Harvey Mudd College

Engineering 155
Microprocessor Systems: Design and Application

Internet-Controlled AGV

Authors:
Aaron Rosen
Alex Rich

Professors:
David Harris
Matthew Spencer

December 11, 2015

Abstract

1 Introduction

The Internet-Controlled Autonomous Ground Vehicle is a treaded chassis that holds an FPGA with bluetooth listening capabilities. A companion website allows a user to interact with the vehicle by commanding it to travel it to various places. The motivation behind this project was to build a tank that could navigate within a map and shoot NERF darts. By building a tank that is remotely controlled over the internet, the more difficult part of this concept is realized.

The project involves a website hosted by Apache2 on the Raspberry Pi 2, which sends commands from the a bluetooth dongle to a bluetooth receiver (BlueSMiRF) that is connected to the FPGA. The FPGA drives two motors. Each component is described further in the following sections.

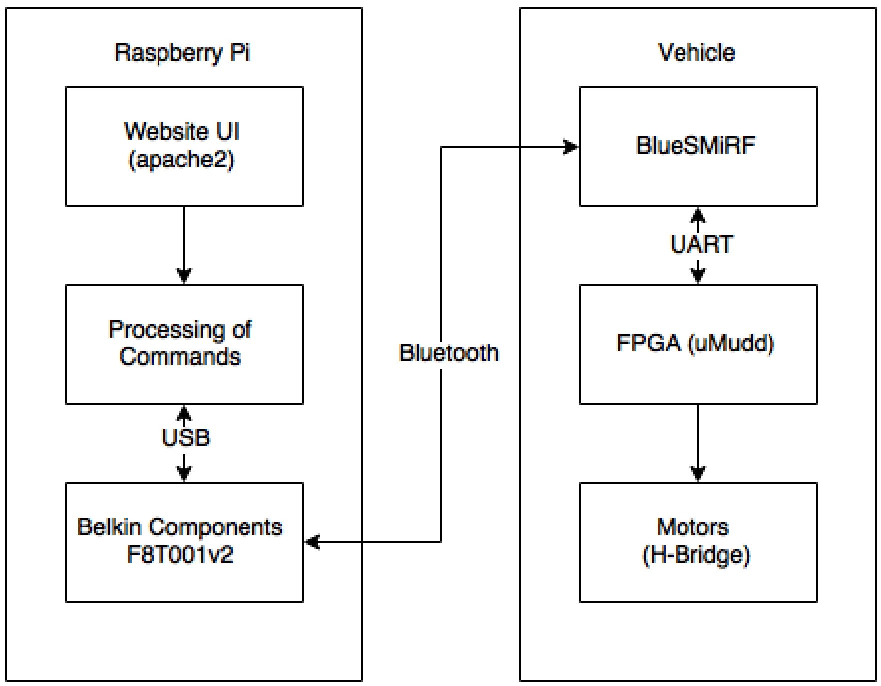

Figure 1: An overview of the system. The system is comprised of two major subsystems: the Raspberry Pi 2 controller and the vehicle, which is controlled by the μMudd board.

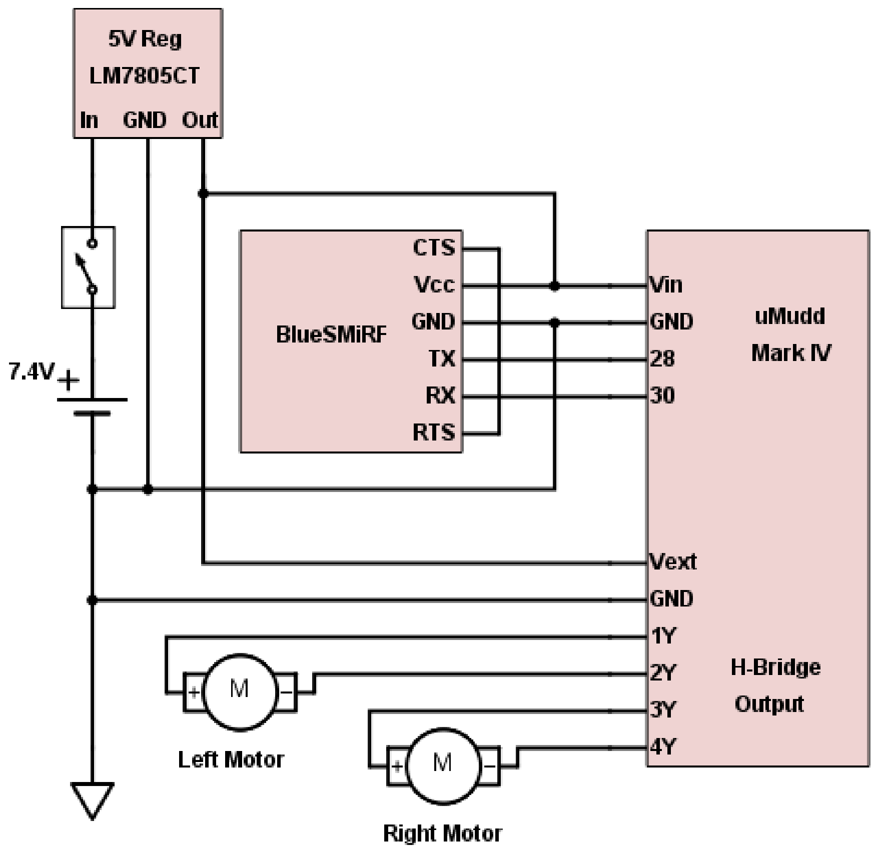

Figure 2: The components on the breadboard. The H-Bridge Output are PWM signals.

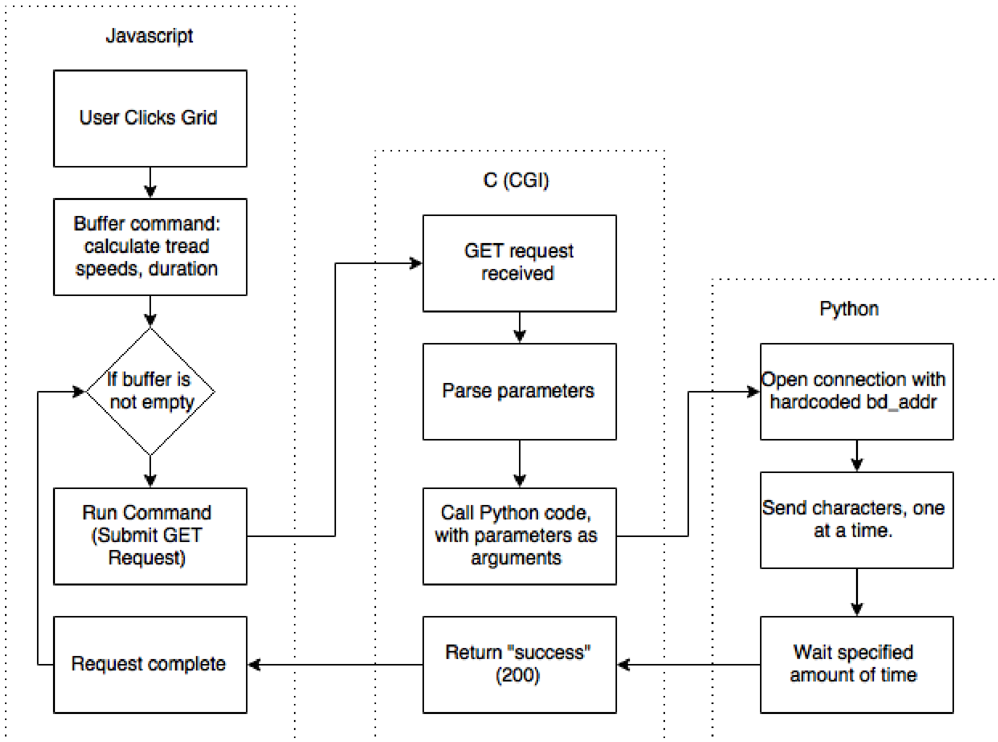

Figure 3: The flow of control and data through the Pi.

2 New Hardware

2.1 BlueSMiRF

2.2 Vehicle

3 Schematics

4 Raspberry Pi

The Raspberry Pi 2 serves several functions. It hosts an Apache2 website accessible via the internet and contains code to submit commands to a bluetooth dongle. These two functions are integrated such that a user interacts with the website, indirectly sending commands over bluetooth. Figure 3 show the flow of data and control through the Pi.

4.1 Website

The website contains a visual user interface that contains instructions for use, a grid on which to input locations for the robot to maneuver to, a list of the commands currently buffered for sending. The code for the website is shown in Appendix A. The website was built using HTML, JavaScript, and Bootstrap CSS.

The Pi hosts the website. Upon clicking in a grid space, the page's JavaScript calculates the left and right tread speeds and duration of movement required to get the tank to move from its original position to the new position. The current algorithm involves first moving north/south, turning, moving east/west, and finally turning to reorient itself vertically. Once the commands are generated, the JavaScript makes an HTTP GET request to the inputChars resource of the Pi. When the request is completed, the page updates, either submitting the next command in the buffer or waiting for another input from the user.

4.2 Python/C

After receiving a GET request, the common gateway interface (CGI) reads the input parameters (three integers) and converts them into a format that the vehicle understands. This involves converting the numbers to sign/magnitude bit representations. Because of how the UART works, the C also reverses the bits.

calls a Python script. The Python script utilizes the `bluetooth` module to allow sending data using the bluetooth dongle. Since the robot has a constant bluetooth device address, this address is hard coded into the Python script. When called, the python script sends the commands, one at a time, to the robot. Currently, the system sleeps for an amount of time to theoretically give the vehicle enough time to move. However, in the future, the Pi will wait for acknowledgement from the FPGA. The code on the Pi is shown in Appendix B.

// Depending on room, I can include the code used to call Python, since that seemed to be a common problem.

5 FPGA Design

The FPGA reads data from the BlueSMiRF using UART hardware coded in SystemVerilog, processes and executes the command, and then sends an acknowledgment back to the Raspberry Pi. It is constructed as a controller-datapath pair with three main submodules in the datapath - receiveMSG, executeCommand, and sendAck. The SystemVerilog code installed on the FPGA is shown in Appendix C. The FPGA and BlueSMiRF are the only two electrical components on the breadboard. Two motors are connected to the μMudd board's H-bridge screw terminals. The schematic is shown in figure.

The clock used to interface with the BlueSMiRF is implemented using a PLL that oversamples the 115.2 Kbaud UART frequency at 921.6kHz, or a factor of 8. This oversampler determines if there is an incoming message. The actual sampling of the BlueSMiRF's TX line is accomplished using a frequency divider that allows for sampling at the correct rate. The divider's phase can be frozen when the start bit has not yet been detected. This ensures that the sampling of the line is as close to the center of the transmission's clock as possible. The Raspberry Pi sends three characters, which are flushed by the Bluetooth module's buffer at the same time, so the command appears as a 30-bit message. The sampler stops sampling when it sees a stop bit, and either begins a new message if the next bit is a start bit, or signals to the controller that a command has been received of an entire command if the line has remained high.

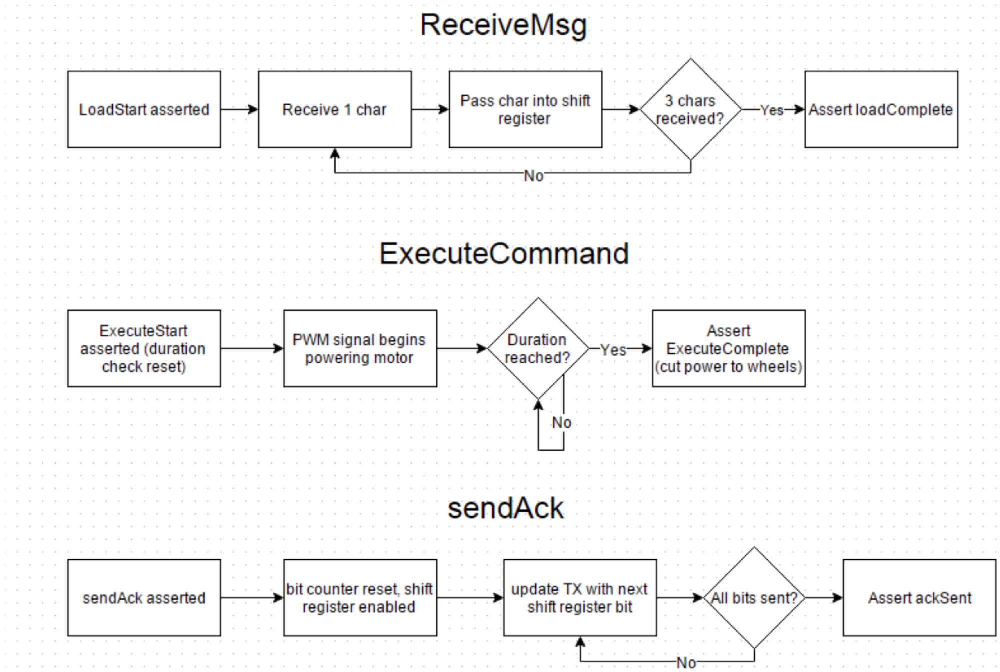

Figure 4: Schematic overview of the FPGA.

The FPGA executes the command received by controlling the two motors via the H-Bridges on the μMudd board. Each command consists of a PWM setting and rotation direction for each motor and a duration for which the motors should be turning. A counter is used to create a reference clock for PWM; the power levels are referenced against this counter to determine the correct duty cycle. Multiplexers are used to route the power to the correct pins on the H-Bridge, allowing for both forward and backwards movement. To prevent the vehicle from running indefinitely, the timer stops incrementing when the requested duration is reached, and a signal is output that is used to cut power to the motors. Each LSb of the duration character corresponds to roughly one-tenth of a second.

Once the requested duration has been reached and power to the motors cut, the FPGA transmits the character 'A' back to the BlueSMiRF as an ACK code. After this ACK has been sent, the FPGA will return to the receiveMSG state and start to sample for a new command.

A flowchart detailing each state of the FPGA is shown in Figure 4.

6 Results

7 References

[1] REFERENCES?

8 Parts List

This is the bill of materials for the project.

Item	Description	Source	Cost
Tracked Vehicle Chassis Kit	Chassis for the tank, includes treads and frame.	Amazon	15.39
Tamiya 70168 Double Gearbox	Gives tank flexibility to turn by controlling each tread independently.	Amazon	11.99
μMudd Board	Controls the vehicle.	E155	0.00
Raspberry Pi 2	Provides website interface and sends commands to vehicle.	E155	0.00
BlueSMiRF	Wirelessly communicate via Bluetooth between Pi and μMudd board.	E155	0.00
Belkin Components FT8001	Send bluetooth data from Pi.	E155	0.00
2X TrustFire 14500	Li Ion battery, 3.7 V	Aaron Rosen	0.00
			27.38

9 Appendices

9.1 FPGA Code

VehicleControl.sv

```
//Aaron Rosen and Alex Rich
//E155 Final Project

module VehicleControl(input logic clk,
                      input logic RX,
                      output logic TX,
                      output logic [1:0] HL,HR,
                      output logic HbridgeEN,
                      output logic [2:0] state,
                      output logic [7:0] char,
                      output logic sampler, loadComplete
```

```
                                                    ,loadStart ,
                                                    RXdone,
                                                    ackSent);

        //top level module
        logic[7:0] lmotor,rmotor,dur;
        //logic loadStart,loadComplete;
        logic executeStart,executeComplete; //executeStart
           should pulse when starting rather than staying high
        logic ackStart;
        logic pllclk;
        logic reset,locked;
        logic[1:0] HLled,HRled;

        assign HbridgeEN = 1;
        assign HLled = HL;
        assign HRled = HR;
        assign char = lmotor;

        PLLclk2 pll(reset,clk,pllclk,locked); //sampler/UART
           clk

        controller control(clk,loadComplete,executeComplete,
           ackSent,loadStart,executeStart,ackStart,state); //
           datapath controller

        receiveMSG RXin(clk,pllclk,RX,loadStart,lmotor,rmotor,
           dur,loadComplete,sampler,RXdone); //UART msg Receive
        executeCommand executor(clk,(loadComplete |
           executeStart),(~executeComplete & loadStart),lmotor,
           rmotor,dur,executeComplete,HL,HR); //powertrain
           control
        sendAck TXout(pllclk,ackStart,ackSent,TX); //UART msg
           transmit

endmodule

module controller(input logic clk,
                                        input logic
                                           loadComplete,
                                        input logic
                                           executeComplete,
                                        input logic ackSent,
                                        output logic loadStart,
                                        output logic
                                           executeStart,
                                        output logic ackStart,
```

```
                                        output logic[2:0] outputState);
    //datapath controller
    reg[2:0] state;
    assign outputState = state;
    logic execute,executeDelayed;
    always_ff @(posedge clk)
        begin
            case(state)
                3'b001: begin
                            if(loadComplete)
                                state<=3'b010; //state becomes execute
                            else
                                state<=3'b001; //state remains load
                        end
                3'b010: begin
                            if(executeComplete)
```

```
state <=3'b100; // state becomes ack
else
state <=3'b010; // state remains execute
end
3'b100: begin
if(ackSent)
state <=3'b001; // state becomes load
else
```

```
                                                            state
                                                            <=3'
                                                            b100
                                                            ; //
                                                            state

                                                            remains
                                                             ack
                                                    end
                                  default:

                                          state<=3'b001; //default to
                                    load
                            endcase
                      end
      assign {ackStart,execute,loadStart}=state; //delegate signals
         accordingly
      flop executeDelay(clk,execute,executeDelayed);
      assign executeStart = execute & ~ executeDelayed;
endmodule

module receiveMSG(input logic clk,PLLclk,
                                    input logic RX,loadStart,
                                    output logic[7:0] lmotor,rmotor,dur,
                                    output logic loadComplete,sampler,
                                       RXdone);
      //top level for message recive subsystem
      logic[7:0] char;
      logic[15:0]idle;
      logic shiftSig;
      UARTRX receiveChar(clk,PLLclk,RX,loadStart,char,RXdone,sampler)
         ;
      pulse receivedChar(clk,RXdone,shiftSig);
      shift16 samplerIdle(sampler,PLLclk,idle);
      assign loadComplete = loadStart & idle[15] & ~(|idle[14:0]);
      always_ff @(posedge shiftSig)
            begin
                  if(loadStart) {lmotor,rmotor,dur}={rmotor,dur,
                     char};
            end
endmodule

module executeCommand(input logic clk,
                                                      input logic
                                                         resetDur,
```

```
                                                  presetDur ,
                                                input logic
                                                  [7:0] lmotor
                                                  , rmotor , dur
                                                  ,
                                                output logic
                                                  executeComplete
                                                  ,
                                                output logic
                                                  [1:0] HL,HR
                                                  );
    //top level for message execute subsystem
    logic LPWM,RPWM;

    durcheck #(30) duration(dur, clk, resetDur, presetDur,
       executeComplete);
    pwm lmotorPWM(lmotor[6:0], clk, resetDur,LPWM);
    pwm rmotorPWM(rmotor[6:0], clk, resetDur,RPWM);
    hBridgeIn LHbridge(LPWM, executeComplete, lmotor[7],HL);
    hBridgeIn RHbridge(RPWM, executeComplete, rmotor[7],HR);
endmodule

module sendAck(input logic clk,
                             input logic ackStart,
                             output logic ackSent,
                             output logic TX);
    //top level for ack send subsystem
    UARTTX sendChar(clk, ackStart,TX, ackSent);

endmodule

module UARTTX(input logic clk,
                         input logic ackStart,
                         output logic TX,
                         output logic msgSent);
    //UART TX Pin
    //ACK is "A" (0x41), msg is 11'b0_0100_0001_11 = 11'
       b001_0000_0111 = 11'h107
    //TODO: Change shift register to more directly include TX
    logic resetTrigger;
    logic ackStartPulse;
    logic clk2;
    logic [10:0] msg;
    logic [3:0] count; //keeps track of the number of bits sent
    slowclk baudrate(clk,1'b1,clk2);
    always_ff @(posedge clk2)
          begin
```

```
                if(ackStart) {msg[9:0],msg[10]}={msg[10:0]}; //
                    this is an 11-bit shift register
                else {msg[9:0],msg[10]} = {10'h107,1'h0};
                        //reset message loop to default
                    position
            end
    assign TX = msg[0];
    timeren #(4) bitCount(clk2,resetTrigger,ackStart,count);
    assign msgSent = (count == 4'hB) & ackStart; //message send
        high after 11th bit sent
    delay #(1) resetSig(clk,(msgSent|ackStartPulse),resetTrigger);
    pulse ackPulse(clk,ackStart,ackStartPulse);

endmodule

module UARTRX(input logic clk,PLLclk,
                    input logic RX,
                    input logic loadStart,
                    output logic[7:0] char,
                    output logic done,UART);
    //UART RX Pin
    logic[3:0] validCheck;
    logic valid;
    logic UARTclk;
    logic stopBit;
    logic resetTrigger;
    logic startBit;
    logic loadStartDelayed,doneDelayed;
    assign UART = UARTclk;

    always_ff @(posedge clk,posedge done)
            begin
                if(done) valid <= 0;
                else valid <= valid | (~|(validCheck));
            end
    shift4 sampler(RX,PLLclk,validCheck);
    slowclk baudrate(PLLclk,valid,UARTclk);
    always_ff @(posedge UARTclk,posedge resetTrigger)
            begin
                if(resetTrigger) {done,startBit,char,stopBit} =
                    11'h001;
                else {done,startBit,char,stopBit}={startBit,
                    char,stopBit,RX}; //this is an 12-bit shift
                    register
            end
    delay2 #(1) doneDelay(PLLclk,done,doneDelayed);
    delay #(1) loadStartDelay(clk,loadStart,loadStartDelayed);
```

```
        assign resetTrigger = doneDelayed | (~loadStartDelayed &
           loadStart);
endmodule

module hBridgeIn(input logic pwr,done,direction,
                                output logic[1:0] out);
        //cuts power to H-Bridge when done is asserted
        logic[1:0] sig;
        assign sig[0]=0;
        assign sig[1] = pwr & ~done;
        assign out = direction?{sig[0],sig[1]}:sig; //direction is sign
            in sign/mag number
endmodule

module pwm(input logic [6:0] power,
                input logic clk,reset,
                output logic wave);
        //Takes in an input signal and outputs corresponding PWM signal
        logic [6:0] count;
        timer #(7) pwmTimer(clk,reset,count);
        assign wave = (power > count);
endmodule

module durcheck #(parameter WIDTH=30)
                                (input logic[7:0] dur,
                                input logic clk,reset,preset,
                                output logic done);
        //checks the duration and cuts power to the wheels when done
        logic[WIDTH-1:0] durTime;
        always_ff @(posedge clk,posedge reset)
                begin
                        if(reset) durTime <=0;
                        else if(preset) durTime <= {dur,{WIDTH-8{1'b0
                           }}};
                        else if(done) durTime <= durTime;
                        else durTime <= durTime + 1'b1;
                end
        assign done = (dur == durTime[WIDTH-1:WIDTH-8]);
endmodule

module shift3rst(input logic in,clk,reset,
                        output logic[2:0] out);
        //3-register shift register with reset
        logic c,d,e;
        always_ff @(posedge clk,posedge reset)
                if(reset)
                        begin
```

```
                        c <=0;
                        d <=0;
                        e <=0;
                end
        else
                begin
                        c<=in;
                        d<=c;
                        e<=d;
                end
    assign out = {e,d,c};
endmodule

module shift4(input logic in,clk,
                  output logic[3:0] out);
    //4-register shift register, outputs all shifted bits
    always_ff @(posedge clk)
        begin
            out[0]<=in;
            out[1]<=out[0];
            out[2]<=out[1];
            out[3]<=out[2];
        end
endmodule

module shift16(input logic in,clk,
                  output logic[15:0] out);
    //4-register shift register, outputs all shifted bits
    always_ff @(posedge clk)
        begin
            out <= {out[14:0],in};
        end
endmodule

module slowclk(input logic clk,valid,
                   output logic clk2);
    //creates a second slow timer that is reliant on valid for
       centering
    logic [2:0] count;
    always_ff  @(posedge clk)
        begin
            if(valid) count <= count + 3'h1; //if the
                signal is valid, increment the counter
                normally
            else
                begin  //if the signal is not valid,
                    hold the slow clock right before the
```

```
                                        transition
                                            count[2] <= 0;
                                            count[1] <= 1;
                                            count[0] <= 1;
                                   end
                  end
         assign clk2=count[2];
endmodule


module timer #(parameter WIDTH=8)
                            (input logic clk,reset,
                             output logic [WIDTH-1:0] timeout);
         //a WIDTH-bit timer
         always_ff @(posedge clk,posedge reset)
                  begin
                           if(reset) timeout <= 0;
                           else timeout <= timeout + 1'b1;
                  end
endmodule

module timeren #(parameter WIDTH=8)
                            (input logic clk,reset,enable,
                             output logic [WIDTH-1:0] timeout);
         //a WIDTH-bit timer with enable
         always_ff @(posedge clk,posedge reset)
                  begin
                           if(reset) timeout <= 0;
                           else if(enable) timeout <= timeout + 1'b1;
                  end
endmodule

module flop #(parameter WIDTH=1)
                                    (input logic clk,
                                     input logic [WIDTH-1:0] d,
                                     output logic [WIDTH-1:0] q);
         always_ff @(posedge clk)
                  begin
                           q <= d;
                  end
endmodule

module flopen #(parameter WIDTH=1)
                                    (input logic clk,en,
                                     input logic [WIDTH-1:0] d,
                                     output logic [WIDTH-1:0] q);
         always_ff @(posedge clk)
```

```
		begin
			if(en) q <= d;
			else   q <= q;
		end
endmodule

module delay #(parameter WIDTH=1)
				(input logic clk,
				 input logic [WIDTH-1:0] d,
				 output logic [WIDTH-1:0] q);

	logic[WIDTH-1:0] p;
	always_ff @(posedge clk)
		begin
			p <= d;
			q <= p;
		end
endmodule

module delay2 #(parameter WIDTH=1)
				(input logic clk,
				 input logic [WIDTH-1:0] d,
				 output logic [WIDTH-1:0] q);

	logic[WIDTH-1:0] p1,p2;
	always_ff @(posedge clk)
		begin
			p1 <= d;
			p2 <= p1;
			q <= p2;
		end
endmodule

module pulse(input logic clk,in,
				output logic out);
	//creates a pulse when the input signal goes high
	logic delayed;
	delay inDelay(clk,in,delayed);
	assign out = in & ~ delayed;
endmodule
```