out_seq[datacnt] = inp_seq[datacnt];
    }
}
```

入力データ(16ビット)をもらうための配列
出力データ(32ビット)を返すための配列
これがCRCエンコーダの処理をする関数
パリティ計算用の変数
処理したビット数をかぞえるカウンタ
最初のデータを使ってparityを初期化



一つ目のループ(計算の本体)
inp_seqを読みながらparityを更新していく
入力inp_seqを出力out_seqへコピー

```
        for (datacnt = 0; datacnt <= 15; datacnt++) {
            // write output sequence
            out_seq[datacnt + 16] = parity[15];

            // back up current 'parity'
            strncpy(prev_parity, parity, 16);

            // shift parity register
            parity[15] = prev_parity[14];
            parity[14] = prev_parity[13];
            parity[13] = prev_parity[12];
            parity[12] = prev_parity[11];
            parity[11] = prev_parity[10];
            parity[10] = prev_parity[9];
            parity[9]  = prev_parity[8];
            parity[8]  = prev_parity[7];
            parity[7]  = prev_parity[6];
            parity[6]  = prev_parity[5];
            parity[5]  = prev_parity[4];
            parity[4]  = prev_parity[3];
            parity[3]  = prev_parity[2];
            parity[2]  = prev_parity[1];
            parity[1]  = prev_parity[0];
        }
    }


    void main(void)
    {
        char out_seq[32];
        int  i;

        crcenc("1101101110110011", out_seq);     // sample input vector

        for (i = 0; i < 32; i++)
            fprintf(stdout, "%c", out_seq[i]);
        fprintf(stdout, "¥n");
    }
```

二つ目のループ（出力をつくる）

parity[]をシフトしながら，parity[15]を出力として出していく．通常のソフトではこんなループをまわさなくてもparity[]からout_seq[]へmemcpyなどすればよいが，ここではハード化を考えて，わざと処理のしかたをハードに似せた感じにしてある

2-2　回路の動作を固める

回路設計にあたって最初にやることは，回路動作（処理アルゴリズム）を決めることです．

● 例題①：CRCのビヘイビア

CRC回路の設計では，データの送信側に使うエンコーダと，受信側に使うデコーダの両方を作ります（図2-2）．エンコーダは，送信データに付加するデータ（パリティ）を計算します．デコーダは，パリティ付きのデータを受け取って，そのデータが正しいかどうかを判定します．

CRC-CCITTでは，パリティの長さは16ビットです．送信データの長さはある程度自由に設定できますが，ここでは16ビットとします．

▶ エンコーダのアルゴリズム

エンコーダのアルゴリズムをC言語で書いたものを図2-3に示します．C言語のプログラムとしてはあまりきれいなものではありませんが，一つの関数crcenc()にすべての処理を書いてあります．

この関数は，引き数の配列inp_seq[]に入れた16ビットの送信データからパリティを計算し，送信デ