Lab 6 - Taylor Cowley and Andrew Okazaki

- Programmable Interrupt Timer (PIT)
    - Register Descriptions
    - Timing Diagrams
    - Driver API
    - Bug Report

Programmable Interrupt Timer (PIT)

Register Descriptions

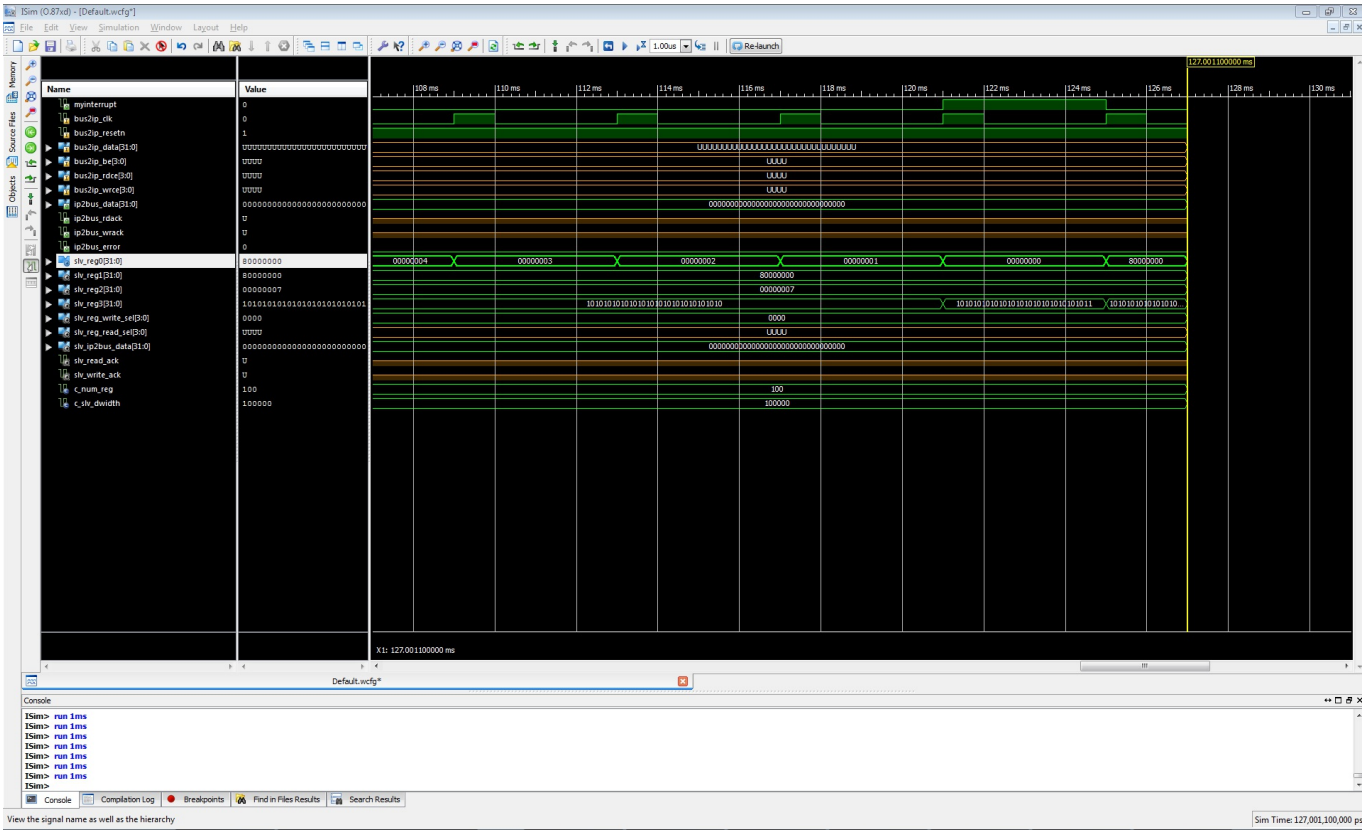
I our PIT we used four different registers named, slv_reg0, slv_reg1, slv_reg2, slv_reg3.

Register Name	Register Address (in xparameters.h)	Purpose
slv_reg0	0x7bc00000	Used as a 32 bit counter
slv_reg1	0x7bc00004	Contains the delay number
slv_reg2	0x7bc00008	The control register, if bit_0 = 1 allow it to decrement else do not. If bit_1 = 1 then show interrupts else do not show interrupts.,If bit_2 = 1 allow the value to reload once it hits 0 else do not reload.
slv_reg3	0x7bc00010	Bit 0 contains output to signal an

		interrupt. Bits 31-1 are a pre-set value used for debugging

Timing Diagram

This is a simulation of our PIT timer. Note at the top, because the count register has reached 1, and the control register's 1 bit is on, an interrupt is generated for one clock cycle



Driver API

The api was automatically generated in `pit.h` by the EDK. The important functions are listed below. BaseAddress should always be `XPAR_PIT_0_BASEADDR` (generated in `xparameters.h`), and the RegOffset should always be `0`.

```
// Used to write a value to reg0, the count register
#define PIT_mWriteSlaveReg0(BaseAddress, RegOffset, Value
```

```
) \
    Xil_Out32((BaseAddress) + (PIT_SLV_REG0_OFFSET) + (Re
gOffset), (Xuint32)(Value))

// Used to write a value to reg1, the reload register. A
higher number makes the PIT generate interrupts slower
#define PIT_mWriteSlaveReg1(BaseAddress, RegOffset, Value
) \
    Xil_Out32((BaseAddress) + (PIT_SLV_REG1_OFFSET) + (Re
gOffset), (Xuint32)(Value))

// Used to write a value to reg2, the control register. T
he bits that matter are listed in the above table
#define PIT_mWriteSlaveReg2(BaseAddress, RegOffset, Value
) \
    Xil_Out32((BaseAddress) + (PIT_SLV_REG2_OFFSET) + (Re
gOffset), (Xuint32)(Value))

// Used to write a value to reg3. Dangerous. Do not use.
#define PIT_mWriteSlaveReg3(BaseAddress, RegOffset, Value
) \
    Xil_Out32((BaseAddress) + (PIT_SLV_REG3_OFFSET) + (Re
gOffset), (Xuint32)(Value))


// The following are to read from the registers.
#define PIT_mReadSlaveReg0(BaseAddress, RegOffset) \
    Xil_In32((BaseAddress) + (PIT_SLV_REG0_OFFSET) + (Reg
```

```
Offset))
#define PIT_mReadSlaveReg1(BaseAddress, RegOffset) \
    Xil_In32((BaseAddress) + (PIT_SLV_REG1_OFFSET) + (Reg
Offset))
#define PIT_mReadSlaveReg2(BaseAddress, RegOffset) \
    Xil_In32((BaseAddress) + (PIT_SLV_REG2_OFFSET) + (Reg
Offset))
#define PIT_mReadSlaveReg3(BaseAddress, RegOffset) \
    Xil_In32((BaseAddress) + (PIT_SLV_REG3_OFFSET) + (Reg
Offset))
```

Bug Report

In our code we had a hard time with hardware often we would not include hardware that was needed. Often we did not know why or when some of the hardware modules were dropped from of our code. We ended up having to generate our hardware multiple times while trying to pass this lab off. The second error that we ran into was that our PIT interrupt timer was not generating an interrupt. This was a difficult problem but after simulating we were able to update the logic so that the pit would register a high value after a given amount of time. After knowing that the PIT timer was working correctly using it in our code took a while to figure out. But after using the generated C functions we were able to write and read checking that everything was working properly.

```c
/*
 * helloworld.c: simple test application
 * Currently used to test lab 3 for Space Invaders.
 * Taylor Cowley and Andrew Okazaki
 */

#include <stdio.h>
#include <stdint.h>
#include "platform.h"
#include "xparameters.h"
#include "xaxivdma.h"
#include "xio.h"
#include "time.h"
#include "unistd.h"
#include "tank.h"
#include "interface.h"
#include "aliens.h"
#include "bunkers.h"
#include "mother_ship.h"
#include "util.h"
#include "sound/xac97_l.h"
#include "sound/sound.h"
#include "xgpio.h"
#include "mb_interface.h"
#include "xintc_l.h"
#include "sound/sound.h"
#include "pit.h"



#define DEBUG

#define SCREEN_RES_X 640    // Our screen resolution is 640 * 480
#define SCREEN_RES_Y 480    // Our screen resolution is 640 * 480
#define BLACK 0x00000000    // Hex value for black
#define BLUE  0x2222FF

#define ONE_SECOND  100     // 100 ticks in a second
#define HALF_SECOND 50      // 50 ticks in half a second
#define QUARTER_SECOND 25   // 25 ticks in a quarter second
#define EIGHTH_SECOND 12    // 12 ticks in an eigth second
#define TENTH_SECOND 10     // 10 ticks in a tenth second
#define TWENTIETH_SECOND 5  // 5 ticks in a twentieth second
#define SUPER_FAST 2        // super fast

#define MOTHER_SHIP_SPEED TENTH_SECOND      // Mother ship moves slowly
#define MOTHER_SHIP_SPAWN_CONSTANT 1000     // Mother ship spawns infrequently
#define ALIEN_SHOT_SPAWN_CONSTANT 100       // Aliens shoot frequently
#define ALIEN_MOVE_SPEED HALF_SECOND        // aliens move very slowly

#define BUTTON_UP       0x4 // Constants for button masks
#define BUTTON_DOWN     0x10
#define BUTTON_LEFT     0x8
#define BUTTON_RIGHT    0x2
#define BUTTON_CENTER   0x1

// All speed modifiers for the PIT timer
#define PIT_TENTH_SPEED 10000000            // This is really slow. like paused
```

```
#define PIT_FIFTH_SPEED 5000000             //
#define PIT_THREEFOURTHS_SPEED 7500000      //
#define PIT_NORMAL_SPEED 1000000            // The speed the FIT was running at
#define PIT_1_5x_speed 750000               //
#define PIT_2x_speed 500000                 //
#define PIT_4x_speed 250000                 //
#define PIT_10x_speed 100000                // 10x speed
#define PIT_100x_speed 10000                // 100x speed
#define PIT_LUDICROUS_SPEED PIT_100x_speed  // (same as 100x speed)

#define PIT_NO_OFFSET 0                     // For writing to registers
#define PIT_CONTROL_RUN 0x00000007          // Control value to make it RUN
#define PIT_GOOD_INITIAL_VALUE 100000000    // initial value for counter


void print(char *str);          // print exists!

#define FRAME_BUFFER_0_ADDR 0xC1000000  // Starting location in DDR

//-----------------------------
void timer_interrupt_handler();             // interrupt handler for timer
void pb_interrupt_handler();                // interrupt handler for buttons
void interrupt_handler_dispatcher(void *);  // dispatch the interrupts
void init_pit();                           // Init our pit registers to good values
void change_speed_on_input();
//-----------------------------


XGpio gpLED;  // This is a handle for the LED GPIO block.
XGpio gpPB;   // This is a handle for the push-button GPIO block.
uint32_t* framePointer0 = (uint32_t*) FRAME_BUFFER_0_ADDR;
int32_t currentButtonState;     // Current button being pressed
int32_t mother_ship_points;
uint32_t cpu_usage_timer = 0;
uint32_t sound_count = 0;


void timer_interrupt_handler(){
    static uint32_t timerCount;                 // Timer for timing
    static uint32_t mother_ship_move_counter;   // Timer for mother ship

    tank_update_bullet(framePointer0);          // update all bullets
    aliens_update_bullets(framePointer0);       // update all bullets

    timerCount++;                               // Increment all counters
    mother_ship_move_counter++;
    mother_ship_points++;

    int32_t r = rand();
    if(r%ALIEN_SHOT_SPAWN_CONSTANT == 0){
        alien_missle(framePointer0);    // Make the aliens fire
    }
    if(r%MOTHER_SHIP_SPAWN_CONSTANT == 0){
        mother_ship_spawn();            // mother ship spawns!
    }
    if(mother_ship_move_counter >= MOTHER_SHIP_SPEED){  // MS moves
        mother_ship_move_counter = 0;
        mother_ship_move();
```

```
    }
    if(mother_ship_points > TENTH_SECOND){
        mother_ship_points = 0;         // Mother ship points will display
        mother_ship_points_blink();
    }
    if(timerCount >= HALF_SECOND ){
        timerCount = 0;
        aliens_move(framePointer0); // move the aliens
    }


    // Now to check the buttons.
    if(currentButtonState & BUTTON_LEFT){
        tank_move_left(framePointer0);      // Moving the tank left
    }
    if(currentButtonState & BUTTON_RIGHT){
        tank_move_right(framePointer0);     // Moving the tank right
    }
    if(currentButtonState & BUTTON_CENTER){
        tank_fire(framePointer0);           // Fire the tank!
    }
    if(currentButtonState & BUTTON_UP){     // Not functional yet
        sound_vol_up();
    }
    if(currentButtonState & BUTTON_DOWN){       // Not functional yet
        sound_vol_down();
    }

}

// Interrupt handler for the push buttons
void pb_interrupt_handler(){
    XGpio_InterruptGlobalDisable(&gpPB);    // Can't be interrupted by buttons

    currentButtonState = XGpio_DiscreteRead(&gpPB, 1);
    // Time to clear the interrupt and reenable GPIO interrupts
    XGpio_InterruptClear(&gpPB, 0xFFFFFFFF);
    XGpio_InterruptGlobalEnable(&gpPB);
}

// We are making sound here :)
void sound_interrupt_handler(){
// Making sound!
    sound_run();
}

// Main interrupt handler, queries 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 timer_interrupt_handler() called after ack'ing controller
// but pb_interrupt_handler() is called before ack'ing the interrupt controller?
void interrupt_handler_dispatcher(void* ptr) {
    int intc_status = XIntc_GetIntrStatus(XPAR_INTC_0_BASEADDR);
    // Check the FIT interrupt first.
    if (intc_status & XPAR_PIT_0_MYINTERRUPT_MASK){
        XIntc_AckIntr(XPAR_INTC_0_BASEADDR, XPAR_PIT_0_MYINTERRUPT_MASK);
        timer_interrupt_handler();  // It was a timer interrupt! call that fn
```

```
    }
    // Check the push buttons.
    if (intc_status & XPAR_PUSH_BUTTONS_5BITS_IP2INTC_IRPT_MASK){
        pb_interrupt_handler();     // It was a button interrupt!
        XIntc_AckIntr(XPAR_INTC_0_BASEADDR, // Acknowledge the interrupt
                XPAR_PUSH_BUTTONS_5BITS_IP2INTC_IRPT_MASK);
    }
     // Check the sound card
    if (intc_status & XPAR_AXI_AC97_0_INTERRUPT_MASK){
    // Acknowledge that interrupt
    XIntc_AckIntr(XPAR_INTC_0_BASEADDR, XPAR_AXI_AC97_0_INTERRUPT_MASK);
    sound_interrupt_handler();  // Make sound!
    }
}



// Initializes our PIT with proper values in its registers
void init_pit(){
    // Set up our count register with a good initial value
    PIT_mWriteSlaveReg0 (XPAR_PIT_0_BASEADDR, PIT_NO_OFFSET, PIT_GOOD_INITIAL_VALUE);

    // Set up our reload register with normal value: 100x a second, same as fit
    PIT_mWriteSlaveReg1 (XPAR_PIT_0_BASEADDR, PIT_NO_OFFSET, PIT_NORMAL_SPEED); // 100x a
second

    // Put value in control register to enable interrupts, reload, and count
    PIT_mWriteSlaveReg2 (XPAR_PIT_0_BASEADDR, PIT_NO_OFFSET, PIT_CONTROL_RUN);

}

void init_interrupts(void){
    int32_t success;
    init_pit();
    print("\n\rHello . Let's have a fun \e[31m\e[1mtime \e[21m\e[0m\n\r");
    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);
    // Register the interrupt handler
    microblaze_register_handler(interrupt_handler_dispatcher, NULL);
    // And enable interrupts
     XIntc_EnableIntr(XPAR_INTC_0_BASEADDR, // interrupts to enable
    (XPAR_PIT_0_MYINTERRUPT_MASK |  // fit timer
            XPAR_PUSH_BUTTONS_5BITS_IP2INTC_IRPT_MASK | // buttons
                XPAR_AXI_AC97_0_INTERRUPT_MASK));   // sound card
    // Master the enable
    XIntc_MasterEnable(XPAR_INTC_0_BASEADDR);
    // And enable again
    microblaze_enable_interrupts();
}

// This changes the speed of the pit timer, and hence the game, based
// on a key input of 0-9
// THis is done by changing the value of the timer's reload register.
```

```
void change_speed_on_input(){
    char input = getchar();
    switch (input){ // And change the speed based on input
    case '1':
        PIT_mWriteSlaveReg1 (XPAR_PIT_0_BASEADDR, PIT_NO_OFFSET, PIT_TENTH_SPEED);  //
10x a second
        break;
    case '2':
        PIT_mWriteSlaveReg1 (XPAR_PIT_0_BASEADDR, PIT_NO_OFFSET, PIT_FIFTH_SPEED);  //
50x a second
        break;
    case '3':
        PIT_mWriteSlaveReg1 (XPAR_PIT_0_BASEADDR, PIT_NO_OFFSET,
PIT_THREEFOURTHS_SPEED);    // 75x a second
        break;
    case '4':
        PIT_mWriteSlaveReg1 (XPAR_PIT_0_BASEADDR, PIT_NO_OFFSET, PIT_NORMAL_SPEED); // 1x
speed
        break;
    case '5':
        PIT_mWriteSlaveReg1 (XPAR_PIT_0_BASEADDR, PIT_NO_OFFSET, PIT_1_5x_speed);   // 1.5
 speed
        break;
    case '6':
        PIT_mWriteSlaveReg1 (XPAR_PIT_0_BASEADDR, PIT_NO_OFFSET, PIT_2x_speed); // 2x
speed
        break;
    case '7':
        PIT_mWriteSlaveReg1 (XPAR_PIT_0_BASEADDR, PIT_NO_OFFSET, PIT_4x_speed); // 4x
speed
        break;
    case '8':
        PIT_mWriteSlaveReg1 (XPAR_PIT_0_BASEADDR, PIT_NO_OFFSET, PIT_10x_speed);    //
10x speed
        break;
    case '9':
        PIT_mWriteSlaveReg1 (XPAR_PIT_0_BASEADDR, PIT_NO_OFFSET,
PIT_LUDICROUS_SPEED);   // LUDICROUS SPEED
        break;
    case '0':
        PIT_mWriteSlaveReg1 (XPAR_PIT_0_BASEADDR, PIT_NO_OFFSET, PIT_NORMAL_SPEED); //
100x a second NORMAL SPEED
        break;
    }
}

int main() {
    sound_init_AC_97();
    init_platform();                    // Necessary for all programs.
    init_interrupts();
    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 displaying some stuff on the screen.
    // The variables framePointer and framePointer1 are just pointers to the base address
    // of frame 0 and frame 1.
```

```
    uint32_t* framePointer0 = (uint32_t*) FRAME_BUFFER_0_ADDR;
    // Just paint some large red, green, blue, and white squares in different
    // positions of the image for each frame in the buffer (framePointer0 and
framePointer1).
    int row=0, col=0;
    for( row=0; row<SCREEN_RES_Y; row++) {
        for(col=0; col<SCREEN_RES_X; col++) {
            framePointer0[row*SCREEN_RES_X + col] = BLACK;
        }
    }

    bunkers_init(framePointer0);            // Init the bunkers
    tank_init();                            // initialize the tank
    tank_draw(framePointer0, false);        // draw the tank
    interface_init_board(framePointer0);    // draw the tanks at the top
    aliens_init(framePointer0);             // initialize aliens
    mother_ship_init(framePointer0);        // Init the mother ship

    // This tells the HDMI controller the resolution of your display (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");
    }

    // ----------------------------------------------------------------
    // Required, or the whole program halts for an unidentified reason
    srand((unsigned)time( NULL ));
    // Why is this necessary?
    // ----------------------------------------------------------------


    while(1){
        //      cpu_usage_timer++;
        // Now we wait for input. You can input 0-9, with varying speeds for the
        // clock depending on the number
        change_speed_on_input();
    }
    cleanup_platform();
    return 0;
}
```

```
/*****************************************************************************
* Filename:
C:\Users\superman\Desktop\byu-ee-427-labs\PIT\MyProcessorIPLib/drivers/pit_v1_00_a/src/pit
.h
* Version:           1.00.a
* Description:       pit Driver Header File
* Date:              Wed Nov 02 16:29:39 2016 (by Create and Import Peripheral Wizard)
*
*
* Taylor Cowley and Andrew Okazaki.
* Note: This file was auto-generated by the EDK and unchanged by us
*
*
*****************************************************************************/

#ifndef PIT_H
#define PIT_H

/***************************** Include Files *******************************/

#include "xbasic_types.h"
#include "xstatus.h"
#include "xil_io.h"

/************************** Constant Definitions ***************************/


/**
 * User Logic Slave Space Offsets
 * -- SLV_REG0 : user logic slave module register 0
 * -- SLV_REG1 : user logic slave module register 1
 * -- SLV_REG2 : user logic slave module register 2
 * -- SLV_REG3 : user logic slave module register 3
 */
#define PIT_USER_SLV_SPACE_OFFSET (0x00000000)
#define PIT_SLV_REG0_OFFSET (PIT_USER_SLV_SPACE_OFFSET + 0x00000000)
#define PIT_SLV_REG1_OFFSET (PIT_USER_SLV_SPACE_OFFSET + 0x00000004)
#define PIT_SLV_REG2_OFFSET (PIT_USER_SLV_SPACE_OFFSET + 0x00000008)
#define PIT_SLV_REG3_OFFSET (PIT_USER_SLV_SPACE_OFFSET + 0x0000000C)

/**************************** Type Definitions *****************************/


/***************** Macros (Inline Functions) Definitions *******************/

/**
 *
 * Write a value to a PIT register. A 32 bit write is performed.
 * If the component is implemented in a smaller width, only the least
 * significant data is written.
 *
 * @param   BaseAddress is the base address of the PIT device.
 * @param   RegOffset is the register offset from the base to write to.
 * @param   Data is the data written to the register.
 *
 * @return  None.
 *
 * @note
```

```
 * C-style signature:
 *  void PIT_mWriteReg(Xuint32 BaseAddress, unsigned RegOffset, Xuint32 Data)
 *
 */
#define PIT_mWriteReg(BaseAddress, RegOffset, Data) \
    Xil_Out32((BaseAddress) + (RegOffset), (Xuint32)(Data))

/**
 *
 * Read a value from a PIT register. A 32 bit read is performed.
 * If the component is implemented in a smaller width, only the least
 * significant data is read from the register. The most significant data
 * will be read as 0.
 *
 * @param   BaseAddress is the base address of the PIT device.
 * @param   RegOffset is the register offset from the base to write to.
 *
 * @return  Data is the data from the register.
 *
 * @note
 * C-style signature:
 *  Xuint32 PIT_mReadReg(Xuint32 BaseAddress, unsigned RegOffset)
 *
 */
#define PIT_mReadReg(BaseAddress, RegOffset) \
    Xil_In32((BaseAddress) + (RegOffset))


/**
 *
 * Write/Read 32 bit value to/from PIT user logic slave registers.
 *
 * @param   BaseAddress is the base address of the PIT device.
 * @param   RegOffset is the offset from the slave register to write to or read from.
 * @param   Value is the data written to the register.
 *
 * @return  Data is the data from the user logic slave register.
 *
 * @note
 * C-style signature:
 *  void PIT_mWriteSlaveRegn(Xuint32 BaseAddress, unsigned RegOffset, Xuint32 Value)
 *  Xuint32 PIT_mReadSlaveRegn(Xuint32 BaseAddress, unsigned RegOffset)
 *
 */
#define PIT_mWriteSlaveReg0(BaseAddress, RegOffset, Value) \
    Xil_Out32((BaseAddress) + (PIT_SLV_REG0_OFFSET) + (RegOffset), (Xuint32)(Value))
#define PIT_mWriteSlaveReg1(BaseAddress, RegOffset, Value) \
    Xil_Out32((BaseAddress) + (PIT_SLV_REG1_OFFSET) + (RegOffset), (Xuint32)(Value))
#define PIT_mWriteSlaveReg2(BaseAddress, RegOffset, Value) \
    Xil_Out32((BaseAddress) + (PIT_SLV_REG2_OFFSET) + (RegOffset), (Xuint32)(Value))
#define PIT_mWriteSlaveReg3(BaseAddress, RegOffset, Value) \
    Xil_Out32((BaseAddress) + (PIT_SLV_REG3_OFFSET) + (RegOffset), (Xuint32)(Value))

#define PIT_mReadSlaveReg0(BaseAddress, RegOffset) \
    Xil_In32((BaseAddress) + (PIT_SLV_REG0_OFFSET) + (RegOffset))
#define PIT_mReadSlaveReg1(BaseAddress, RegOffset) \
    Xil_In32((BaseAddress) + (PIT_SLV_REG1_OFFSET) + (RegOffset))
#define PIT_mReadSlaveReg2(BaseAddress, RegOffset) \
```

```
    Xil_In32((BaseAddress) + (PIT_SLV_REG2_OFFSET) + (RegOffset))
#define PIT_mReadSlaveReg3(BaseAddress, RegOffset) \
    Xil_In32((BaseAddress) + (PIT_SLV_REG3_OFFSET) + (RegOffset))

/************************** Function Prototypes *****************************/


/**
 *
 * Run a self-test on the driver/device. Note this may be a destructive test if
 * resets of the device are performed.
 *
 * If the hardware system is not built correctly, this function may never
 * return to the caller.
 *
 * @param   baseaddr_p is the base address of the PIT instance to be worked on.
 *
 * @return
 *
 *    - XST_SUCCESS   if all self-test code passed
 *    - XST_FAILURE   if any self-test code failed
 *
 * @note    Caching must be turned off for this function to work.
 * @note    Self test may fail if data memory and device are not on the same bus.
 *
 */
XStatus PIT_SelfTest(void * baseaddr_p);
/**
*  Defines the number of registers available for read and write*/
#define TEST_AXI_LITE_USER_NUM_REG 4


#endif /** PIT_H */
```

```
1  ------------------------------------------------------------------------------
2  -- user_logic.vhd - entity/architecture pair
3  ------------------------------------------------------------------------------
4  --
5  -- ****************************************************************************
6  -- ** Copyright (c) 1995-2011 Xilinx, Inc.  All rights reserved.            **
7  -- **                                                                        **
8  -- ** Xilinx, Inc.                                                          **
9  -- ** XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"         **
10 -- ** AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND       **
11 -- ** SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,        **
12 -- ** OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,        **
13 -- ** APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION           **
14 -- ** THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,     **
15 -- ** AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE      **
16 -- ** FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY              **
17 -- ** WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE               **
18 -- ** IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR        **
19 -- ** REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF       **
20 -- ** INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS       **
21 -- ** FOR A PARTICULAR PURPOSE.                                            **
22 -- **                                                                        **
23 -- ****************************************************************************
24 --
25 ------------------------------------------------------------------------------
26 -- Filename:          user_logic.vhd
27 -- Version:           1.00.a
28 -- Description:       User logic.
29 -- Date:              Wed Nov 02 16:29:35 2016 (by Create and Import Peripheral Wizard)
30 -- VHDL Standard:     VHDL'93
31 ------------------------------------------------------------------------------
32 -- Naming Conventions:
33 --   active low signals:                    "*_n"
34 --   clock signals:                         "clk", "clk_div#", "clk_#x"
35 --   reset signals:                         "rst", "rst_n"
36 --   generics:                              "C_*"
37 --   user defined types:                    "*_TYPE"
38 --   state machine next state:              "*_ns"
39 --   state machine current state:           "*_cs"
40 --   combinatorial signals:                 "*_com"
41 --   pipelined or register delay signals:   "*_d#"
42 --   counter signals:                       "*cnt*"
43 --   clock enable signals:                  "*_ce"
44 --   internal version of output port:       "*_i"
45 --   device pins:                           "*_pin"
46 --   ports:                                 "- Names begin with Uppercase"
47 --   processes:                             "*_PROCESS"
48 --   component instantiations:              "<ENTITY_>I_<#|FUNC>"
49 ------------------------------------------------------------------------------
50
51 -- DO NOT EDIT BELOW THIS LINE --------------------
52 library ieee;
53 use ieee.std_logic_1164.all;
54 use ieee.std_logic_arith.all;
55 use ieee.std_logic_unsigned.all;
56
57
```

```
58  ------------------------------------------------------------------------------
    ------------------------------------
59  -- uncomment the next two files
60  --library proc_common_v3_00_a;
61  --use proc_common_v3_00_a.proc_common_pkg.all;
62
63  -- DO NOT EDIT ABOVE THIS LINE --------------------
64
65  --USER libraries added here
66
67  ------------------------------------------------------------------------------
68  -- Entity section
69  ------------------------------------------------------------------------------
70  -- Definition of Generics:
71  --   C_NUM_REG                  -- Number of software accessible registers
72  --   C_SLV_DWIDTH               -- Slave interface data bus width
73  --
74  -- Definition of Ports:
75  --   Bus2IP_Clk                 -- Bus to IP clock
76  --   Bus2IP_Resetn              -- Bus to IP reset
77  --   Bus2IP_Data                -- Bus to IP data bus
78  --   Bus2IP_BE                  -- Bus to IP byte enables
79  --   Bus2IP_RdCE                -- Bus to IP read chip enable
80  --   Bus2IP_WrCE                -- Bus to IP write chip enable
81  --   IP2Bus_Data                -- IP to Bus data bus
82  --   IP2Bus_RdAck               -- IP to Bus read transfer acknowledgement
83  --   IP2Bus_WrAck               -- IP to Bus write transfer acknowledgement
84  --   IP2Bus_Error               -- IP to Bus error response
85  ------------------------------------------------------------------------------
86
87  entity user_logic is
88    generic
89    (
90      -- ADD USER GENERICS BELOW THIS LINE ---------------
91      --USER generics added here
92      -- ADD USER GENERICS ABOVE THIS LINE ---------------
93
94      -- DO NOT EDIT BELOW THIS LINE ---------------------
95      -- Bus protocol parameters, do not add to or delete
96      C_NUM_REG                     : integer              := 4;
97      C_SLV_DWIDTH                  : integer              := 32
98      -- DO NOT EDIT ABOVE THIS LINE ---------------------
99    );
100   port
101   (
102     -- ADD USER PORTS BELOW THIS LINE ------------------
103     --USER ports added here
104     myinterrupt : out std_logic;
105     -- ADD USER PORTS ABOVE THIS LINE ------------------
106
107     -- DO NOT EDIT BELOW THIS LINE ---------------------
108     -- Bus protocol ports, do not add to or delete
109     Bus2IP_Clk                    : in  std_logic;
110     Bus2IP_Resetn                 : in  std_logic;
111     Bus2IP_Data                   : in  std_logic_vector(C_SLV_DWIDTH-1 downto 0);
112     Bus2IP_BE                     : in  std_logic_vector(C_SLV_DWIDTH/8-1 downto 0);
113     Bus2IP_RdCE                   : in  std_logic_vector(C_NUM_REG-1 downto 0);
```

```
114       Bus2IP_WrCE                    : in  std_logic_vector(C_NUM_REG-1 downto 0);
115       IP2Bus_Data                    : out std_logic_vector(C_SLV_DWIDTH-1 downto 0);
116       IP2Bus_RdAck                   : out std_logic;
117       IP2Bus_WrAck                   : out std_logic;
118       IP2Bus_Error                   : out std_logic
119       -- DO NOT EDIT ABOVE THIS LINE ---------------------
120     );
121
122     attribute MAX_FANOUT : string;
123     attribute SIGIS : string;
124
125     attribute SIGIS of Bus2IP_Clk    : signal is "CLK";
126     attribute SIGIS of Bus2IP_Resetn : signal is "RST";
127
128   end entity user_logic;
129
130   ------------------------------------------------------------------------------
131   -- Architecture section
132   ------------------------------------------------------------------------------
133
134   architecture IMP of user_logic is
135
136     --USER signal declarations added here, as needed for user logic
137
138     ---------------------------------------
139     -- Signals for user logic slave model s/w accessible register example
140     ---------------------------------------
141     signal slv_reg0                       : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
142     signal slv_reg1                       : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
143     signal slv_reg2                       : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
144     signal slv_reg3                       : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
145
146     signal slv_reg_write_sel              : std_logic_vector(3 downto 0);
147     signal slv_reg_read_sel               : std_logic_vector(3 downto 0);
148     signal slv_ip2bus_data                : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
149     signal slv_read_ack                   : std_logic;
150     signal slv_write_ack                  : std_logic;
151
152   begin
153
154     --USER logic implementation added here
155
156     -- slv_reg0 = counter
157     -- slv_reg1 = delay_number
158     -- slv_reg2 = control register
159     -- slv_reg3 = unused for now
160
161
162
163
164
165     ---------------------------------------
166     -- Example code to read/write user logic slave model s/w accessible registers
167     --
168     -- Note:
169     -- The example code presented here is to show you one way of reading/writing
170     -- software accessible registers implemented in the user logic slave model.
```

```
171     -- Each bit of the Bus2IP_WrCE/Bus2IP_RdCE signals is configured to correspond
172     -- to one software accessible register by the top level template. For example,
173     -- if you have four 32 bit software accessible registers in the user logic,
174     -- you are basically operating on the following memory mapped registers:
175     --
176     --    Bus2IP_WrCE/Bus2IP_RdCE   Memory Mapped Register
177     --                     "1000"   C_BASEADDR + 0x0
178     --                     "0100"   C_BASEADDR + 0x4
179     --                     "0010"   C_BASEADDR + 0x8
180     --                     "0001"   C_BASEADDR + 0xC
181     --
182     ---------------------------------------------
183     slv_reg_write_sel <= Bus2IP_WrCE(3 downto 0);
184     slv_reg_read_sel  <= Bus2IP_RdCE(3 downto 0);
185     slv_write_ack     <= Bus2IP_WrCE(0) or Bus2IP_WrCE(1) or Bus2IP_WrCE(2) or
        Bus2IP_WrCE(3);
186     slv_read_ack      <= Bus2IP_RdCE(0) or Bus2IP_RdCE(1) or Bus2IP_RdCE(2) or
        Bus2IP_RdCE(3);
187
188     -- implement slave model software accessible register(s)
189     SLAVE_REG_WRITE_PROC : process( Bus2IP_Clk ) is
190     begin
191
192       if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
193         if Bus2IP_Resetn = '0' then
194           --slv_reg0 <= (others => '0');
195           slv_reg0 <= (others => '1');  -- counter resets to FF FF FF FF
196
197           slv_reg1 <= (others => '0');  -- delay resets to 00 00 00 00
198           slv_reg2 <= (others => '0');  -- control disables interrupts, does not load
        delay, and no decrement.
199           slv_reg3 <= (others => '0');  -- register to store what the interrupt should
        be. (only use LSB)
200         else
201           case slv_reg_write_sel is
202             when "1000" =>
203               for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
204                 if ( Bus2IP_BE(byte_index) = '1' ) then
205                   slv_reg0(byte_index*8+7 downto byte_index*8) <= Bus2IP_Data(byte_index
        *8+7 downto byte_index*8);
206                 end if;
207               end loop;
208             when "0100" =>
209               for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
210                 if ( Bus2IP_BE(byte_index) = '1' ) then
211                   slv_reg1(byte_index*8+7 downto byte_index*8) <= Bus2IP_Data(byte_index
        *8+7 downto byte_index*8);
212                 end if;
213               end loop;
214             when "0010" =>
215               for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
216                 if ( Bus2IP_BE(byte_index) = '1' ) then
217                   slv_reg2(byte_index*8+7 downto byte_index*8) <= Bus2IP_Data(byte_index
        *8+7 downto byte_index*8);
218                 end if;
219               end loop;
220             when "0001" =>
```

```
221               for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
222                 if ( Bus2IP_BE(byte_index) = '1' ) then
223                   slv_reg3(byte_index*8+7 downto byte_index*8) <= Bus2IP_Data(byte_index
      *8+7 downto byte_index*8);
224                 end if;
225               end loop;
226             when others =>
227
228
      ------------------------------------------------------------------------------
      -----------
229             -- Begin custom code for our PIT Timer
230
231             -- decrement? or no? This is based on bit 0 of our control register
232               if(slv_reg2(0) = '1') then
233                 -- We allow it to decrement
234                 slv_reg0 <= slv_reg0 - 1;
235               else
236                 -- no decrementing allowed
237                 slv_reg0 <= slv_reg0;
238               end if;
239
240             -- What happens when we hit 0? This is based on but 2 of our control register
241               if(slv_reg0 = "00000000000000000000000000000000") then
242                 -- we either reload or nothing
243                 if(slv_reg2(2) = '1') then
244                   -- we reload!
245                   slv_reg0 <= slv_reg1;
246                 else
247                   -- we do NOT reload, nor do we continue ticking
248                   slv_reg0 <= (others=>'0');
249                 end if;
250               end if;
251
252             -- make an interrupt if we ever hit 1. This way we can hold at zero without
      generating interrupts
253             -- This is based on but 1 of our control register
254             -- Notice the interrupt is stored in the LSB of reg3. The real interrupt
      signal will map to this at the end.
255               if((slv_reg0 = "00000000000000000000000000000001") and (slv_reg2(1) = '1'
      )) then
256                 slv_reg3(0) <= '1';
257               else
258                 slv_reg3(0) <= '0';
259               end if;
260
261             -- A custom value that reg3 should always be held at for debugging purposes
      (except the LSB; that is the interrupt)
262               slv_reg3(31 downto 1) <= "1010101010101010101010101010101";
263
264               -- End custom code for our PIT Timer
265
      ------------------------------------------------------------------------------
      ---------------
266
267           end case;
268         end if;
```

```
269       end if;
270     end process SLAVE_REG_WRITE_PROC;
271
272     -- implement slave model software accessible register(s) read mux
273     SLAVE_REG_READ_PROC : process( slv_reg_read_sel, slv_reg0, slv_reg1, slv_reg2,
      slv_reg3 ) is
274     begin
275
276       case slv_reg_read_sel is
277         when "1000" => slv_ip2bus_data <= slv_reg0;
278         when "0100" => slv_ip2bus_data <= slv_reg1;
279         when "0010" => slv_ip2bus_data <= slv_reg2;
280         when "0001" => slv_ip2bus_data <= slv_reg3;
281         when others => slv_ip2bus_data <= (others => '0');
282       end case;
283
284     end process SLAVE_REG_READ_PROC;
285
286     ------------------------------------------
287     -- Example code to drive IP to Bus signals
288     ------------------------------------------
289     IP2Bus_Data  <= slv_ip2bus_data when slv_read_ack = '1' else
290                     (others => '0');
291
292     IP2Bus_WrAck <= slv_write_ack;
293     IP2Bus_RdAck <= slv_read_ack;
294     IP2Bus_Error <= '0';
295
296
      ------------------------------------------------------------------------------
      --
297     -- The interrupt port is always the LSB of slv reg 3 :)
298     myinterrupt <= slv_reg3(0);
299     --
300
      ------------------------------------------------------------------------------
      -----
301
302   end IMP;
303
```