CHAPTER 2

FSM State Assignment and VHDL Synthesis

In this chapter you will practice how to assign the state codes in a Finite State Machine for minimizing power consumption (section 2.1). This means that you should already know how to describe an FSM using VHDL language. For this reason a final section (2.3) of the chapter gives an example of FSM described in VHDL for your convenience.
Furthermore you will learn how to synthesize a circuit (section 2.2), and you'll use as a working example your section 2.1 results.

2.1 FSM State Assignment (TL & TP)

Different state assignments of a finite state machine may result in different power consumptions. The code assignment that guarantees the minimum Hamming distance between states is not always the best choice. A good state assignment for low power should take into account the transition probability. Given a state transition graph (STG) $G = \{V, E\}$ where the set of vertices V is the state set $V = \{S_1, ..., S_N\}$, and the edges E are annotated with the probabilities p_{ij} of having a transition from S_i to S_j under the condition of being in state S_i, good results are obtained if one minimizes the cost function

$$\gamma = \sum_{over\ all\ edges\ (i,j)\in E} p_{ij} H(S_i, S_j).$$

The simulated annealing heuristic algorithm is often used to find a good solution.

However, the state assignment alone may not produce very good results because the output function depends on the state assignment as well. A good state assignment which minimizes the cost function γ might imply the use of more logic gates and a higher activity in the output function. Therefore new cost functions are proposed by researchers to take into account this additional source of power consumption in FSM's.

Most of the times the algorithms are implemented in Electronic Design Automation (EDA) tools that are designed to be used in a "push-button" fashion so that the designer has not much control of what the tool is going to do. When the designer uses its know-how and intelligence in the early stages of the design, the results are often much better. With this in mind let's go to analyze the following problem.

Suppose you have to implement a circuit to sum 6 numbers

$$s = a + b + c + d + e + f$$

and that your datapath contains **only one adder**, as depicted in figure 2.1.

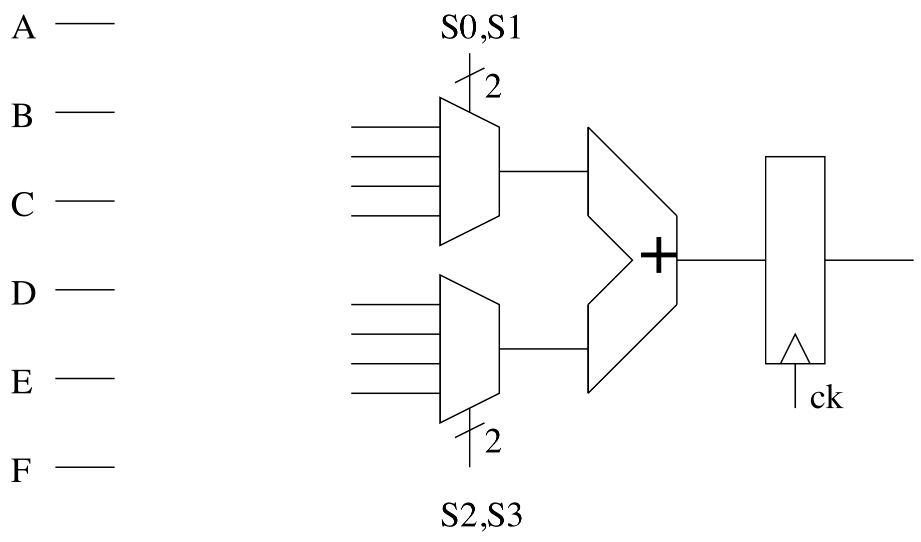

Figure 2.1: Cicruit implementation of an adder.

Design a FSM (behavioral VHDL with the three processes) to control muxes control signals S0-S3 in order to implement the sum of 6 numbers. Choose a state encoding that minimizes the FSM activity (both next state computation and output function). Choose also how to connect inputs A-F to muxes inputs so that the power consumption is minimized. To evaluate the power consumption of your implementation consider only the activity (toggle counts) of the FSM and of S0-S3 bits, **do not consider the adder inputs and output**. For sake of simplicity every net will be weighted as 1 in your toggle count and power evaluation, regardless the type of gate that produces that output. The work group that will minimize the toggle count will win the game.

Once you have defined your FSM and your optimal state assignment, you are ready to describe in VHDL it (you can use the simple FSM in the previous section as a starting point) and simulate it together with the adder. In order to report a correct state encoding you can use the *attribute* VHDL command (see the example reported below). Keep in mind that the aim of this exercise is to find the optimal state assignment with the minimum number of states. Thus, in this specific case the definition of a IDLE state should be avoided.

```
ARCHITECTURE arc_enumtype OF enumtype IS

-- enumerated type state machine encoding
TYPE states IS (idle,state1, state2, state3, state4, state5);
ATTRIBUTE enum_encoding            : string;
ATTRIBUTE enum_encoding OF states : TYPE IS "000 100 101 111 110 010";
SIGNAL state_vector                : states;

BEGIN
```

This will assign the values “000” to idle, “100” to state1, etc. The state assignment in this example is gray coded.

You are already provided of the adder code and you will find it using the usual command:

```
prompt> cp /home/repository/lowpower/ese2/dpadder.vhd .
```

So you need to create a new entity including both the adder and your FSM. You will use this top entity, let's call it FSM-ADDER in next section as well.

Remark point

In summary you have to connect the inputs to muxes, to design the state assignement (reducing the total amount of the transitions) and the VHDL structure of the FSM, to simulate the output, to evaluate the power consumption both for the output and the next state computation.

2.2 VHDL Synthesis (TL Homework, TP optional but STRONGLY suggested)

In this section you will synthesize your FSM and you will learn how to constrain the synthesis so that you can optimize your goal (timing, area, power....).

In general the phases you should follow using a synthesis tool are reported below:

In your new directory (ese2) generate a new subdirectory named for example **synth**, go in there (**cd synth**) and copy the file:

/home/repository/lowpower/.synopsys_dc.setup

Remember hat it gives some information to the tool, such as which is the library you are going to use and where it can be found. Anyway, you don't need to modify it. But remember, whenever you need to synthesize you should copy this file in your working directory.

Now the environment variables needed to launch the Synopsys synthesizer must be defined: this is done in a script file that you can execute by typing the command:

```
prompt> setsynopsys
```

As a last preliminary operation create a directory in which the synthesizer save temporary files using during compilation:

```
prompt> mkdir work
```

A final important comment. With the informations you will be given hereinafter, you should be able to synthesize all the VHDL designs you have already completed. Anyway two are the main ways to perform a synthesis: using the graphical user interface and going to and from around the monitor clicking with your mouse here and there, or using a command window in which you are allowed to execute singular commands or a script grouping a command sequence that you execute once. The use of the command window is obviously the clever way, but it is not for a newbie. So you'll learn here how to click in the right sequence and understand what's going on; sometimes informations and suggestions will be given so that, hopefully, you will be automatically transformed from a newbie to an experienced user.

2.2.1 Synthesis of your structural fsm-adder

Now we are ready to synthesize the design you completed in previous section. We will call herein **fsmadder_struct.vhd** (Of course you can use other filenames and have more than one file, e.g. your fsm file and the datapath_adder file). Be sure the files are in the current directory (ese2/synth). Now launch the synthesizer by typing:

prompt> **design_vision &**

A main window opens with title: "Design Vision". Immediately note the bottom command line, from which you can manually set commands. In the space above, the results of your commands are displayed. If you operate using the main window menu, the name of the corresponding command and its results are displayed in the command window as well. For the moment use this command line only when suggested.
Please note exactly above the command line the menu with the alternative "Log", which corresponds to the log window above, and "History": select it. Each time you will type a command in the command line, or give a command by the Design Vision buttons, you will have it in this history window. You will be able to use it for editing, executing or saving such commands, so that you will easily generate scripts for you next sessions.

Now let's go back to the Log window and let's use the interface: using its menu perform the following operations:

Analyze file: From menu select: **File→analyze**; a window opens from which you can type "Add", select the file you want to read. For example select the **dpadder.vhd** file and click SELECT and OK to confirm your selection. In this phase the code is checked and errors are displayed if it is not synthesizable. Note the corresponding command in the Log window and in the history window.
Now analyze your **fsm.vhd** file (or whatever is the name you used) and the **fsmadder_struct.vhd** file with the same steps. Please note that we are no more using the test bench: the top entity is the FSM_ADDER. **Be careful herein**: if you have your circuit described in more than one file you must read all the files from bottom level to top level in terms of hierarchy.

Elaborate From menu select: **File→elaborate**. During this step the compiler creates an internal graph of your circuit and maps it virtually to general gates. You need to elaborate only the top level entity, that is the FSM_ADDER. In the "Elaborate design" window select the "WORK" library, and the "FSM_ADDER" option.
You can now explore this "logical" structure of the "FSM_ADDER". In the left "Logical Hierarchy" window select the "FSM_ADDER" structure. Now, in the top menu the "Create Symbol View" and the "Create Design Schematic"appear. Click the "Create Symbol View" button: a "FSM_ADDER" symbol appears. Now double click inside of it: the schematic view appears. You can surf in it by using the right mouse button (Zoom selections... ESC to leave the Zoom option). Choose one of the inside blocks and double click in it. What you see is the logical block representation created using generic basic ports. These are not the cells in the library we are going to use for the synthesis.
Before steeping over, click on the UP ARROW (on the top menu) till you arrive to the top view. Check that in the log window appears: **current_design "FSM_ADDER"**

Look one moment in the history window the correspondence between your selection and the given command.

Create clock: supposing you used the name "clock" in your VHDL for describing the clock signal, you must now define the clock waveform. This is more easily done using the command window, in which you can type:

create_clock -name "CLK" -period 10 {clock}

where "CLK" is the name of the signal generator associated to your pin "clock". Pay attention: the name of the pin must be correct, otherwise a clock generator will be created, but you won't see effects on the design.

If you want to check if the clock has been created just type on the command window bottom bar:

report_clock

If you want to know what a command performs, you can type in the command window: "man command_name", e.g. "man report_clock", and a detailed explanation will be displayed.

Attention: all the following commands will be executed on the current design, that is the one which is active in the display window. You must click on it once to be sure it is the active one.

Synthesize: for the moment we do not use constraints different from the clock period, so go to the main window and select **Design→Compile Design**. A small window opens, in which you don't need to select anything different from the default: just click OK. Now the tool is mapping your design. This corresponds to giving in the command window the instruction

compile -exact_map

Now the tool is mapping your design on the library we are using ($0.045\mu m$). When it has finished you can explore the results from the main window exactly as before.

You can plot to a file the schematic displays in the main window by selecting **File→Print schematic**, checking the "Print to file" option and then typing the file name.
If you want to go up in hierarchy you must click on the up arrow displayed on the top bar of the main window. Remember that whatever command you are giving, it will be applied to the hierarchical level you are in.

Save the design: We want to save this compiled design so that, in case we optimize it or change something, we can start again from here without reproducing the previous steps. From the main menu select: **File→Save as**, and in the new window put a meaningful name, eg. "FSM-ADDER-simple.ddc". Hereinafter you will be able to get your design at this point by simply reading it from the main menu (**File→Read** and select it.)
Please look at the command in the history page and remember it.

Generate to VHDL file: We want to see the VHDL synthesized netlist of the whole design so that later we will be able to perform a backannotated simulation.

From the main menu select: **File→Save as** and choose vhdl as FORMAT (vhdl, not vhd). Now look at the file end analyze the mapped structure: clearly it suppose to have the entity description of each of the component declared.
Please look at the command in the history page and remember it.

List reports: report area Now we want to analize the circuit performance. From the main menu select **Design→Report Area**. A new window opens. You can decide (do decide it now..) to write the report on an external file by checking the "to File" box and inserting the file name, e.g. *FSM_ADDER_area_simple.txt.* The same command can be obtained by typing in the command line

report_area > FSM_ADDER_area_simple.txt

Hereinafter instructions for saving the reports will not be given: it's up to you to decide when you need to save the results on a file or to write down the results you will find from these reports so that you can compare them with future results obtained by constraining the synthesis.

Report fsm assignment: Another interesting report to be analyzed is the state encoding for the FSM:

type **report_fsm** ... for the state encoding

(this works only if you have described our FSM with virtual STATE definitions (*state_vector*), otherwise your one is a normal sequential circuit and the tool doesn't recognize it as an FSM).

Report timing: Now report the timing analyses: from the main menu select **Timing→Report Timing paths** and leave the default settings. Save the report on a file. You will see the timing report for the worst critical paths on the report window: try to understand the given information.
The same results could have been obtained simply writing in the command line (try it) the command:

report_timing

In this way the worst critical path is displayed.

Go back to the top level. It is possible to see in order also the others paths, for example, the 10 worst C.P. can obtained with the command:

report_timing -nworst 10

or select from the main many **Timing→Report Timing paths** and choose in the "Worst path per endpoints" 10.

Which are the differences among the paths?

Now use this interesting graphical feature: from the main menu select **Timing→Endpoint Slack**. Leave the default settings and see the results: a path distribution is being displayed. Click on the first histogram rectangle. The worst slacks are shown. Click on the second: what's being displayed? Play with with the display options as you like....

Report power: Let's look at the power dissipation:

report_power

or by the main menu: **Design→Report Power**, leaving the default configuration (save file). The report you have here is general for the whole FSM_ADDER, and separated in three contributions: do you remember the source of these power dissipation? You can have more detailed information by selecting in the **Design→Report Power** window not summary only, but cells only: you have now the contribution on each cell in the top level circuit. This action corresponds to the command:

report_power -hier

Note the difference among the internal switching power, the driven net switching power and the cell leakage power in the sub-blocks. A further information you can get on power is the toggle count of each net. Just type:

report_power -net ... for the power report regarding the circuit nets or from the **Design→Report Power** window selects “nets only”.

Now let’s go inside the sunblocks. Type:

list_designs

and a list of the current available blocks are listed. Now we want to analyze the FSM power dissipation. Type:

current_instance FSM

(if this is your FSM design name). If the cell cannot be found, specify the full cell path within your design. Now report the cells (report_power -cell) and net (report_power -net) power as before for this unit.

There are three important points you should remember.

1. If you don’t change the values a default 0.5 toggle probability is assumed for the inputs, and the consequent probability is derived for the internal nodes. (This point will be clarified during the next lessons). Methods are available for importing the real switching activity after a back-annotated simulation has been performed.
2. You know that the power is a function of frequency, that is of period. The period used for computation here is the one we defined before, that is 10ns (the library timing unit is 1ns). In the following we will change it and see what happens to to power report.
3. **Again, pay attention at which is your hierarchy level:** if you are at top level you will report information regarding the whole FSM and ADDER, while if you want a detailed report for the FSM only you should go at the FSM level and type the report command in that case.

If you want to save you reports you can redirect on a file the command results. For example:

report_power > powerfilename1

Constraint the design You can play now by changing for example the clock period, or defining a power constraint. Go up to the top level design. Using the command defined before define the clock period as the maximum critical path you found by your previous timing report. Now display the new power dissipation (report_power): note that the design didn’t change, only the working frequency did.

Now let’s constrain the power dissipation. Again in the command window type:

set_max_dynamic_power POWER

where for POWER you can insert a number lower than the one you measured before in the unconstrained synthesis power report (unity is 1pW); now go through the optimization again: **Compile -exact_map**. You can now display a new report and compare the results with respect to previous synthesis. Now play with the synthesizer until you find the power limit.

Another hint: you can also list information directly on the constraints by typing in the command window the command

report_constraints

Save files: as you learned before, save the ddc file, the synthesized VHDL file and the new SDF file (now your circuit has changed).

Generate script At the bottom of the main window go the the history menu and save the command history in a file "my_first_synthesis_script.scr". Now edit it and leave only the effective instructions (delete the display instructions...). Hereinafter you can repeat all this instructions simply typing in the command window (try it before closing...)

source "my_first_synthesis_script.scr"

VERY IMPORTANT Hereinafter you are invited to use script files for rapidly synthesize and constraint the synthesis results. Use the history window for help.

MOST IMPORTANT Hereinafter it is expected that you use the command line help, e.g. if you write on the command window : "man set_max_dynamic_power" the manual page will be displayed (note that the shell has the completion..). However, depending on the server configuration and Synopsys version, the "man" may not be found.
Furthermore from an external shell you can use the command "SOLD" to get the whole synopsys manual documentation. For synthesis you can use all the Design Vision and Design Compile manuals.

Remark point

How to synthesize a design, how to report its performance and saving reports, how to constraint the design (creation of fsm, timing, power and constraints reports). Initial hints on how to use synthesis scripts.

2.3 FSM in VHDL

VHDL offers a great number of modeling possibilities for FSM. The main characteristics are:

- the FSM is always in one of the possible states: the CURRENT-STATE
- this state is memorized in a specific register
- the NEXT-STATE is computed using the CURRENT-STATE and the inputs
- the OUTPUT is computed (Moore machine) considering the output values
- during each period the state register is updated with the previously computed next state

Then the three main operations to be used for describing the FSM are:

- NEXT-STATE computation
- CURRENT-STATE computation
- output assignements

This can be done using a single VHDL process, and it works, but the elements inferred during the synthesis phase would be expensive (see example from "Circuit Synthesis with VHDL" given by the teacher). Thus one of the most used style of description splits these three phases in three processes. In the following you have an example of a simple two state FSM describing an odd parity checker. The input is a bit serial stream and the output is and ODD/EVEN count flag (0 is even, 1 is odd). The state transition diagram is in figure 2.2

Read the comments embedded in the code below.

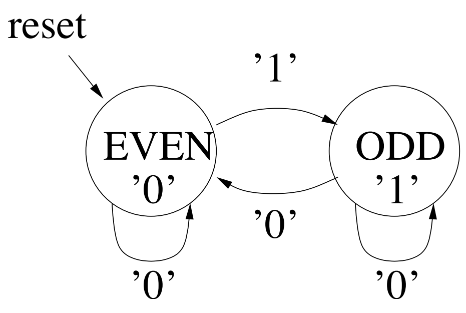

Figure 2.2:

```
library IEEE;
use IEEE.std_logic_1164.all; --  libreria IEEE con definizione tipi standard logic


entity odd_parity_checker is
port(   A:       in std_logic;
        clock:   in std_logic;
        reset:   in std_logic;
        O:       out std_logic
);
end     odd_parity_checker;

---------------------------------------------

architecture FSM_OPC of odd_parity_checker is

        type TYPE_STATE is (S0, S1);
        signal CURRENT_STATE : TYPE_STATE;
        signal NEXT_STATE : TYPE_STATE;

begin
        P_OPC : process(CLOCK, RESET)
        begin

                if reset='1' then
                        CURRENT_STATE <= S0;
                elsif (CLOCK ='1' and CLOCK'EVENT) then
                        CURRENT_STATE <= NEXT_STATE;
                end if;
        end process P_OPC;

        P_NEXT_STATE : process(CURRENT_STATE,A)
        begin
                NEXT_STATE <= CURRENT_STATE;
                case CURRENT_STATE is
                        when S0 => if A ='0' then
                                        NEXT_STATE <= S0;
                                   elsif A ='1' then
                                        NEXT_STATE <= S1;
                                   end if;
                        when S1 => if A ='0' then
                                        NEXT_STATE <= S1;
                                   elsif A ='1' then
                                        NEXT_STATE <= S0;
                                   end if;
                end case;
        end process P_NEXT_STATE;


        P_OUTPUTS: process(CURRENT_STATE)
        begin
                --O <= '0';
                case CURRENT_STATE is

                        when S0 => O <= '0';
                        when S1 => O <= '1';
                end case;
        end process P_OUTPUTS;
end FSM_OPC;


configuration CFG_FSM_OPC of odd_parity_checker is
        for  FSM_OPC
        end for;
end CFG_FSM_OPC;
```

Here is the test bench:

```
library IEEE;

use IEEE.std_logic_1164.all;



entity TBFSM is
end TBFSM;

architecture TEST of TBFSM is

        signal T_IN: std_logic;
        signal T_CLOCK: std_logic;
        signal T_RESET: std_logic;
        signal T_OUT: std_logic;

        component odd_parity_checker
        port (  A:      in std_logic;
                clock:  in std_logic;
                reset:  in std_logic;
                O:      out std_logic);
        end component;

begin

        U_fsm_opc: odd_parity_checker
        Port Map (T_IN, T_CLOCK, T_RESET, T_OUT);

        process
        begin
                T_CLOCK <= '1';                 -- clock cycle 6 ns
                wait for 3 ns;
                T_CLOCK <= '0';
                wait for 3 ns;
        end process;

                T_RESET <= '1' after 2 ns, '0' after 10 ns;
                T_IN <= '1' after 3 ns, '0' after 11 ns, '1' after 19 ns, '0' after 25 ns, '1' after 34

end TEST;
```

```
configuration CFG_TB of TBFSM is
        for TEST
        end for;
end CFG_TB;
```

The "VHDL cookbook" (you can read it using the command **helpvhdl** from a shell) describes the FSM in a complex example that you are invited to read in chapter 7 (pag. 7-39 and following).