Hardware/Software Codesign for Wireless Systems (E168b)

Harris

Lab 2: GPS Correlator

Introduction

In this lab, you will build a time-multiplexed correlator to search for and track GPS satellite signals. You will receive a Verilog file containing most of the modules. You will write the correlator module, then synthesize it with Synplify and verify it with ModelSim. For the sake of time, we will not use the Xilinx tools in this lab. However, the GPS code could be integrated into Platform Studio and called from C just like the popcount code if you wished; it would replace the user_logic.v file.

The purpose of the correlator is to compute

$$I(f_{if},k,s)=\sum_{n=0}^{N-1}x[n]\cos\left(2\pi\frac{f_{if}}{f_s}n\right)C_s[n-k]$$

$$Q(f_{if},k,s)=\sum_{n=0}^{N-1}x[n]\sin\left(2\pi\frac{f_{if}}{f_s}n\right)C_s[n-k]$$

where x[n] is the nth sample of the signal from the GPS antenna, f is related to the intermediate carrier frequency and sampling rate, and C_s[n-k] is the code from satellite s delayed by k samples. I and Q are the in-phase and quadrature phase signals obtained by summing with a carrier 90 degrees out of phase (cosine vs. sine). The GPS receiver searches through many f's and k's to find the Doppler frequency and code delay that best matches the incoming signal.

We will use test data recorded from a USB dongle attached to an antenna with an sampling frequency of f_s = 16.3676 MHz and an intermediate frequency (IF) of f_{if0} = 4.1304 MHz. The correlator typically integrates (sums) for a T_{int} = 1 millisecond, so N is chosen to be $T_{int}f_s$ = 16368 samples.

The GPS acquires the incoming signal by correlating to find I and Q and looking for peaks indicating a particular satellite s at a particular intermediate frequency f_{if} and code offset k. To find the satellites in the sky, it searches for all 32 satellites (s). Each satellite could have a code offset of [0, 1023) chips, and the receiver searches with ½ chip resolution. The received frequency is $f_{if} = f_{if0} + f_{doppler}$, where $f_{doppler}$ is in the range of +/- 5 KHz. To get a good correlation, the carrier should drift by no more than ½ a period during the time of integration, so the searcher must search with a frequency resolution of $1/(2T_{int})$ = 500 Hz. Putting this all together, there are a large number of correlations that must be performed.

The sampling frequency is relatively slow compared to the processing capabilities of a chip. Therefore, it is common to time-multiplex the correlator to search several different values of f_{if}, k, or s during a single sample period (corresponding to a single value of n). Also, a correlator is a relatively simple hardware block compared to the capacity of the chip, so it is common to build multiple channels in parallel, each with its own time-multiplexed correlator. In other words, by taking advantage of temporal and spatial parallelism, we can greatly increase the number of correlations per sampling period.

Once the satellites are acquired by finding the correlation peaks, they are tracked to precisely measure the phase and frequency of the carrier and code and to decode the data bits being transmitted. The receiver must track a satellite for 37 seconds to receive the full data message.

1) How many correlations must be found to acquire the satellites?
2) If the GPS hardware has a single correlator operating in real time, how long would it take to acquire the satellites? Would you find this acceptable as a user?
3) Our system will use 12 correlators that are 3-way time multiplexed. How long would it take to acquire the satellites? Would you find this acceptable as a user?

Our design with 36 effective correlators (12 3-way multiplexed) is typical of a commercial design. The time multiplexing is used to search early, prompt, and late versions of the code delayed by -½, 0, and ½ chip time.

In comparison, SiRF has recently brough the SiRFstarIII chip to market, featuring 200,000 effective correlators. This allows a hand-held GPS to aggressively search the sky and lock onto weak satellite signals even in canyons and under trees where most GPS units lose lock.

Correlator Module

Copy the lab2 folder from the E168b directory on Charlie into your working directory and rename it lab2_xx.v. Browse through the code. gps.v describes the GPS correlator and the memory-mapped interface. Look through the code and see how it is organized. gpstest.v applies a millisecond of recorded data and programs the memory-mapped registers to search for various satellites. It produces an error message if the energy detected doesn't match the expectation.

The correlator module takes receives the ifdata (x[n]), the sine and cosine values at time n, and codechip (the chip value C_s[n-k]). It also provides three one-hot enable signals indicating whether the codechip is the early, prompt, or late version; this is used to specify the 3-way time multiplexing. It produces six outputs representing the running totals of I and Q for early, prompt, and late. The correlator contains two multipliers to compute x[n]*cos() and x[n]*sin and two integrators to further multiply by the codechip value and add.

Your task is to code the integrate module in Verilog. A good solution will take fewer than 20 lines, (aside from comments and white space). The inputs to the module are clk, reset, baseband, codechip, msintrpt, ene, enp, and enl, and the outputs are e, p, and l.

The integrator outputs should be zeroed when reset or msintrpt are applied. It should then start adding to the output specified by the enable. Baseband is a 5-bit number in sign/magnitude form with the most significant bit indicating the sign. Codechip is +/- 1, with 0 representing -1 and 1 representing 1. The outputs should be 32-bit twos-complement numbers.

Synthesis

Once you have coded your design, you will simulate and synthesize it. Normally, simulation would occur first. However, Synplify Pro has much better error messages and the schematics it produces help you visualize the hardware, so I have found that synthesizing first eliminates many of the problems that are tedious to track in simulation.

In previous classes, you have invoked Synplify and Modelsim from Xilinx. However, these programs can be run directly and are much faster to use directly without the baggage of Xilinx, especially when you are debugging a complicated system.

Invoke Synplify Pro from the Start menu of the computer. From the File menu, create a new project called lab2syn_xx in your directory. From the Project menu, add gps.v. From the options menu, choose Configure Verilog Compiler. Under the Device tab, choose Xilinx Virtex2p XC2VP30 with a speed grade of -7. Undert the Constraints tab, set the frequency to 100 MHz. Under the Verilog tab, set the top level module to integrate. From the Run menu, choose Synthesize. In the bottom pane, look at the messages tab. You should see a variety of notes. If there any warnings or errors, fix them. Under the HDL Analyst menu, bring up the RTL view. Look at the schematic and make sure that it matches expectations.

Bring up the technology view and decipher how the schematic mapped to LUTs. Under the HDL Analyst menu, choose Show Critical Path. Browse through the path. The first number is the arrival time. The second number is the slack. As you move through the circuit, you should see the arrival time increasing.

4) How long is the critical path of your design? Will it work with the 100 MHz clock on the Virtex board? Explain the main elements on the critical path at a conceptual level (don't just say it goes through a certain number of MUXCY_L blocks, but explain at a level that relates to the Verilog code).

Simulation

Now that you have eliminated the most serious bugs through synthesis, simulate your design.

Invoke ModelSim from the Start menu. From the File menu, create a new project named lab2sim_xx in your directory. Add gps.v and gpstest.v. From the Compile menu, choose Compile All. ModelSim and Synplify have different Verilog parsers, so it is not unusual to find a few more warnings or errors in ModelSim to fix.

Once your code compiles correctly, choose Simulate * Start Simulation. You will be prompted to choose which unit to simulate. Expand work, then scroll down and select integrate. The

simulator will start and some windows will open so that you can explore your design. Select View * Debug Windows * Wave to open a waveform viewer; some of the other debug windows may be helpful too. In the objects pane is a list of the signals in the integrate module. Select them all and drag them into the waveform pane so that you can watch them during simulation.

Type simulation commands into the transcript pane at the bottom of the screen. For example, type:

```
force clk 0
force reset 1
force baseband 'h13
run 100
```

These commands set values on some of the inputs, then runs the simulation for 100 time steps (nanoseconds) with these values. Try applying a few commands to verify that your integrator resets correctly and adds a value properly. Learn to scroll in and out. Learn to display signals such as baseband in binary and hexadecimal form.

If you want to start the simulation again, type restart (or better yet, restart –f to avoid the annoying dialog box).

If you need to make corrections to the code, choose Compile All again. Then type restart to begin the simulation again with the corrected code; there is no need to exit the simulator as you would have to in Xilinx.

Self-checking Testbench

Applying test vectors manually is tedious, especially if you have errors and have to repeat the tests after you correct the code.

Write a self-checking testbench called intbench_xx.v to test the integrator. You may wish to model it after gpstest.v, which is a self-checking test bench for the entire GPS unit. Your testbench should instantiate the integrator and generate a periodic clock and apply reset for a cycle at the beginning. It should then read test vectors from a file. The test vectors should specify the values on the other inputs (baseband, codechip, msintrpt, ene, enp, and enl). It should also contain the expected values of e, p, and l. The testbench should compare the actual outputs against the expectations on the negative edge of the clock and report any discrepancies, and should also report if all test vectors were applied with zero errors. Make up a simple test file that convinces a skeptical viewer that the integrator works correctly.

Add intbench_xx.v to the project and recompile. Choose Simulate * End Simulation, then start a new simulation with intbench_xx as the top-level module. Type run 10000 to run for 10 microseconds. You shouldn't have to specify any other stimulus by hand.

Fix any problems you encounter.

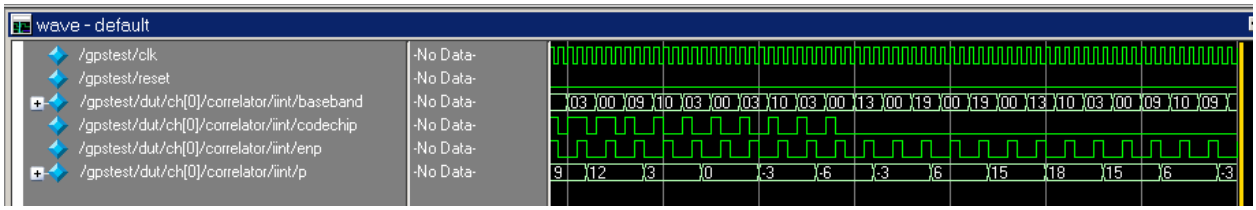
GPS Testbench

The gpstest module tests the entire GPS Verilog code by writing values to the memory-mapped registers to configure Channel 0 to search for satellite 5 at a Doppler offset of -1450 Hz and a code offset of 741.5 chips. It checks if the prompt results for I and Q are FFFFF3C9 and 3148, respectively. Read through the module and gps.tv to understand the tests being applied.

Begin a new simulation, selecting gpstest. Run for 500000 ns. If all goes right, you should see:

```
# write fffa31b7 to register 00000010
# write 00000004 to register 00000080
# write 000005cb to register 00000040
# Pausing until next ms
# read fffff3c9 from register 00000200
# read 00003148 from register 00001000
# read cc741348 from register 00004000
# read ffbdbf89 from register 00008000
```

If you read the wrong values from registers 200 or 1000, your integrator is incorrect. Track down the bug and squash it. You can navigate through the workspace tab to find the modules inside the gpstest module and add their waveforms for debugging. The first microsecond of results from the correct simulation appear below.



5) By approximately what factor is your simulation slower than real time (your answer can be accurate to the nearest order of magnitude)? How long would it take to acquire the satellites using the real time numbers from question 3?

Key Ideas

In this lab, you have practiced a number of skills that are very important in professional design. You have coded and debugged an interesting Verilog module and learned to use the synthesis tool and simulator in stand-alone mode. You have created and used self-checking testbenches. You have also made estimates about runtimes that guided the architecture of the system (time-multiplexed with multiple channels) and showed that simulating the entire system to acquire lock is impractical. These kinds of quick estimates are an important part of the engineer's craft.

What To Turn In

1. Answers to the questions through the lab.
2. Your integrate Verilog module.
3. A description of your synthesis and simulation results for integrate. Were there any warnings or errors?
4. A copy of your self-checking testbench for integrate. Does it prove that the integrator works?
5. The results of running gpstest. Are IP and QP read correctly?