CHAPTER 4

Bus encoding

In this chapter you will experiment some of the existent bus encoding techniques: **bus invert** and **transition based** that are best candidate for the data bus cases, and **gray encoding** and **T0** best for the address bus cases. You will use as a reference the performance of a non-encoded bus. Please note that the correct way to evaluate the advantage due to the encoded bus is to analyze both the power dissipation due to the bus itself and the power due to the encoder and the decoder. For this reason in a first step you will simulate the encoded system and evaluate the performance in terms of number of transition in the bus; in a second step you will synthesize the system for evaluating both the power dissipation on the bus and the one due to the encoder and the decoder. In this second phase the power analysis will be based on the toggle count computed during the simulation phase (SAIF method learned in previous class).

4.1 Simulation (TL & TP)

For the simulation phase we will use a test bench in which are, or should be, present:

- as components:
  - busses:
    - non-encoded bus (given and included) used as a reference
    - bus-invert (given but not included) for data busses
    - transition-based (given but not included) for data busses
    - gray (given but not included) for address busses
    - T0 (to be implemented) for address busses
- as processes:
  - an address bus like input data (partially incremental data, partially randomly jumping)
  - a data bus like input data (random data)

You are provided with the VHDL of the test bench and of the busses, that you can copy from the usual directory together with all the given files:

prompt> cp -r /home/repository/lowpower/ese4/* .

The recursive option -r allows to copy the subdirectory **synth** you will be using later.

4.1.1 Non-encoded

We will use the non-encoded bus as starting point to compare the power performance of the encoded busses, i.e. for both the data and address busses.

Let's consider a simple 8 bit bus, as shown in figure 4.1. Note that, depending on the technology used and on the bus length, each wire on the bus has an associated load; this, together with the switching activity on the bus, contributes to the bus power dissipation.

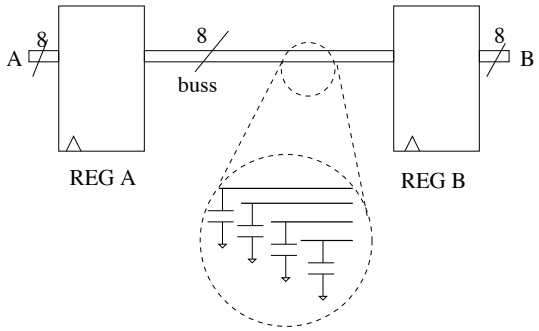

Figure 4.1:

Read the testbench file **tb_encdec.v**: please note that each bus description (encoded or not) should be a top entity as sketched in figure 4.2 in the left, while the internal description, for each encoded bus

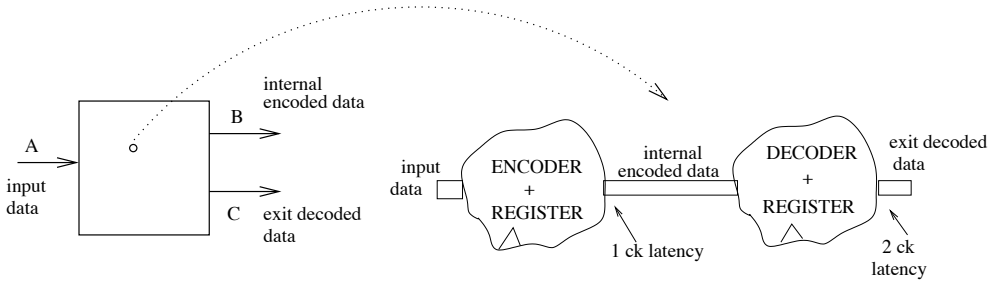

Figure 4.2:

should be as in the right sketch in the same figure. You have an example of such a structure in the file **busnormal.vhd** used to describe the non-encoded bus. In the test bench, as initial exercise, only the non-encoded bus is included, so that you can easily analyze how the test bench behaves. After the first simulation you will insert (next paragraph) the bus-invert, the transition based and the gray technique as well (you should have them in your directory right now).

If you go through the reading of the test bench you will find two **always** blocks enclosed within two macros: **ADDRESS** for address like input for the bus and **DATA** for data like input for the bus. Both blocks do not directly define the input variations, but simply read a stream file *rndin.txt* where each row is the input for the bus at each clock cycle. Read the two processes and try to understand how they differently behave. Note that the random data has not a low 1 probability: one of the encoding technique could not work properly in this case: do you remember which one?
So we will use two input data type. The process selection is done with two macros defined at the top of the testbench:

- ADDRESS: address like
- DATA: data like

So you are ready to simulate the non-encoded bus using the files: **tb_encdec.v** and **busnormal.vhd**. Use the address input process as first step. The random data are generated for 10000 clock cycles, so read in the test bench the clock period and run the simulation for 10000 clock cycles. Look at the output waveforms and be sure to have understood how the test bench and the bus are described. Remember that with the **power report** command you can find the switching activity of the internal bus **B**. Compare the switching activity in the two DATA and ADDRESS case.

4.1.2 Bus-invert, Transition based, Gray

Now you are ready to compare the non-encoded bus with the other given techniques. Before simulating let's go through a brief bus encoding tour.

Do you remember the **bus invert** technique? The encoding algorithm is such that, if the transition from one data to the next one implies the switching of a number of bits greater than n/2, then it is more convenient to use the inverted data, sending to the receiver an information (the invert bit) on the decision (the data received is inverted with respect to the original one).
You are given the VHDL code in the file **businvbeh.vhd**. Read it carefully and note the difference in the internal encoded data std_logic_vector length.

For the **Transition based** case you have the description in the file **transbased.vhd**: analyze it and check if it is coherent with the structure in figure 4.3. How does it work?

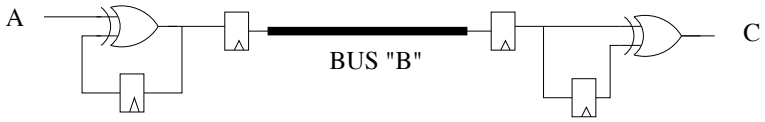

Figure 4.3:

The simplest way to realize a **gray encoder** and decoder is the one sketched in figure 4.4 (here for 4 bits only). In the VHDL (behavioral architecture) in file **grayencoder.vhd** it is implemented for an 8 bit bus. Check the code and verify if an incremental input can be correctly encoded in a grey sequence and then decoded.

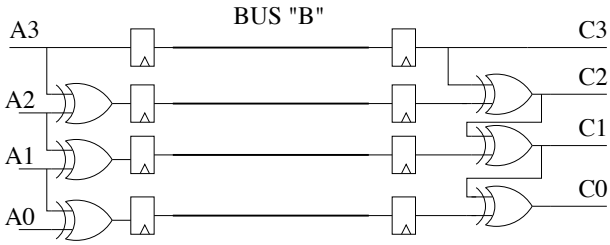

Figure 4.4:

Comparison

Now you are ready to simulate and evaluate the performance of the three busses with respect to the non-encoded one. So you need to include in the **tb_encdec.v** the "businvbeh", the "transbased" and the "grayencoder" components. Moreover you must add the correct signals and define the port mapping. For your utility try to call the internal bus (B) for all the techniques with a similar name

(e.g. COUNTBUSNORM, COUNTBUSINV, COUNTBUSTRAN...) so that later on it will be easier to select those signals for the power report command.

Finally you have to assign the input for all the components used.

Once you have prepared your new test bench you are ready to perform your two round simulation: the first one, ADDRESS, for the address input (DATA commented) the second one, DATA, for the random data input (ADDRESS commented). If you prefer you can split the test bench in two files, each with the macro process uncommented.

For each bus you should note that the internal bus signal B is different. So you will have a different switching activity. For annotating such activity you must, before running the simulation, use the "power add" command, and for selecting only the input activity and the internal bus activity you can use the trick:

VSIM> power add /testbench/A /testbench/COUNT*

(Of course this will work only if you have called all the signal for internal bus B with an initial COUNT..., otherwise you must specify all the signal names).

After the simulation you can run the power report command and redirect it on a file, for example, for the address input process:

VSIM> power report -file activityADD.rpt

You will be using these data later during the synthesis phase, so do not forget this step.

Now repeat the simulation for the DATA process, and save the report. Analyze the different performance.

4.1.3 T0

As a last case you should remember the T0 encoding. It is not requested to implement it right now, but to keep it in mind for the final project of for the final lab report. For those who have chosen the lab. reports and not the projects it is requested to implement this bus as an homework. You can compare its performance and extract the encoded switching activity exactly as for the other three encoding techniques. You should remember that the function used is:

$$\left(B^{(t)}, INC^{(t)}\right) = \begin{cases} \left(B^{(t-1),1}\right) & if \quad b^{(t)} = b^{(t-1)} + 1 \\ \left(b^{(t)}, 0\right) & otherwise \end{cases} \tag{4.1}$$

where $b^{(t)}$ is the input data at time t, $B^{(t)}$ is the encoded data at time t, and $INC^{(t)}$ is the increment bit at time t.

You can instantiate this case in the test bench, but remember that, as in the case of the bus-invert, a further bit is sent to the decoder. Again compare the behavior of this technique for the three process cases and compare to the other busses performance.

Remark point

How the encoding techniques work and their different performance; how to report their encoded activities.

4.2 Synthesis (TL Homework, TP optional but STRONGLY suggested)

We will see how to calculate the power consumption of the bus in presence the two different input probabilities for ADDRESS and DATA for the given four types of bus encoders (the T0 case is left as homework). In this part of the laboratory session we will use the SAIF backannotation process to make the power estimation process easy. Since the synthesis process will be automatic, it is necessary to copy the necessary scripts for synthesis in the appropriate locations.

Make sure that you have the `.synopsys_dc.setup` file in the `synth` directory. From one workspace run `modelsim` (first prepare the environment variables by running `setmentor`). Make sure that your current directory is `/ese4`. Take a look inside these directories. In `synth` you should have different scripts. You don't need to use these scripts one by one, you only need to run two of them, that is `create_saif.scr` and `backward_all.scr`. Inside the upper level directory (`ese4`), you have other two important scripts called `fill_forward.scr` and `datatype.def`.

All these scripts are organized in such a way that they perform SDF forward annotation (from Synopsys to Modelsim), VCD file extraction (Modelsim simulation that annotate activities for the nodes) and SAIF backward annotation with power report. This last step involves the annotation of the switching activities acquired from the Modelsim simulation, synthesis and power report. Take a look and try to understand their purpose before continuing. You should find out that the previously mentioned scripts recall smaller scripts and generally have the following purposes,

- **create_sdf.scr** `[Design Vision]` - Synthesize the architectures and saves the designs, the SDF files and the corresponding verilog netlist in `/ese4`.
- **fill_forward.scr** `[Modelsim]` - Reads the SDF information and simulates annotating the activity within a VCD file, which is then converted into SAIF for the successive backannotation.
- **backward_all.scr** `[Design Vision]` - Backannotates activities, synthesize the designs and calculate power consumption.
- **datatype.def** `[noarch]` - Configuration file for the logical identification of the results.

You can also note that a `definitions.scr` is present in the directory `synth`: It defines variables for changing the load of the bus and other strings for the creation and the load of the SAIF file. It is strongly recommended to run the **Modelsim** and the **Synopsys** tools by two different shells located in two different workspaces. We suggest also to use a third workspace to view and modify the scripts for synthesis and backannotation.

It is necessary that you instantiate the components in the testbench with a SPECIFIC NAME. Thus, in the `tb_encdec.v` you MUST instantiate the components `busnormal`, `businvbeh`, `transbased` and `grayencoder` as **UBUSNORM**, **UBUSINV**, **UBUSTRAN** and **UBUSGRAY**.

Let's suppose that we want to calculate the power consumption in presence of ADDRESS probabilities at the encoder input. Open the file `datatype.def` and make sure that the command

set DATATYPE ADDRESS

is the only uncommented one. This variable configures the file creation process: In these conditions the scripts will generate and look for files names with the appended string "ADDRESS". Since we want to store the input probabilities for addresses make sure that the testbench uses the **ADDRESS** macro for generating the encoders inputs. Run `synopsys` within `ese4/synth` and run the **create_sdf** script. Read carefully the commands it includes before passing over,

```
source definitions.scr

variable targetcompilation "../busnormal.v"
variable top_hierarchy "busnormal"
analyze -format vhdl $targetcompilation
elaborate -library work $top_hierarchy
compile -exact_map
write -hierarchy -format ddc -output $top_hierarchy.ddc
ungroup -all -flatten
change_names -hierarchy -rules verilog
write_sdf ../$top_hierarchy.sdf
write -f verilog -hierarchy -output ../fromsynopsys$normtag.v
```

After having defined the symbols with the execution of `definitions.scr` the script analyzes, elaborates and compiles the designs (in this example the `busnormal` entity) and saves the SDF, the verilog netlist and synthesized design as .dcc file. Now you are ready to run it, so from the Design Vision command window type

source create_sdf.scr

It should create four SDF files and the verilog netlists in the `ese4` directory and the designs as .dcc within `ese4/synth`. Let's take a look at them by considering the first part of the file `busnormal.sdf`,

```
(DELAYFILE
(SDFVERSION "OVI 2.1")
(DESIGN "busnormal")
(DATE "Fri Apr 26 10:25:48 2019")
(VENDOR "NangateOpenCellLibrary")
(PROGRAM "Synopsys Design Compiler cmos")
(VERSION "F-2011.09-SP3")
(DIVIDER /)
(VOLTAGE 1.10:1.10:1.10)
(PROCESS "typical")
(TEMPERATURE 25.00:25.00:25.00)
(TIMESCALE 1ns)
(CELL
  (CELLTYPE "INV_X1")
  (INSTANCE U4)
  (DELAY
    (ABSOLUTE
    (IOPATH A ZN (0.095:0.095:0.095) (0.048:0.048:0.048))
    )
  )
)
```

It contains timing data for each cell in the design for use with VITAL (VHDL Initiative Toward ASIC Libraries) compatible models. All the information reported use the format (Min : Typ : Max).
For example, for the `INV_X1` cell, the first round bracket encloses the rise time, while the second one the fall time of the cell with their corresponding minimum, typical and maximum delay.
The aim is to use the SDF file with the estimated timing data with the cell library verilog models in `modelsim`. The verilog VITAL models, which have been already compiled for you, contains delays

and constraints information of each cell. Some of the delay information contains default values, that will be replaced the values contained within your SDF file during the simulation.
Run `modelsim` within the `ese4` folder, close all the open projects, make a new project and save it inside the `ese4` directory. Compile all files, i.e. the verilog testbech and all the verilog netlists generated by synopsys whose names start with `fromsynopsys_*_.v`. Now you are ready to run the **Backward Annotation**. This is achieved by executing the `fill_forward.scr` script. Before running it, let's analyze it.

```
# Defines DATA TYPE
do datatype.def

# Loads the technological library and the SDFs
vsim -voptargs=+acc -L /software/dk/nangate45/verilog/msim6.5c -sdftyp /testbench/UBUSNORM=busnorma

# Generates the VCD file and add all the DUT signals
vcd file frommentor_$DATATYPE.vcd
vcd add /testbench/UBUSNORM/*
vcd add /testbench/UBUSINV/*
vcd add /testbench/UBUSTRAN/*
vcd add /testbench/UBUSGRAY/*

# runs the simulation
run 100us
```

First it defines the DATATYPE variable in **datatype.def**, then it loads the testbench for simulation, includes the cell library for annotation, specifies the SDF files for each instance, generates the VCD file adding all instances and run the simulation.
Make sure that you have named the component instantiations with the same convention used in the script, otherwise the backannotation won't run. From the Modelsim command window type,

do fill_forward.scr

and wait until the simulation stops. The script should have created one VCD file in the same `ese4` directory. You can now convert this file into the SAIF file required by synopsys for the backannotation. Open another shell within the `ese4/synth` folder, run `setsynopsys` and run:

vcd2saif -64 -input ../frommentor_ADDRESS.vcd -output ../backward.saif

If we analyze the content of these files we realize that that Modelsim has modified the properties of each signal by appending other information. Let's refer to the SAIF file `frommentor_ADDRESS.saif`,

```
(SAIFILE
(SAIFVERSION "2.0")
(DIRECTION "backward")
(DESIGN )
(DATE "Fri Apr 26 11:19:22 2019")
(VENDOR "Synopsys, Inc")
(PROGRAM_NAME "vcd2saif")
(VERSION "F-2011.09-SP3")
(DIVIDER / )
(TIMESCALE 1 ps)
(DURATION 99)
(INSTANCE testbench
```

```
  (INSTANCE UBUSGRAY
    (NET
      (A\[0\]
        (T0 49988100) (T1 50007000) (TX 0)
        (TC 10000) (IG 0)
      )
      (A\[1\]
        (T0 49988100) (T1 50007000) (TX 0)
        (TC 5000) (IG 0)
      )
      (ck
        (T0 50000000) (T1 50000000) (TX 0)
        (TC 20000) (IG 0)
      )
```

Note that each signal has new properties called $T0$, $T1$, TX and TC. The quantities $T0$ and $T1$ represent the time (measured in timescales) spent by the signal at the '0' and '1' state, respectively. The quantity TX represents the time spent in the undefined state and finally TC represents the number of the 0-to-1 and 1-to-0 transitions within the simulation time. For the **ck** signal, since duty cycle is 50% the quantities $T0$ and $T1$ are equal. Moreover the TC of 20000 suggests us that the clock frequency is 100MHz because within 100000000 ps, that is 100000 ns the clock toggles twice per period (10 ns): 100000 ns/10 ns=10000 periods, $TC = 10000 \cdot 2 = 20000$.

Now we are ready to estimate power consumption by running the script `backward_all.scr` in the Design Vision command window. After having opened and analyzed in detail the script, go back to the Design Vision command window and invoke it by typing

source backward_all.scr

Considering that this process may take a while, let's see what's going on. You may find out that most of the work is carried out by the script **backward.scr**,

```
# Creates the clock
create_clock -name "clock" -period $period {ck}

# Reads data from backward.saif file...
# THE CLOCK IN THE TESTBENCH MUST BE THE SAME AS THE CLOCK 'CK'
# DECLARED UNDER DESIGN VISION!!
read_saif -input ../backward.saif -instance testbench/$BUSTYPE -scale 1 -unit_base ns

variable loadset 1
source save_power_report.scr

set_load $load1 "B"
compile -exact_map
set loadset $load1
source save_power_report.scr

set_load $load2 "B"
compile -exact_map
set loadset $load2
source save_power_report.scr
```

```
set_load $load3 "B"
compile -exact_map
set loadset $load3
source save_power_report.scr
```

The script is fully parametric to allow the maximum reuse. First, the clock is created. Then, with the command `read_saif` the script reads data in the SAIF files and annotates activities. For file name generation purposes a new variable `loadset` is defined, then `save_power_report.scr` is invoked to store the power report. Then the script imposes new load (`$load1`) to bus "B" and after having forced recompilation it saves the power report once again in a new file. This iteration is repeated for all the loads defined in the script `definitions.scr`. When this process has completed you should find 16 new files in the `synth` directory, 4 for each bus encoder. By opening each file one by one you can compare the power results. **If you want you can use the linux program "diff" to see the differences. Syntax:** `diff <file1> <file2>`**.**

Now you should understand how to obtain the power report in the case of process **DATA**. After having commented **ADDRESS** in the verilog testbench, uncommented the **DATA** macro and recompiled `tb_encdec.v`, change the variable set in `datatype.def` from ADDRESS to DATA. This allows to identify the files differently with respect to the ADDRESS case. Repeat all the steps to obtain automatically a power report in the case of DATA input probabilities. To better compare the obtained results, you can fill the following table (the T0 case is left as homework).

POWER	ADD				DATA			
[uW]	1f	10f	50f	100f	1f	10f	50f	100f
BUSNORMAL								
BUSINVERT								
TRANSITION-BASED								
GRAY								
T0								

Remember that if you want to see the differences among the five syntheses you can execute each line script separately in the command window and navigate for appreciating the resulting schematic. Now try to find out how you can modify these scripts in such a way that they synthesize and perform SAIF annotation also in the T0 case.

Remark point

How to force a switching activity on a bus; how to synthesize using scripting. Recognizing the impact on power of capacitance and the logic gate of the encoder and the decoder. T0 bus encoding with power reports and comparison with the other bus encoding techniques.