WIRELESS ROVER

6.111 FINAL PROJECT REPORT

RYAN DAMICO - RYAN MANUEL

12 MAY, 2004

ABSTRACT

This document details the design and development of a wireless rover capable of remote operation and wireless data transmission. Discussion includes design specifications, methodology, testing, and debugging. Emphasis is placed on designing a robust system and employing good general engineering practices.

TA: JIA FU

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.111 - INTRODUCTORY DIGITAL SYSTEMS LABORATORY

WIRELESS ROVER

6.111 FINAL PROJECT REPORT

TABLE OF CONTENTS

OVERVIEW 4

BASE CONTROL STATION (RYAN MANUEL) 4
WIRELESS INTERFACE (RYAN DAMICO) 4
ROVER (RYAN DAMICO) 4
MINI-FPGA (RYAN DAMICO) 5

MODULE DESCRIPTION AND IMPLEMENTATION 5

BASE CONTROL STATION (RYAN MANUEL) 5
Major Finite State Machine Module *5*
PlayStation Controller Finite State Machine *6*
Stack Finite State Machine *7*
Store Data Finite State Machine *7*
Replay Finite State Machine *8*
WIRELESS TRANSCEIVER (RYAN DAMICO) 8
Data packet format *8*
Transmit Interface Module *9*
Receive Interface Module *11*

DESIGN METHODOLOGY 12

BASE CONTROL STATION 12
WIRELESS INTERFACE 12

TESTING 13

BASE CONTROL STATION 13
WIRELESS INTERFACE 14

APPENDIX 14

BASE CONTROL STATION 14
Excerpts from controller_fsm.v *14*
Excerpts from store_data.v *15*
Data on the PlayStation Interface *16*
The Playstation Controller Pinouts *16*
The PSX Controller Signals *17*
The PSX Controller Data *18*
WIRELESS INTERFACE 18
Pulse-Width Modulation code *18*
Code to load the CC1010 wireless microcontroller with data from an FPGA *19*
Top level instantiations for transmit and receive interface modules *19*
FSM from receive interface module *20*
Demonstrates shifting data to match incoming bitstream. Contains code to visually verify the data after transmission (must be enabled) *20*

LIST OF FIGURES

FIGURE 1: BLOCK DIAGRAM FOR BASE STATION 1
FIGURE 2: FSM FOR THE PLAYSTATION CONTROLLER 6
FIGURE 3: DIAGRAM STACK FSM 7
FIGURE 4: COMMAND AND SENSOR DATA PACKET BREAKDOWN.................. 9
FIGURE 5: FPGA/WIRELESS TRANSCEIVER INTERFACE..................................... 10
FIGURE 6: LOGIC ANALYZER CAPTURE OF SYNCHRONOUS SERIAL INTERFACE.. 10
FIGURE 7: TRANSMIT INTERFACE MODULE FSM 11
FIGURE 8: RECEIVE INTERFACE MODULE FSM 11

Overview

Base Control Station (Ryan Manuel)

The goal of this part of the lab was to create a base station that would take in input from a PlayStation controller to control the rover. In addition, the base control station records a history of the robots movements so that the user can playback this history in reverse.

The PlayStation controller controls the rover in a fairly straightforward manner. The four directional controls on the controller tell the rover which direction to go and the other buttons provide control for the various tools on the rover like the light and claw. The station takes the input from the controller and sends appropriate commands to the rover.

At the same time, the rover saves the motor speeds in a RAM. To save space the base station records only the motor speeds and how long the rover has been going at that speed. If the rover gets stuck and wants to reverse its history, it simply reads the history in reverse out of the RAM and reverses the motor speeds for the same length of time.

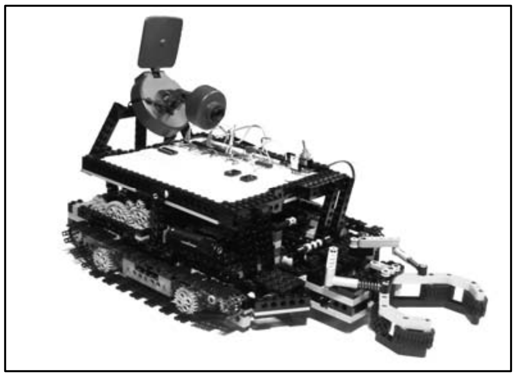
Wireless Interface (Ryan Damico)

The rover depends on a wireless link with the base station to receive movement commands and transmit sensor data. Wireless data transmission is implemented with a Chipcon CC1010 integrated RF transceiver and microcontroller. The microcontroller runs C code, and acts as the major FSM for the entire system. Doing so allowed great flexibility in quickly debugging the myriad problems and complexities associated with the transceiver. The transceiver controls two major modules on the FPGA, the transmit interface module and the receive interface module. The transmit interface module is invoked by the microcontroller when data needs to be transmitted, and the receive interface module is invoked when a packet has been received and must be handed off to the FPGA.

The major FSM for the system is a simple send, receive, send, receive loop. This simple setup was sufficient to instruct the rover and base station to send data and receive commands regularly, though many hours were spent writing C code that implemented the hardware send and receive abstractions.

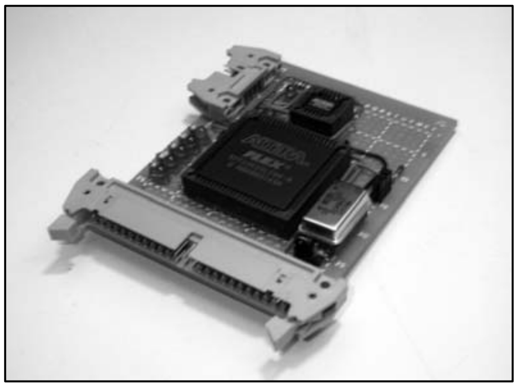
Rover (Ryan Damico)

The wireless rover was built from LEGOs and designed specifically for 6.111. It features dual treads with independent, speed-adjustable motors, a high torque gripping claw with rubber grips, an inclination control to pitch the claw up and down to prevent dragging of objects, a search light, three integrated protoboards, dual battery bays, and an independent wireless video camera system.



Mini-FPGA (Ryan Damico)

The wireless rover utilizes a custom-built mini-FPGA to allow it to be truly mobile. Based on the schematics for the FPGAs used in lab kits, the mini-FPGA includes an Altera Flex10K10 and EPROM, JTAG programming connector, swappable crystal oscillator, 5 debugging LEDs, and a 50-connector adapter offering access to 20 I/O pins, power, and the onboard oscillator. The mini-FPGA will be made available to future 6.111 students allowing them to design powerful, portable systems.



Module Description and Implementation

Base Control Station (Ryan Manuel)

While designing this traffic control system, it was easiest to break the system up into several modules and interconnect them. This modularization made the system easier to visualize and describe on paper. It also allowed for easier testing and debugging because one could just debug and test each small module individually as opposed to debugging and testing one big system. The breakdown of the system can be seen in **Error! Reference source not found.**. There is an overlaying major/minor FSM structure that will be discussed more below. There is also a PlayStation controller and RAM that are external to the FPGA. Also there are modules whose jobs were to format and convert data that is either sent to or received from the wireless module. Lastly we had a simple display module that was going to display temperature values via hex LEDs and heading of the robot via 8 LEDs located in the compass positions.

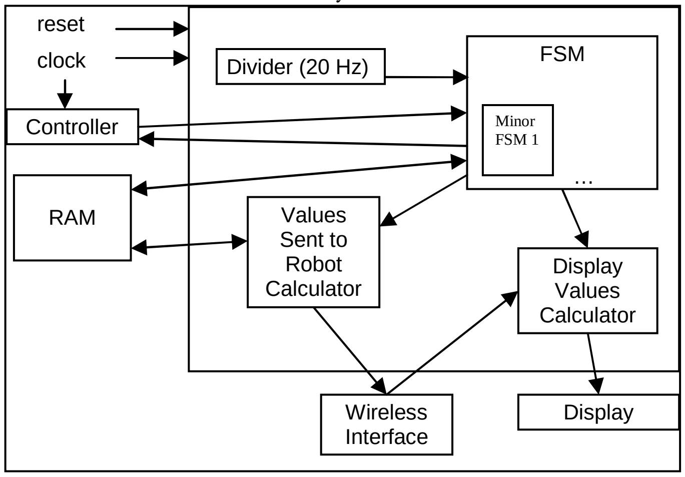


Figure 1: Block Diagram for Base Station

Major Finite State Machine Module

The control for the base station relies on a major/minor FSM structure. The FSM cycles through each state at 20 Hz. In each state other minor FSMs are initiated and then the major FSM waits around until the minor FSM is finished before it moves on to the next FSM.

The basic control is fairly simple and for the sake of brevity, the FSM diagram is omitted. The FSM waits in the idle state until it receives a sample signal. At this point the FSM starts the controller FSM. Then depending on whether a user has indicated they want to replay the history or not, the FSM either moves to the store data state or replay state. Once either of these have finished, the FSM moves back to the IDLE state.

PlayStation Controller Finite State Machine

The PlayStation Controller FSM is in charge of controlling the communication between the base station and the PlayStation Controller. The interface to the PlayStation Controller is fairly simple. Data is sent between the controller and the FPGA serially in groups of eight bits. In between each byte an *ack* signal is sent from the controller to the FPGA to let the FPGA know it is ready for the next round of eight bits.

The FSM begins in an IDLE state and waits around until it hears a start signal. When the FSM receives this signal, it moves to the INIT_CONTROLLER state and begins a handshaking process between the controller and the base station. The handshaking process begins when the FPGA sets the att signal low and sends 0x01 to the controller. This gets the controller's attention and initializes the controller. After the FPGA receives an ack from the controller, the FSM moves into the REQUEST_DATA state where it sends 0x42 to the controller telling the controller to send its button data. After another ack signal, the FPGA moves to GET_READY_FOR_DATA state where it waits for the controller to send it a byte signal letting the FPGA know that the data is coming. This ends the handshaking process.

After an ack is received in the GET_READY_FOR_DATA state, the FSM moves into the READ_BUTTONS1 state where it receives information on several of the buttons from the controller. The FSM waits for another ack and then receives data on more buttons. After another ack and another round of data the controller is finally finished and without waiting for a further ack the FSM sets att high and switches back to the IDLE state.

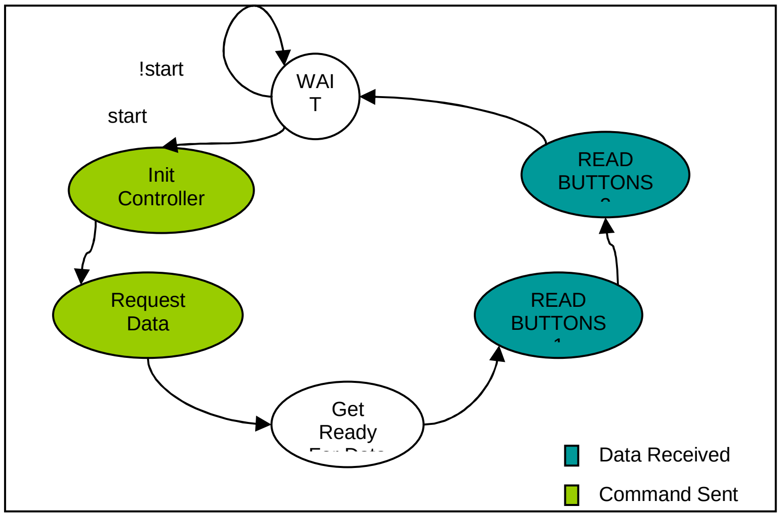


Figure 2: FSM for the PlayStation Controller

The entire process can be seen visually in Figure 2.

Stack Finite State Machine

The job of the Stack Finite State Machine is to implement a stack interface to the RAM. The stack allows two operations: push and pop. Push first adds the current motor speeds to the RAM. The motor speeds are stored as one eight bit piece of data comprised of four bits of right motor data and four bits of left motor data. After the speeds are added to the RAM, the push operation then increments the top of the stack by one and adds the duration of the motor speeds. The duration is also an eight bit data value that represents the number of cycles that the rover had been traveling at the motor speeds that were just stored. Finally, the top of the stack is again incremented.

Pop simply reverses the process of a push. First, the duration of the motor speeds is removed from the RAM and the top of the stack is decremented. Then, the motor speeds themselves are read from the RAM and the top of the stack is decremented again.

See Figure 3.

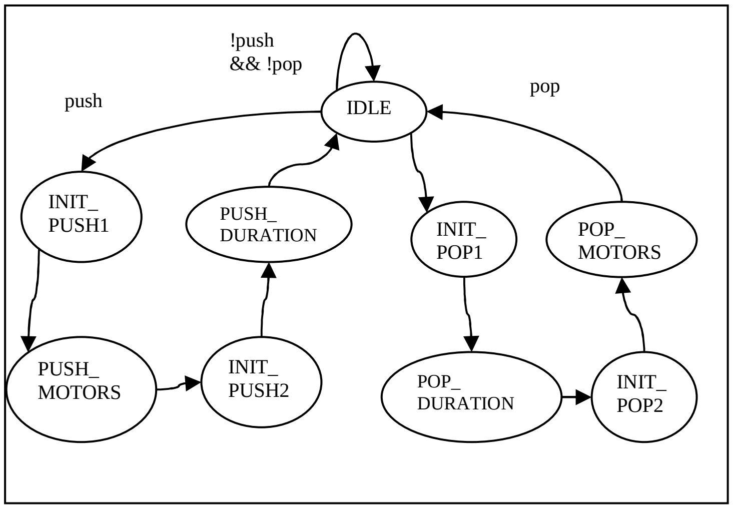


Figure 3: Diagram Stack FSM

Store Data Finite State Machine

The Store Data FSM is in charge of recording the history of the rover's movements. Each time start is set high for this FSM, the FSM checks if the rover's current speed is the same as the rover's previous speed. If it is, then the FSM simply increments a duration counter. Otherwise, the FSM initiates a push on the Stack FSM with the old motor speeds and the old duration counter as the values being pushed on the stack.

If the user has just requested a replay of history, then the FSM pushes the old motor speeds and the old duration counter right away. This is because the system is about ready to start a replay and needs to have the most current data stored in the RAM in order to accurately backtrack. For the sake of brevity I won't include a diagram of the FSM since it's fairly simple. It has an IDLE state, a DETERMINE_DATA state that figures out if information needs to be stored, and a WAIT_FOR_STORE state that waits for the memory to complete the storing process.

Replay Finite State Machine

The Replay FSM is in charge or playing back history for the user. When start is set high for this FSM, the FSM checks to see if the user has requested a replay or if a replay is already underway. If neither is the case, then the FSM goes back to IDLE.

If a replay is underway or is about to be started, the FSM checks a counter to see if it is 0 meaning the current set of motor speeds has been held for as long as it said in memory. If the counter is positive, then the FSM decreases the counter by 1. If the counter is 0, then another set of motor speeds and durations need to be popped from the stack. Consequently, the FSM moves to a state where it initiates a pop and waits for it to finish. Once finished, the FSM sets the counter to the duration retrieved from memory and the motor speeds to the speeds retrieved from memory.

Wireless Transceiver **(Ryan Damico)**

Data packet format

During nominal operation, the rover and base station are constantly sending information back and forth over the wireless link. This flow of data is responsible for regulating the rover's mechanical operation, as well as returning sensor data from the rover to the base station. Segmenting this data into packets in a way that is manageable and fault-tolerant ensures reliable operation of the rover.

There are two types of data packets the rover and base station use to communicate: command packets and senor packets. Command packets instruct the rover to operate in a specific way, controlling such parameters as motor speeds and state of the gripping claw. Sensor packets contain information gathered by the rover's onboard sensors, including ambient temperature and heading.

The size of both command packets and sensor packets is held constant at 5 bytes. This design parameter was chosen so that each packet is long enough to contain an entire rover command or set of sensor values. This makes each packet independent of the next, making the system fault-tolerant of dropped packets, a common problem in wireless systems. By sending command and sensor information in complete chunks, the loss of one or several packets will not affect the ability to decipher future packets. The rover and base station are designed to only update their systems on reception of a complete packet, so in the event of transmission problems, the rover and base station will simply maintain their configuration until a verified packet is received.

The command and sensor data packets are segmented as shown in Figure 4. The command data packet contains a complete snapshot of the rover's operation. The motor speeds are specified in sign-magnitude format (direction and speed), the search light is either on or off, the claw inclination is specified in magnitude, and the claw motion is either opening, closing, or idle. The sensor data packet contains the 8-bit magnitude value of the temperature sensor (in Celsius, so there is no reasonable need for a sign bit). Likewise the heading is an 8-bit representation of the rover's direction as read from a Hall-effect analog compass. It is noteworthy that the CC1010 inserts a 4-byte preamble at the beginning of each packet before transmission (the details of which are beyond the scope of this project). However it is important to take into consideration because though transparent to the FPGA, this virtually doubles the size of each packet, reducing how quickly they can be transmitted.

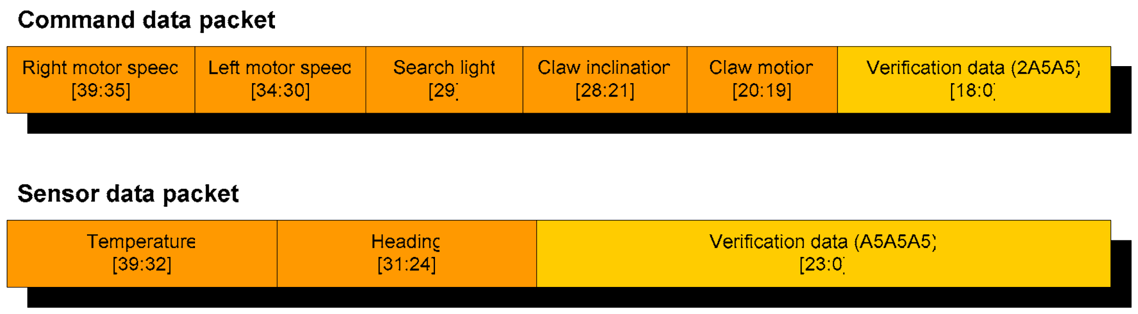


Figure 4: Command and sensor data packet breakdown

Space not used by actual data is consolidated into a verification segment, and is used to protect against certain errors. Because of the levels of abstraction between the CC1010 and FPGA, there are only two possible errors that can occur during data handling. One is that the packet is malformed or lost during physical transmission or reception, which is handled by the CC1010. The CC1010 includes automatic CRC error detection, and will not notify the FPGA of a received packet unless it knows it is intact. The other is that the FPGA encounters a sync error while loading the transmit buffer or downloading the receive buffer (causes include motor-induced noise or wire inductances). In this case it is possible that all or part of a packet is incorrectly received. Since packets are transmitted from MSB to LSB, a sync error will affect all lower-order bits from where the error occurred. Thus, by placing the verification data in this suspect region, the FPGA can be hard-coded to look for that value at the end of every packet. If there was a sync error, the FPGA will know and discard the data.

Transmit Interface Module

When the wireless transceiver executes a transmit command, it reads data from a buffer in RAM and broadcasts its contents to the other transceiver. Since this transmit buffer is located within the wireless transceiver but the data originates from the FPGA, the two units must transfer data before a transmission can occur. The transmit interface module is responsible for communication between the two.

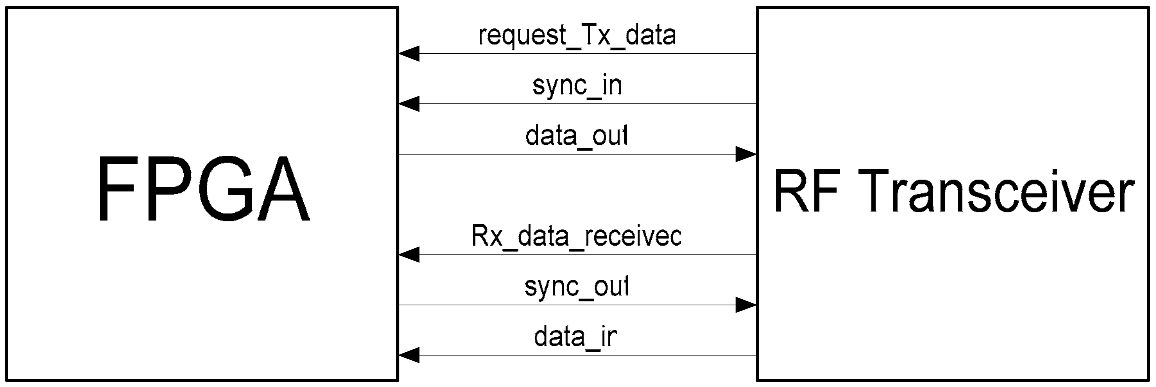


Figure 5: FPGA/Wireless transceiver interface

Data is transferred from the FPGA to the wireless transceiver's transmit buffer through a high speed synchronous serial connection. Three data paths between modules facilitate transfering of data: a transmit request line (request_Tx_data), a sync pulse line (sync_in), and a data line (data_out) (see Figure 5). Upon initiation of a transfer, the wireless transceiver sends a series of sync pulses to the FPGA. The FPGA in turn sends the data to the transmitter one bit at a time, updating at the edge of each sync pulse. This process lasts until all 40 bits of the 5-byte data packet have been uploaded (see Figure 6).

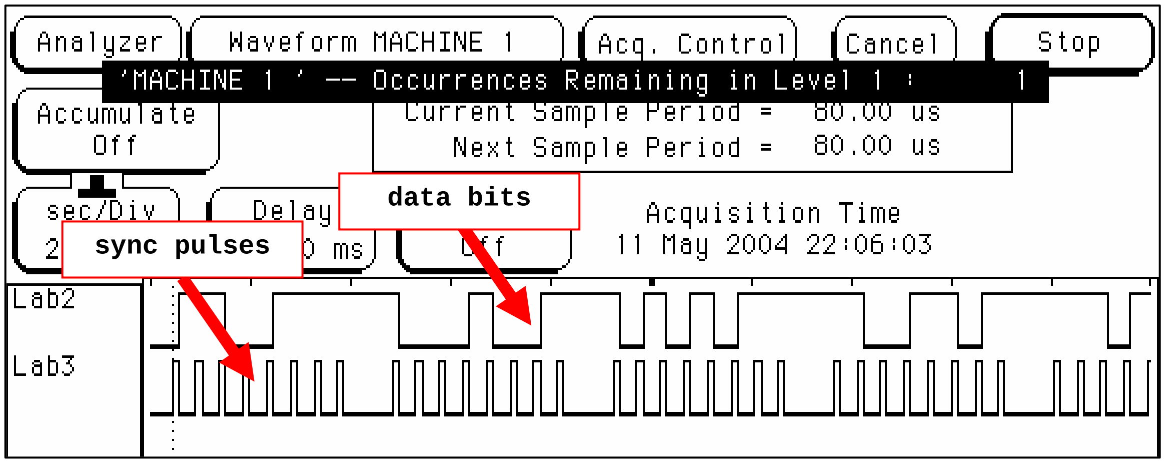


Figure 6: Logic analyzer capture of synchronous serial interface

The transmit interface module is controlled by a five-state FSM (see Figure 7). It begins in the *Idle* state, where the sync pulse counter is reset. The FSM then transitions into the *Wait start* state, looping back on itself until a request is received to upload data to the transmit

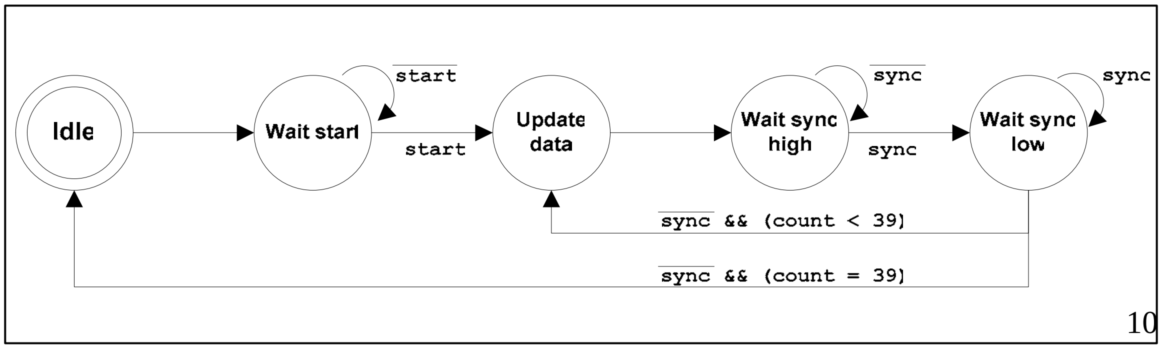

Idle
Wait start
$\overline{\text{start}}$
start
Update data
Wait sync high
$\overline{\text{sync}}$
sync
Wait sync low
sync
$\overline{\text{sync}}$ && (count < 39)
$\overline{\text{sync}}$ && (count = 39)

Figure 7: Transmit interface module FSM

buffer. When the request arrives via a pulse on the request_Tx_data input, the FSM transitions to the *Update data* state. Here the 40-bit reg containing the data packet is shifted to the left by one and the most significant bit is sent to the data_out output. After the data has been updated and the count incremented, the FSM continues to the *Wait sync high* state, followed by the *Wait sync low* state. These states wait until the negative edge of the next sync pulse before continuing operation. If there is still data left to load into the wireless transceiver (i.e. count < 39), the FSM returns to the *Update data* state and reiterates the loop. Otherwise the FSM returns to the *Idle* state, resets the count variable, and waits for the next transmit request.

Receive Interface Module

When the wireless transceiver receives a packet, it stores the incoming data into an onboard receive buffer stored in RAM. To capture this information and distribute it to the appropriate subsystems, the FPGA must download the data from the transceiver. The receive interface module is responsible for this function, using a high speed synchronous serial interface to download the data.

The operation of the receive interface module is analogous to the transmit interface module, and is connected to the wireless transceiver in a similar way (see Figure 5). Its FSM begins in an *Idle* state, resetting its counter and preparing to store the incoming 40 bits into two reg's (Max+PlusII allows no more than 32 bits to be explicitly stored in one reg). It then transitions into the *Wait packet received* state. Here the FSM loops back on itself until the Rx_data_received input is pulsed high. It then moves into the *Delay* state, waiting for 50 microseconds and generating the low half of the sync pulse. After the timer expires, the FSM transitions to the *Send sync* state. Here it sets the sync output high and waits an additional 50 milliseconds (the minimum time required for the wireless's microcontroller to pick up the signal). Once the timer expires, the FSM moves into the *Update data* state, acquiring the data bit from the wireless transceiver and shifting it into an internal reg. At this point the FSM will either transition back to the *Delay* state or the *Idle* state, depending if it has finished receiving all 40 bits.

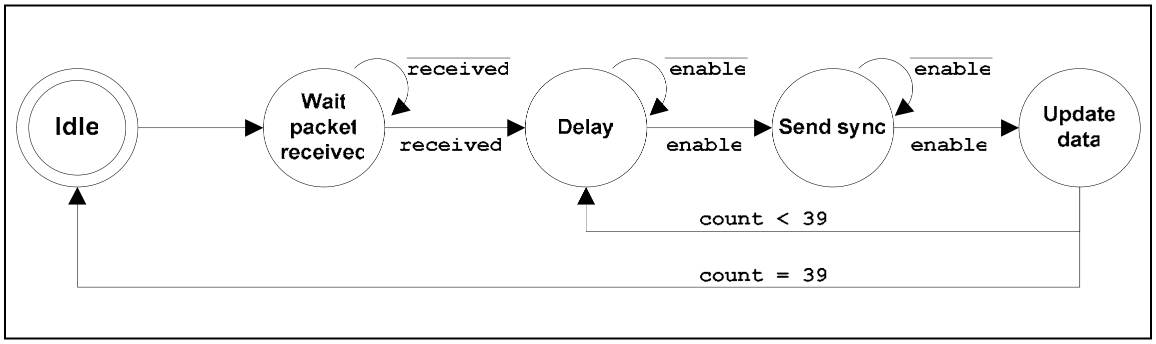


Figure 8: Receive interface module FSM

Design Methodology

Base Control Station

A balance needed to be maintained in the base control station between having a sampling rate that was too high or too low. If the sampling rate was too high, then the major FSM might not have enough time to finish all of its work. If the sampling rate was too low, then the rover might not seem responsive enough to what the user is doing with the PlayStation controller.

The original design of our project called for a PlayStation II controller. We felt that this controller's analog joysticks would provide better control over the rover's speed and direction. After many unsuccessful attempts to get the PlayStation II controller to work and a bit more research, we discovered that the PlayStation II controller expects a clock rate that is much higher than any clock found in our lab kits (at least 35 Mhz or so).

Since the old FPGA's can't handle clock rates much faster than 10 Mhz we decided to look into using a PlayStation I controller. Fortunately, the PlayStation I controller worked. There was still some trial and error to get the timing of commands and received data exactly right but we eventually arrived at a relatively stable configuration. See 0 for more information about the interface of the PlayStation controller.

One important design decision in this process was to include a timeout for the ack signal coming from the controller. We would have periodic problems where the ack signal wouldn't register in the FPGA and consequently the controller FSM would hang indefinitely. By including a timeout we were able to stop these periodic problems from causing the system to be completely unresponsive.

Wireless Interface

The design of the wireless rover incorporated many design practices and methodologies that were learned in labs one through three.

The wireless microcontroller was made the major FSM in an effort to modularize the system in a way that would allow reliable operation as well as fast development and debugging. By triggering the FPGA transmit and receive interfaces from the wireless microcontroller in a master/slave format, the Verilog modules were self-sufficient and did not need to be constantly reprogrammed. They were designed first, and most of the time after that was spent designing and debugging the wireless code on the microcontroller.

Serial transmission was chosen for loading data to and from the FPGA and wireless buffers to minimize the wiring and add complexity to the project. FSMs had to be written both in Verilog and C, which was a unique task and involved generalizing principles learned in 6.111 to environments outside of Verilog and Max+PlusII. The results were very satisfying, and worked out well.

Error detection was a major part of the design philosophy for the wireless rover. Adding support to detect occasional synchronization errors by comparing verification data worked well, and added reliability to the system. However, the most dangerous error proved to be when a missing synchronization bit caused either the FPGA or wireless microcontroller to stall because it was waiting for a nonexistent bit to arrive. Missing sync bits were very rare, but happened occasionally and caused the entire system to lock up. In the case of the transmit interface module, which waits for 40 sync pulses before finishing, a missing sync would cause it to get stuck in the *Wait sync high* state. This was corrected by having the wireless microcontroller hold the request_Tx_data signal high as long as it was send data to the FPGA. The transmit interface module was then modified to still start on the rising edge of the request_Tx_data signal, but reset on its negative edge. That way, even if it missed a sync pulse, the module would force a reset when the transceiver was done sending data. The analogous case for the microcontroller's receive code, which waits for 40 sync bits, involved setting interrupt handlers for watchdog timers, which forced a reset if all 40 bits were not received on time. In retrospect, implementing a digital system across two programming languages and processing environments was an incredible opportunity to generalize concepts learned throughout the course.

Testing

Base Control Station

The testing process for the base control station was relatively straightforward. We began working on the interface to the PlayStation controller right from the start. We first coded up the FSM for the controller. Before testing it on the FPGA we needed to make sure that appropriate signals would be generated and received appropriately. This was accomplished by simulating the FSM on MaxII+.

Once we were relatively certain the FSM was functioning properly, we set up a simple program that would just sample the controller continuously and would light up LEDs whenever certain buttons were pressed. At this point we started discovering how important pull-up resistors on two of the wires coming in from the controller were for any sort of consistent behavior of the system. After a lot of trial and error we finally found a somewhat stable combination of pull-up resistors that worked.

After testing the controller, we moved on to creating and testing the RAM interfaces. Before we tried to do anything too complex with the RAM we first tested the RAM itself using a program created in an earlier project. We did this as a sanity check to make sure the RAM was functioning properly and to make sure our assumptions about the timing constraints of the RAM were correct.

Once we were relatively certain the RAM was working, we started implementing the Stack, Replay, and Store_Data FSMs and the overall major FSM that would control everything. Admittedly here we skipped some physical testing of each FSM but we were able to successfully test each FSM on MaxII+. It took some time to figure out the exact control logic that was necessary to ensure that the replays would go all the way back to the beginning of memory, stop replaying, and then start sampling and storing controller data again.

Once we were relatively certain all of the FSMs were working properly we set up a main program that incorporated everything and loaded it onto the FPGA. To debug, our main program outputted speeds of the two motors as well as the current top of the stack on hex LEDs. This was so that we could tell if the correct speeds were being read and written to the correct addresses during storing and replay. Once we had everything working satisfactorily we were ready to incorporate the base station with the wireless module.

Wireless Interface

Testing was carried out by using the logic analyzer, status LEDs on the FPGA, and the wireless module's built-in interface to Telnet applications. The logic analyzer was used primarily to observe data transfers between the FPGA and wireless controller (see Figure 6), and helped spot an error in the receive interface module. It was behaving erratically, displaying different data every time it communicated with the wireless unit. On close inspection, the sync pulse train was seen to be "jumping" around, occasionally misaligning with the data bits. This helped determined that an error in the Verilog code prevented the sync pulse timer from resetting, causing the first pulse to have a random duration, skewing all future pulses. The visual indication of the LEDs was helpful in observing slow time-varying signals that were perceptible by the human eye. Examples included watching when the transmit or receive interface modules were invoked, or when a module got stuck in a certain state. It also helped verify the functionality of the receive interface module by slowly scrolling the 40 received bits after they were transferred, to verify the transaction completed properly.

The design and construction of the wireless rover was an excellent opportunity to apply the teachings and design practices learned in 6.111. Tackling design problems similar to real-world situations was an invaluable experience that will undoubtedly benefit future digital design projects.

Appendix

Selected Verilog Code

Base Control Station

Excerpts from controller_fsm.v

Here is an example of how we received button data from the controller. Enable_clock is a signal that enables the clock that is output to the controller. The clock signal only goes to the controller when a data byte is being sent or received.

```
assign select = data_receive_1[0];
assign start = data_receive_1[3];
assign up = data_receive_1[4];
assign right = data_receive_1[5];
assign down = data_receive_1[6];
assign left = data_receive_1[7];

…

ROUND4:
begin
       if(count < 7)
       begin
              next = ROUND4;
              count_int = count + 1;
              enable_clock_int = 1;
              case(count)
              0:
                     data_receive_1_int[0] = data_receive;
              1:
                     data_receive_1_int[1] = data_receive;
              2:
                     data_receive_1_int[2] = data_receive;
              3:
                     data_receive_1_int[3] = data_receive;
              4:
                     data_receive_1_int[4] = data_receive;
              5:
                     data_receive_1_int[5] = data_receive;
              6:
                     data_receive_1_int[6] = data_receive;
              default
                     data_receive_1_int = data_receive_1;
              endcase
       end
       else if(ack && count == 7)
       begin
              data_receive_1_int[7] = data_receive;
              count_int = count + 1;
              next = ROUND4;
              enable_clock_int = 1;
       end
       else if(count == 7)
       begin
              data_receive_1_int[7] = data_receive;
              count_int = 0;
              next = WAITFORACK4;
              enable_clock_int = 0;
       end
       else if(ack)
       begin
              count_int = count;
              next = ROUND4;
              enable_clock_int = 0;
       end
       else
       begin
              next = WAITFORACK4;
              count_int = 0;
              enable_clock_int = 0;
       end
end
```

Excerpts from store_data.v

Here is the section of code that deals with determining if data should be stored and if so, what.

```
DETERMINE_DATA:
begin
       if({right_motor, left_motor} == data_motor_prev)
```

```
        begin
                if(!replay)
                begin
                        next = IDLE;
                        write_data_length_int = write_data_length + 1;
                end
                else
                begin
                        data_motor_prev_int = 0;
                        write_data_length_int = write_data_length + 1;
                        write_data_motor_int = data_motor_prev;
                        push_int = 1;
                        next = WAIT_FOR_STORE;
                end
        end
        else
        begin
                if(!replay)
                begin
                        data_motor_prev_int = {right_motor, left_motor};
                        write_data_motor_int = data_motor_prev;
                        push_int = 1;
                        next = WAIT_FOR_STORE;
                end
                else
                begin
                        data_motor_prev_int = 0;
                        write_data_motor_int = {right_motor, left_motor};
                        write_data_length_int = 1;
                        push_int = 1;
                        next = WAIT_FOR_STORE;
                end
        end
end
```

Data on the PlayStation Interface

This information is taken from http://www.gamesx.com/controldata/psxcont/psxcont.htm.

The Playstation Controller Pinouts

```
        LOOKING AT THE PLUG
        -------------------------------
 PIN 1->| o  o  o | o  o  o | o  o  o |
        \_____________________________/

PIN # USAGE

   1.  DATA
   2.  COMMAND
   3.  N/C (9 Volts unused)
   4.  GND
   5.  VCC
   6.  ATT
   7.  CLOCK
   8.  N/C
   9.  ACK


DATA
        Signal from Controller to PSX.
        This signal is an 8 bit serial transmission synchronous to the falling edge of
        clock (That is both the incoming and outgoing signals change on a high to low
        transition of clock. All the reading of signals is done on the leading edge to
        allow settling time.)
COMMAND
```

Signal from PSX to Controller.
This signal is the counter part of DATA. It is again an 8 bit serial transmission on the falling edge of clock.

VCC

VCC can vary from 5V down to 3V and the official SONY Controllers will still operate. The controllers outlined here really want 5V.
The main board in the PSX also has a surface mount 750mA fuse that will blow if you try to draw to much current through the plug (750mA is for both left, right and memory cards).

ATT

ATT is used to get the attention of the controller.
This signal will go low for the duration of a transmission. I have also seen this pin called Select, DTR and Command.

CLOCK

Signal from PSX to Controller.
Used to keep units in sync.

ACK

Acknowledge signal from Controller to PSX.
This signal should go low for at least one clock period after each 8 bits are sent and ATT is still held low. If the ACK signal does not go low within about 60 us the PSX will then start interrogating other devices.

It should also be noted that this is a bus of sorts. This means that the wires are all tied together (except select which is seperate for each device). The DATA and ACK pins can be driven from any one of four devices. To avoid contentions on these lines they are open collectors and can only be driven low. If used with an FPGA DATA and ACK must use a pull-up resistor.

The PSX Controller Signals

All transmissions are eight bit serial LSB first. All timing in the PSX controller bus is synchronous to the falling edge of the clock. One byte of the transmissions will look like this.

```
               |BIT 0|BIT 1|BIT 2|BIT 3|BIT 4|BIT 5|BIT 6|BIT 7|
    CLOCK -----___---___---___---___---___---___---___---___-----------

    DATA  -----000000111111222222333333444444555555666666777777--------
                  *     *     *     *     *     *     *     *
    CMND  -----000000111111222222333333444444555555666666777777--------

    ACK   ----------------------------------------------------------__-
```

The logic level on the data lines is changed by the transmitting device on the falling edge of clock. This is then read by the receiving device on the leading edge (at the points marked *) allowing time for the signal to settle. After each COMMAND is received by a selected controller, that controller needs to pull ACK low for at least one clock tick. If a selected controller does not ACK the PSX will assume that there is no controller present.

When the PSX wants to read information from a controller it pulls that devices ATT line low and issues a start command (0x01). The Controller Will then reply with its ID (0x41=Digital). At the same time as the controller is sending this ID byte the PSX is transmitting 0x42 to request the data. Following this the COMMAND line goes idle and the controller transmits 0x5A to say "here comes the data".

This would look like this for a digital controller

```
ATT    -______________________________________________________________
              |   Byte 1     |    |   Byte 2     |    |   Byte 3     |
CLOCK  ---_-_-_-_-_-_-_-_-----_-_-_-_-_-_-_-_-----_-_-_-_-_-_-_-_-----
                   0xFF                0x41                0x5A
DATA   -------------------------__________--__----__--__----__--__----
                   0x01                0x42
CMND   -----_____________-----__--________--__------------------------

ACK    --------------------__-------------------__-----------------__-
```

```
After this command initiation proccess the controller then sends all its data bytes (in
the case of a digital controller there is only two). After the last byte is sent ATT will
go high and the controller does not need to ACK.

The data transmision for a digital controller would look like this (where
A0,A1,A2...B6,B7 are the data bits in the two bytes).

    ATT    _______________________________________-------
              |   Byte 4     |    |   Byte 5     |
    CLOCK  ---_-_-_-_-_-_-_-_-----_-_-_-_-_-_-_-_--------

    DATA   ---D0D1D2D3D4D5D6D7----E0E1E2E3E4E5E6E7-------

    CMND   ----------------------------------------------
                                                 ***
    ACK    --------------------__------------------------

    NOTE: No ACK.
```

The PSX Controller Data

```
    Standard Digital Pad

    BYTE    CMND    DATA

     01     0x01    idle
     02     0x42    0x41
     03     idle    0x5A    Bit0 Bit1 Bit2 Bit3 Bit4 Bit5 Bit6 Bit7
     04     idle    data    SLCT           STRT UP   RGHT DOWN LEFT
     05     idle    data    L2   R2    L1  R1   /\   O    X    |_|

    All Buttons active low.
```

Wireless Interface

Pulse-Width Modulation code

Useful code for controlling the speed of DC motors. Use with H-bridges to add direction control. Can be modified to control position of servos (see comments).

```
module PWM (clk, reset, duty_cycle, enable, LED_status, LED_2status, d_enable);
  input clk, reset;
  output [3:0] LED_status;
  output d_enable;
  reg d_enable;
  reg [3:0] LED_status;
  input [3:0] duty_cycle; // 4 bits of speed control (percentage)... could be increased
later
  output enable, LED_2status;
  reg [19:0] count;
  reg enable_int;

  assign LED_2status = enable;
  assign enable = enable_int;

  always @ (posedge clk) begin
    LED_status = duty_cycle;
    if (reset == 1) // no reset
      count <= 0;  // for fast simulation
    else if (count == 62) begin // 62 -> about 30 KHz for motors; 36864 -> 50 Hz for
servo
      d_enable = 1;
      count <= 0;
      if (duty_cycle != 0)  // don't even start the pulse if the duty cycle is 0
        enable_int <= 1;
    end
    else begin
      d_enable = 0;
      count <= count + 1;                  //60/15
      if ( (enable == 1 & count <= duty_cycle*(4) ) | duty_cycle == 15 )
        enable_int <= 1;
      else
        enable_int <= 0;
    end
  end
endmodule
```

Code to load the CC1010 wireless microcontroller with data from an FPGA

Serial transmission follows protocol described earlier.

```
void load_Tx_buffer() {
       int count = 0;
       int Tx_buffer_temp = -1;
       printf("LOAD_TX_BUFFER() CALLED\n");

       // reset buffer
       for (n = 0; n < TEST_STRING_LENGTH; n++) {
              Tx_buffer[n] = 0x00;
       }

       YLED = LED_ON;

       PORTBIT(0,0) = 1;
       halWait(1, CC1010EB_CLKFREQ);
       PORTBIT(0,0) = 0;

       for (n = 0; n < TEST_STRING_LENGTH; n++) {
              for (count=7; count>=0; count--) {
                     Tx_buffer[n] += (0x11 & PORTBIT(2,0)) << count;
         Tx_buffer_temp = PORTBIT(2,0);
         printf("%d ", Tx_buffer_temp); // read and display data *takes a LONG TIME if
enabled!
//                   halWait(1, CC1010EB_CLKFREQ);
                     PORTBIT(1,2) = 1;                                          //
ready_next = 1
//                   halWait(100, CC1010EB_CLKFREQ);
                     PORTBIT(1,2) = 0;                                          //
ready_next = 0
//                   halWait(100, CC1010EB_CLKFREQ);
              }
              printf("> data byte %d = %X\n", n, Tx_buffer[n]); // read and display data
*takes a LONG TIME if enabled!
       }
       YLED = LED_OFF;


       printf("\n");
       // transmit();
}
```

Top level instantiations for transmit and receive interface modules

Demonstrates simple unpacking of command data packet, minus verification

```
module  tx_rx_top  (clk,  ready_next,  data_out,  sync_pulse,  start,  reset,  Rx_received,
data_in,  // need
                              //latched_data,  // latched Rx data
                              LED_I, d_count, //busy, LED_data,  // don't need
                              RM_on,  RM_dir,  LM_on,  LM_dir,  claw_on,  claw_dir,  angle_on,
light  // motor, etc. values
                         );
```

```
output RM_on, RM_dir, LM_on, LM_dir, claw_on, claw_dir, angle_on, light;

input clk, ready_next, start, reset;
output LED_I;
output data_out, sync_pulse;
input Rx_received, data_in;
reg data_out, sync_pulse;
output [3:0] d_count;
reg [39:0] latched_data;


tx_buffer_v   tx_buffer_v   (.clk(clk),   .ready_next(ready_next),   //.d_count(d_count),
//.LED_II(LED_II), .LED_III(LED_III),
                                       .data_out(data_out), .start(start), .reset(reset),
                                       .data(32'h6789AB),  .data_II(32'hCDEF)  //  example
values (shows ordering between reg's)
);

rx_buffer_v    rx_buffer_v    (.clk(clk),    .sync_pulse(sync_pulse),    .LED_I(LED_I),
.LED_data(LED_data),
                                              .data_in(data_in),
.Rx_received(Rx_received), .reset(reset), .latched_data(latched_data)
                                             );
assign d_count = latched_data[24:21];
assign angle_on = (&latched_data[23:16]);
assign claw_dir = (&latched_data[15:8]);
assign claw_on = (&latched_data[7:0]);
assign light = (|latched_data[39:32]);
assign LM_dir = (|latched_data[31:24]);
assign RM_on = (|latched_data[23:16]);
assign RM_dir = (|latched_data[15:8]);
assign LM_on = (|latched_data[7:0]);

endmodule
```

FSM from receive interface module

Demonstrates shifting data to match incoming bitstream. Contains code to visually verify the data after transmission (must be enabled).

```
case (state)
    IDLE: begin
         count_int = 0;
         data_I_int  = 32'h00000000; // initial string
         data_II_int = 32'h00000000; // initial string
         next = WAIT_RX_RECEIVED;
    end

    WAIT_RX_RECEIVED: begin
         LED_I = 1;
         if (Rx_received_s) begin // wait for an incoming packet
           next = DELAY;
      end else begin
           count_int = 0; // set to zero in case of sync error
        div_reset = 1;
           next = WAIT_RX_RECEIVED;
         end
    end

    UPDATE_DATA: begin // update data bit to be transmitted
         if (count <= 31) begin      // first piece of data stream
           data_I_int = {data_I[30:0], data_in_s};
         end else begin
           data_II_int = {data_II[30:0], data_in_s};
         end
      next = DELAY;

         if (count == 40) begin // end of data stream:
           test_int = {data_I, data_II[7:0]};
```

```
          latched_data_int = test_int; // latch new value of data
          next = IDLE;
        end
        else
          count_int = count + 1;
  end

  DELAY: begin
    if (enable)
      next = SEND_SYNC;
    else
      next = DELAY;
  end

  SEND_SYNC: begin // wait for ready_next signal from microcontroller
    sync_pulse_int = 1;
    if (enable)
      next = UPDATE_DATA;
    else
      next = SEND_SYNC;
  end

      DISPLAY_DATA: begin
        if (Rx_received_s) begin
          test_int = test >> 4;
          LED_I = ~LED_I;
          next = DISPLAY_DATA_I;
        end else
          next = DISPLAY_DATA;
      end

      DISPLAY_DATA_I: begin
        if(!Rx_received_s)
          next = DISPLAY_DATA;
        else
          next = DISPLAY_DATA_I;
      end

  default:
    next = IDLE;
endcase
```