Adler-32 Checksum Offload Engine

Ben Heard

Due: Friday, April 21, 2022
@ 5:00 PM

Abstract

1 Checksums

Perhaps the best description of a checksum comes from Wikipedia.

> "A checksum is a fixed-size datum computed from an arbitrary block of digital data for the purpose of detecting accidental errors that may have been introduced during its transmission or storage. The integrity of the data can be checked at any later time by recomputing the checksum and comparing it with the stored one. If the checksums match, the data were almost certainly not altered (either intentionally or unintentionally).
>
> The procedure that yields the checksum from the data is called a checksum function or checksum algorithm. A good checksum algorithm will yield a different result with high probability when the data is accidentally corrupted; if the checksums match, the data is very likely to be free of accidental errors." (Checksum)

An alternative way to describe the above is to imagine that you are sending a large amount of data and want to make sure that it arrives correctly. One mechanism for demonstrating correctness is to calculate a checksum value as you are sending data and have the receiver compute a checksum using the same algorithm as the data are received. At the end of sending data the sender passes its computed checksum and the receiver compares its computed checksum (over the same data) to the senders computed value. If the values are the same then the data are presumed correct.

A checksum is generally computed as an accumulated function of the previously computed checksum value and the new datum. Figure 1 below shows a very high level structure of a checksum function.

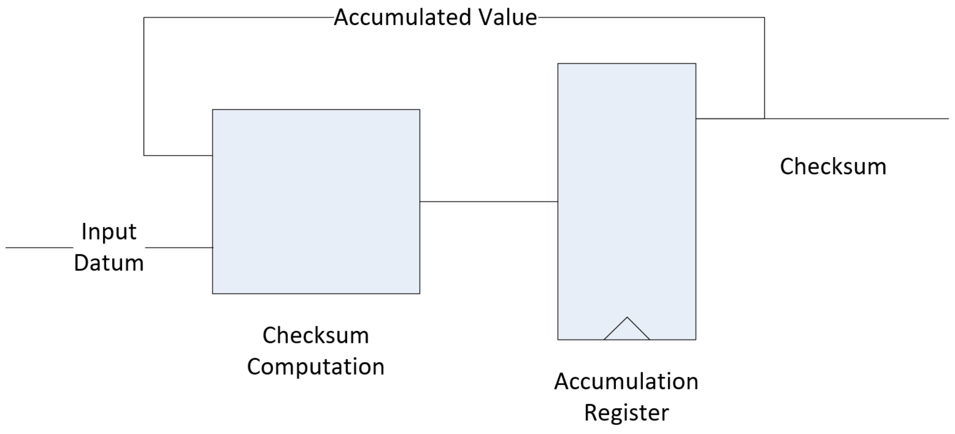

Figure 1: Accumulator Top Level

2 Adler-32 Checksum

The Adler-32 checksum is a checksum that has been developed by Mark Adler as part of the zlib compression library. (Adler-32) If you review the content at Wikipedia about the Alder-32 checksum you'll see that there is a general weakness in the algorithm for small amounts of data. We won't worry about this limitation for this project.

> "An Adler-32 checksum is obtained by calculating two 16-bit checksums A and B and concatenating their bits into a 32-bit integer. A is the sum of all bytes in the stream plus one, and B is the sum of the individual values of A from each step.
>
> At the beginning of an Adler-32 run, A is initialized to 1, B to 0. The sums are done modulo 65521 (the largest prime number smaller than 216). The bytes are stored in network order (big endian), B occupying the two most significant bytes.

Char	Dec	A	B
H	72	1 + 72 = 73	0 + 73 = 73
e	101	73 + 101 = 174	73 + 174 = 247
l	108	174 + 108 = 282	247 + 282 = 529
l	108	282 + 108 = 390	529 + 390 = 919
o	111	390 + 111 = 501	919 + 501 = 1420

Table 1: "Hello" Example

> The function may be expressed as
>
> $$A = (1 + D_1 + D_2 + \dots D_n) \,\%\, 65521$$
>
> $$B = (1 + D_1) + (1 + D_1 + D_2) + \dots (1 + D_1 + D_2 \dots + D_n)$$
> $$B = (n \cdot D_1 + (n-1)D_2 + (n-2)D_3 + \dots D_n) \,\%\, 65521$$
>
> The completed Adler-32 computation may then be expressed as the contatenation of B and A.
>
> $$Adler32(D) = B \cdot 65536 + A$$
>
> where D is the string of bytes for which the checksum is to be calculated, and n is the length of D." (Adler-32)

So, what does this mean? Perhaps we should show an example. You can find another example at Wikipedia. For simplicity we'll compute the Adler-32 checksum for the simple string "Hello" Since this is an example we're starting with the string itself; your project will receive bytes. The computation proceeds in table 1.

We've computed final decimal values for A and B. Let's convert them to hexadecimal and show they are concatenated into a single 32-bit result. A, 501, is 16'h01f5 and B, 1460, is 16'h058c. The Adler-32 result is the concatenation of A to B where B is the most significant 16-bit quantity. So, the final result of B, A is 32'h058c01f5.

3 System Description

You have been tasked to create an Adler-32 offload engine. As part of a larger system you will be providing resources to compute the Adler-32 checksum on data being sent to you a byte at a time. The process proceeds as follows.

- The source will provide a count of the number of bytes to be accumulated.
- Following a start pulse a source will begin to send you bytes, one at a time.
- As the engine receives a valid byte it accumulates the Adler-32 checksum.
- Once the number of bytes expected has been received the checksum computation is complete.
- With the computation complete, the engine will provide the result.

There is no indication of the last byte that is being sent; your machine will have to track this information from the number of bytes that was provided. You are assured that no size will come while data bytes are begin transferred and for at least 10 clock cycles following. There is no guarantee in the number of cycles between a size and the first byte; it may come in the next clock cycle or many clock cycles later.

3.1 What to turn in

3.1.1 Implementation

Submit your Verilog files using the submit utility on Wolfware. It is unimportant how many Verilog files you submit or their names. The only things that is important is that there is a module called **adler32** and that that module contains the functionality of your system and that all files that make up your system have the .v extension. You needn't submit a testbench as your project will be graded based on its performance with a testbench of my own. The graders will use "vlog *.v" to compile the code that you submit so make sure that this command will compile your **adler32** top level module, everything that is required for the top level module, and that it completes successfully.

3.2 Module Name and Port List

Your module should have the signals listed in table 3.2 and the name **adler32**.

Signal	Direction	Description
clock	input	synchronous clock input
rst_n	input	active low synchronous reset
size_valid	input	control signal for transferring size
size	input[31:0]	size of the message data
data_valid	input	control signals for transferring bytes
data	input[7:0]	input byte of message data
checksum_valid	output	transfer a checksum out
checksum	output[31:0]	output checksum

Table 2: Top Level Ports

3.3 Input Interface

The input interface to the offload engine is broken into two parts. The first transfers the size of the data to the offload engine. The second passes the actual data.

3.3.1 Transferring Message Size

The input to transfer the message size to your offload engine is described as follows. A portion of a timing diagram for capturing size is given in Figure 2.

1. An input size, 32-bits wide, is driven to contain the number of bytes that the offload engine shall process for the next checksum.
2. Coincident with size, another input, size_valid, is asserted to indicate that message size being presented is valid and represents the number of bytes of the next message.
3. At the next rising clock edge, with size_valid asserted, size is transferred to your offload engine.

3.3.2 Transferring Data

Some time following the transfer of the message size the sender will begin to assert data_valid for each data byte that it sends. A data byte should only be accepted into the checksum engine in the same clock cycle that data_valid is asserted. The process continues until all data bytes have been sent, as known from the size provided at the beginning of the transaction. Note

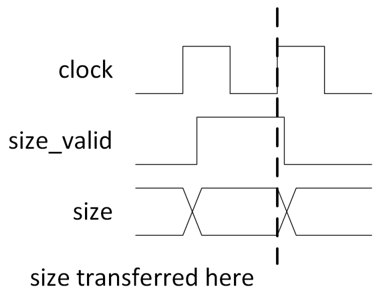

Figure 2: Input Timing Diagram

that there is no indication of the end of data; this must be tracked by your engine.

See Figure 3 as an example timing diagram. Note that while the diagram shows the checksum and checksum_valid the timing of the checksum output relative to the last byte of data needn't be the next cycle.

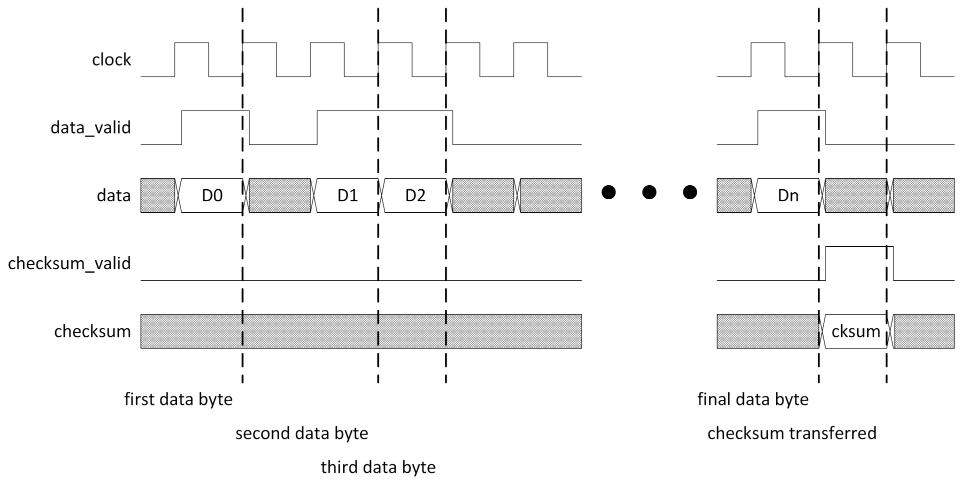

Figure 3: Data Timing Diagram

3.4 Output Interface

The output interface looks the same as the message size transfer input interface. The offload engine asserts checksum_valid along with a computed checksum. See Figure 4.

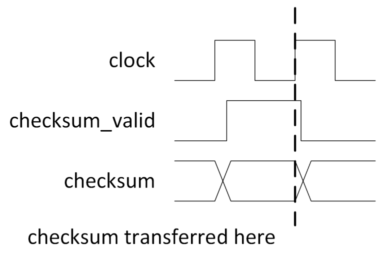

Figure 4: Output Timing Diagram

3.5 Additional Requirements

3.5.1 Modulus

There is a modulus operator in Verilog (%) that will provide the remainder to an unsigned integer division. However, we can't know how this will synthesize. I.e. what will the logic look like that a synthesis tool creates to implement the modulus. Therefore, **you may NOT use the modulus operator (%)** in your implementation. You must find another way, one whose implementation you can explain, to implement the modulus. See the hints below.

4 Some Hints

4.1 Addition Modulo 65521

Addition modulo anything is the computation of the sum of two values followed by taking the remainder if the sum is divided by the modulus. Since the modulus is computed after each addition it can only ever be the sum itself (if the sum is less than the modulus) or the sum minus the modulus (since the addition of an 8-bit quantity to a 16-bit quantity can't be greater than 2 times the 16-bit quantity). See the following examples (base 10).

$$\begin{aligned}
val &= (3+8) \bmod 50 \\
val &= (11) \bmod 50 \\
val &= 11 \\
\\
val &= (47+6) \bmod 50 \\
val &= (53) \bmod 50 \\
val &= 53 - 50 = 3
\end{aligned}$$

4.2 Hierarchy

I recommend that you implement your system the way that we've been implementing systems in class. Start with a drawing of the top level (a single box with all the inputs and outputs shown). Then move on to the second level showing a datapath block and a controller. There won't be signals between them yet.

Draw out the logic for the datapath. As you do you'll identify new signals that are needed from the controller to determine when to accumulate your data. These will be the signals between the datapath and controller. Next, draw your state diagram showing all states, the transitions between them, and the value of the outputs at all time.

Finally, take the drawing of the datapath and the state diagram and write the Verilog against those. In that way you'll have a clear understanding of how the system should work.

5 Rubric

The following rubric is **incomplete** but represents what we'll be looking for in the grading. Too, I've not assigned points but know that the majority of points will come from whether your implementation provides correct results to the input stimulus. A couple of reminders:

1. You may not use the modulus operator in your design
2. There is no report required for this assignment

The following are references that will be used for grading.

1. Did you submit anything?

2. Was it submitted on time (-10 points per day late)
3. Does it compile with vlog *.v?
4. Hello_sim_tb.v
    (a) Will it load into the simulator?
    (b) Will it simulate at all?
    (c) Does it run to completion?
    (d) Does it provide the correct result?
5. Multi_msg_tb.v
    (a) Does it run to completion?
    (b) Does it provide correct results?
6. Hidden short testbench
    (a) Does it run to completion?
    (b) Does it provide correct results?
7. Hidden long testbench
    (a) Does it run to completion?
    (b) Does it provide correct results?
8. Does your implementation use the modulus operator (-40 points)

As you can see there are opportunities for points even if the implementation doesn't provide correct results; please submit something. However, the Academic Integrity requirements hold; do not submit someone else's work just to submit something.

6 References

Wikipedia contributors. "Checksum." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 21 Feb. 2012. Web. 25 Feb. 2012.

Wikipedia contributors. "Adler-32." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 14 Feb. 2012. Web. 25 Feb. 2012.