第10回　より安くより速く

CPUの性能とコスト

論理合成と圧縮

サブルーチンコールの実装で、CPUとして一通りの命令の装備が終わった。そこで、ここで、論理合成して性能とコストを見積ってみよう。

今回、論理合成にはdc_shellを用いることにする。今まで使って来たdesign_visionは、合成後の回路図などを見て、合成結果を確認するのには便利だったが、規模が大きくなると現実的ではなくなる。そこで、スクリプトでバッチ処理的に合成するのがdc_shellである。dc_shellは一種のシェルで、コマンドを打ち込むことで合成を制御するのだが、通常はスクリプトファイルを作っておいてこれを実行させる。この制御には、tcl(Tool Command Language)と呼ばれるスクリプト言語を用いるが、奥が深いので、ここでは深く解説しない。興味がある方は各自調べて下さい。以下の記述をpoco.tclというファイルに入れておく。

```
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 20 clk
set_input_delay 0.1 -clock clk [all_inputs]
set_output_delay 0.1 -clock clk [all_outputs]

set_max_fanout 12 [current_design]

set_max_area 0

compile -map_effort medium -area_effort medium

report_timing -max_paths 10

report_area

write -hier -format verilog -output poco.vnet

quit
```

最初のset文は、ライブラリのパスの設定等の環境設定である。次のread_verilogで、Verilogの記述を読み込む。最後にトップ階層をcurrent_designで指定する。次にクロックの設定をcreate_clockで、入出力の遅延をset_input_delayとset_output_delayで行う。この遅延は、今回の設計ではメモリの入出力など外部に素子を接続する場合のタイミングを指定するものだが、ここでは非常に小さく設定しておく。次に最大のファンアウトをset_max_fanoutで指定

し、面積優先ということで set_max_area 0 とする。最後に compile してタイミングと面積を出力する。合成結果は、poco.vnet という名前のファイルで保存する。

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

と打ち込むと、poco.log に合成時のメッセージが記録される。

まず、クリティカルパスが長い方から 10 本表示される。今回は、遅めのクロックを使っているので、余裕で間に合っているはずである。slack とは、残りの余裕の時間である。ここでは最長パスの長さ、5.75ns のみを覚えておこう。すなわち、この CPU は 174MHz で動作することがわかる。

```
  Point                                                   Incr       Path
  ------------------------------------------------------------------------------
  clock clk (rise edge)                                   0.00       0.00
  clock network delay (ideal)                             0.00       0.00
  ir_reg[11]/CLK (DFFSR)                                  0.00       0.00 r
  ir_reg[11]/Q (DFFSR)                                    0.33       0.33 r
  U120/Y (INVX2)                                          0.12       0.46 f
  U294/Y (NAND3X1)                                        0.11       0.57 r
  U119/Y (INVX2)                                          0.06       0.63 f
  U293/Y (NAND3X1)                                        0.36       0.98 r
  U292/Y (NAND2X1)                                        0.25       1.23 f
  U308/Y (INVX2)                                          0.25       1.48 r
  U291/Y (NAND2X1)                                        0.05       1.53 f
  U290/Y (OAI21X1)                                        0.16       1.70 r
  alu_1/b[0] (alu)                                        0.00       1.70 r
  alu_1/sub_7/B[0] (alu_DW01_sub_0)                       0.00       1.70 r
  alu_1/sub_7/U18/Y (INVX1)                               0.14       1.84 f
  alu_1/sub_7/U1/Y (OR2X1)                                0.15       1.99 f
  alu_1/sub_7/U2_1/YC (FAX1)                              0.21       2.20 f
  alu_1/sub_7/U2_2/YC (FAX1)                              0.21       2.41 f
  alu_1/sub_7/U2_3/YC (FAX1)                              0.21       2.62 f
  alu_1/sub_7/U2_4/YC (FAX1)                              0.21       2.84 f
  alu_1/sub_7/U2_5/YC (FAX1)                              0.21       3.05 f
  alu_1/sub_7/U2_6/YC (FAX1)                              0.21       3.26 f
  alu_1/sub_7/U2_7/YC (FAX1)                              0.21       3.48 f
  alu_1/sub_7/U2_8/YC (FAX1)                              0.21       3.69 f
  alu_1/sub_7/U2_9/YC (FAX1)                              0.21       3.90 f
  alu_1/sub_7/U2_10/YC (FAX1)                             0.21       4.12 f
  alu_1/sub_7/U2_11/YC (FAX1)                             0.21       4.33 f
  alu_1/sub_7/U2_12/YC (FAX1)                             0.21       4.54 f
  alu_1/sub_7/U2_13/YC (FAX1)                             0.21       4.76 f
  alu_1/sub_7/U2_14/YC (FAX1)                             0.21       4.97 f
  alu_1/sub_7/U2_15/YS (FAX1)                             0.22       5.20 r
  alu_1/sub_7/DIFF[15] (alu_DW01_sub_0)                   0.00       5.20 r
  alu_1/U104/Y (NAND2X1)                                  0.05       5.24 f
  alu_1/U102/Y (NAND3X1)                                  0.09       5.33 r
```

```
  alu_1/y[15] (alu)                                       0.00       5.33 r
  U146/Y (AOI22X1)                                        0.08       5.41 f
  U145/Y (OAI21X1)                                        0.10       5.50 r
  rfile_1/c[15] (rfile)                                   0.00       5.50 r
  rfile_1/U19/Y (INVX2)                                   0.15       5.65 f
  rfile_1/U50/Y (OAI21X1)                                 0.09       5.75 r
  rfile_1/rf_reg[7][15]/D (DFFPOSX1)                      0.00       5.75 r
  data arrival time                                                  5.75

  clock clk (rise edge)                                  20.00      20.00
  clock network delay (ideal)                             0.00      20.00
  rfile_1/rf_reg[7][15]/CLK (DFFPOSX1)                    0.00      20.00 r
  library setup time                                     -0.18      19.82
  data required time                                                19.82
  --------------------------------------------------------------------------
  data required time                                                19.82
  data arrival time                                                 -5.75
  --------------------------------------------------------------------------
  slack (MET)                                                       14.07
```

ここでは、ir から出発して、ALU を経由して、レジスタファイルに至るパスがクリティカルパスとなっていることがわかる。

次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。

```
                                   Switch   Int      Leak     Total
Hierarchy                          Power    Power    Power    Power    %
------------------------------------------------------------------------------
poco                                  0.264    1.207  104.220    1.471 100.0
```

さて、この結果は、トランジスタの内部の貫通電力 (Swtich Power)、トランジスタ内部および負荷の容量への充放電による電力 (Int 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。

最後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 57754 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。

CPU の性能評価式

CPI

CPU の性能は当然のことながら、プログラムの実行時間により評価する。OS 等のロスを除いた純粋にユーザプログラムの実行時間は以下の式により表される。

$$CPUtime = プログラムの\ CPU\ クロックサイクル数×クロック周期$$

ここで、実行された命令数をカウントし、一命令あたりの平均クロック数を CPI: Clocks Per Instruction とすると、CPU 実行時間は、以下の式で表わすことができる。

$$CPUTime = 命令数× CPI\ ×クロック周期$$

CPI は、CPU に固有のものでなく、実行するプログラムによって変化することに注意されたい。つまり、実行時間の長い命令をたくさん用いるプログラムでは CPI は長くなる。CPI がどのようになるかは、一つのプログラム中での命令の利用頻度に依存する。

POCO の場合、全ての命令は 2 クロックで終了するので、CPI は 2 となるが、通常の CPU は命令毎に実行クロック数が異なるので、どのようなプログラムで動かすかによって CPI は変わってくる。

例題： CPU の性能の尺度に MIPS(Million Instructions Per Second) というのがある。これは、 1 秒間に何百万回命令を実行可能か、という数値である。POCO は何 MIPS になるか？

答： MIPS 値は実行周波数を平均 CPI で割れば良いので、$173/2 = 85.5$MIPS となる。ただし、この MIPS という尺度は、一定時間で実行できる命令数のみを考え、個々の命令の能力が考慮されていないため、命令セットが異なるマシン間の比較には目安程度にしか使えない。

ちなみに、浮動小数点演算能力を持った科学技術用のマシンの性能尺度として MFLOPS(Million Floating Point OPerations Per Second) がある。これは 1 秒間に何百万回浮動小数点演算を行うことができるかを示す。決まった数値計算アルゴリズムを実行するのに要する演算数（命令実行数ではない）は、マシンの命令セットアーキテクチャに依らず一定であるので、この数値は MIPS よりは信頼できるのではないか、というのが MFLOPS の根拠である。しかし、平方根や指数関数など関数演算が入るとこの命令を持つマシンと持たないマシンが出てくるため、話が面倒になる。この問題を解決するため、関数演算を複雑さによって、浮動小数演算の数命令分としてカウントする正規化 MFLOPS という方法も用いられる。このように MFLOPS という尺度は、本来は MIPS 値よりも根拠があるのだが、スーパコンピュータなどで良く用いられるピーク MFLOPS は、そのコンピュータ出すことのできる瞬間最大性能 (逆に考えるとどのような瞬間もこれ以上の性能は出すことができない数値) で実効性能ではないので、性能尺度としては全く信用してはならない[1]。

さらに、同様の考え方として、信号処理用の専用プロセッサである DSP(Digital Signal Processor) などでは、1 秒間に何百万回、所定の演算を実行できるかという性能尺度として、MOPS(Million Operations Per Second) を用いる場合がある。

性能評価手法

POCO の場合、まだキャッシュも装備しておらず、主記憶もシミュレーションモデルなので、評価するのはある意味で簡単だが、一般的なコンピュータシステムの性能は、CPU だけでなく、搭載しているメモリ、キャッシュ、あるいは OS、コンパイラなどのソフトウェアの影響も受ける。ここで、CPU 性能をきちんと評価する方法を解説しておく。

[1]新聞発表などで×× P(ペタ)FLOPS 達成などと書かれているものの多くはピーク値である

評価用のプログラム

CPU の性能は基本的にプログラムを実行して、その時間を測ることで評価する。性能の測定に使われるプログラムは以下のようなものがある。

- 実プログラムに基づくベンチマーク集: 実際のプログラムと、その入力をセットにしたベンチマーク集。一般的な WS/PC の評価には SPEC ベンチマークが用いられる。これには非数値系の SPECint と数値系の SPECfp に分けられる。SPECint は、主として C 言語で記述されており、GCC, Latex, 表計算プログラム、CAD の最適化問題等から成り、SPECfp は FORTRAN が中心で、SPICE などの科学技術計算から成っている。一定の金額を払えば誰でも利用可能である。他にスーパコンピュータ評価用の Perfect Club、マルチプロセッサ用の SPLASH-II、組み込みプロセッサ評価用 EEMBC などがある。

- 簡単なトイプログラム: ソートや、エラトステネスのふるい、8-queen など。アセンブラプログラムで記述できるし、アルゴリズムは誰でも知っているので、ある程度再現性もある。PICO-16 を今評価するならこの方法しか手がないが、一般的にはこの程度の簡単なプログラムでは、特にメモリシステムの挙動がからむ正確な評価を取ることはできない。

- プログラムカーネル: 実プログラムに基づく評価はリアルだが、内部処理が何をやっているのか見にくい。このため、数値計算用のマシンの評価では、様々なプログラムから繰り返し部分のみを抜き出したカーネルが使われる場合も多い。LLL(Lourence Livermore Loops)、Linpack が有名。数値計算のプロはこのループの番号を聞いただけで、その内部の処理の性質を理解することができる。

- 評価専用プログラム: ベンチマークやカーネルは、いくつものプログラムを動かさねばならず、しかもプログラムによって結果の順位が逆転したりして、どちらが高速が判断することが難しい。このため、様々な振る舞いを一つのプログラムに押し込んだ評価専用プログラムを作り、これを一つ動かせば評価を取れるようにしたもの。数値処理用の Whetstone、非数値用の Dhrystone が有名で、かつては評価に頻繁に利用された。しかし、これらのプログラムはコンパイラの最適化を効かなくするために、実際のプログラムではほとんど考えられないような振る舞いをする上、これらの評価用プログラムに特化した最適化を行うコンパイラまで現れた。このため、Hennessy&Patterson のテキストで厳しくこの欠点が指摘された結果、現在はほとんど使われなくなった。

評価結果の整理と報告

評価専用プログラムが使われなくなった現在、評価を行うためには、複数のプログラムを稼働させるのが普通である。この場合は、結果が複数得られることになるが、比較を行う場合は、どちらかのマシンに正規化して、実行したプログラムの相乗平均を取って比較するのが一般的である。これは、ただの相加平均だと基準に取るマシンを変えると比率が代わってしまうことを避けるためである。しかし、相乗平均は非線形性を持つので、出てきた結果がそのまま性能比となるわけではなく、この点には注意しなければならない。

また、評価を取ってこれを報告する場合は、ちょうど自然科学実験のレポートで、実験器具や方法を細かく記述するのと同様に、以下の点を細かく報告しなければならない。この辺は計算機アーキテクチャの世界といえどもきちんとしなければならない。

- ハードウェア:
  - 動作周波数
  - キャッシュ容量
  - メモリ容量
  - Disk 容量
- ソフトウェア:

– OS の種類、バージョン
– コンパイラの種類、オプション

コストの評価

CPU チップとしてのコストはゲート数（面積）と消費電力で評価される。まず、価格は、ゲート数あるいは面積によって支配される。一般論では、半導体のコストは、面積の 4 乗に比例する。これは、面積が大きくなればなるほど、一つの wafer（半導体を作成するための板）から取れる die(チップに格納される半導体 1 個分の切片) の数が減ることと、埃等による欠損から良品が取れる割合（歩留りという）が悪化するためである。とはいえ、新しい半導体チップは開発費およびマスク代が占める割合が大きいため、チップのコストを直接評価するのは、きわめて難しい。

さて、現在の POCO の面積は、0.24mm 角であり、これはかなり小さいチップであることがわかる。ちなみに最近のチップは 1.5cm 角を越えるものも多い。

消費電力評価

合成結果から、今回の POCO は 1.47mW で、かなり小さな電力消費に留まっていることがわかる。これは、動作周波数が低いこと、キャッシュなどのメモリ素子を含まないこと、が大きい。先に述べたように、この値は本来スイッチング率が分からないと信用できるものではない。また、後に紹介するように動作周波数が大きくなるとこれに伴って増大する。

POCO の改良その 1:コスト低減の工夫

それでは POCO のコストを低減する工夫をしてみよう。まず POCO におけるハードウェアの無駄は、IF 状態に ALU が遊んでいるのにも関わらず、別に加算器を使って PC をカウントアップしている点にある。そこで、ALU を PC のカウントアップにも利用できるように改造してみる。このためには、PC の入力側の加算器の機能を ALU に代行させるようにしなければならず、A 入力、B 入力共にマルチプレクサを拡張する必要がある。

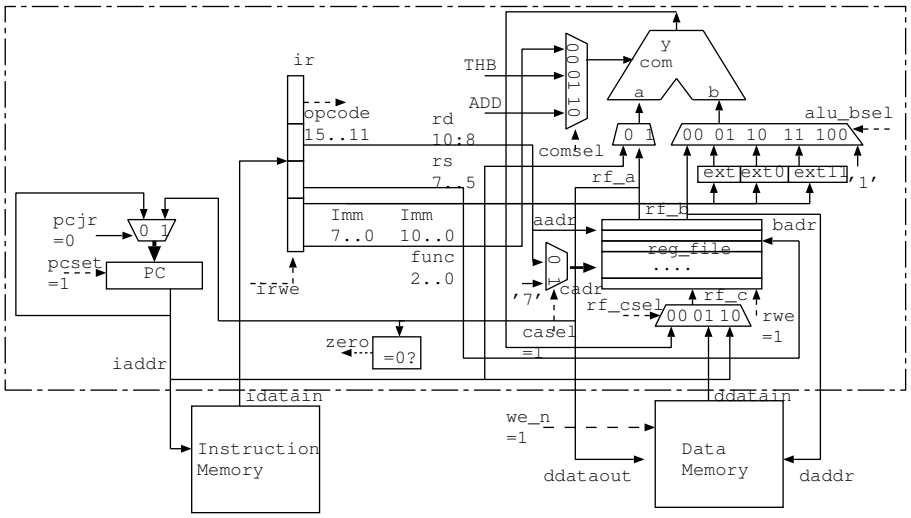

図 1: PC 加算の ALU による実行

図 1 は、このためのパスを付け加えたものである。PC は A 入力に入れる。B 入力では、IF 状態では、+1 するための 1、分岐命令が成立した場合は下位の 8 ビットを符号拡張し、JMP や JAL ならば 11 ビットを符号拡張して入れる。com にも対応した操作を与える必要がある。

Verilog には以下の改造を施せば良い。

```
assign alu_b = stat[‘IF] ? 16’h1:
                (addi_op | ldi_op | branches) ? {{8{imm[7]}},imm} :
                jumps ? {{5{ir[10]}},ir[10:0]} :
                (addiu_op | ldiu_op) ? {8’b0,imm} :
                (ldhi_op) ? {imm, 8’b0} : rf_b;

assign alu_a = (stat[‘IF] | branches | jumps ) ? pc : rf_a;

assign com = (stat[‘IF] | branches| jumps | addi_op | addiu_op ) ? ‘ALU_ADD:
                (ldi_op | ldiu_op | ldhi_op) ? ‘ALU_THB: func[‘SEL_W-1:0];

always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) pc <= 0;
   else if(jumps | pcsel | stat[‘IF] )
     pc <= alu_y;
   else if(jr_op)
     pc <= rf_a;
end
```

この記述を同様に合成してみよう。面積が減っている様子がわかる。

POCO の改良その 2:性能の向上

次に性能の向上を試みよう。論理合成の際にクロックの設定を厳しくしてみる。

```
create_clock -period 2.5 clk
```

この場合、slack(MET) となるが、0.00 すなわち余裕が全くないことになる。したがって、動作周波数は、400MHz で限界ということになる。全ての命令が 2 クロックで動作するため、CPI は 2 となる。すなわち MIPS 値は 200 となる。ただし、面積は 78129 になるので、今までの 57754 からだいぶ増えている。これは高速化のため、面積の大きい加算器を用いたためである。また、消費電力は 13.79mW となり、10 倍近い値となっている。これは、面積が増えたことと動作周波数が増えたことが二重に効いているためである。

さて、ここでクリティカルパスを見ると、やはりレジスタファイル-ALU- レジスタファイルのパスが最大の遅延となっている。そこで、この遅延を切ってみることにする。すなわち、データをレジスタファイルで読み出してマルチプレクサを通した所で格納する。次のクロックで ALU で演算して結果レジスタに格納する。さらに次のクロックでこの結果をレジスタファイルに格納する。このために現在の状態の他に ID(Instruction Decode) と WB(Write Back) を設けて、図 2 のように演算を行って結果をレジスタに格納する命令についてのみ 4 クロックかかるように状態を遷移させて、制御する。

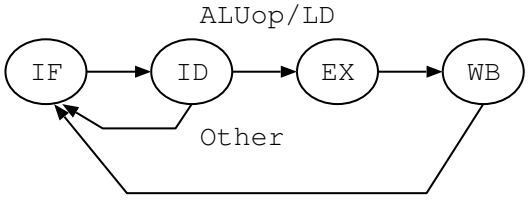

図 2: 4 状態による制御

Verilog 記述は、ALU の入力の alu_a, alu_b をレジスタに変更し、結果を格納する信号もレジスタを rf_c とする。com もレジスタとする必要がある。

```
always @(posedge clk) begin
   if(stat[‘ID]) begin
     if (addi_op | ldi_op)
        alu_b <= {{8{imm[7]}},imm} ;
     else if (addiu_op | ldiu_op)
        alu_b <=  {8’b0,imm} ;
     else if (ldhi_op)
        alu_b <= {imm, 8’b0} ;
     else alu_b <= rf_b;
   end
end

always @(posedge clk)
   if(stat[‘ID]) alu_a <= rf_a;
always @(posedge clk)
   if(stat[‘ID])
     if(addi_op | addiu_op )
        com <= ‘ALU_ADD;
     else if (ldi_op | ldiu_op | ldhi_op)
        com <= ‘ALU_THB;
     else com <= func[‘SEL_W-1:0];

always @(posedge clk)
begin
     if(stat[‘EX]) begin
         if(ld_op)
                rf_c <= ddatain;
         else if(jal_op)
                rf_c <= pc;
         else  rf_c <= alu_y;
     end
end

assign rweop =
        (ld_op  | alu_op | ldi_op | ldiu_op | addi_op | addiu_op | ldhi_op
                | jal_op) ;

assign rwe = stat[‘WB] & rweop;
```

状態遷移は以下のような制御となる。

```
always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) stat <= ‘STAT_IF;
```

```
   else
    case (stat)
      `STAT_IF: stat <= `STAT_ID;
      `STAT_ID: if(rweop) stat <= `STAT_EX;
                else stat <= `STAT_IF;
      `STAT_EX: stat <= `STAT_WB;
      `STAT_WB: stat <= `STAT_IF;
    endcase
end
```

ではこの設計を合成してみよう。クロックの設定をより厳しくして 1.9nsec でも slack が MET する。すなわち、この設計は 526MHz で動作する。今回は、ALU 命令、LD 命令、JAL 命令は 4 クロック、他は 2 クロックで実行する。したがって、実行するプログラム中の命令の比率によって CPI は変動する。ここでは、ALU, LD, JAL が合わせて 50%とする。

$$CPI = 4 * 0.5 + 2 * 0.5 = 3$$

MIPS 値は 176 となり、この場合は 200 よりも下がってしまうことになり、この改造は失敗である。しかし、この方法は、来年習得するパイプライン処理のベースとして重要である。

本日の課題

コストを減らすには、命令用メモリとデータメモリを共通にするという方法がある。これは合成の cell area には影響しないが、システム全体としては大きなコストダウンとなる。ちなみに、メモリを共通化する方法をプリンストンアーキテクチャ、分離する方法をハーバードアーキテクチャと呼ぶ。コストを下げた版の設計を元にして、プリンストン型の POCO を Verilog の設計を変更して実現しよう。idatain, ddatain を datain とし、iaddr、daddr を addr、ddataout を dataout として修正せよ。また合成して動作周波数と面積を求めよ。低出物は、改造後の poco1.v と動作周波数と面積。