Homework 8: Adler 32 Accumulator

Ben Heard

Due: 13 April 2022

Abstract

While a larger system to compute Adler32 checksums as an offload engine and conforming to input and output interface specifications is interesting, it is beyond the scope of a simple homework assignment. This document outlines an assignment meant to jumpstart the development of a larger offload engine by working through an implementation of just the base modulo sum accumulator that makes up the Adler32 datapath.

1 Description

In this assignment you'll produce a design and implementation of the accumulator at the heart of the Adler32 checksum engine. As always you are encouraged to draw out the design before attempting to implement it in Verilog. However, it is only your Verilog that will be assessed.

Since this is just the accumulator portion there is no control of the data coming in. Instead, the general function of this design is to reset to a known state (specifically, A=1 and B=0) and, out of reset, accumulate every clock cycle.

There is no output interface or return to a known state once operation has begun. Instead the output simply continues to change as the input data are accumulated each clock cycle.

1.1 Input Interface

The input interface consists of a synchronous active low reset and the 8-bit data. Out of the reset your system should set the value of A to 1 and the

value of B to 0. Then, every clock cycle that reset is not asserted your design shall accumulate the data byte as

$$A_{new} = (A_{old} + data) \bmod 65521$$
$$B_{new} = (B_{old} + A_{new}) \bmod 65521$$

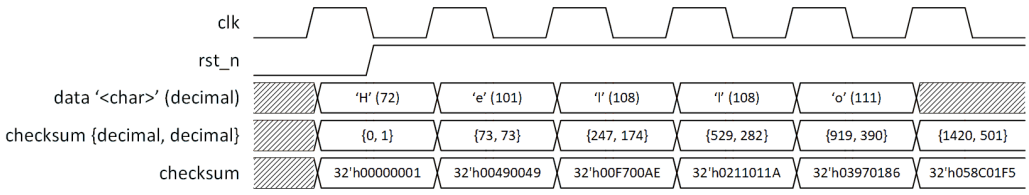
1.2 Output Interface

There is no formal output interface. The intent is that your accumulator concatenate B, A and always present that to the output. So, after reset your output would be $32'h00000001$ and each clock it would update to a new value based on the input data.

1.3 Requirements

Your implementation must conform to the following requirements.

1. The module shall be named **adler32_acc**.
2. The module shall have the following ports
   - input **clk**
   - input **rst_n**
   - input **data**[7:0]
   - output **checksum**[31:0]
3. An active low synchronous assertion of **rst_n** shall set the output to $32'h00000001$ in accordance with the Adler32 specification.
4. At each rising clock edge the accumulator will sum the input data with the value in the 16-bit register A, and then sum that result with the value in the 16-bit register B.
5. There is no end to the data; the accumulator shall continue to sum the input data as long as rising clock edges appear.



1.4 Example

The following timing diagram shows an example operation with the "Hello" string that is part of the project.

2 What to Turn In

Submit a Verilog implementation of an accumulator meeting the above requirements to Moodle. The number of files is not important, only that the top level module be correctly defined.

2.1 Grading

Grading will be based on the following criteria.

- Was anything submitted?
- Did it compile with vlog *.v without errors or warnings?
- Did it load into the simulator with the provided example testbench?
- Did it run with the provided example testbench?
- Did it produce correct results with the provided example testbench?
- Did it run to completion with the hidden testbench(es)?
- Did it produce correct results for the hiddent testbench(es)?

3 Suggestions/Notes

Notice that there is no discussion about how to perform the modulus in this assignment; check the project description for notes on that. Second, notice

that the example testbench doesn't send enough data to see the accumulator exceed the modulus of 65521; you'll want to create some data that will test your design accumulating past that value.

To help with this assignment and the project you may generate Adler32 checksums using an online calculator such as the one found at

`https://hash.online-convert.com/adler32-generator`

Finally, while we have used structurally instanced adders in the past you are free to compute a sum or difference using the Verilog operators + and -. For example, the expression for simple counter could be $count = count + 1;$ as part of a procedural block.