Tutorial 5: The control and main FPGAs of the Sakura-G board

In this tutorial we will review the default designs of both the control and main FPGAs of the Sakura-G with the aim of loading our own designs. The Sakura-G board consist of two FPGAs that are interconnected through a local bus and a USB FTDI interface that allows an external PC to communicate with the control FPGA. Both FPGAs are connected through a local bus clocked, by default, at 1.5 MHz. The control FPGA can send and receive data to the main FPGA, receives external inputs such as a reset, a 48 MHz clock, the state of the reset push switch and of the on board DIP switch. Finally, it also controls the array of leds located next to the FPGA on the board.

On the reverse of the board there is a communication chip (FTDI FT2232H) with two ports (A,B) that are accessed through the USB port of the Sakura. The other communication side is connected to both FPGAs and can be selected through the DIP switch.

The code for both FPGAs can be downloaded from here[1]. Both designs are written in Verilog. However, Xilinx ISE can synthesize projects both based on VHDL and Verilog so it does not matter if your target design has been written in VHDL.

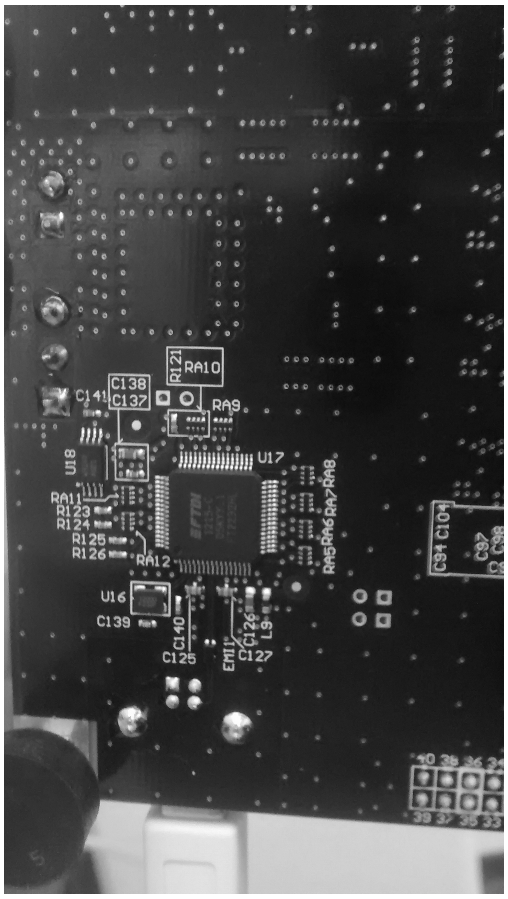


Figure 1: FTDI communication chip on the reverse of the Sakura board

.

[1] http://satoh.cs.uec.ac.jp/SAKURA/hardware/SAKURA-G.html

The control FPGA

The *sakura_g_control* project contains the following components:

- clk_gen: frequency divider that generates the system clock and the clock that is used in the communication bus between the control FPGA and the main FPGA.
- usbif: this component reads and write the Sakura USB interface.
- cipher_if: this component manages the local bus between the control FPGA and the main FPGA.
- rx_fifo, tx_fifo: this FIFOs mediate the communication between the Sakura USB interface, the control FPGA and the main FPGA.

The control FPGA has a clock input of 48 MHz(clkin) and a reset signal (rstnin). The top block groups different sets of ports:

1. There is a group of ports that control the FTDI chip of the FPGA. They receive data from the USB port of the PC. This is a chip FT2232H[2] with two ports (A, related to the control FPGA and B, to the main FPGA).

```
  // FT2232H interface
  input         usb_rxfn,            // USB portA rx flag. High: Not ready
//Low: ready (FTDI_ACBUS0_RXFn_B)
  input         usb_txen,            // USB portA tx enable. High: TX
//disable  Low: TX enable (FTDI_ACBUS1_TXEn_B)
  output        usb_rdn,             // USB portA read enable. Active low
//(FTDI_ACBUS2_RDn_B)
  output        usb_wrn,             // USB portA write enable. Active low
//(FTDI_ACBUS3_WRn_B)
  inout   [7:0] usb_d,               // USB portA data bus (FTDI_ADBUS)
  output        FTDI_ACBUS4_SIWUA,   // *SAKURA-G only
  output        C_FTDI_RESET_B,      // *SAKURA-G only
```

   This chip can be configured either as an UART or as a FIFO, as we did in the prior tutorial. This set of ports is connected to the usbif instance (ft2232h_usbif component), that reads and write the USB port with data coming from the PC and from the main FPGA.

2. Besides, there is another similar set of ports connected to the main FPGA (B PORT) that can be selected and interfaced to the USB port instead of the prior one. This can be done via de DIP switch of the FPGA (related to the input signal C_DIPSW). However, we are not interested in this feature in this tutorial.

```
  // FTDI USB interface portB *SAKURA-G only
  input         PORT_B_RXF,          // USB portB rx flag. High: Not ready
//Low: ready *SAKURA-G only
  input         PORT_B_TXE,          // USB portB tx enable. High: TX
//disable  Low: TX enable *SAKURA-G only
  output        PORT_B_RD,           // USB portB read enable. Active low
//*SAKURA-G only
```

[2]http://www.ftdichip.com/Products/ICs/FT2232H.htm

```
    output        PORT_B_WR,            // USB portB write enable. Active low
  //*SAKURA-G only
    input   [7:0] PORT_B_DIN,           // USB portB data input *SAKURA-G only
    output  [7:0] PORT_B_DOUT,          // USB portB data output *SAKURA-G
  //only
    output        PORT_B_OEn            // USB portB data output enable
  //*SAKURA-G only
```

3. The control FPGA also controls the set of LEDs next to it, together with the input of the push switch that can perform a reset (signal C_PUSHSW).

```
    // LED & Switch
    output [9:0]  led,                  // LED
    input         C_PUSHSW,             // Push switch. Reset input *SAKURA-G
  //only
    input  [3:0]  C_DIPSW,              // DIP switch input *SAKURA-G only
```

4. Both FPGAs are connected through a local bus clocked by default at 1.5 MHz:

```
    // MAIN FPGA Interface
    input         lbus_rdy,             // MAIN FPGA ready
    output        lbus_rstn,            // Local bus(Main FPGA) reset low
    output        lbus_clk,             // Local bus clock(1.5MHz)
    output  [7:0] lbus_wd,              // Local bus write data
    output        lbus_we,              // Local bus write enable
    input         lbus_ful,             // Local bus full

  //input         lbus_aful,            // Local bus almost full   *SAKURA-G
  //intact
    input   [7:0] lbus_rd,              // Local bus read data
    output        lbus_re,              // Local bus read enable
    input         lbus_emp,             // Local bus empty

  //input         lbus_aemp,            // Local bus almost empty  *SAKURA-G
  //intact
```

5. Finally, there are set of signals that show the state of the main FPGA:

```
    // MAIN FPGA configuration signals
    input         cfg_din,              // M_DO_DIN_MISO_R,

    input         cfg_mosi,             // M_MOSI_CSI_B_R

    input         cfg_fcsb,             // M_CSO_B_R,

    input         cfg_cclk,             // M_CCLK_R
    input         cfg_progn,            // M_PROG_B_R
    input         cfg_initn,            // M_INIT_B_R
    input        cfg_rdwrn,          // M_RDWR_B_R

    input        cfg_busy,           // M_DOUT_BUSY_R,

    input        cfg_done,           // M_DONE_R
```

They are connected to the LEDs of the LED array next to the control FPGA:

```
    assign led[0] = ~cnt[21];
    assign led[1] = ~resetn;
    assign led[2] = ( lbus\_rdy & cfg\_done );
    assign led[3] = ( cfg\_initn | cfg\_progn | cfg\_done | cfg\_rdwrn | cfg\_busy);
    assign led[4] = cfg\_din;
    assign led[5] = cfg\_mosi;
    assign led[6] = cfg\_fcsb;
    assign led[7] = cfg\_cclk;
    assign led[8] = C\_DIPSW[0];     // ON : SASEBO-GII software Full compati mode
    assign led[9] = cnt[21];
```

Internal components of the default control FPGA design

In this section we review the main components of the control FPGA: the clock generator, the USB interface, the interface with the main FPGA and the two communication FIFOs.

The clock generator

It generates two clocks of a different frequency desired by the designer via the input clock of 48 MHz.

```
  // Clock genarater
  clk_generater clk_gen
  (
    .rstnin( ext_rstn ), .clkin( clkin ),
    .frequency_sel( C_DIPSW[3:1] ),
    .rstnout( resetn ), .clk( clk ), .usbclk( usb_clk )
  );
```

This component receive an external rst, the clock input of the FPGA, the frequency that we want to get (based on the bits 3:1 of the DIP switch of the FPGA) and generates two new clocks, clk and usb_clk.
By default, clk is the system clock, clocked at 1.5 MHz. On the other hand usb_clk is based on the USB interface and is clocked at 24 MHz. The latter feeds the FT2232H chip equipped on the FPGA. On the other hand, clk feeds the local bus interface and the main FPGA (cipher_fpga).

The USB interface (ft2232h_usbif) and the TX, RX FIFO

This component receives by default a lock of 24 MHz and is connected to to the FTDI signals and to the two FIFOs: *rx_fifo* and *tx_fifo*.

```
module ft2232h_usbif(
  input        RSTn,            // Power on reset
  input        CLK,             // USB interface clock
  // USB controller(FT2232H) Interface
  input  [7:0] USB_DIN,          // USB data input bus
  output [7:0] USB_DOUT,         // USB data output  bus
  output       USB_RDn,          // USB read enable. Active low
  output       USB_WRn,          // USB write enable. Active low
  input        USB_RXFn,         // USB rx flag. High: Not ready  Low: ready
  input        USB_TXEn,         // USB tx enable. High: TX disable  Low: TX
  enable
  output       USB_DEN,          // USB tx data enable

  // Transmitt/Receive port
  input  [7:0] TXD,              // Transmit data input
  output [7:0] RXD,              // Receive data output
  output       TXD_RE,           // Read enable
  output       RXD_WE,           // Write enable
  input        TX_RDY,           // Transmit data ready
  input        RX_BUSY           // Receive data busy
);
```

Both FIFOs have the following inputs: two clock inputs for writing and reading, rst, one data input (D) and an output signal (Q), write and read enable signals (WE, RE) and two flags to know if the FIFO is full or empty.
By default, the input entry of data of the ft2232h_usbif comes from the port A of the FTDI chip tourgh USB_DIN and outputs go through the USB_DOUT port. On the other hand, this component interfaces with the FIFOs:

- RXD: rxdata from the receive data FIFO
- RXD_WE: rx_fifo_write
- RX_BUSY: tx_fifo_busy
- TXD: tx_data from the transmission FIFO
- TXD_RE: it reads the transmission FIFO
- TX_RDY: when the TX FIFO is not empty

Hence, this component reads the USB Transmit Data FIFO (where we write form the PC) and writes on the USB Receive Data FIFO. So this component, ft2232h_usbif reads data from the USB port (FTDI based) and writes in the internal reception FIFO (RX). It also reads the transmission FIFO (TX) and it sends it to the USB port.

The interface with the main FPGA (cipher_if)

This component controls the local bus that is used to communicate the control FPGA with the main FPGA. It reads the reception FIFO (RX) that the component ft2232h_usbif writes and write into the transmission FIFO.
In conclusion, the control FPGA only moves data between the USB port and the control FPGA (via ft2232h_usbif) and between the main FPGA and the control FPGA (cipher_if).

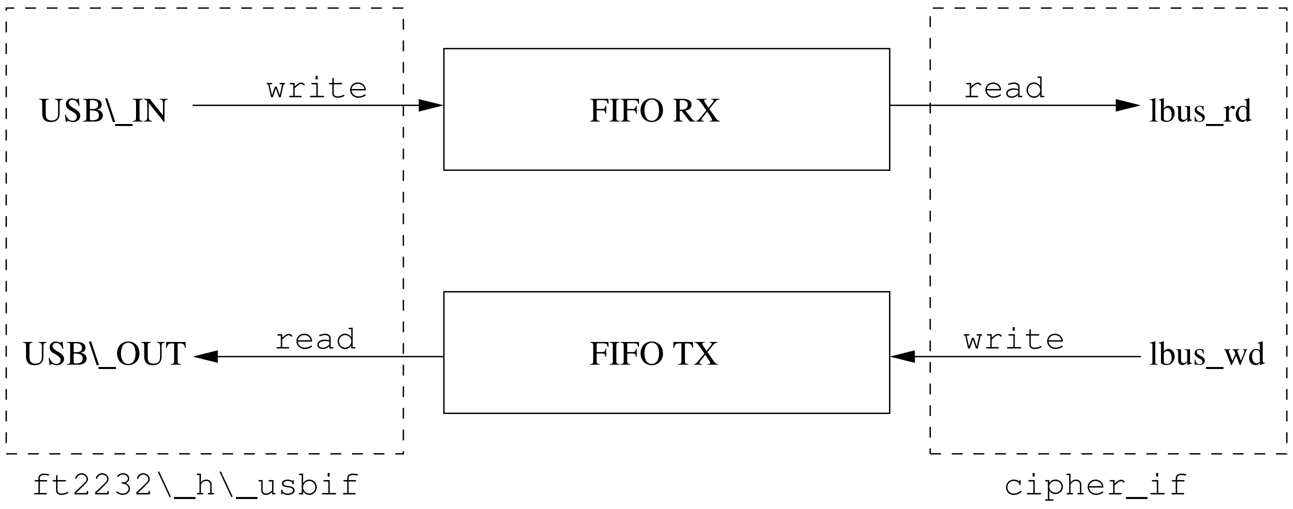


Figure 2: Transmission and reception FIFOs

The main FPGA (AES-128)

This FPGA consist of the design that is attacked together with several interfaces. The first of this interface or set of ports is the counterpart of the control FPGA that enables the communication via a local bus:

```
// Host interface
input         lbus_rstn,    // Reset from Control FPGA
input         lbus_clk,     // Clock from Control FPGA

output        lbus_rdy,     // Device ready
input   [7:0] lbus_wd,      // Local bus data input
input         lbus_we,      // Data write enable
output        lbus_ful,     // Data write ready low

output        lbus_aful,    // Data near write end
output  [7:0] lbus_rd,      // Data output
input         lbus_re,      // Data read enable
output        lbus_emp,     // Data read ready low

output        lbus_aemp,    // Data near read end
output        TRGOUTn,      // AES start trigger (SAKURA-G Only)
```

The next two groups of ports have to do with the FTDI interface. The first one, it is connected to the control FPGA, that reroutes the commands FTDI sent to the port B of the FPGA (only if it is selected via the DPI switch).

```
// FTDI USB interface portB (SAKURA-G Only)
// Control FPGA side
output        PORT_B_RXF,
output        PORT_B_TXE,
input         PORT_B_RD,
input         PORT_B_WR,
input   [7:0] PORT_B_DIN,
output  [7:0] PORT_B_DOUT,
input         PORT_B_OEn
```

Besides, there is a set of ports connected to the chip FTDI directly, corresponding to the port B of the chip:

```
  // FTDI USB interface portB (SAKURA-G Only)
  // FTDI side
  input         FTDI_BCBUS0_RXF_B,
  input         FTDI_BCBUS1_TXE_B,
  output        FTDI_BCBUS2_RD_B,
  output        FTDI_BCBUS3_WR_B,
  inout   [7:0] FTDI_BDBUS_D,
```

The main components of this project are:

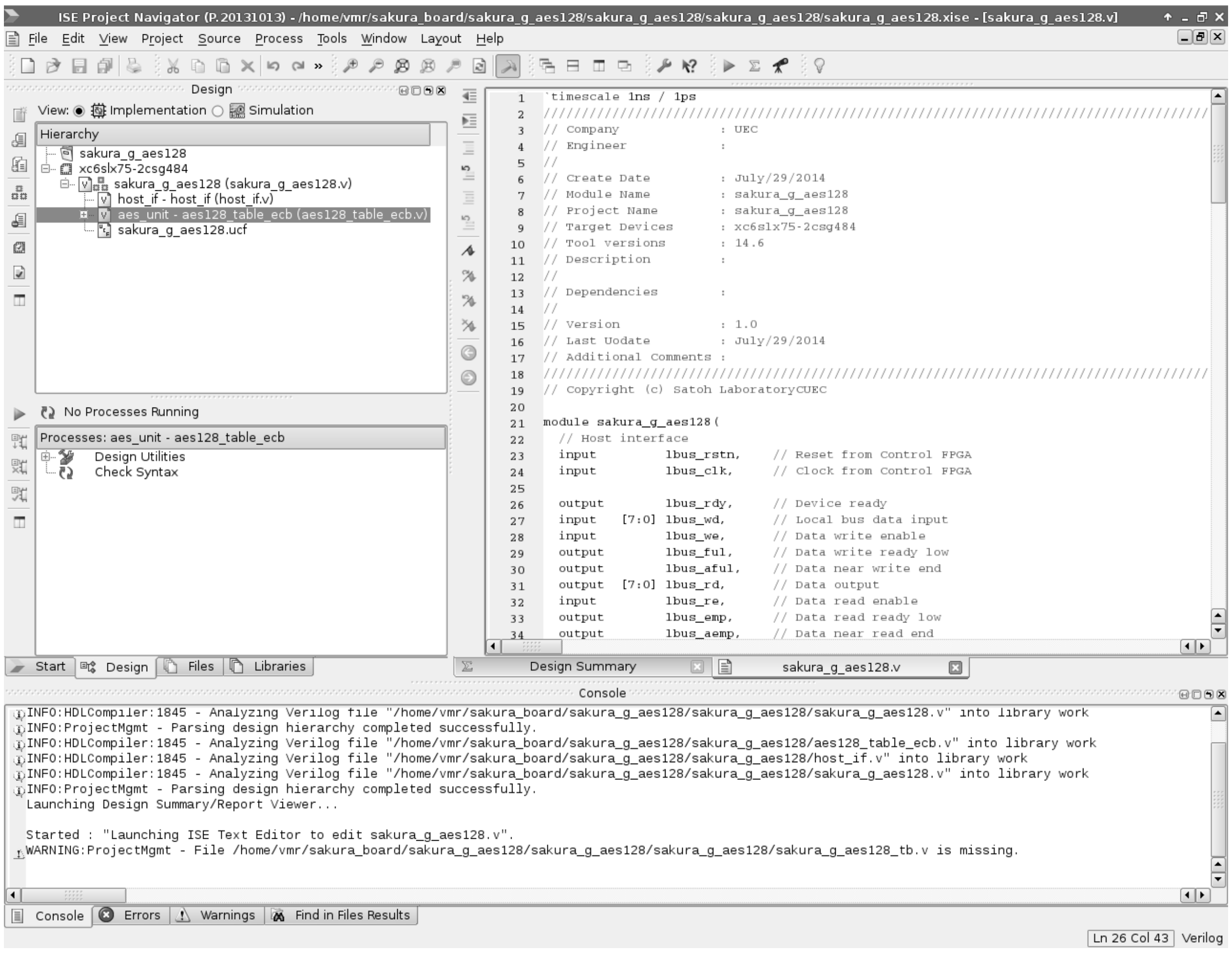
- host_if: this is the interface that controls the code that is being analyzed. It is mainly an FSM.
- AES128_table_ecb aes_wait: this is the implementation AES-128 by default in the Sakura.

In order to modify the design we want to analyze in the Sakura, first we study how it works the implementation of the AES actually works. It has the following ports:

```
  input               resetn,       // Async reset.
  input               clock,        // clock.

  input               enc_dec,      // Encrypt/Decrypt select. 0:Encrypt
//1:Decrypt
  input               key_exp,      // Round Key Expansion
  input               start,        // Encrypt or Decrypt Start
  output reg          key_val,      // Round Key valid
  output reg          text_val,     // Cipher Text or Inverse Cipher Text
//  valid
  input      [127:0]  key_in,       // Key input
  input      [127:0]  text_in,      // Cipher Text or Inverse Cipher Text
  input
  output     [127:0]  text_out,     // Cipher Text or Inverse Cipher Text
  output
  output reg          busy          // AES unit Busy
```

We see that there is several control signals:

- key_exp: start key expansion
- enc_dec: it selects encryption or decryption
- start: start encryption/decryption
- key_val: key_exp has finished
- text_val: encrption/decryption finished
- busy: the ciphertext/plaintext is not ready yet. During key expansion or encryption/decryption is equal to '1'.


ISE Project Navigator (P.20131013) - /home/vmr/sakura_board/sakura_g_aes128/sakura_g_aes128/sakura_g_aes128/sakura_g_aes128.xise - [sakura_g_aes128.v]
File Edit View Project Source Process Tools Window Layout Help
Design
View: Implementation Simulation
Hierarchy
sakura_g_aes128
xc6slx75-2csg484
sakura_g_aes128 (sakura_g_aes128.v)
host_if - host_if (host_if.v)
aes_unit - aes128_table_ecb (aes128_table_ecb.v)
sakura_g_aes128.ucf
No Processes Running
Processes: aes_unit - aes128_table_ecb
Design Utilities
Check Syntax
Start Design Files Libraries
Design Summary
sakura_g_aes128.v
Console
Console Errors Warnings Find in Files Results
Ln 26 Col 43 Verilog

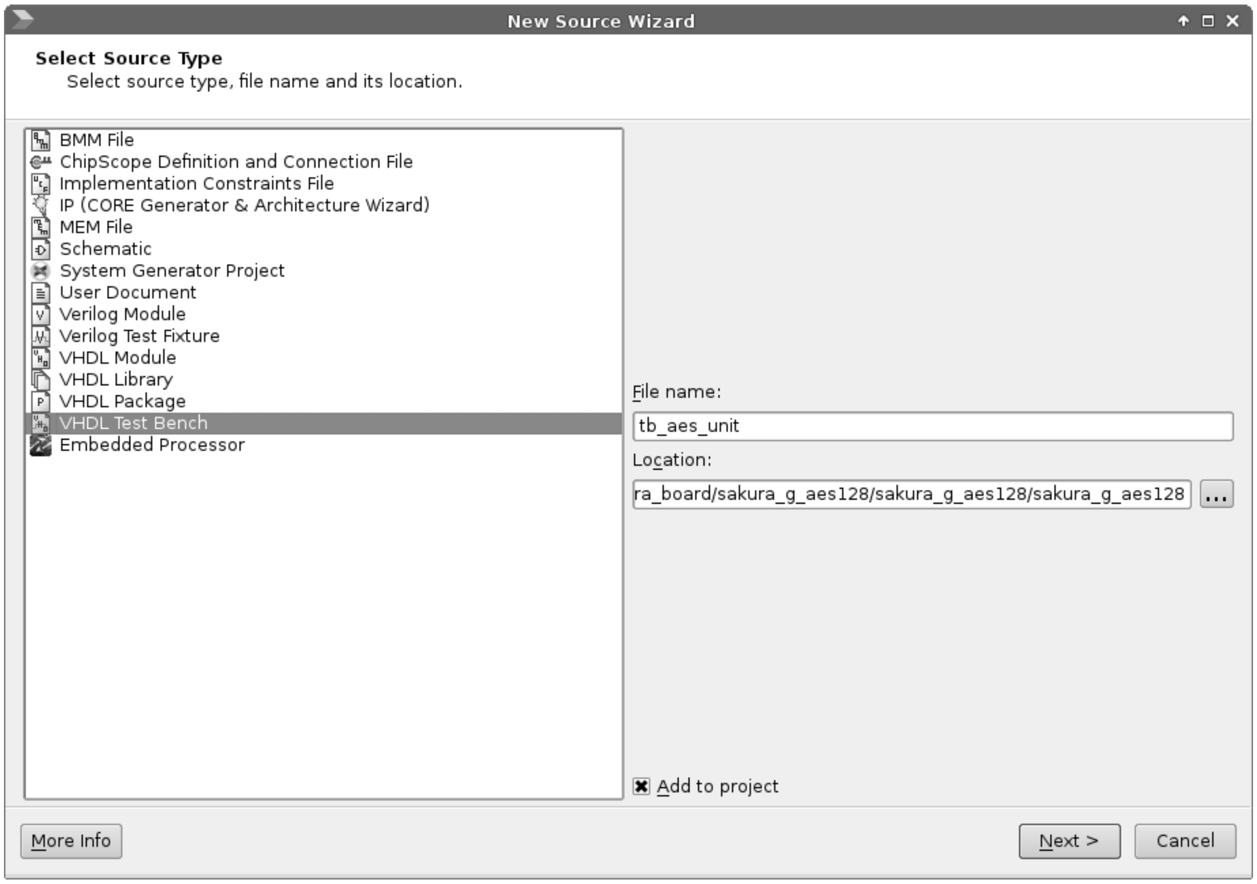
```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////
// Company              : UEC
// Engineer             :
//
// Create Date          : July/29/2014
// Module Name          : sakura_g_aes128
// Project Name         : sakura_g_aes128
// Target Devices       : xc6slx75-2csg484
// Tool versions        : 14.6
// Description          :
//
// Dependencies         :
//
// Version              : 1.0
// Last Uodate          : July/29/2014
// Additional Comments :
////////////////////////////////////////////////////////////////////////////////////
// Copyright (c) Satoh LaboratoryCUEC

module sakura_g_aes128(
  // Host interface
  input         lbus_rstn,    // Reset from Control FPGA
  input         lbus_clk,     // Clock from Control FPGA

  output        lbus_rdy,     // Device ready
  input   [7:0] lbus_wd,      // Local bus data input
  input         lbus_we,      // Data write enable
  output        lbus_ful,     // Data write ready low
  output        lbus_aful,    // Data near write end
  output  [7:0] lbus_rd,      // Data output
  input         lbus_re,      // Data read enable
  output        lbus_emp,     // Data read ready low
  output        lbus_aemp,    // Data near read end
```

```
INFO:HDLCompiler:1845 - Analyzing Verilog file "/home/vmr/sakura_board/sakura_g_aes128/sakura_g_aes128/sakura_g_aes128.v" into library work
INFO:ProjectMgmt - Parsing design hierarchy completed successfully.
INFO:HDLCompiler:1845 - Analyzing Verilog file "/home/vmr/sakura_board/sakura_g_aes128/sakura_g_aes128/aes128_table_ecb.v" into library work
INFO:HDLCompiler:1845 - Analyzing Verilog file "/home/vmr/sakura_board/sakura_g_aes128/sakura_g_aes128/host_if.v" into library work
INFO:HDLCompiler:1845 - Analyzing Verilog file "/home/vmr/sakura_board/sakura_g_aes128/sakura_g_aes128/sakura_g_aes128.v" into library work
INFO:ProjectMgmt - Parsing design hierarchy completed successfully.
Launching Design Summary/Report Viewer...

Started : "Launching ISE Text Editor to edit sakura_g_aes128.v".
WARNING:ProjectMgmt - File /home/vmr/sakura_board/sakura_g_aes128/sakura_g_aes128/sakura_g_aes128/sakura_g_aes128_tb.v is missing.
```

Figure 3: Open the main FPGA project in Xilinx ISE

What we need in our design a set of signals similar. In order to understand better how this implementation of the AES-128 algorithm works, we will create a test bench and we will simulate it.
Then, we will create the test bench: *Right-click* in one of the archives, *New source...*, VHDL Test Bench, *tb_aes_unit*, and select the aes 128 implementation.
It can be that you will select Modelsim SE for Verilog in the project properties and then Compile the simulation libraries by running the *Compile HDL Simulation Libraries*.
We will create a small test bench for encryption and decryption:

```
-- Stimulus process
stim_proc: process
begin
resetn <= ’0’;
wait for clock_period*2 + clock_period/2;

resetn <= ’1’;
enc_dec <= ’1’;
start <= ’1’;
wait for clock_period;
```



File name: tb_aes_unit

Location: ra_board/sakura_g_aes128/sakura_g_aes128/sakura_g_aes128

[✓ Add to project]

Figure 4: Create a new testbench

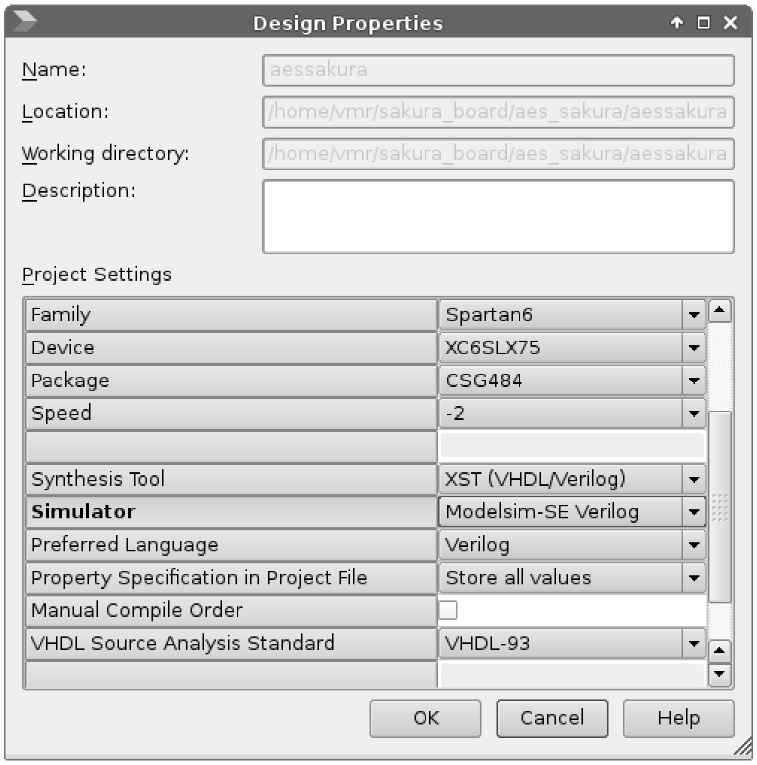
```
   start <= '0';

   wait;
   end process;
```

From the waveform, we see that we need a design that starts the computation after an inverted reset and when start is only '1', during one cycle. When the key schedule finishes, key_val is '1' and when the encryption/decryption finishes, text_val is '1' only during 1 cycle. Finally, we will need a bus signal. This signal, is '1' from start = '1' till text_val = '1'. This signal will be used later for triggering the oscilloscope when obtaining power traces, so we will obtain a power trace during the computation of the target design. Finally, this implementation of AES corresponds to an iterative architecture of the 128 fashion operating in ECB mode.

host_if

This component reads and writes the local bus between the control and the main FPGA. This is done via the following ports:

```
  output         DEVRDY,   // Device ready
  output         RRDYn,    // Read data empty
```



Name:	aessakura
Location:	/home/vmr/sakura_board/aes_sakura/aessakura
Working directory:	/home/vmr/sakura_board/aes_sakura/aessakura
Description:	

Project Settings

Family	Spartan6
Device	XC6SLX75
Package	CSG484
Speed	-2
Synthesis Tool	XST (VHDL/Verilog)
Simulator	Modelsim-SE Verilog
Preferred Language	Verilog
Property Specification in Project File	Store all values
Manual Compile Order	☐
VHDL Source Analysis Standard	VHDL-93

Figure 5: Set the simulator value to ModelSim-SE Verilog

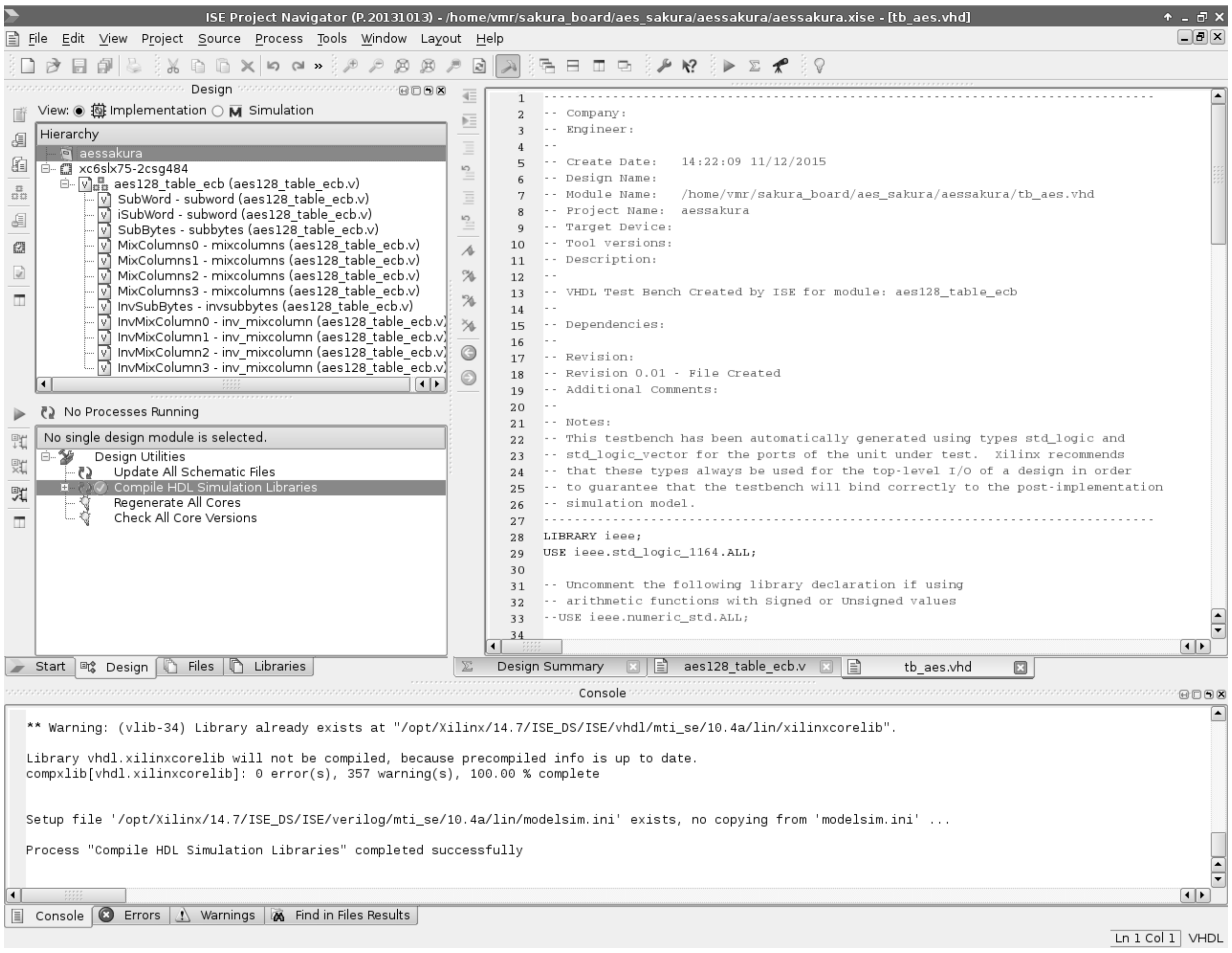
```
  output          WRDYn,    // Write buffer almost full
  input           HRE,      // Host read enable
  input           HWE,      // Host write enable
  input     [7:0] HDIN,     // Host data input
  output    [7:0] HDOUT,    // Host data output
```

connected to:

- lbus_rdy: device ready
- lbus_emp: data read ready
- lbus_rul: data write ready
- lbus_re: data read enable
- lbus_we: data write enable
- lbus_wd: local bus data input, 1 byte
- lbus_rd: local bus data output, 1 byte

Moroever, this component controls the AES implementation through the values that are written in bus. Hence, the following values are sent to the AES design:

- RSTOUTn: internal reset set to the design, **be careful when reseting your component via this interface since it considers that a rst signal is always inverted.**
- ENCn_DEC: encryption or decryption according to the value


ISE Project Navigator (P.20131013) - /home/vmr/sakura_board/aes_sakura/aessakura/aessakura.xise - [tb_aes.vhd]
File Edit View Project Source Process Tools Window Layout Help
Design
View: Implementation Simulation
Hierarchy
aessakura
xc6slx75-2csg484
aes128_table_ecb (aes128_table_ecb.v)
SubWord - subword (aes128_table_ecb.v)
iSubWord - subword (aes128_table_ecb.v)
SubBytes - subbytes (aes128_table_ecb.v)
MixColumns0 - mixcolumns (aes128_table_ecb.v)
MixColumns1 - mixcolumns (aes128_table_ecb.v)
MixColumns2 - mixcolumns (aes128_table_ecb.v)
MixColumns3 - mixcolumns (aes128_table_ecb.v)
InvSubBytes - invsubbytes (aes128_table_ecb.v)
InvMixColumn0 - inv_mixcolumn (aes128_table_ecb.v)
InvMixColumn1 - inv_mixcolumn (aes128_table_ecb.v)
InvMixColumn2 - inv_mixcolumn (aes128_table_ecb.v)
InvMixColumn3 - inv_mixcolumn (aes128_table_ecb.v)
No Processes Running
No single design module is selected.
Design Utilities
Update All Schematic Files
Compile HDL Simulation Libraries
Regenerate All Cores
Check All Core Versions
Start Design Files Libraries
Design Summary aes128_table_ecb.v tb_aes.vhd
Console
Console Errors Warnings Find in Files Results
Ln 1 Col 1 VHDL

```
--------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:   14:22:09 11/12/2015
-- Design Name:
-- Module Name:   /home/vmr/sakura_board/aes_sakura/aessakura/tb_aes.vhd
-- Project Name:  aessakura
-- Target Device:
-- Tool versions:
-- Description:
--
-- VHDL Test Bench Created by ISE for module: aes128_table_ecb
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-- Notes:
-- This testbench has been automatically generated using types std_logic and
-- std_logic_vector for the ports of the unit under test.  Xilinx recommends
-- that these types always be used for the top-level I/O of a design in order
-- to guarantee that the testbench will bind correctly to the post-implementation
-- simulation model.
--------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;
```

```
** Warning: (vlib-34) Library already exists at "/opt/Xilinx/14.7/ISE_DS/ISE/vhdl/mti_se/10.4a/lin/xilinxcorelib".

Library vhdl.xilinxcorelib will not be compiled, because precompiled info is up to date.
compxlib[vhdl.xilinxcorelib]: 0 error(s), 357 warning(s), 100.00 % complete


Setup file '/opt/Xilinx/14.7/ISE_DS/ISE/verilog/mti_se/10.4a/lin/modelsim.ini' exists, no copying from 'modelsim.ini' ...

Process "Compile HDL Simulation Libraries" completed successfully
```

Figure 6: Compile the simulation libraries

- KEY_GEN: Start key schedule
- DATA_EN: start encryption/decryption
- KVAL: key schedule ready
- TVAL: plaintext/ciphertext ready
- KEY_OUT: key
- DATA_OUT: ciphertext/plaintext
- RESULT: ciphertext/plaintext

This component has two registers of 16 bits, *addr_reg* and *data_reg*. In the prior tutorial what we did was to write 2 bytes of address (MSB, LSB) and two of data. What the FSM of the Sakura does is to do transitions of 4 states each for reading:

1. READ1: read MSB addr
2. READ2: read LSB addr

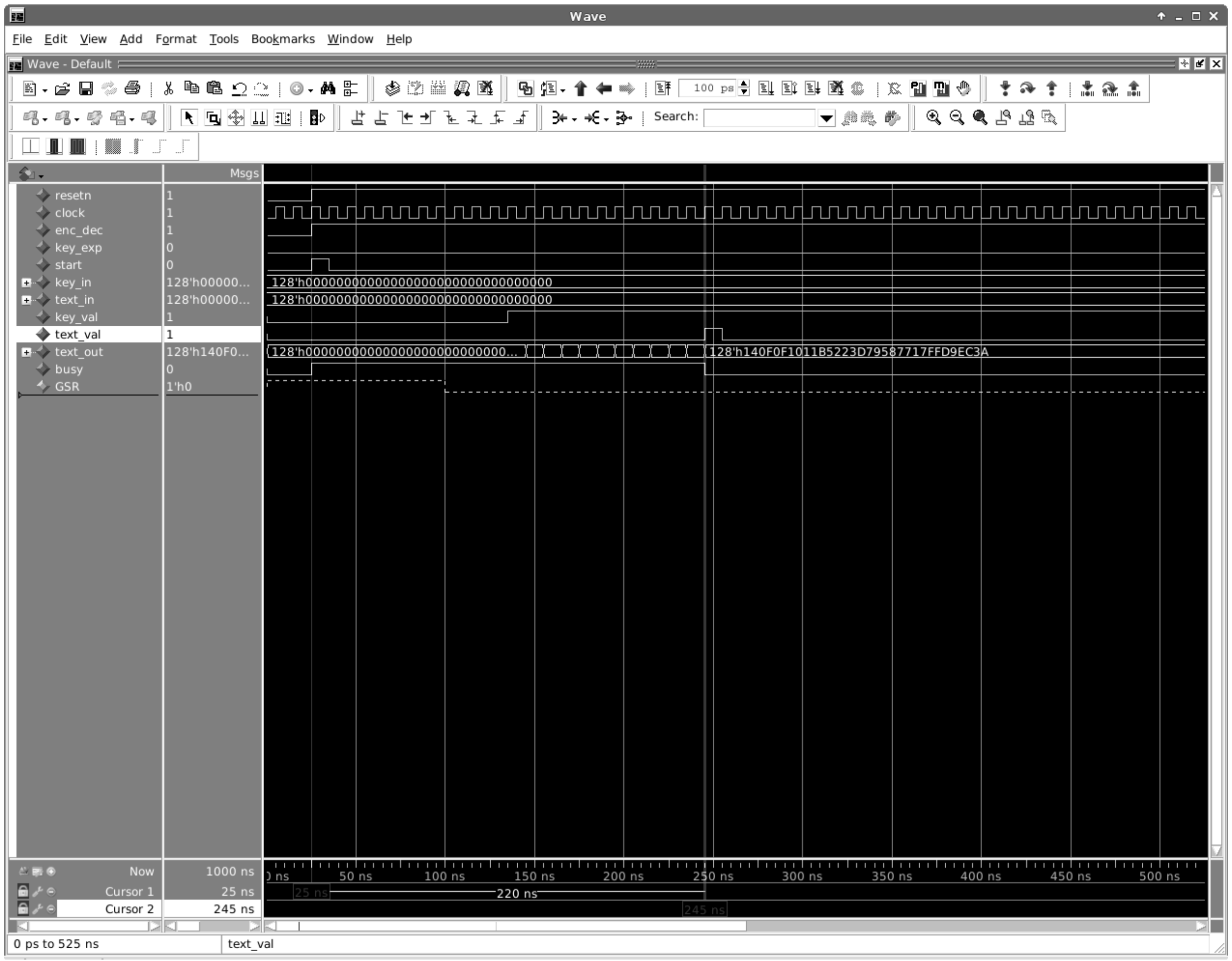

Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
Search:
Msgs
resetn 1
clock 1
enc_dec 1
key_exp 0
start 0
key_in 128'h00000...
text_in 128'h00000...
key_val 1
text_val 1
text_out 128'h140F0...
busy 0
GSR 1'h0
128'h00000000000000000000000000000000
128'h00000000000000000000000000000000
128'h0000000000000000000000000000...
128'h140F0F1011B5223D79587717FFD9EC3A
Now 1000 ns
Cursor 1 25 ns
Cursor 2 245 ns
0 ns 50 ns 100 ns 150 ns 200 ns 250 ns 300 ns 350 ns 400 ns 450 ns 500 ns
220 ns
0 ps to 525 ns
text_val

Figure 7: Waveform with the control signals of the AES implementation

3. READ3: read MSB data
4. READ4: read LSB data

and writing:

1. WRITE1: write MSB addr
2. WRITE2: write LSB addr
3. WRITE3 write MSB data
4. WRITE4: write LSB data

During READ1 and WRITE1, 1 byte is taken from *lbus_din_reg* to *addr_reg[15:8]* and during READ2 and WRITE2 we do the same to *addr_reg[7:0]*. In READ3/WRITE3 and READ4/WRITE4 we do the same with *data_reg[15:8]* and *data_reg[7:0]*.
This design also has two registers of 128 bits called *key_reg* and *din_reg*. There are written the data that we send via Python in the prior tutorial. However, if we need more space in the key or Plaintext/ciphertext register we can use more addresses, before or after *0x0100*/*0x010e* in *key_reg* and increase the size.

When the ciphertext or plaintext is ready, it is written in the RESULT whose result can be also increased. For reading, the interface is done so only some parameters can be accessed:

```
  always @( addr_reg or rst or enc_dec or key_gen or data_ena or KVAL or
TVAL or RESULT ) begin
    case( addr_reg )
      16'h0002: dout_mux = { 13'h0000, rst, key_gen, data_ena };
      16'h000c: dout_mux = { KVAL, TVAL, enc_dec };
      16'h0180: dout_mux = RESULT[127:112];
      16'h0182: dout_mux = RESULT[111:96];
      16'h0184: dout_mux = RESULT[95:80];
      16'h0186: dout_mux = RESULT[79:64];
      16'h0188: dout_mux = RESULT[63:48];
      16'h018a: dout_mux = RESULT[47:32];
      16'h018c: dout_mux = RESULT[31:16];
      16'h018e: dout_mux = RESULT[15:0];
      16'hfffc: dout_mux = 16'h4522;
      default: dout_mux = 16'h0000;
    endcase
  end
```

Hence, is is easy to see that the registers for the key and plaintext can not be read after they are written by default.
Finally, according to the content of the control register (*0x0002*), respective signals are sent to the AES design. Further, key_reg and data_reg are connected to AES too.

Triggering

During the acquisition of power traces, we will indicate the oscilloscope in which moment we want to capture. This can be done via three ways using the Sakura:

1. At the beginning of the encryption process
2. At the end of the encryption process.
3. During the encryption process.

The default design of the main FPGA outputs via the PINs 1,2,3 of the CN3 connector the signals start, text_val and busy respectively. Moreover, it sends the start signal via the SMA connector J4.

Exercises

1. According to your target design, increase the size of the key and plaintext registers.
2. Connect the Sakura board to the oscilloscope. Run the script you created in the prior tutorial in order to generate triggers that can be visualized into the oscilloscope. You might want to select *single* in the triggering menu of the oscilloscope each time.
3. Replace the AES design with your target design[3]. This process is faster. However, remember that you will lose the FPGA configuration each time you turn it off.

[3]Where testing the code on the Sakura board you do not need to program the SPI memory all the time. You can instead load the generated *.bit* file on the FPGA. In Impact, say *NO* to program the SPI of the flash FPGA and select instead the *.bit* file. Then, do right-click on the FPGA and select *Program*.

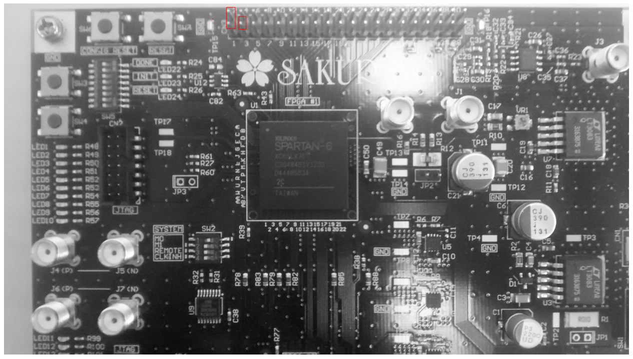


Figure 8: PINs used for triggering

(a) Create ports in your top block for the start, busy and done signals.

(b) Create an FSM in your design that accepts a *start* signal that is '1' during one cycle and start the encryption/decryption process. At the same time, it generates a *busy* signal as described in this tutorial. You can rely on the FSM below as an example[4]:

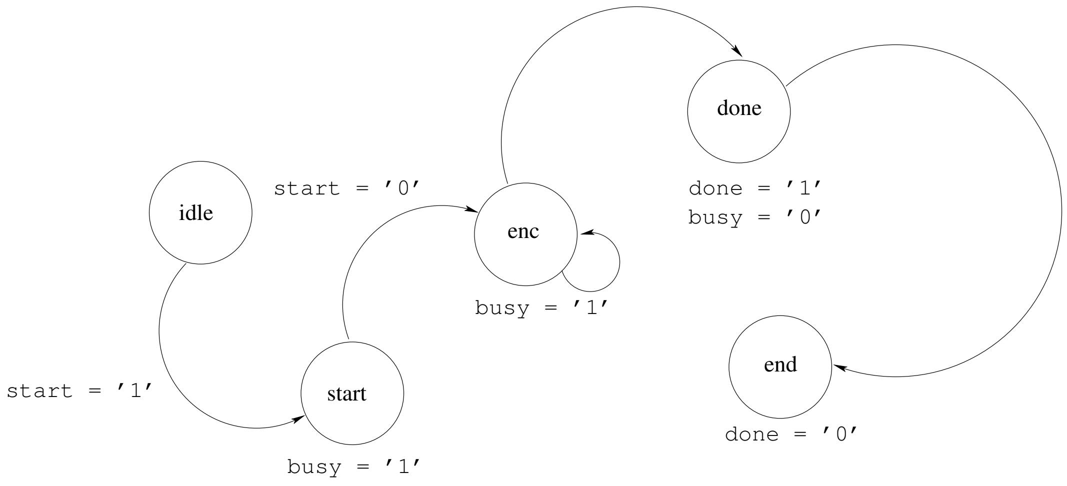


Figure 9: Transmission and reception FIFOs

(c) Adapt the script you wrote in the prior tutorial to send and receive plaintexts and ciphertexts from the Sakura using your target design.

[4]You can assume that the key schedule, if needed, is performed in *enc* or you can add another state, *key_gen*, prior to *enc* that triggers *busy* to '1'