Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.115 Microprocessor Project Laboratory Spring 2024

Laboratory 0 (Pre-lab)
Waiting for Skynet

Issued: February 6, 2024 Due: February 13, 2024

GOALS: Learn useful stuff before we issue kits ☺.
Get familiar with the 8051 assembler AS31 and simulator Edsim.
Install and see PSoC Creator, familiarize yourself with C.
Practice writing organized, logical code.

PRIOR to starting this lab:
READ "Reading Schematics" on the last page of this lab.
Complete your check-in and kit issue form and return to Professor Leeb.
Visit: web.mit.edu/cdev and download JAVA (if you need it), the AS31 zip, Edsim, PSoC Creator, ExpressPCB, and the Datasheets Zip file.

Hardware is coming!

This is a class about building systems – not just hardware or software or mechanical components or sensors, but rather, the magic we can create putting all of these components together to make powerful solutions. Your gear will materialize next week! First, let's take advantage of the time to begin learning how we will program the 8051 and PSoC hardware. DOCUMENT your work, including commented code and Edsim screen captures, in a careful lab report that, for this Lab 0, you will submit as a PDF to the course website. Use Microsoft Word or similar for preparing your lab report. Write as you work! Use your phone or screen capture to insert pictures of your work. (We will use lab notebooks in later labs.)

EXERCISE 1: Install your software

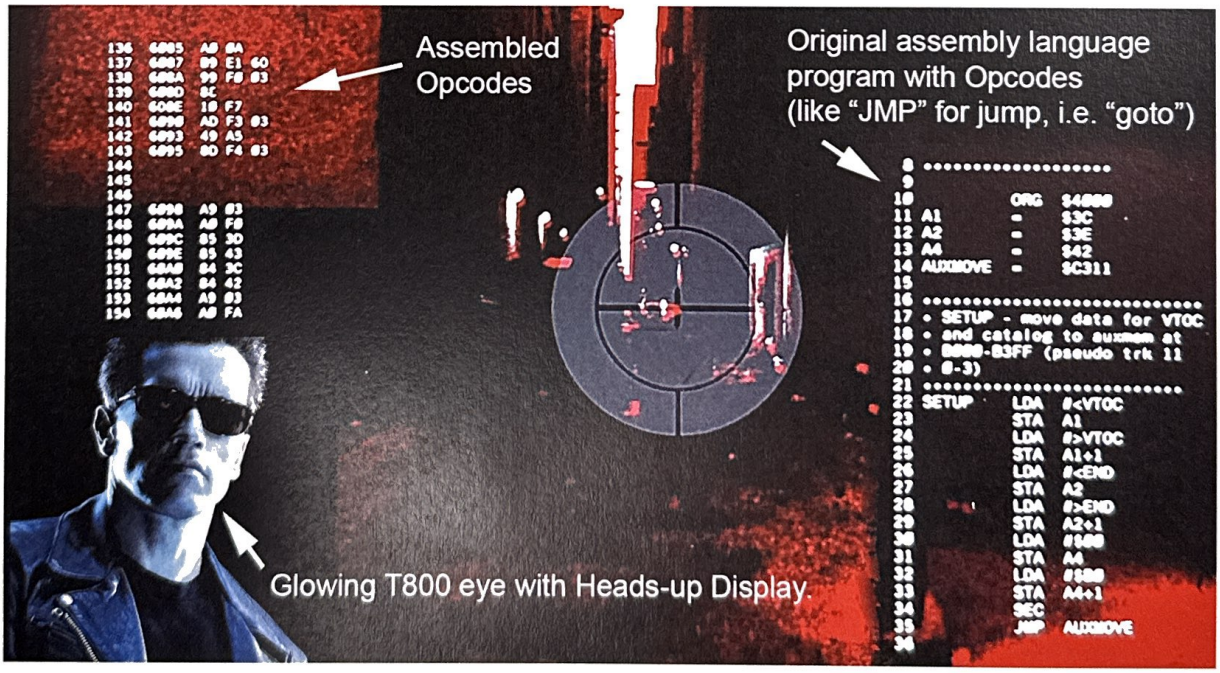
Please go to **web.mit.edu/cdev**, where you can download the AS31 zip file, PSoC Creator, the Edsim zip file, the JAVA installer, ExpressPCB, and the Datasheets zip file.

Please do the following:

- Download and install ExpressPCB. You will get two executables when you run the installer, ExpressPCB and ExpressSCH. The "SCH" program is useful for drawing clean schematics for your lab reports. You can find useful "custom components" for 6.115 at the course website, here: http://web.mit.edu/6.115/www/page/miscellaneous.html
- If you do not already have it on your Windows 10 machine, install JAVA. Download and run the "JAVA installer" exe file from the website. You need the JAVA interpreter to run Edsim.
- Download the Edsim zip file and unpack the contents into a convenient directory on your Windows 10 machine. You should find an Edsim51di.jar file in the directory. Assuming JAVA is on your machine, if you double click on Edsim51di, you should see the Edsim simulator window. Try this.
- Download the Datasheets zip file and unpack the contents into a convenient directory on your Windows 10 machine. Scan the contents. The "INTEL_8051" pdf file is an extremely useful data book about the 8051 microcontroller that we will be using.
- Also download the PSoC Creator installer. Run the installer on your Windows 10 machine. Make sure you can run Creator on your machine. We will use this later with the Cypress PSoC.
- Download the AS31 zip file. Open the zip file, and copy the contents to a convenient directory on your Windows 10 machine. You can read AS31.pdf and AS31-TIPS.txt, both in the zip file, for useful tips about AS31. Don't panic. The tips will make more sense as we start using AS31.

EXERCISE 2: Assembly language party with AS31

So here's a potential surprise: the T800 Terminator is programmed in assembly language! Check it out in this scene from the first Terminator movie – a "must-see" classic:



I have added a few annotations in the picture so you can notice the details. In the inset picture, behind the sunglasses, we see the T800's eye glowing to reveal a HUD (heads-up display). The "rest of the image" is what the Terminator sees, with a targeting reticle in the center. To the right of the targeting reticle, we see an "assembly language" code listing, with arcane but "almost readable" instructions that a programmer could write or understand with reference to an appropriate data book. For example, the "JMP" instruction at the end of the listing (line 35 in the code listing) is a command to "jump" or "goto" or transfer program execution to a subroutine named "AUXMOVE" somewhere in program memory. The Terminator's processor core stores and manipulates information in binary voltage levels, i.e., logical "zeros" or "ones" that are represented as a zero-volt level ("low") or a five-volt or 3.3-volt level ("high") in the processor circuitry. That is, the Terminator is not capable of directly using the opcode program. The opcodes are a language for the convenience of the programmer – soon to be you. The finished opcode program must be converted into numbers – ultimately, binary numbers – that can be loaded into circuit memory. We can conclude at least three things from the picture above:

1. The T800 Terminator probably uses more than one processor, thus explaining why the HUD might be displaying code that will eventually be loaded on a sub-processor. The code is being written or reviewed by the main processor in the T800's "brain."
2. The Terminator writes commented code! Bravo! Please follow this example!
3. To breathe life into your projects, you will want to know how to write assembly language opcode programs and "assemble" the programs into a form that our 8051 hardware can use.

Where to begin? The notation "JMP," for example, is an operation code or "opcode" that a programmer selects from a list of possible instructions that a particular processor has been designed to perform. You can find a complete listing of opcodes for the 8051 in the "INTEL_8051" pdf in your datasheets zip file. Officially called the "MSC-51 Microcontroller User's Manual," we will refer to this pdf as the "Intel Manual." A listing of all available opcodes is provided in Chapter 2 of the Intel Manual, pages 2-21 through 2-27. Additionally, pages that follow in Chapter 2 provide individual explanations of each opcode instruction. For example, we find this explanation of the "SJMP" instruction on page 2-69:

SJMP rel

Function: Short Jump

Description: Program control branches unconditionally to the address indicated. The branch destination is computed by adding the signed displacement in the second instruction byte to the PC, after incrementing the PC twice. Therefore, the range of destinations allowed is from 128 bytes preceding this instruction to 127 bytes following it.

Example: The label "RELADR" is assigned to an instruction at program memory location 0123H. The instruction,

SJMP RELADR

will assemble into location 0100H. After the instruction is executed, the PC will contain the value 0123H.

(*Note:* Under the above conditions the instruction following SJMP will be at 102H. Therefore, the displacement byte of the instruction will be the relative offset (0123H-0102H) = 21H. Put another way, an SJMP with a displacement of 0FEH would be a one-instruction infinite loop.)

Bytes: 2

Cycles: 2

Encoding:

1 0 0 0	0 0 0 0	rel. address

Operation: SJMP
(PC) ← (PC) + 2
(PC) ← (PC) + rel

As designers programming our microcontroller, we might like to refer to this instruction as "SJMP," an opcode that almost sounds like English ("short jump"). The 8051 processor hardware, however, "understands" only the binary code of high and low voltages that correspond to this SJMP command. The Intel manual is a kind of Rosetta stone that lets us translate from the almost-English opcodes to the binary numbers that correspond to the opcodes. In this case, note that the Intel manual reveals that SJMP is an opcode that assembles or is expressed as the 8-bit binary number 10000000b. In this case, the "most significant bit" or "MSB" is a one, and the "least significant bit" or "LSB" is a zero. The "b" indicates that this is a binary number and not just a list of ones and zeros. An eight-bit binary number is also sometimes called a "byte," which is made of two "nibbles" or four-bit binary numbers. The two "nibbles" in this case are 1000b and 0000b. We can also refer to this number as 128d or 128 in decimal, or 80h in hexadecimal. That is:

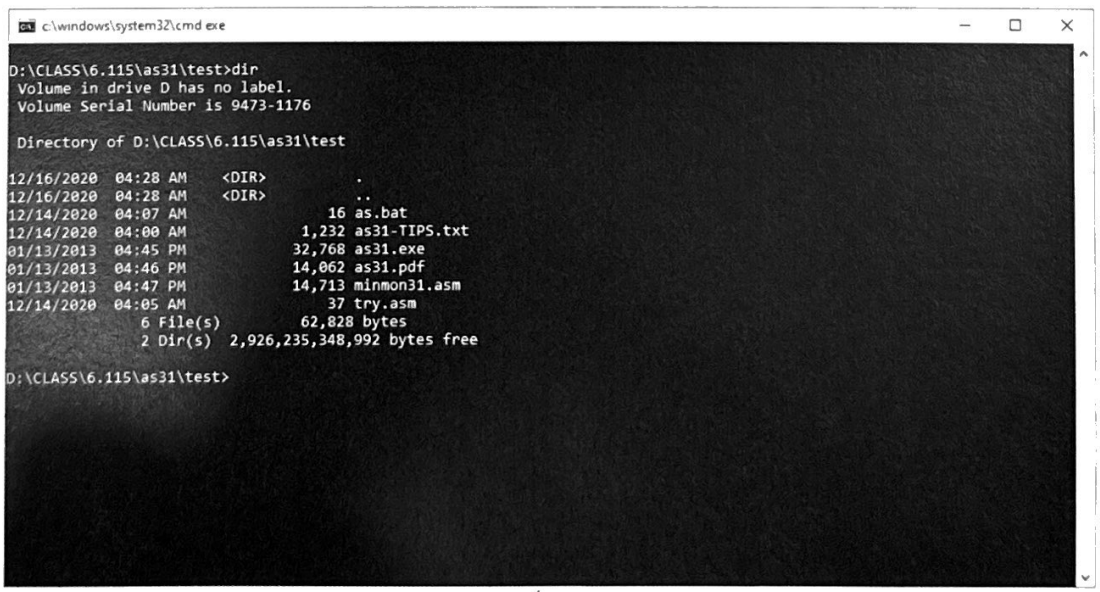
$$10000000b = 128d = 80h$$

You can read more about these number systems, and how to translate between them, here:

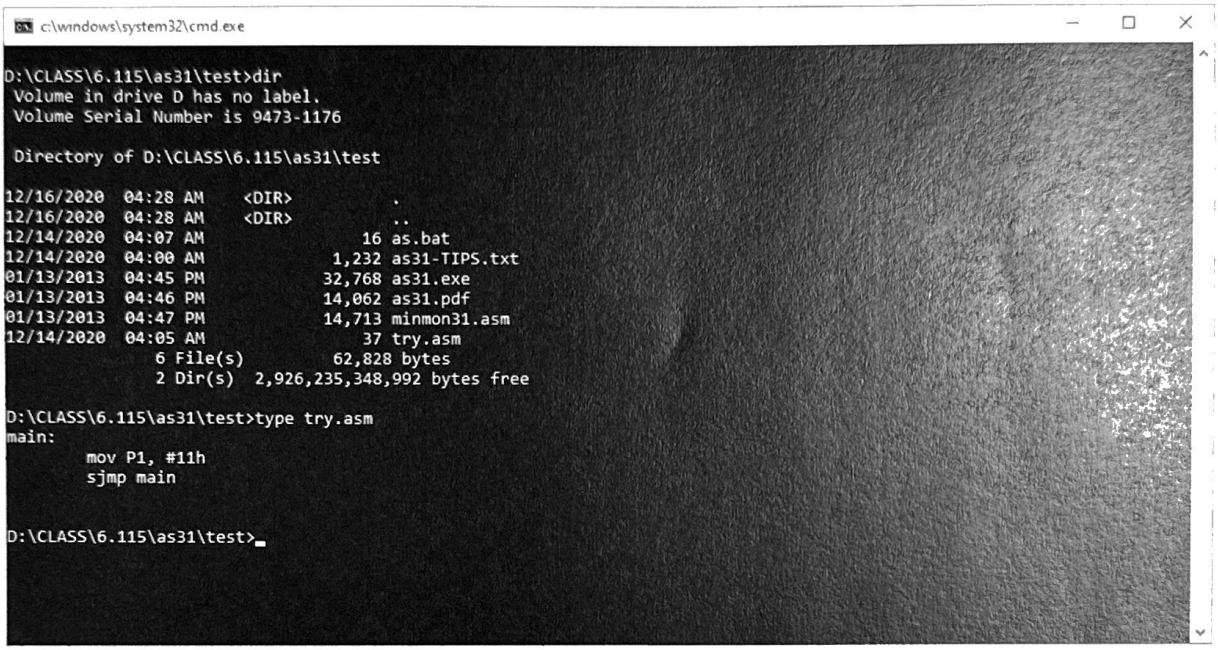
https://www.mathsisfun.com/binary-decimal-hexadecimal.html

We can therefore use the information in the Intel manual to know that the SJMP opcode corresponds to a "HEX code" of 80h. Writing a good program in assembly language with opcodes is a lot of fun and a great way to hone your skill as a programmer of any language. No experience will get you closer to understanding the interconnection between code and hardware. However, translating your program from opcodes to the corresponding HEX codes and then loading these HEX codes as binary numbers to processor memory is tedious and less fun. Fortunately, we don't have to do the translation ourselves. We use an "assembler" that reads a text file of opcodes written by a programmer. The assembler converts the program to a "HEX file" that is easy to load into the microcontroller's program memory. The process of programming the 8051 microcontroller, therefore, involves first writing a text program in opcodes on your PC; then, assembling the text program to a HEX file; and finally loading the HEX file numbers into hardware memory. The specific assembler we will use is called AS31, and you should now have this executable file in a directory on your Windows 10 machine.

You "run" the AS31 assembler in a CMD window in Windows 10, sometimes called a DOS window, very similar to an xterm window you may have seen in linux or on an Athena machine. Here's the directory on my machine where I unpacked the AS31 zip file, shown in a CMD window:



Note that this directory contains the file "try.asm" that we have provided for you. The file was written in a "clean" text editor, in this case, Notepad, which comes "for free" in Windows 10. You can use any clean text editor you like for this class, but do not use word processing programs like Word, which insert formatting characters and other invisible-to-the-eye information that will confuse the assembler. Programs like Notepad, Emacs, and VI are fine. Choose something you like. I can see the contents of try.asm by using the "type" command in the CMD command window:



Note that "try.asm" is a **four**-line program ending **with a required carriage return** ("Enter" on your keyboard), reproduced below:

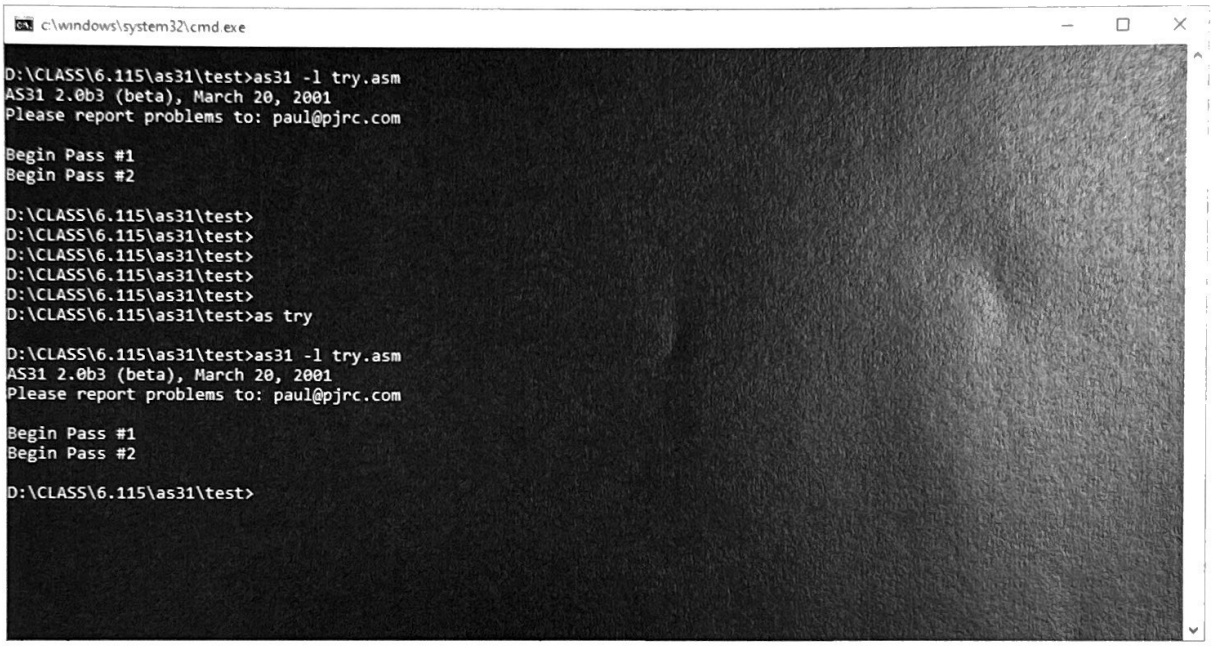
```
main:
    mov  P1, #11h
    sjmp main
```

The first line with "main:" is just a label that indicates the start of the program. It did not have to be called "main," and in fact did not have to have a label at all. It's nice to have a label to identify routines or subroutines, that is, parts of your program that might provide a specific function, behavior, or response inside of a larger overall program. The labels (like main:) help a person with reading the code, and they are also convenient in case we want to jump to a particular routine or program location.

The second line uses a hardware register in the 8051 hardware called "P1". This register controls the state of the voltages on a "port," eight of the metal pins on the 8051 integrated circuit (IC) package. These pins might, for example, be connected to LED lights. So the second line of code would set the pin states to 00010001, and two of the LEDs would be "glowing".

After P1 has been set, the third line with SJMP transfers program execution back to the beginning of the program, and the whole process repeats endlessly in an "infinite loop" until we reset the processor.

Once I have written a program like "try.asm" in my editor, I want help doing the next labor of converting the program into numerical opcodes for the 8051. We will use AS31. If you simply type "as31" at a CMD window prompt and hit return, you'll get a "usage" message that lets you know some of the options for using the assembler. A convenient command for us for assembling a program like "try.asm" might be: as31 –l try.asm. Here's what it looks like in the CMD window:



Note in the example above that I have called AS31 in two different ways, both of which are equivalent. In the first case, I used the full command line: as31 –l try.asm. In the second case, I used a batch file that we provided for you in the AS31 directory and zip file with the simpler command line: as try. Both options produce the same result. Specifically, assuming no mistakes in the try.asm file, we get two files: try.lst and try.hex. You can look at them after the assembler runs by using the "type" command in the CMD window. Here's what you'll find in the LST and HEX files:

LST FILE:

```
                        main:
0000: 75 90 11          mov P1, #11h
0003: 80 FB             sjmp main
```

HEX FILE:

```
:0500000075901180FB6A
:00000001FF
```

At each stage of the process, from ASM to LST to HEX file, the file becomes less easily readable to a human and more easily readable to a machine. The LST file is the Rosetta stone that connects the human programming to the machine world. Examine the LST file carefully. On the right in the file,

you see the original "human" code from our try.asm file, essentially identical to what the Terminator sees on the right side of the HUD. The left side of the LST file is the same program, but this time converted into HEX codes, closer to what the machine will understand, along with the starting memory location for each line. This is similar to the listing on the left side of the Terminator's HUD.

Look at the LST file carefully. The "line numbers" like "0000:" and "0003:" are the locations in program memory where the assembler thinks that the code should be stored. The flag "main:" is not an opcode, so it does not translate into any HEX code. This "main:" flag is just a directive to the assembler to indicate that the programmer wants to refer to address location 0000h with the name "main," rather than writing out 0000h. The next line, "mov P1, #11h" will be loaded in memory and will occupy three bytes at 0000h, 0001h, and 0002h. The three bytes for this line of code are the opcode for MOV (75h), the address location for Port 1 (P1, or 90h), and the data byte that will be moved to P1 (11h). The next line of code therefore starts at memory address 0003h with the opcode for SJMP, which we already know is 80h. The last byte (FBh) is the relative jump to return back to the memory address for main (0000h). We will talk more about this, but, FBh is a "negative jump" for SJMP, if you read the Intel Manual carefully. You can find these HEX codes in the HEX file if you look carefully. They are surrounded by some other formatting bytes that we will discuss in lecture.

Please do the following:

- Use AS31 to assemble the try.asm file and produce a LST file and a HEX file. Examine the LST file and the HEX file, and confirm the observations made in the discussion above.
- Make your own try2.asm file with a total of five lines. This new ASM file should move the byte 11h to P1 as before. Then move 22h to P1. Then move 33h to P1. Then loop back and do it endlessly. Assemble your try2.asm file to produce LST and HEX files. Be prepared to explain your LST and HEX files. (Don't worry for now about the "extra" bytes in the HEX file. Just find your program HEX codes.)
- Make your own try3.asm file. This file should move the byte 11h to P1. Then move 22h to P1. Then the program should enter a loop that endlessly writes 55h to P1. Assemble your try3.asm file to produce LST and HEX files. Be prepared to explain your LST and HEX files.
- Why is it important that all of our files end in a controlled way, with a loop to somewhere? Suppose your entire program was just one line: mov P1, #11h. If you try assembling this one line with AS31, it should assemble with no trouble. Why is this program a bad idea? Why would the following three line program, which achieves the same goal for P1, be a better idea?:

```
mov P1, #11h
loop:
sjmp loop
```

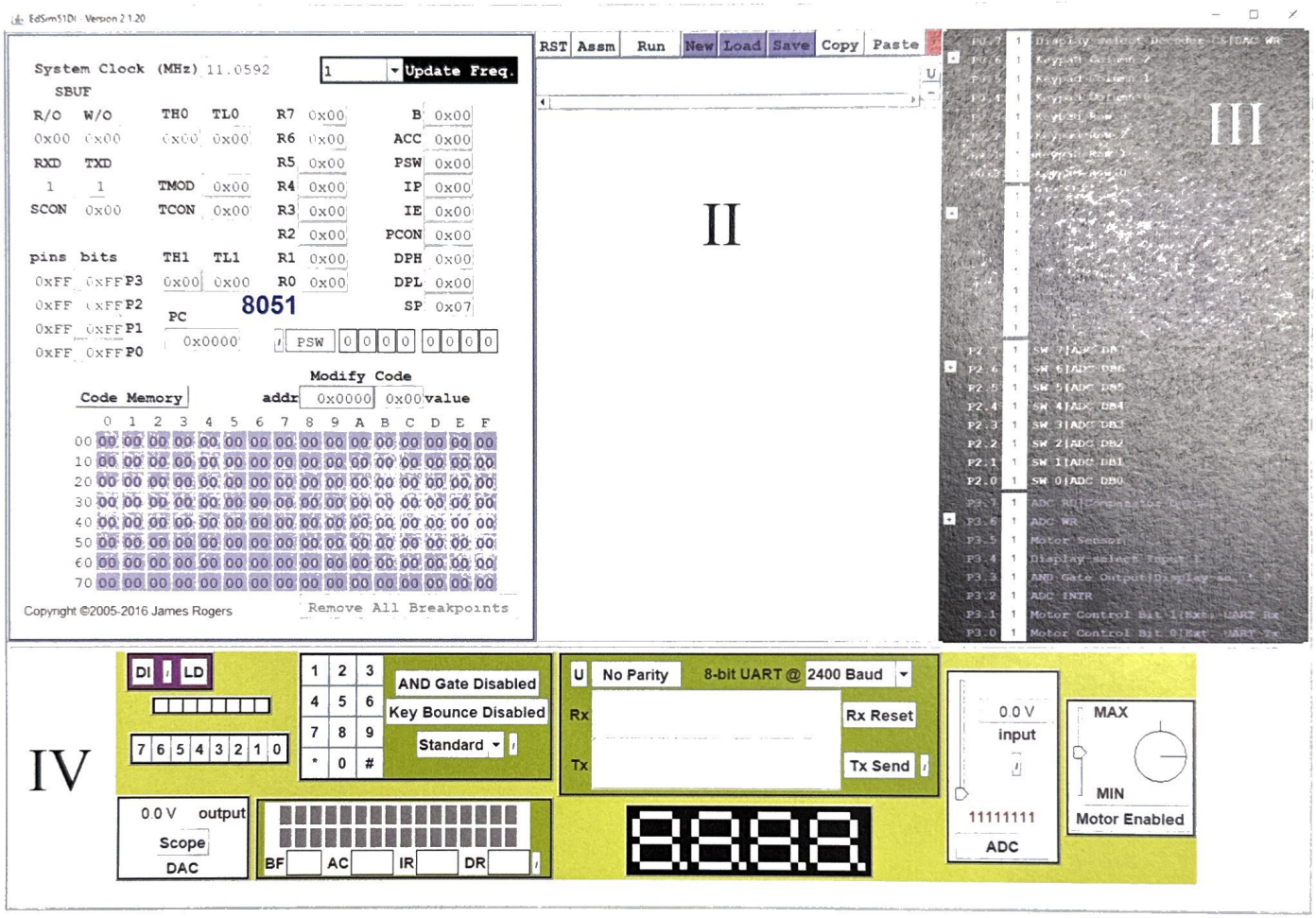
- Based on the information you have in the pictures above and the Intel manual, is the T800 Terminator using an 8051 family microcontroller?

EXERCISE 3: Run your code!

Once we have an Intel HEX file, we want to test the code! While we are waiting for the arrival of hardware, we can use an emulator for the 8051 microcontroller. Among many other possibilities, this emulator, called Edsim, will let you load the HEX file created by AS31 and "run it". Of course, nothing substitutes for real hardware. However, Edsim offers some interesting capabilities for testing and debugging code. You can read about Edsim, including a "User's Guide," at this website:

https://www.edsim51.com/

Make sure you have loaded JAVA on your Windows 10 machine, and then confirm that you can run the Edsim51.jar file in your Edsim directory. When you run Edsim, you will see this window on your PC:



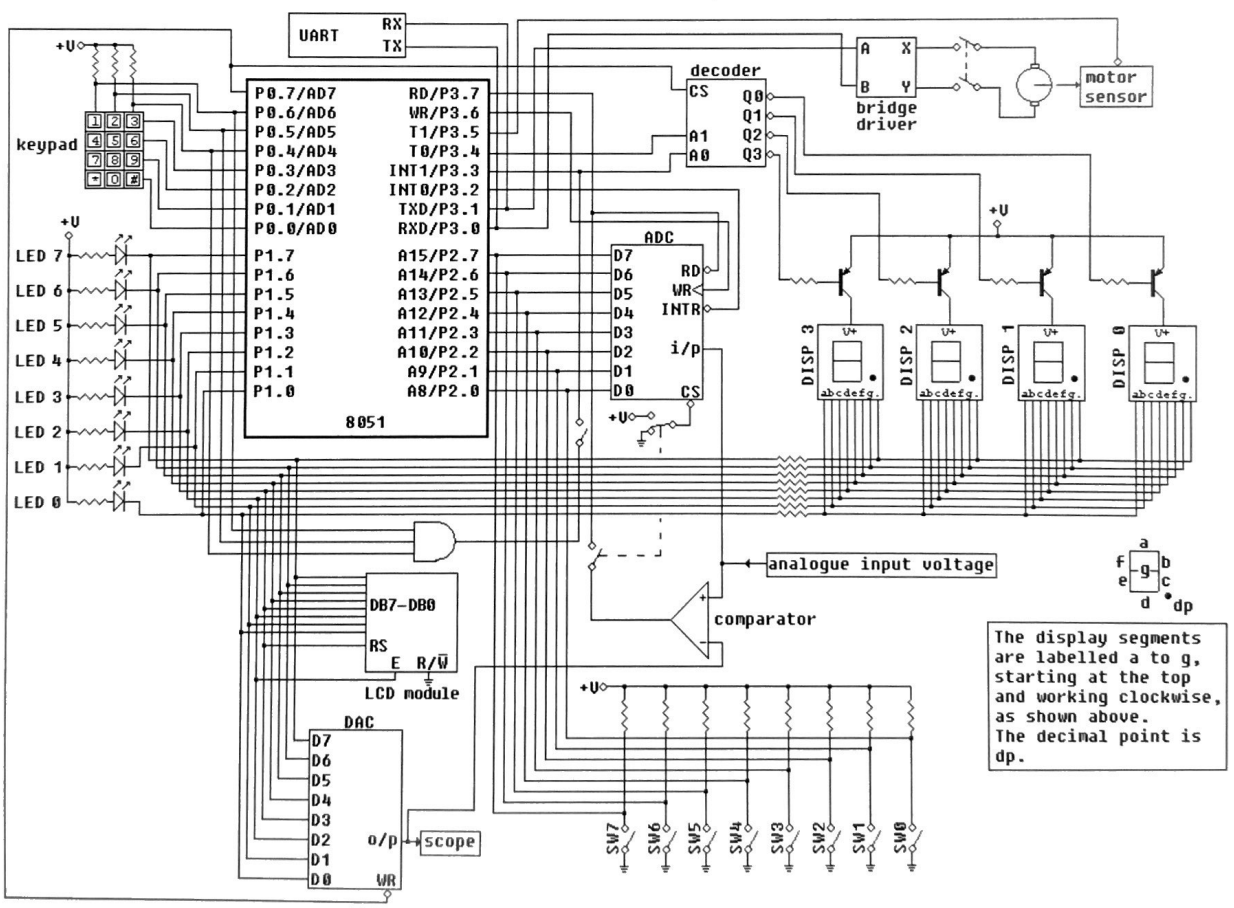
The Roman numeral annotations indicate four different sections within the Edsim window. The Roman numeral I indicates an area of the Edsim window that shows the state or contents of the internal 8051 registers. This is exciting because we can "see" inside the 8051, not normally possible with hardware, but easy in the emulator. Notice how many registers are present in the 8051. We will discuss many of them in class, including that the program counter PC that indicates where execution is occurring in program memory, and the P1 register that we already discussed in Exercise 2. A description of the rest can be found in the Intel Manual. This section also shows the "code memory" that contains the HEX codes for the program we plan to run. Notice also that, in I, you can set the system clock frequency and the simulation update frequency. Make sure that your system clock is set to 11.0592 MHz; this is the frequency for the R31JP hardware we will give you. You can choose the update frequency, which determines how often the screen is updated. An update frequency of "1" means that Edsim will update the screen after every instruction that it executes in your code. An update frequency of "100" indicates that the screen will update after every 100 instructions, and so forth. More frequent updates allow you

to watch the program and its behavior unfold on the screen. However, for more complex programs with lots of instructions, you may want to use a higher update frequency of 100 or 1000 to make progress through your program at a reasonable rate.

You can "load" or "save" code in the area indicated with II. The area labeled with III shows you the state of the simulated "metal pins" on physical ports P0, P1, P2, and P3. The associated registers may or may not contain the same values, e.g., a P1 pin might have a voltage on it applied by external hardware, but the P1 register may not reflect the physical pin state until the port is "read" with a MOV opcode. The area III presentation lets you see the actual hardware state of each of the 32 available port pins on the simulated 8051. Finally, area IV shows you the state or behavior of some emulated devices or "virtual peripherals" that Edsim assumes are connected to these port pins, including:

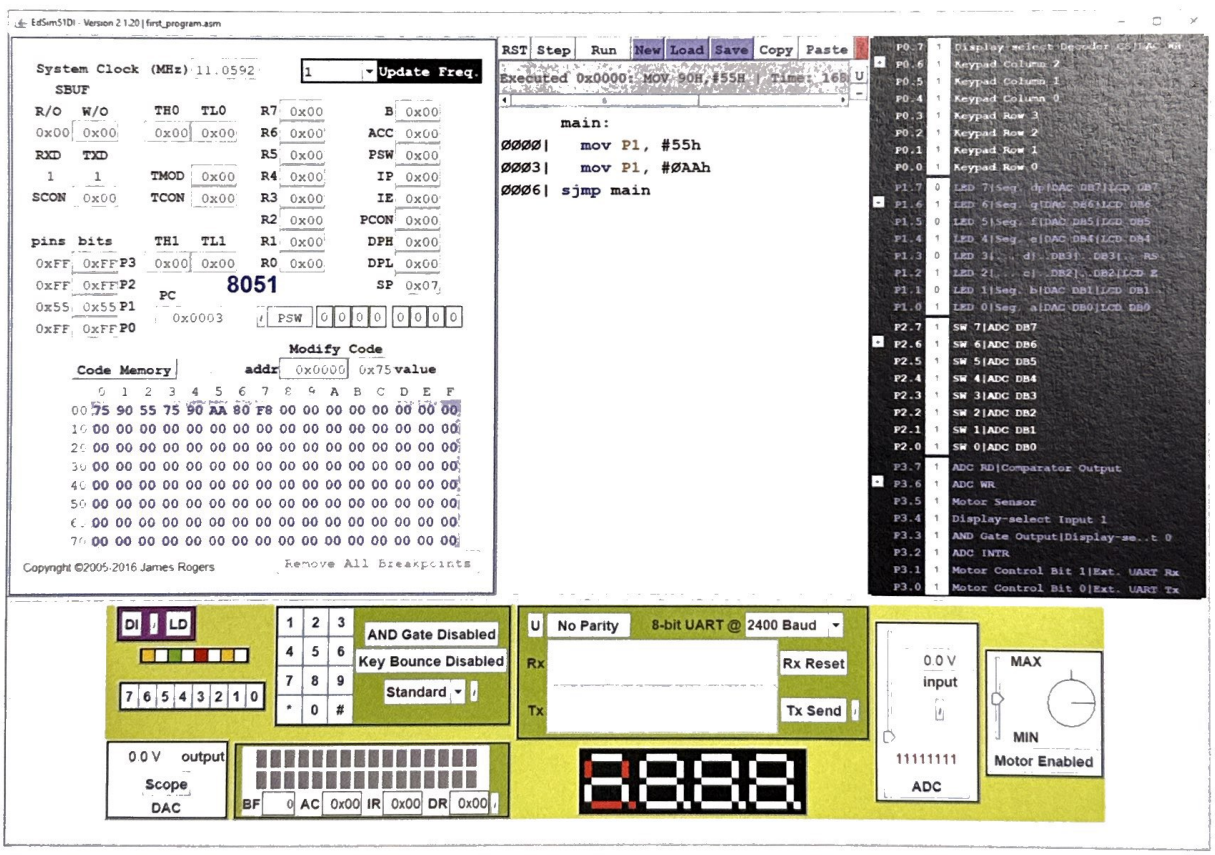
- Analog-to-Digital Converter (ADC)
- Comparator
- LCD module
- UART (Serial communication port)
- 4 Multiplexed 7-segment Displays
- 4 X 3 Keypad
- 8 LEDs
- DC Motor
- 8 Switches
- Digital-to-Analogue Converter (DAC) - displayed on a simulated oscilloscope

A schematic of the "imagined circuit" that Edsim is simulating is shown below:



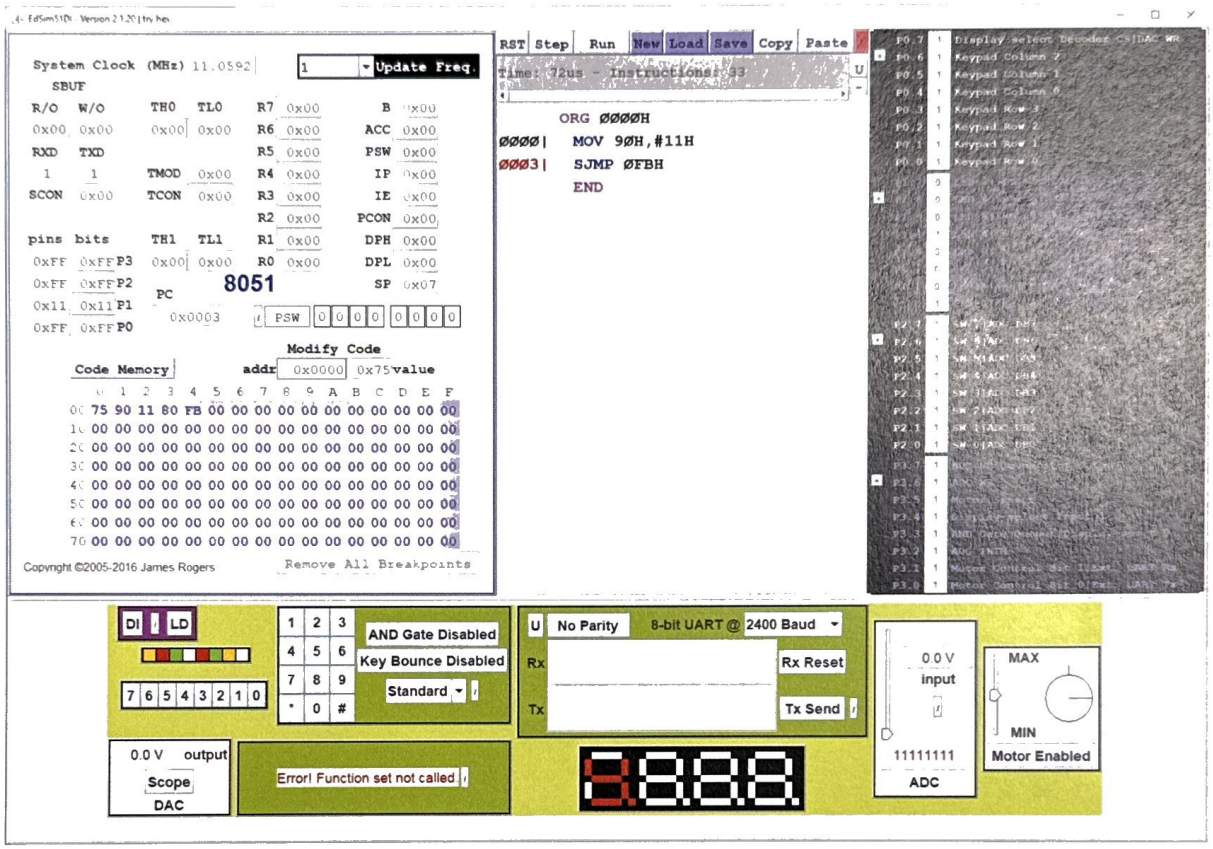
Don't panic! We don't expect you to understand all of these peripherals or even the 8051 fully yet. The 8051 microcontroller that Edsim simulates is shown in the upper left of the schematic. The peripherals like the light emitting diodes (LEDs) are shown on the schematic, connected to various port pins on the 8051. We will be discussing this in lecture, and reading about these items. Do take a minute to look at the schematic and see what might make sense. For example, the eight LED's, LED0 through LED7, are connected to the P1 or Port 1 pins. We can set the state of these pins on the 8051 by writing to P1. That means we can turn lights on or off in Edsim by writing to P1. Cool! Notice that Edsim is imagining that the LED's are connected to positive voltage (+V) all the time. The other end of the LED is connected to a P1 pin. If the pin is high at +V, then the LED will be off. If the pin is low at ground or zero volts, than there will be voltage across the LED. In this case, the LED will glow (a color, not white) in the Edsim simulator window IV. Recognize that this means that the lights in the simulator are "inverted". They will glow a color when we write a zero to the particular port pin for that LED. Notice also that the 8 segment character displays, good for showing numbers, are connected in parallel with the bank of individual LEDs. So, for example, if we write to P1 and make some LEDs glow, we will unavoidably also be writing to one of DISP 0, 1, 2, or 3 at the same time. Remember that when you write to P1, you will be writing to both the LED's and the digit displays.

Here's a screenshot of Edsim after I have written **and** assembled a program in the Edsim area II:



Try this yourself. Type the program shown above in the Edsim area II directly into your area II window. Then, play with Edsim. You can "Run" the program and watch it work. Notice also that you can "single step" through the program using the "Step" button in II. Watch the LED lights in area IV as you run or step through the program. Watch the state of P1 in areas I and III. Generally, we will avoid writing our programs directly in Edsim. For most of the term, our goal is to get real hardware to do real

functions. We will therefore use AS31 to generate "real code" for our target hardware. The Edsim internal assembler is not as full featured as AS31. So we will generally want to write our code in something like Notepad, assemble the code with AS31 to produce a HEX file, and then run the HEX file, either in hardware or in Edsim. Fortunately, Edsim lets us load an Intel HEX file for an 8051 microcontroller from any source and run it! For example, here is what Edsim looks like if we load in the "try.hex" file from Exercise II:

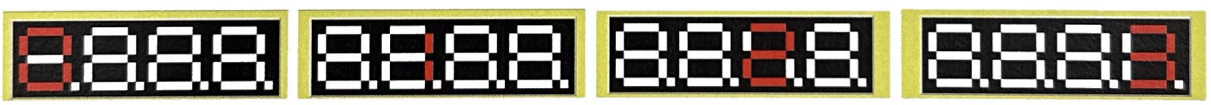


Notice several aspects of the screenshot above when you examine the try.asm code again. Note that Edsim area II does not show a "main:" label. Remember, the label is an assembler directive that marks a location for reference for the assembler. It won't show up in the Intel HEX file. Similarly, notice that the line beginning at 0003h in memory has the SJMP command, but again, no direct reference to "main". Again, the assembler has translated the reference to "main" to the necessary address offset argument (0FBh) for the SJMP command. So, it makes the most sense to read your code in your editor, e.g., Notepad, edit as needed, compile with AS31, and then run the code in Edsim. This way you're using a full-featured assembler (AS31) that can produce files directly for your hardware, but also getting the benefits of seeing Edsim run your code at the rate of your choice and with full visibility into the 8051 internal registers. Notice, for example, in the screenshot above in Edsim area I that you can read the state of P1, which contains 11h, just as we would expect from running try.asm.

Please do the following:

- In this exercise, and for all exercises in 6.115, save ALL of the variants of the code that we request in these bulleted "please do the following" exercises. Submit the carefully commented and presented code in your lab report for each request, and also please have your code solutions available for demonstration at the lab checkoffs.

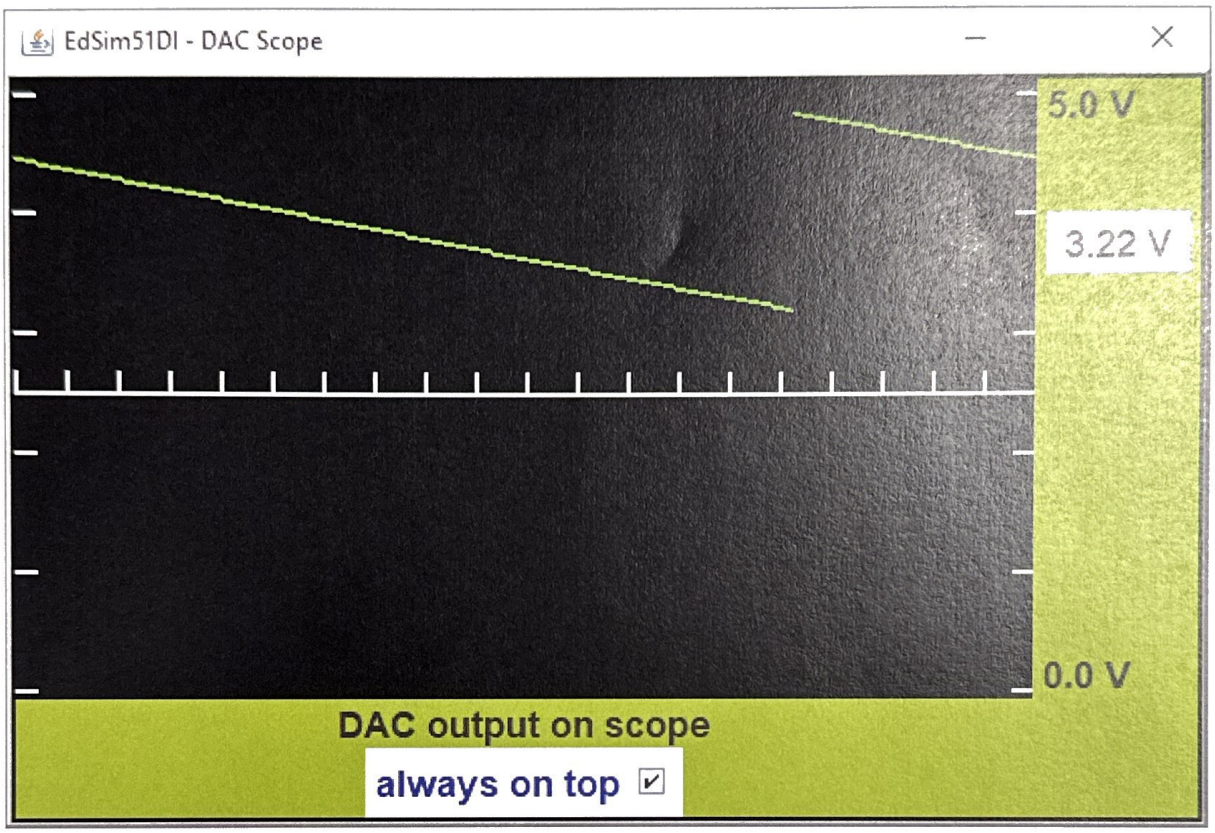
- Use AS31 to assemble the try.asm file (in your AS31 zip file) and produce a LST file and a HEX file. Load the try.hex file into Edsim. Run it using "step," and also using "run". Explain what you see in Edsim areas I, III, and IV, including comments on the state of P1, the "individual LEDs", and the character LED at the bottom of area IV.
- Notice that P1 could have any of 256 different values, ranging between 00h and FFh (same as 0d through 255d). Write a program in a file called try4.asm that cycles P1 through all possible values sequentially. In your program, be sure to use the 8051 opcodes: MOV, DEC, SJMP. Assemble the program using AS31 and then run your program HEX file in Edsim. What update frequency makes sense for watching Edsim execute this program? Explain what you see in each Edsim area I, II, III, and IV. Using your try4.lst file, explain what you see in the Edsim area I "code memory" section after you have loaded "try4.hex". Does your program count P1 up or down? What do the individual LED lights do in area IV? Why?
- Set Edsim for an update frequency of 100. Run your try4.asm program file from the previous exercise again. What do you see on the LEDs? Why? Fix it. Leave the update frequency at 100. Modify your code to use the DJNZ opcode and register R0 to create a reasonable delay that lets you see all of the counting action on the LEDs. Note that this is a "real" problem. The 8051 executes opcodes very quickly with a clock frequency of 11.0592 Mhz. (How quickly? Where do you find the timing information for each opcode?) At this clock frequency, we will often need to add delays or other tricks in order to interact with a user at "human" speeds.
- Now let's write a program that displays something interesting on the character LEDs, the "digit displays" that look like a number 8 at the bottom of area IV in Edsim. There are four digit displays. By default, Edsim will drive the leftmost digit display, which is labeled DISP 3 in the Edsim peripheral hardware schematic. Figure out the ten hex numbers or codes you would have to write to display each of the decimal numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 on the digit display. Then, write a program that counts down from 9 to 8 to 7 sequentially down to 0 and then repeats this endlessly. Use AS31 and Edsim to write and test your program.
- Let's try another program on the digit displays. This time, let's use all four digit displays, DISP 3 through DISP 0, one at a time. First, display the number "0" on DISP 3. Then "1" on DISP 2, then 2 on DISP 1, and finally 3 on DISP 0. Then repeat endlessly. Some points to consider: notice that you select which DISP display you want to use by properly setting port pins 3.3 and 3.4 (thus controlling the decoder). Use the opcodes SETB and CLR to set the pin states for pin 3.3 and 3.4. Also note that you should display a number on a particular DISP, but then clear the number from the display before moving to the next DISP. If you don't do this clearing step, your number will show up on two DISP's sequentially. Here's are snapshots of the digit display as the staff solution runs in Edsim; your program should create the same digit count progressively, looping endlessly:



Include screenshots of your LED displays in your lab report.

- There is another peripheral lurking on the P1 pins – a DAC or "digital-to-analog" converter. We will talk more about this part in class. But, for now, you can assume that it is a peripheral that takes in a digital input byte, from P1 in this case, and converts that digital byte to an analog output voltage. An input byte of 00h yields an output voltage of 0 volts. An input byte of FFh yields an output voltage of 5 volts. The output voltage scales linearly with the input byte between 00h and FFh. You can see the DAC voltage output in area IV in Edsim, lower left corner. Notice that, despite the fact that we have been writing to P1 in previous exercises, the DAC did not respond. Why? Because the DAC has a control pin (WR) that must be "toggled" to write a byte to the DAC.

The WR pin on the DAC is connected to a pin on Port 0 (P0), specifically, P0.7. Write a program that makes a voltage ramp like the one shown in the Edsim scope screen shown below. Make use of the opcode DEC. You can "view" the scope screen by clicking on "scope" in area IV. Use the opcodes CLR and SETB to lower and then raise the P0.7/WR line in order to "toggle" data into the DAC. Note that the Edsim "scope" is not much of a scope. It's more of a chart recorder with no ability to "trigger" for a stable image or to change time base to squeeze different amounts of the waveform on the "scope screen." With an update frequency of 1000, our staff code produced a "scope screen" in Edsim like this one:



Include a screenshot of your Edsim scope in your lab report.

- Continuing with the DAC, write another program to make a square wave that oscillates between 0.5 volts and 4.5 volts. Note that the scope will not show "edges" of the square wave, since the slew rate or rate of change between levels is infinite in the simulation.
- Let's make another version of the DAC square wave generator, with a little more flexibility. At the very start of your program, use the opcode PUSH to load the 8051 stack with a "high value" (like 4.5) and a "low value" (like 0.5). Then enter a main routine that uses the opcode POP to load the high and low values into R0 and R1. The main routine should use the opcode LCALL to call a subroutine named "dacme" that loads the value in the accumulator acc to the DAC. Create the square wave by having the main routine put a high value in ACC, call dacme, put a low value in ACC, call dacme, and then loop forever to continue creating the square wave. Then create a second version of this program that lets you adjust the duty cycle and period of the waveform based on values in R2 and R3. ("Duty cycle" is a fraction between zero and one that describes the relative amount of time the square wave is at its high value compared to the overall period.)

- Write a program that uses the opcode CJNE (more than once) to select and call different functions when given different input numbers in R0. Your program should respond to four possible command values in R0, the numbers 1, 2, 3 or 4. A command value of 1 should output AAh to P1. A command value of 2 should display a "3" on DISP 3. A command value of 3 should set the DAC output to 2.5 volts. Finally, a command value of 4 should set the DAC output to 4 volts. Note that, in Edsim, you can always pause the program, click on R0 in area I, directly modify the value in R0, and run the program again. That is, you should be able to explore the effect of different command values without having to rerun AS31 or retype new programming in Edsim area II.

EXERCISE 4: See about C.

We program the PSoC using the Creator software tool and the C programming language. Before you delve into the deep waters of the mighty PSoC, first get familiar with the C programming language.

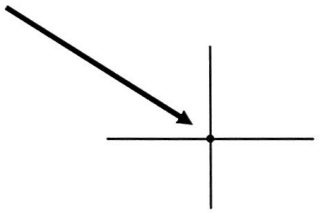
Please:

- Point your browser at: https://www.learn-c.org/
- Complete the 10 "Learn the Basics" exercises, including "Hello, World!", "Variables and Types," and so on through "Static". Make sure you understand how the website works. For each exercise, at the bottom of the webpage, you can type your own code, run it, and see the output, all in the browser. Do not simply look at the solutions! Write your own programs, and modify them to add your own customizations. For example, in the "Hello, World!" program, change the message to something like "6.115 Rules!" and print several messages.
- Also complete the following exercises in the "Advanced" section: Pointers, Arrays and Pointers.
- Comment your code and neatly paste it into your lab report document in an orderly and clear format for all of these exercises as part of your report for Exercise 4 of this lab.

READING SCHEMATICS:

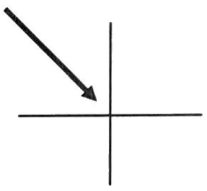
For graphical convenience, it is sometimes necessary for two wires in a schematic to cross. Sometimes, this means that there is an electrical connection. Other times it simply means that the two lines passed over one another on the drawing, but they are not actually intended to be connected to each other in the physical circuit. This situation is illustrated below:

The dot in the center of this crossing indicates that the wires are tied together electrically. That is, all four wires connect to the exact same electrical node.



These two wires are connected at the midpoint.

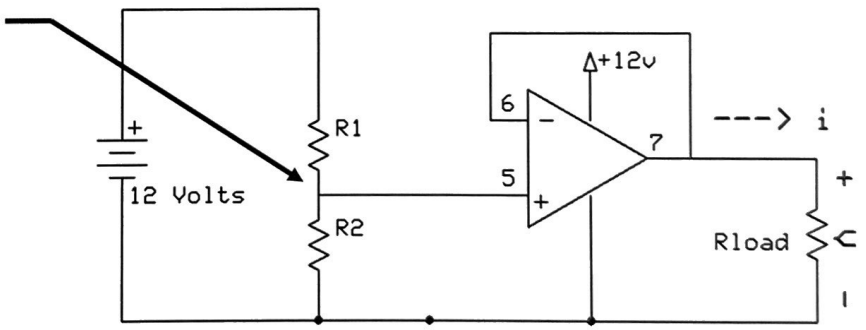
There is no dot at the center of this crossing. There are two distinct wires in this picture, one vertical, one horizontal. They are not the same, and are not joined at the midpoint electrically. They simply pass over one another on the drawing.



These two wires are not. They pass over one another, but there is no electrical connection.

Sometimes, the ExpressPCB schematic editor that we will use in this class will omit the dot when it is "obvious" from context. For example, in this circuit:

You should assume that this node has three wires or connections to it, one for R1, one for R2, and one for pin 5 of the op-amp. It wouldn't make sense for the op-amp pin 5 to simply "dead end" at the resistor mid-point without making a connection.

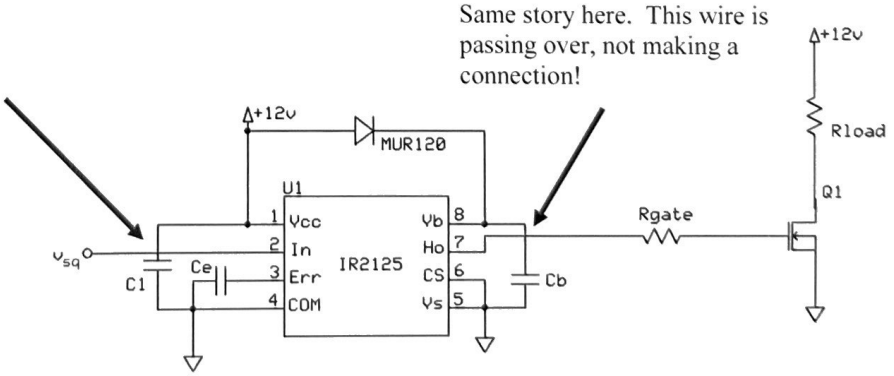


HOWEVER:

In this circuit, Pin 2 is NOT connected to Pin 1 or to C1. The wire input that provides the Vsq signal to the input pin 2 is simply passing over the C1/Pin 1 connection for graphical convenience here:

If this was an actual electrical connection, there would be a dot joining the pin 2 wire to the C1 capacitor to eliminate any ambiguity.

Same story here. This wire is passing over, not making a connection!



Well Done!