Lab Assignment 10
Controlling a Robotic Arm via a WiiMote through the ARM CPU and the FPGA

Introduction

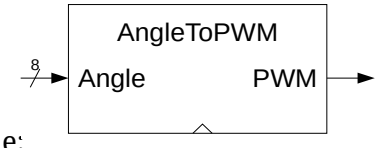
The goal of this lab is to bring together all of the elements you have developed so far in previous labs, and combine them in a hardware/software co-designed implementation. In this lab you will control the Robotic arm using the FPGA (as you did in Lab 9) and use the ARM processor on the ZedBoard to receive commands from the WiiMote and control the FPGA. You will be writing a C program that receives signals from the WiiMote connected via Bluetooth and sends commands to the FPGA to carry out a specific movement by the robotic arm. The FPGA will generate the necessary PWM signals to control the Robotic arm to carry out a specific action involving multiple servo motors.

Pre-Lab Assignment

[1] Review materials on AXI-Lite protocol in blackboard

[2] **Single PWM Signal**

Implementing a module that, given an angle, produces a PWM signal as an output. The module has the interface shown in the figure:



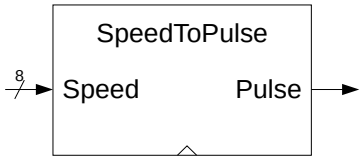
Here are some steps that will guide you through the design of this module:

1. Start with the PWM signal generator created in the previous lab assignment. Make a copy of this design in a file named *AngleToPWM.slx* and apply your changes on this file. The main difference in the current design is that the current pulse width is not stored in a counter, but instead calculated as a function of the current input.
2. The main challenge here is calculating the pulse, given as an operand to the *Relational Operator* component, from input *Angle*. An angle of 0° should produce a pulse width of 0.6ms, while an angle of 180° should generate a pulse of 2.4ms. You can perform the necessary arithmetic calculations by hardware using Simulink library components *HDL Coder → Math Operations → Product* and *HDL Coder → Math Operations → Divide*. If your operation uses a multiplication or division by a power of 2, you should use the *HDL Coder → HDL Operations → Bit Shift* component instead.
3. In order to make a *Divide* synthesizable, you need to edit its properties, and activate checkbox *Saturate on Integer Overflow*. Also, option *Integer Rounding Mode* should be set to *Zero*. When multiplying or dividing by constants, make sure you set the *Sample Time* and *Data Type* fields of the corresponding *Constant* blocks to the proper values.
4. Once you complete your design, modify the properties of input port *Angle* and set its data type to *uint8*. Then create a sub-system called *AngleToPWM*.
5. In the top-level design, connect a *Constant* block to the *Angle* input, and set its data type to *uint8*. Add a 2-input *Scope* component, and connect it to input *Angle* and output *PWM* of the *AngleToPWM* block.
6. Remember to adjust the counter values assuming a clock frequency of 50kHz for simulation purposes instead of 50MHz that's needed by the Zedboard.

Simulate your design for three different angles: 0°, 90°, and 180°. Add a screenshot of your design, and one screenshot of the *Scope*'s output for each of these three simulations. Zoom into the PWM signals until you distinguish the exact pulse width produced in each case, and report the observed values. Do they match your expectations?

Lab 10.1 Pulse generations with configurable speed

As an intermediate step to controlling the speed at which angles vary, we will now design an intermediate component that produces pulses at a configurable speed. This circuit takes an 8-bit integer value as an input, representing a speed in pulses per second. The circuit has a 1-bit output with a default value of 0 and occasional 1-cycle pulses at the given speed.



Here are some hints to build this circuit:

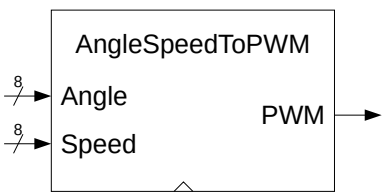
1. As done in previous circuits generating pulses, you can use an *HDL Counter (Free Running and 32 bit Word Length)* and a *Compare To Constant* block. When the counter outputs a value of 0, the magnitude comparator will output a value of 1 for one cycle. This signal is the output of the *SpeedToPulse* circuit.
2. The frequency at which this pulse occurs is determined by the maximum value of the counter. This value is now determined by the *Speed* input of the circuit. You need to add a *Local Reset* port to the counter by editing its properties. The counter should run continually, so you can get rid of its *Enable* input as well.
3. Finally, you need to add some combinational logic that determines when to reset the value of the counter, with the following principle: a *Reset* signal should be set when the current value of the counter exceeds the number of cycles in a second divided by the current speed. You will need a *Divide* and a *Relational Operator* block to implement this functionality.
4. When using *Divide* blocks, remember to activate check box *Saturate on Integer Overflow*, and to set option *Integer Rounding Mode* to *Zero*.
5. When you finish your design, pack it in a circuit named *SpeedToPulse*. Add a *Scope* component to monitor the value at input *Speed* and at output *Pulse* during the simulations.

Assignment 1

Add a screenshot with the design of circuit *SpeedToPulse*. Run two simulations of your design with two different values for input Speed. In each case, include screenshots from the output of the *Scope* component. Explain how your simulations prove the correct behavior of your design.

Lab 10.2 Configurable PWM signal with angle and speed

With the help of circuits *AngleToPWM* and *SpeedToPulse*, we will design a more comprehensive logic block called *AngleSpeedToPWM*. The inputs of this block are an angle between 0° and 180°, and a speed given in angles per second. The output is a PWM signal controlling a specific servo motor.



This circuit stores the current angle internally. When the angle provided as an input differs from the current angle, the latter is updated progressively, at the speed indicated in the second input. Here are some hints to build this circuit:

1. An *HDL Counter* block should store the current angle. The counter should only take values between 0 and 180, and its initial value should be 90. The counter should expose an *Enable* and a *Direction* signal, allowing for external increments or decrements by 1.
2. The combinational logic connected to the counter's *Enable* input can be designed with the following principle: the counter should be activated when the current angle (counter output) does not match the *Angle* input, and the *SpeedToPulse* component is generating a pulse.
3. To design the combinational logic connected to the counter's *Direction* input, keep in mind that the counter should go up when input *Angle* (counter output) is greater than the current angle, and down otherwise.
4. When you complete the design, pack it in a logic block named *AngleSpeedToPWM*. Add a *Scope* component monitoring the values for inputs *Angle* and *Speed*, as well as output *PWM*.

Assignment 2

Add a screenshot with the design of circuit *AngleSpeedToPWM*. Run a simulation for 1 second of virtual time by setting input *Angle* to 80° and input *Speed* to 10° per second, and add a screenshot with the outputs of the *Scope*. Zoom in and record the first and last pulse width of the *PWM* signal. Explain your results and prove that the behavior of the circuit is as expected.

Lab 10.3 Design verification in hardware

Once you have verified the *AngleSpeedToPWM* block through simulation, you can synthesize it into the FPGA with these steps:

1. Adjust the values of the counters and constants to reflect the fact that the actual FPGA frequency is 50MHz.
2. In the *AngleToPWM* component, you may even need to replace the *Divide* logic block with a *Product* logic block, according to the new formula to convert angles into PWM pulse cycles.
3. On the main *AngleSpeedToPWM* block, add one *Constant* block set to 180 and connect it to the *Angle* input. Add another *Constant* block set to 15 and connect it to the *Speed* input. Remember to edit the *Sample Time* and *Data Type* properties of both blocks.
4. Use the *HDL Workflow Advisor* to synthesize your design into the FPGA. In step *Set Target → Set Target Interface*, map the only output of your circuit (*PWM*) to port JA1[4]. This corresponds to the gripper servo.

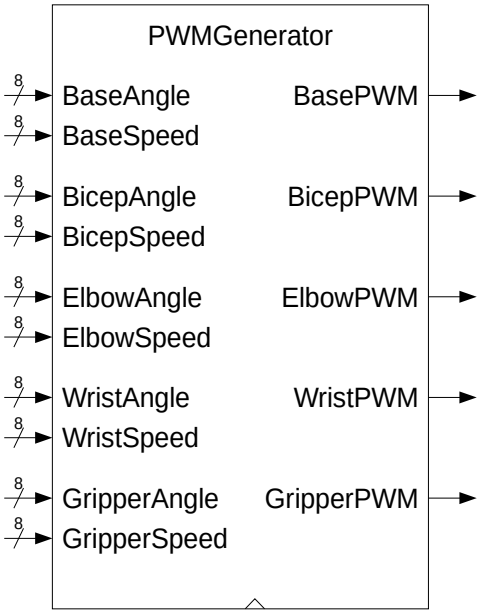
Assignment 3

Follow all steps to upload the design into the FPGA, and verify its correct behavior. Once you complete the last step, you should see how the gripper servo moves from its original 90° position into a target 180° position, at a speed of 15° per second.

Lab 10.4 Hybrid hardware-software design through the AXI4 interface

After validating the correct functionality of module *AngleSpeedToPWM* on the ZedBoard, you can now replicate this design to build a circuit that controls all servos by hardware, producing five different PWM signals. You can call the new circuit *PWMGenerator*.

As the circuit interface illustrates, each servo is controlled by two 8-bit values, containing a target angle and speed. We will configure the FPGA in such a way that these inputs can be controlled through memory locations accessible by the ZedBoard's ARM processor, in such a way that we can directly modify their values from our C++ programs. This is done through the so-called AXI4 (Advanced eXtensible Interface), a communication protocol designed for hardware-software interaction in embedded devices.



The circuit synthesis can be done through the following steps:

1. Design the *PWMGenerator* module and save it in a file named *PWMGenerator.slx*.
2. Use the *HDL Workflow Advisor* to synthesize the circuit into the FPGA. In step *Set Target → Set Target Interface*, connect the angle and speed inputs to memory-mapped I/O addresses by selecting *AXI4-Lite* in field *Target Platform Interfaces*. In field *Bit Range/Address/FPGA Pin*, choose the following memory offsets for each input of the circuit:

Input	Address
BaseAngle	x"100"
BaseSpeed	x"104"
BicepAngle	x"108"
BicepSpeed	x"10c"
ElbowAngle	x"110"
ElbowSpeed	x"114"
WristAngle	x"118"
WristSpeed	x"11c"
GripperAngle	x"120"
GripperSpeed	x"124"

Output	Pmod Connector
BasePWM	JA1 [0]
BicepPWM	JA1 [1]
ElbowPWM	JA1 [2]
WristPWM	JA1 [3]
GripperPWM	JA1 [4]

3. Use Pmod port JA1 to connect each PWM output signal to the corresponding servo motor shown above:

4. Complete the circuit synthesis. The FPGA should stay programmed with this design for the rest of this lab session. Make sure that the robotic arm is disconnected, since the AXI4 memory locations may contain garbage values that could send dangerous signals to the servos.

Assignment 4

Add a screenshot of the design of circuit *PWMGenerator*. Add another screenshot of the *HDL Workflow Advisor* at the stage where you have completed the mapping between circuit ports and AXI4 addresses or Pmod ports (step 1.2 Set Target Interface).

Lab 10.5 Programming complex movements

The FPGA is now set up as a *PWMGenerator* logic block. Its inputs are mapped into AXI4 memory locations, and its outputs into the robotic arm servos. We can now write a C++ program that sends values to the circuit inputs by just writing into the corresponding memory addresses. The following files include the interface and implementation of class RoboticArm, as well as a small main program that moves the arm's gripper and wrist servos in an infinite loop. Study the code, compile it, and run the program to verify its correct behavior.

RoboticArm.h

```
#ifndef ROBOTIC_ARM_H
#define ROBOTIC_ARM_H

// Physical base address of GPIO
const unsigned gpio_address = 0x400d0000;

// Length of memory-mapped IO window
const unsigned gpio_size = 0xff;

const int base_angle_offset = 0x100;  // Offset for base servo angle
const int base_speed_offset = 0x104;  // Offset for base servo speed
const int bicep_angle_offset = 0x108;  // Offset for bicep servo angle
const int bicep_speed_offset = 0x10c;  // Offset for bicep servo speed
const int elbow_angle_offset = 0x110;  // Offset for elbow servo angle
const int elbow_speed_offset = 0x114;  // Offset for elbow servo speed
const int wrist_angle_offset = 0x118;  // Offset for wrist servo angle
const int wrist_speed_offset = 0x11c;  // Offset for wrist servo speed
const int gripper_angle_offset = 0x120;  // Offset for gripper servo angle
const int gripper_speed_offset = 0x124;  // Offset for gripper servo speed

class RoboticArm
{
        // File descriptor for memory-mapped I/O
        int fd;

        // Mapped address
        char *pBase;

        // Write a value into the given memory offset in the memory-mapped I/O.
        void RegisterWrite(unsigned offset, unsigned value);

        // Read a value from the given memory offset in the memory-mapped I/O.
        int RegisterRead (unsigned offset);

public:

        // Class constructor
        RoboticArm();

        // Destructor
        ~RoboticArm();

        // Move a servo to a target position with the given speed. ARgument
        // 'angle' is a value between 0 and 180. Argument 'speed' in an integer
        // greater than 0, given in angles per second.
        //
        // Argument 'id' can take the following values:
        //      0 - Base
        //      1 - Bicep
        //      2 - Elbow
        //      3 - Wrist
        //      4 - Gripper
        //
        void MoveServo(int id, int angle, int speed);
};

#endif
```

RoboticArm.cpp

```
#include "RoboticArm.h"
#include <fcntl.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <iostream>


RoboticArm::RoboticArm()
{
        // Open memory mapped I/O
        fd = open("/dev/mem", O_RDWR);

        // Map physical memory
        pBase = (char *) mmap(NULL, gpio_size, PROT_READ | PROT_WRITE,
                        MAP_SHARED, fd, gpio_address);

        // Check success
        if (pBase == (void *) -1)
        {
                std::cerr << "Error mapping memory - fogot sudo?\n";
                exit(1);
        }

        // Initial servo positions
        for (int i = 0; i < 5; i++)
                MoveServo(i, 90, 180);
}


RoboticArm::~RoboticArm()
{
        // Unmap physical memory
        munmap(pBase, 0xff);

        // Close memory mapped I/O
        close(fd);
}


void RoboticArm::RegisterWrite(unsigned offset, unsigned value)
{
        * (volatile unsigned *) (pBase + offset) = value;
}

void RoboticArm::RegisterRead (unsigned offset)
{
        return * (volatile unsigned *) (pBase + offset);
}


void RoboticArm::MoveServo(int id, int angle, int speed)
{
        // Check valid servo
        if (id < 0 || id > 4)
        {
                std::cerr << "Invalid servo ID\n";
                exit(1);
```

```
        }

        // Verify valid angle
        if (angle < 0 || angle > 180)
        {
                std::cerr << "Invalid angle\n";
                exit(1);
        }

        // Set memory locations
        RegisterWrite(base_angle_offset + id * 8, angle);
        RegisterWrite(base_angle_offset + id * 8 + 4, speed);
}
```

main.cpp

```
#include "RoboticArm.h"
#include <unistd.h>

int main()
{
        RoboticArm robotic_arm;

        while (true)
        {
                robotic_arm.MoveServo(4, 45, 45);
                robotic_arm.MoveServo(3, 45, 45);
                sleep(1);

                robotic_arm.MoveServo(4, 90, 45);
                robotic_arm.MoveServo(3, 90, 45);
                sleep(1);
        }
}
```

Assignment 5

Program the robotic arm to grab an object (a pen, a ball, a box, ...) from the table and throw it as far as possible. Bring a tape measure and record the distance reached by the object. We will be competing in class for whose robot can throw objects the farthest!

a. First, use the provided program to identify the position for each movement
b. Then, create a set of `MoveServo` statements to move the robotic arm accordingly.

HINT: It takes the robot a while to move to the desired location. Consider this in the program.

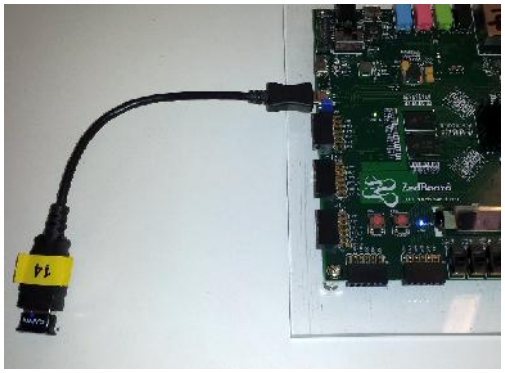
Record a video with a duration of at most 30 seconds and upload it on Blackboard, together with your lab report.

Controlling Robotic Arm via a Wiimote

This section introduces you to non-blocking and blocking file I/O. With a blocking read, the file read only returns (i.e., execution continues) after all the required bytes are available. However this is problematic if the code should read from multiple sources. If both reads are blocking, input from another source will be missed. The solution is to use non-blocking calls. Using a non-blocking call, the read will terminate (return) even if insufficient data is received. This allows the code to continue execution. However, the code should include error-handling logic to deal with receiving fewer bytes than expected.

Lab 10.6 Establish Bluetooth Connection to WiiMote

In the following parts, we will use the C++ programs from Lab4 (the Wiimote lab) and the C++ code above for reading acceleration and buttons, and use the hardware speed control (downloaded Simulink design) for controlling the robotic arm. Note that, the `RoboticArm.h` and `RoboticArm.cpp` programs above will replace the `ZedBoard.h` and `ZedBoard.cpp` programs from Lab4.



1. Connect the WiiMote
   a. Connect the USB Bluetooth dongle (through the standard USB-to-micro USB adapter) to the ZedBoard in the micro USB port on the ZedBoard named “PROG” (the label is on the board).
   b. Log into the ZedBoard with your username.
2. You will need the Wiimote connection script `WiimoteConect.sh` from Lab4.
3. Assign execution permissions to the shell script with the following command:

```
$ chmod +x WiimoteConnect.sh
```

4. Start the Bluetooth adapter:

```
$ sudo hciconfig hci0 up
```

5. Then, press the red sync button again, and before the pairing time window expires, call the script to connect. You should invoke the script using the `sudo` command, as follows:

```
$ sudo ./WiimoteConnect.sh
```

6. The LED 1 on WiiMote should turn on. *Consult Lab4 steps if you have problems at this point.*

Lab 10.7 Control Robot through Wiimote

This is an open-ended assignment. Your task is to design and realize how to control the robotic arm through the WiiMote buttons and accelerometer values.

With the buttons and accelerometer events coming from different device files (buttons from `event2`, acceleration from `event0`), it is a challenge to read both at the same time. To eliminate the effect of blocking calls on button events, so that we can read accelerometer values without waiting for button presses, we will introduce how to turn button events into a non-blocking call.

1. Change the file open for the buttons in `WiimoteBtns.cpp` from:

   ```
   fd = open("/dev/input/event2", O_RDONLY);
   ```

   to:

   ```
   fd = open("/dev/input/event2", O_RDONLY | O_NONBLOCK);
   ```

   By adding the option `O_NONBLOCK`, a read call will return, even if we have not received the expected number of bytes.

2. Design a mechanism to control the robotic arm using buttons and acceleration as inputs. You can use any of the 11 buttons (do not use the power button) and accelerometer values from 'X' and/or 'Y' axes to control 5 servo motors.
3. Record design decisions of how you control the robot in your report.

Assignment 6
Realize your design decisions by implementing them. Validate your design decisions by executing on the robotic arm. Please also record mistakes/improvements of your design decision. Record a video of your robotic, and describe your findings in the report.

Laboratory Report

You should follow the lab report outline provided on Blackboard. Your report should be developed on a wordprocessor (e.g., OpenOffice or LaTeX), and should include graphics when trying to present a large amount of data. Include the output of compiling and running your programs. Upload the lab report on blackboard. Include each program that you wrote in the appendix of your report.

Appendix: If you need to reload the Simulink bit file

Hint: How to Reprogram the FPGA without HDL Advisor

Going through the HDL Workflow Advisor until "Programming FPGA" performs two steps. It creates a bit file, which is basically the "program" for the FPGA, and programs (downloads and loads) the bit file into the ZedBoard. The FPGA configuration is lost upon power cycling the board.
After creating the bit file once, the FPGA can be configured directly without going through the HDL Workflow Advisor. Here are the steps:

1. Locate the bit file. It should be in the following directory structure. Set your MATLAB path to this location:

   ```
   hdl_prj/pa_prj/pa_prj.runs/impl_1/system_stub.bit
   ```

2. Enter the following MATLAB command to configure the FPGA:

   ```
   hdlturnkey.tool.EDKTool.downloadBit ('system_stub.bit')
   ```

   An output of 1 indicates success, and 0 indicates failure