歴史的に見れば、生活の糧を得るために働く必要のない人達が文化を築くのかもしれません。しかし、そうではない多くの人達の能力を有効に吸い上げるためには、ビジネス分野からみんなが幸せになる形で資金を投入できる仕組みが必要なのだと思います。その時に、囲い込みによる独占利益でなく、先行者の投資に対して十分なリターンができる枠組みを作れなければ投資資金の流入は難しいですね。

「コンピュータビジネスの中で優れたビジネスモデルを築いてきたエンタープライズ分野をうまく取り込むことなく、Linuxなどフリーソフトウェアの将来を明るくするのは難しいのでは」と、そんなことを考えている中で、OSDLの日本ラボの立ち上げということで、その船出を祝して1月の終わりにセミナーの講演をしてきました（記事末のResource [1] を参照）。講演内容は本コーナーで紹介したこともあるSuper Page Kernelの話です。幸いにも多くの人から面白かったと言っていただき、のこのこ出かけた甲斐がありました。

ついでに、PS2Linuxも正式発表になりました。あまりEEのベクトル機構を使っている方を見かけないようですが、Linux Japanの2001年10月号でも初歩的な説明をしているので、ぜひいろいろなアーキテクチャに触れてみてほしいものです。PS2固有の機能を使わなければ、ただのメモリの少ない遅いマシンにすぎませんからね。

また、2002年3月号で付録CD-ROMに収録する予定のファイルが手違いで収録されていませんでした。Linux JapanのWebページに置いていただくことになりましたので、ダウンロードしてください（[2]、今月号の付録CD-ROMにも収録しています）。

今月号では、まだ積み残しになっているアセンブラの話は置いておいてハードウェアの基礎を復習してみます。デジタル回路の教科書でも一通り復習すれば済むような内容ではありますが、私なりの視点でまとめてみたいと思います。

■デジタルデータの伝送

プロセッサに限らず、デジタルデータの伝送は多くのデジタル回路の動作の基本になっています。伝送するからには送り側と受け側があるわけで、どこからどこへ送るのかが問題になります。発信側は電子回路の中で情報を記憶する回路、すなわち、メモリやラッチ、フリップフロップなどの回路になります。そして受ける側も当然これらの記憶回路を最終地点としてデータを伝送することになります。特に複雑な制御を高速に実行する必要のある回路の中ではラッチやフリップフロップという記憶回路が重要になります。

これらの記憶回路の中でも比較的シンプルなDラッチ回路を図1に示します。

この回路はクロック信号であるCが1の間は入力したデータを出力Qに出すのですが、クロック信号が0になると出力自身をループさせることになります。このループが正帰還という仕組みで情報を記憶することになります。

クロック周波数0Hzから動作する記憶素子としてはこのDラッチ回路はもっともシンプルなものの1つですが、大きな問題があります。というのは、クロック信号が1である間はずっと入力は出力に筒抜けになるので、データを記憶するという役割を果たさないのです。これを解決するには2つの位相の異なるクロック信号を用います（図2）。

このクロックφ1、φ2を用いる記憶素子を相互に並べることでデータの筒抜け（これをレーシングと言います）を防ぎます。でも、クロック信号の分配はすべてのラッチに同時に信号が到着するようにする必要があるので、大変面倒でそれを何本も送るのは実は配線を難しくすることになります。そこで、この2つのラッチをくっつけてクロック信号を1本だけにするやり方もあり、こちらのほうが考えやすくなります（実は、私は仕事でクロック信号を8本も持ったマシンを設計したことがありますが、自分では主として2本しか使わなかったし、最後までなんでそんなに必要なのか理解できませんでした。非同期なメモリとのタイミングとかイロイロ理由はあるのでしょうけれど、位相を増やすより高速クロックを分配することを考えたほうが、多くの面で意味がありそうなのですけどね）。

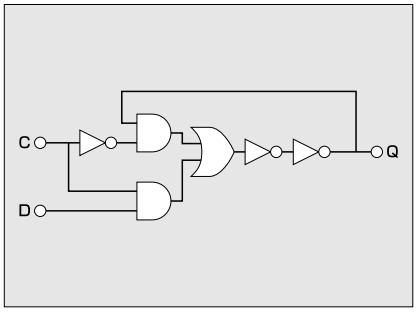
クロック1本だけでレーシングせずにデータを転送できるラッチのことをエッジトリガ・フリップフロップとか、マスター・スレーブラッチとか呼びますが、この記事は教科書じゃないので細かな説明は省略させていただきます。

この連載のプロセッサではクロック信号を1本だけ使うように回路を構成しますので、レジスタやラッチと言ったら、このエッジトリガ・フリップフロップを指すと思ってください。

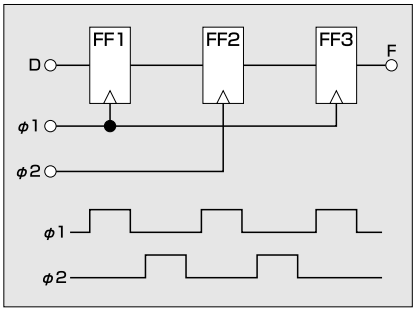
簡単にマスター・スレーブラッチを構成するにはDラッチを2つ組み合わせます。そして、前段（スレーブ）ラッチにクロックの逆相のクロックを入力します（図3）。

すると、クロックが0のときにスレーブラッチがデータを通し、クロックが1になった瞬間にスレーブラッチはデータをホールドしてマスターラッチがその値を通すことになります。このようにマスター・スレーブラッチを用いると、クロックの立ち上がりの瞬間を捕らえてデータを記憶できるようになります。電

【図1】



【図2】



気回路としては動作時間などに絡んでいろいろと難しい問題を抱えているのですが、ここでは割愛しておきます。

デジタル回路ではラッチからラッチへデータを伝送し、その途中に組み合わせ回路による演算を行います。そこで、この経路のデータ伝送が正しく行える条件を遅延時間を元に検討します。これを機械的に行うのが遅延シミュレーションで、大規模回路の開発ではサイクルベースシミュレーションと遅延シミュレーションを組み合わせて効率の良い開発が行えることになります。ラッチを使ってデジタル信号を伝送する場合の条件を図4を使って説明します。

まず、システムのクロック信号は送り側も受け側にも同時に到着するように回路設計をします。しかし、実際には完全に同時にクロックを入力することは物理的に不可能なので、送り側ラッチと受け側ラッチのクロック信号の到着には時間差が生じます。この時間差をクロック・スキューと呼びます。

クロック・スキューは1組のラッチの間だけを考える分には、これを遅延に組み込んでしまえばいいのであまり問題にはなりません。しかし、システム中の全体のデータ転送を考えたときには、あるラッチ間の転送は受け側が遅れて、別のラッチ間の転送では受け側の方が早くなるなどといったバラバラなことになってしまいます。すると、システムとしてどれくらいのクロック・スキューがあるのかを設計指針として示して、その分のスキューが存在することを前提に伝送を考える必要が生じます。といっても扱いは簡単で、遅延時間にスキュー分を加えて計算すれば良いだけですけれど。余談ですが、私が商用プロセッサの設計をしていたころは、このスキューだけで結構大きな値になっていたので、今のプロセッサのようなGHzオーダーのクロックは本当に夢のように感じます。

次に、受け手のラッチへのデータ到着は、少なくともクロックが立ち上がる前に初段のスレーブラッチをデータが通過する時間の余裕が必要になります。というのは、クロックが立ち上がったところでスレーブのクロックは閉じるので、この時にマスターラッチの入力にデータが到着している必要があるのです(実際は、スレーブの正帰還が正しく情報をホールドできるだけあればいいのですが、ここでは定性的な解析ということで簡単化しておきます)。この時間のことをセットアップタイムと呼びます。

このように、データの伝送には本来のゲート遅延の他にもこのような時間を考えなくてはならないのです。ただし、セットアップを含めてゲート遅延として解析した方が妥当な場合も多く、この辺りの回路的な扱いはHDLからネットリストに落としたところで考える場合にセルライブラリの作り方と絡んできて、結構面倒です。

ですが、ここでは、「こういった考え方で遅延を計算して回路の性能を測るんだ」ということだけ理解していてくれれば結構です。後は遅延シミュレータがやってくれます。遅延シミュレータの出力の妥当性を大体把握できるようになっていさえいれば、アマチュア的には問題ないと思います。もっと詳しい話を知りたい方は少し古い本になりますが、丸善から出ている「VLSIシステム設計―回路と実装の基礎」([3])が参考になると思います。パッケージングなど新世代の話題は網羅されてはいないのですが、LSI内部の話はデジタル回路の設計者として必要な程度のことは書いてあります。

■ハードウェア記述言語(HDL)

少し大きなデジタル回路を作成するときにはHDLを用いることが多くなっています。この連載においてもハードウェアの設計はHDLを用いて行うことにします。良く使われるHDLとして、VerilogやVHDLがあります。しかし、これらのHDLを処理する処理系の中でプロセッサを作成するのに耐えられ、かつ、無料で利用できるものは多くはありません。もちろん、苦労して機能の限定されたものを使いこなすというのも悪くはないのですが、アマチュア的にはつまらないところで苦労をしたくないので、ここでは無償で処理系が提供されるSFLを記述言語として用います。言語はテキストで入力するので処理系として必要な機能は

- サイクルベース論理シミュレーション
- 論理合成
- 遅延シミュレーション

ということになります。合成した結果をCPLDボードにダウンロードするにはWindows上で動作するベンダの無償ツールを用いることになるので、完全にLinuxだけで完結するわけではありません。しかし、主要な開発ステージはLinuxのみでできる優れものです。

HDLの話をすると、メーカーで使っているVerilogやVHDLを使わなければ役に立たないという人が必ず現れるのですが、これらの言語と記述能力自体に大きな差があるわけではありません。HDLでデジタル回路を設計するという最終目標のためには使える言語を使っておけばいいでしょう。プログラム言語においても、Cを勉強した人がPascalでプログラムを作れないとか、その逆はあまりないと思います(まあ、いきなりCからLISPに移れと言われたら面食らうかもしれませんが、SFLとVerilogは大きな差はありません)。

SFLの処理系PARTHENONはNTTが開発したもので、「PARTHENONのダウンロード」というWebページから無償でダウンロードできます([4])。PARTHENON処理系の一般的な解説は、PARTHENONのWebページをご覧ください([5])。

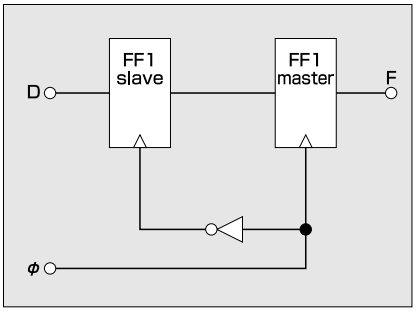
このページには言語解説なども載っているので、こちらを参考にプロセッサを作ってください……なんて書いてしまうと連載の意味がないので、デジタル回路をSFLで記述するための基本を解説してみます。まず、SFLでは回路をモジュール(module)と呼ぶ単位で構成します。モジュールは階層的に組み合わせて用いることもできます。他のモジュールを呼び出すときに、接続を固定的に記述する方法と制御端子を用いてメッセージ渡しのように記述する方法があり、PARTHENONの解説などではメッセージ渡しを推奨しています。私自身はどちらでも好きなように使えばいいと思っており、使いやすいように、記述が分かりやすいように言語の機能を使うように心がけています。

■SFL、PARTHENONの基礎

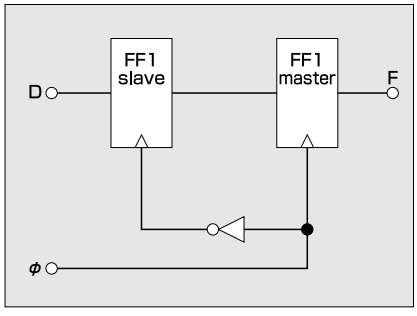
組み合わせ回路と呼ぶ回路は、記憶素子がない回路です。組み合わせ回路は入力端子に入力されるデータだけで出力が決まります。SFLでは組み合わせ回路を表すためにブール代数を用います。個々の演算記号については説明しませんから、詳しくは前述したWebページの説明を参照ください。

例えば、2つの入力の論理積を出力するモ

【図3】



【図4】



ジュールをリスト1に示します。

モジュールの端子には入力端子、出力端子、双方向端子、制御入力端子、制御出力端子などの種類があります。ここでは、入力端子と出力端子を用いています。この書き方は眺めるとなんとなく分かるような気がしてきませんか？

このモジュールを`ex1.sfl`という名前のテキストファイルとして作成します。言語仕様とは関係ないのですが、合成系の処理系の都合で、合成するモジュール名とテキストファイルのベース名は一致させておいた方が便利です。

さて、せっかく論理回路のソースコードができたので、これの動作確認をしてみましょう。動作確認にはPARTHENONの論理シミュレータsecondsを用います。PARTHENONをインストールして、環境変数やパスの設定が終わっていれば、

```
$ seconds
```

とコマンド名`seconds`を入力するとスタートメッセージが出てくるはずです。secondsのコマンドプロンプトは

```
SECONDS>
```

となっているはずです。

うまくいかなかった人もいるかもしれませんが、ライセンスファイルが古かったり、パスが通っていなかったり、環境変数が設定されていなかったりと、うまくいかない理由はいくつもあるので、じっくりと対策してください。ちなみに、`seconds`から抜けるコマンドは「`bye`」になります。「`quit`」では抜けられませんからご注意を。私はこれを何度も間違えるんです。

それでは、シミュレータに論理ファイルを読み込ませましょう。

```
SECONDS> sflread ex1.sfl
```

ファイルの中には複数のモジュールを記述できるので、最上位となるモジュール名を指定してシミュレーションイメージを作成します。これは、シミュレータ内部にシミュレーションを実行できるようなデータ構造を構築することを言います。

```
SECONDS> autoinstall ex1
```

ここではモジュールは1つしかないので、その名前を指定します。シミュレーションイメージができると、その中の部品（コンポーネント）がシミュレータから見えるようになります。コンポーネントを表示するには`lc`コマンド（list components）を用います。`ex1`モジュールにはモジュール自身と入出力の端子ならびに代入文があるので、これらが表示されます（**実行例1**）。

次に、シミュレーションでモニタしたいデータ端子を指定します。secondsにはレポートという機能があって、あらかじめレポートを登録しておくと、クロックを進めたときや、明示的にレポートを要求したときに登録した形式で端子などの情報を出力します（**実行例2**）。

ここでは、`ex1`というレポートを登録します。このレポートには3つの入出力端子を記述し、その出力をバイナリで表示します。`%B`と大文字でバイナリを指定すると、端子の値が不定の場合に不定を意味する「`u`」を出力することになります。論理値として`0`と`1`の他に端子がドライブされていないことを表す「`z`」と、この不定の「`u`」の4値で論理信号を表します。Cでプログラムを書いたことがある方なら、この書式はほぼ推測がつくものではないでしょうか？

そう、printfの書式に大変良く似ています。

例えば、このままの状態でレポートを表示させてみます。レポートの強制表示は`report do`コマンドで実行します（**実行例3**）。

参照された端子がドライブされていないという警告が出ていますが、その結果、出力は不定になっています。シミュレーションで端子に値を設定するために`set`コマンドを用います。このコマンドは、当該シミュレーションサイクルに対してだけ値をセットするので、シミュレーションサイクルが進むと再びドライブは切れます。しかし、今回は組み合わせ回路でシミュレーションサイクルを進めるわけではないので、その点は気にしなくて結構です。順序回路でシミュレーションサイクルを進めても、同じ値を与え続けたい場合には`hold`コマンドを併用します（**実行例4**）。

入力端子に値が設定されていると、`seconds`は文句を言わずにレポートを出力します。今、入力`a`と入力`b`に双方とも`1`を設定しましたが、違う値を入れて実験してみてください。

ついでに、このファイルを論理合成にかけてみましょう。いったん、`seconds`を`bye`コマンドで抜けて、次のように合成のコマンドを実行します。

```
$ auto ex1 ps DEMO demo
```

このコマンド（`auto`）は、論理合成のためのスクリプトになっています。最初の引数はモジュールのベース名です。スクリプト中で「`.sfl`」を付加したファイル名を利用するのと、この名前を合成モジュール名としても利用しているため、この処理系ではファイル名と合成モジュール名を同一にしておく必要があります。論理合成はHDLからネットリストを生成するものですが、ここでは回路図を合わせて生成するために2番目の合成のオプションに`ps`を指定しています。回路図が必要ない方は`ps`の代わりに`nld4`を指定してください。すると、EDIFのネットリストまで出力して終わります。後の2つの引数は、合成に用いるセルライブラリと

【リスト1】

```
/* example 1 ex1.sfl */
module ex1 {
 input a<1>, b<1>;
 output f<1>;
 f = a & b;
}
```

＊筆者注　信号は<n>でデータの幅を指定しますが、指定しなければ1ビットになります。実は、**リスト1**の例ではどうせ1ビットの信号しか扱っていないので、データ幅を指定しなくても構いません。処理系の制限で、端子への値の代入は例え端子が複数ビットからなっていても一括して行う必要があります。また、論理だけのループは禁じられていますが、処理系は信号名だけでループを判定するのでVHDLなどで良くやるようなビットをずらして代入するような文は書けませんが、こういったものは書けなければ他の書き方をすれば良いだけです。モジュールには無条件に評価対象となる文を1つ定義できます。例題ではこの文にブール代数を書いていますが、もっと複雑なことをさせたい場合には複数の文を並列評価するpar構文や、選択的に評価するanyやalt構文を用いることになります。このようにSFLのモジュールの構文はモジュール中で記述場所が決まっているものが若干あるので注意が必要です。

【実行例1】

```
SECONDS> lc
/            (module          )
  a                  (term input      )
  b                  (term input      )
  f                  (term output     )
  1                  (simple statement) : write terminal
```

【実行例2】

```
SECONDS> rpt_add ex1 "a=[%B] b=[%B] f=[%B]\n" a b f
```

【実行例3】

```
SECONDS> report do
Error :(forward) on 0 in /
  (W)Referred terminal /b is not driven.
  (W)Referred terminal /a is not driven.
End of Errors
a=[z] b=[z] f=[u]
```

【実行例4】

```
SECONDS> set a B1
SECONDS> set b B1
SECONDS> report do
a=[1] b=[1] f=[1]
```

呼ばれるLSIのライブラリを指定します。ライブラリはファウンダリの会社名と、その中のターゲットプロセス名という順番で指定します。DEMO社のdemoライブラリというのは処理系に標準添付される仮想のライブラリで、0.8umCMOS相当のライブラリになっています。

論理合成は多くの中間ファイルを生成しています。合成が終わったディレクトリの中身は次のようになっています（リスト2）。

リスト2の中で、合成結果として必要なファイルはネットリストであるex1.edif200です。また、ex1.psは、ネットリストを回路図に変換したポストスクリプトファイルなので、どのような合成がなされたのか回路図で確認したい場合に便利です。この例題では図5のような結果になります。CMOSのセルライブラリからNANDとINVを用いて論理積を作成している様子が分かりますか？

比較的単純な例題とそのシミュレーションスクリプトをいくつか示しておきます。これらのHDL記述がどのように動作するのか、ぜひ確認してSFLの使い方に対する、おおよその感覚を養っていただきたいと思います。

リスト3はレジスタへの転送を行っている例題です。レジスタの値を出力端子に出す文とレジスタに値を書き込む記述を同一のクロックサイクルで実行するために、並列実行文であるparを用いています。単なる値の転送は等号を用いるのですが、レジスタへの書き込みのように結果が次のサイクルに反映される転送は「:=」を用います。この回路の動作確認にはリスト4のようなスクリプトを使いましょう。このスクリプトを例えば「ex2.sim」のようなテキストファイルとして作っておけばsecondsから

```
SECONDS> listen ex2.sim
```

というようにスクリプトの実行が可能です（listenは省略可能です）。

リスト4のスクリプトでは、シミュレーションサイクルをforwardコマンドで進めながら、レジスタ（フリップフロップ）の動作を確認しています。forwardコマンドではシミュレーションサイクルを進めるたびにレポート報告が自動実行されます。

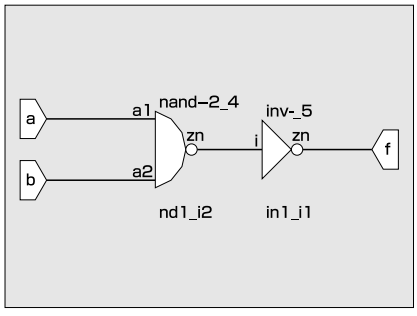
リスト5はSFL特有の制御端子を用いた例です。制御端子には、制御入力端子であるinstrinと、制御出力端子であるinstroutならびに内部制御端子であるinstrselfがあります。制御端子がアサート（信号の有効化）されたときの動作をあらかじめ記述しておけることと、ドライブされていない制御端子は値0として扱うことが入力端子と異なる点です。この例題ではマルチプレクサを構成しています。

シミュレーションスクリプトをリスト6に示します。ここで2つの制御入力を同じに1にすると、回路的には転送文が同時に起動され、おかしなことになることに気が付かれると思います。secondsはこのような状態の時にエラーを出力します。ですが、最適化した合成結果は単純なマルチプレクサを出力するので、実回路ではエラーは出力されることはありませんし、モノが壊れることもないことに注意してください。

さて、今度はもっと難しくなります。リスト7の例題は、3つの状態s1、s2、s3を持つ有限状態機械（Finite State Machine。略してFSM）を実現するものです。SFLでは状態間の遷移はgoto文で行います。状態の動作記述はstate文で行い、状態を持つ順序回路はstage文の中に記述されるということで少し階層が多くなっていますが、慌てずじっくり眺めてみてください。

外部からFSMを起動するための制御入力信号sがアサートされると、「stage seq」のタスクが起動されます。タスクの名前は複数付けられるので、起動条件などでステージの中で動作を切り替えたいときには便利です。しかし、この例題ではタスクをひとつしか使っていませ

【図5】



【リスト2】

```
ex1.0off  ex1.2off   ex1.4th/   ex1.ons     ex1.sfl
ex1.1off  ex1.3off   ex1.cpb    ex1.op1     ex1.tsc
ex1.1st/  ex1.3rd/   ex1.edif200 ex1.op2
ex1.2nd/  ex1.4off   ex1.hsl    ex1.ps
```

【リスト3】

```
/* example 2 ex2.sfl */
module ex2 {
 input a<1>;
 output f<1>;
 reg r<1>;
 par {
  r := a;
  f = r;
  }
}
```

【リスト5】

```
/* example 3 ex3.sfl */
module ex3 {
 input a<1>,b<1>;
 output f<1>;
 instrin sa,sb;
 instruct sa f = a;
 instruct sb f = b;
}
```

【リスト4】

```
sflread ex2.sfl
lmc
autoinstall ex2
lc
rpt_add ex2 "a=[%B] r=[%B] f=[%B]\n" a r f
report do
set a B0
report do
set a B1
report do
forward +1
set a B0
forward +1
```

【リスト6】

```
sflread ex3.sfl
lmc
autoinstall ex3
lc
rpt_add ex3 "a=[%B] b=[%B] f=[%B] sa=[%B] sb=[%B]\n" a b f sa sb
report do
set a B0
set b B1
report do
set sa B1
report do
set sb B1
report do
```

んので、タスク名runは宣言のところと制御入力の動作宣言のところのステージ起動部分にしか出てきていません。FSMは最初の状態はs1であることが宣言されていますので、リセット後タスクが起動されるとs1の状態に入ります。

その後、クロックが進むたびにs2、s3と進み、s3のところで初期状態に分岐するとともにステージを終了させています。SFLでは、現状態はステージが終了しても保持されるので、次の起動のときに再びs1から起動させたい場合には、このように終了時に初期状態への分岐を書いておく必要があります。一見不便なようですが、コルーチン的に複数のステージ間で互いに呼び合いながら、少しずつ動作を進めるような場合には、こちらの方が便利が良くなります。

シミュレーションスクリプトは簡単です(リスト8)。1ステップずつシーケンスが進んでいく様子がお分かりですか？

だいぶお疲れかも知れませんが、もう少しがんばりましょう（笑)。

リスト9は何かというと、パイプライン構成

【リスト7】

```
/* example 4 ex4.sfl */
module ex4 {
 input a1,a2,a3;
 output f<1>;
 instrin s;
 stage_name seq {task run(); }
 instruct s generate seq.run();
 stage seq {
   state_name s1,s2,s3;
   first_state s1;
   state s1 { f=a1; goto s2;}
   state s2 { f=a2; goto s3;}
   state s3 par { f=a3; goto s1; finish; }
 }
}
```

【リスト8】

```
sflread ex4.sfl
lmc
autoinstall ex4
lc
rpt_add ex4 "a=[%B%B%B] f=[%B]\
s=[%B] seq=[%B]\n" a1 a2 a3 f s seq
report do
set a1 B1
set a2 B0
set a3 B1
hold a1
hold a2
hold a3
set s B1
report do
forward +1
forward +1
forward +1
forward +1
```

【リスト9】

```
/* example 5 ex5.sfl */
module ex5 {
 input a;
 output p1,p2,p3;
 reg r1,r2,r3;
 instrin s;
 stage_name q1 {task run(r1); }
 stage_name q2 {task run(r2); }
 stage_name q3 {task run(r3); }
 instruct s generate q1.run(a);
 stage q1 {
 par { p1 = r1; relay q2.run(r1); }
 }
 stage q2 {
 par { p2 = r2; relay q3.run(r2); }
 }
 stage q3 {
 par { p3 = r3; finish; }
 }
}
```

【リスト10】

```
sflread ex5.sfl
lmc
autoinstall ex5
lc
rpt_add ex5 "a=[%B] s=[%B]
p=[%B%B%B]\n"\
 a s p1 p2 p3
set a B1
set s B1
hold a
report do
forward +1
set a B0
set s B1
forward +1
set s B0
forward +1
set s B1
forward +1
```

【リスト11】

```
declare cpa {
   input in1, in2, ci;
   output out, co;
   instrin do;
   instr_arg do(in1,in2,ci);
}

module ex6 {
   input in1<4>, in2<4>, ci;
   output out<4>, co;
   cpa a1,a2,a3,a4;

 par {
     out = a4.do(in1<3>,in2<3>,a3.co).out ||
           a3.do(in1<2>,in2<2>,a2.co).out ||
           a2.do(in1<1>,in2<1>,a1.co).out ||
           a1.do(in1<0>,in2<0>,ci).out;
     co = a4.co;
     }
}

module cpa {
   input in1, in2, ci;
   output out, co;
   instrin do;

   instruct do par {
     out = in1 @ in2 @ ci;
     co  = (in1&in2)|(in1&ci)|(in2&ci);
     }
}
```

の回路にデータを流しているものです。パイプラインの各ステージがstage q1からq3となっていて、パイプラインラッチにはr1、r2、r3を用いています。次のステージを起動するためにはrelay文を用いますが、これは次ステージの起動と本ステージの終了を組み合わせた文になっています。各ステージでは、自分の受け取った値を出力端子に出力して、その値を次のステージに転送します。タスクには引数を定義できて、パイプラインラッチが比較的スマートに記述できている点に注目ください。

スクリプト（リスト10）はもう示すまでもないかもしれませんが……。あんまり難しいことはやっていません。

さて、単一モジュールの記述についていろいろ示してきましたが、最後に複数モジュールを階層的に用いる方法を加算器を例に示してみたいと思います（リスト11）。

リスト11の例題は1ビット全加算器であるcpaを4つ組み合わせて4ビット全加算器を構成しています。リプルキャリーと呼ぶ遅い演算方式を取っているので実用性はありませんが、例題として記述の簡単さを重視しています。このように階層構造を用いる場合には、下位階層のプロトタイプをdeclare文の中に記述する必要があります。ここで示すように、制御端子には引数を定義できるので、下位階層を利用する場合に関数呼び出しのように引数を与えて使うことができます。

今回の例題では論理シミュレーションは各自やっていただくこととして、論理合成と遅延シミュレーションについて調べてみましょう。autoコマンドで論理合成が可能だと書きましたが、合成結果のサマリーは標準出力に流れていって、どこを見て良いのか分かりにくいですね。そこで、リスト12のようにサマリを取り出しましょう。

まず、autoコマンドの出力をファイルにもリダイレクトしておきます。次に、できたファイルをsedで加工してサマリ記述のあるところを抜き出します。合成の統計データとしてリスト13のような部分が見つかると思います。これを見ると合成の概要が分かります。消費電力の単位はμW/MHzで1MHz当たりの電力となっています。チップ面積の単位は1000 μm^2となっています。ゲート数は2入力NANDを基本としていたと思います。これらの値はすべてライブラリに依存するのでライブラリが異なるときには解釈も変わることにご注意ください。

遅延シミュレーション結果はリスト14のように表示されていると思います。この他にも遅延の分布のヒストグラムも出ているのですが、クリティカルパスをまずは求めてみましょう。

DEMOライブラリでは遅延の単位は1nSになります。遅延ルートの途中のゲートの名前が表示されるのですが、階層構造を使っている場合には、階層名もゲートの名前の中に出現するので比較的追いやすいと思います。合成結果のEDIFネットリストと付き合わせれば確実ですが、ここではもう少し簡単に回路図上で追いかけてみましょう。論理合成を行った結果の回路図は

```
ex6.ps
```

という名前で合成したディレクトリの中にあると思いますので、これをghostviewなどで表示してみてください。個々のゲートに名前がついていますので、クリチカルパスのゲート名と比較して、どのルートがクリチカルなのかを把握できます。

駆け足でハードウェア記述言語の説明をしてきましたが、この記事だけで把握できない内容はPARTHENONのWebページのチュートリアルなどで補っていただきたいと思います。この程度の知識があれば、プロセッサの設計くらい普通にできるというのも技術の進歩ですね。GHzクラスのプロセッサを設計するにはレイアウトや配線に対するもっと深い考察が必要になりますが、基本はそれほど大きく変わってはいないはずです。

【リスト12】

```
auto ex6 ps DEMO demo | tee ex6.log
sed '1,/### summary/d' ex6.log|more
```

【リスト13】

```
position   = /
        type       = NLD
        class_name = ex6
        power      = 187.9
        area       = 21.38
        gates      = 84
```

【リスト14】

```
maximum rise delay path 1
        src  0  max 2.00000e+01
        nml  1  max 2.09160e+01  (/i       ) to (/zn      )  /inv-_41
(in1_i1)
        nml  2  max 2.16420e+01  (/a1      ) to (/zn      )  /nand--2_34
(nd1_i2)
        nml  3  max 2.37930e+01  (/c2      ) to (/zn      )  /oai--222_37
(oa_i222)
        nml  4  max 2.57846e+01  (/a2      ) to (/zn      )  /oai--21_21
(oa_i21)
        nml  5  max 2.64586e+01  (/a2      ) to (/zn      )  /nor--2_20
(nr1_i2)
        nml  6  max 2.84502e+01  (/a2      ) to (/zn      )  /oai--21_17
(oa_i21)
        nml  7  max 2.98534e+01  (/a2      ) to (/z       )  /a4_eor-1
(xo1_i2)
        snk  8  max 3.09866e+01  (/a2      ) to (/z       )  /a4_eor-2
(xo1_i2)
```

Resource

[1] JLA／OSDLセミナーのご案内
http://www.osdl.jp/events/2002_jan_24.html

[2] Linux Japan
http://www.linuxjapan.com

[3] VLSIシステム設計ー回路と実装の基礎
H．B．Bakoglu著／中澤喜三郎、中村宏監訳／8600円／丸善株式会社出版事業部／ISBN-4-621-04049-9／1995年3月発売

[4] PARTHENONのダウンロード
http://www.kecl.ntt.co.jp/parthenon/html/download_j.htm

[5] PARTHENONのWebページ
http://www.kecl.ntt.co.jp/parthenon/index_j.htm