

## Finite State Machines - Arbiter

In this lab, we specify a finite state machine (FSM) to be implemented later on an FPGA.

**Hand-in instructions:** Prepare a small report with block diagrams. Submit the report as a PDF with the source code files through the lecture moodle per the moodle submission deadline. Your report should include the FSM, the VHDL code, and the testbench.

### Important information

**Please always check these things before you start a lab or when you have issues!**

#### **FPGA:**

- Remember to use the reset button/switch on the FPGA to reset your design and to turn this off again if using a switch.
- Connect the FPGA board's Ethernet port to a computer, router, or any other device with an Ethernet port that can power the Ethernet chip on the FPGA board (no internet connection is needed). See the Lab 2 manual for an explanation.

#### **VHDL:**

- For combinational logic, use `process(a11)`. Do not write your own sensitivity lists! Please remember to change files using `process(a11)` to VHDL 2008. This is done by selecting the file in the Source tab. Look at the Source File Properties tab and change the type to VHDL 2008 by clicking on the 3 dots in the Type field.
- For defining registers, use the clocked process style `process(CLKxCI, RSTxRI)`. Do not define combinational logic like `CNTxDP <= CNTxDP + AxDI` inside a clocked process! See Task 4 in Exercise 3 and its solution for further explanation.
- Never write to the same signal in multiple concurrent statements! This means that if you assign to a signal in a process that signal can only be assigned to in that process and nowhere else. The only place you are allowed to assign multiple times to a signal is inside the (single) process where it is assigned to.

#### **Virtual machines:**

- EDA server users must start Vivado with `vivado -source load_board_files.tcl` as described in Lab 1. If you do not see the board files in the Vivado GUI, you have likely used the command with a spelling error or something similar.

#### **Windows users:**

- Avoid spaces and special characters in your filepaths! Vivado projects can become corrupted if you have spaces and special characters in your filepaths.
- You may have to disable your antivirus tool before running simulations in Vivado.

**For common questions/hints to this lab, please see the last page of this document which contains various hints and best-practices.**

## Arbiter - General Description

The goal is to implement an arbiter, which is a circuit that manages access to shared resources. For example, several processors may share the same block of memory, or several peripheral devices may be connected to the same bus. This lab considers an arbiter with two subsystems needing mutually exclusive access to some shared resources, as illustrated in Figure 1.

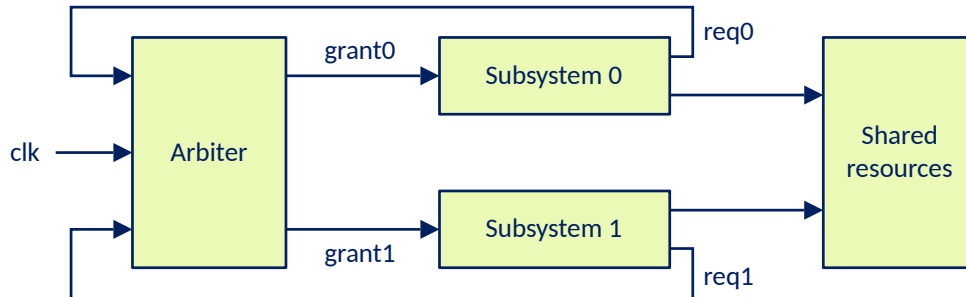


Figure 1: System diagram with subsystems, shared resource, and an arbiter that grants access.

Each subsystem may request access to the shared resources with a request signal *req0* or *req1*. The arbiter then decides to grant the access with a grant signal *grant0* or *grant1*. Once a subsystem has its grant signal activated, it has the permission to access the resources for doing a particular task. After the task has been completed, the subsystem releases the resources by deactivating its request signal. Since the arbiter's decision is partially based on previous events, i.e., previous requests and grant statuses, it needs internal states to keep a record of what happened which can be done with an FSM.

There are several issues in designing an arbiter:

- Handling of simultaneous requests: A priority scheme must be defined.
- Fairness: Each subsystem should have equal access to the resources.
- Availability: A subsystem should not block a resource by keeping its request for too long.

## Door Arbiter (FSM) - Description

Our arbiter controls access to a door. The requests are given by the two signals *Key0* and *Key1*, coming from the buttons on the FPGA board.

### Inputs/outputs

The arbiter indicates who has access to the door using the two RGB LEDs on the board. These are controlled by the signals *GLED0* and *RLED0* for the 1st LED (corresponding to *Key0*) and *GLED1* and *RLED1* for the 2nd LED (corresponding to *Key1*). A green LED indicates that access is requested and granted. A red LED indicates that access has been requested, but denied (due to another user accessing the resource). If no users request access, the green and red LEDs are all off.

### Timeout

The arbiter has a timeout of 2 seconds. This timeout ensures that none of the users/subsystems can occupy (keep its request) the resource for more than 2 seconds if the other user requests access. Hence, if a user is granted access and keeps its request active, the other user gains priority (i.e., it gets access as soon as it makes a request), while access is removed from the user that has occupied the resource. If no competing request is received, the user that has

access keeps its access until a competing request arrives, even if the timeout occurs. The timeout restarts when access passes from one user to the next.

#### Circuit designer's toolbox

We often have discussions with students on **the number of states necessary for a given problem. What really matters is clarity!** Your FSM should clearly describe your system, with each state being easy to understand.

- For the electronic lock system in the previous lab, we had a state for every correct/wrong button press, and states for the lock being open/closed (see the solution). This is a very explicit way of modeling our system using an FSM.
- If the electronic lock system was more advanced (supporting setting new passwords, etc.) then we may have implemented it using more high-level states, i.e., the correct sequence is not tracked with explicit states but with hardware outside the FSM process like a shift-register.

Note also that an FSM with few states is not a guarantee for a small hardware area, and an equivalent FSM with more states may be simpler for Vivado to implement. It is as Donald Knuth said: ***“Premature optimization is the root of all evil”<sup>a</sup>***.

<sup>a</sup><https://wiki.c2.com/?PrematureOptimization>

## Door Arbiter (FSM) - Tasks

### Task 1: FSM of Door Arbiter

Develop a state diagram of the arbiter. You are free to use fewer or more states, the main criteria is just to correctly and clearly model the arbiter. Remember to indicate your initial state (the state where the FSM starts after being reset) and all the state transitions, even the ones that go from a state to itself.

Note that you have some freedom in how immediate you want to give access, that is, it may be that in some cases, depending on the current state in your FSM, it takes one cycle to give access and you end up with a cycle where no one has access. The most important thing is just that the priority scheme is implemented correctly.

#### Hints and common errors

The FSM can be realized with 4 states and you will need one counter. Note that

- You can use more states if you want, but it is not necessary. For the number of states we propose, you can have 2 states to indicate that either subsystem 0 or 1 has 2 seconds of undisturbed access and 2 other states to indicate that subsystem 0 or 1 has priority over the other user if it makes a request.
- The task can also be done with 2 states, with only 1 state indicating undisturbed 2 seconds of access to a specific user and the other state indicating priority. However, this would require a bit more logic in each state to determine which user has priority.
- To determine if your FSM makes sense before dedicating time to writing it up in VHDL, you may also try to step through the FSM states using pen and paper with the test plan provided in Task 3 to see if your FSM behaves as expected.

## Task 2: VHDL Implementation

Implement the arbiter in VHDL. The FSM should be modeled using the case statement in VHDL inside a process(a11) block. The different states can be represented with enumeration types. Note that when you write the code for your FSM, you can generally avoid writing explicit code for indicating that no state change occurs in the FSM, it is enough to put a STATExSN <= STATExSP at the top of the process(a11). Please see the end of this document for an example on default values in a process(a11). For your interface, we propose a naming scheme as shown in Listing 1.

```
entity arbiter is
  generic (
    CNT_TIMEOUT : integer := 250000000 -- Timeout after 2 seconds @125 MHz
  );
  port (
    CLKxCI : in std_logic;
    RSTxRI : in std_logic;

    Key0xSI : in std_logic;
    Key1xSI : in std_logic;

    GLEDOxSO : out std_logic;
    RLEDOxSO : out std_logic;
    GLED1xSO : out std_logic;
  );
end entity;
```

Listing 1: Entity declaration for arbiter in arbiter.vhdl.

## Task 3: Verification

Write a simple testbench to simulate and test the arbiter. For the simulation, reduce your timeout to something small like 2 milliseconds or 200 nanoseconds.

Verify the following scenarios for users *U0* and *U1* (matching LED numbers):

- A) *U0* requests access which is granted immediately.
- B) *U0* requests access which is granted immediately. Then, *U1* requests access while *U0* keeps its request active. *U1*'s access request is denied and it releases its request before a timeout occurs for *U0* meaning *U0* can keep the access even after a timeout occurred.
- C) *U0* requests access which is granted immediately and releases its request before the timeout. *U0* requests access again and maintains its request. *U1* requests access almost immediately after but access should only be granted after the timeout for *U0*.
- D) *U0* requests access which is granted immediately. Then, *U1* requests access while *U0* keeps its request. *U1*'s access is initially denied, but is granted after the timeout. *U0* maintains its request and the access goes back to *U0* after another timeout period. Let this test run for some time continuous arbitration, i.e., both users should continue to request access for the remainder of the simulation.

## Task 4: FPGA Implementation

Implement and test your code on the FPGA. Remember to change the timeout to 2 seconds.

**Important!** For the design to be fully working, it is necessary to connect the FPGA with an Ethernet cable to a computer, router, or any other device with an active Ethernet port. This is because the FPGA clock comes from the Ethernet chip on the board. However, you should still be able to make it work without an Ethernet cable, most of the time.

## Common Questions

Common questions/remarks for this lab are:

- **How do I create a VHDL file?** After creating a project in Vivado you can click File → Add Sources → Add or create design sources. Alternatively, just use your normal code editor and create new files with the .vhd1 (recommended) or .vhd extensions.
- **How do I resolve the warning 'The PS7 cell must be used in this Zynq design ...'?** This warning can be safely ignored as it's unrelated to what we do on the FPGA.
- **How do I resolve the error 'Unconstrained Logical Port'?** While VHDL itself is case-insensitive, .xdc **constraints are case sensitive** and your port names should match those in the .xdc file in case as well.
- **Outlook does not allow .vhd files:** You cannot send .vhd files in Outlook, try renaming to .vhd1 as the .vhd extension is also used for **Virtual Hard Disk** on Windows.
- **Remember that order matters in processes!** Since the order of assignments are done sequentially in a process, meaning that in the example below DxSO is only assigned AxSI and BxSI and never AxSI or BxSI.

Listing 2: This implementation ignores the line AxSI or BxSI as it is always overwritten by the final assignment to DxSO.

```
process(all)
begin
  if (CxSO = '0') then
    DxSO <= AxSI or BxSI;
  end if;

  CxSO <= not AxSI;
  DxSO <= AxSI and BxSI;
end process;
```

- **Remember to separate the description of the flip-flops from the combinational logic!** Use a single process for updating the flip-flops and a separate process or concurrent assignments for updating the adder as shown below. This is really important! We also discuss this in Exercise 3.

Listing 3: VHDL code to show how to define flip-flops for a counter.

```
CNTxDN <= CNTxDP + 1; -- Increment outside clock-process

process(CLKxCI, RSTxRI)
begin
  if (RSTxRI = '1') then
    CNTxDP <= (others => '0');
  elsif CLKxCI'event and CLKxCI = '1' then
    CNTxDP <= CNTxDN;
  end if;
end process;
```

- **Remember to never write to the same signal in multiple concurrent statements!** When assigning to a signal in a process, you can only assign to that signal in that (single) process. The code below in Listing 4 shows the signal CNTxDN being assigned to in two different concurrent statements, which is not allowed. With this code, you will see an 'X'

for CNTxDN in the waveform viewer. The solution is to put the default assignment in a process like shown in Listing 5.

Listing 4: VHDL code which shows how not to assign to a signal!

```
CNTxDN <= CNTxDP;  
  
process(all)  
begin  
    if (In0xSI = '1') then  
        CNTxDN <= CNTxDP + 1;  
    end if;  
end process;
```

- **Use default values in your process!** To avoid introducing errors in your code from missing assignments to signals, you should always use a default value for all signals assigned to in a process(all) when describing combinational logic as shown in Listing 5. This is done as the first thing in a process.

Listing 5: VHDL code which shows the assignment of a default value.

```
process(all)  
begin  
    -- Default values  
    CNTxDN <= CNTxDP;  
    AxD    <= (others => '0');  
    BxD    <= (others => '0');  
  
    -- Actual logic after default values  
    if (In0xSI = '1') then  
        CNTxDN <= CNTxDP + 1;  
        AxD    <= In1xSI;  
    elsif (In1xSI = '1') then  
        CNTxDN <= CNTxDP - 1;  
        BxD    <= In1xSI;  
    end if;  
end process;
```