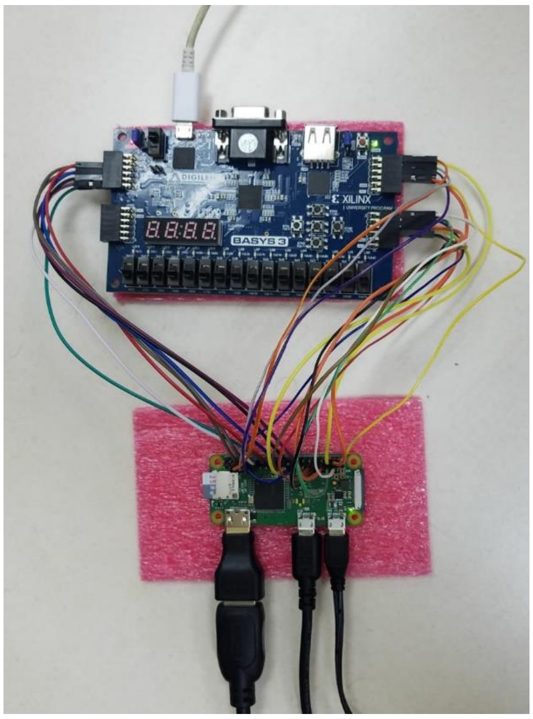
FPGA Design and Implementation of Cubehash16/32 - 256



By: Elchanan Schnaidman & Michael Eldin
Supervisor: Mr. Uri Stroh

Project Carried Out at Jerusalem College of Technology (JCT)

Team number: xohw19-303

Video is available at:
https://www.youtube.com/watch?v=nfsf__5VdeM

1. Introduction

CubeHash [1] is a cryptographic hash function submitted to the NIST hash function competition by Daniel J. Bernstein [2]. CubeHash has a 128-byte state, uses wide pipe construction, and is ARX based. Message blocks are XORed into the initial bits of a 128-byte state, which then goes through an r-round bijective transformation between blocks.

CubeHash has 5 parameters, a certain instance is denoted by CubeHashi+r/b+f-h

- i - is the number of initial rounds
- r - is the number of rounds per block
- b - is the block size in bytes, defined for {1, 2, 3, ...128}
- f - is the number of final rounds
- h- is the size of the hash output in bits, defined for {8, 16, 24, 32, ... 51}

In the original NIST submission, i and f was fixed to 10r. The obsolete notation CubeHashr/b-h indicates i and f being implicitly 10r

The strength of this function increases as b decreases towards 1, and as r increases.

However, there is a security versus time tradeoff. A more secure version will take longer to compute a hash value then a weakened version.

In this project, we have implemented CubeHash16/32-256.

As in many cases, one of the designers' main challenges is to carefully maintain balance between Timing considerations and Area considerations. This implementation demonstrates a fair combination of these two factors.

This implementation is combined of two boards, Basys 3 & Raspberry Pi Zero.
The Basys 3 development board by Digilent, which is designed exclusively for Vivado Design Suite, featuring Xilinx Artix-7 FPGA architecture and is perfectly suited for students. The Raspberry Pi Zero which is affordable board to display the user interface.

The Basys 3 gets message from the user through Raspberry Pi Zero, encrypting it and send it back to the Raspberry Pi Zero so the user can see the results on the screen.

As it reasonable to do, the Basys 3 in our design is used for implementing the algorithm, and the Raspberry Pi Zero is used for dealing with user input & output operations.

The project had several motives:

1. To get familiar with different aspects of FPGA design.

2. To get familiar with the unique Basys 3 devices architecture.

3. To develop a full practical system, which can be used for information security and digital signature.

4. To achieve communication between an FPGA board and an external software board.

5. To explore and experience some advantages of hardware design over software.

2. Design

Following is a brief description of the design. Additional details and explanation can be found in the corresponding source files.

Platform: The Basys 3 development board, Raspberry Pi Zero which connected with 22 jumper wires (as shown in the front-page picture).

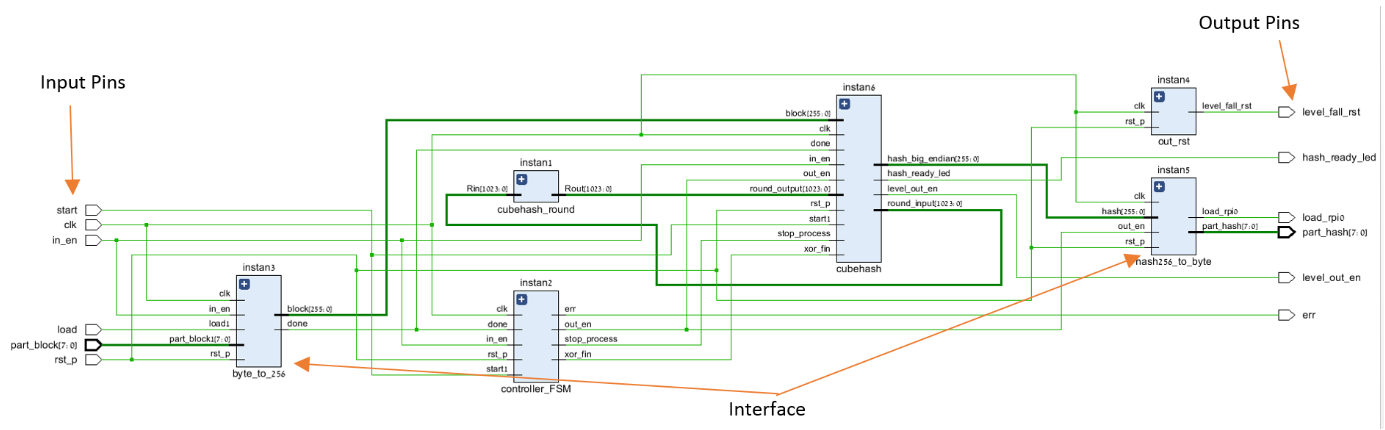


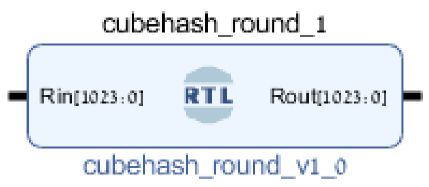
Figure 1. Scheme made by Vivado

2.1 Hardware

The source files were written in Verilog. Their names, structure and logic are reasonable and reveal their purpose.

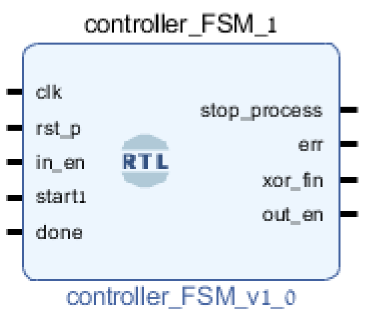
Our source files include:

1) **cubehash_round**:



One Cubehash round on the internal state.

2) **controller_FSM**:



The FSM has 4 states: idle, round, waiting and fin. We decided to design it with one-hot encode. The main reasons for our decision are because in one-hot it's easy to detect illegal states, it allows the FSM to run at a faster clock than any other encoding. We're not worried about the number of flip-flops it requires because of the Artix-7 abundant flip-flops [3].

In addition, when some scenarios that are illogical happen – the FSM raises an error signal that turns on a LED.

As follows, a brief description of each state:

idle: At this state all the internal registers of the design are after reset, waiting for an event to start encryption.

round: At this state the rounds of the Cubehash are made.

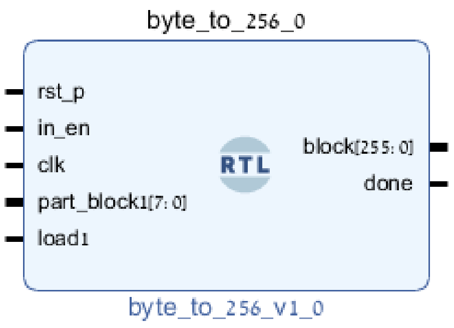
As we chose on our implementation r = 16, the FSM stays on this state for 16 clock cycles because we implemented the Cubehash on iterative implementation.

Waiting: We had to deal with the external software slowness. The 16 rounds last 160ns and the design on the Artix-7 expect to receive another block or a signal that inform that it was the last block. But on 160ns the Raspberry Pi Zero barely passes a line on the code, so on this state a clock enable signal is raised and it freeze the registers that hold the output of the cubehash_round and it won't continue to the input of cubehash_round till it pass to other state.

fin: on this state the Cubehash enters to finalization process.

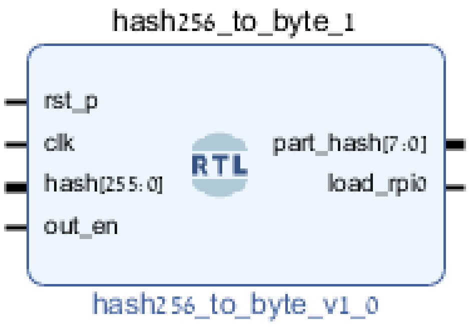
The finalization process requires a XOR and afterwards - 160 rounds. Also, this state rises a signal that indicates that the hashed message is ready to be transmitted – when it’s ready.

3) **byte_to_256**:



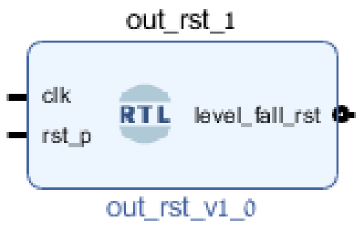
Logic that gets the 256 bits block in 32 packages of 8 bit each, and concatenates it to a 256 bits array (32b as written on the spec).

4) **hash256_to_byte**:



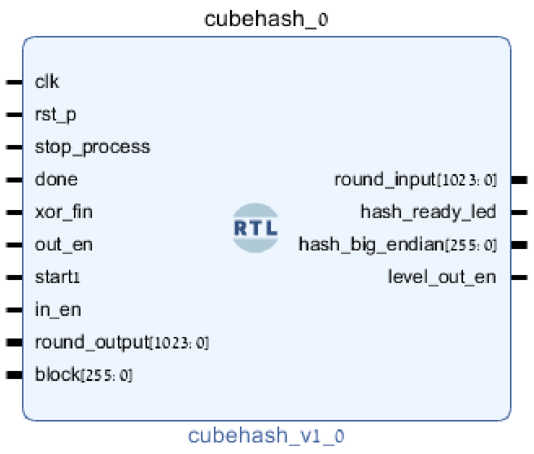
Logic that gets 256 bits of Cubehash result (hashed message) and transmit it in 32 packages of 8 bit each and deals also with slowness of the Raspberry Pi Zero.

5) **out_rst**:



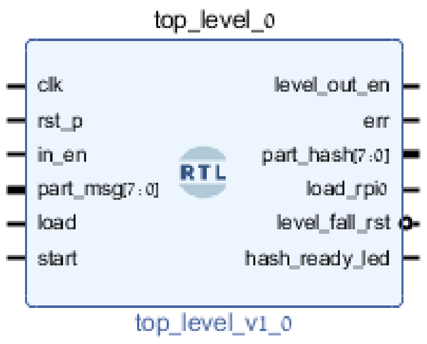
Creates a stable signal which synchronizes between the resets of Basys 3 & Raspberry Pi Zero.

6) **cubehash**:



This module generates the rounds logic and puts flip-flops between the output of the round logic and its input (it puts also a little bit of logic gates and MUXs). In addition, at the end of encryption, it changes the hash to a little-endian format (as required on the spec).

6) **top_level**:



Makes all the instantiations between the modules.

Buttons & LEDs:

The RESET button was placed in the center button of the Basys 3.

Two LEDs were designed to help the user/developer to know about scenarios happened with the Basys 3.

Firstly, there is the LED that indicates that the hash is ready and the whole hash process have been passed successfully [LD8].

Secondly, there is the LED that indicates that some error occurred during hash process and the hash process stopped in the middle without finishing [LD0].

2.2 Software

The software uses python code which gets a message (plain text) from the user, padding it and transmit it to the Basys 3 board in packages of eight bit (Because the limitation of I/O ports in Basys 3 and Raspberry Pi Zero). At the end, it receives the encrypted message (cipher text) and displays the hashed message on the screen.

Our script includes some functions:

Setup: setups all the GPIOs in the Raspberry Pi Zero.

Pad: pads the message.

Transmit: Transmit the message (plain text) into Basys 3 in packages of 8 bit.

Receive: receives the encrypted message (cipher text) and concatenate it.

2.3 Design Reuse

The software program is in little - endian. It can be executed on any 32-bit or 64-bit processor with a Python compiler. The GPIO designed for Raspberry Pi, but except of this function, the functions were written in a way that they can be reused easily on other processor.

The hardware program is in little – endian. It doesn't depend on any special feature of the Basys 3 Artix-7 FPGA architecture, so it can be implemented in any FPGA device, given sufficient area and sufficient number of I/O's.

The connections between the Hardware (Basys 3) and Software (RPi Zero) are specified on the table below:

Note: The RPi Zero GPIO detailed below are BOARD numbers (**not** BCM)

Basys 3	**Raspberry Pi Zero GPIO**	**Explanation**
pmod JB1 (input)	5 (output)	In enable signal – there are more blocks to send
pmod JB2 (input)	7 (output)	Start signal – notify Basys 3 that the first block was sent
pmod JB3 (input)	11 (output)	Load block to Basys 3 signal – pulse when sending 8 bits (part of block)
pmod JB4 (output)	38 (input)	Big & stable pulse when the reset button was pressed
pmod JB7 (output)	36 (input)	Load to RPi Zero – pulse when sending 8 bits of hashed message
pmod JB10 (output)	40 (input)	High when Basys 3 finished encryption process
pmods [JA1, JA2, JA3, JA4, JA7, JA8, JA9, JA10] (output)	[26, 24, 22, 18, 16, 12,10, 8] (input), respectively	Part (8 bits) of hashed message.
pmods [JC1, JC2, JC3, JC4, JC7, JC8, JC9, JC10] (input)	[37, 35, 33, 31, 29, 23, 21, 19] (output), respectively	Part (8 bits) of message to encrypt (hash)

3. Results

The main challenge while implementing the design was building accurate and robust modules, which implement the algorithm without wasting area, and without wasting any clock cycles within the main algorithm module Cubehash or while transferring signals between Basys 3 and RPi0.

The all algorithm for an empty message in utopian world takes only 178 clock – cycles.

(1clock to load the block + 16 clocks for rounds + 160 clocks for finalization + 1 clock to hash pass the registers of output = 178)

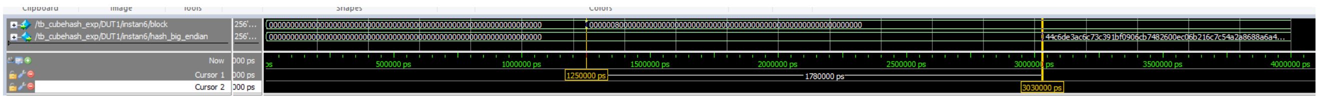
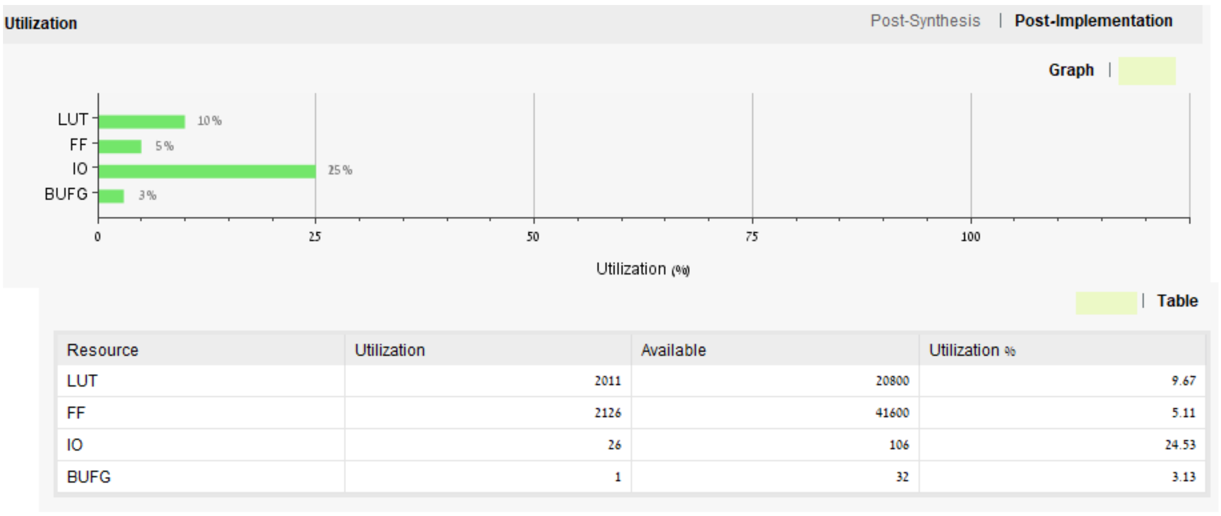


Figure 2. Verilog simulation (Mentor Graphics' ModelSim)

In figure 2 it's shown that the whole hash took 1780ns, since we use 10ns clock so the Cubehash took 178 clock cycles as we expected.

We used Vivado 2018.3 to implement this project on an Artix-7 FPGA device with sufficient number of I/O's (XC7A35TCPG236-1)

Some of the results we got:



Resource	Utilization	Available	Utilization %
LUT	2011	20800	9.67
FF	2126	41600	5.11
IO	26	106	24.53
BUFG	1	32	3.13

Figure 3. Utilization made by Vivado

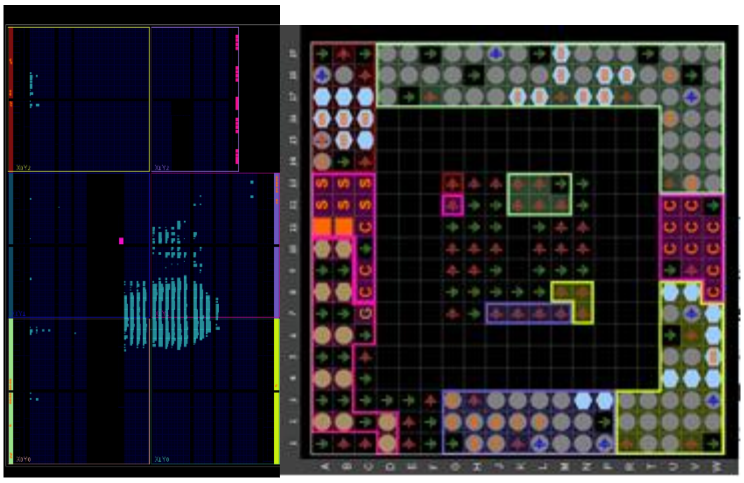


Figure 4. P&R and I/O planning made by Vivado

In figure 3 we can see that this implementation uses a small percentage and small amount of FFs and LUTs. In figure 4 we can see the way Vivado arranged the optimization design and the I/O that our chosen artix-7 has the I/O that we use. According to timing report provided by Vivado, the minimum slack between two flip-flops is 4.5 ns. We used internal Basys 3 clock (100 MHz). It means that our maximum frequency for the design is 181 MHz.

As shown and explained, our project uses low area with large frequency clock.

4. Conclusion

Every aspect of this project required a very steep learning curve on our part. As one of our goals was to enter the world of FPGA development without any prior knowledge or experience, some extensive self-education was in order. Luckily, in our day and age, the internet provides more than enough resources on every possible topic, and Xilinx itself has an abundance of user manuals, workshops, and tutorials (in a sometimes-overwhelming quantity), to guide us. Before diving into any relevant work, we had to first learn VERILOG, understand the work relations between the Basys 3 and Raspberry Pi Zero, the workflow and structure of Vivado, and how to use the tools they provide. We learned about creating our own IPs, and gradually became more comfortable with the tools at our disposal.

On the theoretical side, we learned about Cryptography, and found a vast new world with a large community, and countless academic papers on every imaginable topic – from various algorithms, to implementation methods (which are in a constant state of development and improvement) and more.
Due to time constraints (in the real-world, not on the FPGA), the steep learning curve and a lot of technical difficulties on the way, we weren't able to venture very far into this interesting world, but we hope to continue the development under more relaxed circumstances in the future.
Overall, we have succeeded to implement the algorithm in a simple elegant way, and create a relatively-low-area design that can fit into many FPGA devices, and enable use of digital signature even in limited resources environments. That was done while "keeping an eye" on the design timing and achieving a fair clock speed.
In this project, we created a platform which can serve as a starting point for future development.
The project can be endlessly expanded – from encrypting passwords, to improving the existing algorithms and creating new ones (e.g. implementing the software in C and/or use a faster micro-processor which will cause the program run much faster).

5. References

[1] Cubehash:

https://en.wikipedia.org/wiki/CubeHash

[2] Spec:

https://cubehash.cr.yp.to/submission/spec.pdf

[3] One hot:

https://en.wikipedia.org/wiki/One-ho

[4] Project on GitHub:

https://github.com/michaeldin/cubehash

Appendix

Full source code for user generated hardware, software and constraints

1. Verilog files

2. Python file

3. Constraints file (XDC)

Verilog files:

cubehash.v:

```
/*
this module generates inputs & outputs to the block that does one cunehash
round.

in several times on the cubehash some XORs and freezing (freezing is
because the communication that we chose to
implement) should be done, and this module takes care of the logic that
does it.

 */

module cubehash(
input clk,                          // internal 100MHz clock
input rst_p,                        // active high synchronous reset
input stop_process,                 // this signal freeze all the design
input done,                         // inform that a new block arrived to
the board
input xor_fin,                      // inform to XOR the internal state
(internal state is the 1024 bits that the cubehash is made of them)
input out_en,                       // inform that the cubehash finished
and should be transmitted outside the board
input start1,                       // when high it means that now was the
first block
input in_en,                        // High when there are blocks to send
for encryption
input [1023:0] round_output,        // the result of one round
input [255:0] block,                    // the block to be processed
output reg [1023:0] round_input,    // the 1024 bits that should do a round
output hash_ready_led,              // a led turns on when the hash is
ready
output [255:0] hash_big_endian,     // as specified on spec, we work on
little endian but we want to display in big endian
output reg level_out_en             // a stable constant signal that inform
the other board that cubehash finished
);

// this initial vector is a function of h,b and r - and was calculated in
advance
parameter [1023:0] iv =
1024'hea2bd4b4_ccd6f29f_63117e71_35481eae_22512d5b_e5d94e63_7e624131_f4cc12
be_c2d0b696_42af2070_d0720c35_3361da8c_28cceca4_8ef8ad83_4680ac00_40e5fbab_
d89041c3_6107fbd5_6c859d41_f0b26679_09392549_5fa25603_65c892fd_93cb6285_2af
2b5ae_9e4b4e60_774abfdd_85254725_15815aeb_4ab6aad6_9cdaf8af_d6032c0a;

reg [255:0] hash_little_endian;

// the flip-flops between output & input of module of one round
reg [1023:0] data_out;

// current internal state
reg [1023:0] current_state;

// signal that inform if to take the iv (at the beginning of cubehash) or
the result of round (not in the very beginning of cubehash)
reg iv_rou;
```

```verilog
wire [1023:0] state_xored_block;
reg [1023:0] mid_state;

/////////////////////////////////////////////////////////////

//logic that even when the input is not connected, the design won't receive
"Z".
reg start;

always @(posedge clk) begin
    if ( rst_p == 1'b1)
        start <= 0;
    else if(in_en ==1'b1)
        start <= start1;

end

/////////////////////////////////////////////////////////////

// logic that decides if now it's the very beginning of cubehash or not

always @(*)
begin
    if(stop_process == 1'b0 && start == 1'b1 && done == 1'b1 )
        iv_rou = 1'b1;
    else
        iv_rou = 1'b0;

end

////////////////////////////////////////////////////////////

// XORs the new block with the internal state
assign state_xored_block = {mid_state[1023-:256]^block, mid_state[1023-
256:0]};

// MUXs that decides which 1024 bits to take
always @(*) begin

    if(iv_rou == 1'b1)
         mid_state = iv;
    else
         mid_state = round_output;
end


always @(*) begin
    if(done == 1'b1)
        current_state = state_xored_block;
    else
        current_state = mid_state;
end


// logic that enables the flip-flops just when is needed, otherwise the
states freeze
always @ (posedge clk) begin
    if(rst_p == 1'b1)
        data_out <= {1024{1'b0}};
    else if(stop_process == 1'b0)
```

```
        data_out <= current_state;
end

// logic that XORs the last word with 1 - when needed
always @(*) begin
        if(xor_fin == 1'b1)
            round_input = {data_out[1023:1] , data_out[0]^1'b1};
        else
            round_input = data_out;
end


////////////////////////////////////////////////////////////////////

// flip-flops between this module to the module that transmit the hash,
they are loaded just when cubehash finished successfully
always @(posedge clk) begin
    if (rst_p == 1'b1)
        hash_little_endian <= {255 {1'b0}};
    else if( out_en == 1'b1)
        hash_little_endian <= data_out[1023-:256];

end


// to display the result, we want to change to big endian format
genvar j;

for(j=0; j<256; j = j+32) begin

     assign hash_big_endian[j+:4]       = hash_little_endian[(j+24)+:4];
     assign hash_big_endian[(j+4)+:4]   = hash_little_endian[(j+28)+:4];
     assign hash_big_endian[(j+8)+:4]   = hash_little_endian[(j+16)+:4];
     assign hash_big_endian[(j+12)+:4]  = hash_little_endian[(j+20)+:4];
     assign hash_big_endian[(j+16)+:4]  = hash_little_endian[(j+8)+:4];
     assign hash_big_endian[(j+20)+:4]  = hash_little_endian[(j+12)+:4];
     assign hash_big_endian[(j+24)+:4]  = hash_little_endian[j+:4];
     assign hash_big_endian[(j+28)+:4]  = hash_little_endian[(j+4)+:4];

end

//////////////////////////////////////////////////////////////////////////////////////////

//logic that generates HIGH on pmods when the process has finished.
always @(posedge clk) begin
    if (rst_p == 1'b1)
        level_out_en <= 1'b0;
    else
        level_out_en <= out_en | level_out_en;
end

// another signal to drive it to leds also.
assign hash_ready_led = level_out_en;

endmodule
```

controller_FSM.v:

```
/*
This module is probably the heart of the design.
At this module are specified the signals that run the cubehash.

this module contains the state machine with all the possible states in our
design.



Explanation about inputs:

clk     - internal 100MHz clock
rst_p   - active high synchronous reset
in_en   - High when there are blocks to send for encryption
start1  - pulse when the first block had been transmitted
done    - pulse when a block arrived to the board


Explanation about outputs:

stop_process    - high when the design waits to signal to determine what it
needs to do. this signal freeze all the design
err             - high when an illogical scenario occurs
xor_fin         - pulse when the design enters to finalization process. the
design needs to do a XOR before starting finalization
out_en          - pulse when the hash is ready



Explanation about the states:

idle    - at this states the design is waiting for a trigger to start
cubehash

round   - at this states the FSM generates 16 rounds

waiting - at this state the design waits for a trigger to decide what to do
next. at this state the whole logic freezes.

fin     - at this state the design enter to finalization process (XOR and
160 rounds).

 */

module controller_FSM (
    input clk, rst_p, in_en, start1, done,
    output stop_process, err, xor_fin,
    output reg  out_en
);



parameter SIZE = 4;
parameter idle  = 4'b0001, round = 4'b0010, waiting = 4'b0100, fin =4'b1000
; // one-hot encode
reg [SIZE-1:0] state, next_state;


reg [3:0] cnt_16;   // counts 16 rounds
reg [3:0] cnt_10;   // counts 10 of 16 rounds for finalization (actually
counts 160 rounds)
```

```
/////////////////////////////////////////////

//logic that even when the input is not connected, the design won't receive
"Z".
reg start;

always @(posedge clk) begin
    if ( rst_p == 1'b1)
        start <= 0;
    else if(in_en ==1'b1)
        start <= start1;

end

////////////////////////////////////////////////


always @ (*)
begin : FSM_COMBO
next_state = 4'b0000;



case(state)
    idle :  begin
                if (start == 1'b1 && in_en == 1'b1 && done == 1'b1) // when
it occurs it triggers to start encrypting
                    next_state = round;
                else
                    next_state = idle;
            end

///////////////////////////////////////////////////////////////////////////
////

    round : begin
                if (cnt_16 == 15)           //  finished 16 rounds
                begin
                    if (in_en == 1'b0)      // finished 16 rounds and all
the blocks have successfully received
                        next_state = fin;
                    else
                        next_state = waiting;// finished 16 rounds and
there are more blocks to receive

                end
                else                        // in the middle of 16 rounds

                    if( done == 1'b0)       // in the middle of 16 rounds
of current block and there is no new block (this is good)

                        next_state = round; // continue with the rounds

                    else                    // in the middle of 16 rounds
for current state an a new block arrived (illogical scenario)

                        next_state = idle;  // stop cubehash and return to
idle
```

```
            end

////////////////////////////////////////////////////////////////////////////
////

    waiting :   begin
                    if (done == 1'b1 && in_en == 1'b1)      // new block
arrived
                        next_state = round;
                    else if(in_en == 1'b1 && done == 1'b0)  // should
arrive new block but it didn't arrive yet.
                        next_state = waiting;
                    else if(in_en == 1'b0 & done == 1'b0)   // finished
receiving the message to encrypt
                        next_state = fin;
                    else                                    // shouldn't
arrive block but somehow arrives a new block (illogical scenario).
                        next_state = idle;                  // stop
cubehash and return to idle

                end

////////////////////////////////////////////////////////////////////////////
////

    fin:    begin
                if (cnt_16 == 15)               // finished 16 rounds
                begin
                    if (cnt_10 == 9)            // finished 10 times 16
rounds (160 rounds)
                        next_state = idle;      // end of successful
cubehash encryption

                    else begin                  // finished a 16 rounds
cycle but didn't achieve yet 160 rounds

                        if (done  == 1'b0)      // no new block arrives
(this is good)
                            next_state = fin;   // continue with
finalization

                        else                    // at finalization and a
new block arrives (illogical scenario).
                            next_state = idle;
                    end
                end
                else begin                      // in the middle of 16
cycle of 16 rounds

                    if (done == 1'b0)           // no new block arrives
                        next_state = fin;       // continue with
finalization

                    else                        // at finalization and a
new block arrives (illogical scenario).
                        next_state = idle;      // stop cubehash and return
to idle
                end
            end
```

```
///////////////////////////////////////////////////////////////////////////
////

   default : next_state = idle;
  endcase
end


//----------Seq Logic-----------------------------
always @ (posedge clk)
    begin : FSM_SEQ
        if (rst_p == 1'b1)
            state <=  idle;
        else
            state <=  next_state;
    end

//////////////////////////////////////////////////////////////

// stop_process is High when there is no enough information to know what to
do
assign stop_process = ((state == round && in_en == 1'b1 && cnt_16 == 15) ||
(state == waiting && done == 1'b0 && in_en == 1'b1)) ? 1'b1 : 1'b0;

//  err is High when an illogical scenario occurs
assign err = ((state == round && cnt_16 != 15 && done == 1'b1) || ( state
== fin && (cnt_16 != 15 || cnt_10 != 9) && done ==1'b1) ||(in_en== 1'b0 &&
done== 1'b1)) ? 1'b1 : 1'b0;

// xor_fin is high when finalization process is starting
assign xor_fin = (state == fin && in_en == 1'b0 & done == 1'b0 && cnt_10
==0 && cnt_16 == 0) ? 1'b1 : 1'b0;

// out_en1 is high when cubehash is finished successfully
assign out_en1 = (state == fin && cnt_10 == 9 && cnt_16 == 15) ? 1'b1 :
1'b0;




////////////////////////////////////////////////////////////////////////

// a signal that indicate whether cubehash has finished. it passed to other
blocks one clock after it really occurs because there is a delay of
// one clock cycle till the hash is loaded in the registers
  always @( posedge clk)
      if (rst_p == 1'b1)
          out_en <= 1'b0;
      else
          out_en <= out_en1;

////////////////////////////////////////////////////////////////////////

// logic that counts 160 rounds (16*10 rounds)
always @ (posedge clk)
begin
        if (rst_p == 1'b1)
            cnt_10 = 0;
        else if (state == fin && cnt_16 == 15 && cnt_10 != 9)
            cnt_10 = cnt_10 + 1;
end
```

```
////////////////////////////////////////////////////////////////////////

// logic that counts 16 rounds
always @ (posedge clk) // cnt_16
begin
        if (rst_p == 1'b1)
            cnt_16 = 0;
        else if (state == round || state == fin)
            cnt_16 = cnt_16 + 1;
end




endmodule
```

cubehash_round.v:

```
/*
this module does one round as specified on spec.
 */

module cubehash_round(
input [1023:0] Rin,
output  [1023:0] Rout
);


// all the 10 steps

wire [31:0] PLUS1[0:15];
wire [31:0] ROT7[0:15];
wire [31:0] SWAP1[0:15];
wire [31:0] XOR1[0:15];
wire [31:0] SWAP2[0:15];
wire [31:0] PLUS2[0:15];
wire [31:0] ROT11[0:15];
wire [31:0] SWAP3[0:15];
wire [31:0] XOR2[0:15];
wire [31:0] SWAP4[0:15];

// when we use it as two-dimensional it's easier to understand what happens
on this module
wire [31:0] arr_Rin [0:31];
wire [31:0] arr_Rout [0:31];

genvar i;

for (i = 0; i < 32; i = i + 1)
    assign arr_Rin[i] = Rin[1023-32*i: 992 - 32*i];



//------------------------------------------- here the magic of the round
is happening


//1
```

```
for (i = 0; i < 16; i = i + 1)
    assign  PLUS1[i] = arr_Rin[i] + arr_Rin[i + 16];

//2
for (i = 0; i < 16; i = i + 1)
    assign ROT7[i] = {arr_Rin[i][24:0],arr_Rin[i][31:25]};


//3
for (i = 0; i < 8; i = i + 1) begin
    assign SWAP1[i] = ROT7[i+8];
    assign SWAP1[i + 8] = ROT7[i];
end

//4
for (i = 0; i < 16; i = i + 1)
    assign XOR1[i] = PLUS1[i] ^ SWAP1[i];

//5
for (i = 0; i < 2; i = i + 1) begin
    assign SWAP2[i]    = PLUS1[i+2];
    assign SWAP2[i+2]  = PLUS1[i];
    assign SWAP2[i+4]  = PLUS1[i+6];
    assign SWAP2[i+6]  = PLUS1[i+4];
    assign SWAP2[i+8]  = PLUS1[i+10];
    assign SWAP2[i+10] = PLUS1[i+8];
    assign SWAP2[i+12] = PLUS1[i+14];
    assign SWAP2[i+14] = PLUS1[i+12];
end

//6
for (i = 0; i < 16; i = i + 1)
    assign PLUS2[i] = XOR1[i] + SWAP2[i];

//7
for (i = 0; i < 16; i = i + 1)
    assign ROT11[i] = {XOR1[i][20:0],XOR1[i][31:21]};

//8
for (i = 0; i < 4; i = i + 1) begin
    assign SWAP3[i]    = ROT11[i + 4];
    assign SWAP3[i + 4]  = ROT11[i];
    assign SWAP3[i + 8]  = ROT11[i + 12];
    assign SWAP3[i + 12] = ROT11[i + 8];
end

//9
for (i = 0; i < 16; i = i + 1)
    assign XOR2[i] = SWAP3[i] ^ PLUS2[i];

//10
for (i = 0; i < 8; i = i + 1) begin
    assign SWAP4[i*2]   = PLUS2[i*2+1];
    assign SWAP4[i*2+1] = PLUS2[i*2];
end

//-------------------------------------------- here the magic of the round
is done

// receiving the results
for (i = 0; i < 16; i= i+1) begin
```

```
    assign arr_Rout[i] = XOR2[i];
    assign arr_Rout[i+16] = SWAP4[i];
end


// one dimensional array because the it's easier like this for the rest of
the code
 for (i = 0; i < 32; i = i + 1)
    assign Rout[1023-32*i: 992 - 32*i] = arr_Rout[i];


endmodule
```

byte_to_256.v:

```
/*
This module receives 32 times 8 bits from input pins and concatenate it to
256 block to do encryption.

when it finishes receiving, transmit a pulse (done) that confirm that the
block is ready to encrypt
 */

module byte_to_256(
    input rst_p,            // active high synchronous reset
    input in_en,            // High when there are more blocks to send for
encryption
    input clk,              // internal 100MHz clock
    input [7:0] part_block1,    // 8 bits as part of a block
    input load1,            // when high- new 8 bits had been loaded to
pins
    output reg  [255:0] block,//concatenated 32 - 8 bits
    output reg done         // pulse when all 32 bytes of block arrived
    );
```

```
reg [4:0] adrs;
reg msb_adrs;

reg [255:0] tmp_block;
wire tc;
wire load_en, load_block;
reg r1, r2, r3 ,r4;

wire done_en;

reg [7:0] part_block;
reg load;

/*
Because the Raspberry Pi Zero pulses are not exact with their length and
not reliable,
we made logic that detects positive edges and do an internal - clock width
- pulse and reliable.

 */


always @(posedge clk) begin // logic that avoids Z state on  inputs.
    if ( rst_p == 1'b1)
        load <= 0;
    else if(in_en == 1'b1)
        load <= load1;
end

always @(posedge clk) begin // logic that does a copy of load delayed one
and 2 cycles
    if ( rst_p == 1'b1) begin
        r3 <= 1'b0;
        r4 <= 1'b0;
    end
    else if(in_en == 1'b1) begin
        r3 <= load;
        r4 <= r3;
    end
end

assign load_en = load & !r3; // pulse of load

always @(posedge clk) begin  // load registers when the pins are loaded
with the information.
    if ( rst_p == 1'b1)
        part_block <= 0;
    else if(load_en == 1'b1)
        part_block <= part_block1;
end


assign load_block = r3 & !r4; // delayed pulse of load

always @ (posedge clk) begin

    if (rst_p == 1'b1) begin // logic that concatenates 32 - 8 bits to a 32
bytes block array
        adrs <= 6'h00;
        tmp_block <= {255{1'b0}};
    end
    else if (load_block == 1'b1) begin // load_block
        tmp_block[248 - 8*adrs+:8] <= part_block;
        adrs <= adrs +1;                      // each new 8 bits the address
proceed in one
    end
end


always @(posedge clk) begin // logic that send the new block when the whole
block arrived
    if (rst_p == 1'b1)
        block <= {255{1'b0}};
    else if(done_en ==1'b1)
        block <= tmp_block;
    end

// below there is a logic that sends the done signal a clock delayed after
it enters to registers
always @(posedge clk) begin
    if (rst_p == 1'b1)
        msb_adrs <= 1'b0;
    else if(load_en == 1'b1)
            msb_adrs <= adrs[4];
end


assign tc = (msb_adrs == 1'b1 && adrs == 5'b00000) ? 1'b1 : 1'b0;

always @(posedge clk) begin //change
    if (rst_p == 1'b1) begin
        r1 <= 1'b0;
    end
    else begin
        r1 <=tc;
    end
end


assign done_en = !r1 && tc; //change



always @ (posedge clk) // sends done signal to other blocks
begin
    if(rst_p == 1'b1)
        done <= 1'b0;
    else begin
        done <= done_en;
    end
end

endmodule
```

top_level.v:

```
/*
this module specifies the instantiation between all the modules.
```

```verilog
we chose to do cubehash in iterative implementation,
due consideration of optimization.

there is no need to faster implementation that uses more area because
with our resources of external board, a faster implementation
will not benefit





 */

module top_level(
input clk,              // internal 100MHz clock
input rst_p,            // active high synchronous reset - recommended from
a button
input in_en,            // High when there are more blocks to send for
encryption
input [7:0] part_block, // 8 bits as part of a block
input load,             // when high- new 8 bits had been loaded to pins
input start,            // pulse when the first block had been transmitted
output level_out_en,    // a stable constant signal that inform the other
board that cubehash finished
output err,             // high when an illogical scenario occurs
output [7:0] part_hash, // 8 bits of the 256 bits hashed message
output load_rpi0,       // High when needs to inform Rpi Zero to read the
pins of 8 bits of hashed message
output level_fall_rst,  // stable pulse of RESET
output hash_ready_led   // a led turns on when the hash is ready
);


wire [1023:0] round_input,round_output;
wire [255:0] hash;
wire [255:0] block;
wire stop_process;
wire done;
wire xor_fin;
wire out_en;



cubehash_round instan1(

.Rin(round_input),
.Rout(round_output)
);

controller_FSM instan2(
.clk(clk),
.rst_p(rst_p),
.in_en(in_en),
.start1(start),
.done(done),
.out_en(out_en),
.xor_fin(xor_fin),
.stop_process(stop_process),
.err(err)
);
```

```
byte_to_256 instan3(
.rst_p(rst_p),
.clk(clk),
.part_block1(part_block),
.load1(load),
.block(block),
.done(done),
.in_en(in_en)
);

out_rst instan4(
.clk(clk),
.rst_p(rst_p),
.level_fall_rst(level_fall_rst)
);

hash256_to_byte instan5(
.rst_p(rst_p),
.clk(clk),
.hash(hash),
.out_en(out_en),
.part_hash(part_hash),
.load_rpi0(load_rpi0)
);

cubehash instan6(
.clk(clk),
.rst_p(rst_p),
.round_input(round_input),
.round_output(round_output),
.hash_big_endian(hash),
.stop_process(stop_process),
.done(done),
.xor_fin(xor_fin),
.out_en,
.block(block),
.hash_ready_led(hash_ready_led),
.level_out_en(level_out_en),
.in_en(in_en),
.start1(start)

);



endmodule
```

hash256_to_byte.v:

```
/*

At this module when the hash is ready - 32 registers are loaded with the 32
bytes of the hash
and each clock enable the information continue to next register till all
the bits had been transmitted.
```

```
according to our experiments, the optimal pulse for the Raspberry Pi Zero
load signal is 20ms,
less than this it won't notice the pulse

 */

module hash256_to_byte(
    input rst_p,            // active high synchronous reset
    input clk,              // internal 100MHz clock
    input [255:0] hash,     // hashed message to transmit
    input out_en,           // when High start transmitting
    output [7:0] part_hash, // 8 bits of the 256 bits hashed message
    output load_rpi0    // High when needs to inform Rpi Zero to read the
pins of 8 bits of hashed message
    );

reg [21:0] cnt22bit;    // counts the time load_rpi0 is high and low

wire clk_en;            // proceeding the hashed message on the 32
registers

reg level_out_en;       // High when the hashed message is ready.


// declaration of 32 registers of hashed message
reg [7:0] hash_div0;
reg [7:0] hash_div1;
reg [7:0] hash_div2;
reg [7:0] hash_div3;
reg [7:0] hash_div4;
reg [7:0] hash_div5;
reg [7:0] hash_div6;
reg [7:0] hash_div7;
reg [7:0] hash_div8;
reg [7:0] hash_div9;
reg [7:0] hash_div10;
reg [7:0] hash_div11;
reg [7:0] hash_div12;
reg [7:0] hash_div13;
reg [7:0] hash_div14;
reg [7:0] hash_div15;
reg [7:0] hash_div16;
reg [7:0] hash_div17;
reg [7:0] hash_div18;
reg [7:0] hash_div19;
reg [7:0] hash_div20;
reg [7:0] hash_div21;
reg [7:0] hash_div22;
reg [7:0] hash_div23;
reg [7:0] hash_div24;
reg [7:0] hash_div25;
reg [7:0] hash_div26;
reg [7:0] hash_div27;
reg [7:0] hash_div28;
reg [7:0] hash_div29;
reg [7:0] hash_div30;
reg [7:0] hash_div31;

// to facilitate writing the hashed message will be on two-dimensional
array  32*8
wire [7:0] arr_hash [0:31];
```

```verilog
reg [4:0] cnt_32; // counts 32 pulses of clk_en

wire finish_transmitting;


// a small FSM to control the sending information process
parameter SIZE = 2;
parameter idle  = 2'b01,send_bytes = 2'b10;

//-------------Internal Variables---------------------------
reg   [SIZE-1:0]          state        ;// Seq part of the FSM
reg   [SIZE-1:0]          next_state   ;// combo part of FSM

always @ (*)
begin : FSM_COMBO_SEND
    next_state = 2'b00;
    case(state)

////////////////////////////////////////////////////////////////////////

        idle :

        if (out_en == 1'b1)
            next_state = send_bytes;
        else
            next_state = idle;

////////////////////////////////////////////////////////////////////////

        send_bytes :

        if (finish_transmitting == 1'b0)
            next_state = send_bytes;
        else
            next_state = idle;

////////////////////////////////////////////////////////////////////////

        default : next_state = idle;

    endcase
end



always @ (posedge clk)
begin : FSM_SEQ_SEND
  if (rst_p == 1'b1) begin
    state <=  idle;
  end else begin
    state <=  next_state;
  end
end


// wiring the hashed message in two - dimensional array
genvar i;

for (i = 0; i < 32; i = i + 1)
```

```
    assign arr_hash[i] = hash[255-8*i: 248 - 8*i];


// logic that generates a 22 bits counter, starting when the hash is ready
always @ (posedge clk) begin

    if(rst_p == 1'b1)
        cnt22bit <= 0;
    else if(level_out_en == 1'b1 && state == send_bytes)
        cnt22bit <= cnt22bit + 1;
end


//logic that generates a pulse when new 8 bits should load to the pins of
Basys 3 (load_rpi0 will be high also)
assign clk_en = (cnt22bit == 22'h3fffff && cnt_32 != 5'h1f) ? 1'b1 : 1'b0;




// logic that creates a delayed pulse when the hash is ready (one clock
after it's ready)
reg r1;
wire r2;

always @ (posedge clk) begin
    if(rst_p == 1'b1)
        r1 <= 1'b0;
    else
        r1 <= level_out_en ;
end

assign r2 = level_out_en  & !r1; // r2 will be High for one clock just
after one clock the hash is ready.



always @ (posedge clk) begin


    if (rst_p == 1'b1) begin    // reset the 32 registers

        hash_div0   <= 8'h00;
        hash_div1   <= 8'h00;
        hash_div2   <= 8'h00;
        hash_div3   <= 8'h00;
        hash_div4   <= 8'h00;
        hash_div5   <= 8'h00;
        hash_div6   <= 8'h00;
        hash_div7   <= 8'h00;
        hash_div8   <= 8'h00;
        hash_div9   <= 8'h00;
        hash_div10  <= 8'h00;
        hash_div11  <= 8'h00;
        hash_div12  <= 8'h00;
        hash_div13  <= 8'h00;
        hash_div14  <= 8'h00;
        hash_div15  <= 8'h00;
        hash_div16  <= 8'h00;
        hash_div17  <= 8'h00;
        hash_div18  <= 8'h00;
        hash_div19  <= 8'h00;
        hash_div20  <= 8'h00;
        hash_div21  <= 8'h00;
        hash_div22  <= 8'h00;
        hash_div23  <= 8'h00;
        hash_div24  <= 8'h00;
        hash_div25  <= 8'h00;
        hash_div26  <= 8'h00;
        hash_div27  <= 8'h00;
        hash_div28  <= 8'h00;
        hash_div29  <= 8'h00;
        hash_div30  <= 8'h00;
        hash_div31  <= 8'h00;

    end
    else if(out_en == 1'b1 || r2 == 1'b1)   begin   // initialize them with
the hash

        hash_div0   <=      arr_hash[0] ;
        hash_div1   <=      arr_hash[1] ;
        hash_div2   <=      arr_hash[2] ;
        hash_div3   <=      arr_hash[3] ;
        hash_div4   <=      arr_hash[4] ;
        hash_div5   <=      arr_hash[5] ;
        hash_div6   <=      arr_hash[6] ;
        hash_div7   <=      arr_hash[7] ;
        hash_div8   <=      arr_hash[8] ;
        hash_div9   <=      arr_hash[9] ;
        hash_div10  <=      arr_hash[10];
        hash_div11  <=      arr_hash[11];
        hash_div12  <=      arr_hash[12];
        hash_div13  <=      arr_hash[13];
        hash_div14  <=      arr_hash[14];
        hash_div15  <=      arr_hash[15];
        hash_div16  <=      arr_hash[16];
        hash_div17  <=      arr_hash[17];
        hash_div18  <=      arr_hash[18];
        hash_div19  <=      arr_hash[19];
        hash_div20  <=      arr_hash[20];
        hash_div21  <=      arr_hash[21];
        hash_div22  <=      arr_hash[22];
        hash_div23  <=      arr_hash[23];
        hash_div24  <=      arr_hash[24];
        hash_div25  <=      arr_hash[25];
        hash_div26  <=      arr_hash[26];
        hash_div27  <=      arr_hash[27];
        hash_div28  <=      arr_hash[28];
        hash_div29  <=      arr_hash[29];
        hash_div30  <=      arr_hash[30];
        hash_div31  <=      arr_hash[31];


    end
    else if(clk_en == 1'b1 && state == send_bytes)  begin   // move content
to next reg when clk_en has pulse

        hash_div0   <=  hash_div1   ;
        hash_div1   <=  hash_div2   ;
        hash_div2   <=  hash_div3   ;
        hash_div3   <=  hash_div4   ;
```

```verilog
        hash_div4   <=  hash_div5   ;
        hash_div5   <=  hash_div6   ;
        hash_div6   <=  hash_div7   ;
        hash_div7   <=  hash_div8   ;
        hash_div8   <=  hash_div9   ;
        hash_div9   <=  hash_div10  ;
        hash_div10  <=  hash_div11  ;
        hash_div11  <=  hash_div12  ;
        hash_div12  <=  hash_div13  ;
        hash_div13  <=  hash_div14  ;
        hash_div14  <=  hash_div15  ;
        hash_div15  <=  hash_div16  ;
        hash_div16  <=  hash_div17  ;
        hash_div17  <=  hash_div18  ;
        hash_div18  <=  hash_div19  ;
        hash_div19  <=  hash_div20  ;
        hash_div20  <=  hash_div21  ;
        hash_div21  <=  hash_div22  ;
        hash_div22  <=  hash_div23  ;
        hash_div23  <=  hash_div24  ;
        hash_div24  <=  hash_div25  ;
        hash_div25  <=  hash_div26  ;
        hash_div26  <=  hash_div27  ;
        hash_div27  <=  hash_div28  ;
        hash_div28  <=  hash_div29  ;
        hash_div29  <=  hash_div30  ;
        hash_div30  <=  hash_div31  ;
    end


end


assign part_hash = hash_div0;   // the 8 bits that need to be transmitted
are always on the last register.


// logic that generates a pulse when the data is ready to be transmitted
with 50% duty cycle.

/* always @ (posedge clk) begin

    if(rst_p == 1'b1)
        load_rpi0 <= 0;
    else if(cnt22bit <=8'h7f && state == send_bytes) //change
        load_rpi0 <= 1'b1;
    else
        load_rpi0 <= 1'b0;
end */

assign load_rpi0 = (state == idle) ? 1'b0 : !cnt22bit[21];
////////////////////////////////////////////////////////////

// logic that counts 32 bytes of hash
always @ (posedge clk) begin

    if(rst_p == 1'b1)
        cnt_32 <= 0;
    else if(clk_en == 1'b1 && state == send_bytes)
        cnt_32 <= cnt_32 + 1;
end
```

```
// a flag that inform the FSM that all the data had been transmitted
assign finish_transmitting = (cnt22bit == 22'h3fffff && cnt_32 == 5'h1f) ?
1'b1 : 1'b0;



/////////////////////////////////////////////////////////////////

always @ (posedge clk) begin
    if(rst_p == 1'b1)
        level_out_en <= 1'b0;
    else
        level_out_en <= out_en | level_out_en;
end






endmodule
```

out_rst.v:

```
/*
we made tests and we found out that if we want that a signal will pass
through
the jumper wires and the Raspberry Pi Zero will notice them,
they should be stable for relatively long time.

At this module we send this kind of signal when RESET button is pressed,
send a stable signal and ignore bouncing

 */


module out_rst (
input clk,                // internal 100MHz clock
input rst_p,              // active high synchronous reset
output level_fall_rst     // stable pulse of RESET
);





reg [12:0] cntr_fall; // large counter for sufficient time

always @(posedge clk) begin

    if (rst_p == 1'b1)
        cntr_fall <= 0;
    else if(cntr_fall < 13'h1fff)
        cntr_fall <= cntr_fall + 1;

end

assign level_fall_rst = (cntr_fall != 13'h1fff) ? 1'b1 : 1'b0; // High till
the counter achieves is Max value.
```

```
endmodule
```

Python file: (we kindly do not recommend to use this version due python language is sensitive to indentation, and “copy & paste” may change indentation and perhaps it won’t work. We recommend to use the Run_Cubehash.py file directly.)

```
import binascii
import time
import  RPi.GPIO as GPIO
GPIO.setmode(GPIO.BOARD)

##############

# GPIO
part_hash = [8, 10, 12, 16, 18, 22, 24, 26] # part of cubehash result
load_hash = 36 # indicate to read part_hash
part_msg = [19, 21, 23, 29, 31, 33, 35, 37] # part of message
in_en = 5 # indicates that there are blocks to send
start = 7 # indicates that the first block was sent
load = 11 # load message
done = 40 # indicates that the encryption process is done
level_rst = 38 # steady signal of reset
block = 256 # Block size
cipher = ['']*256

def setup(): #setups and reset the pins.
    GPIO.setup(part_msg, GPIO.OUT)
    GPIO.setup(load, GPIO.OUT)
    GPIO.setup(start, GPIO.OUT)
    GPIO.setup(in_en, GPIO.OUT)
    GPIO.setup(done, GPIO.IN)
    GPIO.setup(level_rst, GPIO.IN)
    GPIO.setup(part_hash, GPIO.IN)
    GPIO.setup(load_hash, GPIO.IN)
    GPIO.output(part_msg, GPIO.LOW)
    GPIO.output(in_en, GPIO.LOW)
        GPIO.output(start, GPIO.LOW)
        GPIO.output(load, GPIO.LOW)

def recieve(): # recieve the result of cubehash

    for i in range(32):
        if GPIO.input(load_hash) == 1:


            for j in range (8):
                cipher[j+i*8] = GPIO.input(part_hash[j])
            while GPIO.input(load_hash) == 1:
                pass
            if i != 31:
                while GPIO.input(load_hash) == 0:
                    pass

    return map(str, cipher)


def pad(msg): # padding the message
    if msg != "":
        if msg[0] == "0" or msg[0] == "1" or msg[0] == "2" or msg[0] == "3"
or msg[0] == "4" or msg[0] == "5" or msg[0] == "6" or msg[0] == "7" or
msg[0] == "8" or msg[0] == "9":
            msg = bin(int(binascii.hexlify(msg), 16)) #convert ASCII to
binary.
            msg = msg.replace("b", "0") # terminating parasitic "b". (we
know it's in binary!)
        else:
            msg = bin(int(binascii.hexlify(msg), 16)) #convert ASCII to
binary.
                        msg = msg.replace("b", "") # terminating parasitic
"b". (we know it's in binary!)

    tail = block - (len(msg)%block)
    if tail != 256:
        msg = msg + '1'
        for i in range(tail - 1):
            msg = msg +'0'
    else:
        msg = msg + '1'
        for i in range(255):
            msg = msg + '0'
    return msg


def endian(msg, blocks): # makes the message in little endian format
    tmp_msg = ['']*256
    msg = list(msg)
    for j in range(blocks):
        for i in range(256*j, 256 + 256*j, 32):
            tmp_msg[i % 256: (i + 8) % 256: 1] = msg[i+24: i + 32: 1]
            tmp_msg[(i + 8) % 256: (i + 16) % 256: 1] = msg[i + 16: i + 24:
1]

            tmp_msg[(i + 16) % 256: (i + 24) % 256: 1] = msg[i + 8: i + 16:
1]

            tmp_msg[(i + 24) % 256: (i + 32) % 256: 1] = msg[i: i + 8: 1]

                msg[256*j: 256 + 256*j:1] = tmp_msg[0:256:1]
    return map(int, msg)



def transmit(msg, blocks): # Transmit the message in packages of 8 bits
    GPIO.output(in_en, GPIO.HIGH)
    for j in range(blocks):
        for i in range (256*j, 248 + 256*j, 8):
            for P in range (8):
                GPIO.output(part_msg[P], msg[i+P])
            GPIO.output(load, GPIO.HIGH)

            GPIO.output(load, GPIO.LOW)
        if j == 0:
            GPIO.output(start, GPIO.HIGH)
        for i in range (248 + 256*j, 256 + 256*j):
            GPIO.output(part_msg[i%8], msg[i])

        GPIO.output(load, GPIO.HIGH)

        GPIO.output(load, GPIO.LOW)
        GPIO.output(start, GPIO.LOW)

    GPIO.output(in_en, GPIO.LOW)
def main():
    setup()
    msg = raw_input('\n' "enter a  message: ")
    msg = pad(msg)
    blocks = len(msg)/block
    msg = endian(msg, blocks)
    print  '\n',  "Please press on the center button of the Basys 3 "
    GPIO.wait_for_edge(level_rst, GPIO.RISING)
    time.sleep(1)
    transmit(msg, blocks)



    if GPIO.input(done) == 1:
        cipher = recieve()
        hex_cipher = ['']*64
        for i in range (64):
                    hex_cipher[i] = hex(int("".join(cipher[4*i: 4 + 4*i:
1]), 2))
        final_answer = "".join(hex_cipher)
        final_answer = final_answer.replace("0x", "")
        print '\n', '\n', "The cipher text is: ", '\n' ,final_answer ,
'\n', '\n'
        GPIO.output(part_msg, GPIO.LOW)
        GPIO.cleanup()
main()
```

XDC file:

```
## Clock CONNECTED TO INTERNAL CLOCK

##clk
set_property PACKAGE_PIN W5 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
create_clock -period 10.000 -name clk -waveform {0.000 5.000} [get_ports
clk]


#### RESET CONNECTED TO BUTTON CENTER ####

##rst
set_property PACKAGE_PIN U18 [get_ports rst_p]
set_property IOSTANDARD LVCMOS33 [get_ports rst_p]


###############
# INPUT DELAY #
###############

set_input_delay -clock clk -add_delay 0.000 [get_ports part_block*]
set_input_delay -clock clk -add_delay 0.000 [get_ports rst_p]
set_input_delay -clock clk -add_delay 0.000 [get_ports in_en]
set_input_delay -clock clk -add_delay 0.000 [get_ports start]
set_input_delay -clock clk -add_delay 0.000 [get_ports load]



################
# OUTPUT DELAY #
################

set_output_delay -clock clk -add_delay 0.000 [get_ports level_out_en]
set_output_delay -clock clk -add_delay 0.000 [get_ports part_hash*]
set_output_delay -clock clk -add_delay 0.000 [get_ports err]
set_output_delay -clock clk -add_delay 0.000 [get_ports level_fall_rst]
set_output_delay -clock clk -add_delay 0.000 [get_ports load_rpi0]
set_output_delay -clock clk -add_delay 0.000 [get_ports hash_ready_led]




#### PMODs CONNENCTED TO PMOD C ####

## INPUT

##JC1
set_property PACKAGE_PIN K17 [get_ports {part_block[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {part_block[0]}]
set_property PACKAGE_PIN M18 [get_ports {part_block[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {part_block[1]}]
set_property PACKAGE_PIN N17 [get_ports {part_block[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {part_block[2]}]
set_property PACKAGE_PIN P18 [get_ports {part_block[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {part_block[3]}]
set_property PACKAGE_PIN L17 [get_ports {part_block[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {part_block[4]}]
set_property PACKAGE_PIN M19 [get_ports {part_block[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {part_block[5]}]
set_property PACKAGE_PIN P17 [get_ports {part_block[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {part_block[6]}]
##JC1
set_property PACKAGE_PIN R18 [get_ports {part_block[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {part_block[7]}]




#### PMODs CONNENCTED TO PMOD B ####

##INPUT

##JB1
set_property PACKAGE_PIN A14 [get_ports in_en]
set_property IOSTANDARD LVCMOS33 [get_ports in_en]

##JB2
set_property PACKAGE_PIN A16 [get_ports start]
set_property IOSTANDARD LVCMOS33 [get_ports start]

##JB3
set_property PACKAGE_PIN B15 [get_ports load]
set_property IOSTANDARD LVCMOS33 [get_ports load]



##OUTPUT

##JB10
set_property PACKAGE_PIN C16 [get_ports level_out_en]
set_property IOSTANDARD LVCMOS33 [get_ports level_out_en]

##JB4
set_property PACKAGE_PIN B16 [get_ports level_fall_rst]
set_property IOSTANDARD LVCMOS33 [get_ports level_fall_rst]

##JB7
set_property PACKAGE_PIN A15 [get_ports load_rpi0]
set_property IOSTANDARD LVCMOS33 [get_ports load_rpi0]



#### PMODs CONNENCTED TO PMOD A ####

##OUTPUT
##JA1
set_property PACKAGE_PIN J1 [get_ports {part_hash[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {part_hash[0]}]
set_property PACKAGE_PIN L2 [get_ports {part_hash[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {part_hash[1]}]
set_property PACKAGE_PIN J2 [get_ports {part_hash[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {part_hash[2]}]
set_property PACKAGE_PIN G2 [get_ports {part_hash[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {part_hash[3]}]
set_property PACKAGE_PIN H1 [get_ports {part_hash[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {part_hash[4]}]
set_property PACKAGE_PIN K2 [get_ports {part_hash[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {part_hash[5]}]
set_property PACKAGE_PIN H2 [get_ports {part_hash[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {part_hash[6]}]
##JA10
set_property PACKAGE_PIN G3 [get_ports {part_hash[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {part_hash[7]}]


#### ERROR indicator connected to led LD0 ####

set_property PACKAGE_PIN U16 [get_ports err]
set_property IOSTANDARD LVCMOS33 [get_ports err]






#### finish hashing indicator led LD8 ####

set_property PACKAGE_PIN V13 [get_ports hash_ready_led]
set_property IOSTANDARD LVCMOS33 [get_ports hash_ready_led]
```