POCO のマルチサイクル化

CPU の性能とコスト

論理合成と圧縮

前回、POCO の論理合成と圧縮を行った結果、125MHz(8nsec) で動作し、演習ではコストを大きくして良ければ 128MHz(7.8nsec) でも動作可能であることがわかった。

今回は、もっとコストを下げ性能を上げるために、POCO の構成を変更しよう。構成変更に掛かる前に、一般的な CPU の性能評価法を紹介する。

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。

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

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。

$$CPUTime = \text{命令数} \times CPI \times \text{クロック周期}$$

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。

現在の POCO の場合、全ての命令は 1 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： 目標の周期を逆数にすれば、実行周波数が求められる。今の POCO は 1 命令が 1 クロックで動作するため、MIPS 値=実行周波数である。前回検討したように、比較的無理をしない合成を用いると、8nsec が周期になるので、125MIPS で動作する。

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 OperationsPer 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。

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

[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

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、0.18um プロセスを用いると、約 0.23mm 角 (0.050μm) 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 3.66mW 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、動作周波数がさほど高くないこと、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。

マルチサイクル化によるコストの低減

POCO は、1 サイクルコンピュータである。1 サイクルコンピュータは、1 クロックですべての命令が終わるので、性能的には有利だが、資源の共有ができない。例えば、命令メモリとデータメモリを別々に持たなければならないし、PC をカウントアップするためにも、ALU と独立した加算器を持たせる必要がある。また、命令が複雑になると、全体のサイクルが伸びてしまうのも問題である。

そこで、他のプロセッサ同様に、POCO をマルチサイクル化、つまり 1 命令を複数クロック掛けて実行するように改造しよう。もっとも容易なのは、以下のように分ける方法である。

- 命令をメモリから取ってくる。(Insruction Fetch:IF) 命令を取ってくることを命令フェッチと呼ぶ。これは昔からの習慣で、「命令ゲット」とは言わない。ここで、取ってきた命令をしまっておくレジスタが必要である。このレジスタを命令レジスタ (Instruction Register: ir) と呼ぶ。

- ir に格納された命令を実行 (Execution:EX) する。

このように分けると、一つのメモリに命令とデータの両方を置くことができるようになる。また、pc のカウントアップ、飛び先の計算も ALU で行うことができる。すなわち、以下のように使う。

- IF: メモリのアドレスに pc を与え、読み出してきた命令を ir に格納する。ALU では pc=pc+1 を計算する。
- EX: メモリのアドレスにはレジスタファイルの b ポートの出力を接続し、読み出してきたデータはレジスタファイルに格納して LD 命令を実行できるようにする。一方、書き込みデータにはレジスタファイルの a ポートの出力を接続して ST 命令を実行する。分岐命令、ジャンプ命令の場合は、ALU 出飛び先アドレスを計算する。

マルチサイクル化したデータパスを図 1 に示す。メモリが単一になっている点に着目されたい。このアドレス入力には pc とレジスタファイルの b 出力がマルチプレクサを介して接続される。メモリから読んできた命令を格納するために ir が設けられている。また、pc に対する演算を行うため、ALU の 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。

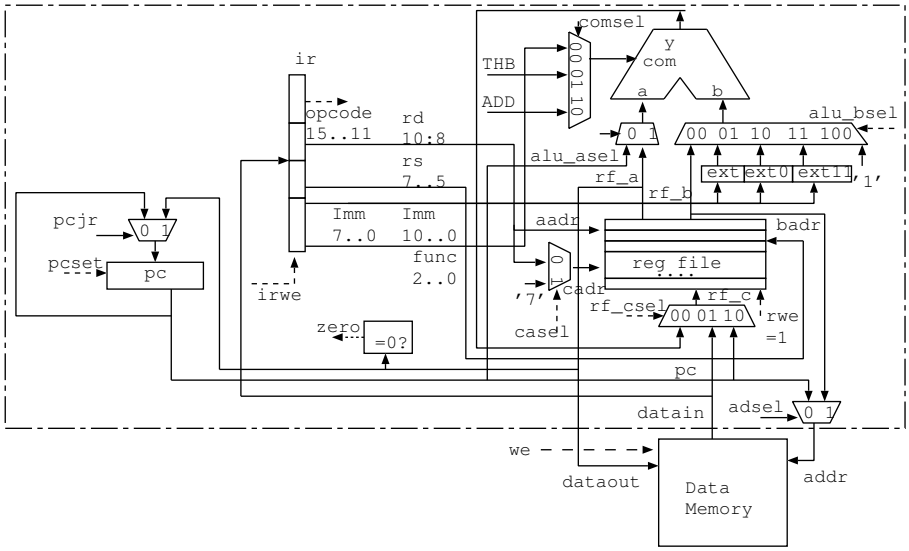

図 1: マルチサイクル化したデータパス

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 にもそれぞれに対応した操作を与える必要がある。

さて、このデータパスを制御するためには、コントローラは今までのような組み合わせ回路ではダメで、「今、命令フェッチをしているのか実行をしているのか」を識別する順序回路が必要である。今回は、単純に IF と EX の 2 つ状態を交互に繰り返す図 2 のような非常に簡単な順序回路を用いる。これは少数のフリップフロップで簡単に実現できる。

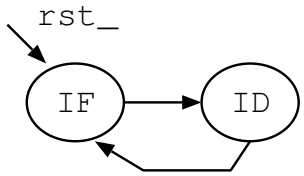

図 2: 状態遷移

各状態で、図中の信号線をどのように制御して良いかを表 1 に示す。単一サイクル制御の場合と違って pc にも、値をセットするかどうかを判断する pcset が必要になる。これが 1 の時だけ入力が設定される。ir も同様である。

表 1: 各命令の制御信号

	pcset	pcsel	adsel	irwe	comsel	alu_asel	alu_bsel	rf_csel	rwe	we
IF	1	0	0	1	-	0	-	-	0	0
ADDI	0	-	1	0	10	1	01	0	1	0
SUB	0	-	1	0	00	1	00	0	1	0
LD	0	-	1	0	-	1	-	1	1	0
ST	0	-	1	0	-	1	-	-	0	1
BNZ	1/0	1	1	0	-	0	-	-	0	0

2 サイクル poco の Verilog 記述

まず、全体を制御するコントローラの記述を示す。

```
reg [‘STAT_W-1:0] stat;
...
always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) stat <= ‘STAT_IF;
   else
    case (stat)
      ‘STAT_IF: stat <= ‘STAT_EX;
      ‘STAT_EX: stat <= ‘STAT_IF;
    endcase
end
```

この記述に関連する定義ファイルの内容は以下の通りである。

```
‘define STAT_W 2
‘define STAT_IF ‘STAT_W’b01
‘define IF 1’b0
‘define STAT_EX ‘STAT_W’b10
‘define EX 1’b1
```

ここでは、状態は stat という 2 ビットのレジスタに格納しておく。0 ビット目が 1 ならば IF 状態、1 ビット目が 1 ならば EX 状態である。このため、Verilog 記述上は、

```
stat[‘IF]
```

が真 (1) ならば IF 状態、

```
stat[‘EX]
```

が真 (1) ならば EX 状態であることがわかる。このように状態と同じ数のビットを用いる方法をワンホットカウンタ (one-hot counter) と呼ばれる。したがってメモリのアドレスバスに、IF 状態では pc を繋ぎ、EX 状態ではレジスタファイルのポート B を接続するためには、以下のように書けば良く、簡単である。

```
assign addr = stat[‘IF] ? pc: rf_b;
```

では、最初から記述を追って行こう。

```
module poco(
input clk, rst_n,
input [‘DATA_W-1:0] datain,
output [‘DATA_W-1:0] addr,
output [‘DATA_W-1:0] dataout,
output we);
reg [‘DATA_W-1:0] pc;
reg [‘DATA_W-1:0] ir;
reg [‘STAT_W-1:0] stat;
wire [‘DATA_W-1:0] rf_a, rf_b, rf_c;
wire [‘DATA_W-1:0] alu_a, alu_b, alu_y;
wire [‘OPCODE_W-1:0] opcode;
wire [‘OPCODE_W-1:0] func;
wire [‘REG_W-1:0] rs, rd, cadr;
wire [‘SEL_W-1:0] com;
wire [‘IMM_W-1:0] imm;
wire [‘JIMM_W-1:0] jimm;
wire pcset, rwe;
wire st_op, bez_op, bnz_op, bmi_op, bpl_op, addi_op, ld_op, alu_op;
wire ldi_op, ldiu_op, ldhi_op, addiu_op, jmp_op, jal_op, jr_op, b_op, j_op;
assign dataout = rf_a;
assign addr = stat[‘IF] ? pc: rf_b;
assign {opcode, rd, rs, func} = ir;
assign imm = ir[‘IMM_W-1:0];
assign jimm = ir[‘JIMM_W-1:0];
```

入出力に関しては、メモリを単一にしたため、addr, datain, dataout ともに一系統で済む。後は ir を宣言した点、JMP や JAL 用に jimm を定義して見やすくした点が異なる。

```
// Decorder
assign st_op = stat[‘EX] & (opcode == ‘OP_REG) & (func == ‘F_ST);
assign ld_op = stat[‘EX] & (opcode == ‘OP_REG) & (func == ‘F_LD);
assign jr_op = stat[‘EX] & (opcode == ‘OP_REG) & (func == ‘F_JR);
assign alu_op = stat[‘EX] & (opcode == ‘OP_REG) & (func[4:3] == 2’b00);
assign ldi_op = stat[‘EX] & (opcode == ‘OP_LDI);
assign ldiu_op = stat[‘EX] & (opcode == ‘OP_LDIU);
assign addi_op = stat[‘EX] & (opcode == ‘OP_ADDI);
assign addiu_op = stat[‘EX] & (opcode == ‘OP_ADDIU);
assign ldhi_op = stat[‘EX] & (opcode == ‘OP_LDHI);
assign bez_op = stat[‘EX] & (opcode == ‘OP_BEZ);
assign bnz_op = stat[‘EX] & (opcode == ‘OP_BNZ);
assign bpl_op = stat[‘EX] & (opcode == ‘OP_BPL);
assign bmi_op = stat[‘EX] & (opcode == ‘OP_BMI);
assign jmp_op = stat[‘EX] & (opcode == ‘OP_JMP);
assign jal_op = stat[‘EX] & (opcode == ‘OP_JAL);
```

```
assign we = st_op;
assign b_op = bez_op | bnz_op | bpl_op | bmi_op;
assign j_op = jmp_op | jal_op ;
```

デコーダは、各演算が有効になるのはEX状態のみである（IF状態ではその前の命令が入っている）ので、EX状態に居るということを条件に付け加えている。それ以外は同じである。また、相対アドレスの分岐命令であることを示すb_op、相対アドレスのジャンプ命令であることを示すj_opを付け加えた。これは記述を見やすくするためである。

```
assign alu_a = (stat[‘IF] | b_op | j_op | jr_op ) ? pc : rf_a;

assign alu_b = stat[‘IF] ? 16’b1:
                (addi_op | ldi_op | b_op) ? {{8{imm[7]}},imm} :
                (addiu_op | ldiu_op ) ? {8’b0,imm} :
                (j_op) ? {{5{jimm[10]}},jimm} :
                (ldhi_op) ? {imm, 8’b0} : rf_b;

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

assign rf_c = ld_op  ? datain : jal_op ? pc : alu_y;
assign rwe = ld_op  | alu_op | ldi_op | ldiu_op | addi_op | addiu_op | ldhi_op
                | jal_op ;
assign cadr = jal_op ? 3’b111 : rd;
alu alu_1(.a(alu_a), .b(alu_b), .s(com), .y(alu_y));

rfile rfile_1(.clk(clk), .a(rf_a), .aadr(rd), .b(rf_b), .badr(rs),
        .c(rf_c), .cadr(cadr), .we(rwe));
```

次にALUのA入力(alu_a)にpcを入れられるようにする。ちなみにIF状態では必ず+1する点に注意されたい。これに対応して、B入力(alu_b)にはIF状態では1を入れる。また、JMP, JAL命令用にirの下位を11ビットを符号拡張したものも入れられるようにする。結果としてこのマルチプレクサはかなり大きくなり、先に解説したバスの形に近くなる。ALUのコマンドにも、IF状態では必ず加算が行われるように設定し、EX状態では分岐、ジャンプ命令にも対応できるようにする。

2サイクルPOCOの合成

さて、この2サイクルPOCOを合成してみよう。1命令を2クロック掛けて実行することを考え、動作周波数を倍すなわち、周期を半分の4nsec(250MHz)に設定してみる。web上のpoco_2c.tclを用いて合成をしてみよう。結果として、動作周波数は達成できるが、面積は以下のようになる。

```
Combinational area:       38234.000000
Noncombinational area:    18272.000000
Net Interconnect area:      undefined  (No wire load specified)

Total cell area:          56506.000000
Total area:                 undefined
```

結果として、1 サイクル POCO に比べて面積は増えてしまっている。pc 周辺の加算器がなくなり、面積が減るはずだったのに、なぜだろう？これには二つの原因が考えられる。

- ir やメモリのアドレスや ALU の入力にマルチプレクサを加えたことによりハードウェア量が増えた。
- 2 倍の動作周波数を実現するため、ALU に高速な演算回路が必要となりコストが増えた。

しかし、メモリは一つで済むので、全体としてコストは削減される。性能面では、CPI が倍になるが、周期が半分となり、結果として同じ性能が実現される。すなわち、MIPS 値は 250/2=125MIPS となる。

3 サイクル POCO

マルチサイクル化を推し進めることで、性能の向上を狙ってみよう。まず、動作周波数を上げるために、2 サイクル POCO のクリティカルパスを見てみよう。レジスタファイルを読み出し、演算を行い、その結果を格納するというパスがもっとも長いことがわかる。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) を設ける。

また、2 サイクル POCO で pc の加算を ALU で行うようにしたが、あまりコストの削減には効果がなかった。そこで、pc のカウントアップや分岐命令の飛び先の計算には、今まで通り専用の加算器を使ってしまうことにする。そして分岐系の命令は 2 クロックで終わるようにしよう。このようにして設計した 3 サイクル POCO の状態遷移を図 3 に示す。

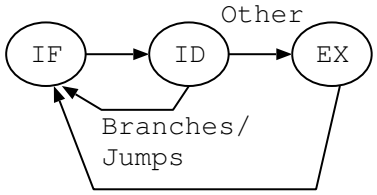

図 3: 3 サイクル POCO の制御回路

3 サイクル POCO

Verilog 記述を、以下のように変更して 3 サイクル POCO を実現する。今回は各命令は ID 状態と EX 状態のどちらで使われるか判らないため、以下のようにデコードする。

```
assign st_op = (opcode == ‘OP_REG) & (func == ‘F_ST);
assign ld_op = (opcode == ‘OP_REG) & (func == ‘F_LD);
assign jr_op = (opcode == ‘OP_REG) & (func == ‘F_JR);
assign alu_op = (opcode == ‘OP_REG) & (func[4:3] == 2’b00);
assign ldi_op = (opcode == ‘OP_LDI);
assign ldiu_op = (opcode == ‘OP_LDIU);
assign addi_op = (opcode == ‘OP_ADDI);
assign addiu_op = (opcode == ‘OP_ADDIU);
assign ldhi_op = (opcode == ‘OP_LDHI);
assign bez_op = (opcode == ‘OP_BEZ);
assign bnz_op = (opcode == ‘OP_BNZ);
assign bpl_op = (opcode == ‘OP_BPL);
assign bmi_op = (opcode == ‘OP_BMI);
```

```
assign jmp_op = (opcode == `OP_JMP);
assign jal_op = (opcode == `OP_JAL);
```

次に ALU の入力の alu_a, alu_b、com をそれぞれレジスタ (reg_a,reg_b,reg_com) に格納する。

```
always @(posedge clk) begin
   if(stat[`ID]) begin
     reg_b <= alu_b;
     reg_a <= rf_a;
     reg_com <= com;
  end
end
```

レジスタファイルへの書き込みは JAL 命令では ID 状態で pc を r7 に格納する。このため、書き込み信号は以下のようになる。

```
assign rf_c = ld_op  ? datain : jal_op ? pc : alu_y;
assign rwe = stat[`EX] & (ld_op  | alu_op | ldi_op | ldiu_op |
                addi_op | addiu_op | ldhi_op) | (stat[`ID] & jal_op) ;
assign cadr = jal_op ? 3'b111 : rd;
```

状態遷移は図 3 に示す制御となる。

```
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(bnz_op | bez_op | bpl_op | bmi_op | jal_op |
                        jmp_op | jr_op ) stat <= `STAT_IF;
                else stat <= `STAT_EX;
      `STAT_EX: stat <= `STAT_IF;
    endcase
end
```

もちろん、stat は 3 ビットに拡張する必要がある。

pc は 1 サイクル POCO と同様に、専用の加算器を用いる設計に戻してやる。

```
always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) pc <= 0;
   else if(stat[`IF])
     pc <= pc+1;
   else if ((bez_op & rf_a == 16'b0 ) | (bnz_op & rf_a != 16'b0) |
                (bpl_op & ~rf_a[15]) | (bmi_op & rf_a[15]))
     pc <= pc +{{8{imm[7]}},imm} ;
   else if (jmp_op | jal_op)
    pc <= pc + {{5{ir[10]}},ir[10:0]};
```

```
   else if(jr_op)
    pc <= rf_a;
end
```

残りの部分は 2 サイクル POCO と同じである。

3 サイクル POCO の合成

ではこの設計を合成してみよう。クリティカルパスにレジスタを入れて 2 クロック掛けて実行するようにした効果があって、クロックの設定をより厳しくして 3nsec でも slack が MET する。すなわち、この設計は 333MHz で動作する。では CPI はどうなるだろう？BNE, BEZ, BMI, BPL, JMP, JR, JAL など分岐命令については 2、それ以外の命令では 3 クロックで動作する。したがって、CPI は実行するプログラムに依存する。

例えば分岐命令が全て合わせて 30%実行される場合を考える。この場合 CPI は以下のようになる。

$$CPI = 2 * 0.3 + 3 * 0.7 = 2.7$$

動作周波数 (MHz) を CPI で割ると MIPS 値を求めることができる。この場合、333/2.7= 123.3 となり、残念ながら 1 サイクル、2 サイクルの 125MIPS よりも遅くなる。分岐命令の確率が 35%の時は、125.7MIPS となり、少しだけ勝つことができる。

一方、面積を見てみよう。

```
Combinational area:       47794.000000
Noncombinational area:    21808.000000
Net Interconnect area:      undefined  (No wire load specified)
Total cell area:          69602.000000
Total area:                 undefined
```

となってかなり増えてしまう。これは、動作周波数を上げたため、高速で高コストな ALU を使う必要が生じたためである。

性能とコストを考えると、どうもこのままでは、この改造はあまりうまく行っていないようだ。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

2 サイクル POCO あるいは、3 サイクル POCO の気に入った JALR 命令を取り付け、合成せよ。周波数を変えて最大動作遅延×面積ができるだけ小さくなるように工夫せよ。前回の synth.pdf 中に記述した通り、最大動作遅延は指定したクロック周期-slack で求めることができる。slack がマイナスになったら遅延は増えるので注意！提出物は、Verilog 記述と合成した場合の最大動作遅延×面積。