EECS 151/251A ASIC Lab 7:
SRAM Integration

Written by Nathan Narevsky (2014,2017) and Brian Zimmer (2014)
Modified by John Wright (2015, 2016) and Arya Reais-Parsi (2019)

Overview

In this lab, we will go over the basics of using circuits other than standard cells in VLSI designs. The most common example of this is SRAM, which is a dense addressable memory block used in most VLSI designs. The process for adding other custom, analog, or mixed signal circuits will be similar to what we use for SRAMs.

To begin this lab, get the project files by typing the following command

```
git clone /home/ff/eecs151/labs/lab7
cd lab7
```

If you have not done so already you should add the following line to your bashrc file (in your home folder) so that every time you open a new terminal you have the paths for the tools setup properly.

```
source /home/ff/eecs151/tutorials/eecs151.bashrc
```

Note: after doing this, you need to manually source your own .bashrc file or open a new terminal. You can test whether your path is setup properly or not by typing `which dc_shell` and making sure that it does not say `no dc_shell in ...`.

Differences in Files Compared to Last Lab

The files here are very similar to what we used for the previous lab, but there is one key important difference in the verilog files. If you take a look at `src/gcd.v`, specifically line 113:

```
SRAM2RW32x16 sram (
    .CE1(clk),
    .CE2(clk),
    .WEB1(web1),
    .WEB2(web2),
    .OEB1(oeb1),
    .OEB2(oeb2),
    .CSB1(csb1),
    .CSB2(csb2),
```

```
    .A1(address_1),
    .A2(address_2),
    .I1(data_in_1),
    .I2(data_in_2),
    .O1(data_out_1),
    .O2(data_out_2)
);
```

This code instantiates an SRAM module, which is a dense memory unit provided by the foundry. This module specifically is a 32x16 SRAM, which means that there are 32 entries of 16 bits. This means that there is a 5 bit address for selecting those entries. In this code one port is configured to be a write port, and the other port is configured to be a read port by setting up the input parameters for the ports. The data for the write port comes from the A_reg signal within the gcd_datapath module, and the address is just constantly looping around. This sets up a history of previous values of the A_reg signal over time, which could be read out for debugging and verification, but that is not included in this design for simplicity.

To simulate this RTL design one change has been made to the Makefile in the vcs-sim-rtl folder, shown below:

```
vsrcs = \
    $(srcdir)/gcd_control.v \
    $(srcdir)/gcd_datapath.v \
    $(srcdir)/gcd.v \
    $(srcdir)/gcd_testbench.v \
    $(srcdir)/gcd_io.v \
    $(srcdir)/gcd_scan.v \
    $(srcdir)/top.v \
    $(UCB_VLSI_HOME)/stdcells/synopsys-32nm/multi_vt/verilog/pll.v \
    $(UCB_VLSI_HOME)/stdcells/synopsys-32nm/multi_vt/verilog/io_wb.v \
    $(UCB_VLSI_HOME)/stdcells/synopsys-32nm/multi_vt/verilog/sram.v \
```

This variable is used in the simulator to include the necessary files, and by adding the last line which points to the SRAM verilog models we should be able to simulate the design. There is no need to run this right now, but just keep this in mind for your project which you will want to be able to simulate.

Differences for Design Compiler and General Setup

These SRAM designs, just like the pll and other blackboxes, are given by the foundry and specified with a set of files that we will go over in this lab. If you take a look at the Makefrag file in the base directory you should see the following code:

```
mw_ref_libs = \
  cells_rvt.mw \
```

```
  io_wb.mw \
  io_pll.mw \
  sram.mw

target_library_files = \
  saed32rvt_$(tt_corner_stdcells).db \
  saed32io_wb_tt1p05v25c_2p5v.db \
  saed32pll_tt1p05v25c_2p5v.db \
  saed32sram_tt1p05v25c.db \
```

This sets up variables that point to the Milkyway (`mw`) libraries, as well as the database library files. After running `make` in the `dc-syn` folder, it creates a `make_generated_vars.tcl` file, which will contain these variables from the Makefrag file, and then the variables that are setup in that file (MW_REFERENCE_LIB_DIRS and TARGET_LIBRARY_FILES) are then used in the scripts for design compiler. Inside design compiler, the following commands are issued that use these variables:

```
set_app_var target_library ${TARGET_LIBRARY_FILES}
set_app_var link_library "* $target_library $ADDITIONAL_LINK_LIB_FILES $synthetic_library"
```

This sets up the `$target_library` variable and adds it to the `link_library` variable within design compiler. This `link_library` variable is used when linking the design, and it will search through those files to find any modules that are not found in the verilog files that are read in.

```
set mw_reference_library ${MW_REFERENCE_LIB_DIRS}
create_mw_lib   -technology $TECH_FILE \
    -mw_reference_library $mw_reference_library \
    $mw_design_library
```

This sets up a variable called `$mw_reference_library`, and then creates a Milkyway library with the reference library set to that variable.

These are all of the changes that need to be made for Design Compiler to integrate an SRAM module, so please run the following commands and we will move on to IC Compiler:

```
cd dc-syn
make
cd ..
```

Question 1: Understanding File Structure

a) What directory are the Milkyway files for the SRAM designs located?

b) What directory are the database library files for the SRAM designs located?

c) Open up the verilog implementation of the SRAM. What are the different sizes available in this process?

Differences in IC Compiler - LEF File

Now that we are running the place and route tool, we need to know information about the physical implementation of any macros that we are including in the design. Macros that we are using include the pll, io cells, and an SRAM module. We are using Milkyway libraries for this implementation information, but we will discuss a different format that is actually human readable. This other format is called a LEF file, or Library Exchange Format, which represents the physical layout of standard cells and other cells used in the design.

Below are a few lines from the lef file for the SRAM Macro that we are using:

```
MACRO SRAM2RW32x16
  CLASS BLOCK ;
  SOURCE USER ;
  ORIGIN 0 0 ;
  SIZE 87.727 BY 61.298 ;
  SYMMETRY X Y R90 ;
```

Here are the specifications for one of the output bits, `O1[8]`:

```
PIN O1[8]
    DIRECTION OUTPUT ;
    USE SIGNAL ;
    PORT
      LAYER M5 ;
        RECT 63.5180 0.0000 63.7180 0.2000 ;
    END
    PORT
      LAYER M4 ;
        RECT 63.5180 0.0000 63.7180 0.2000 ;
    END
    PORT
      LAYER M3 ;
        RECT 63.5180 0.0000 63.7180 0.2000 ;
    END
    PORT
      LAYER M2 ;
        RECT 63.5180 0.0000 63.7180 0.2000 ;
    END
    PORT
      LAYER M1 ;
        RECT 63.5180 0.0000 63.7180 0.2000 ;
    END
    ANTENNADIFFAREA 0.11118 LAYER M3 ;
    ANTENNADIFFAREA 0.11118 LAYER M4 ;
    ANTENNADIFFAREA 0.11118 LAYER M5 ;
```

```
    ANTENNADIFFAREA 0.11118 LAYER M6 ;
    ANTENNADIFFAREA 0.11118 LAYER M7 ;
    ANTENNADIFFAREA 0.11118 LAYER M8 ;
    ANTENNADIFFAREA 0.11118 LAYER M9 ;
    ANTENNADIFFAREA 0.11118 LAYER MRDL ;
    ANTENNAPARTIALMETALAREA 0.1516 LAYER M1 ;
    ANTENNAPARTIALMETALSIDEAREA 0.1516 LAYER M1 ;
    ANTENNAPARTIALMETALAREA 0.1516 LAYER M2 ;
    ANTENNAPARTIALMETALSIDEAREA 0.1516 LAYER M2 ;
    ANTENNAPARTIALMETALAREA 0.274264 LAYER M3 ;
    ANTENNAPARTIALMETALSIDEAREA 0.274264 LAYER M3 ;
    ANTENNAPARTIALMETALAREA 0.1516 LAYER M4 ;
    ANTENNAPARTIALMETALSIDEAREA 0.1516 LAYER M4 ;
    ANTENNAPARTIALMETALAREA 0.1516 LAYER M5 ;
    ANTENNAPARTIALMETALSIDEAREA 0.1516 LAYER M5 ;
  END O1[8]
```

If you look at the rectangles set up for the ports, you should be able to quickly see that all of the coordinates are the same. This means that these metal connections are all directly on top of each other, but this is fine since they are all on different metal layers. This does mean that you can connect to this particular pin of the SRAM on any of the layers specified here. The lines with `ANTENNADIFFAREA`, `ANTENNAPARTIALMETALSIDEAREA` and `ANTENNAPARTIALMETALAREA` are specifying the area of these particular metals within the SRAM cell so that you can verify that the design will pass antenna design rules. Those particular design rules address charge that is picked up in the fabrication process on large areas of metal, and basically make sure that your layout is such that this charge will be small enough to ensure you do not blow up your devices.

If instead you look at the area of the file corresponding to the pin for VSS, you will see there are a lot of different ports. Below is a part of that section:

```
  PIN VSS
    DIRECTION INOUT ;
    USE GROUND ;
    PORT
      LAYER M5 ;
        RECT 47.4510 60.9980 47.7520 61.2980 ;
    END
    PORT
      LAYER M5 ;
        RECT 36.6510 60.9980 36.9500 61.2980 ;
    END
    PORT
      LAYER M5 ;
        RECT 41.1510 60.9980 41.4510 61.2980 ;
    END
    PORT
      LAYER M5 ;
```

```
          RECT 40.2500 60.9980 40.5500 61.2980 ;
       END
       PORT
         LAYER M5 ;
           RECT 42.0500 60.9980 42.3490 61.2980 ;
       END
       PORT
         LAYER M5 ;
           RECT 43.8500 60.9980 44.1500 61.2980 ;
       END
       PORT
         LAYER M5 ;
           RECT 49.2500 60.9980 49.5500 61.2980 ;
       END
       PORT
         LAYER M5 ;
           RECT 45.6510 60.9980 45.9510 61.2980 ;
       END
       PORT
         LAYER M5 ;
           RECT 44.7500 60.9980 45.0500 61.2980 ;
       END
             .
             .
             .
    END
```

Only part of this section is included, since there are a ton of different connections to the `VSS` port within the SRAM module. This makes connecting to this port not only possible, but possible in a way that will still create a low resistance path to make sure there is a small amount of IR drop even inside of the SRAM cell itself.

There is an entry in the LEF file for each pin of the design, as well as a section at the bottom for obstructions. In this case, obstructions are sets of metal routing contained internally to the macro, to make sure that the tools do not create shorts in trying to route over/through the macros. Part of that section is shown below:

```
  OBS
    LAYER M1 ;
      RECT 0.8000 43.2180 86.9270 50.6300 ;
      RECT 0.8000 43.2180 86.9270 50.6300 ;
      RECT 0.8000 43.2180 86.9270 50.6300 ;
      RECT 0.8000 50.5070 86.9270 50.5350 ;
      RECT 0.8000 43.2180 86.9270 50.6300 ;
         .
         .
         .
```

```
    LAYER PO ;
      RECT 0.0000 0.0000 87.7270 61.2980 ;
    LAYER M3 ;
      RECT 0.9000 0.9000 86.8270 61.2980 ;
      RECT 0.9000 0.9000 86.8270 61.2980 ;
      RECT 0.9000 0.9000 86.8270 61.2980 ;
      RECT 0.9000 0.9000 86.8270 61.2980 ;
         .
         .
         .

  END
```

There are entries for each of the metal layers contained within the macro, and hopefully specify all of the connections. You can also manually create blockages within IC Compiler over the area of the macro if you are worried about the obstructions being set up properly. This can be achieved using the following command:

```
create_routing_blockage -bbox {{x1 y1} {x2 y2}} \
-layers [get_layers -include_system -filter {name=~metal*Blockage}]
```

where x1 y1, and x2 y2 are the lower left and upper right corners of the bounding box for the blockage.

Differences in IC Compiler - Placement

If we take a look at `floorplan/floorplan.tcl` there are a few changes for placing the macro cells. Previously, the `create_fp_placement` command places and legalizes the placement of any macros in the design. This is fine if you have a small number of macros and do not care that much about the actual placement, but there are also other commands to fix the placement in either an absolute or a relative sense. What this means is that you can specify exact coordinates of where you want the macros to be, or you can specify that two macros are within a certain distance of each other.

```
create_bounds -name bound_pll -coordinate {310 355 360 470} \
-exclusive {gcd/pll} -cycle_color -type hard

create_bounds -name bound_sram -coordinate {482 500 570 570 } \
-exclusive {gcd/sram} -cycle_color -type hard

create_fp_placement
set_dont_touch_placement [all_macro_cells]
```

The result of these commands is to place the pll on the left side, and the SRAM module in the top right corner. A picture of the result is shown below:
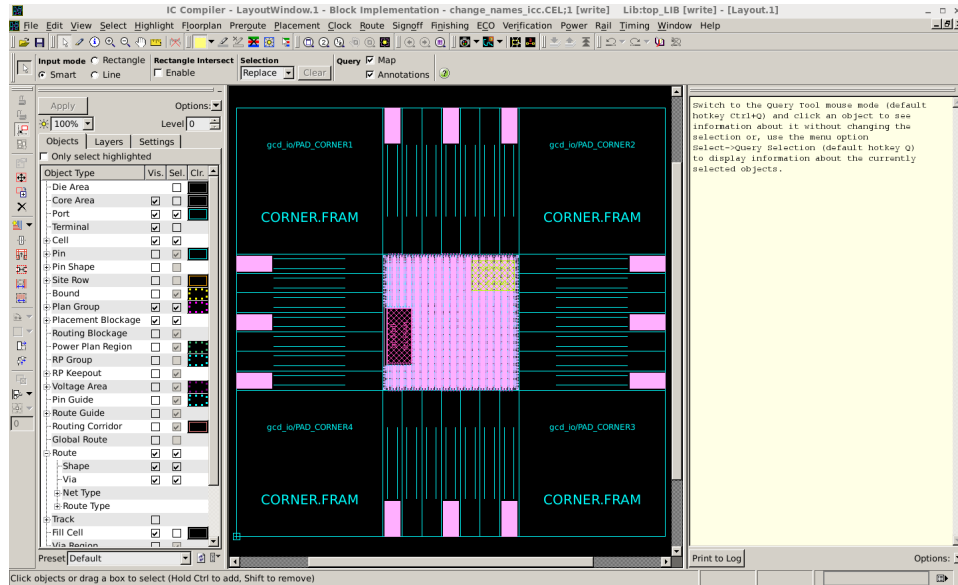
Run the design through ICC using the following commands:

```
cd icc-par
make
```

And load up the gui to see the placement using the following commands:

```
cd current-icc
./start_gui
```

You should see something similar to the picture above. This placement method uses absolute coordinates, so with the design open take note of coordinates of the core area of the design, as well as sizes of the macros.


Question 2: Changing Absolute Placement

a) Suppose we wanted to change the position of the macros such that the pll is in the top left of the core area and the SRAM is in the bottom right of the core area. What would the resulting commands be for the `create_bounds` commands?


We can also specify the placement of macros relative to each other with the following command instead:

```
create_bounds -name bound_rel -dimension {100 400} {gcd/pll gcd/sram}
```

This will make sure that the macros are within 100um of each other in the x dimension, and 400um in the y dimension. The result of this is the following (hiding the metal layers to see the SRAM easier):
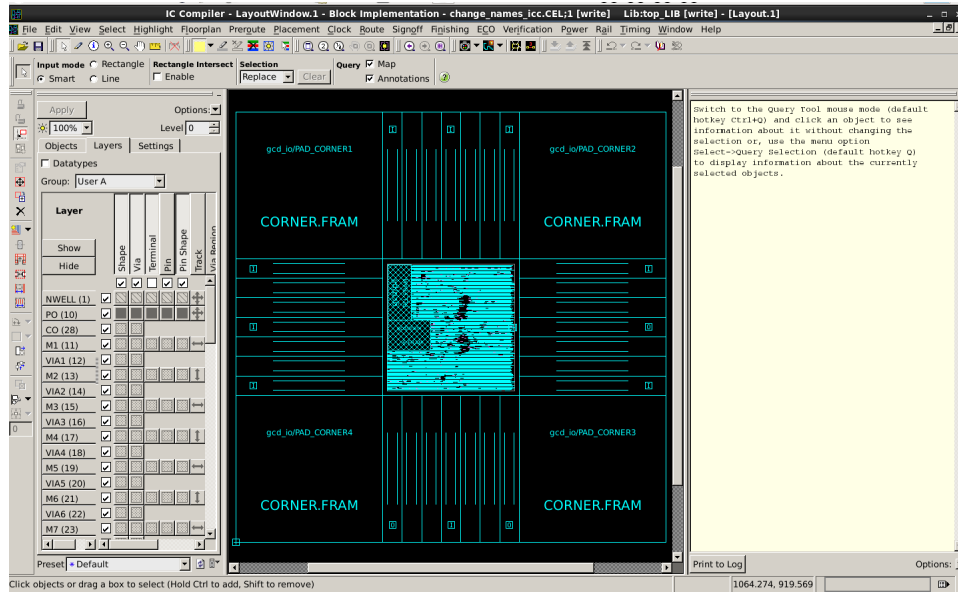
And zoomed in further to show the macros, again without showing the metal:
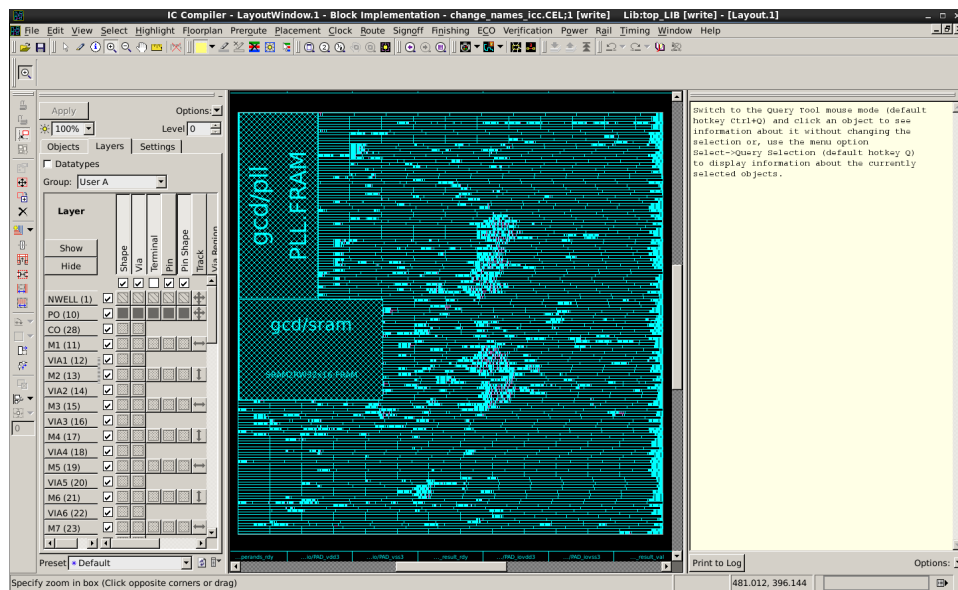
There is another way to specify simply exactly where you want the macro to go. To do that, you can use the following syntax:

```
move_objects {gcd/pll} -to [list llx lly]
set_attribute -quiet {gcd/pll} is_placed true
set_attribute -quiet {gcd/pll} is_fixed true
set_attribute -quiet {gcd/pll} is_soft_fixed true
set_attribute -quiet {gcd/pll} eco_state eco_reset
```

The first command will move the macro to the specified spot, and the rest of the lines will tell the tool not to touch it in any of the remaining steps. In this command, `llx` and `lly` correspond to the position of the lower left corner, setting the x and y coordinates respectively. Please note that this will make part b of the question below actually work properly. Alternatively, you can create the bounds large enough such that the keepout and the macro itself will fit inside, which is annoying to do.

Question 3: More commands

a) Sometimes you would like to place a keepout region (an area of nothing) around the edges of your macros in your design. What is the command to do this, and how do you specify the arguments?

b) Using your modified commands from the previous question, create a keepout of 5um on each edge of your macros with the pll in the top left and the SRAM in the bottom right. Specify the commands you used as well as a screenshot of the result, zoomed in on each macro to show the keepout