CMOS VLSI Design
Lab 2: Datapath Design and Verification

In this lab, you will begin designing an 8-bit MIPS processor. You will first review and simulate a Verilog model of the overall processor. You will learn about datapath design by assembling and connecting wordslices into an ALU. As with all labs, read the whole writeup thoroughly before starting to avoid surprises.

I. Verilog Model RTL Simulation

In the `/courses/e158/11/lab2` directory find `mips.sv` and `memfile.dat`. Make a subdirectory in your directory (e.g. sim). Copy these files into your subdirectory. `mips.sv` is System Verilog RTL for the 8-bit MIPS processer and `memfile.dat` contains test vectors. The processor is detailed in Chapter 1 of CMOS VLSI. The testbench for the processor is different from the previous lab.

Instead of testbench applying and asserting vectors, the external memory module exmemory loads a test program stored in `memfile.dat`. The program tests basic functionality of the processor and, if successful, writes a 7 to memory address 0x4C. testbench checks that the processor wrote the success value. The program is shown below; study it to see what it does.

```
# mipstest.asm
# 9/16/03 David Harris David_Harris@hmc.edu
#
# Test MIPS instructions.  Assumes little-endian memory was
# initialized as:
# word 16: 3
# word 17: 5
# word 18: 12

main:   #Assembly Code        effect                           Machine Code
        lb $2, 68($0)         # initialize $2 = 5              80020044
        lb $7, 64($0)         # initialize $7 = 3              80070040
        lb $3, 69($7)         # initialize $3 = 12             80e30045
        or $4, $7, $2         # $4 <= 3 or 5 = 7               00e22025
        and $5, $3, $4        # $5 <= 12 and 7 = 4             00642824
        add $5, $5, $4        # $5 <= 4 + 7 = 11               00a42820
        beq $5, $7, end       # shouldn't be taken             10a70008
        slt $6, $3, $4        # $6 <= 12 < 7 = 0               0064302a
        beq $6, $0, around    # should be taken                10c00001
        lb $5, 0($0)          # shouldn't happen               80050000
around:slt $6, $7, $2         # $6 <= 3 < 5 = 1                00e2302a
        add $7, $6, $5        # $7 <= 1 + 11 = 12              00c53820
        sub $7, $7, $2        # $7 <= 12 - 5 = 7               00e23822
        j end                 # should be taken                0800000f
        lb $7, 0($0)          # shouldn't happen               80070000
end:    sb $7, 71($2)         # write adr 76 <= 7              a0470047
        .dw 3                                                  00000003
        .dw 5                                                  00000005
        .dw 12                                                 0000000c
```

Read through the testbench and exmemory modules and memfile.dat to see how the RTL works. Compile and simulate it by invoking `sim-nc mips.sv`. You should see `Simulation completed successfully` if the RTL is working. You may wish to run the simulation with the GUI (`sim-ncg`) and watch the top level signals to observe the processor executing the program.

II. Library Organization

The MIPS processor uses an assortment of libraries: `muddlib11`, `wordlib8`, and `mips8`. `muddlib11` is the 2011 release of a simple standard cell library from Harvey Mudd College. `wordlib8` contains 8-bit wordslices used in the MIPS datapath and potentially other datapaths. `mips8` contains cells unique to the 8-bit MIPS processor.

Copy these libraries to your own directory so you have your own working versions to edit. From your ~/IC_CAD/cadence directory, enter

```
cp –r /courses/e158/11/lab2/muddlib11 .
cp –r /courses/e158/11/lab2/wordlib8 .
cp –r /courses/e158/11/lab2/mips8 .
```

Invoke the Cadence tools (cad-ncsu). Add these libraries to your library path by choosing Edit • Library Path in the Library Manager window. In a blank row at the bottom of the Library Path Editor, enter

```
muddlib11 muddlib11
```

Do the same with `wordlib8` and `mips8`. Use File • Save As… to save the new path in your `cds.lib` file (you could have also made these changes by editing the file directly). You should now be able to scroll down and see the new libraries in the Library Manager.

III. Wordslices

The Verilog and schematic contain functional units organized as 8-bit *wordslices*. This is a convenient way to group cells together. Wordslices can be connected with busses, which is much simpler than drawing eight separate wires. To see how a wordslice is created, open the 8-bit flopenr_1x_8 (flip-flops with enable and reset) schematic in wordlib8. Observe that it is formed from an array of eight flip-flops named flopenr_dp_1x <7:0> without having to draw each one. This part of the cell is called the datapath. Inputs and outputs (d<7:0> and q<7:0>) are connected to 8-bit busses. For clarity, the busses are drawn with wide wires. Note that Cadence uses angle brackets (< >) rather than square brackets ([ ]) to represent busses and arrays.

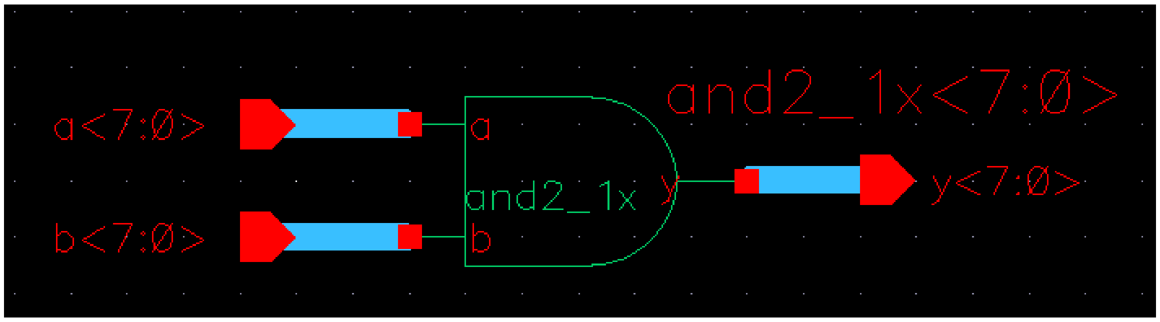
Datapath cells can factor out the inverters from select, clock, and enable signals because it is more efficient to place one inverter at the top of the datapath than one in each bit cell.

These inverters are placed in a *zipper* at the top of the wordslice so that they can drive the entire slice. flopenr_1x_8 has a zipper containing inverters and buffers factored out of the individual one-bit flopenr_1x cells. In this cell, there is a buffer and inverter to drive true and complementary versions of the enable signal, an inverter to drive reset, and a pair of inverters and buffers to drive the two-phase clocks in true and complementary form. The gates in the zipper are typically 4x normal size so that they can drive the entire wordslice in a timely fashion.

Also, look at the 8-bit adder8 schematic, which is constructed from 8 full adders. Notice how the comma notation is used for the carry in and carry out signals in the schematic. This is much easier to draw than 8 separate full adders chained together.

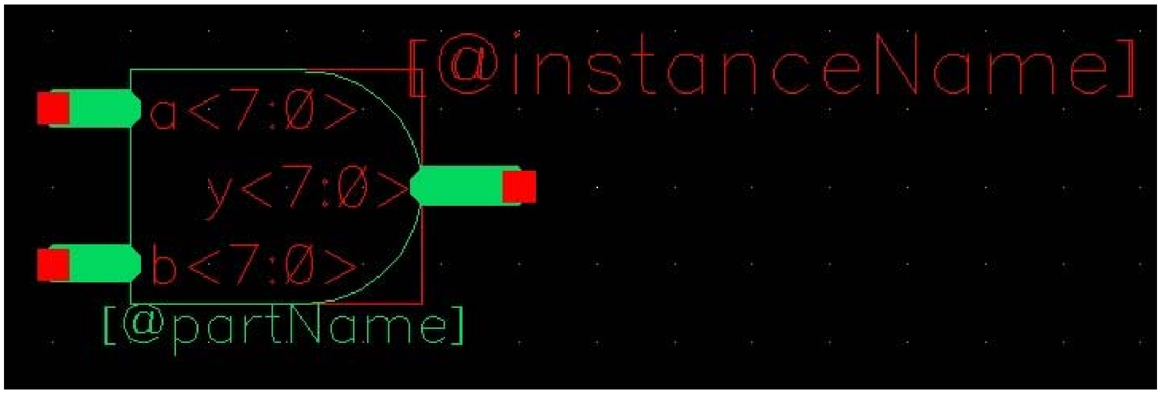
The ALU includes an AND, OR, adder, and SLT. Your first step is to design a wordslice for an 8-bit AND that will be used in the ALU unit. Later in this lab you will design an 8-bit OR and hook the two up to the ALU, and then the ALU to the datapath. The two cells you will create do not have zippers because there are no circuits to factor out.

Create a new schematic called and2_1x_8 in wordlib8. When it is all done, it should look like the one below. Unless otherwise stated, use 1x cells in wordslices.



First instantiate an and2_ 1x from muddlib11. Choose Edit • Properties • Object… and change the Instance Name to and2_1x<7:0> to create 8 copies. Create the input pins (a<7:0> and b<:0>) and output pins (y<7:0>). Choose Add • Wire (wide) and draw busses between the pins and the gate. Check and save and ensure you have no errors.

Next, create a symbol for the gate. The easiest approach is to use the Library Manager to copy the symbol from the and2_1x in muddlib11 to the and2_1x_8 in wordlib8. Ignore any complaints about the prop.xx file. Edit the new symbol. Use Edit • Properties • Object to select each of the three pins and modify its name by adding <7:0> (e.g. changing a to a<7:0>). (Make sure you are editing the pin and not just the label next to the pin.) Move the label to an attractive location. Also edit the properties of each of the lines and modify the width to wide. When you are done, the symbol should look like the one below. Check and save.

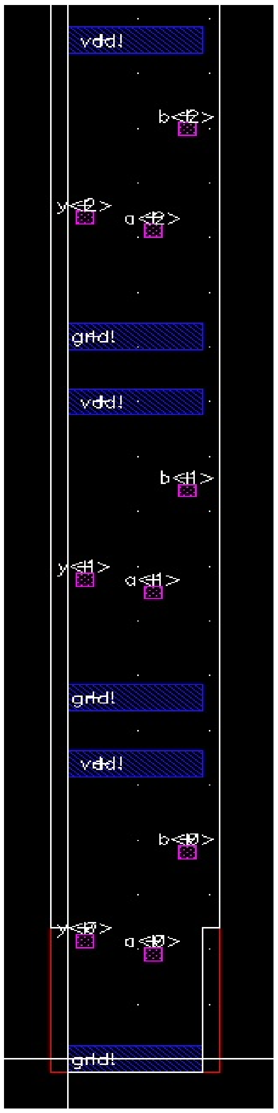


Once the schematic is finished, create the layout for the and2_1x_8. Press 'i' to add an instance. Browse to select the and2_1x layout from muddlib11. In the 'Mosaic' area, change the number of rows to 8 then click somewhere else in the window, 'View' for example, to make the change register. The 'Names' field should have automatically changed from 'I0' to 'M0'. Then change the delta Y to 33 so that each row is spaced 110 λ (33 μm) apart. Now place the instance the same way you would have placed a single cell. There should now be eight copies of the and2_1x layout that are all part of a single instance. Use Options • Display and set the Stop Levels so that you can see the contents of the wordslice.

Next create the pins. Select metal2 in the LSW. Open the create pin window (hotkey 'ctrl-p') and under Terminal Names enter 'a<0:7>'. Note that the order is reversed from usual (<7:0>) because we will be placing bit 0 at the bottom. Select something else in the window to make the change register; for instance make sure the I/O type is input and that 'Display Pin Names' is checked. The boxes next to 'X Pitch' and 'Y Pitch' should no longer be grayed out. Change the Y Pitch to 33. Make sure metal2 is selected and draw the pin over the 'a' pin on the bottom and2_1x gate. When you are placing the pin name it should display as 'a<0:7>' and once you've placed it you shoculd see the name change to just 'a<0>'. Look at the other and gates and you should see that the pins 'a<1>' through a<7> were automatically placed over the other 'a' pins.

Place the b and y pins. Make sure all the settings are correct before placing the pins (e.g. y should be an output). Immediately check that the pins were placed correctly because if they were not, or the I/O direction was wrong or the wrong metal was used, the easiest way to get rid of the pins is to 'undo' (hotkey 'u'). Otherwise you will have to manually delete each pin.

Now place a metal1 gnd! pin in the bottom row over the entire ground wire. Since ground is not a bus you will not be able to use the Y Pitch in the Create Pin. Instead you will make seven copies of the gnd! pin. Go to the CIW (command interpreter window) which is the window that first opens when starting the Cadence tools. Go to Options -> User Preferences and check the 'Option Displayed When Commands Start' box. You may not always want that option checked, however it is necessary for this step. Click on 'Apply' or 'OK' and go back to the layout. Select the gnd! pin you just created. Now copy the pin (hotkey 'c') and you should see a Copy window open. In the Copy window make sure Snap Mode is set to 'orthogonal', change Rows to 7, set Delta Y to 33, and click Apply XY twice. The gnd! pin should have been copied onto the seven other and gates. Next create the vdd! pins. When you are done the layout should look something like the figure below.



Finally, as with any layout, run DRC, Extract, and LVS. The only difference with this cell is that you will need to check the 'Join Nets With Same Name' box in both the DRC and Extract windows so that the tools know you intend to connect gnd! and vdd! later on. Be sure that you eventually do so before finishing your chip; in our case, we will provide power and ground rings in Lab 4.

IV. OR Wordslice

Now that you know how to create a wordslice, design a schematic, symbol, and layout for an 8-bit OR wordslice named or2_1x_8 using or2_1x cells. You are welcome to judiciously employ copy functions if that saves you effort. Verify that your design passes DRC and LVS.

V. ALU Assembly

Open the alu schematic in `mips8`. You'll see named buses for the inputs and outputs of the 8-bit AND/OR cells. Place and connect each. Check and save.

Next, you will complete the alu layout. Change your display options so you can see all of the cells. Study the layout until you can relate it to the schematic. You will see a space in the middle for the AND/OR wordslices. Place the AND wordslice on the left and the OR wordslice on the right. Save your work now so you can revert to this step if you mess up the wiring in the next step.

The alu already has metal3 bitlines for a, b, andresult, and orresult in each of the 8 bitslices. It also has via2s conveniently located so that you can connect the inputs and outputs of the and2_1x_8 and or2_1x_8 gates to the bitlines using vertical metal2 wires. If you've placed your two wordslices properly, all you need to do is add six vertical metal2 wires in each bitslice to connect the wordslices to the metal3. Even though a and b might seem symmetric, be sure to connect the proper bitline to the proper input or LVS will complain. If you add the six wires to the bottom bitslice, then use the copy command to create 7 more rows of the same thing, you can do this with a minimum of labor.

Verify the layout passes DRC and LVS.

VI. Datapath Assembly

Next, open the datapath schematic and layout. Study them and until you can identify the parts and understand why the layout looks as it does. Compare the schematic to Figure 1.53 of the textbook. Compare the datapath to the slice plan in Figure 1.68. Observe how the datapath uses 8 horizontal metal3 tracks per bitslice for routing (centered at 10, 20, …, 80 λ) corresponding to the 8 tracks shown in the slice plan.

Notice that the layout is surrounded by a power ring providing power and ground connections to every row. Thus, you will no longer need to Join Nets with Same Name when doing DRC and extraction, and it is better to turn this off to ensure that the power ring does properly provide power to every cell.

The alu was already part of the datapath, so now that you have added the AND and OR wordslices, the datapath should be complete. Make sure it passes DRC and LVS.

VII. Datapath Simulation

Our final goal is to simulate the datapath schematic and verify that it is correct. Because generating a good set of test vectors for just the datapath would take a good deal of thought, an easier strategy is to resimulate the entire chip with the datapath schematic replacing the behavioral Verilog model of the datapath. If the entire chip works with the schematic, then the datapath is likely correct.

Open the datapath schematic and generate a netlist in a new run directory such as `datapath_run1`. Watch for warnings. It is normal and harmless to see warnings in the multiplexers about split busses across the module ports.

Poke around the run directory. `verilog.inpfiles` contains a list of all the modules involved in the datapath. Identify where the datapath schematic has been netlisted (e.g. `ihnl/cds41/netlist`). Open this netlist in a text editor. Look at the structural Verilog that was produced by the netlister. Note the order of the inputs and outputs in the module declaration.

Then copy `mips.sv` and `memfile.dat` from where you simulated them in the first part of this lab. Our goal is to remove the versions of the datapath and its submodules from `mips.sv` and replace them with the structural netlisted files.

Open `mips.sv` in a text editor. Comment out the datapath because we will be replacing it with the real datapath.

Then look in the mips module. It instantiates a controller and a datapath. We need to replace the datapath instantiation with a new one that puts the ports in the correct order. An easy way to do this is to comment out the old datapath instantiation. Copy the datapath module declaration from the schematic netlist and paste it in, then comment it out as well. It will server as a reference to get the order correct. Then type in a new datapath call that puts the ports in the same order as they are expected in the netlist. Note that instr[5:0] must be connected to funct and instr[31:26] must be connected to op to pass these fields to the controller. When you are done, your mips module may look like the one below.

```
/* ORIGINAL datapath instantiation, wrong port order for netlist
 datapath    dp(.ph1, .ph2, .reset, .memdata, .alusrca, .memtoreg, .iord, .pcen,
                  .regwrite, .regdst, .pcsrc, .alusrcb, .irwrite, .alucontrol,
                  .zero, .op, .funct, .adr, .writedata);*/

/* Datapath module declaration from ihnl/cds41/netlist

   module datapath ( adr[7:0], instr[5:0], instr[31:26], writedata[7:0],
     zero, ALUControl[2:0], ALUSrcA, ALUSrcB[1:0], IRWrite[3:0], IorD,
     MemtoReg, PCEn, PCSrc[1:0], RegDst, RegWrite, memdata[7:0], ph1,
     ph2, reset );*/

/* New datapath instantiation with corrected port order */
   datapath    dp(adr, funct, op, writedata, zero,
                alucontrol, alusrca, alusrcb, irwrite, iord, memtoreg,
                pcen, pcsrc, regdst, regwrite, memdata, ph1, ph2, reset);
```

Invoke the simulation with the following command:

```
sim-nc mips.sv –f verilog.inpfiles
```

`mips.sv` is the first file to read. The `–f` options asks NCVerilog to also load all the **f**iles specified by `verilog.inpfiles`. If all goes well, you will get a "Simulation completed successfully" message. If not, look for compilation errors and fix them (e.g. a typo in your new datapath instantiation). If that doesn't find the error, fire up `sim-ncg`, add some key waveforms, and track down your error. It is often helpful to open two simulations simultaneously, with one showing the expected results using the golden behavioral Verilog module and the other showing the erroneous results with the actual design.

If NCVerilog fails to compile the netlist, look at the compilation report and the ihnl directory for clues. One problem that sometimes occurs is that the adder_8 netlist may be incomplete and sgarbled, as shown below. We haven't yet figured out how to reproduce or fix this problem when it occurs.

```
fulladder fulladder[7:0] ( {cout, c[6], c[5], c[4], c[3], c[2], c[1],
     c[0]}, {s[7:0],
```

VIII. What to Turn In

Please provide a hard copy of each of the following items:

1. Please indicate how many hours you spent on this lab. This will not affect your grade, but will be helpful for calibrating the workload for the future.
2. A printout of your 8-bit OR wordslice schematic and layout.
3. A printout of your ALU schematic and layout
4. What is the verification status of your layout? Do and2_1x_8, or2_1x_8, alu, and datapath all pass DRC and LVS? Does the datapath simulate correctly?