POCO の論理合成

論理合成とは

今までは、Verilog はハードウェア構造を記述し、シミュレーションするだけだった。しかし、記述されたハードウェアは何らかのデバイス上で実現しなければ意味がない。このためには Verilog 記述をゲートレベルの接続図 (ネットリスト) に変換し、様々の最適化を行う作業が必要である。これを論理合成、圧縮と呼ぶ。今日はこの合成ツールの使い方を習得し、これを使って今まで設計してきた POCO を合成、圧縮することが目的である。

論理合成圧縮ツール

論理合成圧縮ツールは、プログラミング言語におけるコンパイラに相当するが、対象がハードウェアであるために、コンパイラよりもはるかに処理が複雑で、時間がかかり、また、ツール自体も高価である。現在、学生レベルで無料あるいは安価に使える論理合成圧縮ツールには以下のものがある。

- FPGA ベンダの合成ツール、Xilinx 社の ise、Altera 社の Quartus II などは、web パックと呼ばれる簡易版は無料である。web パックは一部の機能が制限されているが、POCO レベルの回路ならば十分合成可能であり、実際に 3 年生の実験で用いている。
- PARTHENON、この授業で、以前、用いていた合成ツール。元々NTT が開発したが現在は NPO 法人が管理し、無料である。しかし、独自の記述言語である SFL のみを受け付け、Verilog は受け付けない。

FPGA ベンダの合成ツールは、実際の基板上で動かせる点で面白いが、対象依存性が強すぎ、ある意味で相当癖がある。そこで、これは 3 年の実験で用いることとして、ここでは、本格的な ASIC(Application Specific IC) 合成用ツールである Synopsys 社の Design Compiler を用いることにする。

このツールは、実際に ASIC を設計する際に使う CAD で、本来、非常に高価であるが、VDEC(VLSI Design and Education Center: http://www.vdec.u-tokyo.ac.jp/welcome.html) で、研究教育目的で安価で利用させてもらっている。ライセンス管理が厳密なので、天野研究室のマシンに遠隔ログインして用いる。ログインの方法は web を参照のこと。もちろんであるが、このツールを研究教育以外の目的で使ってはならない。

合成用のライブラリ

今回はセルベースで ASIC(Application Specific IC) を作る場合の方法を学ぶ。ASIC とは、携帯電話、情報家電などに用いる特定目的用の IC であり、現在の日本の半導体企業の主力製品である。ASIC ではセルベースド設計と呼び、高さの揃ったセルライブラリを利用する。セルライブラリの中には NAND, AND など基本的なゲートや複合ゲート、フリップフロップなどが含まれている。論理合成を行うためには、設計対象となるチップのライブラリを指定する必要がある。今回は、オクラホマ大学で開発した TSMC 0.18um　CMOS 用のセルライブラリを使用する。0.18um は、やや時代遅れのプロセスだが、まだ現役で利用されている。これは教育用に公開されているもので、実際にチップを設計する時に使うライブラリに比べるとゲートの種類等が少ないが、現実的な性能、面積で合成してくれる。実際にチップを作る場合には、同様のセルライブラリをチップベンダーから供給してもらう。[1]

design_compiler の使い方

Design Compiler は、バッチ処理で用いる場合が多い。まず、合成用のコマンドを記述したスクリプトファイルを作る。ここでは web から poco.tcl を取って来て中身を見てみよう。

[1] ちなみに、ふんが研に入れば 65nm や 40nm のプロセスが利用可能で、POCO のレイアウト例も見ることができる

```
set search_path [concat "/home/cad/lib/osu_stdcells/lib/tsmc018/lib/" $search_path]
set LIB_MAX_FILE {osu018_stdcells.db  }
set link_library $LIB_MAX_FILE
set target_library $LIB_MAX_FILE

read_verilog alu.v
read_verilog rfile.v
read_verilog poco1.v
current_design "poco"
create_clock -period 8.0 clk
set_input_delay 6.0 -clock clk [find port "ddatain*"]
set_input_delay 2.5 -clock clk [find port "idatain*"]
set_output_delay 7.5 -clock clk [find port "iaddr*"]
set_output_delay 3.5 -clock clk [find port "ddataout*"]
set_output_delay 3.5 -clock clk [find port "daddr*"]
set_output_delay 3.5 -clock clk [find port "ddataout*"]
set_output_delay 3.5 -clock clk [find port "we"]

set_max_fanout 12 [current_design]
set_max_area 0

compile -map_effort medium -area_effort medium

report_timing -max_paths 10

report_area
report_power

write -hier -format verilog -output poco.vnet

quit
```

この記述は、スクリプト記述用言語 tcl で書かれている。tcl は CAD のスクリプト記述用に広く使われ、高い機能を持っている。ここでは触れないことにするが、興味のある方は調べて使いこなせると便利である。

最初の set 文は、ライブラリの読み込みと環境の設定である。これはこのスクリプトが実行されるマシンに依存する。次の read_verilog 文で合成に必要なファイルを読み込む。この際、h ファイルは同じディレクトリに置いておかなければならない。

次に current_design ”poco”により、トップ階層を poco に指定する。

クロックと入出力遅延の設定

design compiler を利用する場合、もっとも神経を使うのはクロックと入出力遅延の設定である。

ここでは、create_clock -period 8.0 clk として、動作クロックを指定する。以後、合成は、この周波数を目標に行われる。すなわち、図 1 に示すように、フリップフロップ間を 8nsec に納め、125MHz の周波数で動作することを狙う。

POCO は現在の所、1 クロックで全ての命令を実行する 1 サイクル CPU であるため、入力と F.F. および F.F. から出力の間の遅延も問題となる。POCO のクリティカルパスがどうなるか見てみよう。

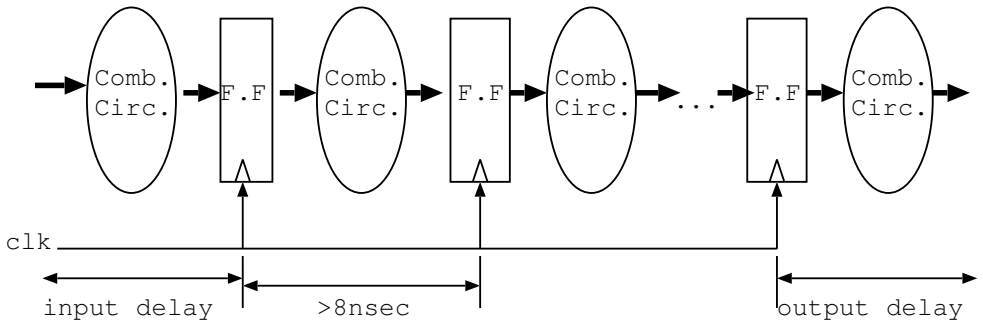

図 1: クロックの指定

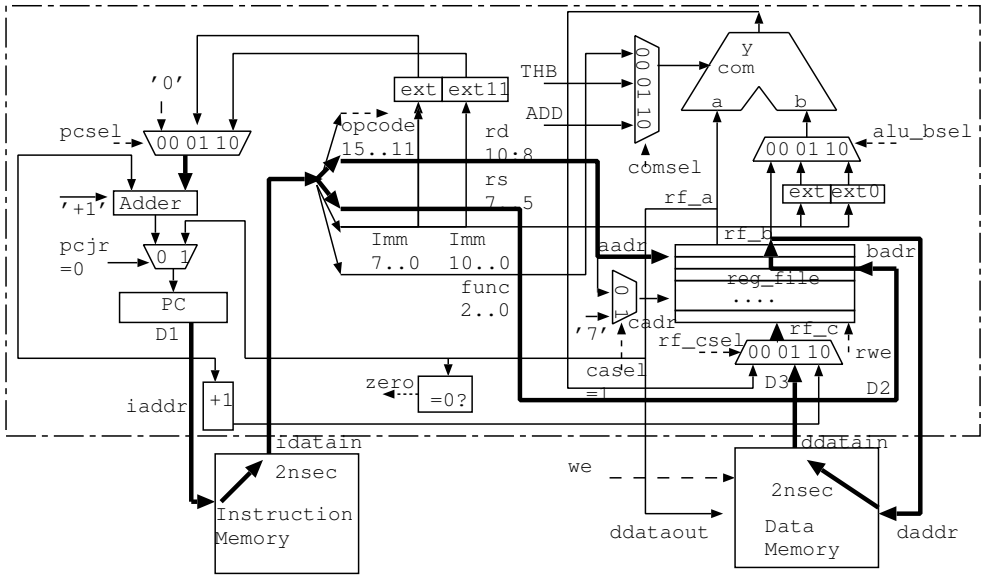

図 2: POCO のクリティカルパス

ここでは、メモリの読み出し遅延を 2nsec としよう。すると、図 2 に示すように、クリティカルパスは、データメモリからデータを読み出す LD 命令実行時に形成されると考えられる。

この場合、クリティカルパスは、以下の和となる。

1. D1:PC が iaddr に出力されるまで
2. 命令メモリの読み出し:2nsec
3. D2:命令が読み出され、この rs フィールドによってレジスタファイルが読み出され、daddr に出力されるまで
4. データメモリの読み出し:2nsec
5. D3:読み出したデータがレジスタファイルに格納されるまで

ここでは、D1 を 0.5nsec, D2 を 2nsec, D3 を 1.5nsec に割り振った。この場合、idatain が到着するまでを 0.5+2=2.5nsec、ddatain が到着するまでを 0.5+2+2+2=6.5nsec となる。そこで、以下のように入力遅延を設定する。

```
set_input_delay 6.5 -clock clk [find port "ddatain*"]
set_input_delay 2.5 -clock clk [find port "idatain*"]
```

次に出力遅延は、所定の時刻に出力が行われるように設定する。すなわち、iaddr は 8-0.5=7.5nsec、daddr は 8-(0.5+2+2)=3.5nsec となる。データメモリ関係の他の信号線も同時期に出力されるので、同じ設定とした。

```
set_output_delay 7.5 -clock clk [find port "iaddr*"]
set_output_delay 3.5 -clock clk [find port "ddataout*"]
set_output_delay 3.5 -clock clk [find port "daddr*"]
set_output_delay 3.5 -clock clk [find port "ddataout*"]
set_output_delay 3.5 -clock clk [find port "we"]
```

次に以下で、ファンアウトすなわち一つの出力に繋がれる入力数を制限してやる。この数値は 10 ー 12 程度が普通である。

```
set_max_fanout 12 [current_design]
```

合成に対する指示

ここでは、合成する面積の目標を入れてやる。面積をできるだけ小さくする場合、以下のように 0 を設定する。もちろん 0 にはならないが、合成系は上記の周波数と遅延を満足した上で、できるかぎり面積を小さくしてくれる。

```
set_max_area 0
compile -map_effort medium -area_effort medium
```

ここでは、がんばり方の程度を medium にする。無理させる際はここに high を指定すると時間をうんと掛けてがんばってくれる。

結果の出力

ここでは、クリティカルパスを長い方から 10 本表示し、面積、電力もレポートする。最後に、合成結果のネットリストを verilog の形で出力する。

```
report_timing -max_paths 10
report_area
report_power
write -hier -format verilog -output poco.vnet
```

合成の実行

```
dc_shell-t -f poco.tcl | tee log
```

で論理合成、圧縮が実行される。ここでは記録が log に格納されるので、結果を覗いて見よう。10 本クリティカルパスが表示される。ここで、slack の所で MET が出れば目標周波数が達成できたことを示す。MET の隣に数字がある場合は、余裕ができた場合で、余裕分の時間が表示される。Violate と出ればこれは、回路の遅延が大きすぎて指定されたクロック周期に間に合わなかったことを示す。この場合、slack の値がマイナスになる。回路の最大動作遅延は、指定した周期-slack になる。slack がマイナスならば、遅延はその分大きいと考える必要がある。

合成された結果のパスをたどってどの辺がもっとも時間がかかるのか調べよう。

```
  Point                                                   Incr       Path
  ------------------------------------------------------------------------
  clock clk (rise edge)                                   0.00       0.00
  clock network delay (ideal)                             0.00       0.00
  input external delay                                    2.50       2.50 r
  idatain[12] (in)                                        0.00       2.50 r
  U68/Y (INVX2)                                           0.06       2.56 f
  U151/Y (NAND3X1)                                        0.16       2.72 r
  U150/Y (NOR2X1)                                         0.12       2.84 f
  U148/Y (NOR2X1)                                         0.31       3.14 r
  U144/Y (NAND2X1)                                        0.19       3.33 f
  U239/Y (INVX2)                                          0.24       3.58 r
  U143/Y (NAND2X1)                                        0.05       3.63 f
  U142/Y (OAI21X1)                                        0.16       3.80 r
  alu_1/b[0] (alu)                                        0.00       3.80 r
  alu_1/sub_7/B[0] (alu_DW01_sub_0)                       0.00       3.80 r
  alu_1/sub_7/U18/Y (INVX1)                               0.14       3.94 f
  alu_1/sub_7/U1/Y (OR2X1)                                0.15       4.09 f
  alu_1/sub_7/U2_1/YC (FAX1)                              0.21       4.30 f
  alu_1/sub_7/U2_2/YC (FAX1)                              0.21       4.51 f
  alu_1/sub_7/U2_3/YC (FAX1)                              0.21       4.72 f
  alu_1/sub_7/U2_4/YC (FAX1)                              0.21       4.94 f
  alu_1/sub_7/U2_5/YC (FAX1)                              0.21       5.15 f
  alu_1/sub_7/U2_6/YC (FAX1)                              0.21       5.36 f
  alu_1/sub_7/U2_7/YC (FAX1)                              0.21       5.58 f
  alu_1/sub_7/U2_8/YC (FAX1)                              0.21       5.79 f
  alu_1/sub_7/U2_9/YC (FAX1)                              0.21       6.01 f
  alu_1/sub_7/U2_10/YC (FAX1)                             0.21       6.22 f
  alu_1/sub_7/U2_11/YC (FAX1)                             0.21       6.43 f
  alu_1/sub_7/U2_12/YC (FAX1)                             0.21       6.65 f
  alu_1/sub_7/U2_13/YC (FAX1)                             0.21       6.86 f
  alu_1/sub_7/U2_14/YC (FAX1)                             0.21       7.07 f
  alu_1/sub_7/U2_15/YS (FAX1)                             0.22       7.30 r
  alu_1/sub_7/DIFF[15] (alu_DW01_sub_0)                   0.00       7.30 r
  alu_1/U104/Y (NAND2X1)                                  0.05       7.35 f
  alu_1/U102/Y (NAND3X1)                                  0.09       7.43 r
```

```
alu_1/y[15] (alu)                                       0.00       7.43 r
U92/Y (AOI22X1)                                         0.06       7.49 f
U91/Y (OAI21X1)                                         0.08       7.57 r
rfile_1/c[15] (rfile)                                   0.00       7.57 r
rfile_1/U89/Y (INVX2)                                   0.16       7.73 f
rfile_1/U185/Y (OAI22X1)                                0.09       7.81 r
rfile_1/r7_reg[15]/D (DFFPOSX1)                         0.00       7.81 r
data arrival time                                                  7.81

clock clk (rise edge)                                   8.00       8.00
clock network delay (ideal)                             0.00       8.00
rfile_1/r7_reg[15]/CLK (DFFPOSX1)                       0.00       8.00 r
library setup time                                     -0.17       7.83
data required time                                                 7.83
-------------------------------------------------------------------------
data required time                                                 7.83
data arrival time                                                 -7.81
-------------------------------------------------------------------------
slack (MET)                                                        0.01
```

ここでは、図 3 に示す、命令コードから ALU を通るパスがクリティカルパスとして報告されている。これは、DC2、3 の設定に余裕があったためであろう。

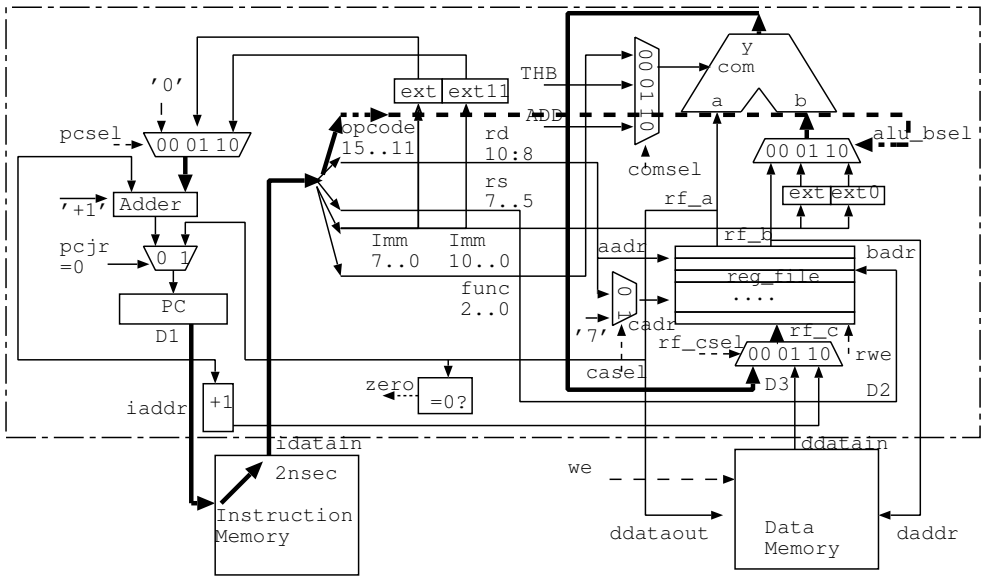

図 3: レポート上のクリティカルパス

次に面積の評価が出力される。ここでは Interconnect area が評価されていないが、これは、配線の面積を指定していないのでやむをえない。最終的な面積は配置配線をしないとわからないのが普通である。ここは、Total cell area の 50477 を覚えておこう。単位は、μm^2 なので、0.24mm 角程度で収まることになる。

最後に消費電力が出力される。一般的に CMOS の消費電力 P は以下の式で表される。

$$P = W * C * Vdd^2 * f + L$$

ここで、Vdd は電源電圧、f は周波数、C は定数、W は稼働率、L は漏れ電流を示す。最初の項が動作することによって消費された動的電力である。合成では、f は指定された周波数を使い、C は各素子のパラメータから算出するが、稼働率 W は分からない。そこで、ここでは W=0.5 として見積っている。この点からこのデータはあくまで目安程度であるといえる。本当に電力を詳しく見積るためには、合成後の回路をシミュレーションして、それぞれのゲートの出力のスイッチング率を求め、そのデータを用いて再計算する必要がある。残念ながら、現在の iverilog ではこの作業がうまく行かず、ここではこの段階は行わないこととする。詳細は 4 年の VLSI 設計論でやることとする。

```
  Cell Internal Power  =   2.7758 mW   (76%)
  Net Switching Power  = 863.1628 uW   (24%)
                         ---------
Total Dynamic Power    =   3.6389 mW  (100%)


Cell Leakage Power     =  88.2152 nW
```

さて、この結果は、トランジスタの内部の貫通電力 (Swtich Power)、トランジスタの負荷容量への充放電による電力 (Internal Power)、漏れ電流 (Leak Power) に分けて表示される。漏れ電流の単位は nW、それ以外は mW である。Switch Power と Int Power が合わせて動的電力に相当する。現状では漏れ電流は無視出来る程小さいが、プロセスが進むにつれてその割合が増大すると言われている。

合成された回路のゲートの接続の様子を見てみよう。これには design_vision というツールを用いる。このツールは design compiler の GUI(Graphic User Interface) である。

```
design_vision
```

と打ち込めば立ち上がる。dc_shell というプロンプトが現れるので、先ほどのスクリプトを実行しよう。

```
source poco.tcl
```

このスクリプトは最後に quit が入っているので、ツールから抜けようとするが、ここは No をクリックしてやる。画面上のゲート表示のメニューをクリックすると、回路図が現れる。POCO がどのように合成されたか確認しよう。

演習 合成をやってみる

前回の課題で、JALR を取り付けた回路に対して、Design Compiler を用いて、合成を行い、回路図を確認しよう。また、面積と電力を求めよ。次に web 上にもっと高速にするような設定を行ったファイルがある。これを使って合成して、面積と電力を求めて提出せよ。