PHYS 432 – Lab 5: Programmable Logic

5.1 Introduction

Note: this lab is probably too long. For full credit, you only need to do 1 of the first 2 major sections (Sec. 5.2 or Sec. 5.3) along with Section 5.4 on FPGAs. The EEPROM section is probably shorter, although the PAL section might be more useful if you want a simple state machine in your project. These two sections share some circuitry, however, so if you have time left, you might want to come back and finish the third section at the end.

In modern electronics, it is very uncommon to produce circuits entirely from discrete component ICs. Modern devices make extensive use of some form of programmable logic, where the specific functionality of a device is "programmed" in advance, and in many cases can be changed. These programs are often called "firmware" to denote something slightly more permanent than software, but not completely fixed either. In this lab, we will explore several examples of programmable logic: an Electrically Erasable Programmable Read-Only Memory (EEPROM), a multipurpose Programmable Array Logic device (PAL), and a Field Programmable Gate Array (FPGA). Of these there, only the FPGA is in common use today, but all three are good examples of programmable logic and can be useful for your projects.

We will be using the PCs in the electronics lab to program the chips. There is only one station to program the PLDs, both chip programmers work for EEPROMS, while all 4 PCs can be used to program the FPGAs. You can do the three sections of this lab in any order, so try to plan with your other lab mates who is doing what, and please be respectful in sharing the equipment. It is probably worth doing this lab with a partner to minimize the number of different groups that need to use the PCs at once.

The intent of this lab is to give you a first look at these topics and demonstrate the techniques for implementing programmable logic. To use these techniques in your project will probably take some more work on your part to learn these topics better. More detailed instructions for how to use the programming software, necessary files, and additional outside documentation can be found on the course website under the Lab 5 "Programmable Logic" link.

Please be careful inserting/extracting these chips from your breadboards!. The chips are very large, and in some cases are rather antiquated and hard to replace.

5.2 EEPROM Look-up Table

In this section, we will use a Read-Only Memory (ROM) to create an arbitrary binary function to drive 8 LEDs. We will use the AT28C16 ROM (you can also use the AT28C64 if we run out) which contains 2048 bytes of storage (2k x 8 bits = 16 kbits of memory). A ROM is similar in some ways to a MUX. An address is used to select one of the 2048 bytes stored in the ROM, and the output data lines are set to the contents of the selected memory location. The memory is filled in advance with data programmed from a PC, with one 8-bit number stored in each of the 2048 locations.

5.2.1 4-bit counter

We will use a 4-bit counter (7493) to select the first 16 addresses of the EEPROM, and use the data in the EEPROM to light up patterns on 8 LEDs. Using the pin-out diagram in Figure 1, wire up the 7493 and make sure you can get a proper counting sequence with the 4 output bits. Note, this is in the same family as the 7490 decade counter you explored last week, so you should drive `CKA` with a debounced button, output `QA` should be connected to `CKB`, and the two `R0` reset lines should be wired to ground.

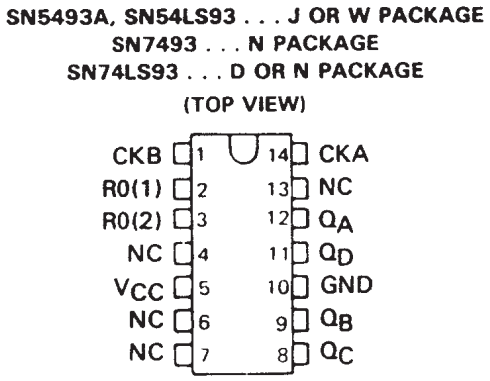

Figure 1: Pin Diagram for 7493 4-bit counter.

5.2.2 EEPROM programming

Once you are convinced your counter is giving a proper 4-bit binary counting sequence, find one of the AT28C16 ROM chips and go to one of the chip programmers. The GQ-4x4 is the best to use, as it can't be used to program PAL chips. The software to drive this is called USBPrg and can be found on the desktop with a big GQ logo. The smaller TL866II also works, and uses the Xgpro software. The use of these tools is basically the same. The programming software will present a spreadsheet-like interface where the contents of each address location can be specified. We will only use the first 16 locations, so it is easy enough to just enter these by hand.

Select the correct chip by part number in the programmer and check the figure to see how to install the chip into the programming socket. Close the latch to make electrical contact with the chip, and you should be ready to program. You can enter data values directly into the spreadsheet-like display. Since we only have a few entries, this direct entry is probably the easiest. A longer pattern can be saved in a file and opened by the software.

We will try the simple pattern shown in Table 1. The address and data values are shown that should give the following pattern on a set of 8 LEDs. Enter this pattern into the first 16 memory locations and write this data to the chip. The programmer should automatically verify that the data written matches the data read back from the chip, but you can also read manually and check the pattern yourself.

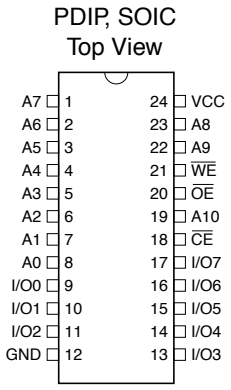

Figure 2: Pin Diagram for the AT28C16 EEPROM.

Address	Data	Pattern
0x00	0x00	00000000
0x01	0x01	00000001
0x02	0x03	00000011
0x03	0x07	00000111
0x04	0x0F	00001111
0x05	0x1F	00011111
0x06	0x3F	00111111
0x07	0x7F	01111111
0x08	0xFF	11111111
0x09	0x7F	01111111
0x0A	0x3F	00111111
0x0B	0x1F	00011111
0x0C	0x0F	00001111
0x0D	0x07	00000111
0x0E	0x03	00000011
0x0F	0x01	00000001

Table 1: Truth table to light up 4 LEDs in order. Note the address and data values are written in hex.

5.2.3 LED display

Figure 2 shows the pin assignments for the AT23C16 EEPROM. To address 2048 locations requires 11 address lines (indicated by A_i). Since we stored patterns in only the first 16 locations, all but the lower 4 bits must be wired to ground (remember, TTL inputs tend to float HI, so leaving them unwired will likely give you a different address than you mean) and the 4-bit counter output should be wired to $A_3A_2A_1A_0$.

The data stored in the EEPROM will be output on the I/O lines. Hook these 8 outputs to the LED indicators on the trainer board. You will also need to wire up the three control lines appropriately. They are $\overline{WE}$ (write enable), $\overline{OE}$ (output enable), and $\overline{CE}$ (chip enable). We want to read (not write) the chip, and both the chip and output enable functions must be asserted. Given that all three control lines are negative true, explain how to wire these three pins to make this happen. You can connect them to DIP switches if you are not sure, or read the data sheet.

Draw a high-level schematic of your final circuit (you do not need to include detailed pin numbers) and verify that the LED pattern comes out as you expect as you cycle through the counter states.

5.3 PAL Combinatoric Logic

Programming a complex combinatoric logic algorithm like a 7-segment decoder using discrete components is a real pain. The first simple programmable devices called programmable array logic (PALs) were PROM-like devices which allowed combinatoric logic to be specified via a parameter file which could then be programmed into the chip by burning a set of internal fuses. Modern devices can emulate simple PALs using flash memory, which allows them to be reprogrammed, although they can also do much more as we will see later. In this section, we will use a 16V8 PAL (the Lattice GAL 16V8 is the same) to implement an encoder algorithm to light up four LEDs in the pattern shown in Tab. 2.

Inputs			Outputs			
D_2	D_1	D_0	Q_3	Q_2	Q_1	Q_0
0	0	0	H	H	H	H
0	0	1	H	H	H	L
0	1	0	H	H	L	L
0	1	1	H	L	L	L
1	0	0	L	L	L	L
1	0	1	H	L	L	L
1	1	0	H	H	L	L
1	1	1	H	H	H	L

Table 2: Truth table to light up 4 LEDs in order.

5.3.1 CUPL file

There are a variety of languages which can be used to program PALs, but one of the most straightforward is a language called CUPL. Download the file `ledenc.pld` from the web site, which is a CUPL file, and write into your log book a brief description of what each line is doing. Use material from class or posted on the website if you are unsure of the syntax. Figure 3 shows the pin assignments for the 16V8 PLD. Note that some pins must be used as inputs, while others can be defined to be either inputs or outputs. Annotate this diagram in your log book to show which input and output signals go to which pins for our CUPL file.

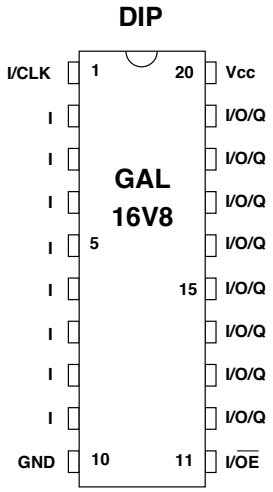

Figure 3: Pin Diagram for the GAL16v8 PLD. Note some of the pins can be configured as either inputs or outputs

The method used for specifying the algorithm in the provided file is common for encoders, where each input pattern specifies a specific and unique output pattern. CUPL can also specify any sort of combinatoric logic by simply defining the input and output pins, and setting the output to be equal to some function of the input values. Write down what you would need to add to this file to implement a three-input AND of pins 1–3 output on pin 15.

5.3.2 PLD programming

Go to the PC with the small TL866II programming box attached. The larger GQ-4x4 programmer will **not** program PAL or GAL devices correctly. This box is a general purpose programmer which can be used to program a wide range of flash-based programmable logic, including EEPROMs and even some microprocessors. Open the WinCUPL application and load he `ledenc.pld` file. Compile this file following the instructions on the PLD Info web page. If everything goes well, you should have a `ledenc.jed` file, which is the hardware-specific JEDEC file to be loaded into the chip by the programmer. For historic reasons, it is still common to call this a fuse map.

Place the PAL chip into the socket, with the notch facing up, towards the lever. Open the Xgpro application which controls the programmer. Click the 'Select IC' box and pick the

correct part (and vendor). The 'Device Info' tab at the bottom of the data entry area will show you the correct orientation of the chip in the programmer socket. Load the `ledenc.jed` file, make sure again that the device is set to the proper value for the chip you are using, and write the file to the chip using the big red P button at the top. Please do not write the LOCK bit. I don't think this actually prevents the programmer from re-writing the chip, but better to be sure. If all has gone well, the program should indicate success. If a chip fails the validation step, try again. If it repeatedly fails, try a different chip. If you can't get the PLD to program properly, and there is nobody around to help, do the EEPROM section instead.

5.3.3 PLD circuit

We will use a 4-bit counter (7493) and send the first 3 bits to the PLD. This is identical to the circuit used in the EEPROM section. Using the pin-out diagram in Figure 1, wire up the 7493 and make sure you can get a proper counting sequence with the 4 output bits. Note, this is in the same family as the 7490 decade counter you explored last week, so you should drive `CKA` with a button, output `QA` should be connected to `CKB`, and the two reset lines should be wired to ground.

We will drive the PLD with the first 3 bits of the 4-bit counter (least-significant bits) and view the results by connecting the PLD outputs to the LEDs on the trainer boards. Take the PLD chip out of the programmer, and install it on the breadboard near the counter. Be very careful when inserting these chips into the breadboard. Draw a high-level schematic of your circuit (with a box for each chip and the main inputs and outputs shown). You do not need to indicate the specific pin numbers unless you want to. When you wire up your circuit, make sure the PLD can be easily removed, as you may need to program this more than once.

Do you get the pattern you expect?

5.3.4 Registered PAL

Now, we will explore some of the more powerful features available on some more modern PLDs. The ability to make arbitrary combinatoric logic goes a long ways towards simplifying the implementation of state machines. The only remaining piece needed is a set of data registers. A "registered" PAL provides exactly this functionality, and a PLD used in "registered" mode emulates these devices.

Download the file `ledcnt.pld` and compare to `ledenc.pld`. This file will produce the same output pattern, but also defines an internal state machine to provide the 3-bit counter, such that the only needed input to the chip is a clock pulse. Describe in your log book what the additional code found in `ledcnt.pld` does.

Compile and download this program to the chip, and modify your circuit such that only the debounced button is being used as a clock input to pin 1 of the PLD. With the outputs attached to the LEDs on the trainer board, check that all of the defined PLD output pins are producing the expected output. In "registered" mode, pins 1 and 11 have specific functionality. Pin 1, for example, must be the clock input. What does pin 11 do? (Hint:

look at the pinout diagram in Figure 3.) You must attach the proper voltage to pin 11 or else your registered PAL outputs may not work properly!

Note that you can easily expand your own LED pattern state machine to more than 8 states, since it is easy to make a 4-bit (or more) counter internally in the PLD. Each output pin has 8 product terms associated with it, so fairly complicated patterns are possible.

5.4 FPGAs

Field Programmable Gate Arrays (FPGAs) are the modern evolution of programmable-logic devices. We will be programming a Lattice ICE40LP8K FPGA installed on a TinyFPGA BX board. This board provides some power management and a USB connection, but the pins on the BX board are directly connected to (some) of the pins on the FPGA, so we have rather direct access to these pins in dual-inline package.

Warning: the FPGA on the BX board uses 3.3V! For this lab, **do not hook the board up to any external power supplies!** The BX board will power the chip safely from the USB connection, so simply use this. You can connect the BX pins as outputs to circuits expecting 5V, but please don't try to put 5V logic signals into the BX board, as you could burn out an input pin.

Figure 4 shows the pin assignments of the BX board. Note there are two ground pins and a regulated 3.3V output that you can use to power your breadboard. For this lab, you should only need a ground connection.

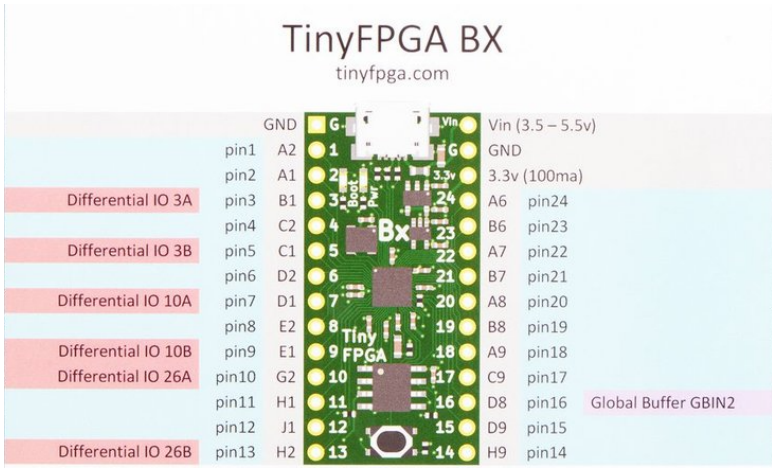

Figure 4: Pin Diagram for the BX TinyFPGA board.

5.4.1 Programming Test

Take a BX board, carefully plug it into your breadboard, but don't make any other connections for the moment. Hook it up via a USB cable to one of the PCs and find the Windows command line (likely in the Utilities directory). Download the files `seconds.v` along with `pins.pcf` and `apio.ini` from the website into a new directory. From the command line, navigate to this directory (cd to change directory, dir to list what is in a given directory)

and once you are there, type the following command to synthesize the bytestream file for the FPGA:

```
apio build
```

This may produce a large number of warnings about unused ports, but it should finish with a line that says `SUCCESS` and there should now be a file `hardware.asc` in the same directory. You can try the build command again, and if everything is fine, it should return quickly with the `SUCCESS` line (the up-arrow will retrieve the last command entered at the command line).

Now, we can try to download the firmware to the BX board. Push the black button at the bottom of the BX board to put the board into programming mode. One of the two red LEDs on the board should be constantly on (indicating power) and the other should be slowly fading in and out. This signals that the board is ready to be programmed. Test that the PC can find the board with the following command:

```
tinyprog -l
```

If the response says no bootloaders were found, try again. When this works, you should see information about the FPGA such as `FPGA: ice40lp8k-cm81`.

Type the following command to download the bytestream file to the FPGA:

```
tinyprog -p hardware.bin
```

Again, you may need to do this a few times, but when this succeeds you will see clear evidence of something being downloaded and the message `Success!` at the end. The FPGA should automatically switch to run mode, and the LED should now be crisply flashing on/off approximately once per second. Congratulations, you have programmed an FPGA!

5.4.2 First Verilog

Open the file `seconds.v` with a text editor and look at the contents. All Verilog files have a `top` module that represents the connections in/out of the FPGA. The 16 MHz system clock is one input `CLK` and the board LED is an output `LED`. The code here is very simple. The `CLK` is an input to a `secondcounter` block that decrements a 24-bit counter and toggles the output value (wired to `LED`) after 8M counts. The hex number `7A1200` is where the count starts, and it resets every time it gets to zero. Change this number to something slightly smaller (around a factor of 2 different), save the file, and reprogram the chip using:

```
apio build
tinyprog -p hardware.bin
```

Don't forget to activate the BX bootloader by pressing the black button before downloading the bytestream file. You should now see the LED flashing more quickly. If it appears constantly on, it is probably flashing faster than your eye can follow.

5.4.3 Counter and Decoder Logic

Create a second directory and download the file `buttontest.v` and also copy the `pins.pcf` and `apio.ini` files.

Open `buttontest.v` and look at the inputs and outputs defined in the `top` module. Note we now have to additional inputs on pins 2 and 13, and 8 additional outputs on pins 14-21. The `USBPU` is a special output pin that disables the USB data connection while the FPGA is running, and can be safely ignored.

Write in your log book all of the defined modules you find in this file. There should be 3 plus the `top` module. Each of these modules is instantiated in the `top` module and has inputs/outputs connected up. Draw a high-level schematic of this logic. Each module should be a block, and the names of the input/output connections should be labeled.

You don't need to understand the button debouncing module, but the `counter8` and `displaydecoder` blocks should be understandable. Briefly describe what each of these blocks is doing. Synthesize this file using `apio build` as before and make sure the command ends in success.

5.4.4 FPGA Circuit

You should wire the ground connections on both sides of the BX board to the rails on your breadboard, as we will use these. We will hook up a simple tactile switch between Pin 2 and ground to provide an input source. *Do not use the debounced switches from the trainer board!*. A second tactile switch from Pin 13 to ground will provide a reset signal. Normally, a switch to ground wouldn't provide a logic signal without a pull-up resistor. To avoid external pull-up resistors, each input pin on the FPGA has an optionally programmable internal pull-up resistor that is specified for these two pins in the `pins.pcf` file. So by pressing the switch, the input pin will be grounded (`LO`) while when the switch is released the pull-up resistor will set the input pin to `HI`.

For the output, connect one LED between each of pins 14-21. I would recommend using individual LEDs from the drawer, or the bargraph LED arrays. Usually, you should always use a current-limiting resistor when driving an LED from a digital logic pin, but the FPGA output pins can only supply 8 mA each, so the LEDs are safe. If you decide to use the LEDs on the trainer board, you will need to provide a ground connection between the BX board and the trainer. **Do not make any other power connections!**

Looking at the Verilog file, try to understand which end of the LED array is the most significant bit (MSB) and least significant bit (LSB) in the internal `LEDreg` bit array.

Write down what you expect to see on the LEDs as you press the main button. Will the LEDs count a binary sequence, or do something more interesting? What do you expect to see when you press the reset button? Be specific (i.e. what LED pattern should you see after reset?)

Now synthesize and program the FPGA with the following commands:

```
apio build
tinyprog -p hardware.bin
```

Once this succeeds, play around with the buttons and explain what you actually see compared to your expecations.

5.4.5 Button Debouncing

Debouncing switches is a necessary step in using buttons with digital logic. While the trainer board provides external circuitry to do this, it is more common to use 'bare' switches and debounce them internally with programmable logic. Find the place in the `top` module where the 8-bit counter is instantiated. Comment out this line and uncomment the line below where the counter is driven directly from `PIN_2`. Synthesize and download this new logic and describe how things are now different. How many steps does the LED pattern shift for each button press?

Put the code back the way it was originally when you are done with this section.

5.4.6 LED Decoding

Find the place in the `top` module where the `displaydecoder` block is created. Comment out this line and uncomment the line below that just assigns the counter value to the LED register. What pattern do you now expect to see on the output LEDs? Try it and check. Which output pin is the LSB?

Put the code back the way you found it and look at the logic in the `displaydecoder` block. The `lut` here is a look-up table very similar in functionality to the EEPROM-based version in Section 5.2. If you have time, sketch out your own 16-step pattern in your log book, determine the values needed to implement that pattern, and try it out by entering those values in the `initial` block (that initializes the `lut` register array). Note that the final line in this block

```
assign display = ~lut[state[3:0]];
```

uses a bit-wise invert operation between `lut` and the `display` output returned by the decoder.

If you don't have time to create your own pattern, remove this invert and change a couple of the numbers in the `lut` assignment and check that when you reload the bytestream file you see the changes that you expect.

Show the final result of this section to your TA and get this section signed off in you logbook.

5.5 Conclusion

In this lab, we have seen a few examples of useful programmable logic which you can use in your projects. FPGAs and PLDs can often both be used for designing all-in-one state machines, and if the logic is simple enough, a PLD may be the easiest solution. This simplicity of programming a PLD makes this a good choice, although extremely complicated algorithms or state machines might not fit into the limited resources available on a given PLD chip, and an FPGA then becomes a more viable option. If you simply need a complicated function of many inputs, an EEPROM is a good choice.