```
/*
 * Copyright (c) 2009 Xilinx, Inc.  All rights reserved.
 *
 * Xilinx, Inc.
 * XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS" AS A
 * COURTESY TO YOU.  BY PROVIDING THIS DESIGN, CODE, OR INFORMATION AS
 * ONE POSSIBLE   IMPLEMENTATION OF THIS FEATURE, APPLICATION OR
 * STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS IMPLEMENTATION
 * IS FREE FROM ANY CLAIMS OF INFRINGEMENT, AND YOU ARE RESPONSIBLE
 * FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE FOR YOUR IMPLEMENTATION.
 * XILINX EXPRESSLY DISCLAIMS ANY WARRANTY WHATSOEVER WITH RESPECT TO
 * THE ADEQUACY OF THE IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO
 * ANY WARRANTIES OR REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE
 * FROM CLAIMS OF INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY
 * AND FITNESS FOR A PARTICULAR PURPOSE.
 *
 */

#include "xgpio.h"          // Provides access to PB GPIO driver.
#include "xintc_l.h"        // Provides handy macros for the interrupt controller.
#include <stdio.h>
#include "platform.h"
#include "xparameters.h"
#include "xaxivdma.h"
#include "xio.h"
#include "time.h"
#include "unistd.h"
#include "aliens.h"
#include "screenState.h"
#include "sounds.h"
#include "thePITS.h"
#include <stdlib.h>
#include <string.h>
#include "nes.h"

#define DEBUG

#define DEBOUNCE_TICKS 5//Number of ticks to debounce for
#define DRAW_TIMER 20//Number of ticks to debounce for
#define FRAME_BUFFER_0_ADDR 0xC1000000  // Starting location in DDR where we will store
the images that we lay.
#define AUTO_INCREMENT_DELAY_TICKS 50//Ticks between auto increments when holding button
down
#define PENULTIMATE_DEBOUNCE_COUNT 1//The second to last count of the debounce counter

#define SHOOT_BUTTON_MASK 0x10//Mask for inc button
#define RIGHT_BUTTON_BIT 0x02//Value of the seconds bit
#define LEFT_BUTTON_BIT 0x08//Value of the hours bit
#define BUFFER_SIZE 100 //Buffer for getting chars from uart
#define LOWEST_VALUE 10000 //Lowest possible count down for PIT

#define VALUE_TO_SHIFT 1 //Value that is shifted assign and find values for the buttons
#define NES_CONTROLLER_LEFT 6 //Number of bits shifted to get the left button status
#define NES_CONTROLLER_RIGHT 7 // Number of bits shifted to get the right button status

#define GAME_TANK_LEFT 3 // Number of bits shifted to tell the tank to move left
#define GAME_TANK_RIGHT 1 // Number of bits shifted to tell the tank to move right
#define GAME_TANK_FIRE 4 // Number of bits shifted to tell the tank to fire
```

```
XGpio gpLED; // This is a handle for the LED GPIO block.
XGpio gpPB; // This is a handle for the push-button GPIO block.

static uint32_t currentButtonState = 0;//Current state of the buttons
static uint32_t currentButtonStateDebounced = 0;//Current state of the buttons after
debouncing

static uint8_t debounceCount = DEBOUNCE_TICKS;//Countdown before button input is
considered debounced

void print(char *str);

void pit_interrupt_handler() {
    screenState_update(currentButtonStateDebounced);

    //If the debounce counter is running
    if(debounceCount != 0)
    {
        //If the denounce counter is on the last count.
        if(debounceCount == PENULTIMATE_DEBOUNCE_COUNT)
        {
            //currentButtonStateDebounced = currentButtonState;//Set the switched debounce
 state
            //Note: we only do this once per button press (which is why it is only done
on the last count of the debounce count)
        }
        //Otherwise just decrement the debounce counter
        debounceCount--;
    }
}
uint32_t button = 0;
void nes_interrupt_handler() {
    button = nes_readButtons();
    //191 is left on NES controller
    //127 is right on NES controller
    //254 is fire on NES controller

    //2 is left in game
    //8 is right in game
    //16 is fire in game
    currentButtonStateDebounced = 0;
    if(!(button & (VALUE_TO_SHIFT << NES_CONTROLLER_LEFT))){ //left
        //Sets the bit that makes the tank move to the left
        currentButtonStateDebounced = currentButtonStateDebounced | VALUE_TO_SHIFT <<
GAME_TANK_LEFT;
    }
    if(!(button & (VALUE_TO_SHIFT << NES_CONTROLLER_RIGHT))){ //right
        //Sets the bit that makes the tank move to the right
        currentButtonStateDebounced = currentButtonStateDebounced | VALUE_TO_SHIFT <<
GAME_TANK_RIGHT;
    }
    if(!(button & (VALUE_TO_SHIFT))){ //fire
        //Sets the bit that makes the tank fire
        currentButtonStateDebounced = currentButtonStateDebounced | VALUE_TO_SHIFT <<
GAME_TANK_FIRE;
    }
}
```

```
// This is invoked each time there is a change in the button state (result of a push or a
bounce).
void pb_interrupt_handler() {

    // Clear the GPIO interrupt.
    XGpio_InterruptGlobalDisable(&gpPB); // Turn off all PB interrupts for now.
    currentButtonState = XGpio_DiscreteRead(&gpPB, 1); // Get the current state of the
buttons.

    debounceCount = DEBOUNCE_TICKS;//Set the debounce counter

    XGpio_InterruptClear(&gpPB, 0xFFFFFFFF); // Ack the PB interrupt.
    XGpio_InterruptGlobalEnable(&gpPB); // Re-enable PB interrupts.
}

uint32_t ticks = 0;
// Main interrupt handler, queries the interrupt controller to see what peripheral
// fired the interrupt and then dispatches the corresponding interrupt handler.
// This routine acks the interrupt at the controller level but the peripheral
// interrupt must be ack'd by the dispatched interrupt handler.
// Question: Why is the timer_interrupt_handler() called after ack'ing the interrupt
controller
// but pb_interrupt_handler() is called before ack'ing the interrupt controller?
void interrupt_handler_dispatcher(void* ptr) {
    ticks++;
    int intc_status = XIntc_GetIntrStatus(XPAR_INTC_0_BASEADDR);
    if(ticks > 1000){

    }

    // Check the NES buttons.
    if (intc_status & XPAR_NES_CONTROLLER_0_IP2BUS_INTERRUPT_MASK) {
        nes_interrupt_handler();
        XIntc_AckIntr(XPAR_INTC_0_BASEADDR,
                XPAR_NES_CONTROLLER_0_IP2BUS_INTERRUPT_MASK);
    }

    /*
    // Check the FIT interrupt first.
    if (intc_status & XPAR_FIT_TIMER_0_INTERRUPT_MASK) {
        XIntc_AckIntr(XPAR_INTC_0_BASEADDR, XPAR_FIT_TIMER_0_INTERRUPT_MASK);
        timer_interrupt_handler();
    }
    */

    // Check the PIT interrupt first.
    if (intc_status & XPAR_PIT_0_IP2BUS_INTERRUPT_MASK) {
        XIntc_AckIntr(XPAR_INTC_0_BASEADDR, XPAR_PIT_0_IP2BUS_INTERRUPT_MASK);
        pit_interrupt_handler();
    }



    // Check the push buttons.
    if (intc_status & XPAR_PUSH_BUTTONS_5BITS_IP2INTC_IRPT_MASK) {
        pb_interrupt_handler();
        XIntc_AckIntr(XPAR_INTC_0_BASEADDR,
                XPAR_PUSH_BUTTONS_5BITS_IP2INTC_IRPT_MASK);
    }
```

```
    // Check AC97 Interrupts
    if (intc_status & XPAR_AXI_AC97_0_INTERRUPT_MASK) {
        sounds_ac97_interrupt_handler();
        XIntc_AckIntr(XPAR_INTC_0_BASEADDR,
                XPAR_AXI_AC97_0_INTERRUPT_MASK);
    }
}

int main() {
    init_platform(); // Necessary for all programs.
    thePITS_init();
    int Status; // Keep track of success/failure of system function calls.
    XAxiVdma videoDMAController;
    // There are 3 steps to initializing the vdma driver and IP.
    // Step 1: lookup the memory structure that is used to access the vdma driver.
    XAxiVdma_Config * VideoDMAConfig = XAxiVdma_LookupConfig(
            XPAR_AXI_VDMA_0_DEVICE_ID);
    // Step 2: Initialize the memory structure and the hardware.
    if (XST_FAILURE == XAxiVdma_CfgInitialize(&videoDMAController,
            VideoDMAConfig, XPAR_AXI_VDMA_0_BASEADDR)) {
        xil_printf("VideoDMA Did not initialize.\r\n");
    }
    // Step 3: (optional) set the frame store number.
    if (XST_FAILURE == XAxiVdma_SetFrmStore(&videoDMAController, 2,
            XAXIVDMA_READ)) {
        xil_printf("Set Frame Store Failed.");
    }
    // Initialization is complete at this point.

    // Setup the frame counter. We want two read frames. We don't need any write frames
but the
    // function generates an error if you set the write frame count to 0. We set it to 2
    // but ignore it because we don't need a write channel at all.
    XAxiVdma_FrameCounter myFrameConfig;
    myFrameConfig.ReadFrameCount = 2;
    myFrameConfig.ReadDelayTimerCount = 10;
    myFrameConfig.WriteFrameCount = 2;
    myFrameConfig.WriteDelayTimerCount = 10;
    Status = XAxiVdma_SetFrameCounter(&videoDMAController, &myFrameConfig);
    if (Status != XST_SUCCESS) {
        xil_printf("Set frame counter failed %d\r\n", Status);
        if (Status == XST_VDMA_MISMATCH_ERROR)
            xil_printf("DMA Mismatch Error\r\n");
    }
    // Now we tell the driver about the geometry of our frame buffer and a few other
things.
    // Our image is 480 x 640.
    XAxiVdma_DmaSetup myFrameBuffer;
    myFrameBuffer.VertSizeInput = 480; // 480 vertical pixels.
    myFrameBuffer.HoriSizeInput = 640 * 4; // 640 horizontal (32-bit pixels).
    myFrameBuffer.Stride = 640 * 4; // Dont' worry about the rest of the values.
    myFrameBuffer.FrameDelay = 0;
    myFrameBuffer.EnableCircularBuf = 1;
    myFrameBuffer.EnableSync = 0;
    myFrameBuffer.PointNum = 0;
```

```
    myFrameBuffer.EnableFrameCounter = 0;
    myFrameBuffer.FixedFrameStoreAddr = 0;
    if (XST_FAILURE == XAxiVdma_DmaConfig(&videoDMAController, XAXIVDMA_READ,
            &myFrameBuffer)) {
        xil_printf("DMA Config Failed\r\n");
    }
    // We need to give the frame buffer pointers to the memory that it will use. This
memory
    // is where you will write your video data. The vdma IP/driver then streams it to the
HDMI
    // IP.
    myFrameBuffer.FrameStoreStartAddr[0] = FRAME_BUFFER_0_ADDR;
    myFrameBuffer.FrameStoreStartAddr[1] = FRAME_BUFFER_0_ADDR + 4 * 640 * 480;

    if (XST_FAILURE == XAxiVdma_DmaSetBufferAddr(&videoDMAController,
            XAXIVDMA_READ, myFrameBuffer.FrameStoreStartAddr)) {
        xil_printf("DMA Set Address Failed Failed\r\n");
    }
    // Print a sanity message if you get this far.
    xil_printf("Woohoo! I made it through initialization.\n\r");

    // Now, let's get ready to start laying some stuff on the screen.
    // The variables framePointer and framePointer1 are just pointers to the base address
    // of frame 0 and frame 1.
    unsigned int * framePointer0 = (unsigned int *) FRAME_BUFFER_0_ADDR;

    screenState_setFramePointer(framePointer0);

    // This tells the HDMI controller the resolution of your lay (there must be a better
way to do this).
    XIo_Out32(XPAR_AXI_HDMI_0_BASEADDR, 640*480);

    // Start the DMA for the read channel only.
    if (XST_FAILURE == XAxiVdma_DmaStart(&videoDMAController, XAXIVDMA_READ)) {
        xil_printf("DMA START FAILED\r\n");
    }
    int frameIndex = 0;
    // We have two frames, let's park on frame 0. Use frameIndex to index them.
    // Note that you have to start the DMA process before parking on a frame.
    if (XST_FAILURE == XAxiVdma_StartParking(&videoDMAController, frameIndex,
            XAXIVDMA_READ)) {
        xil_printf("vdma parking failed\n\r");
    }
    sounds_init();
    screenState_init();
    // Initialize the GPIO peripherals.
    int success;
    success = XGpio_Initialize(&gpPB, XPAR_PUSH_BUTTONS_5BITS_DEVICE_ID);
    // Set the push button peripheral to be inputs.
    XGpio_SetDataDirection(&gpPB, 1, 0x0000001F);
    // Enable the global GPIO interrupt for push buttons.
    XGpio_InterruptGlobalEnable(&gpPB);
    // Enable all interrupts in the push button peripheral.
    XGpio_InterruptEnable(&gpPB, 0xFFFFFFFF);

    microblaze_register_handler(interrupt_handler_dispatcher, NULL);
    XIntc_EnableIntr(
            XPAR_INTC_0_BASEADDR,
            (XPAR_PIT_0_IP2BUS_INTERRUPT_MASK
```

```
                    | XPAR_PUSH_BUTTONS_5BITS_IP2INTC_IRPT_MASK
                        | XPAR_AXI_AC97_0_INTERRUPT_MASK
                            | XPAR_NES_CONTROLLER_0_IP2BUS_INTERRUPT_MASK));
    XIntc_MasterEnable(XPAR_INTC_0_BASEADDR);
    microblaze_enable_interrupts();
    while (1){
        xil_printf("Enter new count down from value in ticks (100,000, 000 is one
second)\r\nPress ENTER to write to register\r\n");
        char array[BUFFER_SIZE];//Buffer for uart input
        char input;//Input from uart
        uint32_t i;//For for loop
        uint8_t problem = 0;//Flag for when the user puts in a non number
        for(i = 0; i < BUFFER_SIZE; i++){//For each buffer value
            input = getchar();//Get input from uart

            if(input == '\r' || i == BUFFER_SIZE){//If the enter key
                array[i] = '\0';//Null terminate buffer
                break;
            }
            if(input < '0' || input > '9'){//If not a number
                xil_printf("Only enter digits (0 to 9), please try again\r\n");
                problem = 1;//Raise flag so we don't try to write value to pit
                break;
            }
            array[i] = input;//Otherwise store the number in buffer
        }
        if(!problem){//If data is valid
            int valueToWrite = atoi(array);//char * to int
            if(valueToWrite < LOWEST_VALUE){//Don't let the value go to low (will never
yield to idle task if tick rate is to high
                valueToWrite = LOWEST_VALUE;
            }
            thePITS_setTicksCountDownFrom(valueToWrite);//Write new count down to control
register in pit
        }
    }
    cleanup_platform();
    return 0;
}
```