Abstract

We built a digital circuit capable of measuring the current speed, ride distance, ride time and average trip speed of a bicycle. By measuring the time elapsed between wheel rotations, our bike computer calculates these quantities. The computer also discerns when a rider is stopped and holds the ride time constant until the rider resumes movement. The rider can reset the computer's data or switch display modes after unlocking the computer through button inputs.

Table of Contents

1. Introduction

Problem Statement: For training or navigational purposes, bikers typically need to know their travelling velocity, the distance they've ridden, the time elapsed over their ride, and their average speed. The computer that relates these quantities must quickly and accurately measure speed, time and distance, display those quantities, allow the rider to reset the computer after a ride's completion, and safeguard a ride's data from accidental deletion or display change.

2.1 Specifications - See Appendix 1 and 1a for Panel Map and Sensor

Inputs:

- Magnetic Sensor to detect wheel rotations
- UNLOCK, RESET, and CHANGE MODE buttons

Outputs:

- 4-digit, 7-segment display shows one of the following modes:
    1. Current Speed (XX.XX MPH)
    2. Ride Distance (XXX.X Miles)
    3. Ride Time (HH.MM)
    4. Average Speed (XX.XX MPH)
- Ride Enabled LED
- Locked LED

Operation:

While the bike computer is powered on, the 4-digit, 7-segment display shows one of the following four quantities: current speed, ride distance, ride time and average speed of the ride. The computer is initialized to show current speed. The units and

decimal point placement of the display mode can be seen in the output specifications above. The current speed display updates automatically every wheel rotation. Time and distance increment for the smallest displayed quantity (minute and tenth of a mile respectively). Average speed updates every second while the biker is riding.

If the speed drops below .5 mph (9.8 seconds between sensor pulses), the bike computer turns off the ride enabled LED, sets current speed as 00.00 mph, stops incrementing distance and ride time, and holds the average speed value. Once the rider resumes motion and exceeds .5 mph, the ride enable LED turns back on and calculations/updates resume.

The FPGA displays one of the calculated quantities on a 4-digit, 7-segment display at any given time. The rider can switch between these modes by pushing the CHANGE MODE button after unlocking the computer. The CHANGE MODE button increments the display along this cycle by one step per press. Current Speed -> Distance -> Ride Time -> Average Speed -> Current Speed ... The FPGA clarifies which mode is displayed by lighting one of four mode indication LEDs.

The rider can also clear the ride data by pressing the RESET button after unlocking the computer. RESET sets the ride time, elapsed distance and average speed to zero. A rider can clear their data in a non-riding and a riding state.

The computer's default locked state prevents the biker from accidently clearing their ride data. To move the computer to an interactive state, the biker presses the UNLOCK button. After twelve seconds of inactivity (after unlocking or pressing CHANGE MODE/RESET) the computer shifts back into a locked state. The computer shows it's in a locked state by illuminating the locked LED.

2.2 Operating Manual -

- Unlock: Press the UNLOCK button. Computer will lock after 12 seconds of inactivity.

- Change Mode: Press CHANGE MODE button while computer is unlocked.

- Clear: Press RESET BUTTON.

- Current Speed, Ride Time, Ride Distance and Average Speed are automatically updated and displayed.*

The bike computer will continuously monitor the signals from the wheel sensor (Hall Effect Sensor) and the rider's inputs while it is powered. See Appendix 1a and 5 for sensor attachment and specifications. In order to attach the bike sensor correctly, the user should attach a strong magnet to one of the bike wheel's spokes. At the same radius away from the wheel's hub, the user should attach the magnetic sensor to the bike frame. When the magnet passes the sensor, the sensor should illuminate a red LED, indicating the presence of a magnetic field.

* Note: Bike computer is currently configured to calculate these quantities specifically for a bike wheel size of 700 mm diameter. Using wheels of other size will result in erroneous calculations and data.

2.3 Theory of Operation

The bike computer relies on knowing when the bike wheel completes a rotation and how much time passed in the most recent rotation to compute speed, distance and time. The wheel sensor input and clock counters provide those key pieces of information. (Our bike computer uses a 1 MHz clock. Knowing that 1 million clock cycles occur each second allows us to compute elapsed time.)

TopLevel: See Appendix 3 for block diagram

Each of the calculation modules produce a 16 bit data bus that attaches to a 4x1 multiplexer. The controller controls which input it selected among the data inputs and forwarded to the 16 bit binary to decimal double dabble converter. The double dabble's output goes to a multiplexed 7-seg display that was passed out earlier in the term for the stopwatch lab.

The controller coordinates enable and clear signals to the ride statistic modules (time, distance and average speed), but the ride enable signal and current speed modules are independent of the controller. Every input signal passes through debouncers and monopulsers before arriving to the computational module.

The top level also includes a slow clock divider to divide the 100 MHz clock down to a 1 MHz clock used by the bike computer.

Multiplexer, Double Dabble A, Double Dabble B and the Decimal Point Output:

The multiplexed 7-segment display needs four 4-bit binary coded decimal inputs instead of a single 16-bit unsigned input to display meaningful information. We multiplex between the four data sources to feed one 16-bit data bus to the double dabble. The double dabble algorithm we found online provided us the fundamental code for the double dabble module. We had to change it to accept 16 bit inputs and produce four 4-bit BCD outputs. The outputs of the double dabble go to the multiplexed 7-segment display module.

When the MultiplexerSelect bits from the controller are "00", the multiplexer passes the current speed bus. When the select bits are "01," the ride distance input

passes through to the double dabble. When the select bits are "10," the multiplexer has slightly different behavior explained below. Finally, if the select bits are "11," the multiplexer sends the average speed data to the double dabble.

When displaying the Ride Time information, the top level utilizes two double dabbles to transform the two concatenated unsigned quantities into two separate BCD The least significant eight bits (unsigned minutes) get concatenated with x"00" on the left (more significant) side and that new 16 bit bus is passed to double dabble A (the default double dabble module). The most significant eight bits (unsigned hours) get left concatenated with x"00" as well and get fed to double dabble B (a secondary double dabble used only for this timer mode). The ones and tens outputs of double dabble A (minutes) connect to the y0 and y1 inputs of the 7-segment display multiplexer. Double dabble b (hours) sends its ones and tens outputs to y2 and y3 respective inputs of the display multiplexer.

Because we desired to preserve greater precision than integer values in our speed data, we decided to calculate our data as some fractional power of ten. We track how many hundredths of miles an hour the biker is travelling and how many tenths of miles the biker has traveled. To convert this fractional quantities back into decimal notation, we simply divide by ten via the decimal point control. When the multiplexer mode is displaying current or average speed (Mux Select Bits = "00" or "11"), the decimal point is activated on the second most significant digit (effectively dividing by a hundred). When the ride time mode is selected (Select Bits = "10"), we approximate the nonfunctional colon with the decimal point between the second and third most significant digits. When display ride distance (Select Bits = "01"), the decimal point between the third and fourth most significant digits illuminates to divide by ten.

Current Speed Calculation: See Appendix 3 for block diagram

The current speed calculation module only receives the debounced, monopulsed wheel sensor signal. Within the module, a clock cycle counter increments every time the clock rises and the wheel signal input isn't high. When the wheel signal is high, the clock counter value is loaded into a register and the counter is cleared.

The register that loads the clock cycle count feeds a divider. Dividing 492,231,200 hundredths of miles per hour by N clock cycles gives the current speed in hundredths in miles an hour. This dividend was found by transforming the quantity 2.20 m / (N clock cycles * 1 microsecond) into hundredths of miles per hour. Hundredths of miles an hour is an advantageous quantity to compute because the digital division will drop fractional remainders, and we would lose accuracy if we divided to find just miles an hour. We can easily convert hundredths of miles an hour to miles an hours by activating the decimal point on the second most significant digit on the display. The current speed module passes the 16 bit, unsigned bus to display multiplexer.

The current speed calculator module also sends the ride enable signal to the rest of the computer and to the output pin. If the clock cycle count exceeds 9,844,624, the counter is disabled and the ride enable signal is turned off. If the rider is travelling below .5 mph, each will rotation will take longer than 9.844624 seconds (1 MHz clock). We decided that the rider is effectively stopped at that point.

Ride Distance: See Appendix 3 for block diagram

The ride distance module receives a ride enable and clear signal from the controller and the wheel tick signal from the sensor. It produces a 16 bit bus output that

represents the number of tenths of miles that the ride has covered. If the clear signal from the controller is high, the module sets its count register to zero.

The module tracks how many hundredths of miles the rider has traveled so far. For each wheel click while the ride enable is high, the register increments by 220. The value stored in the register is the total cm traveled so far (each wheel tick means the rider has travelled 220 cm or 2.2 m). Then, we convert this cm distance total to hundredths of miles per hour via strength reduction (see Appendix 6). In order to do this, we multiply the cm distance by 16680 and then shift the resulting number 28 bits right. Then, by taking the least significant 16 bits of the result, we get the distance travelled in hundredths of miles per hour.

Ride Time: See Appendix 3 for block diagram

The ride time module only receives a clear and ride enable signal from the controller. The ride time module output is a 16 bit data bus. The first 8 bits contain the unsigned number of hours of riding, and the second 8 bits contain the unsigned number of minutes.

The timer computes the ride time by tracking clock cycles, seconds, minutes and hours. While the ride enable is high, the clock pulse counter increments. There are four levels of cascading counters that depend on the pulse counter. They are pulses (1,000,000) -> seconds (59) -> minutes (59) -> hours. After each counter reaches its terminal count, its output enable rises (causing the next counter to increment) and the counter resets. When the clear signal from the controller goes high, all of the counters clear. The 16 bit output is formed by concatenating the last 8 bit counts for minutes and hours. (Time out <= hours & minutes.)

Average Speed: See Appendix 3 for block diagram

The average speed module receives the wheel sensor signal from the input port and clear and ride enable signals from the controller. The module outputs a single 16 bit data bus that goes to the display multiplexer. When the clear signal from the controller is high, the average speed value in the module is set to zero.

Every time a wheel sensor tick appears, this module increments the thousandths of clicks register value by 1000. This odd storage method and increment value help maintain significant figures later on. The module also has another counter that tracks clock pulses while ride enable is high. Every 1,000,000 clock pulses, the seconds counter is incremented. The wheel rotations register and time counter represent the time and distance elapsed over the ride. Then, every time the second counter is incremented, the quotient register is updated with the value of thousands of click / seconds. The output of the quotient register is then converted to hundredths of miles per hour via strength reduction (see Appendix 6). The modules outputs the least significant 16 bits of the result as the average speed in hundredths of miles per hour.

Controller: See Appendix 4 for State Diagram

While the bike computer is powered on, the controller is responsible for managing the clear, change mode and ride enable signals. The clear and change mode signals originate from the RESET, UNLOCK and CHANGE MODE button inputs. The controller receives those input signals through debouncers and monopulsers. The current speed module sends the ride enable signal to the controller. The controller passes clear and enable signals to the time, distance and average speed modules. The

controller also generates the multiplexer control bits ("MuxBits") that pass the 16 bit data bus to the double dabble which controls the 4-digit, 7-segment display.

The Controller only has five states: StartUp, Locked, Unlocked, ClearState and ChangeModeState.

StartUp - The controller starts up in the StartUp state and waits there for 1000 clock cycles. In this state, it is unresponsive to any signals from the buttons or the wheel sensor or buttons. This StartUp state prevents the FPGA from receiving erroneous input signals that occur during the power up sequence. After 1000 clock cycles, the controller enters the Locked state.

Locked - In the locked state, the controller only responds to the UNLOCK button input. Once the controller receives an UNLOCK signal, it moves into the unlocked state. While the controller is locked, it still passes the ride enable signal from the current speed module onto the other modules.

Unlocked - In the unlocked state, the controller's internal unlocked count enable is high. After twelve seconds of inactivity in the unlocked state, the controller returns to the locked state. If RESET or CHANGE MODE is pressed in the unlocked state, the unlocked counter resets.

If RESET is pressed, the controller enters the ClearState. If CHANGE MODE is pressed, the controller moves to the ChanceModeState.

ClearState - The controller only remains in clear state for one clock cycle. The controller returns to the unlocked state on the next clock edge. In this mode, the clear signal goes high, and the unlocked counter is cleared.

ChangeModeState - The controller also only remains in ChangeModeState for one clock cycle; it returns to the unlocked state after one clock cycle. The unlocked counter

is also cleared in this state. The MuxBits two bit signal is incremented by one step along the sequence 0 -> 1 -> 2 -> 3-> 0 -> ... There are four quantities to multiplex between to feed to the display.

2.4 Construction and Debugging - See Attached VHDL test benches and simulations

We built the bike computer by constructing the functional modules first. For each computational module, we built simple test benches with the clear, ride enable and wheel sensor signals.

When we initially thought about creating our circuit, we decided that starting with the individual modules that calculated total time and total distance would be easiest. The timer seemed the easiest so we started by creating that. One issue we ran into this was wrapping around after reaching 59. We first wrapped when the counter reached 60 but realized that the counter starts at 0 so we adjusted our values. Assuring that it cascaded properly was also an issue.

We then moved on to building the total distance calculator. One issue we ran into this was with the strength reduction needed to convert wheel ticks to miles. When we first thought it was working properly, the test bench failed to match the expected results and we realized that we needed to use hundredths of miles instead of simply miles to properly obtain the total distance travelled. Again, comparing test bench values to pen and paper calculations allowed us to know this module also worked properly before moving to the next module.

After this, we worked on the current and average speed modules, since they were similar in concept, however, we learned that the implementation needed for each one would be very different. For the average speed module, we learned from our

mistakes from the distance module and used thousandths of ticks as opposed to simply ticks to keep our outputs having the correct number of significant figures needed to display properly on the FPGA. The math involved in converting clicks per second to hundredths of miles per hour was messy and keeping the proper bit widths around for signals became a hassle. Comparing test bench results to pen and paper calculations proved that this module worked as well.

When we started building the current speed module, we weren't sure exactly how the digital division would resolve itself. Initially we built a manual division block by that completes a division operation by comparing sum incremented by the divisor to the dividend. This manual operation took as many clock cycles as the magnitude of the quotient to complete. However, we found that the division algorithm built into the "NUMBERIC_STD" library adequately handled the division operations we needed. During the initial coding attempts, we encountered several divide by zero errors. We resolved this by ensuring that divisor never has a zero value. We initialize the divisor to all '1's instead of all '0's as customary. Once we confirmed that test bench and manual calculations matched, we moved onto the controller and top level.

In the first rendition of the controller, we included only two states. They were locked and unlocked. We intended to make the reset and change mode processes to occur within the unlocked state, but this caused unintended latches. We made the ClearState and ChangeModeState in addition to the existing two states. ClearState and ChangeModeState enable the clear signal or the multiplexer increment signal respectively. One of the largest issues we encountered with the controller when transitioning from unlocked back to locked. Once we added the counter signal to the update state process, the controller transitioned correctly. Once we programmed the

controller onto the FPGA, we saw that the start up signals on the FPGA sent erroneous wheel sensor signals. To avoid this, we built in an additional nonresponsive StartUp state that the controller would occupy for the first 1000 clock cycles.

3. Justification and Evaluation

We feel that our design is a particularly effective because the current speed calculation, arguably the most important function of the bike computer, updates with every wheel rotation and display its accuracy to two decimal points. Most other recreational bike computers take several seconds to update the current speed display and only show a single decimal point of precision.

We also think that our bike computer's ride-enable signal functionality makes it more clever than a manual "riding/not riding" switch that the biker could control. By making the computer able to discern when the biker is actually riding versus when they are paused, we've relieved the rider their duty to remember to manually pause and restart the computer while they take a break.

One major flaw in our design is our assumption that the bike wheel has a diameter of 700 mm. This is greatly limits the set of possible users. For example, most mountain bikes have different wheel sizes than 700 mm diameter. A truly flexible bike computer needs to have a manual diameter set option that can account for the user's particular wheel and tire configuration. We would incorporate that functionality in the next iteration of our design.

We were also limited to the single 4-digit, 7-segment display on the FPGA which unfortunately limits the user from observing their current speed and one of the other ride statistics. In many situations, the rider will want to know both their speed and their

ride distance for navigational purposes. If we implemented this bike computer in commercial hardware, we would build in a second 4-digit, 7-segment display solely to display the current speed.

Finally, in the next phase of design, we would have like build a RAM into FPGA so the bike computer could store the accumulated miles of the biker even while the computer is turned off. Most bike computers have an internal odometers that track the total number of miles the biker has ridden as measure of experience. We would add such capabilities to the next prototype.

4. Conclusion:

We successfully built a digital circuit capable of measuring the current speed, ride distance, ride time and average speed of a bicycle. By measuring the time elapsed between wheel rotations, our bike computer calculates these quantities. The computer also discerns when a rider is stopped and holds the ride time constant until the rider resumes movement. The rider can reset the computer's data or switch display modes after unlocking the computer through button inputs.

We implemented nearly all of the functionality we wanted the computer to feature in our original proposal. The computer calculates total distance, total time, average speed, and current speed and features buttons for reset, mode shift, and locking and unlocking the computer. We did make some other changes though: some changes simplified the computer and some made the computer more user friendly. For simplification, we decided to eliminate the separate four-digit display for displaying the current speed and included a current speed display mode. To improve the user-friendliness of the computer, we added LED's to indicate the mode, locked state, and if

they were riding. We also initialized the computer in a locked state and made the 'stop' speed at .5 mph since they both seemed more logical than starting in an unlocked state and having the stop speed be .75 mph as they were in the original proposal. These modifications only slightly alter the final product from the original proposal.

If future groups were to consider tackling this project, we offer a few specific and general tips. In general, our biggest recommendation is to test bench every sub-module thoroughly. With the bike computer, we had to carefully test bench the lower level distance, time, average speed, and speed modules. The math on these portions was tricky. We needed to check the simulation results with meticulous hand calculations. Also, knowing that each of the lower level modules work properly is very important for debugging the top-level file. Trusting our lower level files helped us focus on and find errors in the top level. We would also recommend that each person work on related tasks. For instance, if one person writes modules that interact, one author should write both modules since they intuitively understand how the both modules function. The last piece of advice would be to test the external sensors to understand they work before hooking it up to the FPGA. We had issues with getting a usable signal until we tested it on its own. The same can be said about any technology we haven't used in the class yet. This simple things can save groups a lot of time in the future.

5. Acknowledgements:

We would really like to thank Professor Eric Hansen and Dave Picard for all of their help and advise along the way. We could not have done the project without them and they inevitably helped us avoid making some time-consuming mistakes and led us on the right path.

In terms of distribution of labor for designing the circuit, we started with the general knowledge of what computational black boxes the project needed, so we split those tasks. Josh built three of the computational modules, and Matt wrote the final computational module, the controller and the top level.

6. References:

We relied on Professor Eric Hansen's lecture notes, particularly those pertaining to strength reduction in lecture 21 on July 25, 2014. We also referenced Wikipedia for assistance on implementing the Double Dabble to convert unsigned binary numbers to binary coded decimal. See http://en.wikipedia.org/wiki/Double_dabble for example code.