情報工学実験第2 マイクロプロセッサ実験

重要なお知らせ

まずは、「実験補足資料」に含まれる「マイクロプロセッサ実験の概要」をよく読んで欲しい。実験の内容、趣旨、実験に使うFPGAボード、製作するゲーム機の様子、ハードウェア・ソフトウェアの作業内容などが書かれている。

理想的な実験スケジュール（3週間分）は以下の通り。5限までは延長可。

週	時限	内容
1週目	2時限	実験の概要説明。班分け。機材配布。チュートリアル1を実施。
	3時限	チュートリアル2を実施。
	4時限	液晶モニタ（LCD）を使えるようにする（班ごとに演習1、演習2を行う）。開発するゲームの外部仕様および内部仕様を検討。
2週目	2時限	先週に引き続き外部仕様および内部仕様を策定。教員とディスカッション。
	3-4時限	FPGAボード上でゲーム開発を開始。LCDを使ったアニメーション、スイッチを使った当たり判定ができた班は解散。
3週目	2-3時限	FPGAボード上でゲームを完成させる。必要に応じて拡張ボードを追加。
	4時限	ゲームが完成した班は、学生、教員、TAの前でデモ発表を行う。デモ発表後、レポート作成の指示を聞いたら解散。

3週目の最後に、完成したゲーム機の発表会を行う。FPGAの実機ボードを用いてゲームを実演してもらい、ゲームの機能や面白さをアピールしてもらう。教員およびTAが審査委員となり、審査委員の合議によって各班の得点が決まる。

チュートリアル1

製品やツールのひと通りの操作方法を説明したドキュメントのことを「チュートリアル」と呼ぶ。FPGAボード、測定機器、開発ツール、アプリケーションなどには通常のマニュアルに加え、チュートリアルが用意されている場合がある。 実際の開発現場では、まず、チュートリアルをこなして全体の流れを理解したうえで、詳細な使い方については適宜マニュアルを読むという流れが一般的である。

作業にはLinuxマシンを使用する。それでは実際にチュートリアルをやってみよう。

サンプルコードとコンパイル

チュートリアル用のファイル一式を手元にコピーして解凍する。

```
> cp ~aa205875/2023/mips-20230828.tar.gz .
> tar zxvf mips-20230828.tar.gz
```

```
> cd mips
```

上記のaa205875は教員のアカウント名である。同じ班のメンバのアカウント名が分かればディレクトリを覗くことができる。班の中でファイルをやり取りする際に活用しよう。

含まれているファイルは以下の通り。

ディレクトリ名	ファイル名	内容
soft/		プロセッサ上で動作するプログラムはsoft/以下で開発する
	Makefile	makeと打つとtest.cをコンパイルし、機械語データを生成する
	test.01.c	プロセッサ上で動作するC言語プログラム（LED、SWITCHのテスト）
	test.02.c	プロセッサ上で動作するC言語プログラム（LED、SWITCH、タイマーのテスト）
	test.03.c	演習2（ソフトウェア）を参照
	crt0.c	プロセッサの初期化用コード（プログラムをコンパイルする際に必要）
	mips.ld	リンカスクリプト（プログラムをリンクする際に必要）
hard/		プロセッサ、メモリ、I/O等のハードウェアモデル（Verilog HDL言語）
	Makefile	makeと打つと論理合成、配置配線を行い、構成データを生成する
	top.v	FPGA上に実装するハードウェア（mipsモジュールを呼び出す）
	mips.v	MIPS R2000互換プロセッサ
	parts.v	MIPS R2000互換プロセッサで使われている部品
	multdiv.v	MIPS R2000互換プロセッサで使われている乗算器、除算器
	fpga.xdc	topモジュールの入出力信号とFPGAの物理ピンとの対応付け
	fpga_z7.xdc	topモジュールの入出力信号とFPGAの物理ピンとの対応付け（**新FPGAボード用**）
hard/script/		上記のMakefileの中で使われるスクリプト類

プロセッサ上で動作するソフトウェアプログラムはsoft/以下で開発する。プロセッサ、メモリ、I/O等のハードウェアモデル（Verilog HDL言語で書かれている）はhard/以下にある。sim/というシミュレーション用のディレクトリもあるが、最近はメニューから外している。興味があるひとは付録Eの「シミュレーション」を参照されたい。

次にC言語サンプルプログラムsoft/test.01.cの中身を確認して欲しい。

```
> cd soft
> cat test.01.c
```

```
/* Do not remove the following line. Do not remove interrupt_handler(). */
#include "crt0.c"
void interrupt_handler(){}

void main()
{
        volatile int *sw_ptr = (int *)0xff04;
        volatile int *led_ptr = (int *)0xff08;
        for (;;)
                *led_ptr = *sw_ptr;
}
```

先頭3行（正確には2行目と3行目）は消してはいけない。これらについては後述する。

0xff08番地はLEDの番地であり、ここに4-bitの値を書き込むと値の通りに4個のLEDが光る。

0xff04番地はSWITCHの番地であり、4個のスライドスイッチと4個の押しボタンの状態を8-bitの値として取得できる。対応するスイッチは下位ビットから順にスライドスイッチ4個、押しボタン4個である。スイッチがオンならば1、オフならば0が読み出される。押しボタン0と1を同時に押すと強制リセットがかかるようになっている（LEDは光らない）。

上記のプログラムではSWITCHの値8-bitをLEDに表示し続ける。SWITCHのオンオフによってLEDの点灯パターンが変化するので、実機で試してみよう。

このtest.01.cをコンパイルして機械語コードprogram.datを生成したい。このためのMakefileはコンパイル対象のソースファイルをtest.cとしているので、まず、test.01.cをtest.cにコピーする。「test.01.cをtest.cにコピーする」代わりにMakefileのSOURCEの部分を修正しても良い。

```
> cp test.01.c test.c
```

次にmakeコマンドによってコンパイルする。コンパイルできたらlessコマンドでprogram.datの中身を確認しよう。中間ファイルprogram.dumpも参考になる。

```
> make
> less program.dat
> less program.dump
```

実機動作確認

ハードウェアおよびソフトウェアを実際のFPGAボード上で動作させるには、まず、ハードウェア（機械語コードprogram.datも含む）をFPGA向けに論理合成、配置配線し、構成データ（ビットストリーム）を得る。PCとFPGAボードをUSBケーブルでつなぎこの構成データをFPGAに焼き込む。

この操作にはXilinx社のVivadoというツールを使う。機能限定ライセンス（WebPACKライセンス）であればXilinx社のホームページから無料で取得できる。今回使用するZynq Z-7010など比較的小規模なFPGAであればWebPACK版で十分である。

以下のようにしてVivadoを起動する。

```
> vivado &
```

Vivadoを用いた論理合成、配置配線、構成データの焼き込みは最初はGUIベースで行う（CUIベースのやり方はチュートリアル2で扱う）。GUIの操作方法は写真を見たほうが分かりやすいので、「実験補足資料」に含まれる「Xilinx Vivadoの使い方」の通りに作業を進めて欲しい。

既有のFPGAボードが生産中止になってしまったため、新しく調達したFPGAボードは仕様が若干異なる。お手元のボードに黄色の「新」シールが貼ってあったら「**新FPGAボード**」である。

チュートリアル2

一通りの操作方法が分かったところで、今度は別のチュートリアルによってハードウェアとソフトウェアの詳細を見ていこう。

まず、サンプルプログラムsoft/test.02.cの中身を確認して欲しい。

```
/* Do not remove the following line. Do not remove interrupt_handler(). */
#include "crt0.c"

/* interrupt_handler() is called every 100msec */
void interrupt_handler()
{
        static int cnt = 0;
        volatile int *led_ptr = (int *)0xff08;
        cnt++;
        if (cnt % 10 == 0)
                *led_ptr = cnt / 10;
}

void main()
{
        for (;;);
}
```

割込み

プログラム中のinterrupt_handler()は100msec（0.1秒）に1回呼び出される。このような処理を割込みハンドラと呼ぶ。この割込みハンドラは10回に1回、つまり、1秒毎にLEDの値を1ずつインクリメントするので、実機で試してみよう。

これを実現するために、hard/top.vに以下のようなタイマー回路が実装されている。

```
/* Timer module (@62.5MHz) */
timer timer (clk_62p5mhz, reset, irq);
```

100msecに1度だけirqという信号の値が1になる。irqはプロセッサに直接つながっていて、irqが1になるとプロセッサは予め決められた番地（本実験環境では0x0100番地）の命令を実行するようになっている。0x0100番地には上述のinterrupt_handler()を呼び出すためのコードが置いてある。詳細はsoft/crt0.cの__vector__:以下のコードを参照。

割込みハンドラの実行中は追加の割込みは禁止である。ユーザからの入力待ちなど時間のかかる処理を割込みハンドラの中で実行してはいけない。

メモリ空間

上述の通り、0xff08番地はLEDの番地である。SWITCHもメモリ空間上の番地に割り当てられている。まずはメモリ空間について説明する。

FPGA上に実現したMIPS互換プロセッサは65,536 bytes（16,384 words）のメインメモリを内蔵している。メインメモリには、プログラムコードとデータが区別なく格納される。

スタックもメインメモリ上に実現され、C言語のローカル変数の使用または関数コールの度にスタック領域が使われる。スタックは上位番地（0xff00）からスタートして下位番地（0x0000）の方向へ伸びていく。

機械語コードprogram.datは、以下の通り、hard/top.vにてRAMに読み込まれる。RAMのサイズは16,384 wordsである。

```
/* Specify your program image file (e.g., program.dat) */
initial $readmemh("program.dat", RAM, 0, 16383);
```

実験で使うFPGAボードには、SWITCH、LED、LCDなどのI/Oが接続されており、プロセッサのメモリ空間にマッピングされている。つまり、プロセッサからSWITCHの番地の値をロード（lw命令）すればSWITCHの状態を取得できる。プロセッサからLEDの番地にデータをストア（sw命令）すればその値の通りにLEDが点灯する。同様に、決められた手順でLCDの番地にコマンドをストアすれば液晶モニタに文字を表示できる。このようなI/O方式をメモリマップトI/Oと呼ぶ。

メモリ空間とその使途を以下にまとめる。メモリ番地は16進数で書かれている。実験では32-bitプロセッサを使っているため、32-bitのメモリ空間（4,294,967,296 bytes、つまり4 Gbytes）を扱えるが、本実験環境のメモリサイズは64 kbytesである。

番地	極性	内容
0x00000000-0x0000ff00	Read/Write	メインメモリ（プログラムコード、データ、スタック領域）。サイズは16,384 words弱。
0x0000ff04	Read Only	SWITCHの番地。この番地の値をロードすると押しボタンおよびスライドスイッチの値を取得できる。下位8-bitの値は、0-bit目から順にスライドスイッチ4個、押しボタン4個に対応しており、スイッチがオンならば1、オフならば0。上位24-bitは常に0を返す。押しボタン0と1を同時に押すと強制リセットがかかるようになっている。
0x0000ff08	Write Only	LEDの番地。この番地に4-bitの値をストアすると、その2進数表記の通りに4個のLEDが点灯する。上位28-bitの値は無視される。
0x0000ff0c	Write Only	液晶モニタ（LCD）の番地。この番地に11-bitの制御コマンドをストアするとLCDに文字が表示される。0から7-bit目までが8-bitデータ、8-bit目がRW信号、9-bit目がRS信号、10-bit目がE信号である（各信号の意味は「LCD表示の基礎知識」で述べる）。上位21-bitは無視される。
0x00010000-0xffffffff	N/A	未使用

VivadoのCUIモード

メモリ空間を理解したところ、さっそくtest.02.cをFPGAボード上で動かしてみたい。

チュートリアル2ではCUI（コマンドライン）ベースの開発方法を覚えよう。チュートリアル1で使用したGUIベースのツールは操作が直感的で分かりやすいという利点があるものの、ツールに慣れてくるとマウス操作を煩わしく感じることもある。操作に慣れたらCUIベースのツールが便利である。

まず、test.02.cをコンパイルして新しいprogram.datを作る。makeの前に前回のコンパイル結果を削除するためにmake cleanしておく。

```
> cd soft
> cp test.02.c test.c
> make clean
> make
```

次に論理合成、配置配線を行って、構成データを生成する。

新FPGAボードの場合はhard/Makefileを以下のように書き換える。つまり、bitgen.tclの代わりにbitgen_z7.tclが呼ばれるように修正する。makeやvivadoコマンドの前の空白はスペースではなく、タブなので注意。

```
all:
        make clean
        #vivado -mode batch -source script/bitgen.tcl
        vivado -mode batch -source script/bitgen_z7.tcl
```

ここもmakeの前にmake cleanしておく。実行内容の詳細はhard/Makefileおよびhard/script/bitgen.tcl（**新FPGAボード**の場合はbitgen_z7.tcl）を参照。

```
> cd hard
> make clean
> make
```

最後にPCとFPGAボードをUSBケーブルでつなぎ、FPGAボードの電源をOnにしたうえでこの構成データをFPGAに焼き込む。実行内容の詳細はhard/Makefileおよびhard/script/program.tclを参照。

```
> make program
```

プログラムを開始するために、押しボタン0と1を同時に押してプロセッサをリセットしよう。

注意: test.cを書き換える度にsoft/以下でmake clean、makeしてprogram.datを作り直し、そのあとhard/以下でmake clean、make、make programする必要がある。

拡張I/Oボード

FPGAボード上には、すでにSWITCH、LED、液晶モニタ（LCD）などのI/Oが載っている。これだけでゲーム機を作ることもできるが、さらにI/Oを追加するとゲーム機っぽくなる。モグラ叩きの例ならば、モグラを叩いたらブザーが鳴る、ライフ（残基）数を追加のLEDに表示する、回転スイッチ（ロータリーエンコーダ）で速度を調整するなど。以下のPmodボードを貸与する。

- Pmod KYPD: 16ボタンのキーパッド
- Pmod BB: ブレッドボード（ブザーを実装済）
- Pmod 8LD: 8個の追加LED
- Pmod SWT: 4個の追加スライドスイッチ
- Pmod BTN: 4個の追加押しボタン
- Pmod ENC: ロータリーエンコーダ（スライドスイッチ、押しボタン付き）

ソフトウェア部分の基礎知識

本実験環境におけるソフトウェアの動作の仕組みを紹介する。各自、目を通しておいて欲しい。

プログラム動作の仕組み

本実験環境では、メインメモリの0番地目に格納されている命令からプログラムの実行が開始される。C言語のmain()関数からではないことに注意されたい。

サンプルプログラム（test.01.cなど）では先頭でcrt0.cをインクルードしていたことを思い出して欲しい。crt0.cにはmain()が呼び出される前のスタートアップ処理がインラインアセンブラで記述されている。下記はcrt0.cを簡略化したもの。

```
__start__:
        lui $sp, 0
        ori $sp, 0xff00
        li $gp, 0
        li $k0, 0x02000101
        mtc0 $k0, $12
        /* Initialize .data */
        /* Initialize .bss */
        j main
```

上記のluiやoriなどはMIPSプロセッサの命令である。詳細はWikipediaのMIPSアーキテクチャを参照されたい。命令は32-bit長なので、0番地目、4番地目、8番地目、12番地目、...の順に命令が格納される。

まず、スタックポインタ（spレジスタ）の初期化（命令1と2）、gpレジスタの初期化（命令3）、CPUの動作モード設定（命令4と5）を行う。CPUの動作モード設定では割込みを許可している。

具体的なコードは省略しているが、その後、初期値を持つデータ領域や初期値を持たないデータ領域を初期化している（Initialize .data/.bssの部分）。最後に、j命令によってC言語のmain()関数へジャンプしている。

このようなmainの前に呼ばれる初期化ルーチンは「C Run Time start up file」（別名crt0）と呼ばれる。組込みシステムでは、一般的に、1) スタックポインタ、キャッシュ、メモリなど各種ハードウェアの初期化、2) 初期値を持つデータ領域の定数値での初期化、3) 初期値を持たないデータ領域の0初期化、4) main関数の呼び出しなどを行う。詳細はcrt0.cを見て欲しい。

メモリアロケーション

本実験環境ではメインメモリは16,384 words分実装されている。ただし、実際にメモリとして利用可能な領域は0x0000番地から0xff00番地までなので16,384 wordsより若干少ない。メインメモリ上に以下の領域が確保される。

- text領域: 機械語命令。
- data領域: 初期値を持つglobal変数、static変数。
- rodata領域: read-onlyなデータ。つまり定数。
- bss領域: 初期値を持たないglobal変数、static変数（プログラム起動時に0に初期化される）。
- stack領域: local変数。関数コールの度にメモリ空間末尾から先頭方向へ伸びる。

プログラムのリンク時に、リンカスクリプトで指定された通りにメモリアロケーションが行われる。下記はsoft/mips.ldを簡略化したもの（実物はもう少し複雑）。0x0000番地目からtext領域が始まり、その直後にrodata、data、bss領域が置かれる。スタックポインタはプログラム冒頭のインラインアセンブラでメインメモリの末尾に設定済みである。

```
ENTRY(__start__)
SECTIONS
{
        .text   : { *(.text); }
        .rodata : { *(.rodata); . = ALIGN(4); }
        .data   : { *(.data); . = ALIGN(4); }
        .bss    : { *(.*bss); . = ALIGN(4); }
}
```

クロス開発ツールの使い方

C言語プログラム（test.c）がどのようにして機械語コード（program.dat）に変換されるか説明する。ここではsoft/Makefileに沿って説明していく。なお、本soft/MakefileではSOURCE=test.cとしている。ファイル名を変更したければSOURCEの値を書き換えれば良い。

まず、C言語プログラム（test.c）をコンパイルして、アセンブリ言語プログラム（program.asm）に変換する。コンパイルオプションはCFLAGSで定義している。最適化レベルは-O0（オー・ゼロ）としているので一切の最適化は行わない。

```
        $(CC) $(CFLAGS) -c -S $(SOURCE) -o program.asm
```

次に、アセンブリ言語プログラム（program.asm）をオブジェクファイル（program.o）に変換する。オブジェクファイルは機械語で書かれた中間ファイルである。

```
	$(AS) program.asm -o program.o
```

上述のリンカスクリプト（mips.ld）を用いてオブジェクファイル（program.o）をリンクし、実行ファイル（program.bin）を生成する。

```
	$(LD) -T mips.ld program.o -o program.bin
```

通常、実行ファイル（program.bin）はプロセッサ上でそのまま実行できる。ただし、今回はこの実行ファイルをVerilog HDLで書かれたmipsモジュールに組み込むという追加ステップが必要である。そこで、soft/Makefileでは実行ファイル（program.bin）を逆アセンブルして、text、data、rodata、bssなどの領域を16進数で書かれたテキストファイル（program.dump）に保存している。

このために逆アセンブル（objdump）や簡単なテキスト処理（grep、tr、awk）を行っている。逆アセンブルの部分を以下に示すが、残りのテキスト処理の部分は少々込み入っているので、興味がある人はsoft/Makefileを確認されたい。

```
	$(DUMP) -D --disassemble-zeroes program.bin > program.dump
```

ここまで読んだらtest.c、program.asm、program.dump、program.datをよく確認し、機械語コード生成の過程を理解して欲しい。

ハードウェア部分の基礎知識

本実験環境において各種I/Oを制御するための周辺回路の仕組みを紹介する。各自、目を通しておいて欲しい。

SWITCHとLEDの制御回路の解説

FPGAチップからSWITCH、LED、LCDなどのI/Oを制御するには、FPGAチップの物理ピン番号と各種I/Oの対応付けが必要である。この対応付けを管理するのが制約ファイル（Xilinx Design Constraints File、略してXDC）である。さっそくhard/fpga.xdc（**新FPGAボード**の場合はfpga_z7.xdc）を確認してみよう。

```
> cd hard
> less fpga.xdc
```

```
##LEDs
set_property PACKAGE_PIN M14 [get_ports {led[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[0]}]
set_property PACKAGE_PIN M15 [get_ports {led[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[1]}]
set_property PACKAGE_PIN G14 [get_ports {led[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[2]}]
set_property PACKAGE_PIN D18 [get_ports {led[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[3]}]
...
```

上記は4個のLEDとFPGAの物理ピンの対応付けを定義している。FPGAの各物理ピンは「M14」のように、アルファベットと数字を組み合わせた名前が付いている。例えば、FPGAのG14ピンはLEDの3つ目（led[2]）につながっている。FPGAの各物理ピンの先に何がつながっているかはFPGAボードのマニュアルに書いてある。

同様に、4個のスライドスイッチ（sw[3:0]）、4個の押しボタン（btn[3:0]）、LCD制御用信号（lcd[10:0]）、クロック（clk_125mhz）についてもfpga.xdc（**新FPGAボード**の場合はfpga_z7.xdc）を確認しよう。他にも、8-bit汎用入出力ポートA、B、C（ioa[7:0]、iob[7:0]、ioc[7:0]）も実装されているが、これらについては後述する。

次に、fpga_topモジュールの入出力ポートを確認しよう。fpga.xdc（**新FPGAボード**の場合はfpga_z7.xdc）に書かれていた信号名（ledなど）がfpga_topの入出力ポートとして宣言されていることが確認できる。

```
> less top.v
```

```
module fpga_top (
        input                   clk_125mhz,
        input           [3:0]   sw,
        input           [3:0]   btn,
```

```
        output  reg     [3:0]   led,
        output  reg     [10:0]  lcd,
        output  reg     [7:0]   ioa,
        output  reg     [7:0]   iob
);
```

SWITCHとLEDの制御回路を紹介する前に、まず、mipsプロセッサの入出力ポートを説明する。hard/top.vの続きを見て欲しい。

```
/* CPU module (@62.5MHz) */
mips mips (clk_62p5mhz, reset, pc, instr, {7'b0000000, irq}, memwrite,
        memtoregM, swc, byteen, dataadr, writedata, readdata, 1'b1, 1'b1);
```

主要な入出力ポートの使途を以下にまとめる。

- clk_62p5mhz: 入力ポート（1-bit）。FPGAボードから供給される125MHzクロック（clk_125mhz）を半分に分周してプロセッサに与える。
- pc: 出力ポート（32-bit）。プログラムカウンタ。命令を読み出すためのメモリ番地を出力。
- instr: 入力ポート（32-bit）。pcで指定したメモリ番地から読み出したMIPS I命令。
- irq: 入力ポート（1-bit）。タイマー回路からの割込み信号。100msecに1回だけ1になる。
- memwrite: 出力ポート（1-bit）。メインメモリもしくはI/Oに値をストアする際は1を出力。それ以外は0を出力。
- dataadr: 出力ポート（32-bit）。ストア対象のメモリ番地を出力。
- writedata: 出力ポート（32-bit）。ストアする書き込みデータ。
- readdata: 入力ポート（32-bit）。メインメモリもしくはI/Oから読み出したデータの値。

メモリマップトI/Oでは、dataadrを基にロード/ストア対象がメインメモリなのか、SWITCHなのか、LEDなのか、LCDなのか判別する必要がある。そのために、ここでは、ロード/ストア対象がメインメモリの場合はcs0信号を1に、SWITCHの場合はcs1信号を1に、LEDの場合はcs2信号を1に、LCDの場合はcs3信号を1にしようと思う。このような判定を行う回路をアドレスデコーダと呼ぶ。以下にアドレスデコーダの例を示す。

```
/* Memory(cs0), Switch(cs1), LED(cs2), LCD(cs3), and more ... */
assign  cs0     = dataadr <  32'hff00;
assign  cs1     = dataadr == 32'hff04;
assign  cs2     = dataadr == 32'hff08;
...
```

SWITCH（cs1）の値をロードするための回路は以下の通りである。8-bitのSWITCHの値（swとbtn）がreaddata1に継続代入されている（readata1の上位24-bitは0に固定）。cs1が1のとき、readdata1の値がプロセッサに与えられる。なお、readdata0はメインメモリから読み出した値であり、cs0が1のときはreaddata0の値がプロセッサに与えられる。

```
assign  readdata        = cs0 ? readdata0 : cs1 ? readdata1 : 0;
...
/* cs1 */
assign  readdata1       = {24'h0, btn, sw};
```

LED（cs2）に値をストアするための回路は以下の通りである。プロセッサからの書き込み信号（memwrite）が1、かつ、cs2が1のとき、プロセッサからの書き込みデータの下位4-bit（writedata[3:0]）をledに書き込む。

```
/* cs2 */
always @ (posedge clk_62p5mhz or posedge reset)
        if (reset)                      led     <= 0;
        else if (cs2 && memwrite)       led     <= writedata[3:0];
```

LCD表示の基礎知識

本実験環境においてLCDに文字等を表示するための方法を紹介する。各自、目を通しておいて欲しい。

LCDの制御方法の概要

チュートリアル2で示した通り、LCDのメモリ番地は0xff0cである。この番地は書き込み専用であり、この番地に11-bitの制御コマンドやデータをストアするとLCDに文字が表示される。0から7-bit目までが8-bitデータ、8-bit目がRW信号、9-bit目がRS信号、10-bit目がE信号である。上位21-bitは無視される。

各信号の意味を以下にまとめる。

信号	説明
lcd[7:0]	LCDに送る8-bitデータ。この8-bitデータは、RSが0ならばコマンドとして扱われ、RSが1ならばデータとして扱われる。データはLCDに表示するASCIIコード（文字コード）である。例えば、0x41は大文字の「A」に対応する。コマンドはLCDの制御コードである。例えば、0x01は「Clear Display」である。
lcd[8]	LCDのRW信号。書き込み時は0、読み込み時は1に設定する。本実験では常に0とする。
lcd[9]	LCDのRS信号。コマンドを送る場合は0、文字コードを送る場合は1。
lcd[10]	LCDのE信号。この信号を0にして40nsec以上待ち、1にして230nsec以上待ち、0に戻して10nsec以上待つと、lcd[7:0]で指定した8-bitデータがボード上のLCDに取り込まれる。

LCDの利用手順は下記の通り。起動時にLCDの初期化を行わなければならない。

- LCDの初期化。詳細は後述。
- 制御コマンドの書き込み。Function Set、Clear Display、Return Cursor Homeなどの制御を行う。
- ASCIIコードの書き込み。画面に表示したい文字「0-9」「A-Z」「a-z」、各種記号。実はカタカナも出せる。
- 必要に応じてStep iiに戻る。例えば、改行してカーソルを2行目に移動させたり、Clear Displayする場合など。

LCDの制御方法の詳細

ボード上のLCDは、FPGA上のプロセッサよりもずっと低い動作周波数で動いている。プロセッサの動作が速すぎるので適切に「待ち」を入れないとLCDがデータやコマンドを正しく受け取れない。

このような「待ち」を作るにはいろいろな方法があるが、精度は要らないのでここでは「空のforループ」を使って時間を稼ぐ。プロセッサの動作周波数は62.5MHzであるから1命令当たり16nsecかかる。例えば、以下のforループでは、ループ1回あたり9命令の処理（lwからnopまで）に翻訳される。つまりforループ1回あたり144nsecである。この特徴を利用して「待ち」を作ろう。

```
/* C言語による空のforループ */
for (i = 0; i < 100; i++);
```

まずはLCDに制御コマンド（0x01、Clear Display）を送る例を紹介しよう。

1. LCDの番地（0xff0c）に対し値を書き込む。この値の下位8-bitは0x01、8-bit目（RW信号）は0、9-bit目（RS信号）は0、10-bit目（E信号）は0である。
2. 40nsec待つ。空のforループのループ回数は1回で良い。
3. LCDの番地（0xff0c）に対し値を書き込む。この値の下位8-bitは0x01、8-bit目（RW信号）は0、9-bit目（RS信号）は0、10-bit目（E信号）は1である。
4. 230nsec待つ。空のforループのループ回数は2回で良い。
5. LCDの番地（0xff0c）に対し値を書き込む。この値の下位8-bitは0x01、8-bit目（RW信号）は0、9-bit目（RS信号）は0、10-bit目（E信号）は0である。
6. 次のデータもしくはコマンドを送信するには1.64msec待つ必要がある。ループ回数は11,389回以上必要だ。

続いて、LCDにASCIIコード（0x41、大文字の「A」）を送る例を紹介しよう。

1. LCDの番地（0xff0c）に対し値を書き込む。この値の下位8-bitは0x41、8-bit目（RW信号）は0、9-bit目（RS信号）は1、10-bit目（E信号）は0である。
2. 40nsec待つ。空のforループのループ回数は1回で良い。
3. LCDの番地（0xff0c）に対し値を書き込む。この値の下位8-bitは0x41、8-bit目（RW信号）は0、9-bit目（RS信号）は1、10-bit目（E信号）は1である。
4. 230nsec待つ。空のforループのループ回数は2回で良い。
5. LCDの番地（0xff0c）に対し値を書き込む。この値の下位8-bitは0x41、8-bit目（RW信号）は0、9-bit目（RS信号）は1、10-bit目（E信号）は0である。
6. 次のデータもしくはコマンドを送信するには40usec待つ必要がある。ループ回数は278回以上必要だ。

LCDに表示する文字の文字コードを調べるには`man ascii`と打つ。例えば、「A」の16進数（Hex）表記は0x41である。「b」ならば0x62である。

主な制御コードは以下の通り。

- 0x01: Clear Display
- 0x02: Return Cursor Home
- 0x06: カーソル自動インクリメントモード（Entry Mode Set）
- 0x0c: Display On
- 0x38: 8-bitモード、2行表示モード（Function Set。8-bitモードなら4-bit目を1。2行表示なら3-bit目を1）

LCDの初期化シーケンスは以下の通り。これをプログラムの冒頭で1度だけ実行すること。初期化シーケンスが完了するとLCDが利用可能になる。

1. 15msec待つ。空のforループのループ回数は104,167回以上あれば良い。
2. Function Setコマンド（8-bitモード、2行表示モード=0x38）を発行。
3. Entry Mode Setコマンド（カーソル自動インクリメント=0x06）を発行。
4. Display Onコマンド（0x0c）を発行。
5. Clear Displayコマンド（0x01）を発行。
6. 1.5msec待つ。ループ回数は10,417回必要。

以上で初期化は終了。以降、LCDにASCIIコードを表示できる。

LCDの制御関数

LCDにコマンドを送るためのC言語関数は下記の通り。

- lcd_init(): LCDの初期化を行う。上述の「LCDの初期化シーケンス」を参考に「待ち」を入れたり、コマンド（0x38、0x06、0x0c、0x01）を発行する。
- lcd_cmd(int cmd): 8-bitのLCD制御コマンドをLCDの番地に書き込む。
- lcd_data(int data): 8-bitのASCIIコードをLCDの番地に書き込む。
- lcd_str(unsigned char *str): 文字列strから1文字ずつ連続してLCDの番地に書き込む。

上記の解説とソースコードを見比べよう。例えば、lcd_data()内の「E=0,RS=1,RW=0」の行において、なぜ0x00000200と論理和を取っているのかよく考えよう。2進数の010は16進数では2である。2進数の110は16進数では6である。

```
void lcd_wait(int n)
{
        int i;
        for (i = 0; i < n; i++);
}
void lcd_cmd(unsigned char cmd)
{
        /* E, RS, RW, DB[7:0] */
        volatile int *lcd_ptr = (int *)0xff0c;
        *lcd_ptr = (0x000000ff & cmd) | 0x00000000; /* E=0,RS=0,RW=0 */
        lcd_wait(1);
        *lcd_ptr = (0x000000ff & cmd) | 0x00000400; /* E=1,RS=0,RW=0 */
        lcd_wait(2);
        *lcd_ptr = (0x000000ff & cmd) | 0x00000000; /* E=0,RS=0,RW=0 */
        lcd_wait(11389);
}
void lcd_data(unsigned char data)
{
        /* E, RS, RW, DB[7:0] */
        volatile int *lcd_ptr = (int *)0xff0c;
        *lcd_ptr = (0x000000ff & data) | 0x00000200; /* E=0,RS=1,RW=0 */
        lcd_wait(1);
        *lcd_ptr = (0x000000ff & data) | 0x00000600; /* E=1,RS=1,RW=0 */
        lcd_wait(2);
        *lcd_ptr = (0x000000ff & data) | 0x00000200; /* E=0,RS=1,RW=0 */
        lcd_wait(278);
}
void lcd_init()
{
```

```
        lcd_wait(104167);
        lcd_cmd(0x38);          /* 8-bit, 2-line mode */
        lcd_cmd(0x06);          /* Cursor auto increment */
        lcd_cmd(0x0c);          /* Display ON */
        lcd_cmd(0x01);          /* Clear display */
        lcd_wait(10417);
}
void lcd_str(unsigned char *str)
{
        while (*str != 0x00)
                lcd_data(*str++);
}
void main()
{
        lcd_init();
        lcd_str("mog mog");
}
```

演習1（ハードウェア）

「ハードウェア部分の基礎知識」で紹介したSWITCHおよびLED回路を参考に、LCD制御回路を作ろう。

LCDを制御するための物理ピンは11本（11-bit）である。0から7-bit目までが8-bitデータ、8-bit目がRW信号、9-bit目がRS信号、10-bit目がE信号である。各信号の意味は「LCD表示の基礎知識」を参照。

LCDを制御するための物理ピン番号は、すでにhard/fpga.xdc（**新FPGAボード**の場合はfpga_z7.xdc）に書かれているので修正は不要である。参考までにfpga.xdcの該当箇所を以下に示す。**新FPGAボード**のfpga_z7.xdcはピン名が若干違うがどちらにせよ修正する必要は無い。

```
set_property PACKAGE_PIN T20 [get_ports {lcd[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd[0]}]
set_property PACKAGE_PIN U20 [get_ports {lcd[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd[1]}]
set_property PACKAGE_PIN V20 [get_ports {lcd[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd[2]}]
set_property PACKAGE_PIN W20 [get_ports {lcd[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd[3]}]
set_property PACKAGE_PIN Y18 [get_ports {lcd[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd[4]}]
set_property PACKAGE_PIN Y19 [get_ports {lcd[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd[5]}]
set_property PACKAGE_PIN W18 [get_ports {lcd[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd[6]}]
set_property PACKAGE_PIN W19 [get_ports {lcd[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd[7]}]

set_property PACKAGE_PIN Y14 [get_ports {lcd[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd[8]}]
set_property PACKAGE_PIN W14 [get_ports {lcd[9]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd[9]}]
set_property PACKAGE_PIN T12 [get_ports {lcd[10]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd[10]}]
```

演習1（ハードウェア）ではhard/top.vを修正する。具体的には、1) fpga_topの中でcs3のためにアドレスデコーダを修正し、2) cs3とmemwriteが有効であるときにlcdにwritedata（ビット幅に注意）を書き込むようにする。

拡張ボードは8-bit汎用入出力ポートA、B、Cに挿入して使うことになる。fpga.xdc（**新FPGAボード**の場合はfpga_z7.xdc）についてはioa[7:0]、iob[7:0]、ioc[7:0]はすでに定義されているので修正は不要である。hard/top.vについては適宜修正する必要がある。サンプルコードではioa[7:0]とiob[7:0]は出力ポート扱いとなっているので、入力ポートとして使用したければ「output reg」を「input」に変更する。必要に応じて、入力もしくは出力ポートとしてioc[7:0]も追加する。アドレスデコーダとしてはcs4、cs5、cs6を割り当てると良い。入力ポートとして使用する場合はcs1、出力ポートとして使用する場合はcs2の回路を参考にすると良い。以下に参考例を示す。

```
module fpga_top (
        input                   clk_125mhz,
        input           [3:0]   sw,
        input           [3:0]   btn,
        output  reg     [3:0]   led,
```

```
        output  reg     [10:0]  lcd,
        //output reg    [7:0]   ioa,
        input           [7:0]   ioa,
        output  reg     [7:0]   iob
);
wire    [31:0]  pc, instr, readdata, readdata0, readdata1, readdata4, writedata, dataadr;
...
assign  readdata        = cs0 ? readdata0 : cs1 ? readdata1 : cs4 ? readdata4 : 0;
...
/* cs4 */
//always @ (posedge clk_62p5mhz or posedge reset)
//      if (reset)                      ioa     <= 0;
//      else if (cs4 && memwrite)       ioa     <= writedata[7:0];
assign  readdata4       = {24'h0, ioa};
```

演習2（ソフトウェア）

押しボタン、LED、LCDを使うゲームプログラムを実機上で動かしてみよう。ソースコードを読みながらこれはどのようなゲームなのか考えてみよう。

LCD関連の関数は空欄になっているが、ここは「LCD表示の基礎知識」で解説したコードを適宜コピーしてこよう。 実機で上記のコードを試すには、演習1（ハードウェア）の通りhard/top.vを修正したうえで、Xilinx Vivadoを用いた論理合成からやり直す必要がある。

実機に焼き込んだ後は押しボタン0と1を同時に押してプロセッサをリセットしよう。実機で動作を確認できたら、自分たちが作るゲームの構想を練って欲しい。

```
/* Do not remove the following line. Do not remove interrupt_handler(). */
#include "crt0.c"

void show_ball(int pos);
void play();
int  btn_check_0();
int  btn_check_1();
int  btn_check_3();
void led_set(int data);
void led_blink();
void lcd_wait(int n);
void lcd_cmd(unsigned char cmd);
void lcd_data(unsigned char data);
void lcd_init();

#define INIT    0
#define OPENING 1
#define PLAY    2
#define ENDING  3

int state = INIT, pos = 0;

/* interrupt_handler() is called every 100msec */
void interrupt_handler()
{
        static int cnt;
        if (state == INIT) {
        } else if (state == OPENING) {
                cnt = 0;
        } else if (state == PLAY) {
                /* Display a ball */
                pos = (cnt < 16) ? cnt : 31 - cnt;
                show_ball(pos);
                if (++cnt >= 32) {
                        cnt = 0;
                }
        } else if (state == ENDING) {
        }
}
void main()
{
        while (1) {
                if (state == INIT) {
                        lcd_init();
```

```
                        state = OPENING;
                } else if (state == OPENING) {
                        state = PLAY;
                } else if (state == PLAY) {
                        play();
                        state = ENDING;
                } else if (state == ENDING) {
                        state = OPENING;
                }
        }
}
void play()
{
        while (1) {
                /* Button0 is pushed when the ball is in the left edge */
                if (pos == 0 && btn_check_0()) {
                        led_blink();    /* Blink LEDs when hit */
                /* Button3 is pushed when the ball is in the right edge */
                } else if (pos == 15 && btn_check_3()) {
                        led_blink();    /* Blink LEDs when hit */
                } else if (btn_check_1()) {
                        break;          /* Stop the game */
                }
        }
}
void show_ball(int pos)
{
        lcd_cmd(0x01);          /* Clear display */
        lcd_cmd(0x80 + pos);    /* Set cursor position */
        lcd_data('*');
}
/*
 * Switch functions
 */
int btn_check_0()
{
        volatile int *sw_ptr = (int *)0xff04;;
        return (*sw_ptr & 0x10) ? 1 : 0;
}
int btn_check_1()
{
        volatile int *sw_ptr = (int *)0xff04;;
        return (*sw_ptr & 0x20) ? 1 : 0;
}
int btn_check_3()
{
        volatile int *sw_ptr = (int *)0xff04;;
        return (*sw_ptr & 0x80) ? 1 : 0;
}
/*
 * LED functions
 */
void led_set(int data)
{
        volatile int *led_ptr = (int *)0xff08;
        *led_ptr = data;
}
void led_blink()
{
        int i;
        led_set(0xf);                   /* Turn on */
        for (i = 0; i < 300000; i++);   /* Wait */
        led_set(0x0);                   /* Turn off */
        for (i = 0; i < 300000; i++);   /* Wait */
        led_set(0xf);                   /* Turn on */
        for (i = 0; i < 300000; i++);   /* Wait */
        led_set(0x0);                   /* Turn off */
}
/*
 * LCD functions
 */
void lcd_wait(int n)
{
```

```
        /* Not implemented yet */
}
void lcd_cmd(unsigned char cmd)
{
        /* Not implemented yet */
}
void lcd_data(unsigned char data)
{
        /* Not implemented yet */
}
void lcd_init()
{
        /* Not implemented yet */
}
```

以下にいくつか追加のヒントを紹介する。

- lcd_data() を使ってLCDに日本語カタカナも表示できる。LCDに表示可能な文字一覧は「Digilent PmodCLP Reference Manual」の2.4節に書いてある。
- 自分で定義したドット絵（カスタム文字）をLCDに表示することもできる。詳しくは付録Cの「便利なサンプルコード集」を参照されたい。
- LCDの制御コマンドとして「LCDの任意の位置にカーソルを移動する」コマンドもある。詳しくは付録Cの「便利なサンプルコード集」を参照されたい。

ディスカッション

教員とのディスカッションは2週目の午前中に行う。班の中でゲーム機の構想を練ったうえで、適当な紙に外部仕様と内部仕様をまとめる。ディスカッションでは、まず外部仕様と内部仕様が書かれた紙を提出し、計画について口頭でプレゼンする。

内部仕様と外部仕様

外部仕様: ユーザ向けの仕様。ゲームのルールや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。例えば、「〇〇ボタンを押すとゲームがスタートする」「〇〇ボタンを押すとモグラを叩く」「LCDの上段にモグラを表示する」「LEDに残機数を表示する」「残機が0になるとゲームオーバー」「ヒットすると振動する」等々。箇条書きで良い。

内部仕様: どのように実装するか、中身の仕様。データ構造やプログラムの流れなどソフトウェアの仕様を書く。データ構造については、どのようなC言語変数（例えば、スコア、残機数、速度）を使うか決める。プログラムの流れについては、下記のリンクを参考に疑似コード、もしくは、フローチャート（流れ図）を描いて説明する。例えば、「スタートボタンの入力待ち」→「モグラを表示」→「叩くボタンの入力待ち」→「ヒットしたらスコアをインクリメント、ミスしたら残機をデクリメント」→「残機が0ならばゲームオーバー、そうでなければモグラ表示に戻る」等々。計画した機能が実装できるかどうかの自己確認にもなるので、プログラムの流れ図は可能な限り詳細に描くこと。

理解度を確認するための質問

実験で使っているシステムにはOSもスレッドライブラリもない。このようなシステムにおいて、例えばテニスゲームならば、「画面のボールを動かしつつ、ユーザからの入力を待ち、もし入力があればラケットを振る」という同時処理が必要である。

これを実現するにはどのようなプログラムを書けば良いだろうか？自分の言葉で説明して欲しい。

ヒント

以下のプログラムを考えて欲しい。

```
/* interrupt_handler() is called every 100msec */
void interrupt_handler()
{
        ボールの位置を動かす処理;
        画面を更新する処理;
}
void main()
{
        初期化;
        while (1) {
                if (ボタン0が押された) {
```

```
                        ボタンが0が押されたときの処理;
                } else if (ボタン1が押された) {
                        ボタンが1が押されたときの処理;
                }
        }
}
```

ここでは画面として2行16文字のLCDを使う。画面の表示パターンは配列buf[2][16]で管理し、値が1のマスにボールを表示するものとする。

「ボールの位置を動かす処理」は時間に応じてbufの値を更新する。例えば、buf[0][5]=1だったものを100msec後にbuf[0][6]=1にすればボールが右に1マス移動したことになる。

「画面を更新する処理」はbufのパターンに応じて画面を更新する。例えば、bufの値が1のマスに「*」を表示し、それ以外のマスには「 」を表示する。

「ボタン0が押されたときの処理」はbufのパターンに応じてラケットにボールが当たったかどうかの当たり判定を行う。「ボタン1が押されたときの処理」も大体一緒である。

必ずしもこのように作る必要はないが、このようにすることで「画面のボールを動かしつつ、ユーザからの入力を待ち、もし入力があればラケットを振る」という同時処理を簡単に実現できる。詳細は「演習2（ソフトウェア）」を読んで欲しい。

最終レポートに関する注意

最終レポートの内容は内部仕様と外部仕様である。2週目のディスカッションのときよりも詳細に書くこと。分量の目安はA4サイズ4ページ以上。内容が少ないとプロジェクトへの貢献が少ないと判断され、不当に低い点数になってしまうので注意。自分の貢献を積極的にアピールして欲しい。

外部仕様はA4サイズ1ページ以上。ゲームのルール（スコアの加点条件、ゲームの終了条件等）、画面の見方（スコア、残機数等）、スイッチ/ボタンの意味（スタートボタン、叩くボタン等）など。画面の見方やスイッチ/ボタンの意味は図や実機の写真を使って説明すると良い。内容は班の中で共通化して良い。

内部仕様の1ページ目には詳細なフローチャートを入れること。大ざっぱなフローチャートは減点する。このフローチャートはプログラム中のすべての関数（すべての処理）を網羅すること。フローチャート自体は班の中で共通化して良いが、自分が貢献した部分についてはフローチャートに色付けするなどして強調して欲しい。色付けする箇所が班の中で被っても良い。実際、複数人で協力して1つのモジュールを実装することはよくある事である。

内部仕様の2ページ目以降は、自分が貢献した部分（フローチャートに色付けした箇所）を詳細に解説して欲しい（A4サイズ2ページ以上）。ソースコードもしくは疑似コードを適宜含めて、自分の貢献を積極的にアピールして欲しい。ハードウェア実装を頑張った場合はVerilog HDLコードを含めて説明すると良い。拡張ボードの実装に力を注いだ場合は拡張ボードの写真もレポートに含めると良い。繰り返しになるが、内容が少ないとプロジェクトへの貢献が少ないと判断され、不当に低い点数になってしまうので注意。

締切までにレポートを提出すること。遅れると減点もしくは受け取ってもらえない。実験アンケートには必ず回答すること。

付録A: ハードウェア記述言語Verilog HDL

ハードウェア記述言語Verilog HDLは、天野先生の計算機構成同演習（B2秋学期）、コンピュータアーキテクチャ（B3春学期）で扱っている。

実験中にVerilog HDLの文法を確認したくなったら、授業のホームページにある「Verilog HDL入門」（VLSI設計演習、B4春学期、担当:松谷）を適宜参照して欲しい。Verilog HDLに触れるのは今回が初めてという人は一通り読んでおいたほうが良い。

Verilog HDLシミュレーションにはIcarus Verilogを使用する。テストベクタのファイル名をadd_test.v、回路のファイル名をadd.vとするとき、以下のように打つとVerilog HDLソースがコンパイルされる。Verilog HDLファイルが複数個ある場合はすべて指定すること。テストベクタはファイルリストの最初に指定する。

```
> iverilog test_add.v add.v
```

実際のシミュレーションは以下のように実行する。

```
> vvp a.out
```

iverilogコマンドのオプションは以下の通り。

- -h : ヘルプを表示。
- -o <ファイル名> : 出力ファイル名を指定。デフォルトではa.out。
- -I <パス名> : インクルードパスに追加するディレクトリを指定。
- -D <マクロ名>(=<値>) : defineを追加。値はなくても良い。

テストベクタに以下のような記述を入れておくと、シミュレーション中の信号変化を波形として生成できる。例えば、sim/test.vなどを参照。この場合、test.fpgaというインスタンスの信号変化を、dump.vcdという名前のファイルに記録する。

```
initial begin
       $dumpfile("dump.vcd");
       $dumpvars(0, test.fpga);
end
```

波形ビューワ（GTKWave）を起動するには以下のように打つ。

```
> gtkwave dump.vcd &
```

付録B: プログラミング言語C

演算子とその優先順位を以下にまとめる。文法については、市販の教科書もしくはウェブ等を参照。

優先順位	演算子
1	( )　[ ]　間接メンバアクセス ->　直接メンバアクセス .　後置型インクリメント ++　後置型デクリメント --
2	論理否定 !　ビット否定 ~　前置型インクリメント ++　前置型デクリメント --　ポインタ演算子 *　ポインタ演算子 &　sizeof
3	キャスト (型)
4	乗算 *　除算 /　剰余 %
5	足し算 +　引き算 -
6	左シフト <<　右シフト >>
7	関係演算子 <　<=　>　>=
8	等値演算子 ==　!=
9	ビット積 &
10	ビット排他的論理和 ^
11	ビット和 |
12	論理積 &&
13	論理和 ||
14	条件演算子 ?:
15	代入演算子 =　+=　-=　*=　/=　%=　&=
16	カンマ演算子 ,

付録C: 便利なサンプルコード集

以下のサンプルコードでは省略しているが、プログラムの先頭にはcrt0.cのインクルード、及び、interrupt_handler()が必要。

LCD（16文字2段）の任意の位置に文字を表示したい

このLCDでは上から1段目に書き続けても自動的に2段目に改行されない。例えば、2段目左端に文字を表示したければ、次に示す方法でカーソルを2段目左端に移動させたうえで、lcd_str()もしくはlcd_data()を用いて文字を表示すれば良い。

```
/* 1段目左から0番目にカーソルを移動 */
lcd_cmd(0x80);
/* 1段目左から1番目にカーソルを移動 */
lcd_cmd(0x81);
...
/* 1段目左から15番目にカーソルを移動 */
lcd_cmd(0x8f);

/* 2段目左から0番目にカーソルを移動 */
lcd_cmd(0xc0);
/* 2段目左から1番目にカーソルを移動 */
lcd_cmd(0xc1);
...
/* 2段目左から15番目にカーソルを移動 */
lcd_cmd(0xcf);
```

ちなみに、上記LCDの上位互換品として「20文字4段」バージョンもある。3段目と4段目にカーソルを移動するには下記のようにする。

```
/* 1段目左から0番目にカーソルを移動 */
lcd_cmd(0x80);
/* 2段目左から0番目にカーソルを移動 */
lcd_cmd(0xc0);
/* 3段目左から0番目にカーソルを移動 */
lcd_cmd(0x94);
/* 4段目左から0番目にカーソルを移動 */
lcd_cmd(0xd4);
```

数値（スコア等）をLCDに表示したい

以下のlcd_digit3()は引数で与えた3桁の数字をLCDに表示する関数である。

```
void lcd_digit3(unsigned int val)
{
	int digit3, digit2, digit1;
	digit3 = (val < 100) ? ' ' : ((val % 1000) / 100) + '0';
	digit2 = (val <  10) ? ' ' : ((val %  100) /  10) + '0';
	digit1 = (val %  10) + '0';
	lcd_data(digit3);
	lcd_data(digit2);
	lcd_data(digit1);
}
void main()
{
	lcd_init();

	lcd_digit3(9876);
	lcd_data(' ');
	lcd_digit3(432);
	lcd_data(' ');
	lcd_digit3(65);
	lcd_data(' ');
	lcd_digit3(7);
}
```

LCDを使ってアニメーションを表示したい

LCDの左から右に「abcdefghijklmnop」という文字列が流れる。

```
void main()
{
	int i, j;
	unsigned char data[16] = "abcdefghijklmnop";
```

```
        lcd_init();

        for (i = 0; i < 16; i++) {
                /* Return Cursor Home */
                lcd_cmd(0x02);
                /* Display 16 characters in data */
                for (j = 0; j < 16; j++)
                        lcd_data(data[j]);
                /* Shift */
                for (j = 0; j < 16 - 1; j++)
                        data[j] = data[j + 1];
                data[j] = ' ';
                /* Wait for a while */
                lcd_wait(1000000);
        }
}
```

LCDに自分で定義したドット絵を表示したい

横5ドット、縦7ドットの「ドット絵」を使ってカスタム文字を表現できる。

カスタム文字のパターンはCGRAMというメモリ領域に保存する。CGRAMは0番地から7番地まで利用できるので、最大8個のカスタム文字を定義できることになる。

下記の例では、bitmapで示すカスタム文字のパターン（横5ドット、縦7ドット）をCGRAMの3番地に保存している。次にlcd_data(CGRAMの番地)を呼び出すことでこのカスタム文字を表示している。

```
void lcd_customchar(unsigned int addr, unsigned int *bitmap)
{
        lcd_cmd((addr << 3) | 0x40); /* Set CGRAM address */
        lcd_data(bitmap[0]);
        lcd_data(bitmap[1]);
        lcd_data(bitmap[2]);
        lcd_data(bitmap[3]);
        lcd_data(bitmap[4]);
        lcd_data(bitmap[5]);
        lcd_data(bitmap[6]);
        lcd_data(0x00);              /* Last line is used by cursor */
        lcd_cmd(0x80);               /* Set DDRAM address (write to display) */
}
void main()
{
        unsigned int bitmap[7] = {      0x15, /* 10101 */
                                        0x0a, /* 01010 */
                                        0x15, /* 10101 */
                                        0x0a, /* 01010 */
                                        0x15, /* 10101 */
                                        0x0a, /* 01010 */
                                        0x15, /* 10101 */
                                };
        lcd_init();
        /* New character is defined and stored in CGRAM address 0x03 */
        lcd_customchar(0x03, bitmap);
        /* Display the character stored in CGRAM address 0x03 */
        lcd_data(0x03);
}
```

ブザーで音階を作りたい（ソフトウェア版）

8音から成る音階をソフトウェアで作ってみた。出来はあまり良くないので各自で調整して欲しい。

ここではブザーは汎用入出力ポートBにつながっているものとする。入出力ポートBの番地は0xff14としているが、班によっては違う番地を使っている可能性がある。必要に応じて修正して欲しい。

```
void beep(int mode)
{
        int len;
```

```
        volatile int *iob_ptr = (int *)0xff14;
        switch (mode) {
        case 1: len = 13304; break;
        case 2: len = 11851; break;
        case 3: len = 10554; break;
        case 4: len =  9949; break;
        case 5: len =  8880; break;
        case 6: len =  7891; break;
        case 7: len =  7029; break;
        case 8: len =  6639; break;
        }
        *iob_ptr = 1;
        lcd_wait(len);
        *iob_ptr = 0;
        lcd_wait(len);
}
void main()
{
        int i;
        while (1) {
                for (i = 0; i < 250; i++) beep(1);
                for (i = 0; i < 250; i++) beep(2);
                for (i = 0; i < 250; i++) beep(3);
                for (i = 0; i < 250; i++) beep(4);
                for (i = 0; i < 250; i++) beep(5);
                for (i = 0; i < 250; i++) beep(6);
                for (i = 0; i < 250; i++) beep(7);
                for (i = 0; i < 250; i++) beep(8);
        }
}
```

しかし、このソフトウェア版には音を鳴らしている最中に他の処理ができないという致命的な欠点がある。例えば、音を鳴らしている最中に画面を更新するようなことはできない。この問題は次のハードウェア版で解決する。

ブザーで音階を作りたい（ハードウェア版）

13音から成る音階のハードウェア版である。top.vに修正が必要であるが、先にプログラムがどのようになるかを示しておく。汎用入出力ポートBの番地（ここでは0xff14）に音階を書き込んだ後は別の処理ができるという点が利点である。

```
void main()
{
        volatile int *iob_ptr = (int *)0xff14;
        *iob_ptr = 1; lcd_wait(7000000);
        *iob_ptr = 2; lcd_wait(7000000);
        *iob_ptr = 3; lcd_wait(7000000);
        *iob_ptr = 4; lcd_wait(7000000);
        *iob_ptr = 5; lcd_wait(7000000);
        *iob_ptr = 6; lcd_wait(7000000);
        *iob_ptr = 7; lcd_wait(7000000);
        *iob_ptr = 8; lcd_wait(7000000);
        *iob_ptr = 9; lcd_wait(7000000);
        *iob_ptr = 10; lcd_wait(7000000);
        *iob_ptr = 11; lcd_wait(7000000);
        *iob_ptr = 12; lcd_wait(7000000);
        *iob_ptr = 13; lcd_wait(7000000);
}
```

このためにhard/top.vにブザーを制御するためのbeepモジュールを追加する。beepモジュールはfpga_topモジュールのendmoduleの後に追加すると良い。assign文中の定数は、TAさんが何度もブザー音を聞きながらドレミファソラシドに近づけたものである。

```
module beep
(
       input clk_62p5mhz,
       input reset,
       input [7:0] mode,
       output buzz
);
reg  [31:0] count;
wire [31:0] interval;
```

```
assign interval =       (mode ==  1) ? 14931 * 2: /* C  */
                        (mode ==  2) ? 14093 * 2: /* C# */
                        (mode ==  3) ? 13302 * 2: /* D  */
                        (mode ==  4) ? 12555 * 2: /* D# */
                        (mode ==  5) ? 11850 * 2: /* E  */
                        (mode ==  6) ? 11185 * 2: /* F  */
                        (mode ==  7) ? 10558 * 2: /* F# */
                        (mode ==  8) ?  9965 * 2: /* G  */
                        (mode ==  9) ?  9406 * 2: /* G# */
                        (mode == 10) ?  8878 * 2: /* A  */
                        (mode == 11) ?  8380 * 2: /* A# */
                        (mode == 12) ?  7909 * 2: /* B  */
                        (mode == 13) ?  7465 * 2: /* C  */
                        0;
assign buzz = (mode > 0) && (count < interval / 2) ? 1 : 0;
always @ (posedge clk_62p5mhz or posedge reset)
        if (reset)
                count   <= 0;
        else if (mode > 0)
                if (count < interval)
                        count   <= count + 1;
                else
                        count   <= 0;
endmodule
```

fpga_topモジュール中からこのbeepモジュールにアクセスしたい。そのためにfpga_topモジュール中にbeepモジュールのインスタンスを追加する。さらにbeepモジュールが使用するregやwireも追加する。

```
reg     [7:0]   mode;
wire            buzz;
beep beep (clk_62p5mhz, reset, mode, buzz);
```

汎用入出力ポートBがcs4空間に割り当てられている場合、modeおよびiobの制御は次のようになる。

```
always @ (posedge clk_62p5mhz or posedge reset)
        if (reset)                      mode    <= 0;
        else if (cs4 && memwrite)       mode    <= writedata[7:0];
always @ (posedge clk_62p5mhz or posedge reset)
        if (reset)                      iob     <= 0;
        else                            iob[0]  <= buzz;
```

キーパッドを使いたい

キーパッドには4行4列の16個のボタンがある。1行目は左から1、2、3、A。2行目は左から4、5、6、B。3行目は左から7、8、9、C。4行目は左から0、F、E、Dに対応している。

キーパッドの使い方はやや特殊である。8-bitピンのうち、下位4-bitを出力、上位4-bitを入力ポートとして使う。 ここでは、汎用入出力ポートBにキーパッドがつながっていると仮定し、その下位をiob_lo、上位をiob_hiとする（汎用入出力ポートAを使う場合はioa_lo、ioa_hiになる）。

まず、ボタンが押されているか知りたい列を0、そうでない列を1として、4-bitの値を作る。例えば、ボタン1、4、7、0のどれかが押されているか知りたい場合は0111となる。ボタン3、6、9、Eのどれかがが押されているか知りたい場合は1101となる。これをiob_loに出力し、数サイクル待つ。

次に、iob_hiの値を読み取る。読み取った4-bitの値は「指定した列」の行の値に対応している。例えば、値が0111ならば1行目、値が1101ならば3行目が押されているということになる。

以下にサンプルコードを示す。kypd_scan()は押されているボタンの数字に対応した値が戻り値として返される。

```
int kypd_scan() {
        int i;
        volatile int *iob_ptr = (int *)0xff14;

        *iob_ptr = 0x07;                /* 0111 */
        for (i = 0; i < 1; i++);        /* Wait */
        if ((*iob_ptr & 0x80) == 0)
```

```
                return 0x1;
        if ((*iob_ptr & 0x40) == 0)
                return 0x4;
        if ((*iob_ptr & 0x20) == 0)
                return 0x7;
        if ((*iob_ptr & 0x10) == 0)
                return 0x0;

        *iob_ptr = 0x0b;                /* 1011 */
        for (i = 0; i < 1; i++);        /* Wait */
        if ((*iob_ptr & 0x80) == 0)
                return 0x2;
        if ((*iob_ptr & 0x40) == 0)
                return 0x5;
        if ((*iob_ptr & 0x20) == 0)
                return 0x8;
        if ((*iob_ptr & 0x10) == 0)
                return 0xf;

        *iob_ptr = 0x0d;                /* 1101 */
        for (i = 0; i < 1; i++);        /* Wait */
        if ((*iob_ptr & 0x80) == 0)
                return 0x3;
        if ((*iob_ptr & 0x40) == 0)
                return 0x6;
        if ((*iob_ptr & 0x20) == 0)
                return 0x9;
        if ((*iob_ptr & 0x10) == 0)
                return 0xe;

        *iob_ptr = 0x0e;                /* 1110 */
        for (i = 0; i < 1; i++);        /* Wait */
        if ((*iob_ptr & 0x80) == 0)
                return 0xa;
        if ((*iob_ptr & 0x40) == 0)
                return 0xb;
        if ((*iob_ptr & 0x20) == 0)
                return 0xc;
        if ((*iob_ptr & 0x10) == 0)
                return 0xd;

        return 0;
}
void main()
{
        volatile int *led_ptr = (int *)0xff08;
        for (;;)
                *led_ptr = kypd_scan();
}
```

これを実現するには、汎用入出力ポートiob[7:0]をiob_lo[3:0]とiob_hi[3:0]に分ける必要がある。これにはhard/fpga.xdc（**新FPGAボード**の場合はfpga_z7.xdc）の該当箇所を以下のように修正する。

```
##Pmod Header JE
set_property PACKAGE_PIN V12 [get_ports {iob_lo[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {iob_lo[0]}]
set_property PACKAGE_PIN W16 [get_ports {iob_lo[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {iob_lo[1]}]
set_property PACKAGE_PIN J15 [get_ports {iob_lo[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {iob_lo[2]}]
set_property PACKAGE_PIN H15 [get_ports {iob_lo[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {iob_lo[3]}]

set_property PACKAGE_PIN V13 [get_ports {iob_hi[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {iob_hi[0]}]
set_property PACKAGE_PIN U17 [get_ports {iob_hi[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {iob_hi[1]}]
set_property PACKAGE_PIN T17 [get_ports {iob_hi[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {iob_hi[2]}]
```

```
set_property PACKAGE_PIN Y17 [get_ports {iob_hi[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {iob_hi[3]}]
```

hard/top.vの入出力ポート宣言は下記のように修正する。

```
module fpga_top (
        input                   clk_125mhz,
        input           [3:0]   sw,
        input           [3:0]   btn,
        output  reg     [3:0]   led,
        output  reg     [10:0]  lcd,
        output  reg     [7:0]   ioa,
        //output reg    [7:0]   iob
        output  reg     [3:0]   iob_lo,
        input           [3:0]   iob_hi
);
```

汎用入出力ポートBがcs4空間に割り当てられている場合、iob_loとiob_hiの制御は下記の通り。これに加え、cs4とreaddata4によるreaddataの制御も必要になるので忘れずに。

```
/* cs4 */
assign  readdata4       = {24'h0, iob_hi, iob_lo};
always @ (posedge clk_62p5mhz or posedge reset)
        if (reset)                      iob_lo  <= 0;
        else if (cs4 && memwrite)       iob_lo  <= writedata[3:0];
```

ロータリーエンコーダを使いたい

ロータリーエンコーダには回転スイッチとスライドスイッチが実装されていて、回転スイッチのシャフトは押しボタンにもなっている。回転スイッチのシャフトは時計回り、反時計回りに回る。

ロータリーエンコーダ自体の出力は4-bitで、ここでは汎用入出力ポートC（つまり、ioc）の上段4-bitにロータリーエンコーダがつながっていると仮定する。 4-bitのうち、最初の2-bit（AとB）の位相差によって回転スイッチのシャフトが時計回りか反時計回りかを判定し、次の2-bitにはシャフトの押しボタンとスライドスイッチがつながっている。

シャフトの回転方向を判定する部分は専用回路として作る。 hard/top.vにロータリーエンコーダを制御するためのrotary_encモジュールを追加する。次のようなrotary_encモジュールをfpga_topモジュールのendmoduleの後に追加すると良い。

```
module rotary_enc
(
        input clk_62p5mhz,
        input reset,
        input [3:0] rte_in,
        output [9:0] rte_out
);
reg     [7:0]   count;
wire            A, B;
reg             prevA, prevB;
assign  {B, A} = rte_in[1:0];
assign  rte_out = {count, rte_in[3:2]};
always @ (posedge clk_62p5mhz or posedge reset)
        if (reset) begin
                count   <= 128;
                prevA   <= 0;
                prevB   <= 0;
        end else
                case ({prevA, A, prevB, B})
                4'b0100: begin
                        count <= count + 1;
                        prevA <= A;
                end
                4'b1101: begin
                        count <= count + 1;
                        prevB <= B;
                end
                4'b1011: begin
                        count <= count + 1;
```

```
                        prevA <= A;
                end
                4'b0010: begin
                        count <= count + 1;
                        prevB <= B;
                end
                4'b0001: begin
                        count <= count - 1;
                        prevB <= B;
                end
                4'b0111: begin
                        count <= count - 1;
                        prevA <= A;
                end
                4'b1110: begin
                        count <= count - 1;
                        prevB <= B;
                end
                4'b1000: begin
                        count <= count - 1;
                        prevA <= A;
                end
                endcase
endmodule
```

rotary_encモジュールの入力（rte_in）はロータリーエンコーダ本体の4-bitで、rotary_encモジュールの出力（rte_out）は10-bitである。

rte_out[9:0]のうち最初の2-bitは、シャフトの押しボタン（rte_in[3]）とスライドスイッチ（rte_in[2]）がそのままつながっている。 次の8-bitはカウンタ出力になっていて、初期値が128で、シャフトが反時計回りに回されるとカウンタの値が増え、時計回りに回されるとカウンタの値が減る。 なお、カウンタは8-bitなので、カウンタの値が255のときに反時計回りに回すとカウンタの値は0に戻り、カウンタの値が0のときに時計回りに回すとカウンタの値は255になってしまう点に注意。

fpga_topモジュール中からこのrotary_encモジュールにアクセスしたい。 汎用入出力ポートC（つまり、ioc）にロータリーエンコーダがつながっていると仮定するので、hard/top.vの入出力ポート宣言は下記のように修正する。

```
module fpga_top (
        input                   clk_125mhz,
        input           [3:0]   sw,
        input           [3:0]   btn,
        output  reg     [3:0]   led,
        output  reg     [10:0]  lcd,
<中略>
        input           [7:0]   ioc
);
```

次にfpga_topモジュール中にrotary_encモジュールのインスタンスを追加する。さらにrotary_encモジュールが使用するwireも追加する。 ここでは汎用入出力ポートCがcs6空間に割り当てられているものとする。

```
wire    [31:0]  readdata6;
wire            cs6;
wire    [9:0]   rte;
rotary_enc rotary_enc (clk_62p5mhz, reset, ioc, rte);
```

rotary_encモジュールの出力（rte_out）は10-bitなので、これをreaddata6につなぐ。これに加え、cs6とreaddata6によるreaddataの制御も必要になるので忘れずに。

```
/* cs6 */
assign  readdata6       = {22'h0, rte};
```

以下にソフトウェア側のサンプルコードを示す。 ここでは入出力ポートCの番地は0xff18としているが、ここは必要に応じて修正して欲しい。

```
void main()
{
```

```
        volatile int *led_ptr = (int *)0xff08;
        volatile int *rte_ptr  = (int *)0xff18;
        lcd_init();
        for (;;) {
                *led_ptr = (*rte_ptr) & 0x3;
                lcd_cmd(0x80);  /* Cursor first line */
                lcd_digit3((*rte_ptr) >> 2);
        }
}
```

シャフトの押しボタンとスライドスイッチのオンオフ状態をLEDの下位2-bitに出している。 rotary_encモジュールの8-bitカウンタの値はLCDに表示している。値の表示には上述のlcd_digit3()関数を使っている。

7セグメントLEDを使いたい

2桁の7セグメントLEDによって、00～99までの数字を表示できる。

2桁7セグメントLEDの出力は4-bitが2つ横に並んでいるため、ここでは汎用入出力ポートA（つまり、ioa）の上段4-bit、および、汎用入出力ポートB（つまり、iob）の上段4-bitを使うものとする。

ここでは0～99までの数字（7-bit必要）を入力として受け取り、4-bitの制御信号を2つ出力する専用回路を作る。 hard/top.vに2桁7セグメントLEDを制御するためのseg7ledモジュールを追加する。次のようなseg7ledモジュールをfpga_topモジュールのendmoduleの後に追加すると良い。

```
module seg7led (
        input clk_62p5mhz,
        input reset,
        input [6:0] num,        /* 0 ... 99 */
        output [3:0] dout1,
        output [3:0] dout2
);
reg             digit;
reg     [19:0]  counter;
assign  dout1 = seg7out1(digit ? digit1(num) : digit10(num));
assign  dout2 = {digit, seg7out2(digit ? digit1(num) : digit10(num))};
always @ (posedge clk_62p5mhz or posedge reset)
        if (reset) begin
                counter <= 0;
                digit   <= 0;
        end else if (counter < 20'd625000)
                counter <= counter + 1;
        else begin
                counter <= 0;
                digit   <= ~digit;
        end
function [3:0] digit1 (input [6:0] num);
        if (num < 10)           digit1  = num;
        else if (num < 20)      digit1  = num - 10;
        else if (num < 30)      digit1  = num - 20;
        else if (num < 40)      digit1  = num - 30;
        else if (num < 50)      digit1  = num - 40;
        else if (num < 60)      digit1  = num - 50;
        else if (num < 70)      digit1  = num - 60;
        else if (num < 80)      digit1  = num - 70;
        else if (num < 90)      digit1  = num - 80;
        else                    digit1  = num - 90;
endfunction
function [3:0] digit10 (input [6:0] num);
        if (num < 10)           digit10 = 0;
        else if (num < 20)      digit10 = 1;
        else if (num < 30)      digit10 = 2;
        else if (num < 40)      digit10 = 3;
        else if (num < 50)      digit10 = 4;
        else if (num < 60)      digit10 = 5;
        else if (num < 70)      digit10 = 6;
        else if (num < 80)      digit10 = 7;
        else if (num < 90)      digit10 = 8;
        else                    digit10 = 9;
endfunction
function [3:0] seg7out1 (input [3:0] din);
```

```
        case (din)
        4'd0: seg7out1 = 4'b1111;
        4'd1: seg7out1 = 4'b0000;
        4'd2: seg7out1 = 4'b1011;
        4'd3: seg7out1 = 4'b1001;
        4'd4: seg7out1 = 4'b0100;
        4'd5: seg7out1 = 4'b1101;
        4'd6: seg7out1 = 4'b1111;
        4'd7: seg7out1 = 4'b1100;
        4'd8: seg7out1 = 4'b1111;
        4'd9: seg7out1 = 4'b1100;
        default: seg7out1 = 4'b0000;
        endcase
endfunction
function [2:0] seg7out2 (input [3:0] din);
        case (din)
        4'd0: seg7out2 = 3'b011;
        4'd1: seg7out2 = 3'b011;
        4'd2: seg7out2 = 3'b101;
        4'd3: seg7out2 = 3'b111;
        4'd4: seg7out2 = 3'b111;
        4'd5: seg7out2 = 3'b110;
        4'd6: seg7out2 = 3'b110;
        4'd7: seg7out2 = 3'b011;
        4'd8: seg7out2 = 3'b111;
        4'd9: seg7out2 = 3'b111;
        default: seg7out2 = 3'b000;
        endcase
endfunction
endmodule
```

seg7ledモジュールの入力（num）は0から99までの整数で、seg7ledモジュールの出力（dout1、dout2）はそれぞれ4-bitである。numに99より大きな数字を入れないように注意。

fpga_topモジュール中からこのseg7ledモジュールにアクセスしたい。 汎用入出力ポートAとB（つまり、ioaとiob）に2桁7セグメントLEDがつながっていると仮定するので、hard/top.vの入出力ポート宣言は下記のように修正する。 ここではioaとiobはwire型を仮定しているので、ioaとiobは「output reg」ではなくて「output」にする。

```
module fpga_top (
        input                   clk_125mhz,
        input           [3:0]   sw,
        input           [3:0]   btn,
        output  reg     [3:0]   led,
        output  reg     [10:0]  lcd,
＜中略＞
        //output  reg   [7:0]   ioa,
        //output  reg   [7:0]   iob
        output          [7:0]   ioa,
        output          [7:0]   iob
);
```

次にfpga_topモジュール中にseg7ledモジュールのインスタンスを追加する。seg7ledモジュールに与える数字（num）を記憶するためのregも追加する。 ここでnumはcs4空間に割り当てられているものとする。

```
reg     [6:0]   num;
seg7led seg7led (clk_62p5mhz, reset, num, ioa, iob);
```

cs4でnumに値を書き込めるようにする。

```
/* cs4 */
always @ (posedge clk_62p5mhz or posedge reset)
        if (reset)                      num     <= 0;
        else if (cs4 && memwrite)       num     <= writedata[6:0];
```

以下にソフトウェア側のサンプルコードを示す。 ここではseg7ledモジュールに与えるnumの番地は0xff10としているが、ここは必要に応じて修正して欲しい。

```
void interrupt_handler()
{
        static int cnt = 0;
        volatile int *seg7_ptr = (int *)0xff10;
        cnt++;
        if (cnt % 10 == 0)
                *seg7_ptr = cnt / 10;
}
void main()
{
        for (;;);
}
```

タイマー割込みの頻度を変えたい

hard/top.v中のtimerモジュールのカウンタの最大値を変更すれば良い。 動作周波数は62.5MHz（1クロックは16nsec）である。 デフォルトでは100msec間隔でirqを1にするため、カウンタの最大値は6250000になっている。

```
module timer (
        input                   clk, reset,
        output                  irq
);
reg     [22:0]  counter;

assign  irq = (counter == 23'd6250000);

always @ (posedge clk or posedge reset)
        if (reset)                      counter <= 0;
        else if (counter < 23'd6250000) counter <= counter + 1;
        else                            counter <= 0;
endmodule
```

LCDを2つ使いたい

ここではioaとiobに2つ目のLCDを接続する。fpga_topモジュールのポート名はlcd2とする。

この場合、fpga_topモジュールの汎用入出力ポートioaとiobを削除して、出力ポートlcd2を追加する。 また、hard/fpga.xdc（**新FPGAボード**の場合はfpga_z7.xdc）の該当箇所を以下のように修正する。

```
##Pmod Header JD
set_property PACKAGE_PIN T14 [get_ports {lcd2[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd2[0]}]
set_property PACKAGE_PIN T15 [get_ports {lcd2[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd2[1]}]
set_property PACKAGE_PIN P14 [get_ports {lcd2[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd2[2]}]
set_property PACKAGE_PIN R14 [get_ports {lcd2[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd2[3]}]
set_property PACKAGE_PIN U14 [get_ports {lcd2[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd2[4]}]
set_property PACKAGE_PIN U15 [get_ports {lcd2[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd2[5]}]
set_property PACKAGE_PIN V17 [get_ports {lcd2[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd2[6]}]
set_property PACKAGE_PIN V18 [get_ports {lcd2[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd2[7]}]

##Pmod Header JE
set_property PACKAGE_PIN U17 [get_ports {lcd2[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd2[8]}]
set_property PACKAGE_PIN V13 [get_ports {lcd2[9]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd2[9]}]
set_property PACKAGE_PIN T17 [get_ports {lcd2[10]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd2[10]}]
```

RGB LEDを使いたい（新FPGAボード限定）

新FPGAボードのひとは他と若干の違いがあり恐縮だが、良い点としてはボード上にRGB LEDが2個実装されている。

ポート名はled5とled6。それぞれ3-bit幅で、下位から順にR、G、Bに対応する。3'b000は消灯なので計8種類の色で光らせることができる。

hard/top.vの入出力ポート宣言は下記のように修正する。

```
module fpga_top (
        input                   clk_125mhz,
        input           [3:0]   sw,
        input           [3:0]   btn,
        output  reg     [3:0]   led,
        output  reg     [2:0]   led5,
        output  reg     [2:0]   led6,
＜中略＞
);
```

浮動小数点数を使いたい

float型やdouble型変数のこと。結論から言うと現状では使えない。

本実験で使用しているプロセッサは教育用の簡易版であり、浮動小数点演算ユニットを装備していない。ソフトウェアによる浮動小数点ライブラリをリンクするという代案もあるが、現状ではまだ移植できていない。

付録D: 実験環境の基礎知識

本実験ではFPGAボードとしてDigilent社のZybo Zynq-7000 ARM/FPGA SoCトレーナーボードを使用している。以下は、このボードの公式マニュアルである。

- https://digilent.com/reference/programmable-logic/zybo/reference-manual

本実験で使用するLCDの高度な使い方に興味がある場合は以下のマニュアルを読むと良い。

- https://digilent.com/reference/pmod/pmodclp/reference-manual

16ボタンキーパッドのマニュアルは以下にある。付録C「便利なサンプルコード集」のkypd_scan()関数はこれを見て実装した。

- https://digilent.com/reference/pmod/pmodkypd/reference-manual

ロータリーエンコーダのマニュアルは以下にある。付録C「便利なサンプルコード集」のrotary_encモジュールはこれを見て実装した。

- https://digilent.com/reference/pmod/pmodenc/reference-manual

MIPS R2000互換プロセッサの命令セットについては下記のリンクが参考になる。アセンブリ言語コードを見ながらのデバッグで、もし知らない命令が出てきたら適宜参照しよう。「7 MIPS アセンブリ言語」および「8 コンパイラのレジスタ使用規則」は一度目を通しておくと良い。

- https://ja.wikipedia.org/wiki/MIPSアーキテクチャ

本実験ではXilinx社のZynq Z-7010というFPGAを使っている。Zynq 7000シリーズについては下記のリンクをたどると良い。

- https://japan.xilinx.com/products/silicon-devices/soc/zynq-7000.html

付録E: シミュレーション

実機を用いて動作確認するには、毎回、論理合成、配置配線、FPGAの再構成が必要であり、どうしても時間がかかってしまう。実は、実機を使わずにシミュレータ上でプログラムの動作確認を行うこともできる。ここではその方法を紹介する。

シミュレーションのために必要なファイルは以下の通り。

ディレクトリ名	ファイル名	内容
sim/		Verilog HDLのシミュレーションはsim/以下で行う
	Makefile	makeと打つとシミュレーションを行う

ディレクトリ名	ファイル名	内容
	test.v	シミュレーション用のテストベクタ（シミュレーション内容を記述）

チュートリアル

ここでは、「チュートリアル1」で生成した機械語コードprogram.dat使ってVerilog HDLシミュレータの使い方を紹介する。sim/ディレクトリに移動してmakeと打つと、シミュレーションが開始する。

```
> cd sim
> make
```

```
Clk     0  SWITCH value changed to      0000
Clk     0  BUTTON value changed to      0000
Clk     0  BUTTON value changed to      1111
Clk     0  LCD    value changed to      000_0000_0000
Clk     0  LED    value changed to      0000
Clk     6  BUTTON value changed to      0000
Clk    32  Writing value        x to address 0x0000fefc
Clk    38  Writing value    65284 to address 0x0000fef0
Clk    42  Writing value    65288 to address 0x0000fef4
Clk    54  Writing value        0 to address 0x0000ff08
Clk    56  SWITCH value changed to      0101
Clk    56  BUTTON value changed to      0110
Clk    70  Writing value      101 to address 0x0000ff08
Clk    70  LED    value changed to      0101
Clk    86  Writing value      101 to address 0x0000ff08
Clk   102  Writing value      101 to address 0x0000ff08
Clk   106  SWITCH value changed to      0000
Clk   106  BUTTON value changed to      0000
Clk   118  Writing value        0 to address 0x0000ff08
Clk   118  LED    value changed to      0000
...
```

シミュレーションのテストベクタはsim/test.vである。ここでは、メモリやI/Oへの書き込み（sw命令）を監視していて、書き込みがあればログに表示する。

test.01.cでは、SWITCHの番地（0xff04）から読み込んだ値をLEDの番地（0xff08）に書き込み続けている。ログを見て欲しい。例えば、56サイクル目でSWITCHとBUTTONの番地にそれぞれ0101と0110という値を書き込み、70サイクル目でLEDの値が0101に変化している。

ログとにらめっこすることでプログラムの挙動をある程度把握できるが、この例ではストア命令（sw）以外のイベントを捉えられないので内部の挙動を把握することはできない。余程自信があるときを除き、通常は波形を見ながらデバッグしたほうが効率が良い。以下のようにして波形ビューワ（GTKWave）を立ち上げよう。引数のdump.vcdは信号の変化を記録したファイルであり、シミュレーションを実行する度に生成される。

```
> gtkwave dump.vcd &
```

モジュールおよびファイル構成

Verilog HDLシミュレーション、および、FPGAボードを使った実機動作確認では使用するファイルが若干異なるのでここで整理しておこう。

- まず、hard/以下のtop.v、mips.v、parts.v、multdiv.vは共通。soft/以下のprogram.datも共通である。
- シミュレーションでは、fpga_topモジュール（FPGA上の回路）へのSWITCHやBUTTON操作、および、LEDやLCDの監視はsim/test.vに記述されている通りに実行される。
- 実機動作確認では、fpga_topモジュール（FPGA上の回路）へのSWITCHやBUTTON操作、および、LEDやLCDの監視は人間が行う。

シミュレータ上で動作確認するときは、まずsim/test.vを修正することでゲーム機に対してSWITCHやBUTTON操作を与え、波形を見ながら動作確認を行う。

シミュレータを用いたデバッグの実際

シミュレータ上で動作確認するには、まずsim/test.vを修正してSWITCHやBUTTON操作のイベントを発生させ、正しく動作しているか確かめる。

```
> less sim/test.v
```

```
＜中略＞
`timescale 1ns/1ps
module test;
parameter       STEP = 8;       /* System clock is 125MHz (8nsec) */
＜中略＞
/* Generate clock to sequence tests */
always #(STEP/2)
       clk <= ~clk;
...
```

シミュレーションの時間単位は1nsecとした。STEPには8を代入しているので、4nsecごとにclkの値が反転する。clkの0から1への立ち上がりイベントは8nsec周期で発生するから、動作クロックは125MHzである。ちなみに、hard/top.vを見れば分かる通り、メインメモリ（FPGAの内蔵BlockRAM）には125MHzクロックを与え、mipsプロセッサやそれ以外の周辺回路にはその半分の62.5MHzクロックを与えている。FPGAの内蔵BlockRAMとプロセッサの動作タイミングの都合によりこのような実装にした。

```
initial begin
        clk     <= 0;
        counter <= 0;
        sw      <= 4'b0000;
        btn     <= 4'b0000;
        #(STEP/4);
        btn     <= 4'b1111;             /* Reset ON */
        #(STEP*6);
        btn     <= 4'b0000;             /* Reset OFF */

        #(STEP*50);
        sw      <= 4'b0101;
        btn     <= 4'b0110;
        #(STEP*50);
        sw      <= 4'b0000;
        btn     <= 4'b0000;

        /* Specify the simulation time. Default value is (STEP*1000) nsec.
         * If it is not enough, longer value should be specified. */
        #(STEP*1000);
        $finish;
end
```

上記の記述によって、決められた時間にSWITCH操作を発生させている。具体的には、

- まずreset（ここでは4個のbtn同時押し）してから、6 STEP待ち、resetを解除している。これによってmipsプロセッサを含むすべての周辺回路が初期化され、一斉に動作を再開する。
- さらに50 STEP経った時点で、swに0101、btnに0110という値が書き込まれる。これはスライドスイッチの0-bit目と2-bit目、押しボタンの1-bit目と2-bit目が押されたという意味である。
- さらに50 STEP経った時点で、swとbtnに0が書き込まれる。これは全スイッチがオフになったという意味である。
- その後、1,000 STEP進めてからシミュレーションをfinishしている。もしシミュレーション時間が1,000 STEPでは足りない場合は「#(STEP*1000);」の値をもっとずっと大きくすれば良い。

この挙動はシミュレーション結果のログと一致するはずである。チュートリアル1のシミュレーション結果と見比べて欲しい。

```
> cd sim
> make
```

```
Clk     0  SWITCH value changed to      0000
Clk     0  BUTTON value changed to      0000
＜中略＞
Clk    56  SWITCH value changed to      0101
Clk    56  BUTTON value changed to      0110
```

```
＜中略＞
Clk   106  SWITCH value changed to      0000
Clk   106  BUTTON value changed to      0000
...
```

開発中のゲームにおいて、例えば、1つ目の押しボタンを「スタートボタン」として使うとする。シミュレーションによる動作確認のために「100 STEP目にスタートボタンが押され、150 STEP目に離される」などのイベントを発生させたくなると思う。このような場合はtest.vにSWITCHの操作イベントを追記していけば良い。

デバッグの役に立つように、メインメモリ、SWITCH、BUTTON、LED、LCDの値が変わるとシミュレーション時にログが出るようにしている。詳しくは以下のdisplay文を参照。

```
/* Monitoring memory write operations */
always @ (posedge fpga.clk_62p5mhz)
        if (fpga.memwrite) begin
                $display("Clk %5d  Writing value %8d to address 0x%h",
                                counter, fpga.writedata, fpga.dataadr);
        end
/* Monitoring Switches */
always @ (sw)
        $display("Clk %5d  SWITCH value changed to\t%4b", counter, sw);
/* Monitoring Buttons */
always @ (btn)
        $display("Clk %5d  BUTTON value changed to\t%4b", counter, btn);
/* Monitoring LED */
always @ (led)
        $display("Clk %5d  LED    value changed to\t%4b", counter, led);
/* Monitoring LCD */
always @ (lcd)
        $display("Clk %5d  LCD    value changed to\t%3b_%4b_%4b", counter,
                        lcd[10:8], lcd[7:4], lcd[3:0]);
```

このような「printfデバッグ」はお手軽ではあるが、外部への書き込み以外の挙動はまったく見えないのでとんだ勘違いを起こすことがある。波形を見ながらデバッグするほうが良いだろう。

付録F: 環境構築

本実験で使用するツールは、MIPS用クロス開発ツール（gcc、binutils）、FPGA用合成・配置配線ツール（Xilinx Vivado）などである。シミュレーションを行うにはVerilog HDLシミュレータ（Icarus Verilog）、波形ビューワ（GTKWave）も必要になる。

これらはすでにITCで利用可能であるため、新しくインストールする必要はない。参考までにクロス開発環境のインストール方法を以下に書き記す。

クロス開発環境の構築

MIPS I命令セットアーキテクチャ用のCコンパイラ（gcc）、バイナリツール（as、ld、objdump）をインストールする。バイトオーダはbig endianとする。 この例では、インストール先は/home/md401/aa205875/usr/としている。ここの部分は各自の環境に合わせて適宜修正すること。

まずは、バイナリツール（binutils）をインストールする。

```
> wget https://ftp.gnu.org/gnu/binutils/binutils-2.30.tar.gz
> tar zxvf binutils-2.30.tar.gz
> cd binutils-2.30
> mkdir build_mips && cd build_mips
> ../configure --target=mips --prefix=/home/md401/aa205875/usr
> make
> make install
```

次に任意精度数演算ライブラリ（gmp）をインストールする。これはCコンパイラ（gcc）をコンパイルするために必要である。

```
> wget https://ftp.gnu.org/gnu/gmp/gmp-4.3.2.tar.gz
> tar zxvf gmp-4.3.2.tar.gz
> cd gmp-4.3.2
> ./configure --prefix=/home/md401/aa205875/usr
```

```
> make
> make install
```

高品質多倍長浮動小数点ライブラリ（mpfr）をインストールする。こちらもgccをコンパイルするために必要である。

```
> wget https://ftp.gnu.org/gnu/mpfr/mpfr-3.1.2.tar.gz
> tar zxvf mpfr-3.1.2.tar.gz
> ./configure --prefix=/home/md401/aa205875/usr \
        --with-gmp=/home/md401/aa205875/usr
> make
> make install
```

複素数ライブラリ（mpc）をインストールする。こちらもgccをコンパイルするために必要である。

```
> wget https://ftp.gnu.org/gnu/mpc/mpc-1.0.1.tar.gz
> tar zxvf mpc-1.0.1.tar.gz
> ./configure --prefix=/home/md401/aa205875/usr \
        --with-gmp=/home/md401/aa205875/usr \
        --with-mpfr=/home/md401/aa205875/usr
> make
> make install
```

ここで、やっとCコンパイラ（gcc）のインストールに入る。configureスクリプトの引数としてgmp、mpfr、mpcライブラリのパスを指定する必要がある。マシンによってはコンパイルには1時間以上かかる。

```
> wget https://ftp.gnu.org/gnu/gcc/gcc-7.4.0/gcc-7.4.0.tar.gz
> tar zxvf gcc-7.4.0.tar.gz
> cd gcc-7.4.0
> mkdir build_mips && cd build_mips
> ../configure --target=mips --without-fp --enable-languages="c" \
       --disable-libssp --prefix=/home/md401/aa205875/usr \
        --with-gmp=/home/md401/aa205875/usr \
        --with-mpfr=/home/md401/aa205875/usr \
        --with-mpc=/home/md401/aa205875/usr
> make -j4
> make install
```

環境変数の設定

開発ツールを使えるようにするために、シェルの設定ファイル（.bashrc）を修正し、環境変数PATHを設定する。

```
> emacs .bashrc
＜末尾に以下を追加＞
export PATH=/home/md401/aa205875/usr/bin/:${PATH}
> source .bashrc
```