DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
CPEN 211 Introduction to Microcomputers, Fall 2019

Lab 10: Interrupts and Preemptive Multitasking

The handin deadline is 7:59 PM the evening before your lab section the week of Nov 18 to 22

REMINDER: *As outlined in the CPEN 211 Lab Academic Integrity Policy, until all students involved have a grade for Lab 10 you must* **NOT** *share or describe any code you write for this assignment with anyone except your* **authorized** *lab partner for Lab 10,* **NOT** *ask for or use any code offered to you by anyone (other than your authorized lab partner) and* **NOT** *look at or use solution code from anywhere. If you are repeating CPEN 211 you may not reuse any of your code submitted in a prior year.* **Using a compiler to generate ARM from C code for this lab, or Lab 10 or 11 is considered cheating. Use of a compiler will be considered cheating regardless of whether the compiler generated code (or even a portion of it) is used directly, adapted with changes, or a student "just looks at" the compiler generated code to compare with their own.** *Promptly report cases of misconduct you have first-hand knowledge of to the instructor. Your partner is "authorized" to work with you for Lab 10 if https://cpen211.ece.ubc.ca/cwl/lab_partners.php says they are your "current lab partner" at the time you start working together on Lab 10 up until you demo your code. The deadline to sign up or change lab partners using the above URL is 96 hours before your lab section. Your code* **will** *be checked for plagiarism using very effective plagiarism detection tools. As per UBC policy, all suspected cases of academic misconduct must be reported to the APSC Dean's office. Examples of outcomes for misconduct cases at UBC can be found online. See e.g.: https://universitycounsel.ubc.ca/files/2016/07/SD-2014-2015.pdf.*

1 Introduction

All computers need to interface to the outside world. They do this through one or more input/output (I/O) devices. For a computer to operate efficiently, an important question is, "when should the computer access these I/O devices?" In Lab 7 and in the tutorial for Lab 9 we used the slide switches for input and the red LEDs for output. We used a simple loop to repeatedly read the switches then update which red LEDs were on. As the Cortex-A9 on your DE1-SoC operates at 800 MHz each iteration of the loop in Figure 1 of Lab 9 takes only a few nanoseconds. If you changed the switches once a second this loop does nothing useful for 100s of millions of iterations, which is *very* inefficient. The solution is to let the computer do useful work and only talk to the I/O device when it is ready. This is accomplished by using an interrupt service routine that runs when an external input to the processor is asserted.

How do we keep the processor busy if our program must wait for input before proceeding? You may have noticed that multiple programs appear to run at once on a laptop or desktop computer. How do they do this? In Labs 6 through 9 you ran programs that accomplish a single task. In contrast, on a laptop or desktop computer you can easily open up an email client, a web browser, a Word document, Power Point, Quartus and perhaps several other programs. How do all these programs run "simultaneously"? The answer is by using a technique known as "multitasking". Your laptop runs one program for a very short period of time—typically only a few milliseconds—and then the operating system switches to running another program for a short period of time and so on. This switching can be done much faster than you can see so that it *appears* more than one program is running at once (even if the laptop only has one processor "core"). To deal with misbehaving programs it is important this switching is performed "preemptively". This "preemptive" capability is what enables you to shut down a program that has become unresponsive without needing to restart the computer.

In this lab you will learn how to support efficient interaction with I/O devices by writing interrupt service routines for the ARM processor in your DE1-SoC. You will also see how operating systems like Windows, Mac OS X, Android, iOS and Linux run multiple programs by implementing a similar capability on your DE1-SoC using a timer to trigger periodic interrupts that switch between programs.

2 Lab Procedure

This lab is divided into four parts.

2.1 Part 1 [2 marks]: Interrupt Example Program

In this part you use Altera Monitor Program and single step through one of the Altera provided assembly programs that demonstrates use of interrupts using the push button KEYS and HEX display. This part illustrates ARM's Generic Interrupt Controller, ARM processor modes, and masking of interrupts. Follow the steps below:

1. Start the Altera Monitor Program and create a new project ("File" > "New Project...")
2. Set the project directory to "`Z:\cpen211\lab10\`", name the project "`lab10part1`", and then select "ARM Cortex-A9" for architecture, then click "Next".
3. Select "DE1-SoC Computer" and click "Next".
4. Select "Assembly" for program type and check the box that says "Include a sample program with the project" the select "ARM A9 Generic Interrupt Controller" then select "Next". If the box for "Include a sample program with the project" cannot be selected, then you might have selected the wrong system in Step 3. If you have verified this is not the case and still encounter this issue, then manually copy the ".s" files from `C:\altera\15.0\University_Program\Computer_Systems\app_software_asm_arm\GIC_example\DE1-SoC` to the project directory above along with `address_map_arm.s` from Lab 9.
5. The next window that appears should be called "Specify program details". If you had to manually copy the files in the previous step, then add the following files in this order: `interrupt_example.s`, `interrupt_IDs.s`, `config_GIC.s` and `key_isr.s`. Then, click "Next".
6. In the next window, called "Specify system parameters", you should be able to leave the default settings if you were able to specify the sample program in Step 4. Either way, verify that Terminal device says "`JTAG_UART_for_ARM_0`". This is required for Part 3 of the lab. If the only option listed is "semihosting" you may have selected the wrong system in Step 3. Once you have done this click "Next".
7. In the next window, called "Specify program memory settings", make sure "Exceptions" is selected under "Linker Section Presets". Notice the "`.vectors`" section name is associated with addresses between 0 and 0x0000003F. These are the addresses we talked about in class when discussing the slide on the "Vector Table" in Slide Set 10. Click on "Finish".
8. Make sure your DE1-SoC is connected. Click "Yes" to download the .sof for the DE1-SoC Computer to your DE1-SoC.
9. Compile the program (e.g., as in Lab 9)
10. Load the program (e.g., as in Lab 9)
11. Run the program and try pushing KEY0 through KEY3 and observe the changes on HEX0.
12. Stop the program execution (Actions > Stop)
13. In the disassembly window scroll up to address 0 and observe the instruction at address 0x18. Recall from Slide Set 10 this is the address the ARM processor sets the PC to when the IRQ input is asserted.
14. Set a breakpoint on the instruction at address 0x18.

15. Select Actions > Run. Press one of the pushbutton keys (KEY0 through KEY3).
16. Single step through the ISR observing the changes in the registers. Your TA will ask you to explain some of the changes that occur during this step.
17. Examine the code in the files "`interrupt_example.s`", "`config_GIC.s`", "`key_isr.s`", "`address_map_arm.s`" and `interrupt_ID.s`". Your TA will ask you to about some of this code to see if you understand it.

Your mark for Part 1 will be:

1/2 If you can do steps 1 through 17, but cannot answer some of your TA's questions about step 16 or 17.

2/2 If you can do steps 1 through 17 and you can answer all of your TA's questions about step 16 or 17.

2.2 Part 2 [2 marks]: Adding a Periodic Timer ISR

In this part you extend the example code to use a timer. The timer ISR increments a counter and displays its value on the read LEDs. Follow the steps below:

1. Copy `interrupt_example.s` to `part2.s` and copy `config_GIC.s` to `config_GIC2.s`. Add a comment to the top of both `part2.s` and `config_GIC2.s` indicating they are copied and indicate the original file name. Create a new project called "lab2part2" making sure to change the "Linker Section Presets" in "Specify program memory settings" from "Basic" to "Exceptions". Include `config_GIC2.s`, `key_isr.s` and `part2.s` but do *not* include `interrupt_example.s` or `config_GIC.s`.
2. Modify `config_GIC2.s` to enable interrupts from the JTAG UART (Interrupt ID 80) and the built in timer (Interrupt ID 29). The easiest way to do this is to add two additional calls the function `CONFIG_INTERRUPT` inside the function `CONFIG_GIC` inside `config_GIC2.s`. The first argument to `CONFIG_INTERRUPT` is the interrupt ID to enable and the second argument is the target CPU, which should be 1.

Next, modify "`part2.s`" as described in Steps 1 through 3 below:

1. Before the label "IDLE" add code to configure the timer (`MPCORE_PRIV_TIMER` in `address_map_arm.s`). The timer interface is described in Section 2.4.1 on Page 3 of the document "`DE1-SoC_Computer.pdf`". Configure the timer to generate interrupts roughly every 0.5 seconds. You will need to think about the timer clock frequency, which is 200MHz and from this and an understanding of the timer gleaned from reading Section 2.4.1 in `DE1-SoC_Computer.pdf`, work out appropriate values for "Load value" and "Prescalar". Then, set the enable (E), auto (A) and interrupt (I) bits in the Control register each to 1. Make sure you understand what these bits control, which you can do by reading the timer documentation in Section 2.4.1 of `DE1-SoC_Computer.pdf`.
2. Extend `SERVICE_IRQ` to process interrupts generated by the timer. To do this you must leverage what you learned about how `SERVICE_IRQ` works when you stepped through it in Part 1. You first need to extend the existing code which determines which I/O device caused the interrupt so that it recognizes what to do when the timer is the cause of the interrupt. Then, when the timer interrupt occurs, have `SERVICE_IRQ` increment a "global variable" and display the value on the red LEDs. You can declare a global variable in the same way you initialize an array, which you did in Lab 9 (see the hints in the Lab 9 handout). In this case the array has a single element. Use load and store instructions to read and modify the global variable. You can display the value on the red LEDs using the same approach used in Figure 1 in Lab 9 (using a STR instruction).

3. Test the code for Steps 1 through 2 above before continuing.

Your mark for Part 2 will be:

1/2 If you complete part (a).

2/2 If you complete parts (a) through (e) and you can get the timer interrupt to work.

2.3 Part 3 [2 marks]: Adding a Keyboard ISR

Next add support for the ARM Cortex-A9 processor to respond to the keyboard. The keyboard ISR should be triggered when you press a key on the host computer (e.g., computer in MCLD 112) keyboard while the mouse cursor is in the "terminal" window of the Altera Monitor Program. Your keyboard ISR should store the character in a global variable. The code in the IDLE loop running on the ARM processor can then read the character out of that global variable and send it back to the Altera Monitor Program using the JTAG UART. When the Altera Monitor Program receives the character from the JTAG UART it displays it in the terminal window of the Altera Monitor Program. Follow the steps below:

1. Copy part2.s to part3.s and create a new project called "lab10part3" using `part3.s` and `config_GIC2.s` making sure to change the "Linker Section Presets" in "Specify program memory settings" from "Basic" to "Exceptions".

2. Before the label "IDLE" add code to configure the JTAG UART to generate interrupts when you type on your keyboard with the mouse focus in the Monitor program terminal window. The JTAG UART is described in Section 2.5.13 on Page 14 of "`DE1-SoC_Computer.pdf`" and in Section 3.5 on Page 21. Ensure the RE bit is set but the WE bit is clear.

3. Extend `SERVICE_IRQ` to process interrupts generated by the JTAG UART. When an interrupt occurs write the 8-bit ASCII value for the character into a global variable "`CHAR_BUFFER`" and set a global variable "`CHAR_FLAG`" to 1.

4. Modify the IDLE loop to read the global variable "`CHAR_FLAG`". If the value of "`CHAR_FLAG`" is 1 then read the value of "`CHAR_BUFFER`" into register R0 then call `PUT_JTAG` and set "`CHAR_FLAG`" to 0.

```
PUT_JTAG:   LDR      R1, =0xFF201000   // JTAG UART base address
            LDR      R2, [R1, #4]      // read the JTAG UART control register
            LDR      R3, =0xFFFF
            ANDS     R2, R2, R3        // check for write space
            BEQ      END_PUT           // if no space, ignore the character
            STR      R0, [R1]          // send the character
END_PUT:    BX       LR
```

Your mark for Part 2 will be:

1/2 If you complete Steps 1 through 4 above and JTAG interrupt service routine works but characters are not displayed on the terminal when the JTAG UART is properly configured and connected or the timer interrupt stops working.

2/2 If you complete Steps 1 through 4 above and you can get both the timer and JTAG interrupts to work and characters are displayed when the JTAG UART is properly configured and connected.

	Address	Value in memory
PD_ARRAY:	0x00001000	Saved R0 for Process 0
	0x00001004	Saved R1 for Process 0
		… (omitted)…
	0x00001030	Saved R12 for Process 0
	0x00001034	Saved R13 (SP) for Process 0
	0x00001038	Saved R14 (LR) for Process 0
	0x0000103C	Saved R15 (PC) for Process 0
	0x00001040	Saved CPSC for Process 0
	0x00001044	Saved R0 for Process 1
	0x00001048	Saved R1 for Process 1
		… (omitted)…
	0x00001070	Saved R12 for Process 1
	0x00001074	Saved R13 (SP) for Process 1
	0x00001078	Saved R14 (LR) for Process 1
	0x0000107C	Saved R15 (PC) for Process 1
	0x00001080	Saved CPSC for Process 1

Figure 1: Illustration of PD array for Part 4.

2.4 Part 4 [4 marks]: Preemptive Multitasking

In this part of the lab, you will see how a single processor can run multiple programs using multitasking. You will extend the code from Part 3 to enable preemptive multitasking between two different programs. First, you need to learn a *little* bit about how operating systems work.

Each program you see on a laptop, desktop or smartphone is associated with what is known as a "process". Each process has at least one "thread" corresponding to a program like those you saw in APSC 160 last year and the simple assembly programs you have written in Labs 6 through 9. The key "trick" is providing software the appearance that each thread has its own set of registers including the stack pointer and program counter even though the hardware only has a single set of registers. At any given time only a single thread is running on a CPU such as the Cortex-A9 and only the register values for that thread are actually in R0-R15 and CPSR[1]. The other process' registers are saved in a region in memory known as a "process descriptor" (PD) table. The PD Table is just an array of 32-bit integers with one element for each of R0 through R15 and CPSR for each process. For this lab we will assume only two processes are running, Process 0 and Process 1, so this array would look like shown in Figure 1 assuming the assembler places the array `PD_ARRAY` starting at address 0x00001000.

To switch from running one process (e.g., Process 0) to another during a timer interrupt the interrupt service routine (ISR) first copies the values that were in R0-R15 and CPSR when the interrupt occurred to the area of the PD Table reserved for the currently running process. For example, if Process 0 was running when the time interrupt occurred, the ISR would save R0-R15 and CPSR to the memory locations with addresses 0x00001000 through 0x0000103C in Figure 1). You can code this inside your timer interrupt service routine using regular STR instructions for R0 to R12. However, things are a bit more tricker for R13 to R15 and CPSR: After the timer interrupt occurs the ARM processor should be in IRQ mode. Recall the value of R15 (PC) for the interrupted program will be in R14 (LR). You can use a regular STR instruction to save this value into the PD Table in the location for the program counter. For example, if the timer interrupts

[1]If we wish to run programs using floating point registers the PD table, described below, would also need space for S0-S31.

```
int count = 0;
int i;
while (1) {
   count = count + 1;
   *ledr = count; // this line means set the red LEDs as in the Figure 1 in Lab 9
   // the following loop adds a delay
   i = 0;
   do {
      i = i + 1;
   } while( i < LARGE_NUMBER );
}
```

Figure 2: For Part 4 write ARM assembly program corresponding to the above code for Process 1 (PROC1).

Process 0, the ISR would save R14 to address 0x0000103C in Figure 1. Similarly, the value of CPSR for the interrupted program is actually in SPSR when the ARM processor starts executing the timer interrupt service routine. To copy this value to the PD Table you first need to copy it into a regular register (e.g., R0) using the MRS instruction described below. Then you would save it to the CPSR enter in the PD table. For example, if the timer interrupts Process 0, the ISR would save SPSR to address 0x00001040 in Figure 1. To copy the values of R13 (SP) and R14 (LR) for the interrupted process your timer interrupt service routine first needs to switch the mode of the processor to SVC (Supervisor mode) which is the mode Process 0 and Process 1 use[2]. After switching modes, save R13 and R14 to their corresponding locations in the PD Table using regular STR instructions (e.g., if the timer interrupts Process 0, and assuming your `PD_TABLE` starts in memory at address 0x00001000 as in Figure 1 then save to locations 0x00001034 and 0x00001038).

Next, to switch tasks, the timer interrupt service routine restores the saved registers for another process from the PD Table into R0-R15 and CPSR. For example, if we are switching from Process 0 to Process 1, then after saving the registers for Process 0 as described above, the interrupt service routine would read values from memory locations 0x00001044 through 0x00001080 into registers R0 through R15 and the CPSR. For R0 to R12 you can use regular LDR instructions to do this. Restoring R13-R15 and CPSR from the PD Table is tricker. While abstractly this simply requires reversing the process outlined above for saving registers, careful thought is required to do this so that you end up with all registers restored to the correct values for the process you want to switch to.

If done correctly, your timer interrupt service routine will cause the ARM processor to "return" to a different program than was running when the interrupt occurred. For example, if initially Process 0 was running, then after the first timer interrupt occurs, the ARM processor will be running Process 1. On the second timer interrupt your ISR code should cause the ARM processor to again run Process 0. Importantly, your Process 0 should continue running exactly where it was when the timer interrupted it (it should **not** start again from the beginning). Doing these switches between Process 0 and Process 1 repeatedly, with a fast enough timer interval, gives the user the impression both programs are running simultaneously even though at any given time only a single program is actually running.

More details are outlined in the steps below:

1. Copy "`part3.s`" to "`part4.s`" and create a new project called "lab10part4" using `part4.s` and `config_GIC2.s`. Make sure Terminal device is set to "`JTAG_UART_for_ARM_0`" and change the

[2] Note, the steps outlined in Part 4 assume Process 0 and Process 1 run in "Supervisor mode" as in `interrupt_example.s`. We use Supervisor mode instead of "User mode" (like a regular operating system would) only because of a bug in the version of the Altera Monitor Program we are using. This bug occurs while single stepping in User mode code. If you want an extra challenge: Try using User mode for Process 0 and Process 1, but without the benefit of being able to single step through your code to debug your timer interrupt. To do this, you must add code before the IDLE loop to switch to User mode with IRQ enabled, then set the User stack pointer for Process 0, then, to save/restore the stack pointer and link register for User mode, temporarily switch into System mode with interrupts disabled instead of temporarily switching to Supervisor mode.

"Linker Section Presets" in "Specify program memory settings" from "Basic" to "Exceptions". Modify "`part4.s`" as described in Steps (b) through (h) below:

2. After the IDLE loop, add a label "PROC1". This label will indicate the start of a second "process" (Process 1) that we wish the ARM Cortex-A9 to multitask with the code in your IDLE loop, which will be Process 0.

3. Write code for PROC1 that displays the value of a counter on the LEDs. Specifically, write assembly code equivalent to the C code shown in Figure 2. Adjust the value of `LARGE_NUMBER` to ensure you can see the LEDs changing.

4. Remove the code that updates the red LEDs during a timer interrupt from inside `SERVICE_IRQ`.

5. Add a global variable "`CURRENT_PID`" using a label and `.word` directive as below:

```
CURRENT_PID: .word  0
```

   Modify your timer interrupt to use this variable to track the identifier (PID) of the currently executing process. As initially Process 0 (IDLE) runs, the initial value should be 0.

6. Add a global array "`PD_ARRAY`" that contains 34, four byte words declared as follows:

```
PD_ARRAY:   .fill   17,4,0xDEADBEEF
            .fill   13,4,0xDEADBEE1
            .word   0x3F000000  // SP
            .word   0           // LR
            .word   PROC1+4     // PC
            .word   0x53        // CPSR (0x53 means IRQ enabled, mode = SVC)
```

   You will use the first 17 locations to save the values of R0 through R15 along with the CPSR for Process 0 when Process 0 is NOT running. These locations are initialized to the value "0xDEADBEEF" to make it easier to identify them in the "Memory" tab of the Altera Monitor Program while debugging your task switching code that you will write below. Process 0 (PID=0) corresponds to the idle loop, which is initially running. The next 17 locations are for saving register R0 through R15 along with the CPSR for Process 1 (with PID=1) when Process 1 is not running. Process 1 (PID=1) corresponds to the code you placed after the label PROC1. Recall, that initially the code after the label PROC1 is NOT running (instead PID=0 is running the "IDLE" code that calls `PUT_JTAG`). So, we need the values in `PD_ARRAY` for Process 1 to hold initial values for the registers R0 through R15 and the CPSR that make sense when Process 1 does start running. The registers that matter most are the program counter (R15), the stack pointer (R13) and the status register (CPSR). We set the initial program counter to the instruction at the label PROC1 using the following line:

```
            .word   PROC1+4     // PC
```

   The reason for adding four (+4) is we will return from the timer interrupt using "`SUBS PC, LR, #4`". The stack pointer value for process 1 is initialized to 0x3F000000 using the following line:

```
            .word   0x3F000000  // SP
```

   The initial value of CPSR is set to have IRQ unmasked and be in mode SVC using the line[3]:

```
            .word   0x53      // CPSR (0x53 means IRQ enabled, mode = SVC)
```

[3] Bit 8 of the CPSR is the asynchronous data abort mask bit. You may notice that this bit is set when entering an ISR. You do not need to worry about it. The details of the interrupt it masks are best left to CPEN 411 where you will learn about "store buffers".

The initial link register value (0) for Process 1 should never be used because your PROC1 code should contain an infinite loop and thus never return to a caller. The other registers for Process 1 are initialized to garbage values 0xDEADBEE1 because your code for PROC1 should initialize any registers it needs to read. These "garbage" values are used because they can be helpful while debugging your task switching code.

7. Modify `SERVICE_IRQ` so that when a timer interrupt occurs the registers R0 through R15 and the CPSR of the interrupted program are saved in the appropriate half of `PD_ARRAY`. In addition to the discussion above, consider that the first thing the interrupt service routine code from the sample in Part 1 does is save some registers on the stack before overwriting them. These values saved to the ISR's stack contain values that were being used by the interrupted program (e.g., Process 0 or 1). They need to be moved from the stack and into the PD Table. As you need some registers for the ISR to run (e.g., to figure out where the PD ARRAY is), you will still need to temporarily save at least some register to the IRQ stack. As a hint: Think of breaking the process of saving the registers into two parts: First save the registers the ISR did not copy to its own stack to the PD Table, then write code to copy the values saved to the stack to the PD Table.

   Recall that upon entering an interrupt service routine the value of the interrupted program's CPSR is automatically stored in the SPSR. To store it into `PD_ARRAY` while in mode IRQ you must first copy it from the IRQ's SPSR into a normal register (R0-R12) using the MRS instruction. For example:

   ```
   MRS R0, SPSR  // copies CPSR of interrupted program into R0
   ```

   To save (or restore) the value of the stack pointer (R13) and link register (R14) to (or from) the `PD_ARRAY` requires temporarily switching to supervisor mode with interrupts disabled. Look at the code from Part 2-3 that switches processor modes if you forget how to do this.

   Then, `SERVICE_IRQ` should update `CURRENT_PID` to be 1 if it was 0 or 0 if it was 1.

   Then, `SERVICE_IRQ` should load the values of registers R0 through R15 and CPSR from the other part of `PD_ARRAY` corresponding to the new value of `CURRENT_PID`. The last two registers that should be restored are the program counter and CPSR. This can be accomplished in a single instruction using a subtract with status register update (SUBS) with the PC as the destination register[4]:

   ```
   SUBS PC, LR, #4
   ```

   after first loading the value that will be put into CPSR into SPSR and putting the value that will be put into PC in LR while in IRQ mode. We will discuss why it is important to update both the PC and CPSR with a single instruction during lecture on Nov 16. To first move a value into SPSR from the `PD_ARRAY` you will need two instructions: A load that reads in the saved CPSR value from your `PD_ARRAY` to a register followed by an MSR instruction[5] such as:

   ```
   MSR SPSR, R0   // this copies contents of R0 into SPSR
   ```

If implemented correctly, the code for Part 4 will operate as follows: When Process 0 is running any keystrokes on the keyboard will be echoed to the terminal. When Process 1 is running the red LEDs should increment a few times. Each process should run for one half of a second and then the other process should run. For grading purposes this behavior must to be implemented using interrupts and switching of tasks by saving and restoring of registers to the `PD_ARRAY` as described above.

Your mark for Part 4 will be:

[4] ARMv6 also supports simultaneous copying of PC and CPSR using RFE (see Appendix B1). SUBS is simpler to understand.
[5] The MRS and MSR instruction are described in Appendix B1 of COD4e (PDF on Connect).

1/4 If you finish Steps 1 through 6 but didn't finish Step 7.

2/4 If you finish Steps 1 through 7 but your code doesn't work at all.

3/4 If you finish Steps 1 through 7 and your code switches from Process 0 to Process 1 but does not switch back again.

4/4 If your code works correctly. This means it works as described above and uses the task switching approach described in Step 7.

3 Lab Submission

Submit all files by the deadline on Page 1 using handin. Use the same procedure outlined at the end of the Lab 3 handout except that now you are submitting Lab 10, so use:

```
handin cpen211 Lab10-<section>
```

where `<section>` should be replaced by your lab section. Remember you can overwrite previous or trial submissions using `-o`. **You will lose marks if your Lab 10 submission is missing any files needed to run your code (including project files).**

4 Lab Demonstration Procedure

To reduce congestion in the lab we will be dividing each lab section into two one hour sessions. For example, for L1A the first session will run from 9 am to 10 am and the second session will run from 10 am to 11 am. We request that you show up no more than 10 minutes before the start of your assigned one hour "Lab 10 Marking Time", which will be posted on Connect at least 24 hours before your lab section along with your "Lab 10 TA". The TAs will have a randomly ordered list of lab partners and will start working their way down the list marking (look for "Lab 10 Marking Order" on Connect to get a rough idea, but be sure you are in the lab at the beginning in case the TA follows a different order). Your TA should have your submitted code with them and have setup a "TA marking station" where you will go when it is your turn to be marked. Please bring your DE1-SoC.