Adler-32 Checksum

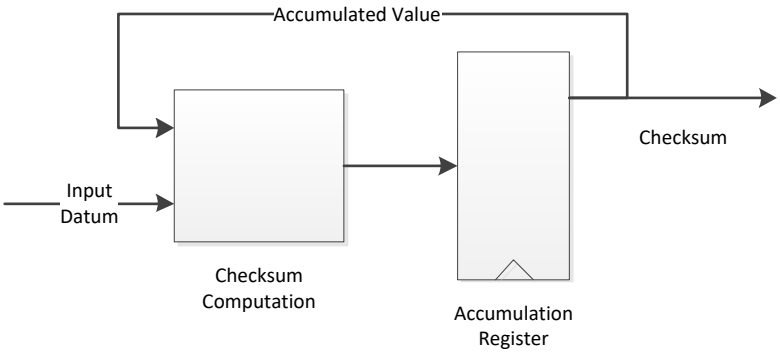
1.0 Checksums

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. The figure below shows a very high level structure of a checksum function.



2.0 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 2^{16}). The bytes are stored in network order (big endian), B occupying the two most significant bytes.
>
> The function may be expressed as
>
> $$A = 1 + D_1 + D_2 + \ldots + D_n \ (mod\ 65521)$$
>
> $$B = (1 + D_1) + (1 + D_1 + D_2) + \cdots + (1 + D_1 + D_2 + \cdots + D_n) \ (mod\ 65521)$$
> $$B = n \cdot D_1 + (n-1) \cdot D_2 + (n-2) \cdot D_3 + \cdots + D_n \ (mod\ 65521)$$
>
> $$Adler32(D) = B * 65536 + A$$
>
> 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 the table below.

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

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.

3.0 Project 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.

1. The source will provide a count of the number of bytes to be accumulated.
2. Following a start pulse a source will begin to send you bytes, one at a time.
3. As the engine receives a byte it accumulates the Adler-32 checksum.
4. Once the number of bytes expected has been received the checksum computation is complete.
5. In the following clock cycle the engine will indicate a valid checksum computation.

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 through your repository. It is unimportant how many Verilog files you submit or their names. The only things that are important are 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 following signals and the name **adler32**.

clock	input	synchronous clock input
rst_n	input	active low synchronous reset
size_valid	input	control signal for transferring message size
size	input[31:0]	bytes in the next message
data_start	input	control signal for transferring the first byte
data	input[7:0]	input byte of message data
checksum_valid	output	control signal for transferring a checksum
checksum	output[31:0]	output checksum for most recent message

3.3 Input Interface

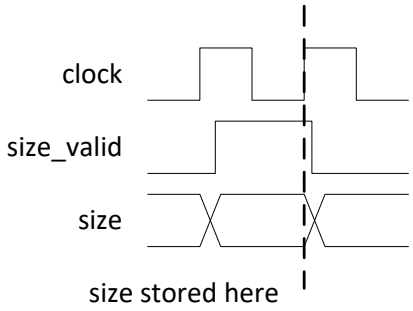
The input interface to the offload engine is broken into two parts. The first transfers the size of the data to the offload engine. The second passes the actual data.

3.3.1 Transferring Message Size

The input to transfer the message size to your offload engine is described as follows.

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.

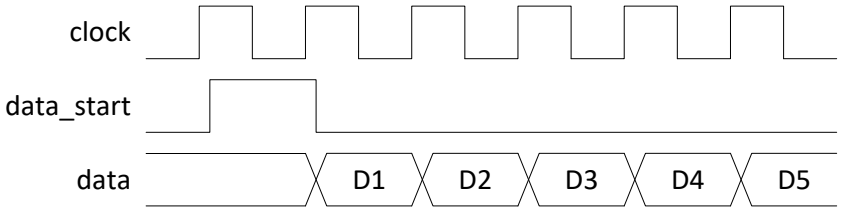
2. Coincident with **size**, another input, **size_valid**, is asserted to indicate that message size being presented is valid and represents the number of bytes of the next message.
3. At the next rising clock edge, with **size_valid** asserted, **size** is transferred to your offload engine.



3.3.2 Transferring Data

Some time following the transfer of the message size the sender will assert a pulse on **data_start** to indicate the beginning of the data bytes. In the clock cycle following the **data_start** pulse, the first **data** byte will appear. In each successive clock cycle additional **data** bytes appear. The process continues until all **data** bytes have been sent, as known from the **size** provided at the beginning of the transaction. Note that there is no indication of the end of **data**; this must be tracked by your engine.

See the figure below as an example 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.

4.0 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).

$$val = (3 + 8)\%\ 50$$
$$val = (11)\%\ 50$$

$$val = 11$$

$$val = (47 + 6)\ \%\ 50$$
$$val = (53)\%\ 50$$
$$val = 53 - 50 = 3$$

4.2 Hierarchy

You may want to consider building the system hierarchically with a separate module for the controller and data path. In this way you can test your each piece before assembling the entire portion.

5.0 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.