  Applied Homework #2

Combinational Circuits

# Description and Homework Objectives

In this Applied Homework, you will create components that you will later use to implement a version of the game Master Mind. In this applied homework, you will create a seven-segment decoder that will display a hexadecimal digit. You will also create an 4-bit 2-to-1 multiplexer that you will need later in the semester.

After this homework, you should be able to:

1. Design combinational circuits for implementation in an FPGA
2. Design enabling logic
3. Use hierarchy to simplify design entry
4. Use simulation to verify circuit functionality

# Applied Homework Tasks

You are expected to work with your partner together on the complete homework rather than sub-divide the tasks. Both partners are expected to thoroughly understand the work that is submitted. During the demo, each person should be able to demonstrate the circuits, explain how they work, and answer questions about them.

Before starting the work, be sure to download and unzip the AHW2 files from the class website—you will need these. Remember to save all your files to your **CAE network storage** so that you can access them later (including at the demo!). You will also need the files you create in later applied homework assignments.

**Be sure to read the Schematic Best Practices section at the end of the document before starting your design!**

You will be asked to submit schematics and waveforms for most, if not all, circuits you create. The full list is given in the report section. Always save the waveforms you are asked to create so that you can use them again later (including during your demo).

## Hex to Seven-Segment Decoder

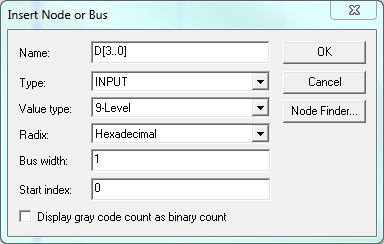
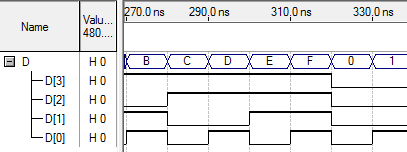
This circuit translates a 4-bit value into seven separate signals that each control one segment of a seven-segment LED display. Inside the decoder, we will have seven separate circuits—one circuit for each of the decoder’s outputs. The circuits will be named based on which segment they produce. Figure 1 shows the names of each segment, and what the display should look like to represent each of the 10 possible values of input **D[3..0]**.

Figure 1: Names of segments (left) and desired character to display for each of the sixteen possible D[3..0] values (right). For each character, segments that should be ON are shown in red; segments that should be OFF are shown in gray.

### Implementing Missing Segment Logic:

1. Open the project in **AHW2\ahw2\_demo**, and the **ahw2\_top\_level.bdf** schematic. This is the full top-level design for Applied Homework 2. Although it appears to be complete, a few modules are empty. For the first part of this work, you will look at the bcd-to-seven-segment decoder. Double-click one of the **hex7seg** symbols. Inside, you should see seven sub-circuits, each connected to the 4-bit input bus **D[3..0]** and one of the output pins. There is another input pin in **hex7seg** that is not yet connected—you will add logic that uses this pin later in this assignment.
2. Each sub-circuit contains the logic for one of the seven segments of the seven-segment LED display. Most are described using textual circuit descriptions written in the Verilog hardware description language. This is a more advanced method of circuit design that you will learn if you take ECE551. For two of the segments, **SegF** and **SegG**, only the input and output pins are present—the logic itself is missing. You will design, implement, and test these circuits for this applied homework.
3. Fill out the provided truth table and K-maps at the end of this document, and solve each K-map for a minimized **PRODUCT OF SUMS (PoS)** equation. Any non-valid BCD inputs should be treated as don’t cares in your t K-maps. You will need to upload a scan of this sheet with your submission. **NOTE: the seven-segment displays are *active-low*.**
4. Using your solution, fill in the missing logic inside of the **SegF.bdf** and **SegG.bdf** schematics. Using functional simulation, verify each module separately to ensure that, for all combinations of **D[3..0]** values, the output matches the corresponding value in the truth table. To make this easier, When adding the **D[3..0]** input to your **.vwf** file, you should enter it as a group, as shown below at right. At this time, you can also set the radix of how the group will be displayed – set it to hexadecimal so we can see the **D** input as a hex number. Now, you can expand the D input to view or modify individual bits, or collapse it to only see the 4-bit hex value. Have the **D[3]**, **D[2]**, **D[1]**, and **D[0]** inputs all start equal to 0, and then change values every 80, 40, 20, and 10ns, respectively (i.e., make the 4-bit **D** value increase by 1 every 10ns). Make sure that on the waveform, the input appears above the output.

You can then compare the waveform results to the truth table by reading from left-to-right along the waveform and comparing from top-to-bottom on your truth table. If one or both of your circuits are not operating correctly, fix them and test them again until you have verified that they work. Don’t forget to set each segment as the top-level entity when you simulate it!

In all applied homeworks, in any schematic, you should put **input pins on the left** of the circuit and **output pins on the right** (unless you are told otherwise). If a pin ordering (top to bottom) is stated, they must be in this order, and if pins are given in a partial schematic, do not rearrange them. Changing the pin order in the schematic will mean that your design **will not correctly connect** to the other circuits we will create later.

In all applied homework waveforms (unless you are told otherwise), the input signals should appear in the **same order** as the input pins (top to bottom), and then below that, the output signals should appear in the **same order** as the output pins (top to bottom).

1. Print out a functional waveform for each segment, starting at time 0 and showing exactly (no more than, no less than) one full set of the sixteen test vectors (i.e., each possible input value 0000 through 1111). On each segment’s waveform, for each of the sixteen test vectors, indicate whether or not the waveform shows the segment is on, and what the test case is (in hex, not in binary). Specifically, on the **SegG** waveform, you would write “ON for 5” beneath test vector 0101, and “OFF for 1” (without quotes) beneath test vector 0001 – provided that is in fact what your waveform indicates. Do this for all test vectors for each of the segments.

### Adding an Enable:

There will be times where we want to turn off all segments of the seven-segment display. The fifth input (**EN\_N**) to the **hex7seg.bdf** schematic serves this purpose. It is an ***active-low*** enable (the decoder should behave normally when **EN\_N** is 0, and be blank when **EN\_N** is 1).

Inside **hex7seg.bdf**, you will insert logic between the outputs of the seven **segA**, **segB**, … , **segG** symbols (we will refer to these collectively as the “**segX**” symbols to avoid listing all of them) and the output pins of the **hex7seg** schematic to implement this behavior. The logic will be identical for each segment; you only need to figure this out once, but insert it seven times. Do not put the enable logic inside the schematics for the individual segments—put it in **hex7seg.bdf**. The table below states which bit of the **hex7seg** output bus represents which segment of the 7-segment display.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Segment** | A | B | C | D | E | F | G |
| **SEG Bus Index** | SEG[0] | SEG[1] | SEG[2] | SEG[3] | SEG[4] | SEG[5] | SEG[6] |

1. Determine the logic function for the **hex7seg** output **SEG[2]** in terms of the output of **SegF.bdf** and signal **EN\_N**. Use the provided K‑map at the end of this document (which you will turn in) to find the equation. Remember, the segment displays are **active-low**. This means an **EN\_N** value of 1 should disable the segment (force it to be off) by forcing the **SEG[2]** output of **hex7seg** to be 1, regardless of the value of **SegF**. On the other hand, if **EN\_N** is 0, then **SEG[2]** should be equal to **SegF**.
2. Disconnect all seven **segX** modules from the output bus inside **hex7seg.bdf**. Insert the necessary logic between the **segX** module outputs and the output bus to implement the enable functionality. The same approach used to add enable functionality to **SegF** should be used to add enable functionality to the other segments.
3. If the segment logic was correct before adding the enable, it is not necessary to re-test all possible cases—we only need to test that the enable signal works properly (passing through **segX** outputs if **EN\_N**=0, and forcing all outputs to 1 if **EN\_N**=1). Functionally simulate **hex7seg.bdf** using a waveform as described below. Be sure in the waveform that **D[3..0]** appears above **EN\_N** to match the port order on the symbol, and that **SEG[6..0]** appears below the input waveforms. Verify that the logic works correctly. If it does not, correct it and re-test it until it does. Remember – you will need to set **hex7seg** as top-level.
   1. **D[3..0]**= 516 for 0ns−40ns; **D[3..0]**= 116 for 40ns−80ns.
   2. **EN\_N** = 1 for 0ns−20ns; **EN\_N** = 0 for 20ns−40ns; etc., to test both of the above cases for both **EN\_N** values.

Remember, you need to make sure that the **SEG[2]** output is identical to the **SegF** output on your previous **SegF** waveform when **EN\_N**=0, and is equal to 1 when **EN\_N**=1. You also need to ensure that **SEG[3]** is equal to your **SegG** output on your previous **SegG** waveform when **EN\_N**=0, and is equal to 1 when **EN\_N**=1. Also check that the other bits of the **SEG** output equal 1 when **EN\_N**=1.

1. Print out a functional waveform for **hex7seg**, showing the time range 0–80ns, which should show all four test cases (two values of **EN\_N** for each of the two **D** vectors). For the case when **EN\_N**=0 and the **D** vector equals 516, write “Enabled: 5” if the waveform output is correct. Label the remaining enabled cases similarly. Label the case where **EN\_N**=1 and the **D** vector equals 516 “Disabled: 5”, if the waveform output is correct. Label the remaining disabled cases similarly.

## 2-to-1 Multiplexer

We will need a 4-bit 2-to-1 multiplexer for the final applied homework. Note: this is a “bitwise” multiplexer. The 4-bit output is chosen by the select signal, which means that there is a separate 2-to-1 multiplexer for each bit position. For example, a single 2-to-1 mux chooses between **A[0]** and **B[0]** to produce **OUT[0]**. Another 2-to-1 multiplexer chooses between **A[2]** and **B[2]** to produce **OUT[2]**, etc.

### 1-bit 2-to-1 Multiplexer

1. Create a new schematic called **mux2\_1b** in the **ahw2\_demo** project. Place three input pins on the left, naming the top one **A**, the middle one **B**, and the bottom one **S.** Place one output pin on the right, and name it **OUT**.
2. Fill in the K-map given at the end of this document for the 2-to-1 multiplexer. Find the minimized sum-of-products (SoP) equation for this K-map, and write it in the provided space.
3. Implement the logic for the 1-bit 2-to-1 multiplexer in the **mux2\_1b** schematic according to your equation. Simulate it for all combinations of **A**, **B**, and **S**. Start with all values at 0, then change **A** every 80ns, **D1** every 40ns, and **S** every 20ns. Verify that the circuit works correctly. If not, correct it and re-test it until it does. Print the functional waveform for **mux2\_1b** so that you can scan it and submit it.
4. Once you are sure your **mux2\_1b** circuit is working, create a symbol for it. Refer back to the tutorial if needed to see how to do this.

### 4-bit 2-to-1 Multiplexer

1. Open the **ahw2\_top\_level.bdf** schematic, and double-click the **mux2\_4b** module. Currently it contains three input pins and one output pin, all of them already named. Do not move or otherwise modify the pins.
2. Insert eight copies of **mux2\_1b** into the module. Try to arrange these in a logical way to make the schematic easy to read. Implement the required connections to complete the 4-bit 2-to-1 multiplexer.
3. Functionally simulate **mux2\_4b** to verify that it works. Have the **A[3..0]** input change every 40ns, checking the values 0x00, 0xFF, 0x5A, 0xA5 (in that order). Have the **B[3..0]** input change every 20ns, checking the values 0x3C and 0xC3 (in that order). Make the **S** input start at 0, and change every 10ns. To make it easier to see the results, “collapse” the 4-bit **A** and **B** inputs and the 4-bit **OUT** output to only show their hexadecimal values, not the values of their individual bits.
4. Print your functional waveform (again, only showing the collapsed waveforms for **A**, **B**, and **OUT**), and after examining the result, write “functions correctly” or “does not function correctly” (whichever is true). The statement should be based on what your waveform demonstrates, not what you think should happen. (Note: if there is a difference between what you think should happen and what the waveform shows, you should go back and fix your circuit and then reprint your waveform!).

# Demo Tasks

DO NOT wait until your demo to read this!

During your Applied Homework Demo, you will test your **hex7seg** (including **SegF** and **SegG**) and **mux2\_4b** (including **mux2\_1b**) modules on the DE2 board by compiling the entire project (**ahw2\_top\_level** set as top level entity) and using it to configure the FPGA. Before your demo, make sure that you can compile the entire project, and take a few minutes to look at how the top level circuit is connected. You will use 16 of the slide switches to enter two 4-bit values, a pushbutton switch to control the mux select, and another slide switch to control the display enable. The BCD values of each of the entered 4-bit numbers will be shown on a pair of 7-segment displays, and the output of the multiplexer on another pair. If you have tested your modules and verified that they function correctly, the top level circuit will work correctly and you’ll see the fruits of your labors on the DE2 board.

As part of demonstrating your understanding of this applied homework, your lab instructor will ask you questions about this work. The below questions are examples of what might be asked, so we recommend thinking about these prior to your demo.

**Applied Homework 2 Questions:**

1. What would your SegF and SegG implementations produce for an invalid BCD number?
2. Values 10-15 are not valid BCD digits. Describe a way that you could indicate to the user if an invalid BCD input was applied to your circuit. If you changed the BCD to segment decoders (e.g., SegF and SegG) to indicate invalid inputs, what would be the impact to the decoders.
3. The FPGA we use during the demos has 4-input LUTs. How many LUTs will be used to implement your **SegF** logic?
4. List the minimal combinations of input values that test every segment being both on and off.  There may be more than one answer. *Note: there is more than one right answer...*

# Applied Homework Submission

Only one member of the group should submit the files. The other partner should only answer question #1 in the assignment. Both group members will receive the same credit for this part of the applied homework (provided both group members attend the demo).

The grade for the demo component may differ based on each person’s demonstrated knowledge and understanding of the submitted work.

You will upload a number of files to the AHW2 activity in Moodle. Follow the directions in the activity carefully, and be sure that your files are named correctly.

First, you will need to upload scans of the completed worksheets at the end of this document, which show your work in the design of several of this assignment’s components. Scans of these pages (as well as scans of the requested waveforms) can be submitted as either PDF or JPG files. Do not use any other file format. Be sure that the scans are **legible**, and limit JPGs to no more than 1920x1080 resolution.

When you upload the requested BDF files (which should each include a title block), you will also upload a JPG image of those BDFs. Each JPG should have the same base file name as the associated BDF (e.g., the JPG image of a schematic named **schematic.bdf** should be named **schematic.jpg**). To create a JPG for a BDF, in Quartus go to File 🡪 Export.

You will also upload **scanned** copies of the waveforms that you have printed and annotated according to the instructions in this tutorial document. These can be submitted in either PDF or JPG format (same as the worksheet scans).

The required files are listed on the following page. You may wish to use this list as a checklist to ensure that you do not forget to upload any of the required files. Be sure that the files that you upload are the ones with the correct extensions (e.g., make sure that you upload **mux2\_1b.b**d**f**, not **mux2\_1b.b**s**f**)

**Submission Notes:**

* **For any** **schematics you turn in this semester for any written or applied homework**, **you must include a title block** (Symbols, “Other”). The title block **must** include the name of the circuit and the names of both partners.
* **For any JPGs of schematics you turn in this semester,** they must match the submitted BDF file. You can create the JPGs by having the BDF open in Quartus, and clicking File 🡪 Export.
* **For any waveforms you turn in this semester**, ensure that your waveforms are zoomed in enough that they can be read, and should include the requested span of time and any requested annotations. If no specific timespan is given for a combinational circuit’s waveform, show each tested input combination **once**. Scan the waveform printout and save it as a JPG or PDF. The resolution of submitted JPGs should be no more than 1920x1080. Be sure to open each file and check it to be sure it is legible!

### Files Required for Submission

**Worksheets (from the end of this document)**

* **worksheet1.jpg/pdf** – the scan of the page with the segment truth tables, K-maps, and equations (showing your work)
* **worksheet2.jpg/pdf** – the scan of the page with the segment enable and 2-to-1 mux K-maps and equations (showing your work)

**SegF**

* **SegF.bdf** – the schematic for the segment C logic
* **SegF.jpg** – the JPG of the **SegF** schematic
* **SegF\_func\_sim.jpg/pdf** – the JPG or PDF of the annotated **SegF** functional waveform.

**SegG**

* **SegG.bdf** – the schematic for the segment D logic
* **SegG.jpg** – the JPG of the **SegG** schematic
* **SegG\_func\_sim.jpg/pdf** – the JPG or PDF of the annotated **SegG** functional waveform.

**hex7seg**

* **hex7seg.bdf** (AFTER implementing the enable feature!) – the schematic for the complete 7-segment decoder with the enable
* **hex7seg.jpg** (AFTER implementing the enable feature!) – the JPG of the **hex7seg** schematic
* **hex7seg\_func\_sim.jpg/pdf** – the JPG or PDF of the annotated **hex7seg** functional waveform.

**mux2\_1b**

* **mux2\_1b.bdf** – the schematic for the 2-to-1 multiplexer
* **mux2\_1b.jpg** – the JPG of the **mux2\_1b** schematic.
* **mux2\_1b\_func\_sim.jpg/pdf** – the JPG or PDF of the **mux2\_1b** functional waveform.

**mux2\_4b**

* **mux2\_4b.bdf** – the schematic for the 4-bit 2-to-1 multiplexer and its JPG
* **mux2\_4b.jpg** – the JPG of the **mux2\_4b** schematic.
* **mux2\_4b\_func\_sim.jpg/pdf** – the JPG or PDF of the annotated **mux2\_4b** functional waveform.

Names:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### Hex to 7 Segment Decoder Truth Tables and K-Maps

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **D3** | **D2** | **D1** | **D0** | **C** | **D** |
| 0 | 0 | 0 | 0 |  |  |
| 0 | 0 | 0 | 1 |  |  |
| 0 | 0 | 1 | 0 |  |  |
| 0 | 0 | 1 | 1 |  |  |
| 0 | 1 | 0 | 0 |  |  |
| 0 | 1 | 0 | 1 |  |  |
| 0 | 1 | 1 | 0 |  |  |
| 0 | 1 | 1 | 1 |  |  |
| 1 | 0 | 0 | 0 |  |  |
| 1 | 0 | 0 | 1 |  |  |
| 1 | 0 | 1 | 0 |  |  |
| 1 | 0 | 1 | 1 |  |  |
| 1 | 1 | 0 | 0 |  |  |
| 1 | 1 | 0 | 1 |  |  |
| 1 | 1 | 1 | 0 |  |  |
| 1 | 1 | 1 | 1 |  |  |

**K-Map for segment F**



**[REMEMBER – PRODUCT OF SUMS!]**

**SegF =**

**K-Map for segment G**



**[REMEMBER – PRODUCT OF SUMS!]**

**SegG =**

Names:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### Segment Enable K-Map (SEG[2] as a function of SegF and EN\_N)



**SEG[2] =**

### 1-bit 2-to-1 Multiplexer



**[REMEMBER – SUM OF PRODUCTS!]**

**Y =**

  Applied Homework

Best Practices

# Schematic Design Best Practices

When creating schematics in this class (and in “real life”), you should follow good design practices. It is important to create schematics that are well-organized, easy to read and understand, and free from clutter. This appendix illustrates some of these best practices for schematic design.

Why do we follow these best practices?

* Your schematic is easier for you to read and understand.
* Your schematic is easier for others to read and understand. This is especially important in a job setting, where you work as part of a team to produce large designs.
* Your schematic is easier to maintain and modify (not to mention, debug!).
* Good, clean design is an excellent habit to develop, whether for schematic diagrams, software code, written articles, or anything else you create for others to appreciate.

## Connection by Name

One good design practice is “connection by name” instead of “connection by wire.” If you draw two nets in your schematic and give them the same exact name, Quartus will automatically connect them invisibly. In this way you can avoid many wires cluttering up your schematic. For example, in the following excerpt from a schematic, two nets are named X1 and are automatically connected:

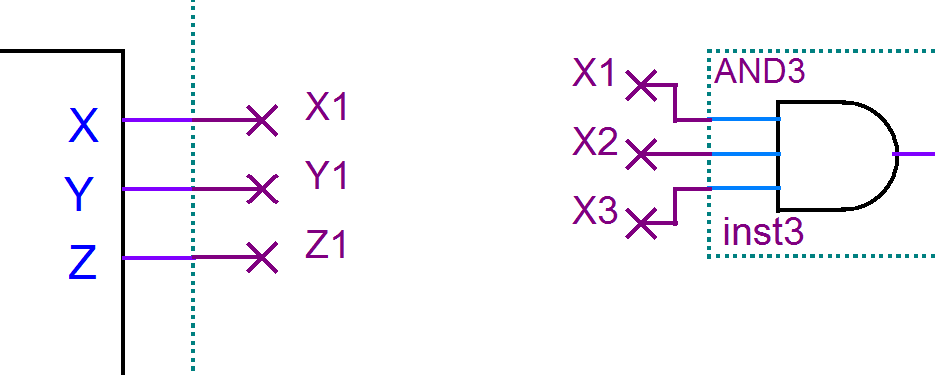


Figure : Connection by name

Connection by name lets you turn a schematic that looks like:

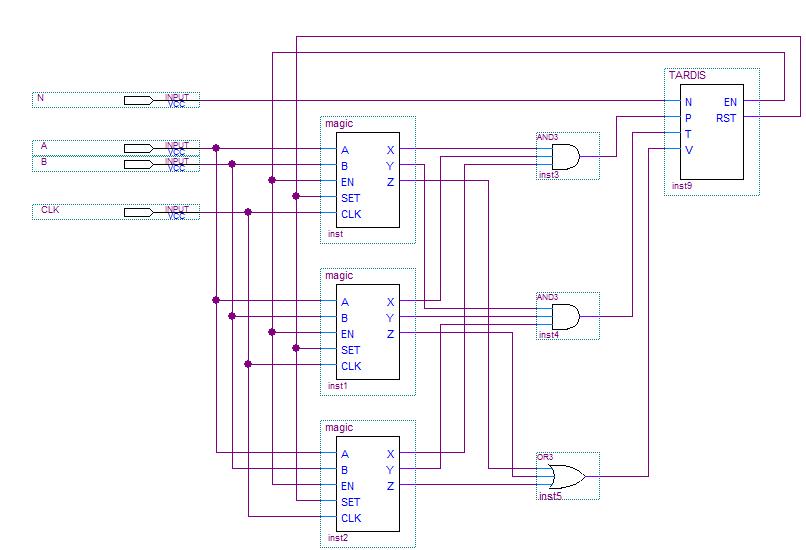


Figure : BAD design: connection by wire

into a much cleaner schematic that looks like:

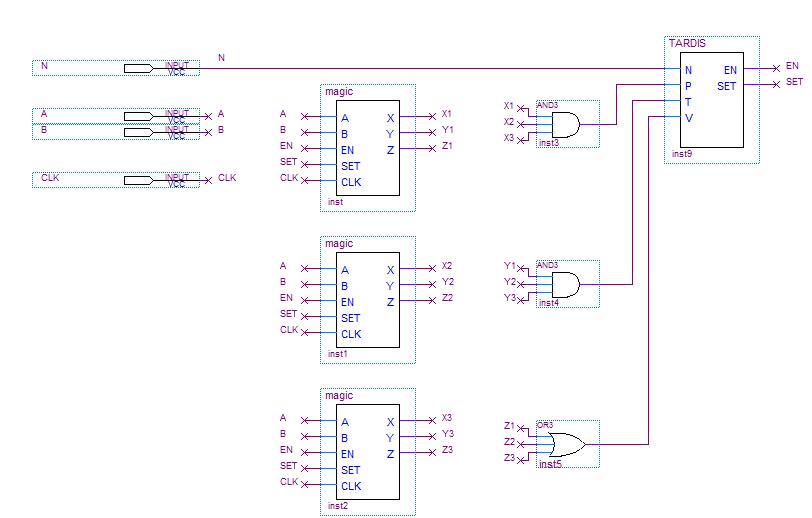


Figure : Good design: connection by name

Note that a few connections in Figure 4 are made using wires; you should connect by name **when it makes your schematic cleaner and easier to read**. The AND and OR gate outputs do not need to be connected by name, since their connections are clear and easy to understand as wires.

## Connecting to Busses

Connection by name extends to busses; you do not need to make graphical “bus taps. For example, in the following figure, the output of the AND gate is already connected to bit #3 of the D[3..0] bus.

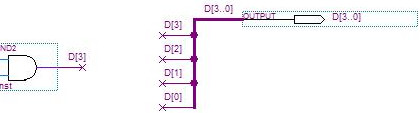


Figure : No, you don't need graphical bus taps

## Avoid Overlap

For clarity and readability, avoid any overlap of modules, connections, and text. All design details should be clearly visible, and connections should route around modules. Instead of:

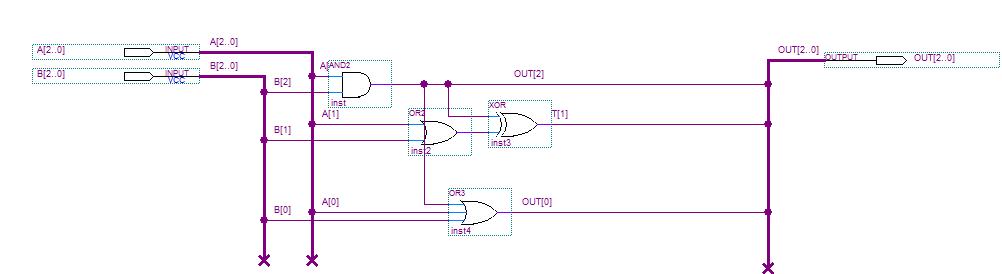


Figure : BAD design: text is obscured, and nets overlap modules

your schematic should look like:

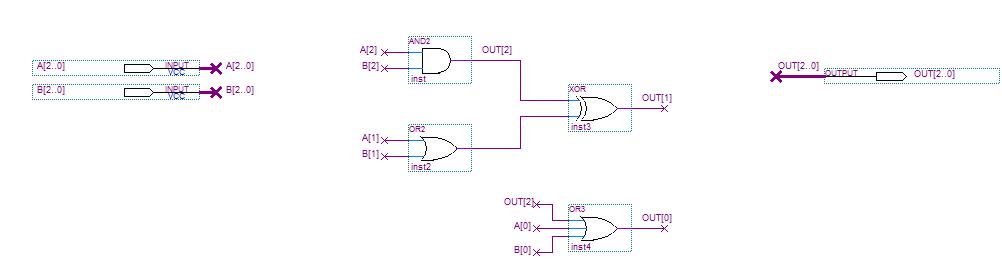
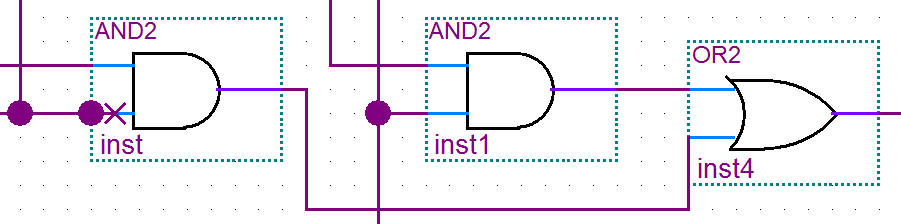


Figure : Good design: all text is visible, and net connections are clear

## Net Stubs are Bad

You may have seen little net “stubs” in schematics. These are short wires that end in a large “X,” as shown here:



These net stubs make your schematic cluttered, and they can make debugging very difficult. Delete them to make cleaner designs that are easier to modify and maintain.

## Final Comments

We have prepared these design best practices to help you succeed, both in class and in your professional development. Following these will make your life easier and likely will result in less headaches during debugging.