EECS 303: Advanced Digital Logic Design
Lab Three - Sequential system design

Robert Dick

Assigned:	4 Nov
Due:	11 Nov

1 Introduction

In this laboratory assignment, you will design an optimized sequential system to repair a damaged video display interface. Laboratory assignments one and two exposed you to the Mentor Graphics tools, Espresso, and SIS. Now you'll be using what you learned to solve a real (but slightly simplified) engineering problem I once encountered during the design of an embedded system. I haven't restated explanations already given in labs one and two. If you feel lost at some point in the lab, the additional details you are looking for are probably in the handouts for the previous lab assignments.

2 Video controller repair

You are a member of a team designing a medical embedded system that will be used to display information about the condition of patient's heart. After many weeks of work, your team has built a prototype and scheduled a demonstration for your company's investors. That demonstration is scheduled for tomorrow and investors are flying in from other cities to attend it.

Today, when putting the finishing touches on the prototype, you made a small mistake in a wiring diagram and another team member made another small mistake when following the diagram. These two mistakes, combined, resulted in an output of your embedded system's video controller chip being connected directly to V_{SS} with a wire. As soon as you noticed it, you fixed the wiring problem. However, the damage was already done.

Now your prototype sort of works but an apparently random 30% of the pixels on the screen hold apparently random incorrect colors. If this isn't fixed by tomorrow, the demonstration will be humiliating.

Your video controller chip is an off-the-shelf part. However, it will take five days to get another one from the supplier. You can't use another type of video controller chip because weeks of code rely on the quirks and features of this chip. The damaged output pin was used to request that your central processor prolong its assertion of data values until the video controller memory was finished writing data. Now, some of the time, the video controller finishes in the minimum data transfer window. However, sometimes, that window needs to be extended and the video controller has no way of doing so. As a result, some of the video memory locations end up holding garbage.

There is no way to get at the internal signals of the video controller chip. However, there is another option. You have used a logic analyzer to monitor writes to video memory and have found that the video controller never requires more than three bus clock cycles of data assertion to finish writing to memory.

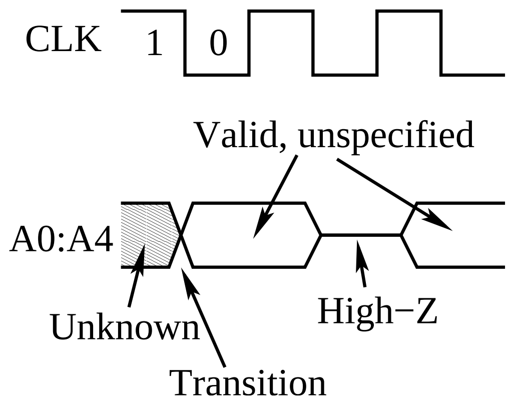

Figure 1: Example timing diagram

Input signals

1. A positive edge-triggering *clock* (C) signal.
2. An active-high *write* (W) signal.
3. A 16-bit *address* bus (A_{15}–A_0).
4. An 8-bit *data* bus (D_7–D_0).

You need to produce a *data hold* (H) signal.

The processor will assert W during a low-to-high transition of C. W will remain high until the last transition during which the data bus is driven. During the subsequent positive transition of C, the processor will drive the address bus, specifying the memory location to which it will write. This address bus will only be driven for one cycle. On the third positive transition of C, the data bus will be driven. The processor will stop driving the data bus **the cycle after** it senses a low value on H. In other words, it will have stopped driving the data bus before the rising clock edge following the first rising clock edge during which H was low. However, regardless of the value on H during the address cycle, the data bus will be driven for at least one cycle. The processor will stop asserting the W signal when it stops driving the data bus. When undriven, address and data signals float high, i.e., they're hooked to V_{DD} via pull-up resistors.

1. Draw and hand in a timing diagram describing the behavior of system during the memory writes. You need not (and can not) show the state of each line on the address and data busses. However, you can indicate whether the values are stable or in transition. Figure 1 provides an example timing diagram.
2. Given that the video controller's memory is accessed only for addresses ranging from 16,384 to 32,767, design a synchronous Mealy FSM with H as the output. Note that your machine will look a lot like a specialized counter. If possible, implement a simple combinational circuit with a one-bit output that will be used as the only input to the FSM. Specify your machine's inputs. Hand in a state diagram for your machine.
3. Manually do state minimization and state assignment. Use one of the methods presented in class, e.g., the implication chart method. Don't use row matching – it is not reliable. Depending on your initial design, it may or may not be possible to minimize the machine. Note that you should try to reduce the number of state variables. However, reducing the number of states isn't as important. In other

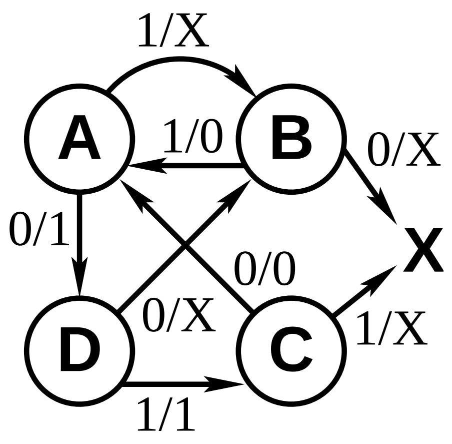

Figure 2: Example Mealy machine

words, reducing from nine to eight states is generally more valuable than reducing from eight to seven states. Note that you may define a one-bit function with multiple input bits. For example, you can define an address range and write recognition function and use its output in your state table instead of having a next-state column for each combination of address values. This will allow your machine to have only one input bit and one output bit. Follow the adjacency guidelines given in lecture when doing state minimization.

Using D flip-flops as your sequential elements, write your state variable and output functions. Simplify these manually. Hand in documentation showing this process.

4. Check your results by specifying your state machine in the BLIF format described on page seven of the "`/vol/cad/sis-1.2/sis/doc/blif.ps`" file. You can view this documentation on-line using the "`gv`" program or print it out with the "`lpr`" program.

   I think this documentation could be a bit clearer so I will include a commented example BLIF file. Please see the Mealy machine in Figure 2. We can also represent this FSM in Mealy form. This is the corresponding BLIF file.

```
.model
# Name the inputs
.inputs i
# Name the outputs
.outputs accept
.start_kiss
.i 1
.o 1
# Input, current state, next state, output
0 A D 1
# Don't-care output
1 A B -
# Go to dummy state X
0 B X -
1 B A 0
0 C A 0
1 C X -
0 D B -
1 D C 1
# Stay in dummy state w. DC output
- X X -
.end_kiss
.end
```

Please make sure the machine has a linefeed after the last line. SIS is picky about linefeeds. Start SIS and read in your BLIF file

```
read_blif /vol/ece303/lab3-example.blif
```

Tell SIS to do state minimization and print out the new state table

```
state_minimize
write_kiss
```

Note that the Stamina algorithm called by SIS reduced the number of states from four to two.

Now, tell SIS to use the Jedi algorithm to do state assignment and print out the results.

```
state_assign jedi -e c
write_blif
```

Note the assignment Jedi produced. Take a look at the unsimplified and simplified resulting networks.

```
print
decomp
full_simplify -m nocomp *
print
```

Compare it to the result we found in class.

The "`simple-dff.genlib`" is a gate library containing basic gates and D flip-flops. I produced this by editing one of the MSU libraries. You won't need to look up gates by their numbers because each gate has a meaningful name. Load in the library, do technology mapping, and print the results. Don't worry about the warning messages when running the "`map`" command.

```
read_library /vol/ece303/simple-dff.genlib
map
print
print_gate
```

Hand in the results of the "`write_blif`", "`print`", and "`print_gate`" command, after mapping. You don't need to draw the circuit. However, pay close attention to the state assignments given by the "`write_blif`" command. Also remember the association between latches and node numbers, e.g,

```
.latch [4] LatchOut_v2
```

5. Make a BLIF format file for your video controller repair circuit called "`fix-video.blif`". Use the same approach as you used in the previous section to come up with an optimized implementation. Hand in the results of the "`write_blif`", "`print`", and "`print_gate`" commands, after mapping. You don't need to draw the circuit diagrams.

6. Input your design into Mentor Graphics Design Architect.

   (a) In the file manager, copy the xor project from the first lab by pressing the RMB over it and going to "Edit→copy". Copy it to a file with the name "`fix-vid`".

   (b) Open the new file using design architect (refer to the first lab if you don't know how to do this).

   (c) Select the whole design and delete it using the delete button in the window to the right.

   (d) Use objects from "`gen_lib`" to enter and wire up your design. Note that a "dff" is a rising edge-triggered D flip-flop. Label each gate with its corresponding SIS node number, e.g., "[210]". You can add text by pressing the RMB and selecting "Draw→Text". Be very careful with this step or you'll end up wasting a lot of time.

   (e) You will need to connect unused active-low preset and clears to "`pullups`". Add an extra reset port to initialize the flip-flops. You will need to preset or clear a flip-flop depending on the state value for the starting state. You can find this out by looking at your state assignments. Put an exclamation mark in front of the port name if it is active-low.

   (f) Don't use the $\overline{Q}$ outputs from the D flip-flops in Mentor Graphics. Keep things consistent between your SIS and Mentor Graphics designs.

   (g) Check the sheet. You can ignore warnings about not hooking up the D flip-flop $\overline{Q}$ outputs to anything. However, fix any other problems that are identified.

   (h) Create a new symbol for the design, and overwrite the old symbol using the procedure described in lab assignment one.

   (i) Check the schematic. You shouldn't have any errors. You can ignore warnings about not hooking up the D flip-flop $\overline{Q}$ outputs to anything.

   (j) Hand in a printout of the circuit diagram.

7. Use QuickSimII to simulate the response of your circuit to the following input sequences.

   (a) Open the viewpoint in QuickSimII using the same procedures described in lab assignment one. Make sure you have a digital viewpoint for the design.

   (b) Set up the timing analysis using the same procedure as in lab assignment one. However, use typical delays instead of minimal delays.

   (c) Trace all the inputs, outputs, and state variables (D flip-flop outputs).

   (d) Click the "`Stimulus`" button in the right window. It will bring up a useful selection of buttons.

   (e) Add a force to the "`!clr`" signal that is 0 at 0 ns and 1 at 1 ns.

(f) Force a clock signal with a period of 1 ns and a stop time of 20 ns on the *clk* port.

(g) In my design, the FSM has only one bit of input that is generated by a trivial circuit that depends on the address and W signals. I forced this signal with a 0 at 0 ns, 1 at 2 ns, and 0 at 3 ns. If your design differs, force signals as required to start holding at 2.5 ns.

(h) You will find the "`Run`" and "`Reset`" icons in the window to the right useful.

(i) Run a 15 ns timing analysis.

(j) Hand in a printout of this simulation trace. You may use additional simulation to confirm that your design is correct but you only need to hand in the timing simulation diagrams for the sequences given above. If the result isn't correct, use the timing analysis to determine the source of the error and fix it using Design Architect.

8. Make a copy of your BLIF input file

```
cp fix-video.blif fix-video2.blif
```

Change the copy to fix the video controller even if it may require seven cycles of data assertion to correctly write its memory. Use SIS to come up with an implementation for the new version. Hand in the results of the "`write_blif`", "`print`", and "`print_gate`" commands, after mapping. You don't need to draw the circuit diagrams or do timing simulation on this more complicated circuit. Doing schematic capture on this larger circuit might teach you a little but would require a huge amount of time – I think you already get the idea by now.

A Source of design problem

A somewhat more complicated version of the design problem in Section 2 occurred on a project of mine. We implemented the repair circuit using a GAL (a 22V10 if I remember correctly). However, there was no clock input available on the video card so we needed to slice a metal tab off another card, superglue it to the video card, and solder an extremely thin wire from it to our GAL. There was also no place for the GAL so we caulked it, upside-down, to the back of the video control application-specific integrated circuit (ASIC). The prototype board looked like a joke but the output of the embedded system was perfect.

B Fully understanding what the software is doing

At this point, you should have a good understanding of the problems Espresso, SIS, and Mentor Graphics are solving for you. However, there are probably individual commands for which you do not understand the internal algorithms. I want you to learn as much as possible in this course, without it turning into a nightmare. Knowing exactly what is going on inside Mentor Graphics and SIS is beyond the scope of this course. If you want to know more about the individual algorithms, please see or email me and I'll give you supplementary reading material. However, although you should learn as much as possible about the tools you use, get used to the idea of using black boxes. Engineers frequently tackle problems beyond the scope of a single human's understanding. Only hierarchy, delegation of responsibility, and black boxes make this possible.