Homework 3:
Clocks, Registers, and Datapaths

Homework is due in class on the date indicated above. Late homeworks will be penalized by weighting errors on those problems that were completed past the due date by the number of days past the due date plus an extra 5% per day. Please indicate which problems, if any, took extra time. Try to start early

By the end of this assignment, you should

- Understand the operation of two-phase nonoverlapping clocks.
- Understand the floorplanning of a datapath design into "vertical" and "horizontal" cells.

1 Charge sharing

10%

In class, we discussed a way to implement a "half-latch" with an inverter and a clocked pass gate.

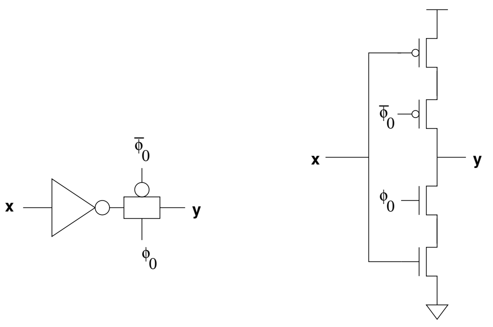

Figure 1: CS181 style "half latch."

In the textbook *Introduction to nMOS and CMOS VLSI Systems Design* by Mukherjee, this circuit appears:

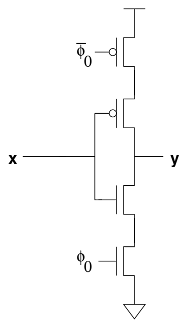

Figure 2: Clocking circuit found in the literature.

Explain why the CS181 version is better from a charge sharing point of view. (*Hint:* think about what the half-latch circuit has to do to obey the *clocking discipline.*) Support your analysis with a timing diagram. Why might one still prefer the version from Mukherjee's book?

2 Clocks, registers and datapaths

55%

You are now ready to design a substantial chunk of a chip datapath. This lab assignment will focus on designing some of the basic datapath cells that are common in CMOS designs. You designed an F-block last time, and now you will get to design *registers* that will allow you to actually compute with the F-block.

Schematic Design: First you will design your datapath in CAST and test it using `irsim`, just like in Lab 2. Once you are satisfied that the design is correct, you will go on to floorplanning and layout. The top level instantiations must be in a file called `datapath.cast`, and it must reference definitions in other files

Important: Please read through the entire assignment before beginning. When designing, keep in mind that some components can be implemented in similar ways. This will help reduce the amount of time spent in layout. Also re-read the helpful hints at the end of the assignment. Do the CAST before the layout!

i. Define a single ported register bit called `regbit`. Call the data input for writing the register `in`, the data output for reading `out`, and the control signals `w` and `r` for writing and reading, respectively; you will probably want to include `w_` and `r_` in the interface, too. Note that when `r` is not asserted, your register should *not* be driving its outputs—it should merely leave them floating so that another unit could drive them. Instantiate a `regbit` in a test file and verify it works. Do *not* try to include the staticizers in the CAST. Use the half-latch that has already been split up when you write the production rules; do not attempt to use the pass-gate version.

Note: Running `alvs` with the option -s will check that the staticizers you have left out of the CAST are implemented properly in the layout. However, `alvs` may also identify some nodes which are never left floating as requiring staticizers.

ii. Define a cell `register_control` that takes as input the `w` and `r` signals and that generates a clocked `w` signal during `phi0` true and a clocked `r` signal during `phi1` true. Use AND gates.

iii. Define an N-bit cell `register`. Its interface includes an unclocked write signal `regw`, an unclocked read signal `regr`, an array of data inputs `in`, and an array of data outputs `out`. When `regw` true and `phi0` true, the values on `in` should be stored in the register. When `regr` true and `phi1` true, `out` should be driven to the values stored in the register. Instantiate a `register` in a test file and verify that it works.

iv. Define a *loadable shift register*, `shift_register`. A loadable shift register is a unit that can hold its data (just like a normal register), write its data to its outputs (just like a normal register), or shift its data by one. In this case, implement a shift register that can shift the data *down* (i.e., towards the least significant bit. If the register holds 5 on one cycle (binary 101) and you command it to shift, it should contain 2 on the next cycle (binary 10).) When it shifts, it should not modify the value stored on the bus. You will be provided with (in fact, you will provide, via `irsim`'s command line) all four clocks: `phi0`, `phi0_`, `phi1`, and `phi1_`. (Note that there is a `phi1` phase before every `phi0` phase and that a write only occurs on `phi0`).

The shift register should have exactly the same structure as the 8-bit register you designed in the previous part, except that it should have another input `shift`. `shift` being asserted during `phi0` should cause the contents of the shift register to be shifted right by one. At the other end of the shift register (the top), make a special cell that will generate new bits for the top register cell to shift in. This cell should take an input `extend`. If `extend` is asserted on `phi0` simultaneously with a `shift`, this special unit causes the bit in the top cell to be copied back into the top cell on `phi0`. If `extend` is not asserted and `shift` is asserted, then the top bit of the register should be cleared on `phi0`.

To make the requirement for the top cell concrete, consider the following example: Shift register contains (binary) 10001100, and we assert `shift` during `phi0`. If we were also asserting `extend` during `phi0`, the shift register should eventually contain 11000110, else it should contain 01000110. (You are free to implement this function in any way you desire).

v. Define a register that will be used in conjunction with the function block, and call it `fblock_register`. The register should have a single input port, but two output ports.

vi. In order to write things into your registers from the outside world, design a very simple, inverting half register that takes an array of inputs `io`, (it consists of N four-transistor cells arrayed vertically.) Call the control wire `read` and the definition of the N-bit half register `input_register.` Make sure it obeys the bus clocking discipline—i.e., make it read from the outside world on `phi1`. The purpose of this register is to serve as a convenient way to enter data into the design in `irsim,` for instance. Or, ultimately, from the outside Real World. Keep Design Guideline 11 in mind!

vii. Initialization: when you start simulating circuits in irsim, all the nodes of the circuit carry unknown values (x). Any computation with x as one of the inputs will generate x as output. Therefore in order to produce meaningful output, you need to initialize all the nodes before you use them for the computation. For example, as can be seen from the graph below, you need to write some data from the bus into the registers before you can use them. However, for the function block (fblock) and `fblock_register` (freg), since one output of freg connects to one input of fblock, the node becomes internal. There is no way you can write data into this node from the bus. In order to set this node to a known value, you need to introduce an explicit `reset` signal which forces the output of the freg to be 0 or 1. At the beginning of the simulation, you assert the `reset` signal which drives the output of the freg to a known value. Then you write some data into fblock from the bus. Both inputs of fblock carry known values and it generates the known value to freg. In this way, the fblock and freg are properly initialized. You can deassert the `reset` signal at this moment and carry on other simulations.

viii. For the final part of this assignment, you are to instantiate what you have designed so far. Refer to the following diagram:

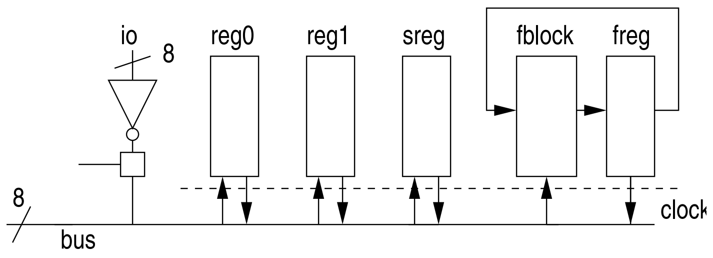

Figure 3: Lab 3 datapath design block diagram. Keep in mind the clocking strategy: The bus is driven during `phi1` and read during `phi0`.

You are to array six units horizontally, where each unit has an 8-bit datapath. Prepend the control bits with the name of the function unit, e.g., control bit `shift` in the shift register `sreg` should be called `sreg_shift` in the top level cell, control bit `g0` in the F-block should be called `fblock_g0`, and so forth. The fblock should take the `a` inputs from the bus and the `b` inputs from the freg. The following is the strict set of inputs and outputs for the datapath.

Inputs: `io0 io1 io2 io3 io4 io5 io6 io7 io_read reg0_regw reg0_regr reg1_regw reg1_regr sreg_regw sreg_regr sreg_shift sreg_extend fblock_g0 fblock_g1 fblock_g2 fblock_g3 freg_regw freg_regr phi0 phi0_ phi1 phi1_`

Outputs: `bus0 bus1 bus2 bus3 bus4 bus5 bus6 bus7`

As in Lab 2, these names are required, but you are allowed to have other names as well. It is probably simplest to use arrays as much as possible. The reason the required names are not arrays is to make it easier for you to type them into `irsim`: unless your circuit works right away, it is safe to say that the effort you spent typing in the extra names will be repaid many times over.

Now test your circuit thoroughly. Once you are absolutely certain it works, proceed to the floorplanning and layout portion.

3 Floorplanning and Layout

35%

Before you begin, make a list of the cells that you will be joining together in your design. The first step is to choose a *uniform bit pitch* for them.

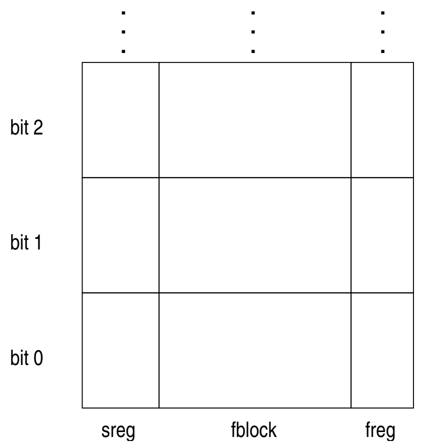

Figure 4: Example schematic fragment of a processor datapath. Notice that the height of the different kinds of cells is the same so that their corresponding bits may easily be connected together. This week's assignment does not, of course, contain any adders or multiplexors.

Plan it out and go for it.

Once LVS stops complaining, double-check the layout by running your test suite on it.

Congratulations! You have implemented most of the datapath of an 8-bit microprocessor. It doesn't do anything terribly interesting yet, but we only have the following things left:

- Control units—e.g., PLAs. (Next week.)
- Arithmetic. (Soon!)

Keep these guidelines in mind as you design the circuit:

1. Try to follow the general layout guidelines in the *CS181 Guidelines for Reasonable Layout.*
2. For this assignment, try to equalize the driving strength of the nFETs and pFETs. A width ratio of about 1:2 usually works well.
3. Choose your "bit pitch" early on! Be generous! Once you have chosen it, it is *very difficult* to change.
4. You will probably find it useful to run all horizontal signals (i.e., the data bits) predominantly on metal2 and all vertical signals (i.e., the control bits) on metal1.
5. You may assume that both senses of the clocks are available, i.e., you may freely use `phi0`, `phi0_`, `phi1`, and `phi1_`.
6. You may assume that the environment will not assert both the write and shift signals to the shift register during the same clock cycle. (Why?)
7. LVS as you work because changing layout often has wide reaching implications.

8. In the CAST, you can define global clock signals rather than aliasing them in cell interfaces. You define a global signal by declaring it in the top-level scope. This can be most easily done by creating a single globals.cast file, then importing this file wherever needed.
9. If you have trouble resetting the freg, you may change the register bit design so that instead of using a weak inverter feedback, it uses a weak NAND feedback with `Reset_`. You may postulate the existence of this global `Reset_` node, which will be **false** for a short time after power-up and then become **true** (Of course this node will also have to be routed throughout the design).
10. Keep the commands of your test suite in a separate file and use `@` in `irsim`. The `assert` command can help you more quickly locate errors.
11. **You will have the fewest problems if you design the circuit so that each node is written on exactly one clock phase (`phi0` or `phi1`) and read on exactly one clock phase.**

Requirements.

1. Run `GND` and `Vdd` in such a way that they need only be connected in one place each. Run them on metal through all your cells (metal1 or metal2 is ok).
2. The bus must be metal2, and the control wires that run through the cells must be connected on metal from top to bottom of each unit. If cannot route metal 2 across your fcell for the bus, put the fcell column all the way to the right, and move the output f from the right edge to the left edge.
3. Make the p-transistors at least 8λ wide and the n-transistors at least 4λ wide (except in weak transistors). The transistors that drive the vertical control wires should be at least *four times* that size.
4. All "dynamic" nodes must be staticized (don't forget the main bus).
5. The clocked write and read signals should be driven by large transistors since they have to power several bits.
6. Read through this list again before you turn in the assigment to make sure you did not forget anything!
7. **Question 1 and 2 are due on Oct 23nd. Question 3 is due on Oct 30th**

Useful `irsim` commands

`help`	Gives a list of commands (`::irsim::help` when run from magic).
`vector clk phi0 phi0_ phi1 phi_`	Defines clk as a vector (allows `setvector clk 0101`)
`clock clk 0101 1001 0101 0110`	Defines four phase clock
`w a`	Watch a on all phases
`c`	Cycle through all clock phases
`p`	Step one clock phase
`stepsize` *n*	Set time between clock phases to n nanoseconds (default 10)
`@` *infile*	Run infile
`assert out 00101000`	Allows you to test if you get right answer