Homework 4: The Integer Divider

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.

!!ALWAYS READ ALL INSTRUCTIONS!!

By the end of this assignment, you should

- Understand the concepts of FSMs and PLAs.
- Be able to design simple arithmetic units.
- Be able to combine control (PLAs) and datapaths into a complete design.
- Be ready for your final project!

1 A simple FSM—Parity checker *Due November 6, 2013.*

10%

This problem is intended as an FSM/PLA warmup.

You are given a serial input stream of bits with *nybble parity* attached. In other words, the bits arrive at input `in` in an infinite sequence a_0,a_1,a_2,a_3,a_4,b_0,b_1,b_2,b_3,b_4,... where the x_4 bits are the parity of the preceding four bits $((x_0 + x_1 + x_2 + x_3)\%2)$. Your assignment is to implement an FSM that checks that this is so, whose input signals are `RESET` and `input`, and its output signals are `accept` and `reject`. (You may object that you know more straightforward ways to do this, but please use a PLA for this one.)

The data inputs start on the cycle after the `RESET` signal is lowered. After five inputs have been assimilated, your circuit is to raise either `reject` or `accept` depending on whether or not a parity error was detected. (It is allowed to delay this by one cycle—i.e., it can output the decision about the first five bits after reading the sixth bit, but it should not need a one-bit "gap" in the input.) Regardless of whether or not there was an error, the circuit should go back to reading the input without preconceptions on the cycle after that.

You can design the FSM as a transition graph. In order to implement it in silicon, translate it to a `peg` program or directly to a truth table. The `peg` program can be translated to a truth table using `peg` and `eqntott`. Minimize the resultant truth table using `espresso`, and then generate the layout for the PLA (this is easy!) by typing (assuming the minimized truth table is in the file `pla.tt`):

```
mpla -I -O -s norandnor pla.tt
```

If you decide to use `peg` and `eqntott`, you might try invoking them as follows:

```
peg pla.fsm | eqntott -l | espresso > pla.tt
```

You are encouraged to look at the intermediate versions of the PLA data! Note that the truth tables outputted by eqntott and espresso are formatted slightly differently. Not all of the tools work correctly with the eqntott form.

Pop up the resultant `pla.mag` in `magic` and wire up the next-state inputs. (These are not connected by `mpla`.) In order to help you with this, look at the output of `peg`—it contains hints that will guide you in this matter. Simulate the PLA with `irsim` to ascertain that it does The Right Thing.

Please hand in:

- A state transition diagram for your FSM.
- The truth table and/or the `peg` program you used to generate it.
- A transcript of a `irsim` session showing the correct operation of your PLA.

Also leave the `pla.mag` and `pla.sim` files in the usual place.

2 An Integer Divider *Due November 6 and November 13.*

90%

Warning: This problem is a lot less structured than previous assignments. You may be required to think.

For this exercise, you are to design an *integer divider.*

Overview

You are to implement a circuit that carries out *unsigned integer division* on 8-bit data. It is asynchronously to accept as input two 8-bit numbers, one after the other, according to a handshake protocol described below. Once the data has been received, it will compute the *quotient* and *remainder.* The quotient and remainder will be driven on output wires (in that order) using another handshake. Once that is completed, the circuit will accept further input.

You may want to refer to the following diagram:

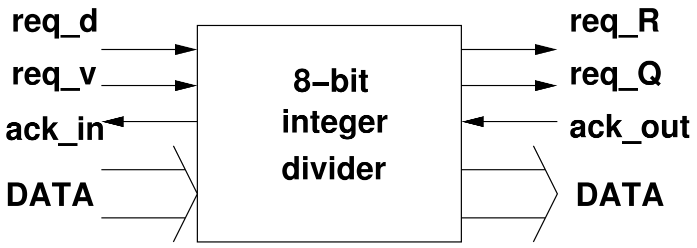

Figure 1: Lab 4 integer divider block diagram.

Communications Protocol

The protocol that your divider will use to communicate with the outside world is known as four-cycle signaling. (Or "four-phase"—in this case it refers not to the number of physical clock wires but rather to the number of phases in *time* that the protocol has.) The environment proceeds by setting the data inputs, then raises the `request` input. At this point, the environment is committed to holding the data inputs (and the `request` input) stable until your circuit responds by raising `acknowledge.` Now, the environment will lower the `request` and can change the data values. It is allowed to reassert `request` only after it has ascertained that your circuit has lowered `acknowledge`.

Your integer divider is given a divisor v and a dividend d and computes the quotient Q and remainder R such that

$$d = Qv + R.$$

Your circuit is to accept d and v in either order—thus two `request` lines, `req_d` (dividend) and `req_v` (divisor) are required on the input side of your circuit. The `ack_in` output should, however, be shared. Call the data input wires `din0` through `din7` and the data output wires `dout0` through `dout7.`

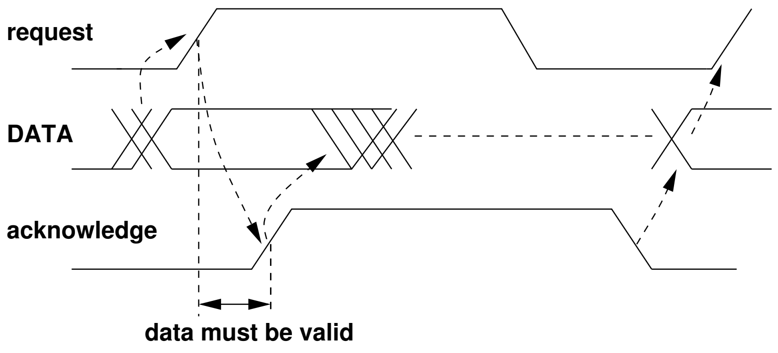

Figure 2: Asynchronous four-cycle signaling. Variations are possible. (E.g., usually one would not allow the *DATA* wires to change while `request` is high, but in this case it does not matter since the receiving circuit has already dealt with the data before raising `acknowledge.`)

On the output side of your circuit, the corresponding signals are called `req_Q` and `req_R` and are to be asserted in that order. Again, the acknowledge is shared as `ack_out.` In this case, your circuit is to obey the protocol described above for the environment.

Apart from the inputs mentioned above, you may assume the presence of the clock inputs `phi0, phi0_, phi1, phi1_,` the power supply inputs `GND` and `Vdd,` and the reset input `RESET.` No other inputs will be provided. In other words, you should not need to access any nodes besides these in order to get your IRSIM simulation to run.

Additional Tools:

- `castpla` - generates CAST for a PLA. This tool takes a minimized truth table as input

To Submit:

1. Description of algorithm (in English). *Due November 5.*
2. A simulation (source code and output) in the high-level computer language of your choice (e.g., C, C++, Python, . . .) of the division algorithm you intend to implement. This should be detailed enough to show the value of every register in your design on every clock cycle. *Due November 5.*
3. Pseudocode or `peg` code, for each PLA you need. *Due November 5.*
4. Transition graphs for each PLA. *Due November 5.*
5. Truth tables for each PLA. *Due November 5.*
6. CAST files for your design. *Due November 5.*
7. `irsim` session showing your entire divider in action using three different examples. *Due November 5.*
   i. 100 / 9 = 11, 100 % 9 = 1 ii. 255 / 6 = 43 , 255 % 6 = 3 iii. division by zero
8. The layout for each cell in the design, and the layout for the entire divider. *Due November 12.*

9. `irsim` session showing your layout for the entire divider in action using the same examples. *Due November 12.*
10. Please mention the path to the files in the materials you hand in.

Requirements:

- You must use the specified signal names.
- You are not required to return anything in particular on (or check for) division by zero, but your divider should not go into an infinite loop or fail in any other unrecoverable way.
- You must LVS the divider except PLAs, since LVS does not work with `mpla`.

Hints:

1. As you design the system, try to keep a reasonable tradeoff between software and hardware design. If you need to figure out in what bit position an 8-bit integer has its leading 1, for instance, do not try to do that with all "stock" (i.e., Lab 3) components. You can almost certainly make the control FSM/PLA much simpler by spending a small amount of time to specialize the hardware for this particular task.
2. To speed up the design process, try to spend more time on the higher level design. Use the PLA and FSM tools as much as possible.
3. When using `eqntott`, use the -l option so that the PLA ports are labeled.
4. `castpla` only works with `espresso` output and not directly with `eqntott` output.

To Help You:

Documentation for the old Berkeley tools is available as `man` pages, except for `peg`. Documentation for `peg` can be found at `http://async.caltech.edu/~cs181/docs/tools/peg.pdf`.