Lab 3.1 in SMD098 Jonas Thor
November 2001

# Interfacing a PS/2 Keyboard

# 1 Introduction

In this lab you will interface a PS/2 keyboard (standard PC keyboard) with the XSV board. The lab is divided in two parts:

1. The first task is to receive scan codes from the keyboard and display the scan codes in hex format on the XSV digit LEDs.
2. The second task is to transmit data to the keyboard when the Caps Lock key is pressed. When a specific data stream is sent the caps lock LED is lit on the keyboard.

The first part of this lab is **compulsory** (Swedish: obligatorisk). The second part is **voluntary** but if you manage to get second part working you will get a 3 point bonus on the final exam. The bonus may help you raise your grade or even to pass the exam. The final exam will have a maximum of 30 points. Note that you should not expect much help for the second part of the lab.

The PS/2 interface is used to connect other devices than keyboards. In this text we refer to the “PS/2 Keyboard Interface”, but the PS/2 interface is a *general* serial interface and the keyboard is just a device that uses the PS/2 interface.

# 2 The PS/2 keyboard interface

Links to descriptions of the PS/2 keyboard interface, written by Adam Chapweske, can be found at the lab home page. A short description of the PS/2 keyboard interface follows.

## 2.1 Electrical interface

The PS/2 interface is a bit serial interface with two signals Data and Clock. Both signals are bi-directional and logic 1 is electrically represented by 5 V and logic 0 is represented by 0 V (digital ground). Whenever the Data and Clock line is not used, i.e. is idle, both the Data and Clock lines are left floating, that is the host and the device both set the outputs in high impedance. Externally, at the PCB, large (about 5 kΩ) pull-up resistors keep the idle lines at 5V (logic 1).

The FPGA/keyboard interface is shown in figure 1. When the FPGA “reads” the Data or Clock inputs both PS2Data_out and PS2Clk_out are kept low which puts the tri-state buffers in high impedance mode. When the FPGA "writes" a logic 0 on an output, the corresponding x_out (x = PS2Data or PS2Clk) signal is set high which pulls the line low. When “writing” logic 1 the FPGA simply sets the x_out signal low.

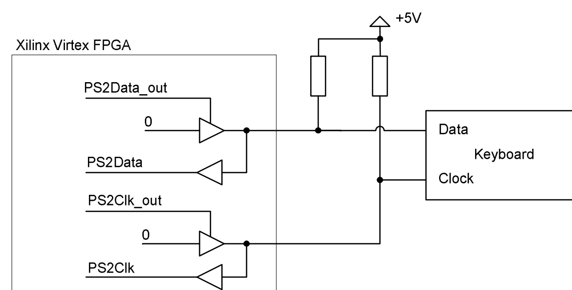



**Figure 1. FPGA/Keyboard Interface**

## 2.2 Protocol for receiving data from the keyboard

Data is received from the keyboard as illustrated in figure 2.

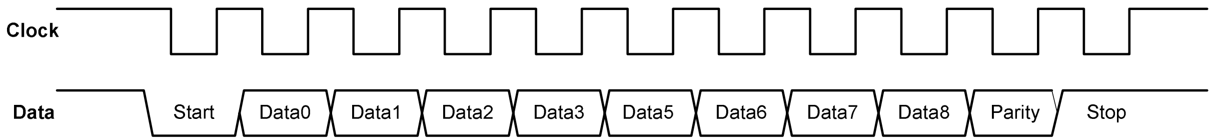



**Figure 2. PS/2 Protocol**

A transfer may be initiated by the keyboard if the Clock line is high. The host (FPGA in this case) may force the Clock low in order to prevent the keyboard from sending data – the host may *inhibit* communication. Note that the keyboard generates the clock. Data is valid at least 5 us (*t* in figure 3) before the falling edge of the clock. The clock period is (*T* in figure 3) is 60 to 100 us.

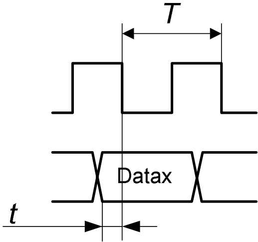



**Figure 3. PS/2 Timing**

Data is sent in bit serially. The first bit is always a start bit, logic 0. Then 8 bits are sent with the least significant bit first. The data is padded with a parity bit (odd parity). The parity bit is set if there is an even number of 1's in the data bits and reset (logic 0) if there is an odd number of 1's in the data bits. The number of 1's in the data bits plus the parity bit always add up to an odd number (odd parity.) This is used for error detection. A stop bit (logic 1) indicates the end of the data stream.

## 2.3 The keyboard scan-codes

The keyboard sends packets of data, *scan codes*, to the host indicating which key has been pressed. When a key is pressed or held down a *make code* is transmitted. When a key is released a *break code* is transmitted. Every key is assigned a unique make and break code so that the host can determine exactly what has happened.

There are three different scan code sets, but all PC keyboards use Scan Code Set 2. A sample of this scan code set is listed in table 1. Please refer to the lab homepage for the full scan code set.

| KEY | MAKE | BREAK |
|---|---|---|
| A | 1C | F0,1C |
| B | 32 | F0,32 |
| C | 21 | F0,21 |
| D | 23 | F0,23 |
| E | 24 | F0,24 |
| F | 2B | F0,2B |
| G | 34 | F0,34 |
| H | 33 | F0,33 |
| I | 43 | F0,43 |
| J | 3B | F0,3B |
| K | 42 | F0,42 |
| L | 4B | F0,4B |
| M | 3A | F0,3A |
| N | 31 | F0,31 |
| O | 44 | F0,44 |
| P | 4D | F0,4D |
| Q | 15 | F0,15 |
| R | 2D | F0,2D |
| S | 1B | F0,1B |
| T | 2C | F0,2C |
| U | 3C | F0,3C |
| V | 2A | F0,2A |
| W | 1D | F0,1D |
| X | 22 | F0,22 |
| Y | 35 | F0,35 |
| Z | 1A | F0,1A |

**Table 1. Scan Code Set 2 (sample)**

# 3 Task 1 – Displaying scan codes

Receive the scan codes from the keyboard and display the corresponding code in hexadecimal format on the XSV board digit LEDs. The LEDs should be updated at a rate of approximately 1 Hz

Following we will partition this problem into more manageable pieces. We will partition the design into a data path and a control path. A block diagram of the complete design is shown in figure 4.

**Figure 4. Block diagram**

All flip-flops in the design are clocked with a 20 MHz clock and the rising edge is the active edge. Four types of positive edge-triggered D-flip-flops are shown in figure 4.

| | |
|---|---|
| FD | No reset/set |
| FDC | Asynchronous reset |
| FDP | Asynchronous set |
| FDCE | Clock enable and asynchronous reset |

The `PS2Data` and `PS2Clk` signals are sampled with FDP flip-flops. The `PS2Data` is fed into a shift register. When 8 data bits (scan code) and one parity bit has been shifted into the shift register the scan code is written to a synchronous FIFO. When the FIFO is not empty the output is read in intervals of approximately 1 second. The FIFO output is connected to two identical ROMs. The ROMs decode the scan code data so that a character “0” – “F” is displayed on each digit LED.

A short description of each module in the data and control path follows.

## 3.1 Data path components

### 3.1.1 Shift register

The shift register is an 8 bit serial in parallel out (SIPO) shift register with a shift enable input.

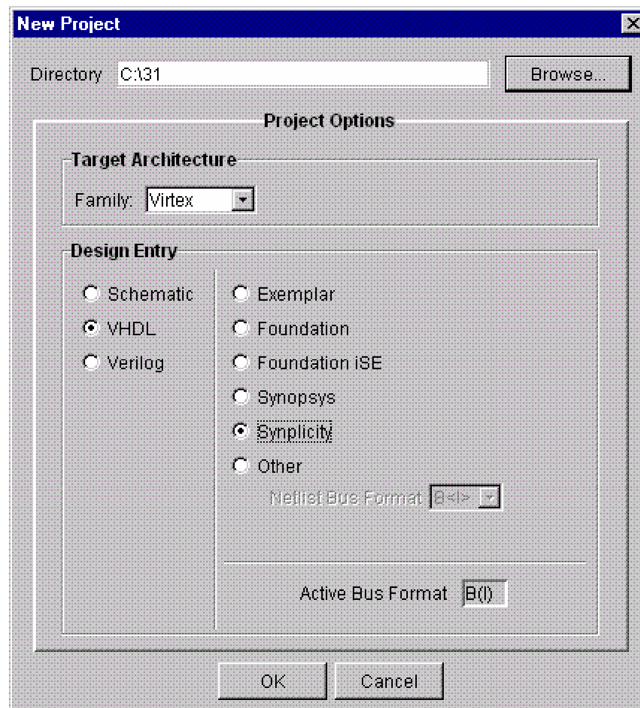
### 3.1.2 FIFO

The FIFO stores a maximum of 256 bytes. The data input and outputs are 8 bits wide. The FIFO is initialized synchronously by setting `sinit` high. When `wr_en` is set high data is written to the FIFO and when `rd_en` is set high data is read from the FIFO. Two flags,

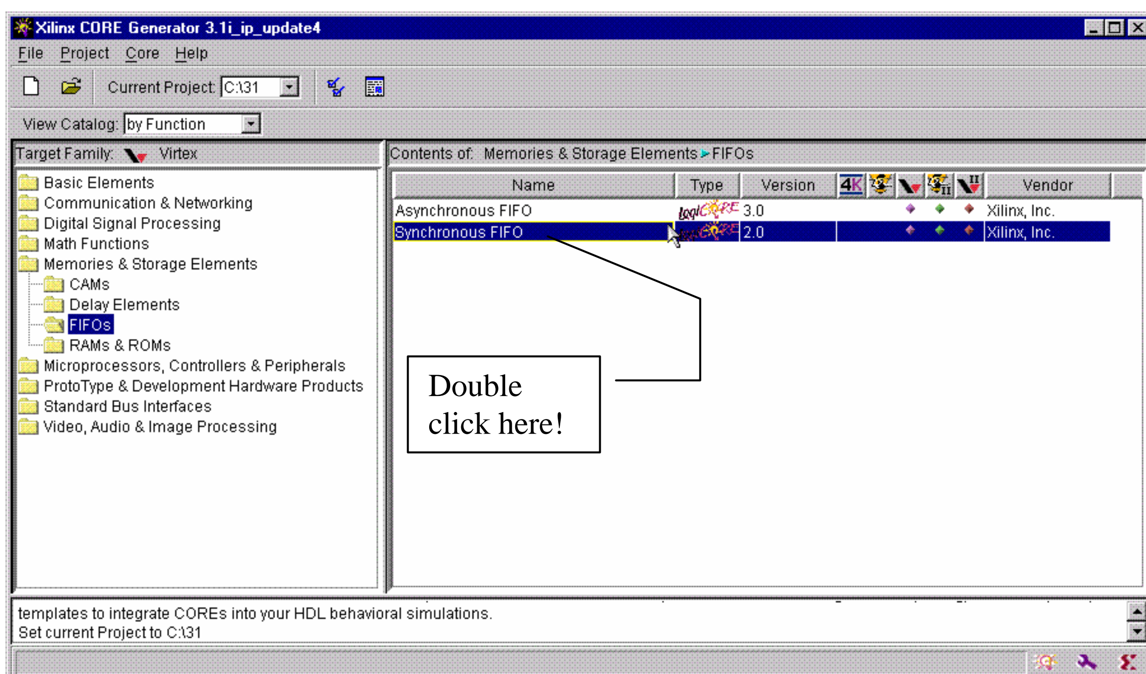
`empty` and `full` indicate the status of the FIFO. The FIFO will be created with the Xilinx Core Generator tool.

#### 3.1.2.1 Creating the synchronous FIFO with Xilinx Core Generator

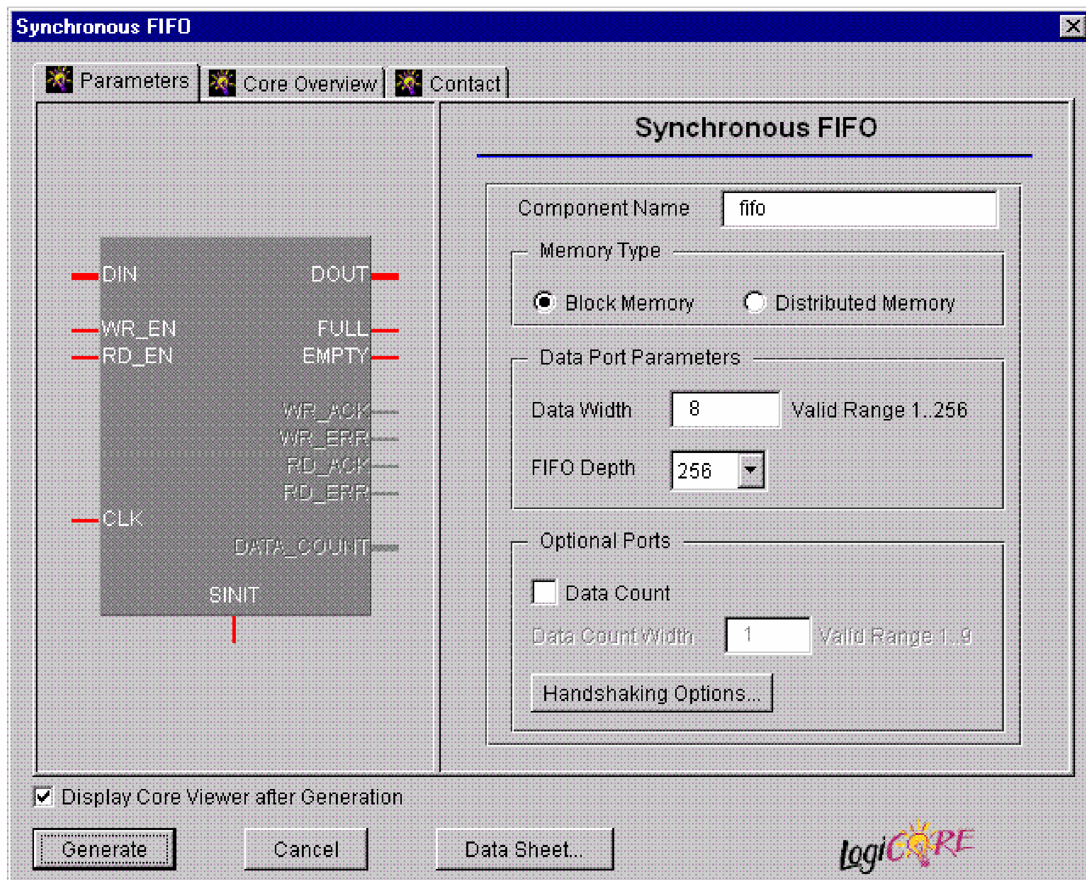
Start Core Generator with the command `coregen`. In the dialog box that appears chose to create a new project. The New Project dialog appears:




Select a directory where you want to place your FIFO files. Choose family "Virtex" and set the Design Entry options to "VHDL" and "Synplicity". Click OK to display the main window.



In the main window navigate to find the Synchronous FIFO and double click. The window below will be displayed:

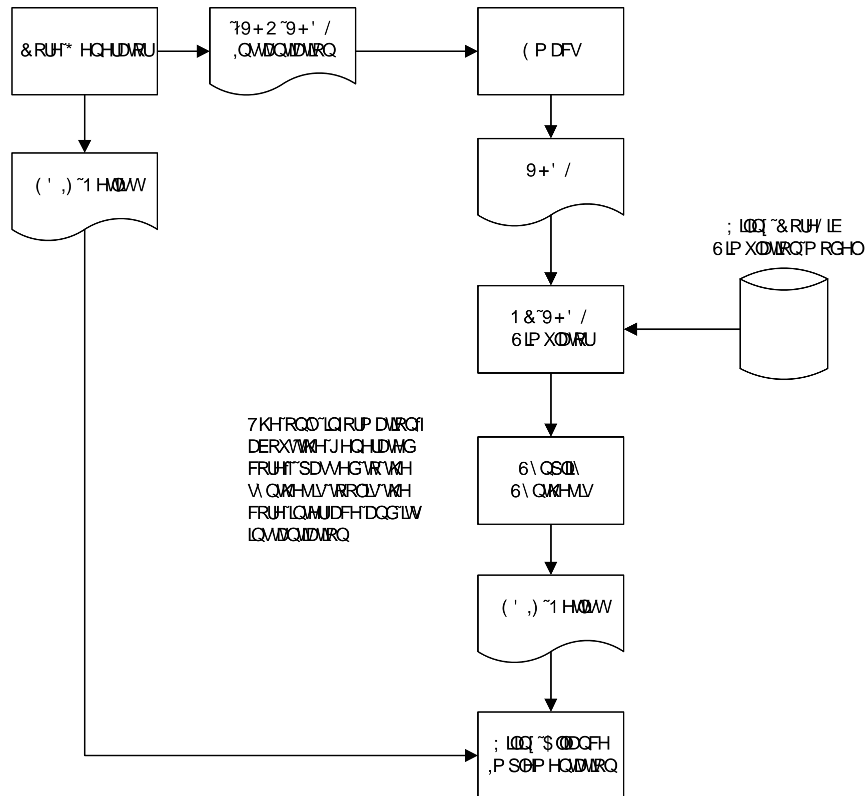



Complete the dialog as indicated in the figure above. Leave “Handshaking Options” to the default values. Click “Generate” to generate the FIFO core. Next click “Data Sheet” and read the specifications for the FIFO. Next click Cancel and exit the Core Generator.

Core Generator has now generated the following files of interest:

| | |
|---|---|
| `fifo.edn` | EDIF implementation netlist for the core. Describes how the core is to be implemented. Used as input to the Xilinx implementation tools. |
| `fifo.vho` | VHDL Template file. The components in this file can be used to instantiate a core. |

The figure below illustrates how the two files are used in the design flow. Note that no information except the core interface and its instantiation is passed to the synthesis tool. The synthesis tool will consider the FIFO to be a “black box”. The output from the synthesis tool is an EDIF net list where the core is instantiated. But this netlist contains no information about how the FIFO is implemented. Instead there will be two separate netlists that are input to the implementation tool. You may see it as the two netlists will be “merged” together in the implementation tool.



### 3.1.3 Parity checker

The parity checker is a combinational unit that outputs a ‘1’ if there is no parity error; else it outputs a ‘0’.

### 3.1.4 ROM

The ROM converts four bits of the scan code so that a digit “0” – “F” is displayed on the digit LEDs. You should implement them with function calls, similar to what was done in lab 1.

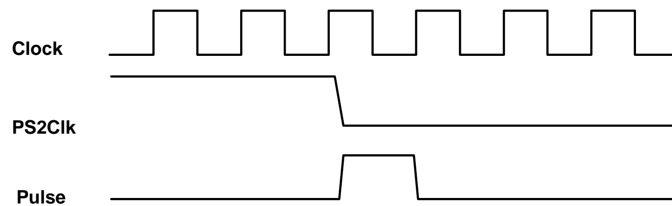
### 3.1.5 Input/output registers

Refer to figure 4 on how to implement the input/output register.

## 3.2 Control path components

### 3.2.1 “One-shot” component

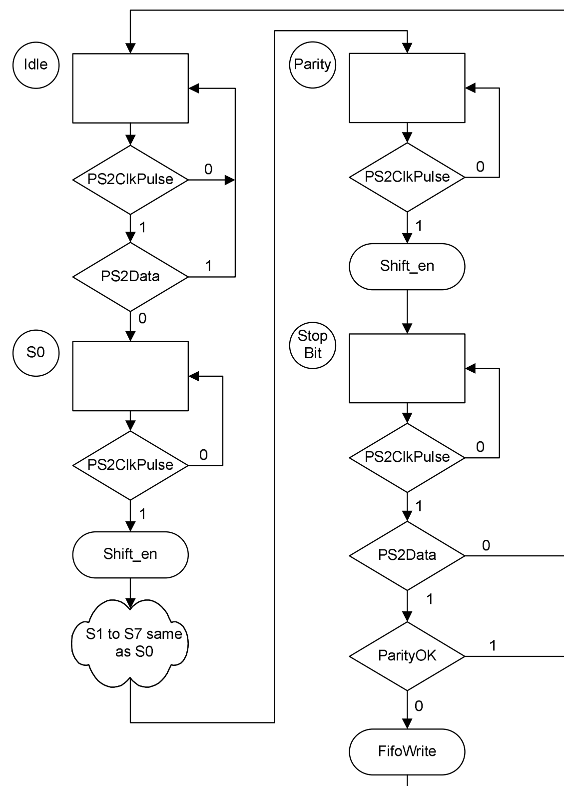
The one-shot component should output a pulse for one clock cycle when it detects that the PS2Clk has transitioned from high to low. See timing diagram below.




The pulsed output will be used to enable the shift register. As previously mentioned PS2Data is valid on the falling edge of PS2Clk. However we do not want enable the shift register every falling edge of PS2Clk. The pulse is passed through a Finite State Machine (FSM) that may suppress the pulse.

### 3.2.2 Receiver FSM

The receiver FSM controls that the PS/2 data is read correctly. An ASM chart for the FSM is shown below. When the FSM powers up or is reset it enters the Idle state. When PS2ClkPulse and PSData are set a start bit has been detected and the FSM enters the state SO. The FSM stays in this state until PS2ClkPulse is high again. When this happens the Shift_en is set high in the same clock cycle and the next clock cycle the FSM enters the state, S0. The Shift_en signal enables the shift register and shifts in PS/2 data bits. States S1 to S7 are identical to S0. The state Parity is also identical to S0. When the FSM enters the StopBit state the shift register contains the scan code and the parity bit. The FSM stays in the StopBit state until another pulse is detected on PS2ClkPulse. When the pulse is detected, a stop bit is detected and the parity is correct the FifoWrite signal is set and the FSM returns to the Idle state. If a stop bit is not detected or the parity is incorrect the FSM enters the Idle state without enabling the FifoWrite signal.



The FSM does not check if the FIFO is full before writing to it. Practically this means that some scan codes may be lost.

Since states `S0` to `S7` and `Parity` are identical the number of states can be reduced. For instance a counter could have been used to keep track of how many bits that have been shifted in to the shift register. You can choose how to implement the FSM – using “many” states or fewer states and a counter.

### 3.2.3 Timer

The timer controls the update rate of the digit LEDs and consequently at what rate the FIFO data is read. Design a timer that updates the digits LEDs at a rate of approximately 1 Hz. Note that you should not read from the FIFO while it is empty.

The output from the timer enables read from FIFO. The output from timer delayed by one clock cycle is used to enable the output registers for the digit LEDs. The FIFO data output is valid one clock cycle after the read signal is activated.

## 3.3 Putting it all together

You have been given a file `Virtex.vhd` which contains the top level entity for the FPGA design. The architecture is also found in this file but it is not complete. Your task is to design and connect all the components previously described. The component declarations are found in the architecture declarative region, so you know exactly the interface of the components. Create separate files for each component and name them `<entity_name>.vhd`.

To save some time: As an example, use the following procedure to create a new file and entity for the `ShiftReg` component:

1. Open the file `Virtex.vhd` in Emacs
2. Locate the `ShiftReg` component declaration and place the cursor in the port region of the declaration.
3. Choose VHDL → Port → Copy
4. Open a new file and name it `ShiftReg.vhd`
5. Choose VHDL → Port → Paste As Entity

The input/output registers and the ROM should be implemented in a process found at the end of the architecture. You may reuse some of the code from lab 1!

The FIFO needs special attention. At the end of the `Virtex.vhd` file you find a VHDL configuration:

```
-- synthesis translate_off
library XilinxCoreLib;

configuration Virtex_cfg of Virtex is
  for Keyboard
    for all : fifo use entity XilinxCoreLib.sync_fifo_v2_0(behavioral)
                  generic map(
                    c_write_data_width => 8,
                    c_memory_type      => 1,
                    c_wr_err_low       => 1,
                    c_read_data_width  => 8,
```

```
                        c_has_rd_ack       => 0,
                        c_wr_ack_low       => 1,
                        c_read_depth       => 256,
                        c_ports_differ     => 0,
                        c_has_wr_ack       => 0,
                        c_has_rd_err       => 0,
                        c_has_dcount       => 0,
                        c_has_wr_err       => 0,
                        c_rd_ack_low       => 1,
                        c_write_depth      => 256,
                        c_rd_err_low       => 1,
                        c_dcount_width     => 1,
                        c_enable_rlocs     => 0);
    end for;
  end for;
end Virtex_cfg;
-- synthesis translate_on
```

The code between the region enclosed by the comments `synthesis translate_off` and synthesis `translate_on` is ignored by the synthesis tool. The purpose of the configuration is to bind the `fifo` component to a simulation model (architecture) found in the library `XilinxCoreLib`. The configuration also passes generic parameters to the simulation model.

The configuration has been copied from the `.vho` file generated by the Core Generator. Note that you must generate the core since you are not provided with a netlist.

## 3.4 Simulation

Your `cds.lib` file should contain this:

```
INCLUDE /digcad/cds/ldv3.1/tools/inca/files/cds.lib
ASSIGN xilinxcorelib TMP ./corelib_tmp
DEFINE worklib ./worklib
```

You must create the directories `corelib_tmp` and `worklib`. The reason why you need the `corelib_tmp` directory is that you do not have write permissions to where the simulation models for the FIFO are stored.

You may use the following commands to compile your code:

```
ncvhdl -v93 smd098_pkg.vhd
ncvhdl -v93 OneShot.vhd
ncvhdl -v93 ParityChecker.vhd
ncvhdl -v93 ReceiverFSM.vhd
ncvhdl -v93 Ser2Par.vhd
ncvhdl -v93 ShiftReg.vhd
ncvhdl -v93 Timer.vhd
ncvhdl -v93 Virtex.vhd
ncvhdl -v93 Virtex_tb.vhd
```

Add the `-messages` and `-linedebug` switches if you need them.

When you elaborate the design it is not the test bench itself you should elaborate, instead it is the configuration for the test bench. To elaborate use:

```
ncelab -v93 worklib.virtex_tb_cfg
```

And it is the configuration you load into the simulator!

```
ncsim -gui worklib.virtex_tb_cfg
```

You have been provided with a test bench. The test bench only apply stimuli, it does not check if the result is correct! You should study the test bench and perhaps extend it.

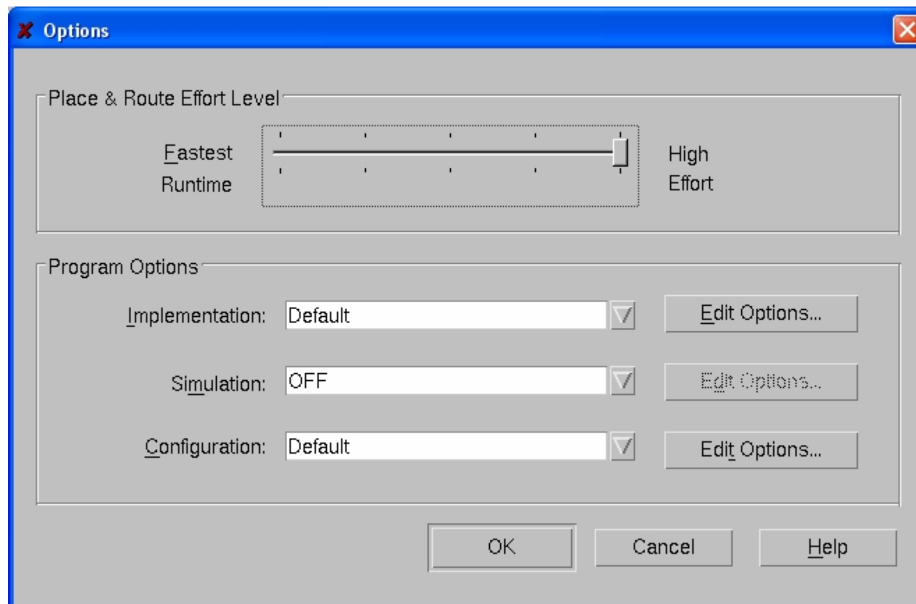
## 3.5 Synthesis

Use same the device as you have used in previous labs – Virtex XSV100, speed grade 4, package PQ240. Constrain the clock frequency to 20 MHz. You may ignore the input/output constraints for this lab.

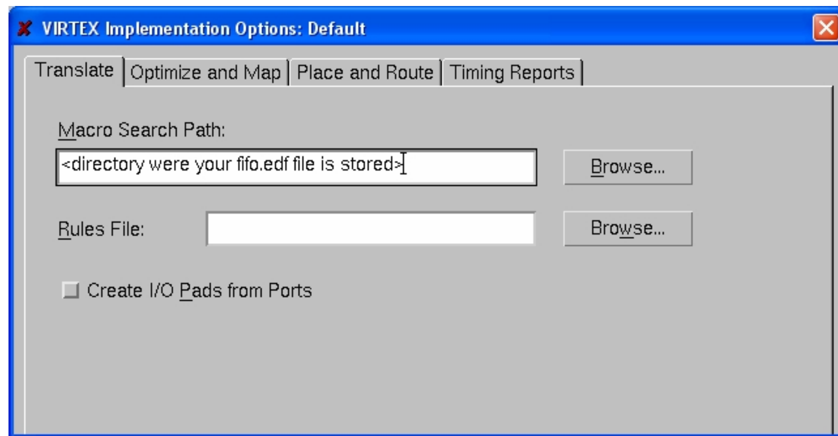
## 3.6 Implementation

Use the constraint file `Virtex.ucf` when you implement your design.

Otherwise set the “Place and Route Effort Level” to “High”. Turn off “Simulation” since you will not perform a timing simulation:




Click the “Edit Options” button for “Implementation” to bring up a new window. In the “Translate” tab you find something called “Macro Search Path”. Here you should enter the path to the directory where the net list for the FIFO is stored (the net list Core Generator generated)




Now press the Play button!

### 3.7 Testing your design

Now head for the lab and test your design. If it is working you have passed this first compulsory part of this lab. If it is not working you have to return to the UNIX lab ☹

You now need to show me (Jonas) or Magnus Lundberg that your design is working. There will be a schedule available for demonstration. You also need to submit some files. Please refer to the lab home page for details.

## 4 Task 2

Extend your design so that the CAPS Lock LED is lit when you press the CAPS lock key. If you press the same key another time the LED should be turned off.

Please archive your files for task 1 before starting with task 2. You should submit the two designs separately. Note that the dead line for task 2 is the exam date, so you may save this task for later.