EE241 Digital Design
Lab #7 Programmable Logic used to Implement 7 Segment Decoder
Demonstrate by March 23, Report due March 31, 2018

Objective:

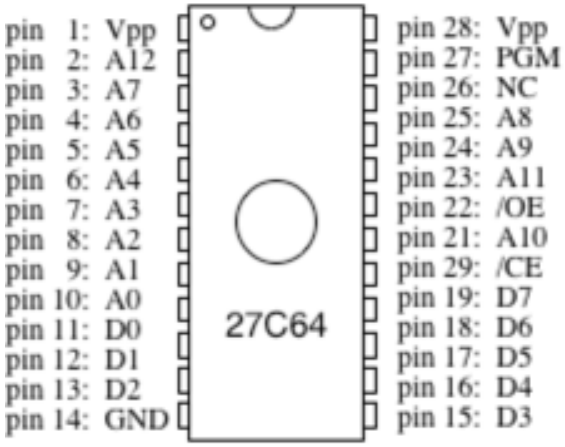
Become familiar with the use of simple programmable devices, specifically EPROMs and PALs/GALs. We will replace the discrete gate logic of the multiplexed display with programmable logic. One device (GAL 16V8 or 27C256 EPROM) will replace all that stuff.

Instructions:

You should already have a set of Boolean equations for the logic of your binary decoder, but you may wish to change your code to something more useful than the assignment made for Lab #5. We will have available 4 bit words when we start using the memory devices later, giving us up to 16 characters, and you could try to extend this further by choosing between two character sets (a fifth input bit is a character set select). However, you will find that the PAL is not large enough for 32 different characters; there are not enough terms provided. You may want to come up with your own expanded set of letters that will be useful for the scrolling display to come. The two types of devices require different approaches:

1. EPROM:

The 27C256 Erasable Programmable Read Only Memory is a 28 pin device capable of storing 256K bits. That's a lot more than we need, but this was the least expensive device of this kind available. The "C" stands for "CMOS", the "27" specifies that this is one of a family of similar EPROMs, and "256" is the size in K bits. The 27256 is similar, but NMOS instead of CMOS. NMOS parts are sometimes less expensive, but more power hungry. The data is stored in "words" (or, "bytes") of 8 bits. This is very convenient for driving 7 segments plus a decimal. There are 32 K bytes in the 256K device. Note that this is an MOS device, and is vulnerable to static damage! When doing your wiring, borrow a defective device or mark off the pins where you will put the device, and only put in your good one once it's programmed to avoid unnecessary risk. The 27C256's we have are 250ns parts; faster ones are available, but cost more. A 27C64 EPROM has the pinouts shown below. The 27256 and 27C256 are also the same except that they have two additional address inputs at pins 26 and 27 (the latter replacing PGM). The pinouts are the same for our purposes. All of the upper address pins need to be grounded, so ground pins 26 and 27 are too.



Power: Connect Vcc and Vpp to 5 Volts. (Pin 27 is grounded for the 27256.) (The programmer will put a higher voltage on Vpp during programming, but you don't have to worry about that.) Connect GND to your ground. Also, connect "Chip Enable" (active low) and "Output Enable" (also active low) to Ground; we will leave the device on all of the time. In computers, /CE and /OE are used to control when the device is being read and when it is not.

Address inputs: Signals A0 on up are the "Address bus" inputs which specify which of the 32K words we want to see. Since the 74189 RAM (which we may want to use to store the character code of our message) has only 4 bit words, we will at least for now, only use the lowest 4 address lines, A0 to A3. (You could use A4 to a DIP switch to select between two different sets of characters.) The unused address lines need to be tied to Ground (0 inputs). Unlike TTL, MOS device inputs should always be tied to a logic 0 (Ground) or 1 (through a resistor to Vcc), never left floating. Leaving them floating can even destroy the device. I've seen it happen.

Data Outputs: Signals D0 to D7 are 8 bits of output, one for each segment plus the decimal. One should go (via resistors) to each segment of your display. Leave unused outputs open.

You should figure out, for each address (corresponding to a 4 bit input value, padded out with leading zeros) what logical values you want on the outputs. Remember that a "0" turns a segment on, a "1" turns it off. This data is in your truth table. Convert the 8 bits for the outputs to the 7 segment display for each input combination into a hexadecimal number. (It is up to you to decide which segment is driven by which output, d0 to d7, of the memory. I prefer segment "a" to map to "d0" because it is consistent with "little endian" bit numbering, but most students prefer otherwise.) With your table of 16 x 8 bit (2 hex digit) values, you are ready to approach the programmer.

As received, EPROMs should be unprogrammed, which means that all of the bits are set to "1". The programmer is used to set certain bits to "0". You get to decide. Later, you can erase the contents by putting the device under a short-wave UV light for about 20 minutes. That sets all of the bits back to "1". Be careful not to look at short wave UV. Unlike the longer wave UV used in "Black Lights", this stuff can seriously damage your eyes. Since you can't see it, you can't tell that it is happening. There is a UV EPROM eraser in the lab. (Or, there will be.)

You can use the DATA I/O 201 (stand alone) programmer. We have another programmer, the EE Tools Unimax, usable for this purpose, but it is a bit more complicated to use, since editing must be done via a computer. We will use it for the GALs. For the EPROM task, the DATA I/O 201 is probably easier to use, and it does not tie up a computer. It can program EPROMs but not PALs or GAL's. It uses keys to navigate a menu that lets you specify the device, edit, blank check your device, and program, as well as a variety of other operations.

To program the EPROM, set the programmer for your manufacturer and part number. (The part number might be 27256 instead of 27C256, the NMOS rather than CMOS version.) Now go into edit mode. I recommend you first fill all of memory with "ff" (all 1's) using the "fill" command. This leaves those addresses unprogrammed (and hence puts less stress on your device, as well as being quicker). Then, for each of the first 16 addresses (address 0000 to 000f,

hexadecimal) type in the hexadecimal values you want at those addresses. Then give the command to program the EPROM. You only need to program the first 16 addresses (which is very quick). Note that all of the numbers used in programming are in hexadecimal.

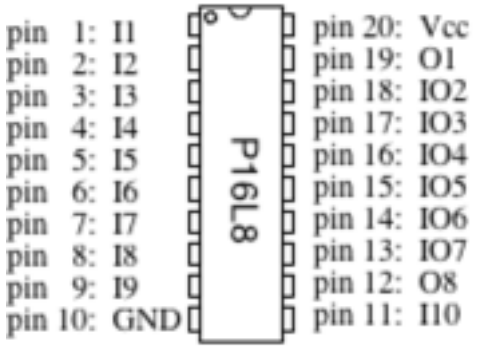
Once your part is programmed, put it into your multiplexed display (in place of the earlier circuit), or drive a separate display device. Ground A3 temporarily, since your multiplexed display has only has 3 bits, for A2, A1, and A0. (You can also try it with A3 to a logical "1" to get the second half of your expanded set of characters. Demonstrate that it works.

2. GAL

You will also implement your set of characters with a GAL16V8. The GAL will be roughly similar, though with perhaps an extra step or two. The GAL16V8 part has up to 8 outputs, and replaces combinational logic with 10 to as many as 16 inputs (those beyond 10 are at the expense of the outputs).

The device performs AND-OR-INVERT logic (or AND-OR logic). See page 377 of the textbook. Each OR has 7 AND (and potentially 8) gates feeding it. "Fuses" connect the any of the16 possible inputs (or their inverses) to the AND gates, so each "AND" gate has 32 (one row) of fuses. Each output can be set to "3 state" (high Z) controlled by an AND gate that can also be programmed to any of the inputs. We will not be using that feature.

Programming a GAL is a matter of "blowing the fuses" we don't want, leaving only the connections that we do want. The "fuses" (terminology from similar bipolar PAL's) are actually memory cells that can be reset electrically. The GAL (Generic Array Logic) is reprogrammable, which is one reason for preferring that type of device. (In fact, it is hard to find the bipolar 16L8 PAL parts now since the 16V8 can do the same and many more things, and can be reprogrammed.) A 16L8 is a TTL device, and will not be harmed by static hazards of reasonable proportions. The GAL is static sensitive, but still fairly robust. It is designed to act like the bipolar PAL's. The units we bought are 25ns parts; you can get GAL's down to 5ns or so. The 16V8 (and PAL16L8) pinouts are shown below. You can also find information in your book.



Power connections are at the corners, similar to most TTL chips. Pins labeled I1 to I10 can be used for inputs. Those labeled O1 and O8 can only be used for outputs. Those labeled IO2 to IO7 can be used as either inputs or outputs (or both, if you don't mind an output being connected back to an input).

To turn the GAL16V8 into a useful device, it needs to be "programmed." It is possible to directly edit in the fuse patterns and program it that way, but doing so is tedious and prone to human error, especially for larger devices. Instead, we will use a software tool called "WINCUPL" to develop the fuse maps, which will be written into a file in "JEDEC" format. We will then transfer the file into the programmer and then program the device.

We are using WinCUPL (instead of ABEL, which is described in the book). (WinCUPL documentation will be placed in the lab.) It is roughly similar to ABEL, but there are some differences. See example WinCUPL files on the lab computers. The book ABEL material is still relevant:

You must write a WinCupl "source" program. Lots of examples are found in the "examples" folder. As with other programming languages, the "source" file is in ASCII text. You can write it in any convenient editor, as long as you end up with ASCII text. You should do this outside of the lab, then bring your source file ready to (try to) compile. A very simple ABEL source file is shown below. See other examples in the textbook. (Then, check WinCUPL examples to see the differences.) There may be some minor variations between what our version accepts and the examples in the book. If in doubt, go by the example files. Some additional materials will be provided.

[This example needs to be replaced.]

```
module add4b
title 'four-bit adder constructed of eight one-bit adders'
      add4b device  'g16v8';
   a0..a3   pin 1..4;              "operand 1
   b0..b3   pin 5..8;              "operand 2
   c1..c3   pin 15..12 istype 'com';   "Adder carry bits
   s0..s3   pin 19..16 istype 'com';   "Sum bits
   c_out    pin 11 istype 'com';   "Carry out

equations
  [ c1   ,s0] = [.x.,a0] + [.x.,b0] + [ 0 , 0];
  [ c2   ,s1] = [.x.,a1] + [.x.,b1] + [ 0 ,c1];
  [ c3   ,s2] = [.x.,a2] + [.x.,b2] + [ 0 ,c2];
  [ c_out,s3] = [.x.,a3] + [.x.,b3] + [ 0 ,c3];
test_vectors([[a3..a0],[b3..b0]]->[[c_out,s3..s0]])
           [     1  ,     1  ]->[        2     ];
           [    5   ,   7  ]->[      12      ];
           [     6  ,    14  ]->[      20      ];
           [   13   ,   15  ]->[      28      ];  "Test carry
end
```

This program has the basic elements you also will need. A "module" statement identifies the name of the device. (You might want to use your initials, since everyone is doing the same kind of function.) The title information in quotes is essentially comments, which can span several lines if you wish. The "device" statement (which should start with the module name)

specifies that a 16V8 is to be used. The format for WinCUPL differs. See the WinCUPL examples. The programmer can actually take code compiled for a PAL and put it into a GAL.)

The next several lines, like variable declarations in C, identify the names of signals. They can also be used to specify which pins you want them to connect to. (If you don't specify pins, ABEL or WinCUPL will assign the signals to pins as it wishes.) The stuff after the quotation marks is a comment; you can put comments at the end of any line. The "istype 'com' " means that the output is combinational (the only kind available on the 16L8). Notice that we can put signals in groups (A0..A5 for example) rather than list them all. With the GAL you can also define sequential logic. We used to do that later, but will do Lab 9 with FPGA's instead.

After inputs and outputs (and perhaps other signals that are neither) are defined, we now get down to the business of saying how the outputs depend on the inputs. There are several ways to do this. You can use arithmetic statements as is done above, Boolean equations, or even give a table (much like you worked out for the EPROM). Note that ".X." means "don't care", .L. means LOW, .H. means high, and .Z. means high Z. Here are examples from other programs:

```
      ON, OFF  = 0,1;
truth_table (bcd -> [ a ,  b ,  c ,  d ,  e ,  f ,  g ])
          0  -> [ ON,  ON,  ON,  ON,  ON,  ON, OFF];      "(only the first line is shown.)
```

In this case, when the (4 bit) value "bcd" is 0, the outputs a to g are variously ON or OFF. the meaning of "ON" and "OFF" are defined to be 0 and 1 respectively, so that we get a 0 to turn something on.

```
equations
      when (select == 1) then Y = A;
      when (select == 2) then Y = B;
```

Here we see an "if" type structure, where select, Y, A, and B are signals.

```
equations
   Count = ((Count.q + 1) & (Mode == Inc)
         # (Count.q ) & (Mode == Hold)
         # (Data ) & (Mode == Load)
         # (0 ) & (Mode == Clear))
```

This is a Boolean equation. The symbol "!" means "not", "&" means "and", and "#" means "or". One can also use an alternate set of symbols (after a "@dcset command"?). Keep in mind the above is for "ABEL". There are many examples of WinCUPL programs in the "examples" directory.

The last section of the program is a series of test vectors. This shows what you are supposed to get at the outputs given what you put at the inputs. It is in tabular form. A GAL can be reprogrammed if it doesn't work right, so it's not as critical an issue now.

Once WinCUPL has compiled your design without errors (do a device dependent compile), you need to transfer the JEDEC ".jed" file from the computer to the programmer. On the PC, start the Max Loader application. It has a fairly straightforward interface. You need to identify the manufacturer and device. What you have is a GAL in a DIP20 package. Do not use "as PAL16L8" or other similar definitions, unless you had compiled for a 16L8 or such. This feature allows the use of programs meant for the earlier bipolar devices on the newer GAL's. Be absolutely sure to get the correct suffix letter. A GAL16V8 and a GAL16V8D program differently, and you can destroy your device if the selection is incorrect.

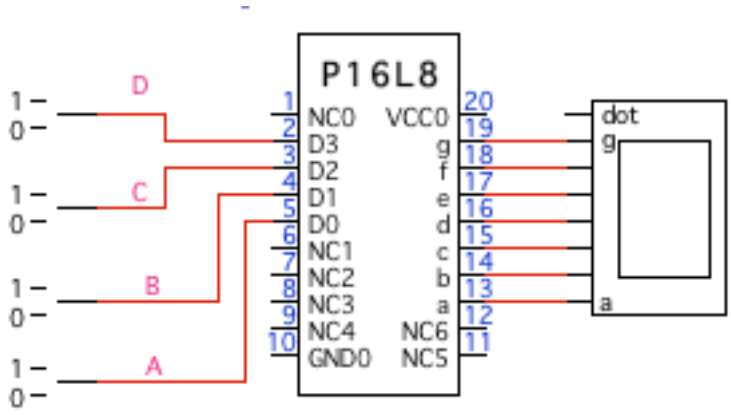
If the part you are programming is a 16V8, place the part in the socket. The end with pin 1 is put toward the top. The lever of the socket should be up when you insert the device. After the device is in, lock it down by lowering the lever. Check to see if the device is blank by clicking on the button to check it. You should see that the device is blank, as shown by a green LED at the socket and a message on the screen. If it is not blank, erase it.

Next "load a file" into the programmer application by selecting the ".jed" file you got from running WinCupl. Load it. You can look at the fuse map in the editor. Do so. See if it looks like your .jed file.

Now, program the 16L8 by clicking the button "program." The programmer will check the function of the device against the test vectors provided and indicate success (or not). One way or the other, your device is programmed. Remove it from the socket, plug it into your breadboard, and see what it does. Hopefully, it will behave correctly, performing your seven-segment decoding function. You can "read" a device then pass its fuse map back to the PC, or edit the fuse map in the programmer, and then program the chip again. You can always erase a device and start again if need be. (Of course, it is possible to damage a device, especially if it is programmed using the wrong settings, such as programming a GAL16V8D using settings for a GAL16V8B. If it is damaged, the device will typically not erase, or you get errors when you try to program it.

Results and Reporting

The timing on this lab report is not a major concern. Because of the limits on resources (only one DATAI/O 201, and one EE tools programmer), it is advantageous to be doing both this lab and Lab 6 (Logic Analyzer) at the same time, as appropriate given contention for resources. You need to do Lab 6 prior to replacing your discrete logic with a PAL / GAL / EPROM but there's no reason to wait to do your programming. You can even test the programmed part with a simple test circuit. It doesn't even have to be demonstrated in the multiplexed display.



This figure shows the use of a ".dwl file" in Logicworks to simulate with the part.

Turn in a short informal report that includes:

1) Your truth table for the EPROM and PAL (including the hex for the programmer).
2) Identification of the inputs and output signals for the ROM and GAL.
3) Your WinCUPL program source for the GAL.
4) Your JEDEC file for the GAL.
5) A short statement reporting success (or lack of same) for the GAL and EPROM.
6) Any other comments or remarks on issues that are relevant or interesting.