FPGA Whack-A-Mole

Trust the Processor.

DCS Notice

I have adhered to the Duke Community Standard in completing this assignment.

April 24th, 2018

Overview

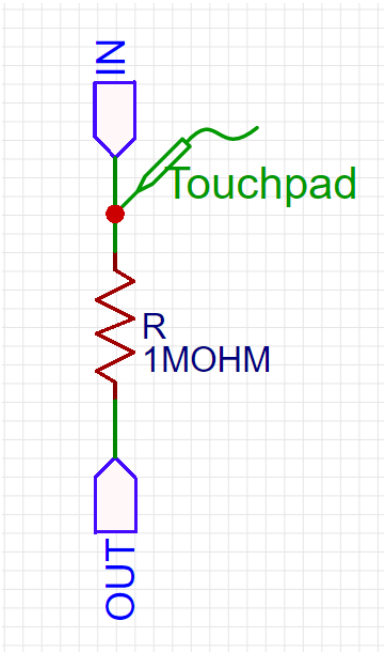
Overview

In our project, we created a Whack-A-Mole game using touch sensors. Our design was strongly hardware-based, and it allowed us to explore some interesting concepts in hardware design as well as mechanical and software design.

Capacitive Sensing

Capacitive sensing is a core component of our game. Instead of using a pushbutton or switch, we decided to improve the user-experience by designing these custom human-touch sensors. To achieve this capability in our processor, we constructed the entire sensor from scratch using behavioral verilog and a single resistor for each. To emphasize the difficulty of such a task, we designed the sensor simply based on the circuit theory - no verilog or code was referenced. A lot of trial and error was used to get the sensors working.

Capacitive sensing is based on the idea that the human body has a non-zero capacitance in relation to virtual ground. Thus, by creating an RC circuit with a large resistance (we used 1MΩ), the drain time of the circuit slightly increases when the human body is connected in parallel to the circuit. The drain time increases on average by 20μs (which is more than enough of a difference for our 50MHz processor to detect).

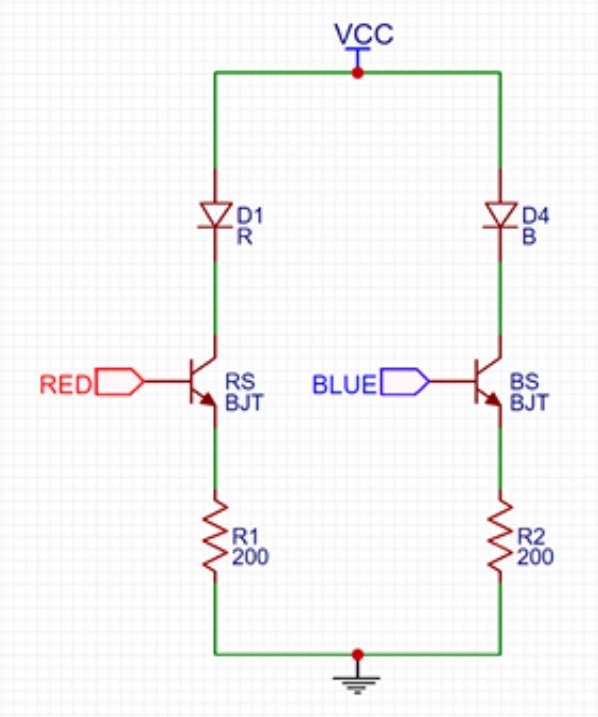


To take advantage of this concept, we connected a GPIO output pin to a GPIO input pin in series with a resistor. We then created a FSM to initially set the output pin high, wait until the input pin is high, then set the output pin low, and count the number of cycles it takes for the input pin to go low. We save this delay in a register. By repeating this FSM with a clock-divided trigger signal, we continuously poll our sensors. When a person touches node between the resistor and input pin, the count increases significantly, enough to get accurate sensing data. With a very conservative trigger signal frequency, we were able to get a new value for each sensor every 2ms.

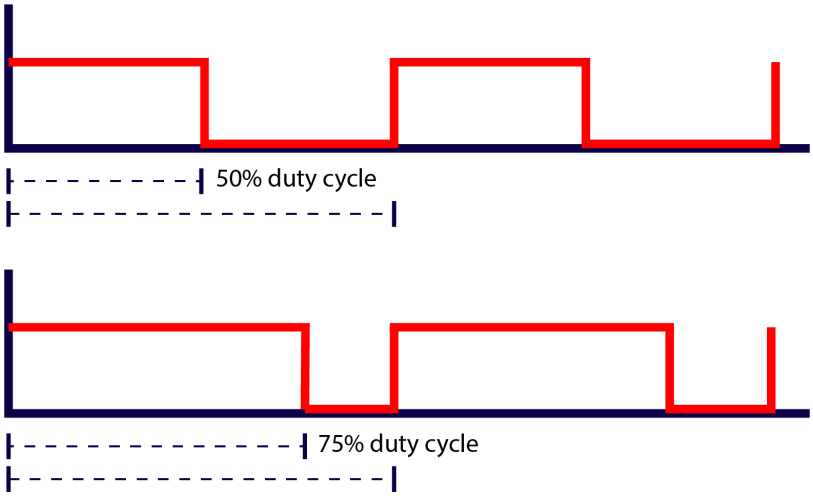
To integrate this with the processor, we defined a new command in the MIP ISA. Essentially the command can read the value of a sensor based on an index and save it to the destination register. In subsequent sections, we will discuss this in more detail.

RGB LED Control

To make the game extra sexy, we used RGB LEDs, with which we can control both their intensity and their color by a simple command in the processor. Each LED is controlled by two PWM signals.



RGB LEDs have four inputs: Common Anode, Red Cathode, Green Cathode, and Blue Cathode. To support 5V LEDs and reduce power consumption to the. We can "fake" a spectrum of intensities by modulating the duty cycle of a clock signal at a constant frequency. Because the duty cycle is the percentage of the time the signal is high on average, it is essentially the percentage of time the LED is on. Since the human brain averages out light signals, we perceive changes in brightness due to changes in duty cycle.



To acheive this effect, we created an FSM counter that counts to 255. At when the counter is greater than a variable threshold, we set the signal low, otherwise the signal is high. We repeat this for each LED signal we use to control.

We have two signals for each LED (red and blue). We avoid using the Green pin - more on that later. To integrate this with the processor, we define an instruction that sets and LED at a certain index (value at $\$R_s$) to a certain color (value at $\$R_t$). The color is defined as a 16-bit number in which the lower 8 bits specify the intensity of the red pin and the higher 8 bits specify the intensity of the blue pin.

Pseudo-Random Number Generation

We created a pseudo-random number generator to allow our game to turn random pins on at random times, for a random duration. As an interesting feature to the number generator, we use the intrinsic noise of our sensors as a seed. This allows us to achieve different games every time the FPGA is reset, since the seeds are essentially random

Our random number generator contains 8 independent 8-bit Fibonacci linear feedback shift registers. The feedback bit of the LFSRs is an XOR of the second and last bits. This type of LFSR is known to produce an m-sequence, which according to a paper by has a balanced output of 1's and 0's and has little correlation between subsequent bits. By extracting the third bit (arbitrarily) from each of these LFSRs, we form an 8-bit psuedo-random number.

As an added bonus, we use the noise from our capacitive sensors to "seed" the LFSRs. To do so, each LFSR reads the last 8 bits of a unique sensor value. When the XOR of bits 3 and 5 of the LFSR is high, we add this seed to our shift register. This allows us to ensure that no two games are exactly the same - even if the processor is reset with the exact same starting state.

Pin Optimization

As a limitation of the FPGAs we are using, we didn't have enough GPIO pins to support 9 "moles". Since each sensor requires 2 pins (1 input, 1 output) and each LED needs 3 outputs, our original design requires $5(9) = 45$ GPIO pins - much more than we actually have. To reduce our number of pins, we made two changes:

Sensors: By combining our sensors in a complex parallel-sensor FSM, we were able to reduce the number of output pins to 1. Thus, we only use 10 pins for capacitive sensors. To do so, we recognized that outputs are relatively similar and combined the output signals. Once all input signals are high, we set the output pin low and each separate "core" of the sensor calculates its drain-time in parallel. Performing this optimization required an overarching FSM to control all sensors and an individual FSM for each sensor.

LEDs: To reduce LED pins, we simplified our design to only use two colors for each LED, thus reducing the pin cost to 18.

The above changes reduced our overall pin cost to 28 pins, which works with the FPGA.

Physical Hardware

1.1 CAD Design

1.1 CAD Design Fabrication

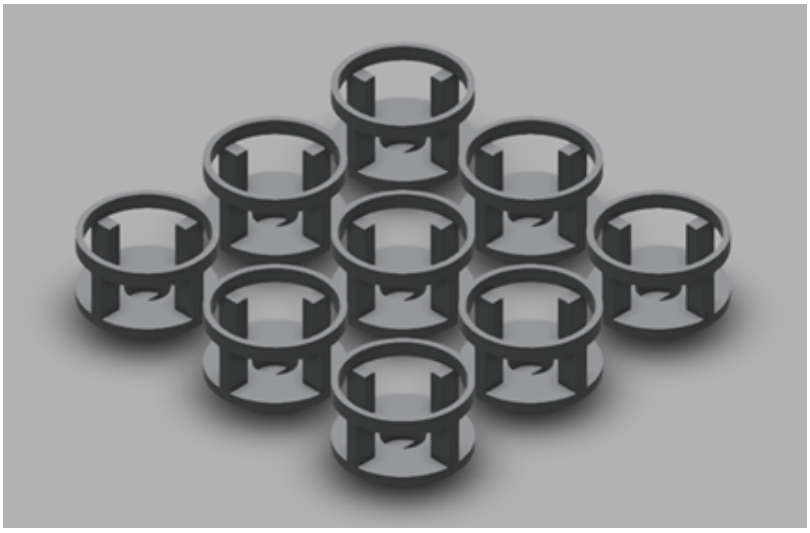
In order to house a grid of capacitive sensors, we needed to design a 3X3 grid capable of holding the capacitive sensors as well as their corresponding LEDs and circuitry.

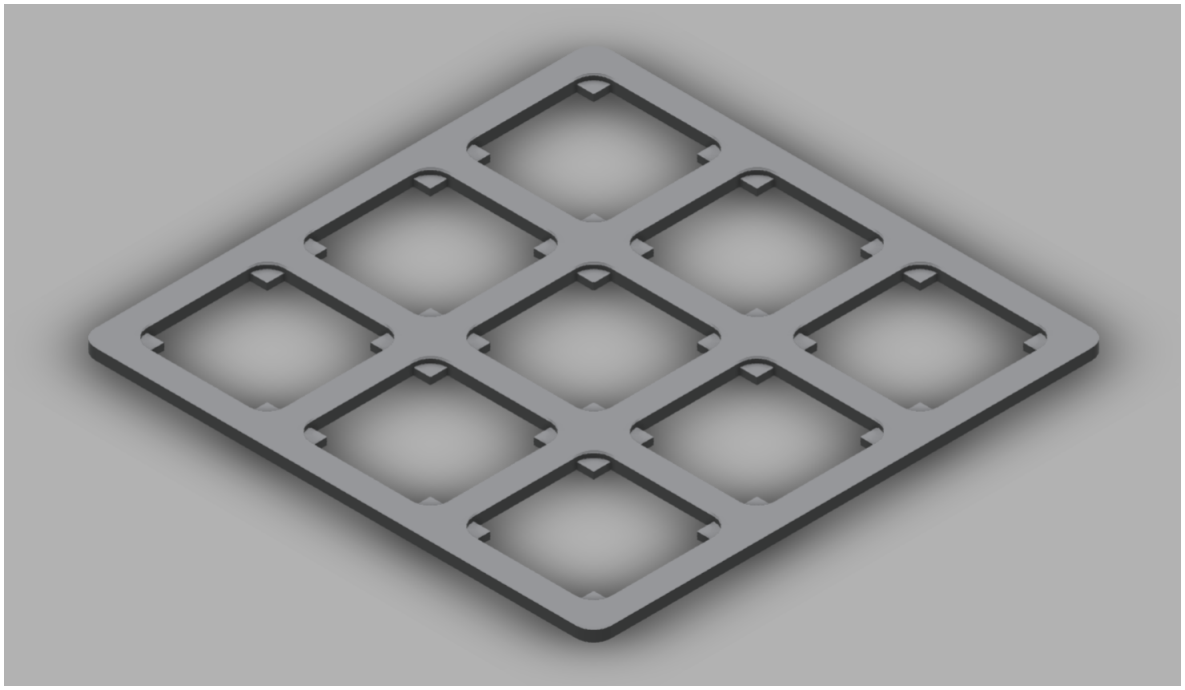
The design process was broken down into three main stages: chassis, sensor interface, and LED sockets. The entire board was made from a single main chassis with 9 individual chambers for each touch sensor. These chambers are about an inch in depth, and were designed to allow the single LEDs to better diffuse through the acrylic pads. The chassis was also built with a bottom compartment to house all of the necessary circuitry. To house the acrylic pads, we designed an MDF-based faceplate interface with a 3X3 cutout that corresponded to the 9 chambers on the main chassis. Most of these components were 3D-printed - the others were either hand-fabricated or laser cut.

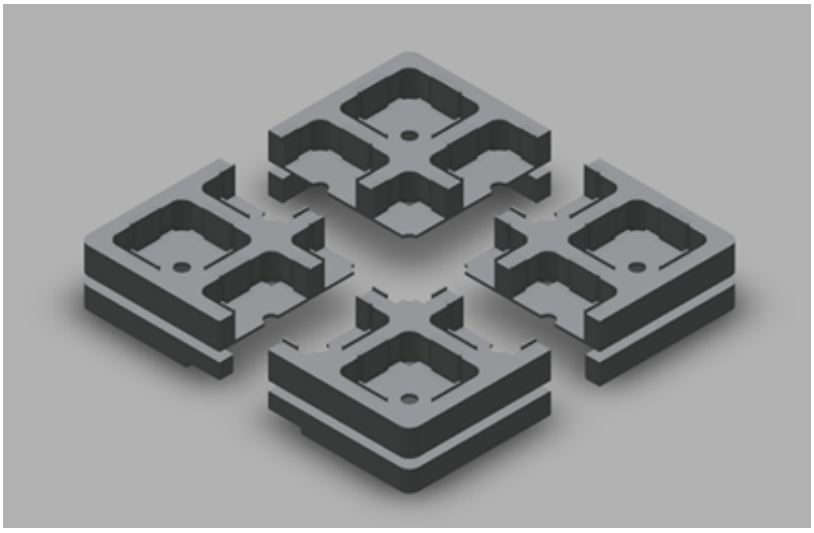
One challenge we faced when designing the touch pads dealt with choosing a proper conductive medium to use for touch sensing. While a traditional metal material (e.g. aluminum foil) would have sufficed, we wanted to ensure that the LEDs would be able to fully permeate through the acrylic pads. Thus, we decided to design the touch interface around indium-tin oxide coated plastic sheets. This would maintain functionality in terms of both practicality and aesthetics.

1.2 PCB Assembly

All of the circuit elements were either soldered onto PCBs (e.g. LED controller) or breadboarded (capacitive sensing). To implement a transistor-pair controller for each LED (R and , we needed to solder an array of 18 total BJT transistors grounded by pull-down resistors. For the capacitive sensors, we wired together a series of 9 1MOhm resistors spread as far apart from each other as possible onto a breadboard. Type a message...







MIPS ISA standards

Our Whack-A-Mole game was written in MIPS using instructions supported by our processor. Our newly added instructions are shown below:

Instruction	Opcode	Type	Operation
beq $rd, $rs, N	01001	I	if ($rd = $rs), PC = PC + 1
led $rd, $rs	01011	I	set led $rd to code $rs[15:0]
cap $rd, $rs	01100	I	$rd = sensorvalue($rs)

Table 1. New Instructions

We used the following MIPS function calling convention in writing our game code:

Register	Alias	Description
$r0	-	zero
$r1	$v0	function return value
...	...	...
$r3	$a0	function argument
...	...	...
$r7	$t0	temporary variables (need not preserve)
$r20	$s0	function variables (preserve before calling)
...	...	...
$r28	$sp	stack pointer (starts at 0)
$r29	$rm	[31:9] zeros, [8:0] pseudo-random number
$r30	$rstatus	exception status
$r31	$ra	return address

Table 2. MIPS Calling Convention

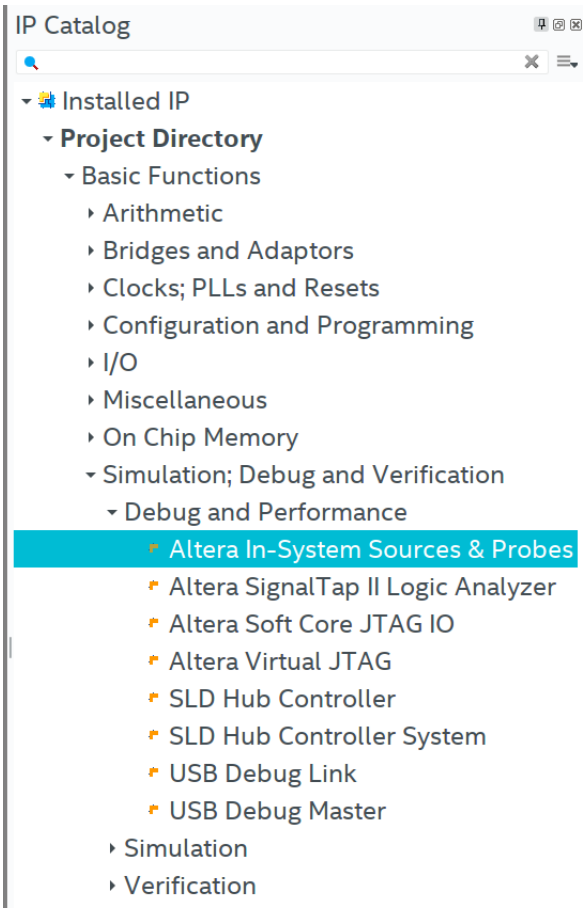
An example function call from our game code is shown below:

```
addi $a0, $r0, 3000
jal delay                # delay for 3 seconds
j end

delay:

   # a0: delay time (ms)

   add $s0, $r0, $a0
   add $s1, $r0, $r0
   addi $s2, $r0, 8333
   mul $s0, $s0, $s2

   delay_loop_begin:     # 3 cycles/loop

      addi $s1, $s1, 1
      blt $s0, $s1, 1
      j delay_loop_begin

   delay_loop_end:

        jr $ra

end:
    nop
```

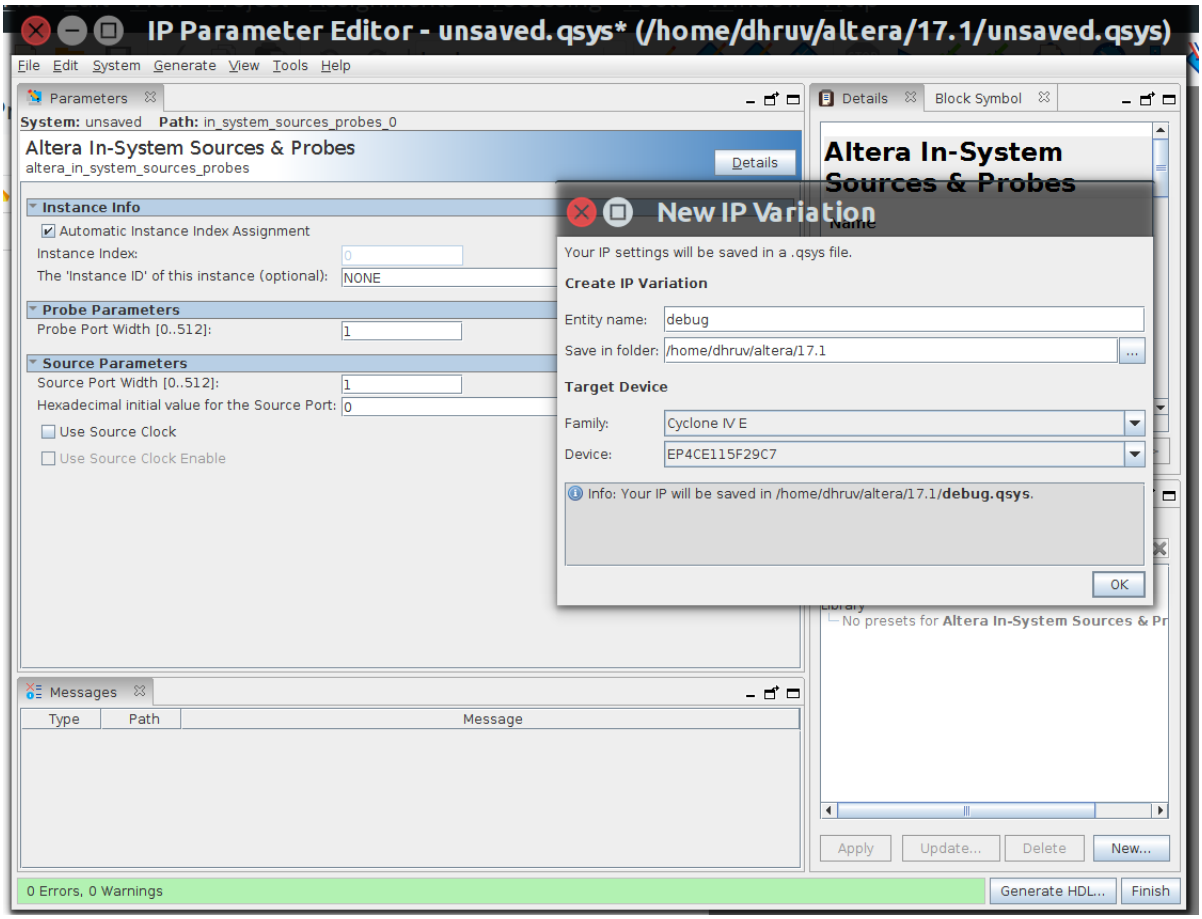
Debugger Probes (For future use)

While working on our project we found a pretty useful megafunction that allows you to view the value of any wire on your FPGA in real-time. It might be a helpful feature for future student to learn, so we included a small tutorial for their benefit.

First, you instantiate the ISSP megafunction:



Define the bus size you want to probe, name it file "my_debugger" and generate the verilog code. What you name this entity is the name you use to instantiate your module later.



Instantiate it in your skeleton as so. You can instantiate multiple probes to view them at the same time by instantiating the module multiple times.

```
/** Debugger **/
wire pin_to_probe
my_debugger d0(.probe(pin_to_probe));
```

Go to: Tools-> In System Sources and Probes Editor. Use this to flash your FPGA and monitor the wires you chose.

Processor Integration

Our pipelined processor was implementing with full bypassing and detection of data and control hazards (stall on RAW for an instruction following lw). The multiplier and divider was also implemented into our processor (rather than a megafunction) for our pseudo-random delay function. We also implemented three new instructions (beq, cap, and led) to use for our game logic and to control our LEDs using the sensor values from our capacitive sensors. Our pseudo-random number generator generates a new random number between 0 and 255 each clock cycle. This value is stored in register $r29. All game data is stored in either the regfile or in data memory.

Game Code

Our interpretation of the game uses 9 LEDs as moles sheltered by acrylic pads that players can hit with their hand or any conductive device. Each mole has the following five atrributes stored in data memory:

1. Index
2. State
3. Time Elapsed
4. Time Limit On
5. Time Limit Off

Each mole corresponds to an RGB LED, so its index is used with our "led" command to set the color of the LED.

Each mole has a state, either *active* or *inactive*. An active mole means that its corresponding LED is set to a nonzero RGB value and has a nonzero *time limit on* parameter. The latter determines the amount of time the mole stays active (in milliseconds). An inactive mole's corresponding LED value is set to off (0) and has a nonzero *time limit off* parameter defining the length of time that the mole will be inactive. These time limit parameters are produced using our random number generator.

Each mole also has a time elapsed parameter defining the length of time that the mole has been in its current state. This value is used in branch instructions to determine when the mole should change states. This parameter is also used to determine the number of points added to the player's score. If the player whacks an active mole, they receive points equal to the time limit on parameter minus the time elapsed. Thus, players with faster reaction times and higher accuracy earn more points. Branch instructions are also used to penalize players by check if a pad was touched while the corresponding mole was inactive.

The whacking action is detected using our capacitive sensing. The "cap" instruction allowed us to use our processor to monitor our sensor values and perform instructions based on those values (via branch instructions). *activate_mole()* and *deactivate_mole()* functions were written to reset/set mole parameters whenever it changes states. Here, the "led" command is used to set a mole's corresponding LED to a specific RGB value using pulse width modulation.