Luhn mod N Checker (N = 16)

1.0 Luhn Algorithm

Perhaps the best description of the Luhn algorithm comes from Wikipedia.

> "The Luhn algorithm or Luhn formula, also known as the "modulus 10" or "mod 10" algorithm, is a simple checksum formula used to validate a variety of identification numbers, such as credit card numbers, IMEI numbers, National Provider Identifier numbers in US and Canadian Social Insurance Numbers. It was created by IBM scientist Hans Peter Luhn and described in U.S. Patent No. 2,950,048, filed on January 6, 1954, and granted on August 23, 1960.
>
> The algorithm is in the public domain and is in wide use today. It is specified in ISO/IEC 7812-1.[1] It is not intended to be a cryptographically secure hash function; it was designed to protect against accidental errors, not malicious attacks. Most credit cards and many government identification numbers use the algorithm as a simple method of distinguishing valid numbers from collections of random digits." (Luhn Algorithm)

Specifically, the original Luhn algorithm is a base 10 operation. Implementing the modulus 10 in hardware is actually quite difficult and adds an unnecessary level of complexity to an ECE310 project. There is, however, a more general form of the algorithm called the Luhn mod N algorithm. Again, from Wikipedia.

> "The Luhn mod N algorithm is an extension to the Luhn algorithm (also known as mod 10 algorithm) that allows it to work with sequences of non-numeric characters. This can be useful when a check digit is required to validate an identification string composed of letters, a combination of letters and digits or even any arbitrary set of characters." (Luhn mod N Algorithm)

Choosing a mod 16 variant of the Luhn algorithm greatly simplifies an implementation of a hardware checker. Computations mod 16 become simple right shifts to remove the 4 least significant bits.

2.0 The Luhn mod 16 Checking Algorithm

Assume that you have a sequence of 8 **hexadecimal** values (4-bits, nibbles).

	Data	Double	Sum Digits	Final Values
First nibble	A	14	1+4=5	5
	3			3
	D	1A	1+A=B	B
	C			C
	1	2		2
	5			5
Start here →	9	12	1+2=3	3

Check nibble	7			7
			Sum	30

Starting with the nibble just before the check nibble, double (*2, shift left by 1, etc.) every other nibble. In the example above this means doubling 9, 1, D, and A. If the doubling of a nibble results in carry out then sum the two digits of the result to yield a nibble. For example, in the case above when D is doubled (4'b1101 << 1 = 5'b11010) the result is 5'h1A. Summing the two nibbles of the result yields 4'hB.

Once the doubling and summing of digits (if necessary) is complete then all of the result nibbles are added together. In the case above, 5+3+B+C+2+5+3+7 = 30. The sequence of nibbles passes the Luhn mod 16 check if the least significant nibble of the summed result is zero (4'b0000).

2.1 Failed Example

The example below shows an 8 hexadecimal sequence that fails the Luhn mod 16 check.

	Data	Double	Sum Digits	Final Values
First nibble	6	C		C
	9			9
	4	8		8
	3			3
	2	4		4
	1			1
Start here →	A	14	5	5
Check nibble	B			B
			Sum	35

Here, the final sum is 35 whose least significant nibble is 5 != 0.

2.2 Odd Length Example

Finally, here is an example of a sequence of 7 hexadecimal bytes.

	Data	Double	Sum Digits	Final Values
First nibble	4			4
	9	12	3	3
	B			B
	C	18	9	9
	3			3
Start here →	D	1A	B	B
Check nibble	7			7
			Sum	30

There is really no difference in the computation. Here, however, notice that the first nibble is not one that gets doubled. Doubling always proceeds from the nibble adjacent to the check nibble.

3.0 Project Description

You have been tasked to create a Luhn mod 16 Checker. As part of a larger system you will be providing resources to identify whether a sequence of nibbles pass a Luhn mod N check. The process proceeds as follows.

1. Your offload engine will accept, as input, the size of the next message, in nibbles (4 bit quantities).
2. A source will begin to send you nibbles, one at a time.
3. As the engine receives a nibble it processes it according to the checker.
4. When the engine has received the number of nibbles in the message the engine outputs whether the sequence of nibbles passes the Luhn mod 16 check.

There is no indication from the source that there are no more data coming once you've exhausted the number of nibbles in the message size; this is something that your machine will need to track. A size for the next message may come at any time during the transfer of data and your engine must accept up to 4 sizes before indicating that it is unable to accept any more.

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 thing that is important is that there is a module called **luhnmod16** and that that module contains the functionality of your system. 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 **luhnmod16** 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 **luhnmod16**.

clock	input	synchronous clock input
rst_n	input	active low synchronous reset
size_valid	input	control signal for transferring message size
size_ready	output	control signal for transferring message size
size	input[7:0]	nibbles in the next message
data_valid	input	control signal for transferring a nibble
data_ready	output	control signal for transferring a nibble
data	input[3:0]	input nibble of message data
check_valid	output	control signal for transferring a check
check_ready	input	control signal for transferring a check
check	output	indication of valid

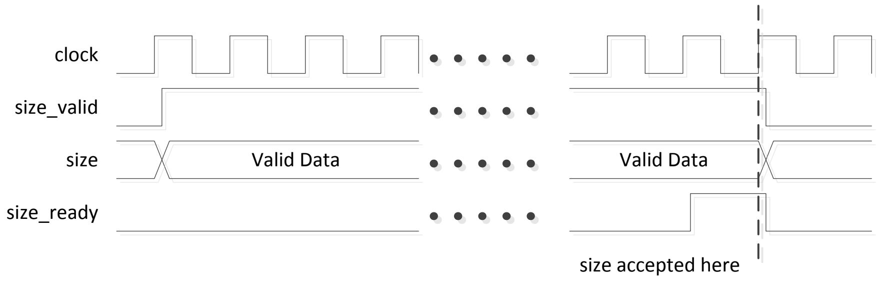
3.3 Input Interface

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 Size

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

1. An input **size**, 8-bits wide, is driven to contain the number of nibbles that the offload engine shall process for the next check.
2. Coincident with **size**, another input, **size_valid**, is asserted to indicate that the size being presented is valid and represents the number of nibbles of the next message.
3. The system sending **size** and **size_valid** then waits until an output from your block, **size_ready**, asserts.
4. When **size_valid** and **size_ready** are asserted in the same cycle, the **msg_size** is presumed to have been accepted by your offload engine.



A couple of notes about this interface follow.

1. **size_valid** will not assert if **size_ready** is active.
2. Once **size_valid** asserts it will not go low nor will the contents of **size** change until **size_ready** asserts and the data are transferred.
3. **size_ready** must not assert until **size_valid** has been asserted. The assertion of **size_valid** will always start the exchange of **size** data.
4. **size_valid** and **size_ready** must go low in the cycle following the data exchange.
5. **size_valid** and **size_ready** must remain low for at least one cycle following the data exchange.
6. The time between **size_valid** asserting and **size_ready** asserting is arbitrary and may be as long as necessary to allow the offload engine time to service the current checks.

Recall that you must be able to store up to four (4) outstanding message sizes. The best way to store these is in a first-in first-out queue (FIFO). The FIFO is loaded as part of this exchange and the size is removed to be used for the transfer of data.

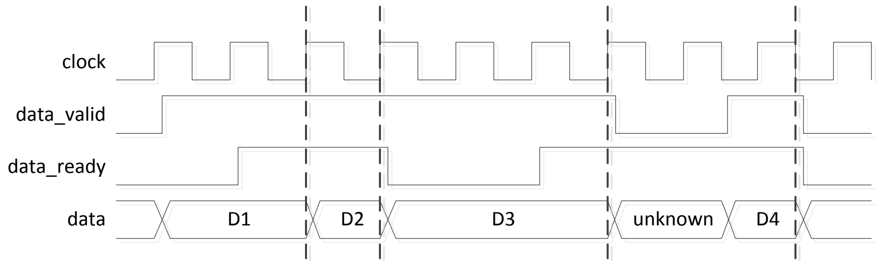
3.3.2 Transferring Data

The transfer of data works in much the same way as the transfer of the message size. However, it is a higher performing interface in that **data_valid** needn't wait on **data_ready** to be low before sending the next byte of data. Therefore, **data_ready** may assert before **data_valid**. However, it is only when

data_valid and **data_ready** are asserted in the same cycle that data are passed. A key note about the data interface is below.

1. **data_valid** will not assert before the size transaction for which the data are being sent.

See the figure below as an example timing diagram. In the diagram note that the dashed lines indicate where data are transferred to the offload engine.



3.3.3 Continuous Data

From above your offload engine must be able to store 4 outstanding sizes. This means that incoming data may be continuous and the first byte of data for a message could be presented immediately following the last byte of a previous message. Your offload engine needn't accept it right away if it's not ready.

3.3.4 Order of Data

The order that the data arrives at the input interface is important since you don't double every nibble, just every other beginning at the nibble adjacent to the check nibble. The data arrive as first nibble first and check nibble last. This means that you will need to discern which nibbles to double based on the number of nibbles you will receive. For example, for even numbers of nibbles you'll double the first, third, fifth, etc. and for odd numbers of nibbles you'll double the second, fourth, sixth, etc.

3.3.4.1Even Length Example

The following is the first example from above. The accumulation is down the right side and the sequence is identified as valid at the end of the process.

Remaining	as Binary	Nibble	Double?	Result	Sum	Accumulation	OK
8	00001000	A	yes	14	5	5	-
7	00000111	3	no	3	3	8	-
6	00000110	D	yes	1A	B	13	-
5	00000101	C	no	C	C	1F	-
4	00000100	1	yes	2	2	21	-
3	00000011	5	no	5	5	26	-
2	00000010	9	yes	12	3	29	-
1	00000001	7	no	7	7	30	yes

3.3.4.2Odd Length Example

The following is the third example from above. The accumulation is down the right side and the sequence is identified as valid at the end of the process.

Remaining	as Binary	Nibble	Double?	Result	Sum	Accumulation	OK
7	00000111	4	no	4	4	4	-
6	00000110	9	yes	12	3	7	-
5	00000101	B	no	B	B	12	-
4	00000100	C	yes	18	9	1B	-
3	00000011	3	no	3	3	1E	-
2	00000010	D	yes	1A	B	29	-
1	00000001	7	no	7	7	30	yes

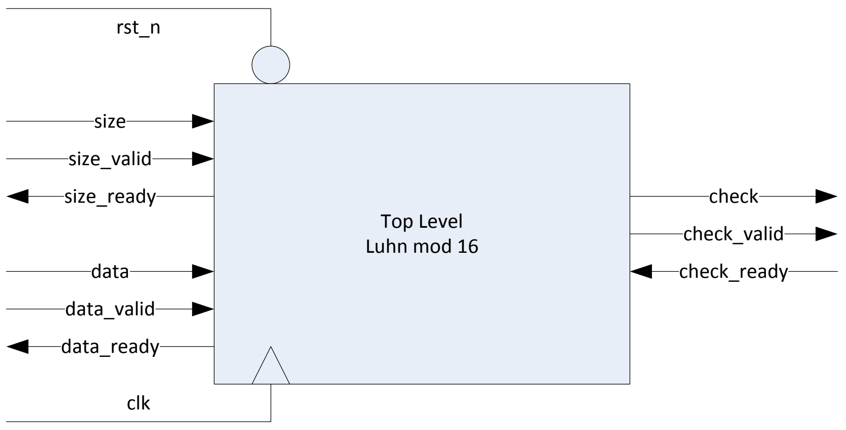
3.4 Output Interface

The output interface looks the same as the size transfer input interface. The offload engine asserts **check_valid** when it has an output to send. It must not assert **check_valid** if **check_ready** is already high; it must wait for **check_ready** to be low. Once **check_valid** is asserted then the engine waits for **check_ready** to assert. In the clock cycle that **check_valid** and **check_ready** are both high, the indication of valid is transferred. Both **check_valid** and **check_ready** must go low immediately and stay low for at least one clock cycle before another exchange may take place.

4.0 Block Diagrams

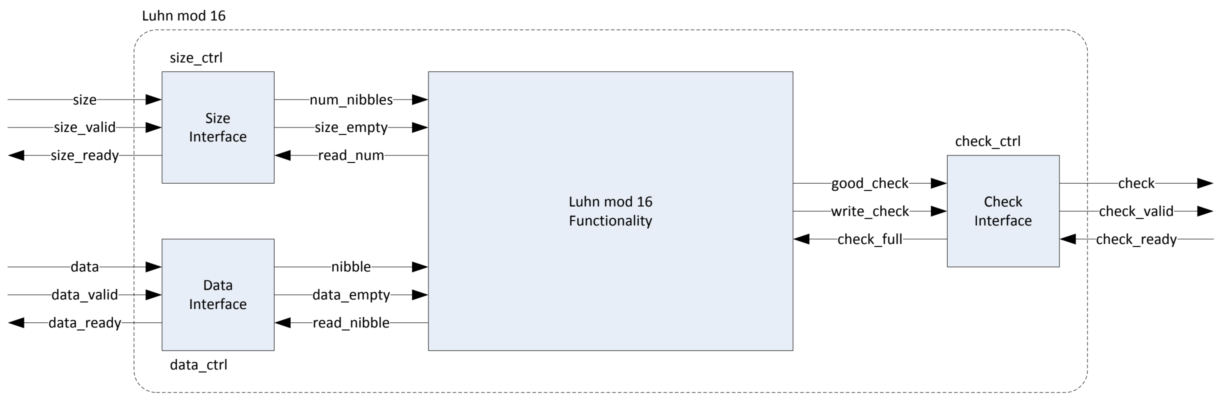
4.1 Top Level Block Diagram

The following is the top level block diagram for the Luhn mod 16 checker. Notice the three interfaces along with the active low synchronous reset and the input clock.



4.2 First Level Decomposition

The first level decomposition of the top level block diagram is shown below. Notice that the three interfaces to the "outside world" are made into their own blocks of functionality. These blocks implement the required interfaces to the outside and provide a much simpler FIFO interface to the internal Luhn functionality.



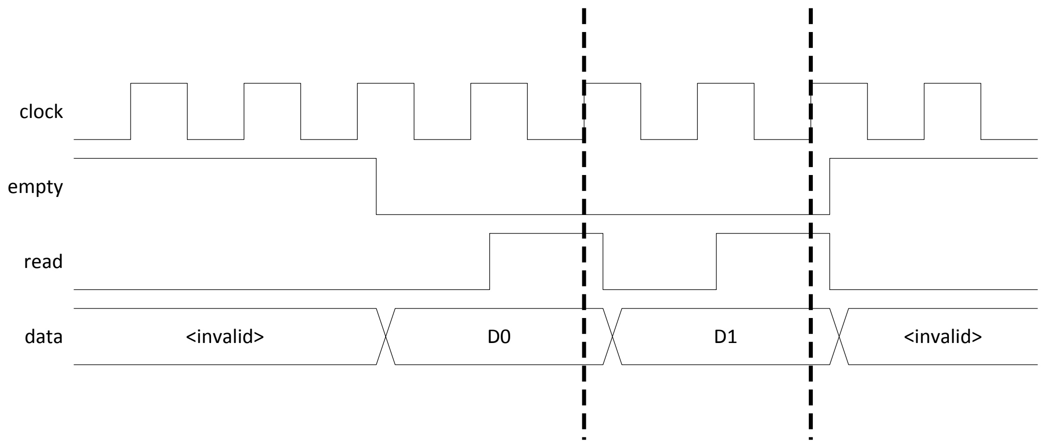
A portion of this project is to have you integrate someone else's Verilog code into your own final system. I have built the necessary FIFOs and state machines to manage the input and output interfaces described above. I.e. you won't need to meet the above interfaces unless you choose to implement the entire design on your own.

Instead, you may build a smaller machine and interface to the FIFOs that I provide. The FIFO interface is very simple and removes timing issues with that make the above interfaces challenging to implement.

4.2.1 FIFO Interface

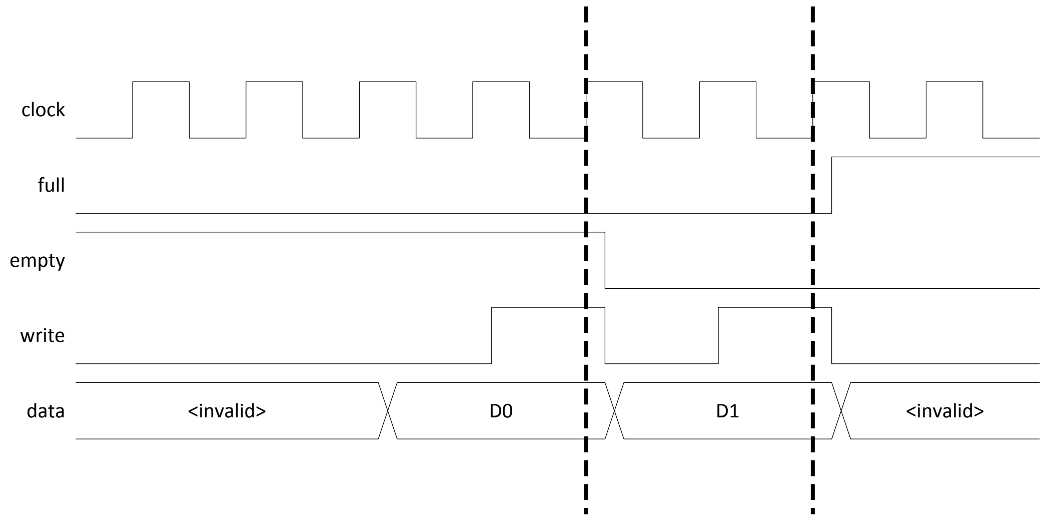
4.2.1.1Reading From a FIFO

Reading from a FIFO can be seen in the figure below. Until the 3rd rising clock edge the FIFO is empty and has no data to provide. When empty asserts low the FIFO is no longer empty and is reflecting the data, D0, on its output. The first read from the FIFO (the read of the data D0) occurs at the 5th rising clock edge. On its write interface (not shown) another entry must have been written for the FIFO to have not gone empty upon reading. Another read occurs taking the value D1 and the FIFO then becomes empty.



4.2.1.2Writing To a FIFO

Writing to a FIFO is very similar. In the figure below you are to assume that the FIFO queue is 2 deep (that is to say that it is capable of storing, at most, 2 elements). Initially, the FIFO is empty. At the 5th rising clock edge the first element is written into the FIFO and it is no longer empty. Two cycles later another element is written and the FIFO becomes full; it is no longer able to accept any more data until an element is read out.



5.0 Some Hints

5.1 Design Guidance

You are to create a Verilog module that instantiates the three controllers that have been provided. In addition, you should create the Luhn mod 16 checking functionality. Structurally, you will connect those four modules together into your top level **luhnmod16** module. You may name the internal connections between the provided modules and your functionality however you choose.

5.1.1 Protected IP (Intellectual Property)

A common business practice is for small companies to develop Verilog modules that satisfy a particular need; for example, USB controllers, memory controllers, video encoders/decoders, etc. Verilog modules that you have been working with are written in a plain text file. Small companies could not remain viable if their intellectual property were unencrypted source (text files). Modelsim offers a mechanism for protecting Verilog modules in an encrypted way. I have used that mechanism to protect the modules that I have created for your use.

These files come as size_ctrl.vp, data_ctrl.vp, and check_ctrl.vp. They may be compiled in the same way as any other Verilog module, with % vlog size_ctrl.vp, etc. Once compiled into your work library you may instantiate the modules in your own designs and use them according to the module interfaces described below.

5.1.2 size_ctrl

The module size_ctrl implements the size interface on one side and a FIFO read interface on the other. You will connect the input interface to the top level size input interface and the FIFO output interface to your module. The module implements the following module header.

```
module size_ctrl(
  input             ck, rst_n,

  /* size valid/ready interface         */
  input             size_valid,
  output reg        size_ready,
  input       [7:0] size,

  /* FIFO output interface              */
  input             read_num,
  output            size_empty,
  output      [7:0] num_nibbles
);
```

5.1.3 data_ctrl

The module data_ctrl implements the data interface on one side and a FIFO read interface on the other. You will connect the input interface to the top level data input interface and the FIFO output interface to your module. The module implements the following module header.

```
module data_ctrl(
  input         ck, rst_n,

  /* data input valid/ready interface     */
  input         data_valid,
  output        data_ready,
  input   [3:0] data,

  /* fifo read data interface             */
  input         read_nibble,
  output        data_empty,
  output [3:0] nibble
);
```

5.1.4 check_ctrl

The module check_ctrl implements the check interface on one side and a FIFO write interface on the other. You will connect the input interface to the top level check output interface and the FIFO input interface to your module. The module implements the following module header.

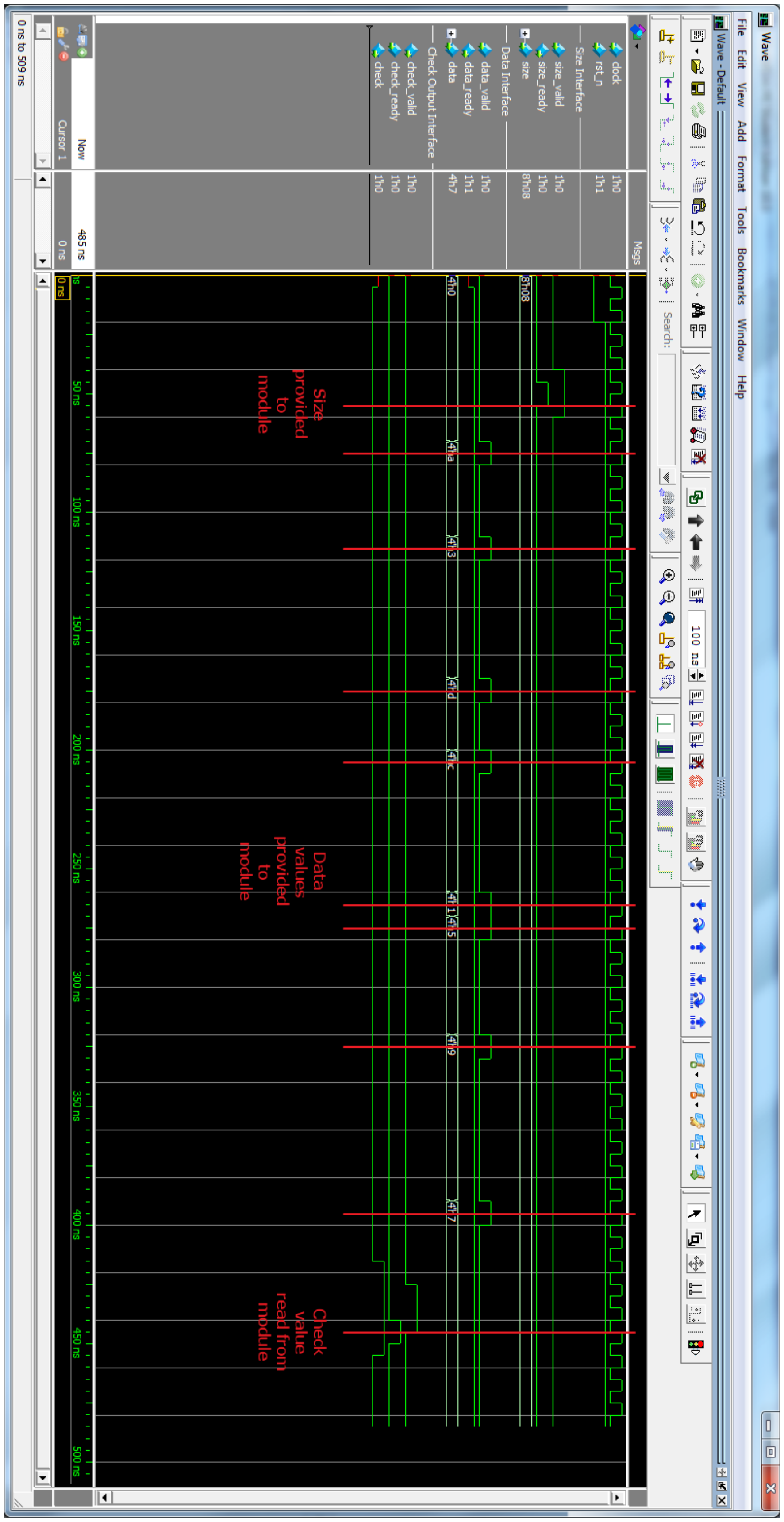
```
module check_ctrl(
  input       ck, rst_n,

  /* output valid/ready interface        */
  output reg check_valid,
  input       check_ready,
  output      check,

  /* input FIFO interface               */
  input       write_check,
  output      check_full,
  input       good_check
);
```

5.2 To Double or Not To Double

Notice that, as you maintain the decrementing number of nibbles remaining (to receive) the least significant bit tells you whether you double the incoming nibble. A LSb of 0 indicates that you should double and a LSb of 1 indicates that you shouldn't. Be careful in this implementation to be assured that your interpretation of the bit comes at the right time.

6.0 Simulation

The Even Length Example of section 3.3.4.1 is shown in the simulation below. This simulation shows the external interfaces as opposed to the internal FIFO interfaces you will have to integrate. The source for this simulation is provided on the project page.


Wave - Default
clock
rst_n
Size Interface
size_valid
size_ready
size
Data Interface
data_valid
data_ready
data
Check Output Interface
check_valid
check_ready
check
Now 485 ns
Cursor 1 0 ns
Size provided to module
Data values provided to module
Check value read from module
0 ns to 509 ns

7.0 References

Wikipedia contributors. "Luhn algorithm." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 14 Feb. 2014. Web. 9 Mar. 2014.

Wikipedia contributors. "Luhn mod N algorithm." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 5 Dec. 2013. Web. 9 Mar. 2014.