

# Course Plans: CS 3710: Computer Design Laboratory

## Notes for Lecture 4

### 1 Action Items

**Project Groups:** Those project groups who sent an email last week and made a webpage earned 3 (out of 100) points for the “lab checkoff” part; others can still earn 1.5 points if they do these by class-time on Thu (9/9). Also you must select a project name.

**Password Protection of Project Page:** Password-protect your project URLs. The TA can try to help you with this.

**Pass-Phrase:** Send pass-phrase to TA by class-time of 9/9, or else you won’t earn any grades for this class (because we can’t record our grades).

**Weekly Progress:** Weekly progress will be measured in the following ways:

- Hardcopy submissions if requested
- Webpage update (always expected after each major task is officially over).  
N.B.: There is no official “late policy” – however that does not mean that we won’t listen to your problems *if approached early* and make accommodations.
- 1-1 meetings. I am making available the following slots for 1-1 meetings. During September, you must meet me during one of these slots for 15 minutes every week and present the details of your work (1-1 meeting bookings are on the class webpage). Beginning October, the TA will schedule these slots:
  - Tue from 2-3pm; each group must email me which 15-min slot they wish
  - Thu from 5-6pm; (ditto)
  - Tue from 5-6pm; (ditto)

#### What’s due now:

- Thin-slice simulator: - printout showing that it works must be turned in by 5pm on 9/10 in the CS3710 drop-box. See Lecture 3’s handout. We mention “turn in assembler and simulator” which is still recommended. However, now only the thin-slice simulator is due. Show how you loaded the memory with the bit-pattern specified in Lecture 3’s handout.

You may either create an ASCII string of 0’s and 1’s in a file as in Lecture 1’s handout or follow the **MCS format**. For the latter, **kindly go through** <http://www.cs.utah.edu/classes/cs3710/kalla-directory/sud-pres.pdf> to know the exact definition of the MCS format.

- Fibonacci circuit simulation: A demo will be given in class today. You must demo the simulation as well as download and show things in a lab setting on 9/14 during the class-time. Note: the lab will be crowded; you are expected to be working on the project right now, and come to the lab only to download and demo. The TA and I will be checking off the demos. Here is additional info on what you must demo: you must build the Fibonacci circuit as will be taught today. Then you must bring-in the `xsa_display` module and create a 7-segment output of the Fibonacci sequence. You must then single-step the Fibonacci generator forward using `s5` and `s4` as explained in Lecture 3’s handout.

You'll also be asked whether you got the pet circuit compiled and downloaded. For those who did not take my CS-3700, a lot of details of `pet` will be new, and the TA will go over the circuit with you.

## 2 FAQ and more Q's on the simulator

It is best to write a 2-pass assembler where the first pass simply collects labels and enters them into a symbol table. The second pass can then work with the benefit of the symbol table. During the second pass of the assembler, for each opcode, emit the correct operands. For a branch, compute the offset. We assume an MCS-format output from the assembler.

The simulator reads the MCS records and initializes the memory of the simulated machine. For each instruction, it calculates the state-change effects and prints all the state elements. Writing the simulator helps you understand the ISA and also debug your assembly programs.

Additional questions given to me recently:

## 3 Hardware Design

In addition to getting the `petsdram.vhd` working, the recommended tasks for you are: test out `lec2/test_mixed`, and `lec3/testbram`. The latter is subsumed by the first file below; the remaining files in `lec4` must also be studied:

- The project in `ramb4_s16_s16`: simulation of a 16-bit Block RAM. Note “strange behavior” if test-bench clock is an ISE-supplied clock (RAM output is delayed by a cycle) as opposed to an explicitly supplied clock (behavior matches spec). We do not understand this strange behavior and attribute it to a simulator bug (i.e. the hardware should work OK).
- The project in directory `fib1` shows how I built the Fibonacci circuit step-by-step.
  - First, I built the schematic `regfile` that tested out the 2-ported block RAM.
  - Then I built the schematic `fib1` that tested the 2-ported block RAM operating together with a controller.
  - Finally I put it all together. In this process, many detailed skills are needed, including designing a clocking scheme for mixed flip-flop / latch circuits. I describe some tricks below. Following the design of the clocking scheme, I built the controller in Verilog.

## 4 Clocking Schemes

Usually we like to design in a 'pure' style of either flip-flop only circuits or latch-only circuits. However, mixed latch/flip-flop circuits are sometimes encountered, as are mixed asynchronous (handshaking) interfaces along with latches and flip-flops. How do we think about clock scheduling in these contexts? Here is where a class such as this allows you to explore some of these details that you may not have previously studied.

### 4.1 Purely flip-flop based circuits

Here, the design can be neatly partitioned into next-state updates and output updates.

Present-state variables, say  $S, S1, S2, \dots$

Next-state variables, say  $S', S1', S2', \dots$

Input variables, say  $I, I1, I2, \dots$

You may have assign output =  $f(S, S1, \dots, I, I1, \dots)$   
for some function  $f$

$S' = g(S, S1, \dots, I, I1, \dots)$   
for some function  $g$

Then we have  $S \leftarrow S'$  happen all at-once, typically when a clock-edge comes

## 4.2 Purely latch-based circuits

Here, the design can be neatly partitioned into state variables

State variables, say  $S, S1, S2, \dots$

Input variables, say  $I, I1, I2, \dots$

You may have assign output =  $f(S, S1, \dots, I, I1, \dots)$   
for some function  $f$

$S2 = g(S, S1, \dots, I, I1, \dots)$   
for some function  $g$

...

The trick is to have assignments that are ‘‘non-circular’’, for example suppose we have four state-variables  $S, S1, S2$ , and  $S3$ . Then

$$S2 = h(S, I)$$

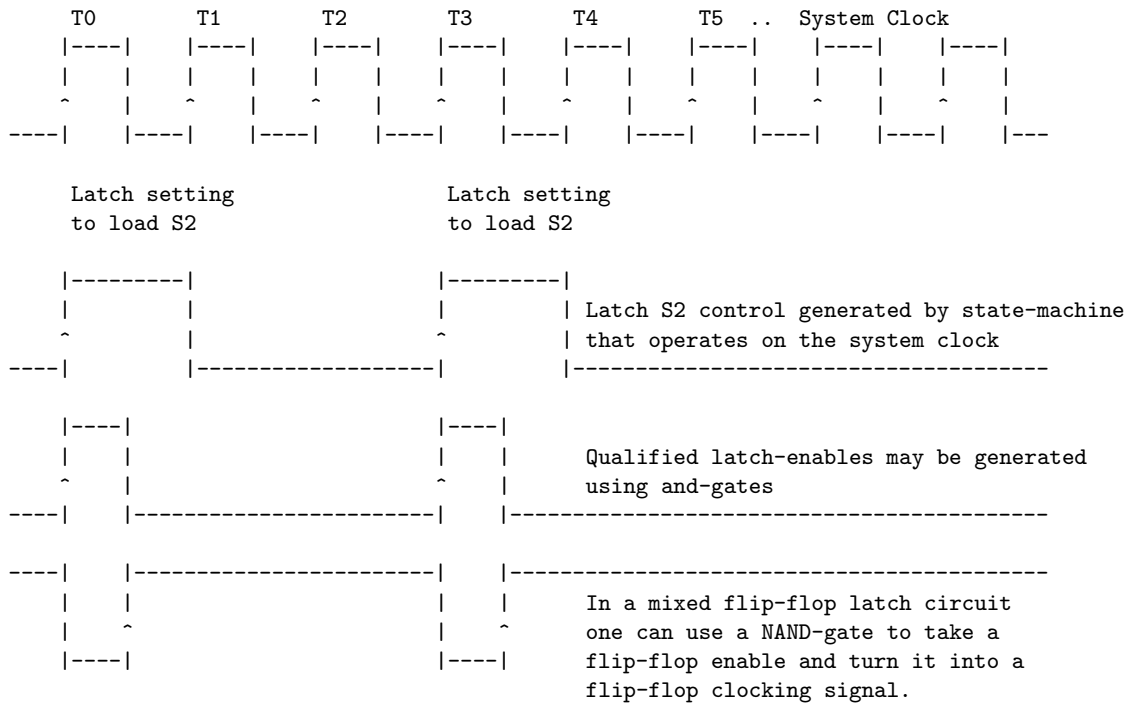
$$S1 = k(S3, I)$$

Now, the trick is to enable only latches  $S1$  and  $S2$  for updation during the current cycle. Here are the waveforms you can generate (speaking about  $S2$  specifically)

## 4.3 Mixed-style timings

The waveforms to follow show how to generate qualified latch-loadings and qualified flip-flop clockings. Async interfaces are easy: let the state-machine generate request and hang on ack. Four-phase is the normal, but you can do a pulse-mode async meaning request for one state time and ack for one state time always (explain in class) - this way the ‘‘return to zero’’ is implicitly done after one cycle... for highly robust interfaces, do a full 4-phase handshake though.

Do **NOT** put arbitrary combinational logic in the clock path, as hazards can multiply clock the latch / flip-flop. Debug your clock design using suitable VHDL / Verilog simulations.



What I do for the fibonacci circuit is roughly the following

=== one cycle ===

```
latches <- memory_read_using_rising_edge_clock
```

=== next cycle ===

```
memory_write_using_rising_edge_clock <- latches
```

=== ... ===

The waveforms are

