NTLM Brute-Force Hash Cracker

Full Project Proposal

Clayton Brandt

Darius Elliott

Adit Ghosh

Tom Yurek

ECE 337

Friday 11:30 a.m.

1/3/2015

Yunus Akhtar

Signatures: Clayton Brandt

Darius Elliot

Adit Ghosh

Tom Yurek

Executive Summary

Our team is proposing to design a Brute-Force NTLM Cracker. Given an NTLM hash or multiple hashes, our system will, given enough time, output the plain text “password” that was used to create the hash or output a list of passwords given a list of hashes. This project is useful for penetration testers and other security researchers. Given access to a physical Windows system, a pen tester could extract the NTLM hashes for the passwords of local users. The system that we are trying to design could assist in the calculation of the plain text passwords that generated the inputted hashes. Then, implementing the given password, a pen tester could test other systems for password reuse and collect information on the security of the passwords and usage.

One feature of our design is the ability to take in a list of hashes and it will output a list of passwords that are matching to the given inputted hashes. This is unique because hashes are typically thought to be irreversible, conducting a brute force method can be effective in finding a list of passwords if they are sufficiently short. The NTLM hash is fairly resource-intensive to calculate, and is designed such that there is only one password of a reasonable length that can generate each hash. Both of these add to the difficulty of using a brute force method of hash cracking.

This idea is more implementable on an ASIC compared to a microprocessor because a proper ASIC design can be much faster. As the password length grows so does the number of calculations needed for this method to be effective. Because we're doing this with brute force, an unknown amount of possibilities need to be calculated and tested. Therefore by running this on an ASIC we are able to achieve our output much faster.

For the design to be effective, it must fall under certain requirements and specifications. The most important requirement is that our outputs are accurate. If the password generated doesn't match the inputted hash, then the system is useless. The next requirement is our system must be relatively fast. The main reason for implementing the design on an ASIC is speed considerations, so it must perform much faster than a microprocessor-based solution.

In the following pages of the report are several different specifications. Among the list include: a high-level block diagram for the system and its usage, descriptions of the different functional blocks, a table of all inputs and outputs of the chip, an RTL block diagram of our design, a description of the requirements of the chip design, and finally an architectural block diagram of the chip.

Design Specifications

System Usage Diagram:

MOSI

MISO

SS

SCLK

NTLM

cracker

Master

Operational Characteristics:

External Signals

|  |  |  |  |
| --- | --- | --- | --- |
| Signal Name | Type:  In/Out/Bidir. | Number of Bits | Description |
| CLK | In | 1 | Clocking signal to oscillate high or low to coordinate memory system elements |
| RST | In | 1 | Asynchronous reset for all memory systems in the design. |
| SCLK | In | 1 | Synchronizes communication between master and slave SPI modules |
| MOSI | In | 1 | Master input data that is transferred from the input file to the slave receiver |
| MISO | Out | 1 | The Slave inputs to the master what the final output is from the calculated hashed values |
| SS | In | 1 | Decides whether you are inputting or outputting |
| Power | In | - | Provides power to the system. |
| Ground | In | - | Provides GND to the entire system. |

Internal Signals

|  |  |  |
| --- | --- | --- |
| Signal Name | Number of Bits | Description |
| byte\_full | 1 | Lets the Controller know that the SPI Slave has received a full byte and that in\_byte can be processed |
| in\_byte | 8 | Holds a byte of input data to be read by the Controller |
| out\_byte | 8 | Holds a byte of output data to be shifted out by the SPI Slave |
| request\_byte | 1 | Lets the Controller know that out\_byte can be overwritten with the next byte of output data |
| num\_hashes | 6 | Keeps track of how many hashes (up to 64) are being checked in this run of the system |
| RW | 1 | Lets Storage know whether it should be reading or writing |
| data\_in | 128 | Data to be written to SRAM in write mode |
| slot | 7 | Which address (0-127) the SRAM should be interacting with |
| data\_out | 128 | Data that has been read from SRAM |
| match\_found | 1 | Lets the controller know if the current hash being checked matches with the hash of the current guess |
| start\_bit | 1 | Lets Generate Guesses know when to increment its guess |
| current\_guess | 128 | The current ASCII string that is being hashed and compared. This size allows for strings up to 16 characters with a byte per character and was chosen to be the same size as an NTLM hash. |
| out\_hash | 128 | The output 16-byte NTLM hash of current\_guess |

Design Block Descriptions:

**Calculate NTLM Hash:**   
This combinational block takes an input of an ASCII string given as a sequence of binary values which represents the current guess of the “Generate Guesses” block. It then implements the MD4 message-digest algorithm to create a 16-byte output hash. The output hash is then compared to the stored hashes that were initially inputted to the system.

**Generate Guesses:**   
This sequential block will generate potential passwords that could be recreated using a hash algorithm to compare with our inputted hashes. The block takes the previous guess (an ASCII string) and output the next guess after a clock cycle. For example, if the previous guess was “ArB7g”, the next guess could be “ArB7h”. The block will need to be able to only generate ASCII strings that are likely to be seen in a real system. As such, it should skip over the first 32 ASCII values.

**Compare:**   
This combinational block takes each input hash and compares it with the output of the NTLM block. This serves to compare the current password guess to any of the hashes that have been inputted to the system. If there is a match, it indicates which hash matches and outputs that hash’s number so that the system can store the current password with the correct corresponding hash.

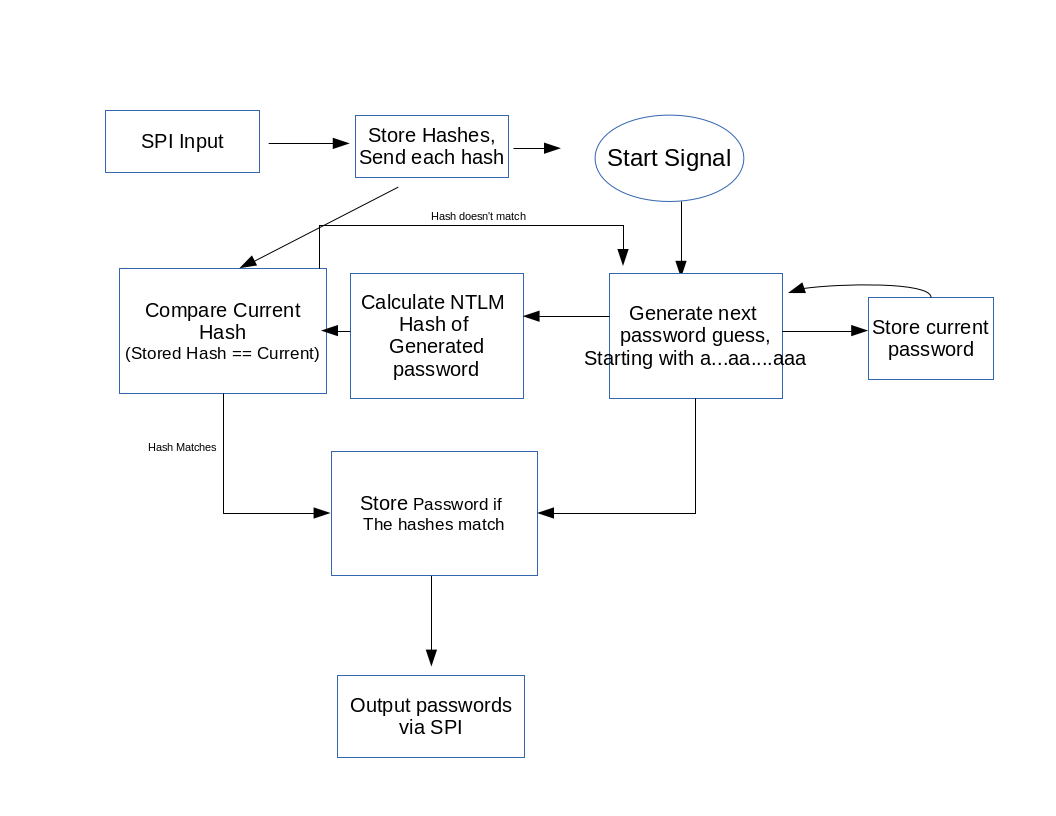
**Storage:**   
This combinational block will first take the input from the SPI via the Controller and store each inputted hash. After the initial loading phase, this block can store and retrieve hashes along with any cracked values the system has found. The desired data is fed back to the Controller and the Compare block. The SRAM this block will have 128 addresses, each containing 128 bits of information. This is to allow for the storage of 64 128-bit hashes and 64 128-bit cracked values.

**SPI Slave:**   
The entry and exit point for data in our system. This sequential block takes a serial input over SPI and translates it into the hashes that will be checked by our system. When outputting data, it takes the calculated passwords from the storage block and shifts them out. This block will also communicate to the Controller signaling when the device is being used for input or output via the slave select line and the request byte line, informing the Controller of what task, if any, for the SPI Slave it needs to perform.

**Controller:**

The controller is going to receive a serial in from the SPI Slave that contains the hashes to the controller. The slave select will be used to tell whether the device is being communicated with by the Master or not. It will also take in a signal that tells the controller whether the Slave is being used for input or output back to the Master. If the controller is being used to input, it will send all data from the SPI and send it to Storage. After done receiving inputs, the controller will then send out a start bit to the Generate Guesses block to start generating passwords. The Controller will also receive a match found signal from the compare block and then it will decide whether or not to save the current guess for a password to the current hash being compared by sending this data to the Storage block. If the controller is being used for output, meaning it receives the request byte signal, then the Controller will begin reading data from Storage, and then send this data on the out byte line back to the SPI slave, noting that the output is the saved passwords for the inputted hashes.

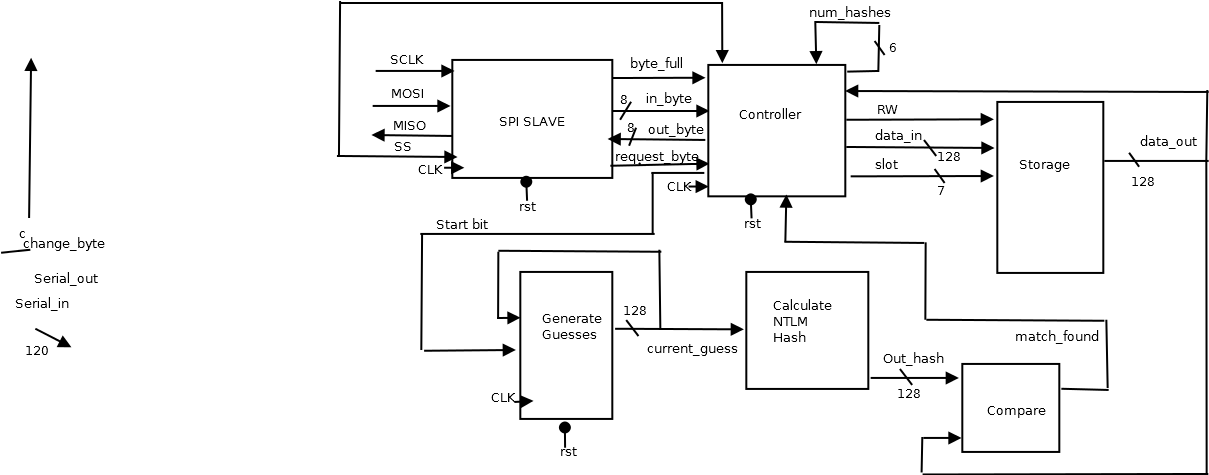
Design Flow Chart



**Requirements:**

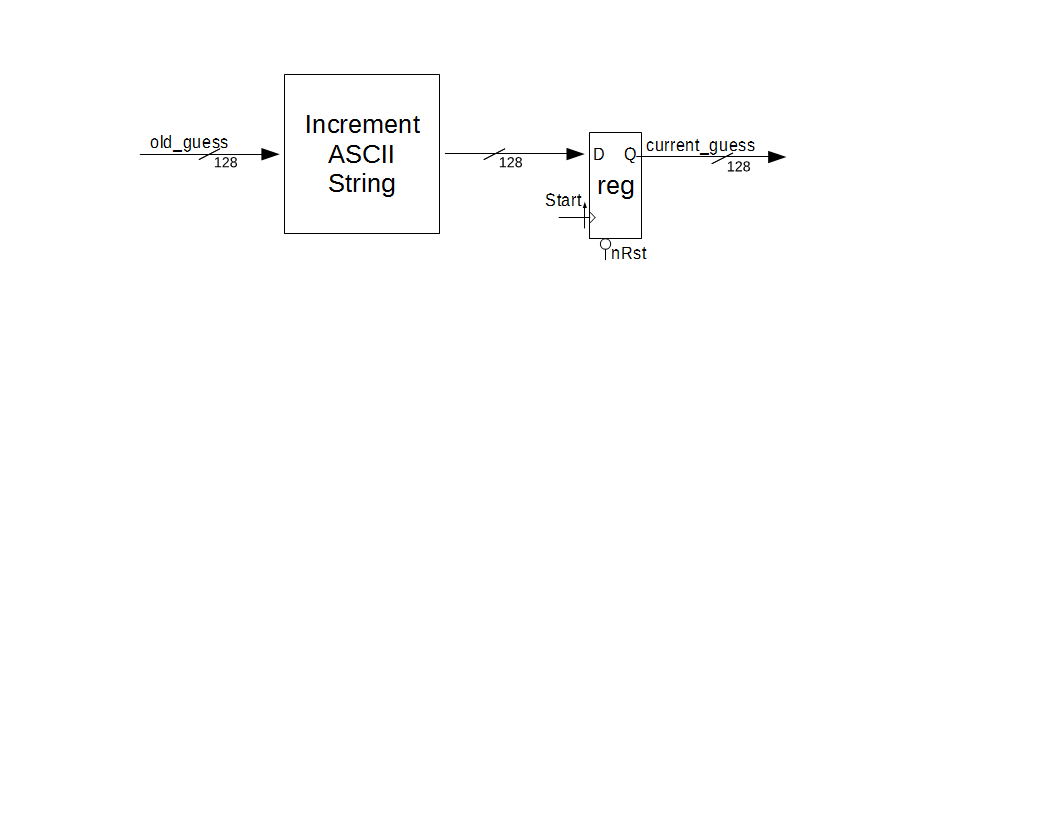
Our project design must meet specific requirements to be considered successful. Of those, the primary optimization focus for this project is speed. Ideally the design will be able to take a large number of hashes and quickly output the correct passwords that correspond to each inputted hash. Because of this, the design will sacrifice power efficiency for a larger number of calculations per second. Mobility isn't necessarily a concern for our project therefore size isn't an optimization constraint, however, we obviously must constrict size due to manufacturing costs and general usability. Throughput will also not be an immediate design constraint. Our project doesn't take high amounts of data constantly and instead most of the time spent will be generating potential passwords and calculating the corresponding hashes. Since the inputs and outputs of the system are both either binary values or ASCII characters, any IO system that gives even a minimal throughput will be sufficient. As of right now, we are looking to achieve the course standard size of 1.5mmx1.5mm in area and a speed of 200 MHz. We obviously would like to achieve as efficient a design as possible, however, at this stage we would like to achieve the standard requirements.   
 Because our project uses SPI, the only needed IO hardware is four Digital IO pins. This will allow for the vast majority of our design area to be devoted to computation. With more area, we can dissipate more heat and as a result perform more calculations per second.

**Design Architecture:**

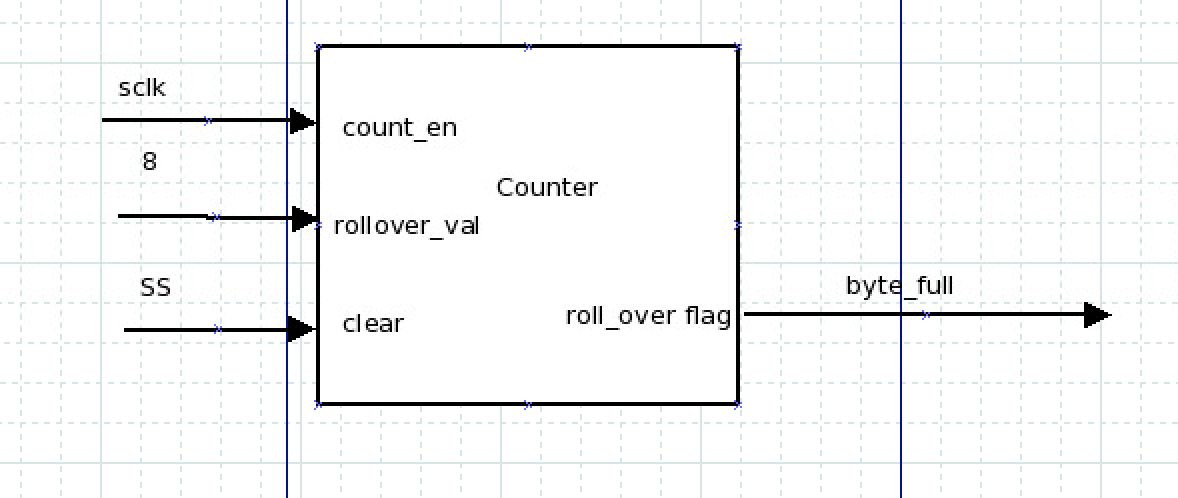
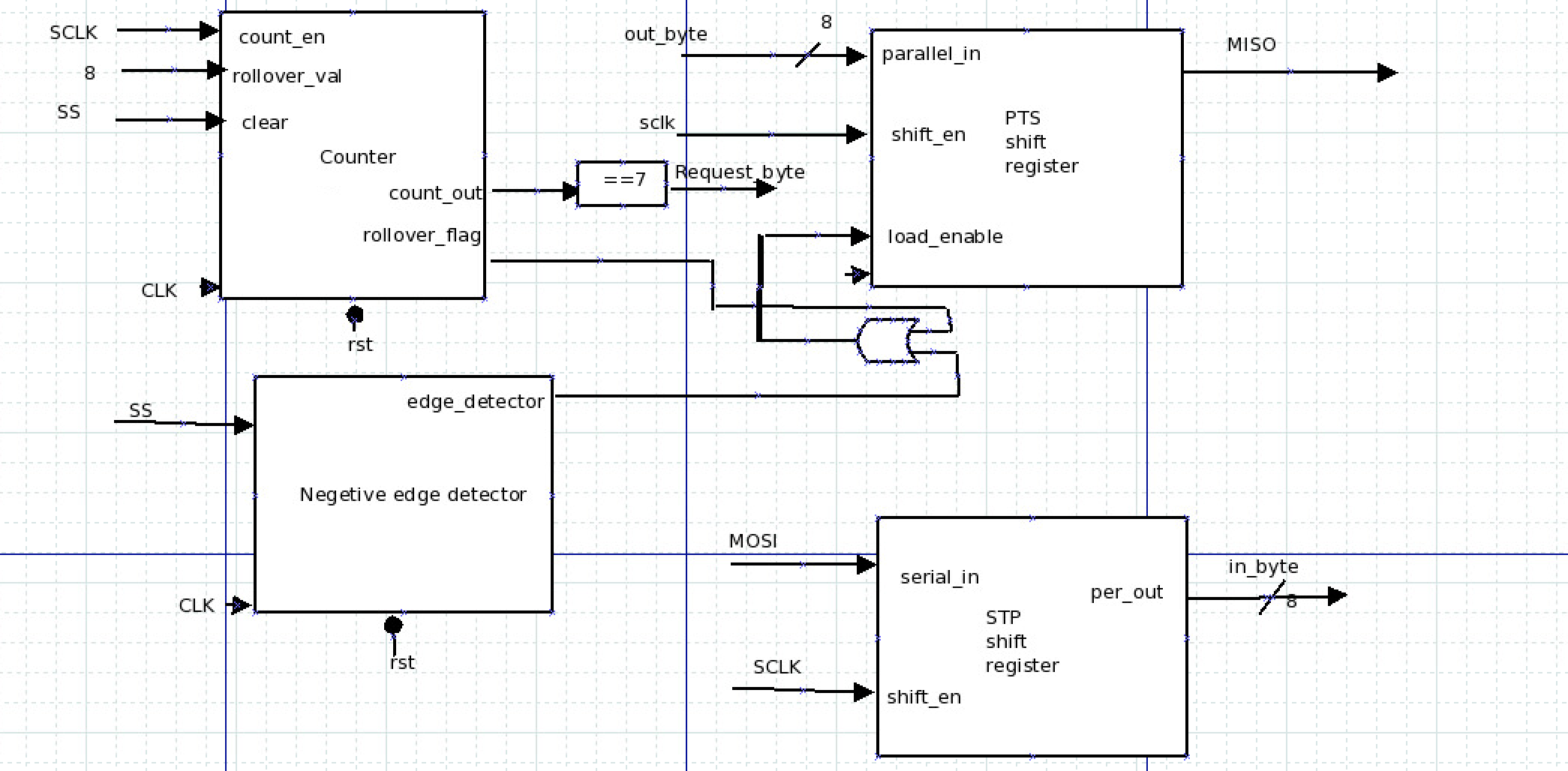


**Block Diagrams:**

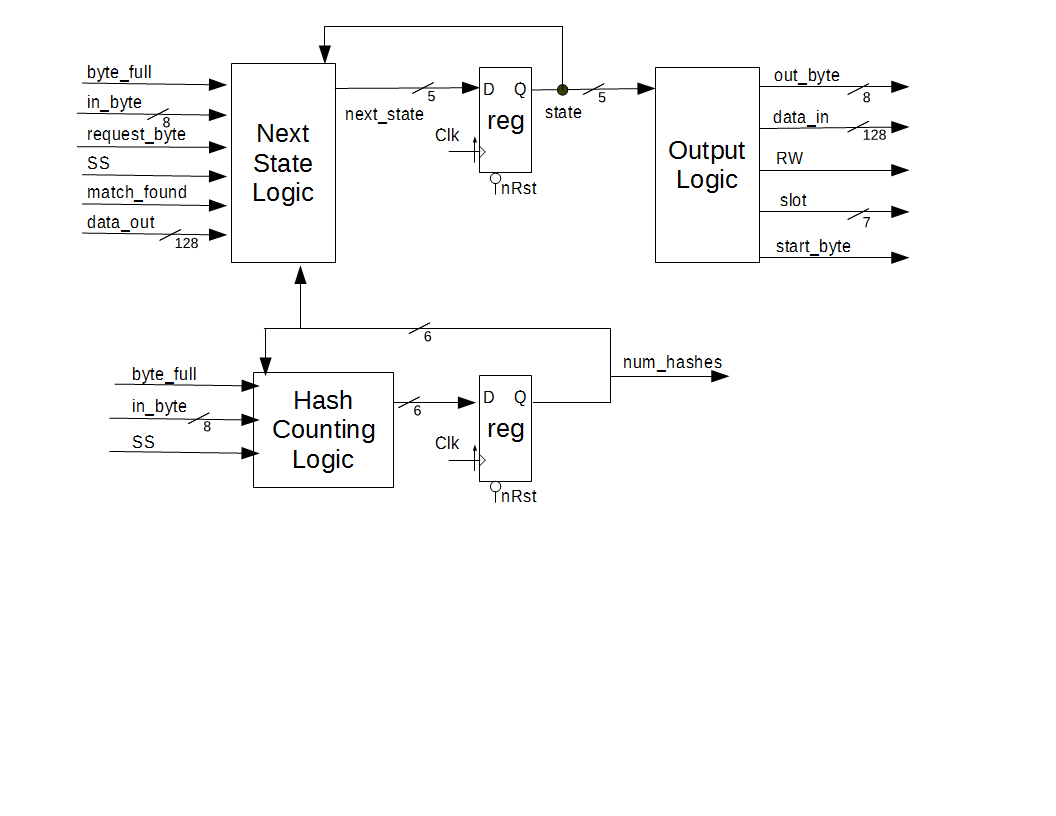
The following block so for the Generate Guesses block. This block will take the old guess from the next guess line, and increment the value to the next ASCII string value.



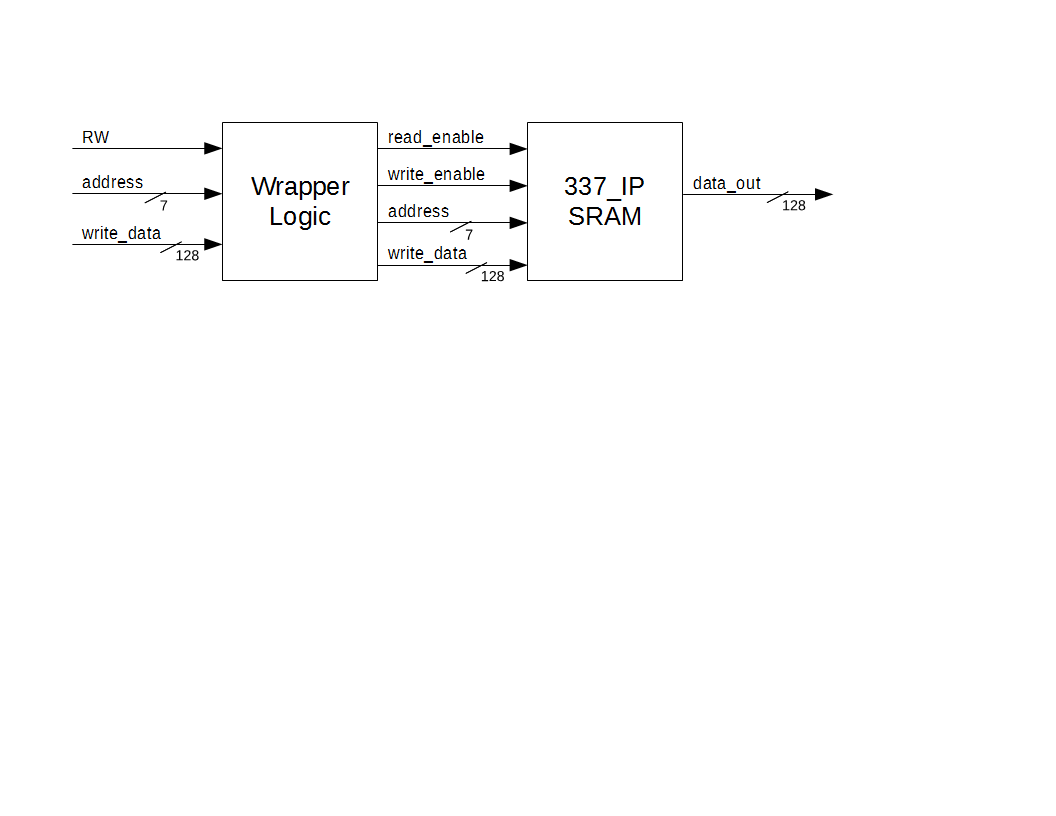
The following is the breakdown of the SPI Slave. The slave will function as a counter and an appropriate shift register that will be able to transmit data either to the controller or back to the Master.



The following is the breakdown of the Controller block. The Controller, while complex is just a machine along with an internal counter that initializes and retains the number of hashes being processed.



The following is a breakdown of the Storage block. While the block is purely combinational, it was included to highlight the usage of 337 IP.



\*\*The Calculate NTLM Hash and Compare blocks have not been included because these blocks are purely combinational logic.

**Projected Timeline and Division of Tasks**

|  |  |
| --- | --- |
| Week #: Week of | Description of Tasks |
| Week 11: 3/23/2015 | Receive feedback from project proposal and begin timing and area budgeting. – Enter team |
| Week 12: 3/30/2015 | Finish preliminary design budgets – Team  Work on beginning Hash Algorithm generation – Tom and Clayton  Begin working on a compare function – Darius and Adit |
| Week 13: 4/6/2015 | Present Design Reviews – Team  Continue Hash Algorithm generation – Tom and Clayton  Begin Testing of compare function – Adit  Begin writing the generate guess function – Darius |
| Week 14: 4/13/2015 | Verification Plan due – Team  Begin testing of Hash Algorithm with compare– Clayton  Begin writing SPI Slave controller – Tom  Begin writing Storage controller – Adit and Darius |
| Week 15: 4/20/2015 | Begin testing SPI Slave controller – Tom and Adit  Begin testing Storage controller with hash generation– Darius and Clayton  Begin implementing design together –Team |
| Week 16: 4/27/2015 | Continue merging components and testing design – Team  Prepare for Final Presentation and write Final report – Team |
| Week 17: 5/4/2015 | If needed finish report and make presentation.  Project wrap-up |

**Success Criteria**

Fixed criteria:

1. Test benches exist for all top level components and the entire design. The test benches for the entire design can be demonstrated or documented to cover all of the functional requirements given in the design specific success criteria.
2. Entire design synthesizes completely, without any inferred latches, timing arcs, and, sensitivity list warnings.
3. Source and mapped version of the complete design behave the same for all test cases. The mapped version simulates without timing errors except at time zero.
4. A complete IC layout is produced that passes all geometry and connectivity checks.
5. The entire design complies with targets for area, pin count, throughput (if applicable), and clock rate. The final targets for these parameters will be determined by course staff based on your design review. Failure to reach any of the targets will result a score of 1 out of 2 provided that you are within 50% on area, 10% on pin count, and 25% on throughput. Doing worse in any category will result in a score of 0 out of 2.

Design specific criteria:

1. Demonstrate by simulation of Verilog test benches that the complete design is able to successfully match a generated password to that of an inputted hash. (2 pts)
2. Demonstrate by simulation of Verilog test benches that the complete design is able to store multiple hashes inputted on an SPI line. (2 pt)
3. Demonstrate by simulation of Verilog test benches that the complete design is able to output a matched hash and password to the SPI. (2 pt)
4. Demonstrate by simulation of Verilog test benches that the complete design is able to ignore starting signals followed by an invalid hash. (1 pt)
5. Demonstrate by simulation of Verilog test benches that the complete design is able to generate a valid NTLM hash. (1 pt)