マイクロプロセッサ実験

基本的な考え方

マイクロプロセッサ実験は、16bit のマイクロプロセッサを実際に FPGA(Field Programmable Gate Array) ボード上で動かして、ハードウェア設計、入出力インタフェース、アセンブラのプログラミングなどを体感することを目的とする。さらに、グループで一つの製品を一定の時間で作るプロジェクトを通じて、計画、設計、実装のプロセスを学ぶ。2 年の計算機構成までの範囲を前提として実験を行うため、3 年のコンピュータアーキテクチャを履修していなくても問題はない。計算機構成を履修していない学生のためには、特別コースを用意するので、特別コース用ドキュメントをダウンロードされたい。

実験のやり方

今回の実験は今まで 2 年や 3 年春学期まででやってきた実験とは根本的に違うのでまず考え方を変えてもらう必要がある。今までの実験の多くはアナリシス型と呼び、実験中は決められた操作を行ってデータを取って、実験後にこれを分析してその意味を理解、分析する。この型は自然科学における実験の基本であり、再現性のある実験を行うために、実験手法、実験器具等を詳細に記述し、場合によっては温度や湿度まで記録する必要がある。実験後の分析が重要であるため、レポート、ディスカッションが重視される。

しかし、シンセシス型の実験はこれとは全く異なる。この型の実験は、実験中にモノを設計、実装することで、作るモノの本質を理解し、企業などにおける製品開発の基本を学ぶのが目的である。評価はできたモノに対して行われ、ディスカッションは、計画段階で行う。レポートは仕様書に過ぎず、作成には時間を要しない。実験器具など分かりきっていることは書くことはなく、気温や湿度など必要ない。一方、実験中に頭を使って考える必要があり、グループでの仕事の割り振り、協力体制の確立も重要である。

ここではまず、課題とそのミニマムリクアイアメントを決める。例えば、「もぐら叩き」を作るという課題があるとする。ミニマムリクアイアメントは以下の通りである。

3 個の LED 上に 3 匹のもぐらがランダム（っぽく見えれば良い）に出現する。これを対応するボタンを押すことによって叩く。当たれば当たった旨を表示する（LED でも液晶ディスプレイでも良い）。

上記の実装が 3 週間で終われば、成績「3」が付く。付加機能をつける（得点、もぐらの出方など）、プロセッサ、プログラム、プロジェクト管理を工夫するなどすれば、それぞれ状況に応じて加点する。点数は 3 週目のデモンストレーション時に決める。ミニマムリクアイアメントを満足しない場合は、その時の製品の状況の応じて「2」「1」が付く。いくら高機能な実装を目指しても 3 週間目に全く動いていなければ「1」であるので注意されたい。

実験は、以下のスケジュールで行う。それぞれの日に、それぞれのマイルストーンまで達していなければならない。

- 1 週目：午前中は、実験環境のチュートリアルを行い、実験環境を理解する。午後から計画を立て、分担を決め、簡単な計画書を作成する。ハードウェア担当は必要な命令の付加等の作業を行う。4 時前後からディスカッションを行う。マイルストーン：外部仕様、内部仕様が決まり、ハードウェア担当の作業が始まる。
- 2 週目：計画書にしたがい、一日設計と実装に費やす。この日は基本的にフリーだが、帰る前に TA か教員にマイルストーンまで達しているかどうかを確認してもらう。マイルストーン：シミュレーション上で実装した機能の動作が確認される。
- 3 週目：実機でテストする。十分だと思ったら、教員を呼んでデモンストレーションを行う。

計画書、レポートの書き方

今回の計画書およびレポートは、外部仕様と内部仕様に分けて書いて欲しい。計画書は各班で一つ、手書きで簡単なものでよい。レポートは個人で出すが、内部仕様は自分の担当パートのみで良い。計画書に基づく 2、3 枚の簡単

なもので良い。計画通りの実装ができなかった場合はなぜそうなったのか理由を書くこと。また出来上がったものの特長をアピールすること。

- 外部仕様：ユーザ向けの仕様。開発する製品がどのような操作に対してどのように動くかを詳細に決める。設計者の間で動き方の理解に違いがあるとバグの原因になる。
- 内部仕様：どのように実装するか、中身の仕様。ハードウェアの仕様とソフトウェアの仕様に分かれる。ハードウェアの仕様は、付け加える命令のフォーマット、動作等をきちんと定義する。ソフトウェアの仕様は、データ構造（メモリやレジスタの使い方）と、プログラムの流れを示す。ここはアセンブリ言語による実装なので、流れ図を使うと良い。

本来、これにスケジュール線表などが必要だが、今回は 3 週間しかなく、各マイルストーンが明確なので省略する。

グループワーク

今回は基本的にランダムに組みあわせたグループメンバ 3 人 (場合によっては 4 人) によるグループワークとなる。ハードウェア担当 1 名、ソフトウェア担当 2 名に分けて分担して作業を進めると良い。役割分担に必ずしもこだわる必要はないが、一人で全部やろうとしても中々上手く行かないので注意。全員がそれぞれの個性と能力を生かして、良いものを作ることができるように調整すること。このような調整能力は企業では極めて重要である。一般の企業はほとんど全てにおいてグループワークであり、かつ気の合ったメンバと組めるとは限らない。一匹狼で生きようと心に決めている人が居るかもしれないが、グループワークを軽々とこなす能力がないと、一人で生きて行くことはできないのが現実だ。今回の実験はこのようなスキルを磨くチャンスなので、積極的に取り組んで欲しい。

実験環境

実験環境の立ち上げ

各人にラップトップコンピュータを毎時間貸与し、実験終了後に返却する。この番号を覚えておき、毎回同じのを使うこと。アクセサリを含めて丁寧に扱うこと。

ラップトップは windows と linux のデュアルブートになっている。必ずネットワークケーブルをさしてから Linux でブートする。ほっておくと windows が立ち上がってしまう場合があるので注意。実は各班で一台だけ後で windows で立ち上げ直す必要があるのだがこれは後に説明する。

ここで使う Vine Linux は、自動的にウインドウが上がる GUI になっているので、コンソールを開けてそこから作業する。ファイルは、cad0, cad1, cad2 というサーバを経由して NFS で共有されている。なお、最初にログインした時に yppasswd というコマンドを使ってパスワードを変更し、これを覚えておくこと。

一番最初に作業環境をコピーする。

```
cp -r /home/staff2/hunga/jikken .
```

これで、実験に必要なファイル類がコピーされる。

POCO の解説

jikken の中の poco1.v は、計算機構成で習った poco のもっとも基本的なハードウェア記述である。POCO の動作は計算機構成の資料を参照していただければ良いが、簡単に抜粋しておく。

図 1 に、POCO のデータパスを示す。マルチプレクサを各所に設けて命令に応じてデータの流れを切り替えている。4 入力のマルチプレクサは、制御信号が 00 ならば 00 入力が、01 ならば 01 入力、10 ならば 10 入力が出力される。今回は 11 入力は使っていないので実際は 3 つの入力から一つを選ぶ。

図中の太線は 16 ビットデータ、細い点線は制御信号、それ以外の線は 16 ビットよりも短い 3 ビットや 8 ビットのデータを示す。また、命令フェッチ時に主に動作する部分は、太い点線で表している。ext と書かれた箱は、符号拡張を行うハードウェアモジュールを指す。符号拡張は MSB を複製してくっつければ良いためハードウェアとしては大変簡単である。ext0 は、0 拡張ハードウェアモジュールで、上位に 0 を補うだけの単純な働きをする。= 0? と書かれた箱は、読み出したレジスタの値が 0 と等しいかどうかを判断するものでこれも OR 回路のアレイでできた簡単なものである。

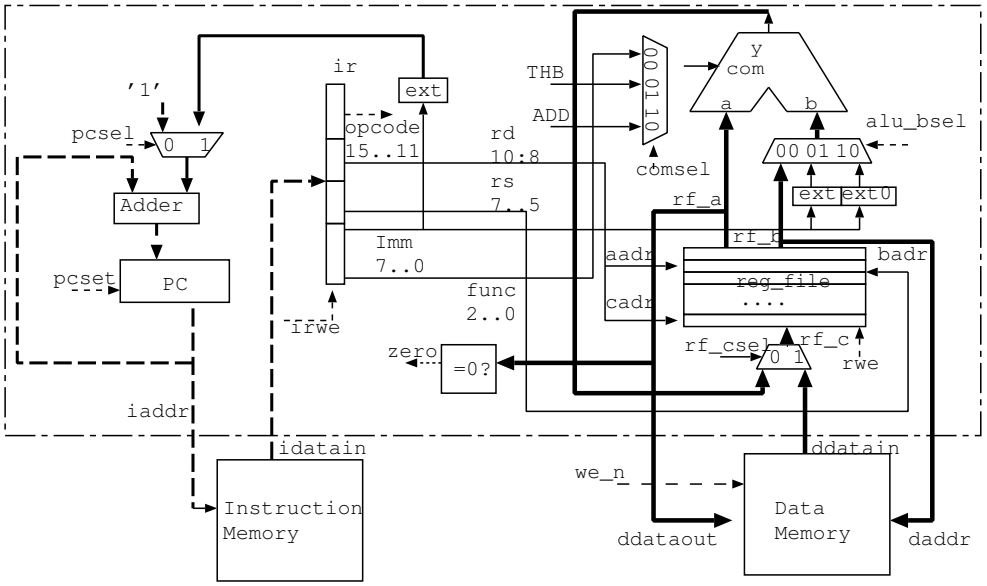

図 1: POCO のデータパス

この図はデータパスを示すため、制御回路は描かれていない。点線の入出力は制御回路に接続されている。

命令フェッチでの動作

それでは、図のデータパスがどのように動くかを解説しよう。まず、アキュムレータマシンと同じく、命令フェッチと実行の二つの状態間を交代に遷移する方式を考える。

命令フェッチでは、この動作はほとんどアキュムレータマシンと同じである。

pc の出力は命令メモリのアドレス iaddr に常に接続されているので、idatain から読み出された命令を ir に格納すれば良い。さらに、pc を一つカウントアップしてやるので、加算器の入力は'1' を選択する。このためには、irwe=1, pcset=1, pcsel=0 とすれば良い。

命令フェッチ時には、演算用のデータパスは動作していないため、演算部の制御信号はどのようなものでも良い。しかし、レジスタファイルやメモリに書き込みを行ってはまずいので、rwe=0, we_n=1 としておく必要がある。

実行時の動作

実行状態では、ir に命令が格納されており、ここでは図 2 に示す二つのタイプに分類できる。まず、I 型は、5bit の opcode(15bit 目-11bit 目)、3bit のディスティネーションレジスタ rd(11bit 目-8bit 目)、8bit の Imm 部 (7bit 目-0 ビット目) を持つ。LDLI, ADDI などのイミーディエイト命令、BNZ などの分岐命令がこのタイプに相当する。

一方、R 型は、5bit の opcode(15bit 目-11bit 目)、3bit のディスティネーションレジスタ rd(11bit 目-8bit 目)、3bit のソースレジスタ rs(7bit 目-5bit 目)、5bit の func(4bit 目-0bit 目) を持つ。

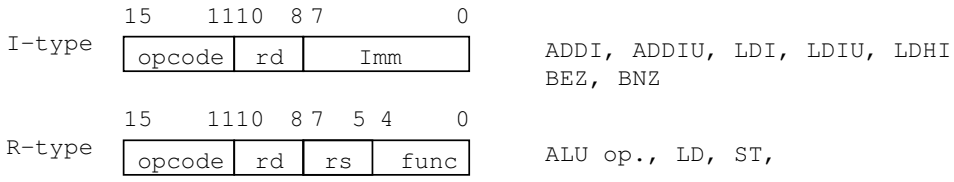

図 2: 命令フォーマット

ここで、ir の rd 部 (11bit 目-8bit 目) は、レジスタファイルの a ポートアドレス (aaddr) と c ポートアドレス (caddr) に、rs 部 (7bit 目-5bit 目) は、レジスタファイルの b ポートアドレス (baddr) に接続しておく。R 型の命令がフェッチされれば、a ポート、b ポートの両方からレジスタが読み出されて演算に使われ、結果は rd の示すレジスタに c ポートから書き込まれる。一方、I 型の命令がフェッチされた場合は、b ポートから読み出されたデータは意味がないことになるが、この場合は使わなければ良いので害はない。

同様に、ir の imm 部 (7bit 目-0bit 目) は、符号拡張ユニット exp,0 拡張ユニット exp0 に常に送られるが、これも必要でないときは使わなければ良いので害になることはない。このようにハードウェアの設計は、準備はしておいて、必要になったら使う、という方針で行う場合が多い。これは余分なマルチプレクサが不要となり、動作速度も向上するからである。しかし、一方で、利用しない回路が動作するため消費電力の点では不利となる。

Verilog 記述の解説

```
‘include "def.h"
module poco(
input clk, rst_n,
input [‘DATA_W-1:0] idatain,
input [‘DATA_W-1:0] ddatain,
output [‘DATA_W-1:0] iaddr, daddr,
output [‘DATA_W-1:0] ddataout,
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_b, alu_y;
wire [‘OPCODE_W-1:0] opcode;
wire [‘OPCODE_W-1:0] func;
wire [‘REG_W-1:0] rs, rd;
wire [‘SEL_W-1:0] com;
wire [‘IMM_W-1:0] imm;
wire pcsel, rwe;
wire st_op, bez_op, bnz_op, addi_op, ld_op, alu_op;
wire ldi_op, ldiu_op, ldhi_op, addiu_op;

assign ddataout = rf_a;
assign iaddr = pc;
assign daddr = rf_b;
```

```
assign {opcode, rd, rs, func} = ir;
assign imm = ir[`IMM_W-1:0];

// Decorder
assign st_op = stat[`EX] & (opcode == `OP_REG) & (func == `F_ST);
assign ld_op = stat[`EX] & (opcode == `OP_REG) & (func == `F_LD);
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 we = st_op;

assign alu_b = (addi_op | ldi_op) ? {{8{imm[7]}},imm} :
(addiu_op | ldiu_op) ? {8'b0,imm} :
(ldhi_op) ? {imm, 8'b0} : rf_b;

assign com = (addi_op | addiu_op ) ? `ALU_ADD:
(ldi_op | ldiu_op | ldhi_op) ? `ALU_THB: func[`SEL_W-1:0];

assign rf_c = ld_op  ? ddatain : alu_y;
assign rwe = ld_op  | alu_op | ldi_op | ldiu_op | addi_op | addiu_op | ldhi_op ;

alu alu_1(.a(rf_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(rd), .we(rwe));

assign pcsel = (bez_op & rf_a == 16'b0 ) | (bnz_op & rf_a != 16'b0);

always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) pc <= 0;
   else if(pcsel)
     pc <= pc +{{8{imm[7]}},imm} ;
   else if(stat[`IF])
     pc <= pc+1;
end

always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) ir <= 0;
```

```
   else if(stat[‘IF])
     ir <= idatain;
end
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

endmodule
```

CPU の状態は、レジスタ stat で記憶し、01 で命令フェッチ、10 で実行を表す。このため、IF を 0,EX を 1 とすると、stat[‘IF] は命令フェッチに居ること、stat[‘EX] は実行状態に居ることを示す。

まず、デコーダの部分を説明しよう。命令レジスタの各フィールドを以下のように分離する。

```
assign {opcode, rd, rs, func} = ir;
assign imm = ir[‘IMM_W-1:0];
```

次に opcode で命令を判別する。これが以下の部分で、これをデコーダと呼ぶ。このデコーダの結果、命令の種類が分かるので、これによって制御信号を簡単に発生できる。

```
// Decorder
assign st_op = stat[‘EX] & (opcode == ‘OP_REG) & (func == ‘F_ST);
assign ld_op = stat[‘EX] & (opcode == ‘OP_REG) & (func == ‘F_LD);
assign 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 we_n = ~st_op;
assign alu_b = (addi_op | ldi_op) ? {{8{imm[7]}},imm} :
(addiu_op | ldiu_op) ? {8’b0,imm} :
(ldhi_op) ? {imm, 8’b0} : rf_b;

assign com = (addi_op | addiu_op ) ? ‘ALU_ADD:
(ldi_op | ldiu_op | ldhi_op) ? ‘ALU_THB: func[‘SEL_W-1:0];
assign rf_c = ld_op  ? ddatain : alu_y;
assign rwe = ld_op  | alu_op | ldi_op | ldiu_op | addi_op | addiu_op | ldhi_op ;
assign pcsel = (bez_op & rf_a == 16’b0 ) | (bnz_op & rf_a != 16’b0);
```

Verilog はマルチプレクサを ？：で表す点に注意されたい。3 入力以上のマルチプレクサは?を 2 回使って条件を順番にチェックしている。

次に以下の記述は、ALU とレジスタファイルの周辺の接続である。

```
alu alu_1(.a(rf_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(rd), .we(rwe));
```

最後の部分はレジスタの記述である。まずは pc である。

```
always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) pc <= 0;
   else if(pcsel)
     pc <= pc +{{8{imm[7]}},imm} ;
   else if(stat[‘IF])
     pc <= pc+1;
end
```

always 文中では、if 文が使えるので、pc の前段のマルチプレクサは if 文で表現されている。また、ここでは+が二ヶ所に表われるが、条件が排他的なので、通常は合成時に一つの加算器が使い回され、図 1 の回路が生成される。

次は、ir だが、これは命令フェッチでは常にセットされるので簡単である。アキュムレータマシンと同じで、irset 信号は明示的に表われない点に注意されたい。

```
always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) ir <= 0;
   else if(stat[‘IF])
     ir <= idatain;
end
```

最後に状態遷移は以下のように記述する。def.h で記述されているが、ここでは、IF は 01、 EX は 10 である。

```
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
```

今回の記述で実装されている命令は以下の通りである。

```
NOP                                        00000 --- --- 00000
MV  rd,rs    rd <- rs                      00000 ddd sss 00001
AND rd,rs    rd <- rd AND rs               00000 ddd sss 00010
```

```
OR  rd,rs    rd <- rd OR rs                  00000 ddd sss 00011
SL  rd       rd <- rd<<1                     00000 ddd --- 00100
SR  rd       rd <- rd>>1                     00000 ddd --- 00101
ADD rd,rs    rd <- rd + rs                   00000 ddd sss 00110
SUB rd,rs    rd <- rd - rs                   00000 ddd sss 00111
ST rs, (ra)     rs -> (ra)                   00000 sss aaa 01000
LD rd, (ra)     rd <- (ra)                   00000 ddd aaa 01001
LDI rd,#X       rd <- X（符号拡張）          01000 ddd XXXXXXXX
LDIU rd,#X      rd <- X（符号拡張なし）      01001 ddd XXXXXXXX
ADDI rd,#X      rd <- rd + X（符号拡張）     01100 ddd XXXXXXXX
ADDIU rd,#X     rd <- rd + X（符号拡張なし） 01101 ddd XXXXXXXX
LDHI rd,#X      rd <- X|0                    01010 ddd XXXXXXXX
BEZ rd, X   if (rd==0) pc <- pc + X          10000 ddd XXXXXXXX
BNZ rd, X   if (rd!=0) pc <- pc + X          10001 ddd XXXXXXXX
```

メモリの接続

imem.v, dmem.v がそれぞれ命令メモリ、データメモリに相当する記述である。このメモリは FPGA の内部に設けられている Block RAM を用いる。Block RAM は作り込みのメモリで IP(Intellectual Property) の一種であり、FPGA に限らず、チップ内のメモリの多くは類似の形式である。配布した記述中で定義されているメモリは小規模で、それぞれ 1K ワードである。実は今回利用する SPARTAN-3AN にはかなりの量の Block RAM が搭載されているのだが、サイズを大きくすると合成に要する時間が長くなるため、このような値に設定されている。大きい容量が欲しい場合、parameter 値を変えれば可能である。このような必要性に迫られた際は TA や教員に相談して欲しい。

imem.v は、

```
$readmemb("led.dat",mem);
```

の部分で、led.dat の初期データを読み込んでいる。この dat データのファイル名をプログラム毎に書き換えると良い。

プログラムの方法については後述する。データメモリである dmem.v では初期データが直接設定されている記述になっている。

```
mem[2] = 3;
mem[3] = 4;
```

これは、imem 同様、ファイルを定義しても良い。

メモリの記述は、この初期設定以外の部分はいじらないで欲しい。これは、下手にいじると Block RAM に自動的に割りつけてくれなくなるためである。メモリと CPU は poco_top.v で接続されている。読み書きのタイミング合わせのため、クロックは反転したものが用いられる。

```
poco poco_1(.clk(clk50M0), .rst_n(rst_n), .idatain(instrd),
            .ddatain(rd), .iaddr(iaddr), .daddr(daddr),
            .ddataout(wd), .we(we));
imem imem_1(.clk(clk50M180), .we(1'b0), .addr(iaddr[9:0]), .d(16'b0), .q(instrd));
dmem dmem_1(.clk(clk50M180), .we(dmem_we), .addr(daddr[9:0]), .d(wd), .q(dmem_rd));
```

I/O の接続

今回用いる SPARTAN スタートキットボードには、スライドスイッチ 4 個 (sw)、ボタンスイッチ 4 個 (north, east, west, south)、LED 8 個、液晶ディスプレイ (LCD) が付いており、これらを実験で利用する。これらの入出力デバイスは、メモリマップト I/O と呼ばれる方法で接続されている。この方法は、I/O にメモリ同様のアドレスを割り付け、そこに LD, ST で値を読み書きすることにより入出力を行う。すなわちメモリと I/O の読み書きが同じように行われる。

ここでは、poco_top.v 中の下の記述で割り当てを行っている。

```
assign led_we = daddr[15] & ~daddr[14] & ~daddr[13] & we;
assign lcd_we = daddr[15] & ~daddr[14] & daddr[13] & we;
assign rst_n = ~btn_rot;
assign rd = daddr[15] & daddr[14] ? {8'b0, north, south, east, west, sw} :
                                                    instrd;
```

すなわち、

- 0x8000: 下位 8 ビットに書き込んだ値が LED に表示される。1 の時点灯する。
- 0xa000: 下位 10 ビットに書き込んだ値が LCD のコマンドとなる。上位 2 ビットが 10 の時には、下位 8 ビットが ASCII コードとして解釈されて、文字が出力される。上位 2 ビットを 00 にして、下位 8 ビットが 0x01 でディスプレイのクリア、0x02 で、左上に書く位置をずらす。それ以外のコマンドに興味がある方は、ボードのマニュアルを参照のこと。(ただし、動かないコマンドもある)。
- 0xc000: 読み出すと、下位 4 ビットがスイッチ、その上が (北、南、東、西) の 4 つのボタンスイッチの値が読める。ボタンスイッチは、基板の上で、大きなロータリースイッチの周辺に配置されているのだが、この位置に応じた名前が付いている。押すと 1 になる。

ちなみに、ボタンスイッチは、押すとチャタリングといって、一定時間データがバタつく。これを防ぐために、チャタリング防止用のカウンタが入っている。したがってシミュレーション上は 1 になるのに非常に時間がかかるため、シミュレーション用のトップでは、この部分は省かれている。スライドスイッチには防止回路が入っていない。

```
chattering chat0(.clk(clk50M0), .rst_n(rst_n), .in(btn_east), .out(east));
chattering chat1(.clk(clk50M0), .rst_n(rst_n), .in(btn_west), .out(west));
chattering chat2(.clk(clk50M0), .rst_n(rst_n), .in(btn_north), .out(north));
chattering chat3(.clk(clk50M0), .rst_n(rst_n), .in(btn_south), .out(south));
```

また、液晶ディスプレイ (LCD) は、実は TA の Moricy さんが作った相当複雑なコントローラを持っている。これが lcdTop であるが、今回はこの中身には立ち入らないことにする。また、クロックは FPGA 内部の DCM というクロック制御用のモジュールを使って整形している。この部分もここではあまり立ち入らないことにする。

```
lcdTop lcdTop0(.clk(clk50M180), .rst(btn_rot),
            .userData(wd[9:0]),
            .userDataEnable(lcd_we),
            .lcdEnable(lcd_e),
            .lcdRegisterSelect(lcd_rs),
            .lcdReadWrite(lcd_rw),
            .dataBitHigh(db[7:4]),
            .dataBitLow(db[3:0]) );
```

```
  pocoDCM PDCM(
                .CLKIN_IN( clk50),
                .RST_IN( btn_rot),
                .CLK0_OUT( clk50M0),
                .CLK180_OUT( clk50M180));
```

今回配布した記述では、ロータリスイッチのボタン機能がリセットに割り当てられている。これを押すとリセットされる。ロータリスイッチの回す部分は使っていない。

実は、このスタータキットは、ここで使う I/O 以外に、ロータリースイッチや音声出力、映像出力を持っており、いろいろ遊ぶことができる。ただし、結構それぞれ使うのが大変なため、実験中には、これらを使わなくても良いようになっている。意欲のある方はマニュアルを参考にして挑戦しても良い。ただし、ちゃんと実験時間内に終わるように。

プログラミング

コンピュータアーキテクチャを履修してくれた方々にはお馴染みの shapa を使ってアセンブリ言語のプログラミングから機械語を生成する。

例えば、スイッチのデータを LED に表示するプログラム led.prg は以下の通りである。

```
        LDHI r2,#0x80   // LED
        LDHI r0,#0xc0   // SW
loop:   LD r1,(r0)
        ST r1,(r2)
        BNZ r2,loop
```

以下のようにすると、led.dat が生成される。

```
./shapa --fpga led.prg -o led.dat
```

ファイル名 led.dat を imem.v 中の readmemb で指定すれば、プログラムに相当する機械語を命令メモリに設定することができる。

シミュレーション

授業中の演習同様、icarus verilog を利用する。ここで、LCD のコントローラはたくさんのファイルを使っているので、スクリプト compv を利用する。実験中にシミュレーションすべきファイルが増やしたい場合は、ここにファイル名を付け加える。ちなみに、シミュレーション用には、DCM やチャタリング防止回路を取り外した poco_top_sim.v を使っている。多分ないとは思うが、poco_top.v をいじったらこちらも同じように変更しないとシミュレーションと実機の動作が違ってしまう。

```
./compv
vvp a.out
```

でシミュレーションの実行が可能である。今回、テストベンチ test_poco.v はかなりいい加減なものなので、各班で用途に応じて書き直して欲しい。ここでは、

```
   initial begin
      $dumpfile("poco.vcd");
```

```
    $dumpvars(0,test);
    clk50 <= `DISABLE;
    btn_rot <= `ENABLE;
    {btn_north, btn_south, btn_east, btn_west, sw} <= 8'b00001010;
  #(STEP*1/4)
  #(STEP*5)
    btn_rot <= `DISABLE;
  #(STEP*100)
    {btn_north, btn_south, btn_east, btn_west, sw} <= 6'b00000000;
  #(STEP*100)
    {btn_north, btn_south, btn_east, btn_west, sw} <= 6'b00001010;
  #(STEP*100)
    {btn_north, btn_south, btn_east, btn_west, sw} <= 6'b00000000;
  $finish;
  end
```

100 クロック毎に、スイッチの値を切り替えている。

また、シミュレーション結果の波形を見るためには、

```
gtkwave poco.vcd
```

と打ち込む。今回の実験の成功のためにはこの波形 viewer を上手く利用することが必要である。演習時はレジスタの値が表示できない欠点があったが、今回は FPGA への実装の都合もあり、レジスタファイルの構造を変えて表示可能なようになっているため、是非利用して欲しい。

実機動作

実機動作のためには、シミュレーションした Verilog ファイル群を論理合成、配置配線を行い、FPGA ボードに送るデータストリーム (poco_top.bit) を作る必要がある。

このために cad0, cad1, cad2 のどれかにリモートログインする。

```
ssh cad0.std.expr.st.keio.ac.jp
```

ラップトップと同じパスワードでログインできる。ログインする度に作業している jikken ディレクトリ中で、以下のコマンドを打ち込む。

```
source setup.csh
```

これで CAD の利用環境が設定される。

論理構成、配置配線は、Xilinx 社の ise WebPack を利用している。この CAD は教育研究用にフリーで公開されており、自宅の PC にインストールすることも可能である。(http://www.xilinx.com を参照) ise はユーザインタフェースを用いて使う場合が多いが、ここでは実験時の面倒な操作を省くため、make 一発でビットストリームができるようになっている。

```
make clean
make
```

と打ち込む。実験中にファイルを増やす場合は、xst_moricy.pl を書き換えてファイルを付け加えてやる。

トラブルがなければ、ビットストリームファイル poco_top.bit が生成される。通常、シミュレーションがきちんと動いた場合、問題なくビットストリームファイルができるはずだが、場合によってはエラーが発生する。この際は TA

に相談のこと。論理合成を試みる前に必ずシミュレーションで動作を確認すること。シミュレーションにより動作を確認していない verilog コードを論理合成に掛けて、エラーが出ても TA は対処しないので注意。

これを SPARTAN ボードに転送する必要があるが、これには Windows マシンが必要になる。このため、実機動作が必要になったら、各班で一台ラップトップをこの用途で使う。デュアルブートで今度は Windows を立ち上げて、初期パスワードでログインする。パスワードは同様に変えておくことが望ましい。Windows マシンへは、ファイル転送プログラム scp で cad0 からファイルを転送する。

さて、ビットストリームファイルをボードに転送するプログラム iMPACT を立ち上げる前に、ボードを接続しておく。まず、ボードを箱から出す。箱の中には黒い電源ケーブルと白い USB ケーブルが入っているので、それぞろを接続する。電源ケーブル脇の POWER と書いてあるスライドスイッチを ON にすると LED が点灯して、ボードに電源が供給されたことがわかる。

次にファイルを転送する。

左下の「すべてのプログラム」から Xilinx ISE Design Suite 11.1web → アクセサリ → ISE→ iMPACT を選択して立ち上げる。

iMPACT Project と出てきて I want to という選択肢が出てくるので、下の方　create a new project の所をクリックして OK ボタンを押す。

次の画面は、そのまま Finish にして JTAG を選ぶ。

すると二つのチップがつながった絵が出て来て、Assign New Configuration File と聞いてくるので、ここで、先ほど作って移動したビットストリームファイルを指定する。

次にもう一度同じことを聞いてくるので、今度は Bypass を指定してやる。

そうすると、Device Programming Properties というのが出てくるので、OK を押してやる。これで転送準備は終わりである。マウス右ボタンをクリックすると、メニューが出てくるので、2 番目の Program FPGA Only をクリックしてやる。これで USB 経由でビットストリームが設定されて FPGA が利用可能となる。転送が成功すると Program Suceeded と書かれた青いマークが現れる。この時ボードは既に自動的にリセットが掛かってプログラムがスタートしている。led.dat を用いた場合、スイッチを ON/OFF したり、ボタンを押したりすると、LED が点灯するのでその様子を確認されたい。

やっかいなのだが、iMPACT を立ち上げっ放しで、同じ名前で 2 回ファイルを転送すると、以前のファイルがもう一度転送されてしまうことがあるので、動作確認をやり直す場合は、ビットストリーム名を変えるか、iMPACT を立ち上げ直すことをお勧めする。

その日の作業が終わったら、ボードは元の通りに静電防止用の袋に入れて箱に入れる。ボードは丁寧に扱うこと。

おまけ

SPARTAN は、Xilinx 社の低価格 FPGA で、ライバル Altera 社の Cyclone と同じく電機製品への組み込みを目標に開発された。このため、低価格に設定されており、数百円クラスから数千円クラスで買える。しかし、性能はバカにできず、今回の実験用ボードの SPARTAN も POCO ならば軽々と実装可能 (利用率は 7%位) で、実際に 50MHz のクロックで動作する（合成上は 100MHz)。FPGA は、開発時プロトタイプ用のデバイス、という概念を覆したチップで、家電、カーエレクトロニクスなど、従来、専用目的の IC(ASIC) を開発していた分野で広く使われるようになっている。

今回のボードも 3 万円程度の価格帯で入手可能で、WebPack もタダなので、実験や趣味で広く利用されている。ただし、それぞれの I/O は、生の形で接続されているので、使いこなすにはマニアックな実装能力が必要である。