LAB TUTORIAL – VLSI DESIGN FLOW: FRONT-END

Original: F. Campi, Adapted By: A. Tino

Objective: This tutorial will guide you through all the EDA material that is necessary to build a small CMOS digital macro which performs a simple DSP task. This tutorial is intended as a step-by-step guide for the lab environment and various tools we will be using throughout the semester.

Background: As a design reference, we will use a very simple image processing hardware module, referred to as rgb2grayscale conversion. There are various methods to perform rgb2grayscale. We will be using the lightness method, which is a fairly straightforward hardware implementation. The lightness method averages the most and least prominent colours:

Gray pixel = (max(R, G, B) + min(R, G, B))/2

We will be using 8-bits to represent each pixel.

An example application for the lightness method is a processor which receives data from a digital camera, and performs a security application, such a motion detection or object recognition, that would typically be performed on grayscale 8-bit images. In the case where many cameras work concurrently, it is necessary to design a single fast and low cost integrated circuit (IC) which could be reused, instead of integrating a (big) FPGA into the system. Depending on how fast and efficient we would like the rgb2grayscale conversion to be, we could also select an older, yet cheaper technology node such as CMOS 180nm, or a faster but more costly technology node such as 45nm.

Front-End Design

A) Setting up the environment

The **front-end design** implies the process of HDL coding, simulation (verification), deriving a constraints definition with synthesis (in this case with two different technologies), and evaluating the results.

In order to build our working environment, we first need to login to our Unix machine with the credentials provided to you. Thereafter, open a terminal which we will reserve for working on our front-end design (FE): Right-click the background of your screen, and select the option “Open in Terminal”.

Change your directory to your selected working area, and make a directory called **ensc450**, and a subdirectory called **rgb2grayscale**, i.e.

```
mkdir ensc450

cd ensc450

mkdir rgb2grayscale

cd rgb2grayscale
```

In the rgb2grayscale directory, we will create the following subfolders:

```
mkdir vhdl

mkdir sim

mkdir syn_045
```

We will now use ModelSim to simulate our HDL, and Synopsys dc_shell to synthesize the design. To use these tools, we must first setup the environment and required licenses with the command:

```
source /etc/profile.d/ensc-cmc.csh

source /CMC/setups/CDS_setup.csh

source /CMC/setups/FE_setup.csh
```

Important note: This source commands above are used to properly configure your terminal for the EDA tools. **If you start a new session, or even if you simply open a new terminal window, *you will have to run these source commands once again***. Therefore it is wise to use a different terminal for different purposes, and the same terminal for the same purpose. You may set a title to your terminal (Terminal -> Set Title) to keep track of windows.

We will now start investigating our HDL solution and performing a functional simulation to verify correctness of the design. Thereafter once verified, we will synthesize the design using two different technology nodes.

I. HDL Coding and Simulation

Our hardware is not a standalone IC, rather it is a digital macro that can be integrated in a more complex design. Since it is a component within another design, we do not know the delay at which we will receive the inputs. Therefore rather than using a "set_input_delay" statement which will impact our speed, we will sample the input data as soon as it arrives in the macro (and again before the new data exits). You may find the code /CMC/setups/ensc450/rgb2grayscale/vhdl/rgb2gray.vhd . **Understand the logic of the VHDL design before proceeding.**

To type the VHDL code, use a text editor of your choice. The most common editors are vim (which used your terminal), emacs, Gedit (both graphic tools that leave your terminal free). If you use emacs or gedit, make sure you invoke the command as a "background" process by using '&' i.e. "gedit & ". The '&' will ensure that your terminal does not get stuck waiting for the editor to terminate before proceeding.

Testbench: In order to simulate our code, we will require a testbench, which as we recall from the lecture, is non-synthesizable VHDL. The testbench will feed test inputs to our macro, which we may verify correct functionality with the waveforms generated.

You can find a sample testbench for our module in /CMC/setups/ensc450/rgb2grayscale/vhdl/tb_rgb2gray.vhd. To simulate the correctness of the VHDL, we will use **ModelSim** (a state-of-the-art simulator tool). Since we don't know the exact timings for the module prior to technology mapping, we will use a conventional 20ns period. We could use any other period, however in RTL simulation the time is not significant since we don't know the precise delay of the cells. That is, we are verifying correct functionality at this stage.

ModelSim needs a configuration file "modelsim.ini" that contains simulator settings. You can utilize the initialization file provided in the reference folder. To do this, ensuring you are in your rgb2grayscale directory:

```
cd sim

cp /CMC/setups/ensc450/rgb2grayscale/sim/modelsim.ini  .

mkdir scripts

gedit scripts/compile.csh &
```

When you are dealing with an HDL project with many files, it is necessary to build a custom script for compiling files in the correct order. An example of a compile script can be found in path /CMC/setups/ensc450/rgb2grayscale/sim/scripts/compile.csh. You can copy the contents of compile.csh to your script.

Ensure you have finished typing the VHDL files (including the testbench), and that they are present in your ../vhdl folder. Run the following commands, ensuring you are located in the "sim" folder:

```
chmod +x scripts/compile.csh

./scripts/compile.csh
```

Once the compilation does not signal any error, we can proceed to simulation. A sample script may be found at/CMC/setups/ensc450/rgb2grayscale/sim/scripts/simulate.tcl. Create a new file in your directory under sim/scripts and name it simulate.tcl. You may copy the contents of the template provided.

To simulate our design, we need to first open Modelsim and the design environment (ensure you are in the sim folder) by entering the command

```
vsim E &
```

Wait for the simulator to open. Once open, go the command window (window titled "Transcript") and run:

```
source scripts/simulate.tcl
```

A waveform window will open. Maximize the window and zoom in on the waveforms to analyze the inputs and outputs. Notice that the delay between the inputs and the corresponding output are caused by input and output registers (as seen in the vhdl file). By inserting these registers, we do not add to the critical path delay. Rather, the delay of the inputs from the preceding blocks and the processing of the following block's output allow our IC logic to work faster (i.e. this creates a form of pipelining). We refer to this process of computing more data per unit time as **throughput**. By increasing throughput however, we may also increase the time between the time the input arrives, to the time the result is produced at the output, which we refer to as **latency**.

You may exit Modelsim once you have a thorough understanding of the rgb2gray design, and the tool.

II. Synthesis

For logic synthesis, we will utilize the Synopsys design compiler through its built-in command shell dc_shell-xg-t. We could also utilize the GUI version, design_vision, however utilizing scripts allows for better documentation and control of the design process. We may use design_vision depicting our synthesized design and critical path graphically, although it may be preferable to view the critical path instead using the netlist.

Synopsys design compiler requires two configuration files, .synopsys_vss.setup and .synopsys_dc.setup. In our context, we will only utilize the first file, which you may copy from our reference tutorial folder. Let's create our synthesis workspace:

Change directories (cd ..) back to your rgb2grayscale folder. Once there, type in the following commands:

```
cd syn_045
mkdir scripts
mkdir results
mkdir work
mkdir logs
```

In the folder 'scripts', we will build the synthesis script called synth.tcl which is required to run Synopsys' dc_shell. A reference script can be found in the tutorial folder location /CMC/setups/ensc450/rgb2grayscale/syn_045/scripts/synth.tcl. Feel free to use this file to create your script. It is important however that you carefully understand the script (and other scripts provided) since you will require them in the future for your own design. The synth.tcl script contains constraints that must be adapted to your specific design environment, from the synthesis step, to Place&Route, to final design checks.

Once you have created your scripts/synth.tcl file, now copy the synopsys configuration file that ensures that Synopsys dc_shell will use the "work" folder to place any temporary files. To do this, cd to your syn_045 directory, then type:

```
cp /CMC/setups/ensc450/rgb2grayscale/syn_045/.synopsys_vss.setup .
```

Now we can proceed to our synthesis process using dc_shell. Ensure you are in your syn_045 folder, type:

```
dc_shell-xg-t -f scripts/synth.tcl
```

You should see a report generate on the terminal with no errors (1 net warning). In order to understand the design tradeoffs, we need to understand the best achievable performance, power and area penalty for the design. Therefore we would like to analyze: 1) the power and area of an unconstrained design

(unconstrained with a higher period such as 500ns), 2) the power and area of the fastest possible design solution, and 3) A few design points between these extremes.

Synthesis results from the dc_shell are available in the "results" folder. The result folder now contains:

1. **Verilog file** representing the "netlist" of the design. Netlist signifying a list of cells which represent a technological equivalent of the behavioural code that we synthesized
2. An internal .ddc file (Synopsys database of our design)
3. A sdc file – represents a set of constraints we used for synthesis
4. A .rpt file – we will use this information to evaluate our design.

The .rpt file is composed of three different report sections:

- **Area** – lists all standard cells which compose the design, specifying the area of each cell, and the overall total area. Note that area strictly depends on our technology node. Therefore if we were to synthesize using a different technology node, we would have significantly different results. *To express a design's complexity independently of the chosen technology*, we often refer to the area of the block as "kilogates", where the reference gate is a 2-input NAND gate library (see phase 1 document for NAND gate size). Therefore Area(Kg) = area(block)/area(nand2). Refer to the lab manual for numeric specifics.

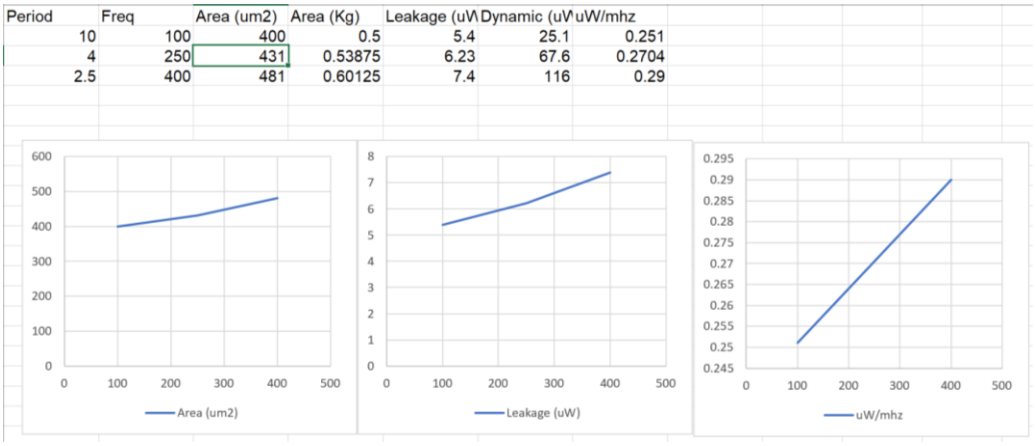
- **Timing** – defines a path, referred to as a "timing arc" between the start pin (usually our block's input pin, or a D pin of an internal FF), and the end pin (output pin of the block, or a Q pin of an internal FF). The tool calculates the path with the longest delay, and compares the delay with the clock period we specified. It will then verify if the timing constraints we imposed are met.

- **Power** – reports the power consumption for a given IC. The report consists of leakage power (power consumed by the circuit during stand-by) and dynamic power (power consumed through transistor toggling – from clock and inputs)

We normally report leakage and dynamic power separately. Dynamic power relies on the clock frequency (not the technology node), whereas leakage power depends on the technology node (not frequency). Therefore, in order to *report dynamic power independently from technology*, we often express dynamic power as mW/MHz (Dynamic power / frequency).

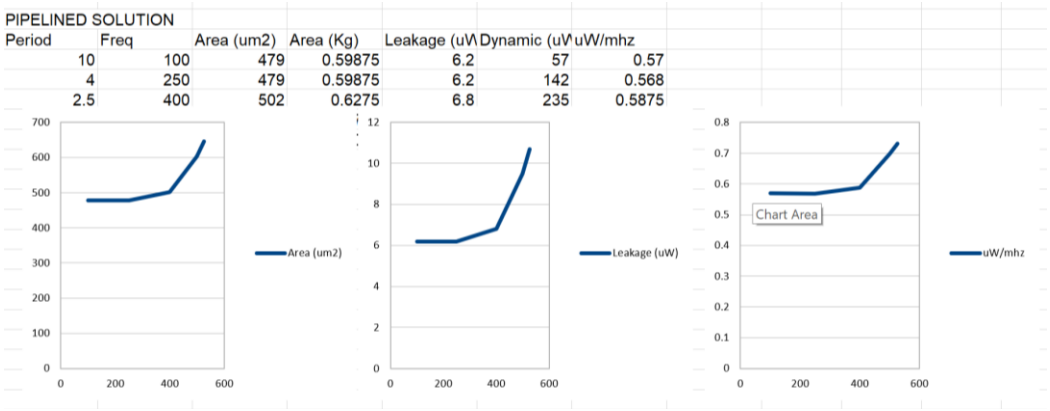
The fastest solution is the fastest design that does not generate violations, which we may obtain roughly with a few attempts. We can use Microsoft Excel, or Linux Open Office (command: ooffice&) to design graphs. An example may be found in /CMC/setups/ensc450/rgb2grayscale/syn_045/rgb2gray.ods. An example of the results obtained with cmos 45nm technology in Fig. 1 (values may differ).

During synthesis, it is possible to fix hold violations, and explore its impact on timing specifications. Since hold times have strong dependence on clock tree skew, it does not make sense to fix any hold violations at this stage. It is advisable to let the tool perform analysis on setups, and leave hold times for the following stages which requires the clock.



Period	Freq	Area (um2)	Area (Kg)	Leakage (uW	Dynamic (uW	uW/mhz
10	100	400	0.5	5.4	25.1	0.251
4	250	431	0.53875	6.23	67.6	0.2704
2.5	400	481	0.60125	7.4	116	0.29

Fig. 1: Timing, Area and Power Trade-offs (45nm) (values may differ)



PIPELINED SOLUTION

Period	Freq	Area (um2)	Area (Kg)	Leakage (uW	Dynamic (uW	uW/mhz
10	100	479	0.59875	6.2	57	0.57
4	250	479	0.59875	6.2	142	0.568
2.5	400	502	0.6275	6.8	235	0.5875

Fig. 2: Timing, Area, and Power Trade-off for Pipelined Design (values may differ)

VCD File Back-Annotation (Achieving accurate power estimations)

Two scripts exist in the folder /CMC/setups/ensc450/Power_scripts entitled generate_vcd.tcl and power_estimate.tcl. Copy (cp) the file generate_vcd.tcl into your **sim/scripts** folder, and power_estimate.tcl into your **syn_045/scripts** folder.

VCD files are ASCII-based dump files generated by EDA tools, such as ModelSim in this case. ModelSim will use the script, generate_vcd.tcl, to dump all the data in your testbench's waveform to a vcd file. The vcd file will thereafter contain all your waveform signal transitions and activity which can be assessed by a power estimation tool to provide an accurate reading on power consumption. Therefore it is important that your testbench adequately tests the system to ensure an accurate estimation is achieved.

The generated vcd file may be read in by dc_shell using the provided script power_estimate.tcl to interpret transistor activity. Note that the previous report_power command executed by dc_shell in the synth.tcl script only provided average power. Make note of these differences in the two scripts.

Change directories to the sim/ folder so that you may relaunch Modelsim in the terminal. Re-synthesize the vhdl files using the command ./scripts/compile.csh When successfully compiled, type vsim E& into the terminal. Once Modelsim has launched, go the command window ("Transcript") within Modelsim and run source scripts/generate_vcd.tcl. If successful, you will find the files rgb2gray.vcd and .saif generated in your working folder. Exit Modelsim when successful.

Change directories to syn_045/ and type dc_shell-xg-t -f scripts/power_estimate.tcl

dc_shell will launch, and a report will be generated in the results folder outlining your new power stats. Compare the average power stats obtained in the previous section to the power stats obtained with the VCD back-annotation. Ensure that the new results do not overwrite the previous results (adjust script if necessary).

III. Design optimization

If we want to achieve better performance and higher throughputs, we can attempt to divide the critical path of the design. By analyzing the timing report, we can see that the critical path of our circuit is divided as such:

1.63ns Max/Min calculation (*U* labeled stdcells)

0.7ns Final Sum and Shift (*add* labeled stdcells)

If we break the critical path in HDL, we can achieve a better result, but will pay a penalty in area and power. Let us explore this trade-off. Re-do the steps above using the pipelined version of the rgb module. You may find a pipelined version of the HDL design in the following folder: /CMC/setups/ensc450/rgb2grayscale/vhdl/rbg2gray_pip.vhd

By performing the same steps as listed above on the pipelined version, we observe that we can reach an operational frequency of 500MHz, but at the cost of an additional 20% area overhead, 30% additional leakage, and 15% dynamic power per sample (see Fig. 2). Consequently, this design choice for the product will only be significant if the 100MHz speed-up will allow us to gain a relevant market share. Costs incurred in this case will include general design costs, mask costs, and battery cost.

Important Note: Many synthesis projects demonstrate the **elbow feature in the plots**, as shown more so in Fig. 2. These diagrams demonstrate when the synthesis tool starts to adopt drastic solutions in order to meet timing constraints. Such a timing constraint will likely not remain valid when it is time to P&R the design due to wire delays and clock distribution effects. Therefore it is **advisable to avoid the elbow region**, and **remain as close to the plot's plateau as possible**. For the pipelined version, we should avoid the 526MHz solution (the 500MHz may also be a bit risky as it may cause some degradation during back-end design).

IV. Post-Synthesis Simulation

A very common error when writing HDL circuits for VLSI systems is using code which simulates, but is not synthesizable. HDL was designed to simulate and document hardware systems, not necessarily to design

them. We can design a reduced subset of HDL constraints, and apply special care when writing HDL code which we would like to synthesize. Note that FPGAs have adopted many HDL constraints that are **not valid for HDL synthesis**.

The HDL code which we develop is usually perfectly legal, but may not correspond to the functionality we require once synthesized: our HDL synthesizes and simulates correctly, but once synthesized for a VLSI system, causes undesired results that are not functionally sound. HDL synthesis tools issue errors and warning to detect such unsynthesizable constructs. A typical example is a flip-flop statement which has a sensitivity list containing all inputs – this may cause unwanted parasitic latches or produce incorrect outputs. As we learned in the lecture, only clock(s) and reset should be present in the list. Please refer to the lecture notes on synthesizable logic and VHDL.

Synthesizing Netlists: To avoid unwanted effects in our synthesizable design, it is useful to run a Modelsim simulation on your synthesized netlist. We can compile the Verilog netlist generated by design_compiler, together with a specific library Verilog file that describes the appropriate behaviour for the specified cells (available in most library instantiations, including our FreePDK 45 lib). Using this simulation, we can verify that our netlist has expected functional behaviour. This procedure may be found in /CMC/setups/ensc450/rgb2grayscale/sim/scripts/compile_netlist.csh. Open the file to observe the contents and contrast the commands with compile.csh.

We can use ModelSim to synthesize our design, using a library Verilog file (with information about the stdcells used). This library file may be found at:
/CMC/setups/ensc450/SOCLAB/NangateOpenCellLibrary_PDKv1_3_v2010_12/Front_End/Verilog/NangateOpenCellLibrary.v.

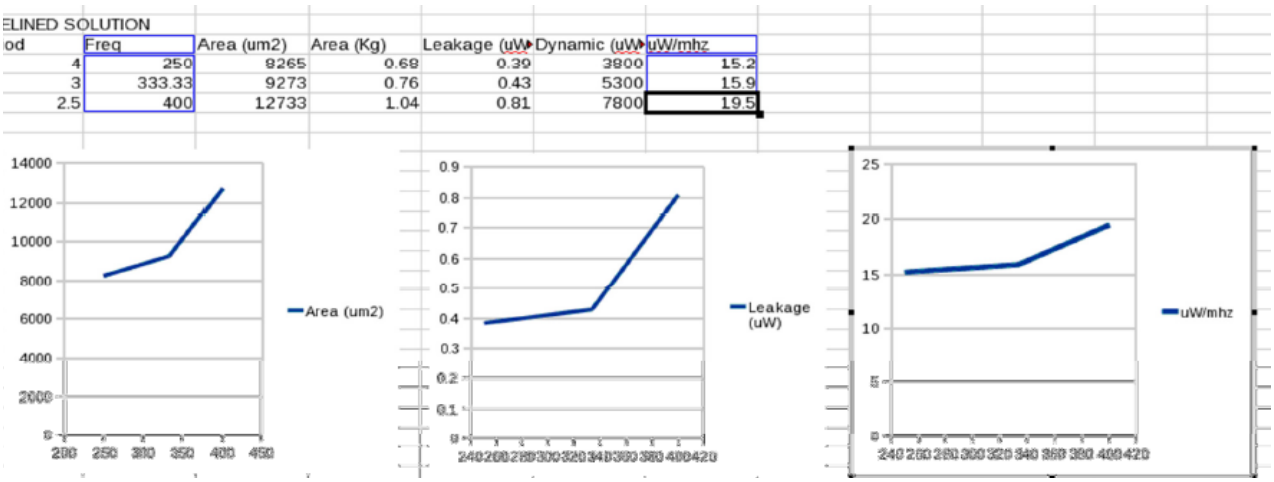
Use this library file, and your netlist file (your file generated in *syn_045/results/rgb2gray.ref.v*) to verify functionality, as we did previously for our behavioural HDL. Using ModelSim, simulate all the cells in your design with appropriate inputs and outputs by using the previous testbench, and check that the synthesized netlist simulates outputs which are the same as our behavioural simulation. **Follow the steps from Part 1 (however substituting with the _netlist.csh script etc). Copy and change the power estimation tcl scripts if necessary to compare average and accurate results from the pre-synthesis version.**

V. Trial on another technology node: cmos180nm (reading material only)

Once we have understood the limits and potential of the 45nm technology node, we may generate and observe comparable results with another technology to determine whether it is worth using a 45nm mask set, or whether we may have comparable results with **a cheaper node.** 180nm was the leading technology around 1999/2000, but is still offered by few fabs, and its mask/diffusion costs are ten times smaller than 45nm.

Let us analyze this technology trade-off with synthesis, before we proceed to our back-end design. From a scripting point-of-view, synthesizing the design does not change much – all we must change are the target specifications to link to the appropriate technology. For simplicity here, we will only consider the pipelined solution. Results are provided in Fig. 3.

As seen in Fig. 3, we do not lose much performance by using 180nm technology. The peak speed seen is 400MHz (2.5ns period) after synthesis, whereas the 45nm node's peak was 500MHz (2ns period). A 0.5ns delay per pixel is sustainable, and 45nm may therefore not be all that cost effective. When observing area and dynamic power however, we see that these values are 20x larger, while leakage is 10x less. Although 20x area may not be too dramatic, area impacts mask costs – 180nm masks cost much less than 45nm masks. The power consumption however is more critical considering battery life. We will wait until the P&R stage to complete a final comparison. (Note: You do not need to test the 180nm node)



ELINED SOLUTION

od	Freq	Area (um2)	Area (Kg)	Leakage (uW	Dynamic (uW	uW/mhz
4	250	9265	0.69	0.39	3900	15.2
3	333.33	9273	0.76	0.43	5300	15.9
2.5	400	12733	1.04	0.81	7800	19.5

Fig, 3: Timing, Area, and Power Trade-off for 180nm Technology