Project 10 Simulation of a cryptographic block cipher

Introduction

The implementation of cryptography on hardware accelerates the computation-heavy operations in these algorithms. Cryptographic algorithms can be accelerated on application-specific platforms (ASICs), which do not permit further modifications once manufactured. An alternative platform for the implementation of cryptographic algorithms is FPGA, which facilitates the design process thanks to its reconfigurable approach.

PRESENT cipher is one of the most prominent lightweight block ciphers in the literature and one of the standard lightweight ciphers accepted by ISO.

<u>In this lab, we will understand how to simulate the PRESENT cipher in Vivado.</u>

Before you begin, you should:

- Understand the design and operation of arithmetic circuits;
- Understand how to specify and design combinational circuits;
- Understand how to specify and design simple sequential circuits;
- Be a confident and capable Vivado user.

After you're done, you should:

- Understand how simple block ciphers function;
- Be able to describe a simple block cipher in Verilog;
- Be comfortable designing more complex combinational and sequential circuits.

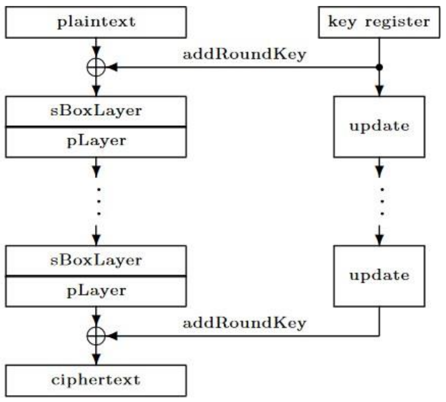
Requirements

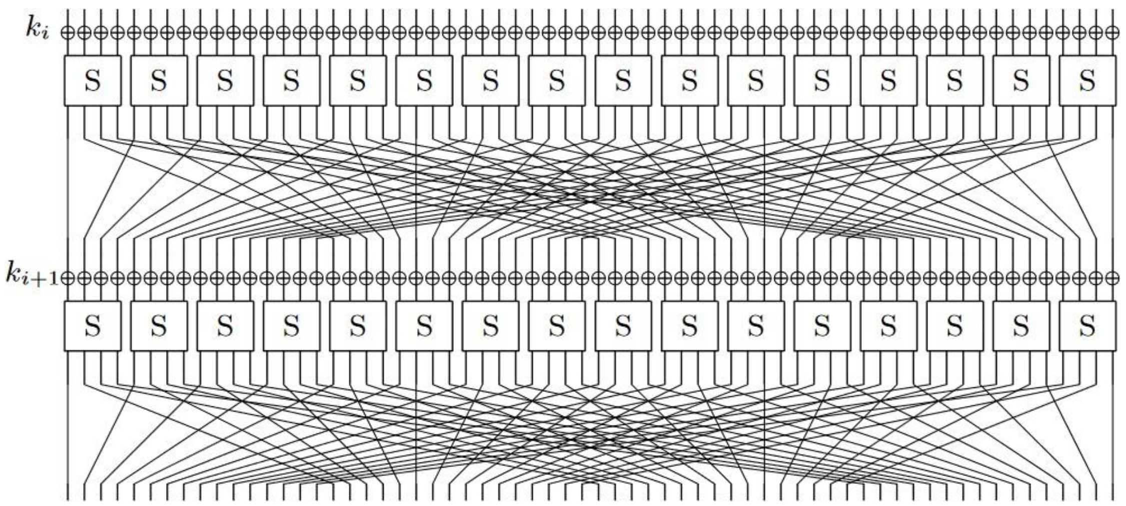
✓ **PRESENT Cipher in Hardware**

PRESENT cipher specification is given at **https://www.iacr.org/archive/ches2007/47270450/47270450.pdf**

A block diagram of the PRESENT algorithm is presented as follows in this specification.

```
generateRoundKeys()
for $i = 1$ to 31 do
    addRoundKey(STATE,$K_i$)
    sBoxLayer(STATE)
    pLayer(STATE)
end for
addRoundKey(STATE,$K_{32}$)
```





sBoxLayer, pLayer, and key update are described as follows.

sBoxlayer. The S-box used in PRESENT is a 4-bit to 4-bit S-box $S : \mathbb{F}_2^4 \rightarrow \mathbb{F}_2^4$. The action of this box in hexadecimal notation is given by the following table.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

pLayer. The bit permutation used in PRESENT is given by the following table. Bit i of STATE is moved to bit position $P(i)$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P(i)$	0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P(i)$	4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$P(i)$	8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$P(i)$	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63

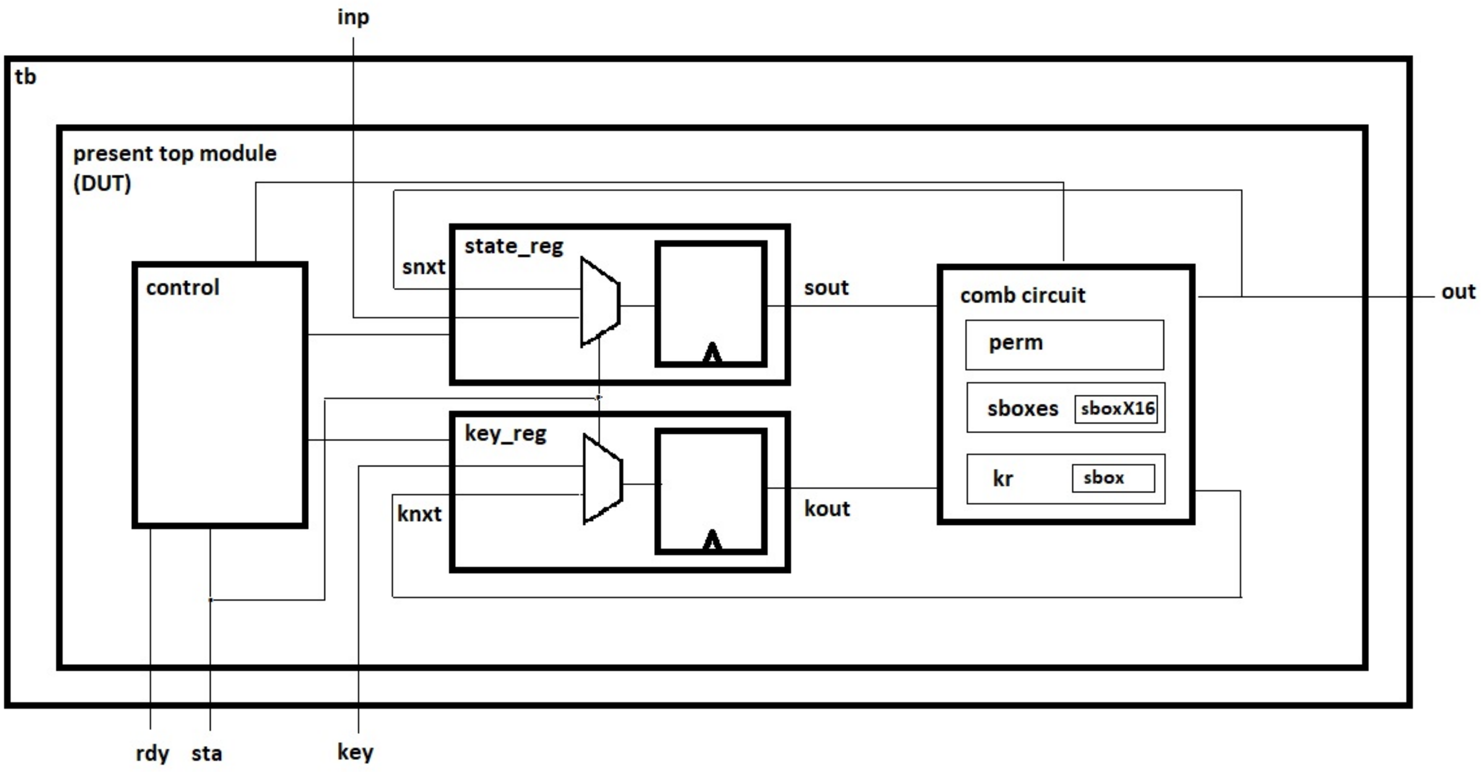
The key schedule. PRESENT can take keys of either 80 or 128 bits. However we focus on the version with 80-bit keys. The user-supplied key is stored in a key register K and represented as $k_{79}k_{78}\dots k_0$. At round i the 64-bit round key $K_i = \kappa_{63}\kappa_{62}\dots\kappa_0$ consists of the 64 leftmost bits of the current contents of register K. Thus at round i we have that:

$$K_i = \kappa_{63}\kappa_{62}\dots\kappa_0 = k_{79}k_{78}\dots k_{16}.$$

After extracting the round key K_i, the key register $K = k_{79}k_{78}\dots k_0$ is updated as follows.

1. $[\mathtt{k}_{79}\mathtt{k}_{78}\dots\mathtt{k}_1\mathtt{k}_0] = [\mathtt{k}_{18}\mathtt{k}_{17}\dots\mathtt{k}_{20}\mathtt{k}_{19}]$
2. $[\mathtt{k}_{79}\mathtt{k}_{78}\mathtt{k}_{77}\mathtt{k}_{76}] = S[\mathtt{k}_{79}\mathtt{k}_{78}\mathtt{k}_{77}\mathtt{k}_{76}]$
3. $[\mathtt{k}_{19}\mathtt{k}_{18}\mathtt{k}_{17}\mathtt{k}_{16}\mathtt{k}_{15}] = [\mathtt{k}_{19}\mathtt{k}_{18}\mathtt{k}_{17}\mathtt{k}_{16}\mathtt{k}_{15}] \oplus \mathtt{round\_counter}$

An example implementation is provided for demonstration in this lab. The block diagram for the hardware design of PRESENT is as shown below.



Create a new project on Vivado and select xc7s50csga324-1 device for the Boolean Board.

The example design given above implements a hierarchical approach. We will generate 9 design sources (sbox, sboxes, perm, kr, comb, control, kreg, sreg, and present) as shown in the above figure.

We first generate state and key registers **sreg.v** and **kreg.v** as separate files as follows.

```
/////////////////////////////////////////////////
// State register
//
module  sreg  ( sta , act , inp , nxt , ck , rn , out ) ;


input [0:63] inp ; // State initial data
input [0:63] nxt ; // Next value of state
input        act ;  // Active signal
input        sta ;  // Start signal
input        ck ; // Rising edge clock
input        rn  ;  // Active low reset

output [0:63] out ; // State output

reg    [0:63]  out ;  // State output


always @ ( posedge ck  or  negedge rn )
  if     ( !rn )  out  <=  0 ;
  else if ( sta ) out <= inp ;
  else if ( act )  out  <=  nxt ;


endmodule
```

```
/////////////////////////////////////////////////
// Key register
//
module  kreg  ( sta , act , inp , nxt , ck , rn , out ) ;


input [0:79] inp ;  // Key initial data
input [0:79] nxt ; // Next value of key
input           act ;  // Active signal
input           sta ; // Start signal
input           ck ; // Rising edge clock
input           rn ;  // Active low reset

output [0:79] out ; // Key output

reg     [0:79]  out ;  // Key output


always @ ( posedge ck  or  negedge rn )
  if      ( !rn ) out <= 0 ;
  else if ( sta ) out <= inp ;
  else if ( act ) out <= nxt ;


endmodule
```

Next we generate the combinational blocks **sbox.v, sboxes.v, perm.v, and kr.v** as separate files.

```
/////////////////////////////////////////////////
// PRESENT Sbox
//
module sbox  ( a , y ) ;


input  [0:3]  a  ;  //  4-bit  input

output  [0:3]  y  ;  //  4-bit  yput

reg [0:3] y ; // 4-bit yput

// Lookup table
always @ ( * )
case ( a )
           4'h0 :  y  =  4'hC ;
  4'h1  :  y  =  4'h5 ;
  4'h2  :  y  =  4'h6 ;
  4'h3  :  y  =  4'hB ;
  4'h4  :  y  =  4'h9 ;
  4'h5  :  y  =  4'h0 ;
  4'h6  :  y  =  4'hA ;
  4'h7  :  y  =  4'hD ;
  4'h8  :  y  =  4'h3 ;
  4'h9  :  y  =  4'hE ;
  4'hA  :  y  =  4'hF ;
  4'hB  :  y  =  4'h8 ;
  4'hC  :  y  =  4'h4 ;
  4'hD  :  y  =  4'h7 ;
  4'hE  :  y  =  4'h1 ;
  4'hF  :  y  =  4'h2 ;
 endcase


endmodule
```

```
////////////////////////////////////////////////
// PRESENT SBoxes module
//
module  sboxes  ( a , y ) ;


input [0:63] a ; // 64-bit input A output

[0:63]  y ;  // 64-bit output Y

wire [0:3]  ba  [0:15] ;  // Input bytes
wire  [0:3]  by  [0:15] ;  // Output bytes


// Split input into bytes
assign  ba[0]  =  a[0:3] ;
assign ba[1] = a[4:7] ;
assign ba[2] = a[8:11] ;
assign ba[3] = a[12:15] ;
assign ba[4] = a[16:19] ;
assign ba[5] = a[20:23] ;
assign ba[6] = a[24:27] ;
assign ba[7] = a[28:31] ;
assign ba[8] = a[32:35] ;
assign ba[9] = a[36:39] ;
assign ba[10] = a[40:43] ;
assign ba[11] = a[44:47] ;
assign ba[12] = a[48:51] ;
assign ba[13] = a[52:55] ;
assign ba[14] = a[56:59] ;
assign  ba[15]  =  a[60:63] ;

// Sbox modules
sbox  m00  ( .a(ba[0])  , .y(by[0])  ) ;
sbox  m01  ( .a(ba[1])  , .y(by[1])  ) ;
sbox  m02  ( .a(ba[2])  , .y(by[2])  ) ;
sbox  m03  ( .a(ba[3])  , .y(by[3])  ) ;
sbox  m04  ( .a(ba[4])  , .y(by[4])  ) ;
sbox  m05  ( .a(ba[5])  , .y(by[5])  ) ;
sbox  m06  ( .a(ba[6])  , .y(by[6])  ) ;
sbox  m07  ( .a(ba[7])  , .y(by[7])  ) ;
sbox  m08  ( .a(ba[8])  , .y(by[8])  ) ;
sbox  m09  ( .a(ba[9])  , .y(by[9])  ) ;
sbox  m10  ( .a(ba[10]) , .y(by[10]) ) ;
sbox  m11  ( .a(ba[11]) , .y(by[11]) ) ;
sbox  m12  ( .a(ba[12]) , .y(by[12]) ) ;
sbox  m13  ( .a(ba[13]) , .y(by[13]) ) ;
sbox  m14  ( .a(ba[14]) , .y(by[14]) ) ;
sbox  m15  ( .a(ba[15]) , .y(by[15]) ) ;

// Join output bytes
assign  y  =  { by[0]  , by[1]  , by[2]  , by[3]  ,
                by[4]  , by[5]  , by[6]  , by[7]  ,
                by[8]  , by[9]  , by[10] , by[11] ,
                by[12] , by[13] , by[14] , by[15] } ;


endmodule
```

```
////////////////////////////////////////////////
// Permutation block for PRESENT
//
module  perm  ( inp , out ) ;


input [0:63] inp ; // 4-bit input

output  [0:63]  out ;  // 4-bit output


// Permutation
assign  out  =  { inp[00],inp[04],inp[08],inp[12],inp[16],inp[20],inp[24],inp[28],
                  inp[32],inp[36],inp[40],inp[44],inp[48],inp[52],inp[56],inp[60],
                  inp[01],inp[05],inp[09],inp[13],inp[17],inp[21],inp[25],inp[29],
                  inp[33],inp[37],inp[41],inp[45],inp[49],inp[53],inp[57],inp[61],
                  inp[02],inp[06],inp[10],inp[14],inp[18],inp[22],inp[26],inp[30],
                  inp[34],inp[38],inp[42],inp[46],inp[50],inp[54],inp[58],inp[62],
                  inp[03],inp[07],inp[11],inp[15],inp[19],inp[23],inp[27],inp[31],
                  inp[35],inp[39],inp[43],inp[47],inp[51],inp[55],inp[59],inp[63] } ;


endmodule
```

```
///////////////////////////////////////////////////
// Key schedule block for PRESENT
//
module  kr  ( kinp , cnt , kout ) ;


input [0:79] kinp ;
input  [0:4]  cnt  ;

output [0:79] kout ;

wire  [0:79]  krot ;


// Rotate
assign  krot  =  { kinp[61:79] , kinp[0:60] } ;

// Sbox
sbox  m00  ( .a(krot[0:3])  , .y(kout[0:3]) ) ;

// XOR
assign  kout[60:64]  =  cnt  ^  krot[60:64] ;

// Assign rest
assign kout[4:59] = krot[4:59] ;
assign  kout[65:79]  =  krot[65:79] ;


endmodule
```

We now combine these combinational blocks in comb.v file to realize the combinational circuit of PRESENT.

```
///////////////////////////////////////////////////
// Combinational circuit block for PRESENT
//
module  comb  ( sp , kp , cnt , r0 , sn , kn ) ;


input [0:63]  sp  ; // State in
input   [0:79]  kp  ; // Key in
input  [0:4]  cnt ; // Counter
input           r0  ; // Round-0 flag

output [0:63] sn ; // State out
output  [0:79]  kn  ; // Key out

wire   [0:63]  ssb ; // Sbox output
wire    [0:63]  ssp ; // Permutation output
wire [0:63] sss ; // Select sbox or permutation
wire  [0:79]  kkr ; // Key schedule output
wire   [0:79]  kks ; // Select initial key or schedule output


// Sboxes on sp
sboxes  sb ( .a(sp) , .y(ssb) ) ;

// Permutation on ssb
perm  pm ( .inp(ssb) , .out(ssp) ) ;

// select sout
assign  sss  =  r0  ?  sp  :  ssp ;

// Kround
kr  kmod  ( .kinp(kp) , .cnt(cnt) , .kout(kkr) ) ;

// Select kout
assign  kks  =  r0  ?  kp  :  kkr ;

// Sout (sn)
assign  sn  =  sss  ^  kks[0:63] ;

// Kout  (kn)
assign  kn  =  kks ;


endmodule
```

As you may have seen in the previous files, we need certain control signals such as round 0 flag, counter, active signal, and ready signal based on a start signal. Create control.v as follows.

```
////////////////////////////////////////////////
// Control module for PRESENT

module control ( sta , ck , rn , r0 , cnt , act , rdy ) ;



input           sta    ; // Start signal
input           ck     ; // Rising edge clock
input           rn     ; // Active low reset


output          r0     ; // Round-0 flag
output [0:4]   cnt          ; // Counter output
                act     ; // Active signal
output          rdy     ; // Ready signal

reg      [0:4] cnt_ps ; // PS of counter wire
         [0:4] cnt_ns ; // NS of counter wire
                r31     ; // Round-31 flag reg
               act_ps ; // PS of active
wire            act_ns ; // NS of active
reg            rdy     ; // DFF output is ready


// Define counter
//
assign cnt_ns = sta ? 0 : ( act ? cnt_ps+1 : cnt_ps ) ;
//
always @ ( posedge ck or negedge rn ) if ( !rn )
  cnt_ps <= 0 ;
  else          cnt_ps <= cnt_ns ;
//
assign cnt = cnt_ps ;
//
assign r0 = ( cnt_ps == 0 ) ; assign r31 = (
cnt_ps == 31 ) ;

// Define active signal
//
assign act_ns = sta ? 1 : ( r31 ? 0 : act_ps ) ;
//
always @ ( posedge ck or negedge rn ) if ( !rn )
  act_ps <= 0 ;
  else          act_ps <= act_ns ;
//
assign act = act_ps ;

// Delay Round-31 flag to get ready output
//
always @ ( posedge ck or negedge rn ) if ( !rn ) rdy
  <= 0 ;
  else          rdy <= r31 ;

endmodule
```

We finally combine **comb.v, control.v, sreg.v, and kreg.v** as follows.

```
////////////////////////////////////////////////
//Top-level module for PRESENT-80 block cipher
//
module  present  ( sta , inp , key , ck , rn , rdy , out ) ;


input              sta ; // Start signal
input [0:63] inp ; // Data input
input  [0:79]  key ;  // Data input
input              ck  ;  // Rising edge clock
input              rn  ;  // Active low reset

output             rdy ; // Ready flag
output  [0:63]  out ;  // Ciphertext

wire              r0  ;  // Round-0 flag
wire              act ; // Active signal
wire    [0:4]      cnt ;  // Counter

wire   [0:63]   sout ;  // State output
wire  [0:63]  snxt ;  // State next value
wire   [0:79]   kout ;  // Key schedule output
wire  [0:79]  knxt ;  // Key schedule next value


// Control unit
control  u_control  (
  .sta ( sta ) ,
  .ck  ( ck  ) ,
  .rn  ( rn  ) ,
  .r0  ( r0  ) ,
  .cnt ( cnt ) ,
  .act ( act ) ,
  .rdy ( rdy )
) ;

// State registers
sreg  u_sreg  (
  .sta ( sta  ) ,
  .act ( act  ) ,
  .inp ( inp  ) ,
  .nxt ( snxt ) ,
  .ck  ( ck   ) ,
  .rn  ( rn   ) ,
  .out ( sout )
) ;

// Key registers
kreg  u_kreg  (
  .sta ( sta  ) ,
  .act ( act  ) ,
  .inp ( key  ) ,
  .nxt ( knxt ) ,
  .ck  ( ck   ) ,
  .rn  ( rn   ) ,
  .out ( kout )
) ;

// Combinational block (round function and key update)
comb  u_comb  (
  .sp  ( sout ) ,
  .kp  ( kout ) ,
  .cnt ( cnt  ) ,
  .r0  ( r0  ) ,
  .sn  ( snxt ) ,
  .kn  ( knxt )
) ;


// Assign last state output as the final output
assign  out  =  sout ;


endmodule
```

We would like to simulate the PRESENT cipher in Vivado.Create the testbench **present_tb.v** as

follows.

```
/////////////////////////////////////////////////
`timescale  1 ns / 1 ns
/////////////////////////////////////////////////
// Testbench for PRESENT-80 module

module  present_tb ;


parameter  per  =  10 ;  // Clock period for 100 MHz

// Inputs declared as "registers"
reg            sta ; // Start signal
reg [0:63] inp ;  // Data input
reg  [0:79]  key ;  // Key input
reg            ck  ;  // Rising edge clock
reg            rn  ;  // Active low reset

// Outputs declared as "nets"
wire           rdy ;
wire  [0:63]  out ;  // Data output

// Internal variables for simulation control
integer  i ;


// Instantiate device under test (DUT)
present  dut  (
  .sta ( sta ) ,
  .inp ( inp ) ,
  .key ( key ) ,
  .ck  ( ck  ) ,
  .rn  ( rn  ) ,
  .rdy ( rdy ) ,
  .out ( out )
) ;


// Define reset
initial
  begin
    rn <= 1'b0 ;  // Initially reset is "0"
    #(per/2) ;      // After (per/2) nanoseconds
    rn <= 1'b1 ; // reset becomes "1" (inactive)
  end

// Define clock
initial  ck  <=  1'b1 ;  // Initially clock is "1"
always #(per/2) // Every (per/2) ns, clock is toggled,
  ck  <=  ~ ck ;  // resulting in a periodic square wave
```

```
// **Other stimulus**

  initial // Use "initial" to define waveforms
  begin  // starting at zero time

    // Initialize everything first

    sta  <= #1  1'b0 ;
    inp <= #1 64'd0 ;
    key  <= #1  80'd0 ;

    // Wait a few periods, change inputs and send start

    #(3*per) ;  // Wait for 3 periods of time

    inp  <= #1  64'h0000000000000000 ;
    key  <= #1  80'h00000000000000000000 ;
    sta  <= #1  1'b1 ;  // Start pulled up
    #(per) ;            // After one period of time
    sta  <= #1  1'b0 ;  // start is pulled down -> a single pulse

    // Wait until ready becomes "1"

    wait ( rdy ) ;
    #(3*per) ;  // Check to see if output is correct

    // Wait a few periods, change inputs and send start

    #(3*per) ;  // Wait for 3 periods of time
    inp  <= #1  64'hffffffffffffffff ;
    key  <= #1  80'hffffffffffffffffffff ;
    sta  <= #1  1'b1 ;  // Start pulled up

    #(per) ;            // After one period of time

    sta  <= #1  1'b0 ;  // start is pulled down -> a single pulse

    // Wait until ready becomes "1"

    wait ( rdy ) ;
    #(3*per) ;  // Check to see if output is correct
     $stop ;
  end


endmodule
```

As you can see, we connect a clock signal as well as a reset to the circuit. We have a start-ready signaling scheme, which starts the processing and signals when the ciphertext is ready.

In this testbench, we demonstrate the correctness of the implementation based on two test vectors given in the specification.

plaintext	*key*	*ciphertext*
00000000 00000000	00000000 00000000 0000	5579C138 7B228445
00000000 00000000	FFFFFFFF FFFFFFFF FFFF	E72C46C0 F5945049
FFFFFFFF FFFFFFFF	00000000 00000000 0000	A112FFC7 2F68417B
FFFFFFFF FFFFFFFF	FFFFFFFF FFFFFFFF FFFF	3333DCD3 213210D2

Try other test vectors and check their correctness as well.
Observe the signals in the design, for instance, counter signal **cnt.**
You can check the values as hex or as an unsigned decimal.