ARC4 DECRYPTION CIRCUIT

ARC4 Decryption Circuit

ARC4 is a symmetric stream cipher once widely used in encrypting web traffic, wireless data, and so on; it has since been broken. Still, the structure of ARC4 is similar to modern symmetric encryption methods.

A stream cipher like ARC4 uses the provided encryption key to generate a pseudo-random byte stream that is XOR'd with the plaintext to obtain the ciphertext. Because XOR is symmetric, encryption and decryption are the same.

The basic ARC4 algorithm uses the following parameters:

Parameter	*Type*	*Semantics*
key[]	Input	Array of bytes that represent the secret key (24 bits in our implementation)
Ciphertext[]	Input	Array of bytes that represent the encrypted message
Plaintext[]	Output	Array of bytes that represent the decrypted result (same length as cipher text)

The following is the ARC4 algorithm pseudocode:

```
-- key-scheduling algorithm: initialize the s array
for i = 0 to 255:
    s[i] = i
j = 0
for i = 0 to 255:
    j = (j + s[i] + key[i mod keylength]) mod 256  -- for us, keylength is 3
    swap values of s[i] and s[j]

-- pseudo-random generation algorithm: generate byte stream (“pad”) to be xor'd with
the ciphertext
i = 0, j = 0
for k = 0 to message_length-1:
    i = (i+1) mod 256
    j = (j+s[i]) mod 256
    swap values of s[i] and s[j]
    pad[k] = s[(s[i]+s[j]) mod 256]

-- ciphertext xor pad --> plaintext
for k = 0 to message_length-1:
    plaintext[k] = pad[k] xor ciphertext[k]  -- xor each byte
```

The length of the secret key can vary but this project uses a smaller key of 24 bits stored in big-endian.

Messages in this lab (both plaintext and encrypted) are length-prefixed strings of any length from 0 to 255 characters. Strings are encoded as an array of bytes where the first bye indicates the length of the string and the remaining bytes are the ASCII values of the characters; thus a string with *n* characters is represented by *n + 1* bytes.

Embedded memories are also used in this project using a generated RAM with Megafunction Wizard on Intel Quartus Prime. The following described the embedded generated RAM built:

- Output bus ~ 8 bits wide
- Size of memory ~ 256 words
- Memory clock type ~ M10K
- Clocking method ~ single clock

The following SystemVerilog shows the generated module:

```
module s_mem (
        address,
        clock,
        data,
        wren,
        q);

        input [7:0] address;
        input clock;
        input [7:0] data;
        input wren;
        output [7:0] q;
```

Initializing ARC4 (***init.sv***)

To begin, we need to initialize the system, the pseudo-code shows what we are implementing:

```
for i = 0 to 255:
    s[i] = i
```

These parameters describe the ***init.sv*** module:

Parameter	*Type*	*Semantics*
clk	Input	System clock
rst_n	Input	Asynchronous reset
en	Input	Enable signal, following ready-enable protocol.
rdy	Output	Ready signal, following ready-enable protocol.
[7:0] addr	Output	Address output from module to RAM.
[7:0] wrdata	Output	Write data output from module to RAM.
wren	Output	Write enable signal from module to RAM.

The module has these following design ideas:

Block	*Functionality*	*Details*
Counter Block	Triggered at the event of an asynchronous reset or positive edge of the clock.	If reset triggers, counter is reset to zero. If counter control signal is on, at every clock cycle that counter will increment by one. If counter done signal is on, counter control signal is off, and the counter block is stopped
State Machine Flow D-FlipFlop	D-flipflop controlling the flow of the state machine upon the trigger of asynchronous reset or positive edge of the clock.	Upon every clock cycle, the current state is moved to the next state. Upon an asynchronous reset, the current state is reset.
Next State Case Block	Determines next state by checking current state and other signals determining the path.	3 defined states (RESET, LOOP, END) to indicate three primary behaviours. Follows ready-enable protocol to begin (READY). Iterates the loop until notified done by a finished signal (LOOP).

		Idles in final state once iteration is finished (END).
Output Case Block	Supplies module outputs defined by the current state as well as controls other signals governing the flow of the circuit.	Properly supplies the rdy, [7:0] addr, [7:0] wrdata, and wren signals for each state. Controls finished signal.

Module and test benches: https://github.com/ryaanluke/ARC4-Decryption-Circuit/tree/main/task1

The Key-Scheduling Algorithm (***ksa.sv***)

The objected of the KSA is to spready the key entropy evenly to precent statistical correlations in the generated ciphertext that could be used to break the cipher. Below it the pseudo-code being implemented:

```
j = 0
for i = 0 to 255:
    j = (j + s[i] + key[i mod keylength]) mod 256   -- for us, keylength is 3
    swap values of s[i] and s[j]
```

These parameters describe the ***ksa.sv*** module:

Parameter	*Type*	*Semantics*
clk	Input	System clock
rst_n	Input	Asynchronous reset
en	Input	Enable signal, following ready-enable protocol.
[23:0] key	Input	Secret Key
[7:0] rrdata	Input	Read data from RAM to module.
rdy	Output	Ready signal, following ready-enable protocol.
[7:0] addr	Output	Address output from module to RAM.
[7:0] wrdata	Output	Write data output from module to RAM.
wren	Output	Write enable signal from module to RAM.

The module has these following design ideas:

Block	*Functionality*	*Details*
Counter Blocks	Triggered at the event of an asynchronous reset or positive edge of the clock.	If reset triggers, counter is reset to zero. If counter control signal is on, at every clock cycle that counter will increment by one. If counter done signal is on, counter control signal is off, and the counter block is stopped.
State Machine Flow D-FlipFlop	D-flipflop controlling the flow of the state machine upon the trigger of asynchronous reset or positive edge of the clock.	Upon every clock cycle, the current state is moved to the next state. Upon an asynchronous reset, the current state is reset.
Next State Case Block	Determines next state by checking current state and other signals determining the path.	18 defined states (RESET, RDY, RDY_DEASSERT, COUNT, READ_S_I_ADDRESS, READ_S_I,

		READ_S_I_WAIT, GET_S_I, CALC_J, CALC_J_WAIT, READ_S_J_ADDRESS, READ_S_J, READ_S_J_WAIT, GET_S_J, SWAP_I, SWAP_I_WAIT, SWAP_J, SWAP_J_WAIT, COUNTER_DONE) to indicate 18 primary behaviours. Follows ready-enable protocol to begin (RESET, RDY, RDY_DEASSERT). Controls loop conditions through counter (COUNT). Read data from RAM with additional idle states considering RAM requires extra clock cycles until our module receives the data (READ_S_I_ADDRESS, READ_S_I, READ_S_I_WAIT, GET_S_I). Calculate index J as part of KSA (CALC_J, CALC_J_WAIT). Read data from RAM with additional idle states considering RAM requires extra clock cycles until our module receives the data (READ_S_J_ADDRESS, READ_S_J, READ_S_J_WAIT, GET_S_J). Swap values and store back into RAM with additional idle states considering RAM requires extra clock cycles to store our data (SWAP_I, SWAP_I_WAIT, SWAP_J, SWAP_J_WAIT) Finished (COUNTER_DONE)
Output if/else Block	Supplies module outputs defined by the current state as well as controls other signals governing the flow of the circuit.	Properly supplies the rdy, [7:0] addr, [7:0] wrdata, and wren signals for each state. Controls counter and finished signal.

Module and test benches: https://github.com/ryaanluke/ARC4-Decryption-Circuit/tree/main/task2

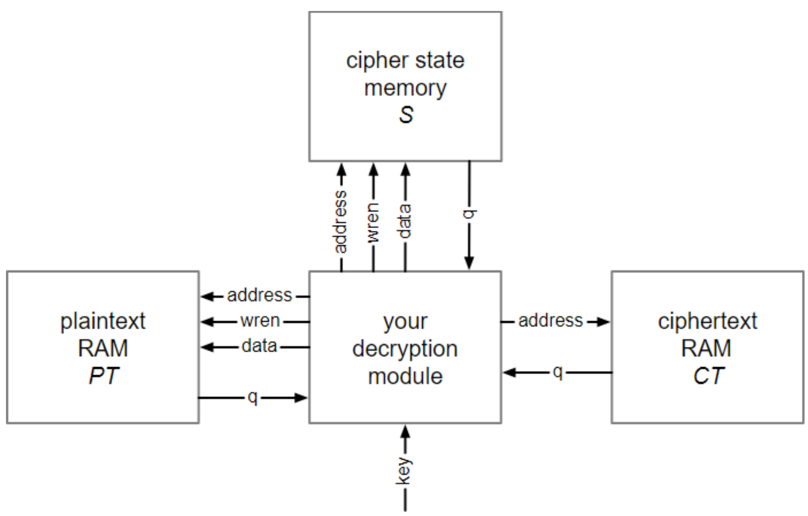
The Pseudo-Random Generation Algorithm (***prga.sv, arc4.sv***)

The final phase of ARC4 generates the bytestream that is then XOR'd with the input plaintext to encrypt the message, or as in our case, the input ciphertext to decrypt it. The following shoes the pseudo-code we are implementing:

```
i = 0, j = 0
for k = 0 to message_length-1:
    i = (i+1) mod 256
    j = (j+s[i]) mod 256
    swap values of s[i] and s[j]
    pad[k] = s[(s[i]+s[j]) mod 256]

for k = 0 to message_length-1:
    plaintext[k] = pad[k] xor ciphertext[k]  -- xor each byte
```

This contains two additional memories: one to hold the ciphertext, another to write the plaintext, and both being identical generated RAM. The following diagram shows how everything would be connected:



These parameters describe the ***prga.sv*** module:

Parameter	*Type*	*Semantics*
clk	Input	System clock
rst_n	Input	Asynchronous reset
en	Input	Enable signal, following ready-enable protocol.
[23:0] key	Input	Secret Key.
[7:0] s_rddata	Input	Read data from cipher state memory RAM.
[7:0] ct_rddata	Input	Read data from cipher text RAM.
[7:0] pt_rddata	Input	Read data from plaintext RAM.
rdy	Output	Ready signal, following ready-enable protocol.

[7:0] s_addr	Output	Address output from module to cipher state memory RAM.
[7:0] s_wrdata	Output	Write data output from module to cipher state memory RAM.
s_wren	Output	Write enable signal from module to cipher state memory RAM.
[7:0] ct_addr	Output	Address output from module to ciphertext RAM.
[7:0] pt_addr	Output	Address output from module to plaintext RAM.
[7:0] pt_wrdata	Output	Write data output from module to plaintext RAM.
pt_wren	Output	Write enable signal from module to plaintext RAM.

The module has these following design ideas:

Block	*Functionality*	*Details*
Counter Blocks	Triggered at the event of an asynchronous reset or positive edge of the clock.	If reset triggers, counter is reset to zero. If counter control signal is on, at every clock cycle that counter will increment by one. If counter done signal is on, counter control signal is off, and the counter block is stopped.
State Machine Flow D-FlipFlop	D-flipflop controlling the flow of the state machine upon the trigger of asynchronous reset or positive edge of the clock.	Upon every clock cycle, the current state is moved to the next state. Upon an asynchronous reset, the current state is reset.
Next State Case Block	Determines next state by checking current state and other signals determining the path.	33 defined states (RESET, RDY, RDY_DEASSERT, GET_MESSAGE_LENGTH_ADDRESS, READ_MESSAGE_LENGTH, READ_MESSAGE_LENGTH_WAIT, GET_MESSAGE_ADDRESS, WRITE_MESSAGE_LENGTH, GET_MESSSAGE_ADDRESS, WRITE_MESSAGE_LENGTH, START_LOOP, INCREMENT_K, INCREMENT_I, GET_I_ADDRESS, READ_I, READ_I_WAIT, GET_I, INCREMENT_J, GET_J_ADDRESS, READ_J, READ_J_WAIT, GET_J, SWAP_S_I, SWAP_S_J, CALC_PAD_INDEX,

		CALC_PAD_INDEX_WAIT, GET_PAD_ADDRESS, READ_PAD, READ_PAD_WAIT, GET_PAD, GET_CIPHER_ADDRESS, READ_CIPHER, READ_CIPHER_WAIT, GET_CIPHER, UPDATE_PLAINTEXT, DONE) to indicate 33 primary behaviours. Follows ready-enable protocol to begin (RESET, RDY, RDY_DEASSERT). Read data from cipher text memory RAM to get the length of the message with additional idle states considering RAM requires extra clock cycles until our module receives the data (GET_MESSAGE_LENGTH_ADDRESS, READ_MESSAGE_LENGTH, READ_MESSAGE_LENTH_WAIT, GET_MESSAGE_ADDRESS). Write message length data into plain text RAM (WRITE_MESSAGE_LENGTH)]. Start loop counter (START_LOOP). Increment K (INCREMENT_K) Increment I, read data from cipher text memory ram with additional idle states considering RAM requires extra clock cycles until our module receives the data (INCREMENT_I, GET_I_ADDRESS, READ_I, READ_I_WAIT, GET_I). Increment J, read data from cipher text memory ram with additional idle states considering RAM requires extra clock cycles until our module receives the data (INCREMENT_J, GET_J_ADDRESS, READ_J, READ_J_WAIT, GET_J). Swap I and J and store values into cipher text memory RAM (SWAP_S_I, SWAP_S_J). Calculate PAD Index and read PAD from cipher text memory RAM with additional idle states considering RAM requires extra clock cycles until our module received the

		data (CALC_PAD_INDEX, CALC_PAD_INDEX_WAIT, GET_PAD_ADDRESS, READ_PAD, READ_PAD_WAIT, GET_PAD). Read cipher text at the same index as PAD from ciphertext RAM (READ_CIPHER, READ_CIPHER_WAIT, GET_CIPHER). Update plaintext at the same index (UPDATE_PLAINTEXT) Finished (DONE)
Output if/else Block	Supplies module outputs defined by the current state as well as controls other signals governing the flow of the circuit.	Properly supplies the rdy, [7:0] addr, [7:0] wrdata, and wren signals for each state. Controls counter and finished signal.

Modules and test benches: https://github.com/ryaanluke/ARC4-Decryption-Circuit/tree/main/task3

Note: ***arc4.sv*** is the topmost level module that uses a state machine to iterate through the beginning to end of building the full ARC4 Decryption Circuit, going from ***init.sv* -> *ksa.sv* -> *prga.sv***.

Cracking ARC4 (***crack.sv***)

Now we can decrypt some encrypted messages without knowing the key ahead of time. For this project, an encrypted message is deemed to be cracked if its characters consist entirely of byte values between ‘h20 and ‘h70 inclusive.

The ***crack.sv*** module is very much like ***arc4.sv*** but instead our cipher text memory RAM and plain text RAM are now internal, and the key is now an output. It will iterate through every possible key, check if the key is valid and return the key if it is.

These parameters describe the ***crack.sv*** module:

Parameter	*Type*	*Semantics*
clk	Input	System clock
rst_n	Input	Asynchronous reset
en	Input	Enable signal, following ready-enable protocol.
[7:0 ct_rddata]	Input	Read data from cipher text RAM.
[23:0] key	Output	Secret Key
key_valid	Output	Valid or non valid key flag.
Rdy	Output	Ready signal, following ready-enable protocol.
[7:0] ct_addr	Output	Address output from module to ciphertext RAM.

The module has these following design ideas:

Block	*Functionality*	*Details*
Counter Blocks	Triggered at the event of an asynchronous reset or positive edge of the clock.	If reset triggers, counter is reset to zero. If counter control signal is on, at every clock cycle that counter will increment by one. If counter done signal is on, counter control signal is off, and the counter block is stopped.
State Machine Flow D-FlipFlop	D-flipflop controlling the flow of the state machine upon the trigger of asynchronous reset or positive edge of the clock.	Upon every clock cycle, the current state is moved to the next state. Upon an asynchronous reset, the current state is reset.
Next State Case Block	Determines next state by checking current state and other signals determining the path.	24 defined states (RESET_CRACK, RDY_CRACK, RDY_DEASSERT_CRACK, READ_MESSAGE_LENGTH_CT, READ_MESSAGE_LENGTH_CT_WAIT, WRITE_MESSAGE_LENGTH_PT, START_KEY_COUNTER,

		INCREMENT_KEY_COUNT, CHECK_KEY_COUNTER, SET_NEXT_KEY, SET_EN_A4, WAIT_A4, DONE_A4, START_KEY_CHECK, CHECK_KEY_LOOP, READ_PT, READ_PT_WAIT, CHECK_KEY_VALID, INCREMENT_MESSAGE_COUNT, KEY_NOT_VALID, KEY_VALID, SET_OUTPUT_KEY, CRACK_DONE, GET_PT_ADDRESS, GET_PT) to indicate 24 primary behaviours. Follows ready-enable protocol to begin (RESET_CRACK, RDY_CRACK, RDY_DEASSERT_CRACK). Read data from cipher text memory RAM to get the length of the message with additional idle states considering RAM requires extra clock cycles until our module receives the data (READ_MESSAGE_LENGTH_CT, READ_MESSAGE_LENGTH_CT_WAIT). Write message length data into plain text RAM (WRITE_MESSAGE_LENGTH_PT)]. Start key loop counter, increment the counter, check loop condition (START_KEY_COUNTER, INCREMENT_KEY_COUNTER, CHECK_KEY_COUNTER). Start ARC4 with current key, wait until done (SET_EN_A4, WAIT_A4, DONE_A4) Start key check, check key check loop condition. Read a character from plaintext RAM and check if each character in the message meets the character byte value condition. (START_KEY_CHECK, CHECK_KEY_LOOP, READ_PT, READ_PT_WAIT, GET_PT_ADDRESS, GET_PT, CHECK_KEY_VALID, INCREMENT_MESSAGE_COUNT). Determined if key is valid or not valid (KEY_NOT_VALID, KET_VALID).

		If key is valid, set output as key (SET_OUTPUT_KEY) Finished (CRACK_DONE)
Output if/else Block	Supplies module outputs defined by the current state as well as controls other signals governing the flow of the circuit.	Properly supplies the rdy, [7:0] addr, [7:0] wrdata, and wren signals for each state. Controls counter and finished signal.

Modules and testbenches: https://github.com/ryaanluke/ARC4-Decryption-Circuit/tree/main/task4

Summary

This project explored building an ARC4 decryption/encryption circuit using the following algorithm:

```
-- key-scheduling algorithm: initialize the s array
for i = 0 to 255:
    s[i] = i
j = 0
for i = 0 to 255:
    j = (j + s[i] + key[i mod keylength]) mod 256  -- for us, keylength is 3
    swap values of s[i] and s[j]

-- pseudo-random generation algorithm: generate byte stream (“pad”) to be xor'd with
the ciphertext
i = 0, j = 0
for k = 0 to message_length-1:
    i = (i+1) mod 256
    j = (j+s[i]) mod 256
    swap values of s[i] and s[j]
    pad[k] = s[(s[i]+s[j]) mod 256]

-- ciphertext xor pad --> plaintext
for k = 0 to message_length-1:
    plaintext[k] = pad[k] xor ciphertext[k]  -- xor each byte
```

Then automating the process of finding the key that cracks ARC4 by iterating through every possible key, running that key through ARC4, and seeing if it returns valid.

Programming / Lessons takeaways:

1. State machines on state machines on state machines on…
   - We built complex modules that used state machines, that other modules used that were also governed by a state machine, that other modules used that … you get the point.

2. Ready-Enable Protocol
   - Developed modules using ready-enable protocol adding complexity to how modules are called and used.

3. Embedded Memories
   - Learned how to create RAM using Megafunction Wizard
   - Write state machines that followed RAM block’s clock cycle timing conditions for reading and writing.
   - Transferring data from one RAM block to another and back and forth.
   - Test benching RAM blocks using commands from Altera libraries.

4. Turning pseudo-code algorithms to digital design
   - Exercising loops, memory access, swapping, reading/writing into arrays, etc.