WES 237A: Introduction to Embedded System Design (Winter 2026)
Lab 3: Serial and CPU
Due: 2/1/2026 11:59pm

In order to report and reflect on your WES 237A labs, please complete this Post-Lab report by the end of the weekend by submitting the following 2 parts:

- Upload your lab 3 report composed by a single PDF that includes your in-lab answers to the bolded questions in the Google Doc Lab and your Jupyter Notebook code. You could either scan your written copy, or simply type your answer in this Google Doc. **However, please make sure your responses are readable.**
- Answer two short essay-like questions on your Lab experience.

All responses should be submitted to Canvas. Please also be sure to push your code to your git repo as well.

Serial Connection

- Using a micro USB cable, connect your board to your laptop
- Connect to board using the serial connection
    - Linux
        - Open a new terminal
        - Run the command
            - *sudo screen /dev/<port> 115200* #port: ttyUSB0 or ttyUSB1
    - MAC
        - Open a new terminal
        - Run the command and check the PYNQ resources for the port
            - *sudo screen /dev/<port> 115200* #port: check resources
            - sudo screen /dev/tty.usbserial-1234_tul1 115200
    - Windows
        - Check the resource for how to connect through serial to the PYNQ board
    - Resources:
        - https://pynq.readthedocs.io/en/v2.0/getting_started.html
        - https://www.nengo.ai/nengo-pynq/connect.html
- After connecting
    - Restart the board (*$ sudo reboot*)
    - Interrupt the boot (keyboard interrupt)
    - List current settings (*printenv*)
    - **Put a screenshot of your *$ printenv* output**

Change Bootargs

- *If you need to return to the default bootargs, you can find them below*
  - *https://github.com/Xilinx/PYNQ/blob/master/sdbuild/boot/meta-pynq/recipes-bsp/device-tree/files/pynq_bootargs.dtsi*
  - *bootargs = 'root=/dev/mmcblk0p2 rw earlyprintk rootfstype=ext4 rootwait devtmpfs.mount=1 uio_pdrv_genirq.of_id="generic-uio" clk_ignore_unused'*
- To edit bootargs:
  - Interrupt the boot
  - Edit boot arguments:
    - *$ editenv bootargs*
    - Insert arguments included the quotations all in one line:
      - Bootargs (default and more) are at here
    - *$ boot*
- Change bootargs to the following
  - *bootargs = 'console=ttyPS0,115200 root=/dev/mmcblk0p2 rw earlyprintk rootfstype=ext4 rootwait devtmpfs.mount=1 uio_pdrv_genirq.of_id="generic-uio" clk_ignore_unused* ***isolcpus=1*** *&& bootz 0x03000000 - 0x02A00000'*
  - **What does isolcpus=1 do?**

Isolate CPU core 1; the scheduler will only use it for real-time or dedicated tasks and not auto-assign any other tasks.

  - **What would isolcpus=0 do?**

This isolates CPU core 0, which is the kernel scheduler, so this will be unable to function properly.

Heavy CPU Utilization

- Download *fib.py* from here. This is a recursive implementation for generating Fibonacci sequences. We just do not print the results.
- Jupyter notebook is hosted at: /home/xilinx/jupyter_notebooks

- Make sure your board is booted with custom bootargs above, including *isolcpus=1*

1) Open two terminals (Jupyter):
   - Terminal 1: run *htop* to monitor CPU utilization
   - Terminal 2: run *$ python3 fib.py* and monitor CPU utilization and time spent for running the script (set terms to lower than 40)
   - **Describe the results of *htop*.**

I had about 96% CPU usage and took about 4.4757 seconds for 30 terms. Most likely ran on 0, because CPU 1 was isolated by bootargs.

2) Repeat the previous part, but this time use *taskset* to use CPU1:
   - Terminal 2: run *$ taskset -c 1 python3 fib.py* and monitor CPU utilization and time spent for running the script
   - **Describe the results of *htop*. Specifically, what's different from running it in 1)?**

I had 95% usage for 30 terms; it took about 4.729 seconds. This may have slightly different overhead or cache behavior, but it proves the core is functional and dedicated to this task.

3) Heavy Utilization on CPU0:
   - Open another terminal and run $ *dd if=/dev/zero of=/dev/null*
   - Repeat parts 1 and 2
   - **Describe the results of *htop*.**

For Python3 (30 terms), it was 100%. For the Taskset (30 terms), it was also 99.7%. The ddcommand consumes all available cycles on the core it's assigned to and defaults to CPU 0. This confirms that even when CPU 0 is pinned at 100%, isolated CPU 1 can still execute my script independently.

Jupyter Notebook CPU Monitoring

Download CPU_monitor.ipynb from here. This is an interactive implementation for plotting in a loop. Running this notebook is a computationally heavy task for your CPU, therefore you do not need to run any additional process to utilize your CPU0.

- Create a Jupyter notebook
  - Use the *os* library to create a Python program that accepts a number from user input (0 or 1) and runs *fib.py* on a specific core (0 or 1).
  - *Hint: look at the os.system() call and remember the 'taskset' function we've used previously.*
- You should have two notebooks running: 1) CPU_monitor, 2) CPU_select
  - **Compare your observations between using Jupyter notebook CPU_monitor and linux command *htop* for monitoring CPU utilization.**

htop seems faster and more of a lightweight tool that shows real-time data without slowing down the board. The CPU_monitor notebook is much slower and actually creates its own heavy CPU load just to draw the graphs. This means htop is better for accuracy, while the notebook is only useful for seeing a visual history of the spikes.

ARM Performance Monitoring (C++)

- Download kernel_modules folder
- Read through CPUcntr.c and reference the ARM documentation for the PMU registers here to answer the following question.

    - **According to the ARM docs, what does the following line do? Are they written in assembly code, python, C, or C++?**
        - *asm("MCR p15, 0, 1, c9, c14, 0\n\t");*

This line is in assembler; it tells the ARM processor to enable the performance monitor interrupt for the cycle counter. Therefore, it enables the hardware that tracks clock cycles for timing measurements.

    - Compile and insert the kernel module following the instructions from the README file.
- Download clock_example folder
- Read through *include/cycletime.h* and take note of the functions to initialize the counters and get the cyclecount (what datatype do they return, what parameters do they take)
    - **What does the following line do?**
        - *asm volatile ("MRC p15, 0, %0, c9, c13, 0\n\t" : "=r"(value));*

This line is also in assembler; it reads the Cycler Count Register. It will take the current number of CPU clocks and store it in a C++ variable.

- Complete the code in *src/main.cpp.* These instructions are for those who have never coded in C++
    - **Declare 2 variables (cpu_before, cpu_after) of the correct datatype**
    - **Initialize the counter**
    - **Get the cyclecount 'before' sleeping**
    - **Get the cyclecount 'after' sleeping**
    - **Print the difference number of counts between starting and stopping the counter**
- After completing the code, open a jupyter terminal and change directory to *clock_examples/*
- Run *$ make* to compile the code
- Run the code with *$ ./lab3 <delay-time-seconds>*
- **Change the delay time and note down the different cpu cycles as well as the different timers.**

When running the lab3 program, the system uses the ARM Performance Monitoring Unit to capture the exact number of clock cycles elapsed during the sleep period. Since the PYNQ operates at 650 MHz, I observed that for every one second of delay, the cycle count increased by approximately 650 million counts, so the total will just be a multiple of whatever the input arg is …. The hardware cycle counter provided a much more precise and consistent measurement of time compared to the software-based LinuxTimer, which showed slight variations due to the OS scheduling overhead(it won't be perfect). I also noted that the program requires the kernel module to be inserted beforehand; otherwise, the CPU triggers an "Illegal Instruction" error because the performance registers are protected. I had to restart a few times to do this properly.. Ultimately, these results confirm that hardware-level monitoring is the most reliable way to measure execution time in an embedded environment.

In [ ]:
```
import os

core = input("What core? 0 or 1 \n")

if core == "0":
    os.system("taskset -c 0 python3 fib.py")
else:
    os.system("taskset -c 1 python3 fib.py")
```

In [ ]:
```

```

```
//
//  main.cpp
//  Lab4
//
//  Created by Alireza o n 2/14/20.
//  Copyright Â© 2020 Alireza. All rights reserved.
//

#include "main.h"
#include "cycletime.h"
#include "timer.h"
#include <unistd.h>

using namespace std;

int main(int argc, const char * argv[])
{
      float cpu_timer;
      unsigned int delay = 1;

      cout << "WES237A lab 4" << endl;

      char key=0;

      // 1 argument on command line: delay = arg
      if(argc >= 2)
      {
            delay = atoi(argv[1]);
      }

    //TODO: declare 2 cpu_count variables: 1 for before sleeping, 1 for after
sleeping (see cpu_timer)
    unsigned int cpu_before = 0;
    unsigned int cpu_after = 0;
    unsigned int cpu_difference = 0;

    //TODO: initialize the counter
    init_counters(1,0);//reset counters (1), no dividers (0)

    //TODO: get the cyclecount before sleeping
    cpu_before = get_cyclecount();
    usleep(delay);

    //TODO: get the cyclecount after sleeping
    cpu_after = get_cyclecount();

    //TODO: subtract the before and after cyclecount
    cpu_difference = cpu_after - cpu_before;

    //TODO: print the cycle count (see the print statement for the cpu_timer
below)
    cout << "start:: "<<cpu_before<< "\n"<<endl;
    cout << "end:: " <<cpu_after<< "\n" <<endl;
    cout << "diff:: " <<cpu_difference<< "\n"<<endl;

      LinuxTimer t;
      usleep(delay);
      t.stop();
      cpu_timer = t.getElapsed();


      cout << "Timer: " << (double)cpu_timer/1000000000.0 << endl;

      return 0;
```

```
}
```