CprE 488 – Embedded Systems Design

MP-1: Quad UAV Interfacing

Assigned: Monday of Week 4
Due: Monday of Week 6
Points: 100 + bonus for automated flight and advanced diagnostics

[Note: the goal of this Machine Problem is for you to work with your group to increase your exposure to three different aspects of embedded system design:

1. *Interfacing – you will reverse engineer an RC transmitter to gain an understanding of the trainer interface and format.*
2. *IP core design and integration – you will use the Xilinx tools to generate an AMBA AXI4-compliant IP core that you will integrate into an existing XPS project.*
3. *Finite State Machine design – you will design and implement an FSM-based hardware module to record and transmit PPM data.*

1) Your Mission. Jeff Bezos believes that within a few years' timeframe, Amazon will have Unmanned Aerial Vehicles (UAVs) deliver packages right to your doorstep. What could possibly go wrong with his vision, right? As a newly-hired engineering intern at a rival e-commerce company, you have been tasked with discovering spectacular answers to this question!

There are significant complexities involved in building a system for autonomous flight, as it requires solving challenges in multiple engineering domains, including image and signal processing, controls, and real-time embedded systems. Fortunately, as a fresh intern your only task (for now) is to design an interfacing and control platform for a simple quadcopter UAV and demonstrate its effectiveness.

2) First Flight. An RC quadcopter is a 4-rotor helicopter with motors that spin in counter-rotating pairs to create a downward thrust while canceling out rotational inertia, with two rotors spinning clockwise and two spinning counter-clockwise. We are using the Mini Fly QuadCopter ARF system which has a MultiWii Controller (MWC)

board and weighs only 325g. Quads of this scale are typically powered by a Lithium polymer (Lipo) battery and are controlled via a wireless transmitter / receiver pair.

Transmitters come in various configurations – ours is a Hobby King 2.4 Ghz 6 channel mode 2 transmitter. In mode 2, the left stick controls thrust (up-down) and yaw (left-right), and the right stick controls pitch (up-down) and roll (left-right). After a brief orientation from the TA, spend 5-10 minutes gaining some familiarity with quad flight. In your writeup, describe your experiences in practicing controlling the quad. Please keep the following considerations in mind when flying the quad:

- **Safety** should be your primary concern. Use the open space in the back of the lab to fly the quad, and make sure it is tethered safely to the ground at ALL times.
- The quads will only fly well when the **batteries** are completely charged. We have plenty of spare batteries and chargers, so just make sure you disconnect your battery from the quad and plug it into a charger when you're done. Note that these Lipo cells can produce a decent amount of current, so don't short-circuit or connect them to the chargers backwards.
- Did I mention **safety**? Do not fly the quad when students are walking nearby, do not hold them in your hand when they are activated, and in general respect the speed in which the propeller blades can spin. Learn from Frankie below!

Figure: former CprE 488 student Frankie Four-Fingers after his unfortunate quadcopter incident

- Start out **slowly**. We have spare blades and other parts, but our general rule is if you break it, you fix it.

3) Trainer Port. As is common in the RC world, our Hobby King transmitters have a trainer port on the back, which uses a 3-pin serial interface to transmit the Pulse Position Modulation (PPM) data that was described in the HW-1. A typical usage scenario of a trainer port is to have two people share control of an RC aircraft: the input (student) transmitter directly sends PPM data over the trainer port to a second output (instructor) transmitter, which copies this data over the RF link to the receiver on the aircraft.

See the illustration below – the intention behind this configuration is that the student can practice flying the aircraft while the instructor can take back control at any time.

Figure: your two TAs illustrating RC flight in a trainer configuration

Using the supplied cable, connect the trainer port of your input transmitter to an oscilloscope, and in your writeup, describe the PPM signals. What do each of the channels correspond to, and what are their minimum and maximum ranges? What is the total length of the individual PPM frames, and what is the minimum length of the idle pulse?

For our purposes, we will be using the ZedBoard as an intermediate processing step between a pair of student and instructor transmitters, and will be connecting the trainer ports directly to PMOD pins on the ZedBoard. Based on the ZedBoard documentation and your oscilloscope measurement of the trainer port, what concerns do you have about making this connection? Be specific, and confirm with your TA before continuing.

4) Project Checkout. Although we have not provided any source code or project files for this assignment, you can check-out an existing directory structure that includes the setup script as well as some documentation on the RC platform:

```
>> cd cpre488
>> svn checkout https://source.ece.iastate.edu/svn/cpre488/MP-1
```

Remember that every time you open up a new console window to work on your labs, you need to source the `setup.sh` file:

```
>> source setup.sh
```

5) System and IP Creation. Similar to the process we followed in MP-0, open up Xilinx Platform Studio (XPS) and create a new AXI System project using the Base System Builder wizard (put it in your `MP-1/system/` directory, and call it `system.xmp`) targeting the Avnet ZedBoard. Keep all the other options as default.

Once the main System Assembly View window comes up, we will be creating a new hardware IP core to interface with the ARM-based Processing System. Follow these precise steps to create your core:

1. Select “Hardware -> Create or Import Peripheral”. Click “Next” and then select the “Create templates for a new peripheral” option.
2. On the next screen, select the option to add the core to an existing XPS project. By default your current project should be selected. Click “Next”.
3. Since we are creating an IP core to transmit PPM data over the AMBA AXI bus, name your component `axi_ppm`. The default versioning should be fine. Click “Next”.
4. For the bus interface, select “AXI4-Lite”. We will only be reading and writing register values in this peripheral, so there is no need for the higher throughput options. Click “Next”.
5. On the next screen, select “Software reset” and “User logic software register”. Click “Next”.
6. For the user software registers, select “32”. Click “Next”
7. The default IP interconnect ports are fine. Click “Next”. Click “Next” again to skip the bus functional model simulation support.
8. On the next screen, select generation of ISE and XST project files, and generation of template software driver. Do not select the Verilog option unless you are considerably more comfortable with Verilog than VHDL. Click “Next”, and then “Finish”.

Your new core will be generated in the `MP-1/system/pcores/axi_ppm_v1_00_a/` directory. Many of the files here are for XPS to be able to automate the integration of the core into the rest of your design – let’s analyze the layout of this directory using the `tree` command:

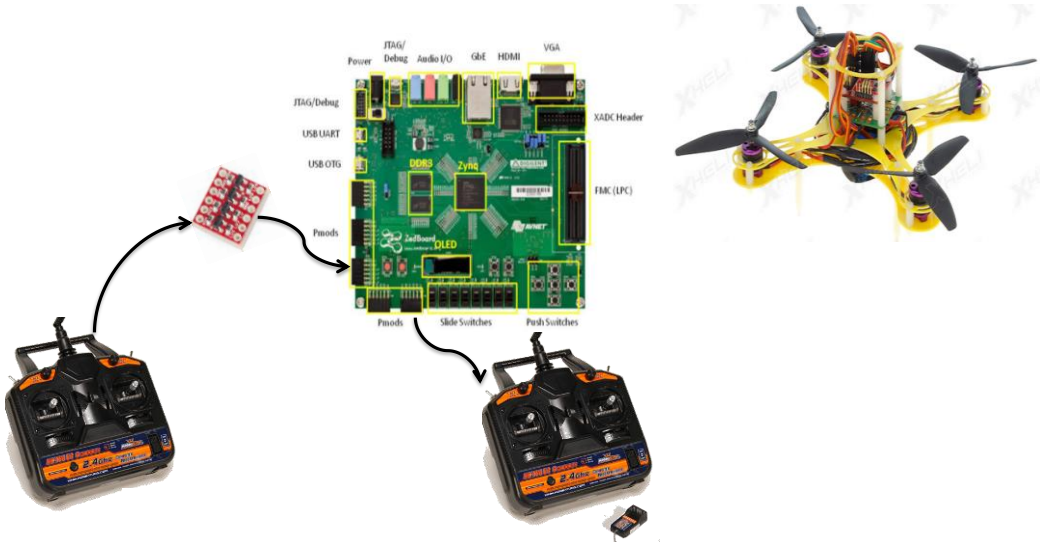
```
axi_ppm_v1_00_a
|-- data
|   |-- axi_ppm_v2_1_0.mpd  // Core port description for XPS
|   `-- axi_ppm_v2_1_0.pao  // HDL file analysis order for XPS
|-- devl
|   |-- projnav       // ISE project file for pcore development
|   `-- synthesis     // Scripts to synthesize this module in XPS
`-- hdl               // Generated VHDL and Verilog files
    `-- vhdl
        |-- axi_ppm.vhd     // Top-level module
        `-- user_logic.vhd  // Main IP functionality placed here
```

For now, use the ISE tool to open up the auto-generated project so that we easily make and test changes:

```
>> cd MP-1/system/pcores/axi_ppm_v1_00_a/devl/projnav
>> ise axi_ppm.xise &
```

Use ISE to navigate the `axi_ppm` peripheral’s structure. In your writeup, provide a structural diagram of the `axi_ppm` design, from the top-level AMBA AXI interface down to the (as of yet unmodified) `user_logic` module. How does an address on the AMBA bus generate a read or write enable signal for the slave registers in your design?

As previously mentioned, we are using the ZedBoard as an intermediate PPM processing step between the pair of transmitters and the quad UAV. Specifically, we are connecting the trainer port of the “input” (student) transmitter to one of the PMOD data ports, and another PMOD data port to the trainer port of the “output” (instructor) transmitter.

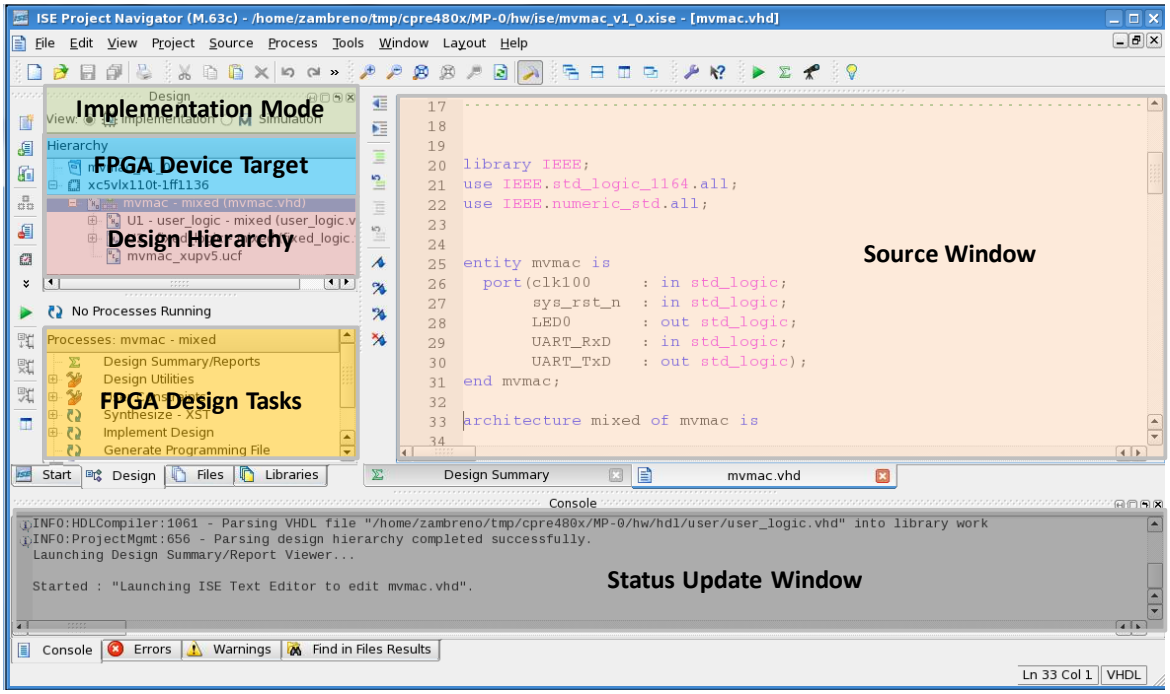


Note the Sparkfun Logic Level converter sitting between the input transmitter and the PMOD interface – sometimes it pays to read through the entire lab document! Based on this setup, modify your `axi_ppm` module to implement the following functionality:

1. At the top level, interface directly to ports PPM_Input and PPM_Output which are data ports on two separate PMODs on the ZedBoard. While you can create a user contraints file to map these ports in ISE, there is no need at this point since we are not mapping ISE's output directly to the ZedBoard. Instead, you will need to modify the `data/axi_ppm_v2_1_0.mpd` file to specify that your IP core has two new external ports.
2. Use slv_reg0 as a general configuration register. For now, when slv_reg0(0) is equal to '0', the core operates in a "hardware relay" mode. In this mode, pass PPM_Input directly to PPM_Output.
3. Create a Capture_PPM Finite State Machine that samples PPM_Input. When the PPM values for 6 channels have been detected, stores the corresponding cycle counts into slv_reg10 through slv_reg15. These registers will only be readable (not writeable) via software, so you can sever their write connection to the AMBA AXI.
4. Once a new frame of PPM values has been captured (ch1 through ch6), increment a counter in slv_reg1. This register will be polled by software to easily determine when new PPM data is available.
5. Create a Generate_PPM Finite State Machine that takes cycle counts stored in slv_reg20 through slv_reg25 and generates an appropriate signal for ch1 through ch6 on sw_PPM_Output. These registers will be writeable via software.
6. When slv_reg0(0) is equal to '1', we are in "software relay" mode. The sw_PPM_Output value should be connected to PPM_Output.

Starting from the generic FSMs in your HW-1 write-up, update the user_logic module (either `user_logic.v` or `user_logic.vhd`) such that the design can appropriately capture and generate PPM values for the Hobby King 6ch transmitter as described above.

Before jumping in, let's briefly analyze the main ISE window:



Implementing an FPGA design requires analysis of HDL code and is partitioned into several distinct stages:

- In the *synthesis* stage, the code is examined for syntax errors and then sophisticated algorithms translate assignment statements in the HDL code to the underlying FPGA hardware resources (LUTs, registers, multipliers, memories, I/O, etc.). This stage can take several minutes to complete, depending on the complexity of your code.
- In the *translate* stage, different components of the design are connected together into a single description, and in the *map* stage the hardware resources of the design are mapped to physical components that make up the selected FPGA device. These stages are relatively quick, although map will fail if the design is too large to fit on the device.
- In the *place-and-route* stage, algorithms determine the physical locations (X and Y coordinates) on the FPGA where that the mapped components should go. Routing refers to the wiring of these placed components. This take can be considerably lengthy, depending on how large your design is relative to the size of the FPGA. Not all designs can be routed successfully, requiring further optimization.
- In the *programming file generation* stage the placed and routed design is then translated to a binary format (the "bitfile" or "bitstream") that the FPGA understands. There is typically a 1:1 relationship between a post place-and-route design and the bitfile.

These stages are all dependent on the previous stages in the chain, so for example, running place-and-route will run synthesis first. ISE is also integrated with both the ModelSim and ISim simulators. While ISim is the default for this project, by selecting "Project -> Design Properties", you can modify the integrated simulator to ModelSim (select "ModelSim-SE Mixed"). Clicking on the "Simulation" button near the upper left-hand corner of ISE will take you to a view of the project files that are for simulation (initially, for this project the files are identical), and from there the option pops up to simulate the design, under "ModelSim Simulator (or ISim Simulator) -> Simulate Behavioral Model".

Note that making these changes to the `user_logic` module is not a trivial task. Some suggestions as you implement and test this module:

a) While it might be tempting to looking for the rising or falling edge of the PPM_Input port in order to store those values, FPGA synthesis tools treat all signals that trigger code on an edge (e.g. the `rising_edge` or `falling_edge` function in VHDL, `@posedge` and `@negedge` in Verilog) as a clock signal, which is not appropriate for this design. Instead, sample PPM_Input and wait until a transition has been stable for multiple clock cycles.
b) Create a testbench! You can either use ModelSim's built-in scripting language to set the input values directly, or create a new VHDL / Verilog module that instantiates `axi_ppm` as the design under test and forces appropriate values. Both options allow for specification of wait periods using real time units which should serve as an appropriate test of correctness.
c) To minimize the chance of synthesis – simulation mismatches (in which a design that appears to work perfectly in behavioral simulation exhibits incorrect functionality once mapped to hardware), use the two process FSM style as described in chapter 7 of the *Free Range VHDL* tutorial, available under `MP-1/docs/Tools/`. In this style, one combinational process sets the next state and output values, and a second sequential process registers the current state. In the combinational process, make sure that every output signal is assigned a default value so as to avoid inferring latches.

6) System Completion. After you are fairly confident in your axi_ppm module, go back to XPS and add it to your MP-1 system project. Note that while the core will be automatically connected to the AXI bus, you will need to connect the PPM_Input and PPM_Output ports to external ports on the design, and constrain these external ports to the appropriate PMOD pins on the FPGA.

After correctly integrating the core into your system project, you are ready to export the design to SDK by clicking the "Export Design" button. As before, the default directory for this option should be ok – click "Export & Launch SDK" and then take a quick nap.

7) Control Software. Hopefully you are now well-rested. As before, it is recommended that you place your XSDK workspace in the `MP-1/sw/` directory, and do NOT check the "Use this as default" option. Create a new Application Project named `rc_control`, targeting the "system_hw_platform" hardware platform, "ps7_cortexa9_0" processor, "standalone" OS/Platform, "C" programming language, and "system_bsp" as a newly created Board Support Package.

Your `rc_control` project should demonstrate the following modes:

1) When SW0 is set to 0 put the system in *hardware relay* mode. Otherwise, set it to *software relay* mode. As previously mentioned, in hardware relay mode the axi_ppm module directly passes PPM_Input to PPM_Output. In software relay mode, the CPU reads the most recent PPM frame (stored in slave registers 10 through 15) and copies the values to slave registers 20 through 25, which are read by the axi_ppm module to generate PPM_Output. Quadcopter flight should not be appear any different in these two modes.
2) If the middle button (BTNC) is pressed, exit the application. This is in general a good idea for all apps that feature an infinite control loop, as the Processing System occasionally locks (requiring a ZedBoard power cycle) up even when the application terminate button is pressed.

3) When SW1 is set to 1, put the system in *software debug* mode. In this mode the software outputs (over UART) the current value of the PPM channels stored in the slave registers in axi_ppm.
4) SW2 turns on *software record* mode. In this mode, the down button (BTND) stores the next PPM frame in an array and increments the array index, and the up button (BTNU) rewinds the recording by decrementing the index to this array.
5) SW3 turns on *software play* mode. In this mode, the right button (BTNR) transmits any stored PPM values over the axi_ppm, while the left button (BTNL) decrements the current play index.
6) SW4 turns on *software filter* mode. In this mode, any values to be transmitted (via) software to the axi_ppm are first analyzed to determine if the quad will be put in an unstable position.

Note that XPS does create a software driver for the `axi_ppm` module, which can be found in the `MP-1/system/drivers/` directory. You may use this driver by copying it into your `rc_control` software project, although this isn't necessary since the `axi_ppm` module is a simple memory-mapped peripheral.

What to submit: a .zip file containing your modified source files (modifications to `user_logic.vhd`, `system.mhs`, `system.ucf`, and `rc_control.c`) and your writeup in PDF format containing the highlighted sections of this document. In the Blackboard submission, list each team member with a percentage of their overall effort on MP-1 (with percentages summing to 100%).

What to demo: at least one group member must be available to demo the current state of your implementation. A full demo score requires a system that can fly using the various hardware and software modes, but partial credit will be given for effort. Be prepared to briefly discuss your source code.

BONUS credit. MP-1 has two separate bonus point criteria. The first is for the group with the best automated flight *longevity*. Using a combination of software record, software play, and software filter modes, how long can you keep your quad in the air without using the input transmitter? To be eligible for these bonus points, the quad has to land safely after its automated flight. (25 bonus points).

The second criterion is advanced *diagnostics*. In your software debug mode, the ZedBoard transmits over UART the current PPM values for the six RC channels. While this is useful for debugging the system, it does not provide much in terms of any useful diagnostics for the flight itself. Upgrade the diagnostic capability of the system, through some combination of software running on the ZedBoard and your host workstation. The teams with the most useful data diagnostics visualization as determined by the TAs will be eligible for this bonus. (25 bonus points).

Each group is limited to 100 bonus points for the entire semester.