EECS 151/251A FPGA Lab 3:
Tone Generator, Simulation, and Connecting Modules

Prof. Borivoje Nikolic and Prof. Sophia Shao
TAs: Cem Yalcin, Rebekah Zhao, Ryan Kaveh, Vighnesh Iyer
Department of Electrical Engineering and Computer Sciences
College of Engineering, University of California, Berkeley

1 Before You Start This Lab

Run `git pull` in `fpga_labs_fa19`.

We suggest that you look through these two documents that will help you better understand some Verilog constructs.

1. wire_vs_reg.pdf - The differences between wire and reg nets and when to use each of them.
2. always_at_blocks.pdf - Understanding the differences between the two types of always @ blocks and what they synthesize to.

1.1 Running the Lab on Your Laptop

If you want to run the tools for this lab locally, refer to the appendix at the end of this spec for installation instructions.

2 Designing a Tone Generator

Let's create a tone generator/buzzer on the FPGA.

2.1 Clock Sources

Look at the Pynq Reference Manual. Read Section 11 about the clock sources available on the Pynq. We are using the 125 MHz clock from the Ethernet PHY IC on the Pynq board connected to pin H12 of the FPGA chip.

Look at the `src/z1top.v` top-level module and its `CLK_125MHZ_FPGA` input.

```
module z1top (
    input CLK_125MHZ_FPGA,
    ...
);
```

Next take a look at the constraints in `src/PYNQ-Z1_C.xdc` and notice how the LOC for the clock net is set to H16, just as specified in the Pynq-Z1 Reference Manual.

```
set_property -dict { PACKAGE_PIN H16 IOSTANDARD LVCMOS33 } \
  [get_ports { CLK_125MHZ_FPGA }];
```

This is how we map top-level signals in Verilog to the physical FPGA pins they are connected to. We can access the clock signal from our Verilog top-level module and can propagate this clock signal to any submodules that may need it.

2.2 Audio Out

Look at Section 14 of the Pynq Reference Manual which describes the mono audio out feature on the Pynq board.

The FPGA pin R18 is connected to the `AUD_PWM` net. The FPGA can drive this net with a PWM signal which goes through a low-pass filter and is driven into the audio jack on the Pynq board.

There's also an `AUD_SD` net connected to FPGA pin T17, which turns off the opamps in the low-pass filter. Setting `AUD_SD` to 1 enables the audio output.

Find these signals in the `src/PYNQ-Z1_C.xdc` file, and note how they appear in the `src/z1top.v` port list.

2.3 Generating a Square Wave

Let's play a 220 Hz square wave out of the Mono Audio Out port on the Pynq. The square wave should have a 50% duty cycle, so for half of the period of one wave the signal should be high and for the other half, the signal should be low. We have a 125 MHz clock we can use to time our circuit.

Find the following:

Question 1: Square Wave Calculations

a) The period of our clock signal (frequency = 125 MHz)?

b) The period of a 220 Hz square wave?

c) How many clock cycles fit in one period of the square wave?

Open `src/tone_generator.v` and design a circuit to output a 220Hz square wave on the `square_wave_out` output. Ignore the `tone_switch_period`, `output_enable`, and `volume` inputs for now.

3 Simulating the Tone Generator

Let's run some simulations on the `tone_generator` in software to check it works before putting it on the FPGA. To do this, we will need to use a Verilog testbench. A Verilog testbench is designed to test a Verilog module by supplying it with the inputs it needs (stimulus signals) and testing whether the outputs of the module match what we expect.

3.1 Overview of Testbench Skeleton

Check the provided testbench skeleton in `sim/tone_generator_testbench.v`. Let's go through what every line of this testbench does.

```
`timescale 1ns/1ns
```

The timescale declaration needs to be at the top of every testbench file.

```
`timescale (simulation step time)/(simulation resolution)
```

The first argument to the timescale declaration is the simulation step time. It defines the granularity of discrete time units in which the simulation advances. In this case, we have defined the simulation step time to be one nanosecond, so we can advance the simulation time by as little as 1ns at a time.

The second argument to the timescale declaration is the simulation resolution. In our example it is also 1ns. The resolution allows the simulator to model transient behavior of your circuit in between simulation time steps. For this lab, we aren't modeling any gate delays, so the resolution can safely equal the step time.

```
`define SECOND 1000000000
`define MS 1000000
// The SAMPLE_PERIOD corresponds to a 44.1 kHz sampling rate
`define SAMPLE_PERIOD 22675.7
```

These are some macros defined for our testbench. They are constant values you can use when writing your testbench to simplify your code and make it obvious what certain numbers mean. For example, `SECOND` is defined as the number of nanoseconds in one second. The `SAMPLE_PERIOD` is the sampling period used to sample the square wave output of the `tone_generator` at a standard 44.1 kHz sample rate.

```
module tone_generator_testbench();
  // Testbench code goes here
endmodule
```

`tone_generator_testbench` is a testbench module. It is not intended to be placed on an FPGA, but rather it is to be run by a circuit simulator. All your testbench code goes in this module. We will instantiate our DUT (device under test) in this module.

```
reg clock;
reg output_enable;
reg volume = 0;
reg [23:0] tone_to_play;
wire sq_wave;
```

Here are the inputs and outputs of our `tone_generator`. Notice that the inputs to the `tone_generator` are declared as `reg` type nets and the outputs are declared as `wire` type nets. This is because we will be driving the inputs in our testbench inside an `initial` block and we will be reading the output. Note we can set the initial value of `reg` nets in the testbench to drive a particular value into the DUT at time 0 (e.g. `volume`).

```
initial clock = 0;
always #(4) clock <= ~clock;
```

This is the clock generation code. The clock signal needs to be generated in our testbench so it can be fed to the DUT. The initial statement sets the value of the clock net to 0 at the very start of the simulation. The next line toggles the clock signal every 4ns, i.e. half period of 125 MHz clock.

```
tone_generator audio_controller (
    .clk(clock),
    .output_enable(output_enable),
    .tone_switch_period(tone_to_play),
    .volume(volume),
    .square_wave_out(sq_wave)
);
```

Now we instantiate the DUT and connect its ports to the nets we have declared in our testbench.

```
initial begin
    tone_to_play = 24'd0;
    output_enable = 1'b0;
    #(10 * `MS);
    output_enable = 1'b1;

    tone_to_play = 24'd37500;
    #(200 * `MS);
    ...
    $finish();
end
```

This is the body of our testbench. The `initial begin ... end` block is the 'main()' function for our testbench, and where the simulation begins execution. In the `initial` block we drive the DUT inputs using blocking (`=`) assignments.

We can also order the simulator to advance simulation time using delay statements. A delay statement takes the form `#(delay in time steps);`. For instance the statement `#(100);` would run the simulation for 100ns.

In this case, we set `output_enable` to 0 at the start of the simulation, let the simulation run for 10ms, then set `output_enable` to 1. Then `tone_to_play` is changed several times, and the `tone_generator` is given some time to produce the various tones.

The final statement is a system function: the `$finish()` function tells the simulator to halt the simulation.

```
integer file;
initial begin
  file = $fopen("output.txt", "w");
  forever begin
    $fwrite(file, "%h\n", sq_wave);
    #(`SAMPLE_PERIOD);
```

```
  end
end
```

This piece of code is written in a separate `initial begin ... end` block. The simulator treats both `initial` blocks as separate threads that both start execution at the beginning of the simulation and run in parallel.

This block of code uses two system functions `$fopen()` and `$fwrite()`, that allow us to write to a file. The `forever begin` construct tells the simulator to run the chunk of code inside it continuously until the simulation ends.

In the `forever begin` block, we sample the `square_wave_out` output of the `tone_generator` and save it in `output.txt`. We sample this value every `` `SAMPLE_PERIOD `` nanoseconds which corresponds to a 44.1 kHz sampling rate. The `tone_generator`'s output is stored as 1s and 0s in `output.txt` that can be converted to an audio file to hear how your circuit will sound when deployed on the FPGA.

3.2 Running the Simulation

There are 2 RTL simulators we can use:

- **VCS** - proprietary, only available on lab machines, fast
- **Icarus Verilog** - open source, runs on Windows/OSX/Linux, somewhat slower

They all take in Verilog RTL and a Verilog testbench module and output:

- A waveform file (.vpd, .vcd, .fst) that plots each signal in the testbench and DUT across time
- A text dump containing anything that was printed during the testbench execution

3.2.1 VCS

If you're using the lab machines, you should use VCS:

```
make sim/tone_generator_testbench.vpd
```

This will generate a waveform file `sim/tone_generator_testbench.vpd` which you can view using `dve`. Login to the lab machines physically or use X2go and run:

```
dve -vpd sim/tone_generator_testbench.vpd &
```

The DVE interface contains 3 panels (Figure 1).

From left to right, you can see the 'Hierarchy', 'Signals', and 'Source Code' windows. The 'Hierarchy' window lets you select a particular module instance in the testbench to view its signals. In the 'Signals' window, you can select multiple signals (by Shift-clicking) and then right-click → 'Add To Waves' → 'New Wave View' to plot the waveforms for the selected signals.

The waveform viewer is shown in Figure 2.

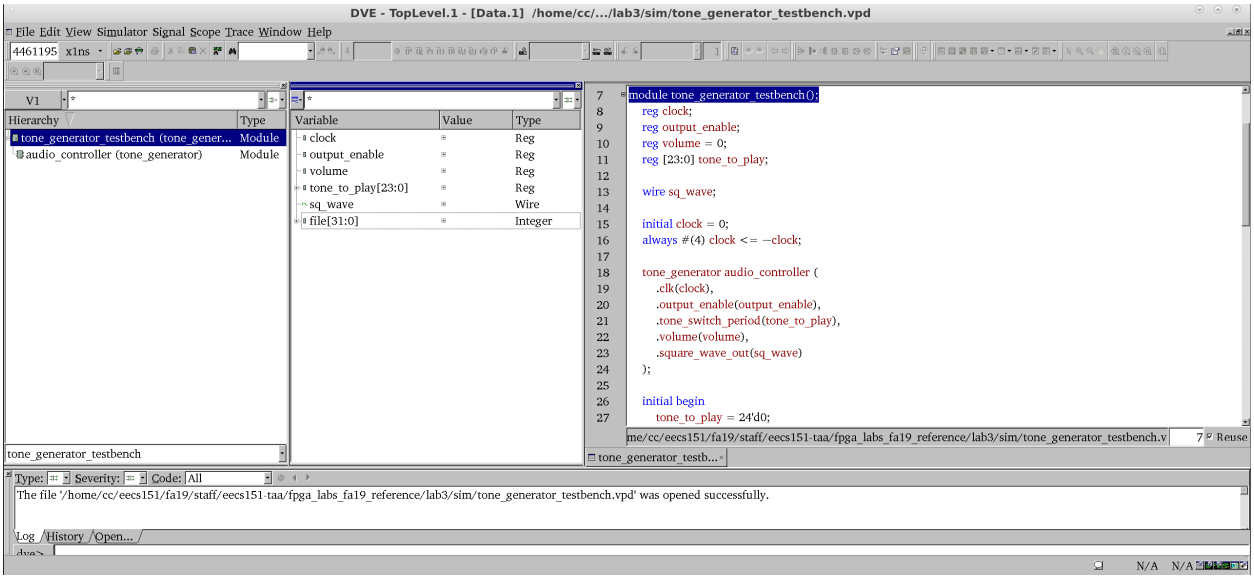

Figure 1: DVE Interface

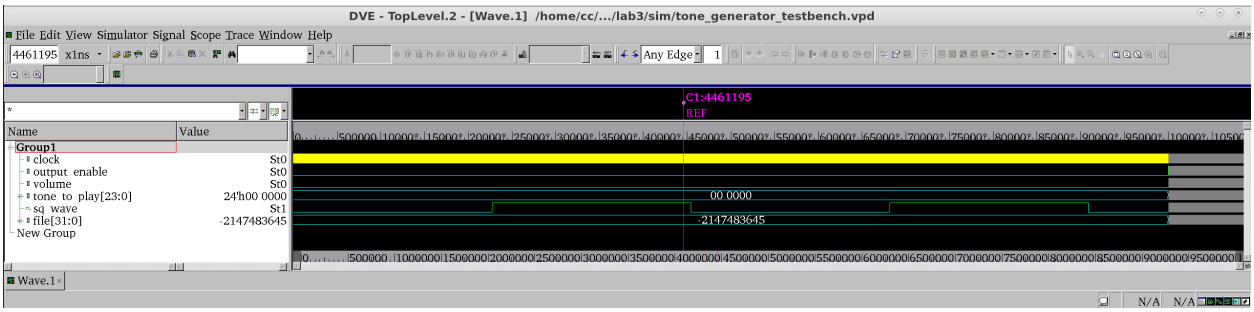

Figure 2: DVE Waveform Viewer

Here are a few useful shortcuts:

- **Click on waveform**: Sets cursor position
- **O**: Zoom out of waveform
- **+**: Zoom into waveform
- **F**: Fit entire waveform into viewer (zoom full)
- **Left Click + Drag Left/Right**: Zoom in on waveform section

3.2.2 Icarus Verilog

Icarus Verilog is available on the lab machines. To install Icarus and gtkwave locally, refer to the appendix.

Run `make sim/tone_generator_testbench.fst` to launch a simulation with Icarus and to produce

a FST waveform file. You can open the FST with gtkwave locally or on the lab machines.

3.3 Analyzing the Simulation

After opening the waveform, you should be able to see the clock oscillate at the frequency specified in the testbench. You should also see the `output_enable` signal start at 0 and then become 1 after 10 ms. However, you may see that the `sq_wave` signal is just a red line. What's going on?

3.3.1 Fixing Unknown Signals

Blue lines (written as 'Z' in Verilog) in a waveform viewer indicate high-impedance (unconnected) signals. We won't be using high-impedance signals in our designs, so blue lines or 'Z' indicate something in our testbench or DUT isn't wired up properly.

Red lines (written as 'X' in Verilog) in a waveform viewer indicate unknown signals. At the start of simulation, all registers in your DUT contain unknown values (represented as 'x'). Since we don't have an explicit reset signal for our circuit to bring the `clock_counter` to a defined value, it may be unknown for the entire simulation.

Let's fix this. In the future we will use a reset signal, but for now let's use a simpler technique. In `src/tone_generator.v` add an initial value to any registers in your design.

```
// Original code:
reg counter;

// Change to:
reg counter = 0;
```

This tells the simulator that the initial value for this register should be 0. For this lab, when you add new registers in your `tone_generator` or any other design module, you should instantiate them with their initial value in the same way. **Do not set an initial value for a 'wire' type net; it will cause issues with synthesis, and may cause X's in simulation.**

Now run the simulation again.

3.3.2 Helpful Tip: Reloading Waveforms

When you re-run your simulation and you want to plot the newly generated signals in DVE or gtkwave, you don't need to close and reopen the waveform viewer. Use `Shift + Ctrl + R` in gtkwave or `File` → `Reload Databases` in DVE to reload the waveform file.

3.3.3 Listen to Your Square Wave Output

Look at the file written by the testbench at `lab3/sim/output.txt`. It contains a sequence of 1s and 0s that represent the output of your `tone_generator` sampled at 44.1 kHz. Use a Python script that can take this file and generate a `.wav` file that you can listen to.

Go to the `lab3/` directory and run the command:

```
python3 scripts/audio_from_sim.py sim/output.txt
```

This will generate a file called `output.wav`. Run this command to play it:

```
play output.wav
```

If `play` doesn't work, try running `aplay output.wav`. You should hear a 220Hz square wave. Compare it with a reference tone generator.

4 Top-Level Wiring and Tone Generator on the FPGA

Open `src/z1top.v` and instantiate the `tone_generator`. Connect `square_wave_out` to `aud_pwm`. Drive the unused inputs of the `tone_generator` to 0. Set `aud_sd` to 1 to enable the audio output.

4.1 Make-Based FPGA Flow

We're no longer using the Vivado GUI to run the FPGA flow, but a Makefile driven flow instead. Inside `lab3` you can run the following:

- `make lint` - Lint your Verilog with Verilator; checks for common Verilog typos, mistakes, and syntax errors
- `make synth` - Synthesize `z1top` and put logs and outputs in `build/synth`
- `make impl` - Implement (place and route) the design, generate the bitstream, and put logs and outputs in `build/impl`
- `make program` - Program the FPGA with the bitstream in `build/impl`
- `make program-force` - Program the FPGA with the bitstream in `build/impl` *without* re-running synthesis and implementation if the source Verilog has changed
- `make vivado` - Launch the Vivado GUI

You should start with `make synth`, and check the log in `build/synth/synth.log` for any warnings or errors. Then build a bitstream by running `make impl`. Program the FPGA by running `make program`.

Warning: the audio output will be loud, don't put your headphones near your ear. Plug in headphones and make sure you hear a buzzing noise at 220Hz. Again, compare the tone to a reference tone generator. To stop the buzzing, you can press the `SRST` button on the top-right of the Pynq.

5 Enhancements

5.1 Switching the Wave On and Off

Now you have a tone, but it can't be toggled on and off without pulling the power to the FPGA board or resetting it. Let's use the `output_enable` input of the `tone_generator` module to gate the square wave output. When `output_enable` is 0, you should pass 0 to the `square_wave_out` output, but when `output_enable` is 1, you should pass your square wave to `square_wave_out`.

Wire up the `output_enable` signal to the first slide switch (`SWITCHES[0]`) in `z1top`.

Run the design flow and program the board. You should now hear a buzzing noise at 220Hz that can be turned on or off by toggling the first slide switch.

5.2 System-Level Testbench

We previously tested the `tone_generator` on its own as a unit-test. We can also test the top-level module `z1top` which contains the `tone_generator`. An example testbench is in `sim/z1top_testbench.v`. Run the system-level testbench with: `make sim/z1top_testbench.vpd`.

Play around with the testbench by altering the clock frequency, changing when you turn on `output_enable` and verifying that you get the audio you expect.

Question 2: Testbench Observations

a) If you increase the clock frequency from 125 Mhz, would you expect the tones generated by your `tone_generator` to be of a higher or lower frequency than was generated with the 125 MHz clock? Why?

b) Prove that the `output_enable` input of your `tone_generator` actually works in system-level simulation. Take a screenshot.

5.3 Volume Adjustment

The tone from the FPGA is too loud! To fix this, when the square wave is high, do not emit a continuous 1 on the `square_wave_out` port, but instead emit a PWM waveform with a duty cycle selected by the `volume` input.

- `volume = 1` $\rightarrow$ duty cycle = 50% when square wave is high
- `volume = 0` $\rightarrow$ duty cycle = 25% when square wave is high

An example of the PWM technique is shown in Figure 3.

You can refer to section 14.1 of the Pynq Reference Manual for help on PWM waveforms.

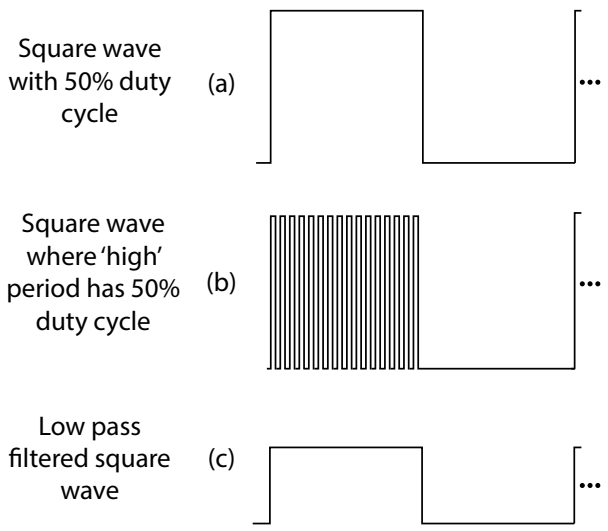

Figure 3: Example of using PWM to drive a 50% duty cycle wave when the square wave is high

Question 3: Verify Volume Adjustment

a) Modify the system-level testbench to iterate through both `volume` settings. What changes were made?

b) Take a screenshot of the waveform showing how `square_wave_out` is PWMed for each `volume` setting.

Wire up the `volume` signal to the second slide switch (`SWITCHES[1]`) in `z1top`. Create a bitstream and test the volume configuration on the FPGA.

5.4 Configurable Frequency tone_generator

Let's extend our `tone_generator` so that it can play different notes. There is a 24-bit input to the `tone_generator` called `tone_switch_period`.

The `tone_switch_period` describes how many cycles of the `clk` should pass before the square wave output is toggled. For example a `tone_switch_period` of 150000 tells us to toggle the square wave output every 150000 clock cycles, which for a 125 Mhz clock translates to a ~ 417 Hz square wave. Here is the derivation:

$$\frac{2 \cdot 150000 \text{ cycles}}{1 \text{ second}} = \frac{125 \times 10^6 \text{ cycles}}{1 \text{ second}} \div \frac{f \text{ cycles}}{1 \text{ second}}$$
$$f \approx 417 \text{ Hz}$$

Note we multiply the `tone_switch_period` by 2 to get the actual period of the square wave.

You should toggle the square wave output every `tone_switch_period` cycles. Remember to initialize any new registers declared in your `tone_generator` to their desired initial value to prevent unknowns during simulation. You should also handle the case when `tone_switch_period` is 0. In this case disable the tone output.

Open `src/tone_generator.v` and implement the functionality above. Extend the `tone_generator_testbench` to play different tones and verify the tone generator works as expected.

Question 4: Verify Configurable Frequency

a) Create a testbench that plays some simple melody that you define. Save the audio file for checkoff.

b) Verify that when `tone_switch_period` is set to 0, the `square_wave_out` output doesn't toggle. Attach a screenshot.

5.4.1 Try the Configurable Frequency tone_generator on the FPGA

Modify `z1top.v` to wire `tone_switch_period` to `BUTTONS[3:0]` input to the `tone_generator`. You should tie the `tone_switch_period` to `BUTTONS[3:0]`, left-shifted by 16 bits (effectively a multiplication by 65536). This will allow you to control the `tone_switch_period` from 65536 to 983040. Leave `SWITCHES[0]` to control `output_enable` and `SWITCHES[1]` to control `volume`.

```
tone_generator audio_controller (
    .output_enable(SWITCHES[0]),
    .volume(SWITCHES[1]),
    .tone_switch_period(BUTTONS[3:0] << 16),
    // ...
);
```

Run the usual flow to put your new `tone_generator` on the FPGA. Verify that pushing the buttons changes the frequency of your `tone_generator`.

6 Checkoff

In any lab session, show the TA the following:

1. Show the RTL you used to create your tone generator
2. Play an audio file that demonstrates playing a melody using the tone generator
3. Demonstrate your tone generator on the FPGA
   (a) Demonstrate muting the output with a switch
   (b) Demonstrate the volume control
   (c) Demonstrate the runtime frequency configurability with the buttons

Submit a lab report with answers and screenshots for the questions in this lab to Gradescope.

A Local Dev Setup

We aim for the labs to be do-able using your laptop connected to a Pynq board. Refer to the Lab 2 spec's appendix to install Vivado locally.

The other dependencies for this lab are `make`, `iverilog`, and `gtkwave`. We'll cover installing them for each OS.

A.1 Linux/OSX

1. Add vivado to your `$PATH` by adding this in your `.bashrc`:

   ```
   export PATH="/opt/Xilinx/Vivado/2019.1/bin:$PATH"
   ```

2. Install Icarus Verilog
   - Linux: `sudo apt install iverilog`
   - OSX: `brew install icarus-verilog`
3. Install gtkwave
   - Linux: `sudo apt install gtkwave`
   - OSX: Download the app

In a terminal you should be able to successfully run `vivado`, `iverilog`, `gtkwave`.

A.1.1 Sharing Pynq USB Device from OSX to Linux VM

TODO add screenshots

A.2 Windows

Windows packages for Icarus are available. You should be able to download the appropriate .exe for your machine. Gtkwave can be installed along with Icarus (just make sure you check the box for it). Lastly, here is a link to install Cygwin. When you install Cygwin, you need to check the boxes for make and git as shown in Fig. 4.

After installing Icarus, gtkwave, and cygwin, you'll need add Vivado, Icarus, and gtkwave to your Windows PATH with the below steps.

- Go the windows control panel and find the "Edit the system variables menu". Alternatively, if you just search for "system variables", the menu should pop up.
- Click the "Environment Variables" button near the window's bottom right corner (Fig. 5).
- Double click the "Path" variable in the user variables dialog (Fig. 6).
- Click the next empty row and paste the path of the appropriate program (Fig. 7). Hit okay

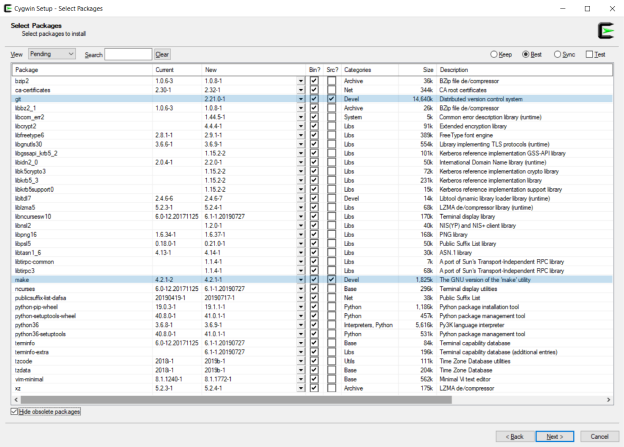

Figure 4: Check make and git with your cygwin install!

and you're done!

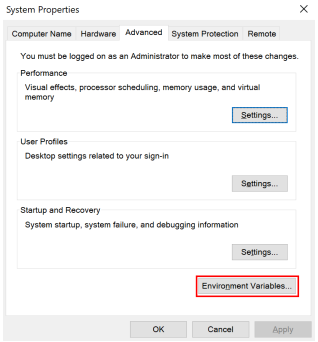

Figure 5: The System variables dialogue.

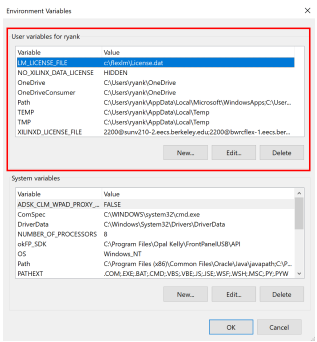

Figure 6: Environment variables dialogue.

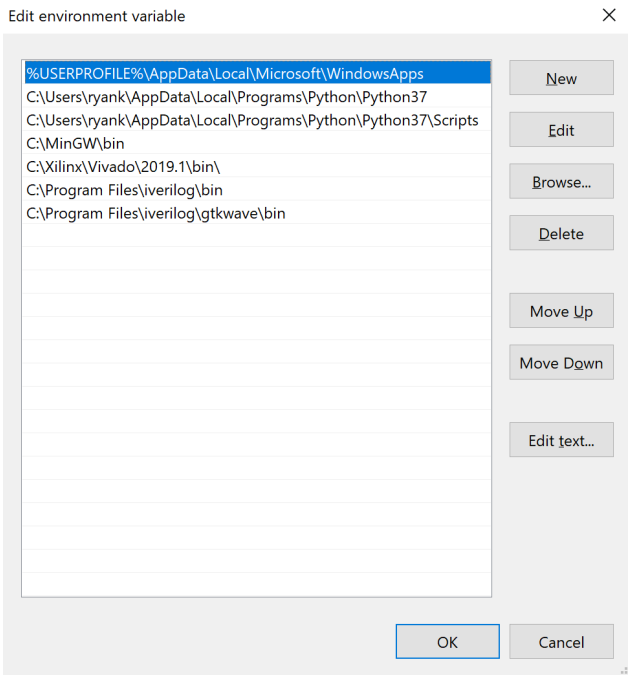

Figure 7: Adding programs to your PATH.

Ackowlegement

This lab is the result of the work of many EECS151/251 GSIs over the years including:

- Sp12: James Parker, Daiwei Li, Shaoyi Cheng
- Sp13: Shaoyi Cheng, Vincent Lee
- Fa14: Simon Scott, Ian Juch
- Fa15: James Martin
- Fa16: Vighnesh Iyer
- Fa17: George Alexandrov, Vighnesh Iyer, Nathan Narevsky
- Sp18: Arya Reais-Parsi, Taehwan Kim
- Fa18: Ali Moin, George Alexandrov, Andy Zhou