UNIVERSITY OF CALIFORNIA SAN DIEGO

Course # WES 237A

Course Title: Intro to Embed Sys Des

Assignment #3

Professor Nadir Weibelt
TA Chen Chen

Author	Student ID
Abdullah Ajlan	A69028719

https://github.com/ajlan-UCSD/Assignment-3

Assignment 3

In lab, we experimented with C++ code for initializing the PMU counters and retrieving the cyclecount. In this assignment, you'll be setting up your PMU counter to use in Python.

Part A3.0: New kernel_module code

- Download the new kernel_module.zip code from canvas.
- Make and insert this new module following the 'make' and 'insmod' instructions from lab and in the README.
- Check that it is inserted on both CPUs by checking the dmesg | `tail` output

Solve Part A3.0:

1. **Unzip the File**: Use the `unzip` command to extract the contents of `kernel_module.zip`:

```
unzip kernel_module.zip
```

2. **Compile the Module**: Navigate to the directory containing the Makefile and run the `make` command:

```
cd <directory name>
make -C /lib/modules/$(uname -r)/build M=$(pwd) modules
```

3. **Insert the Module**: Use the `insmod` command to insert the module into the kernel:

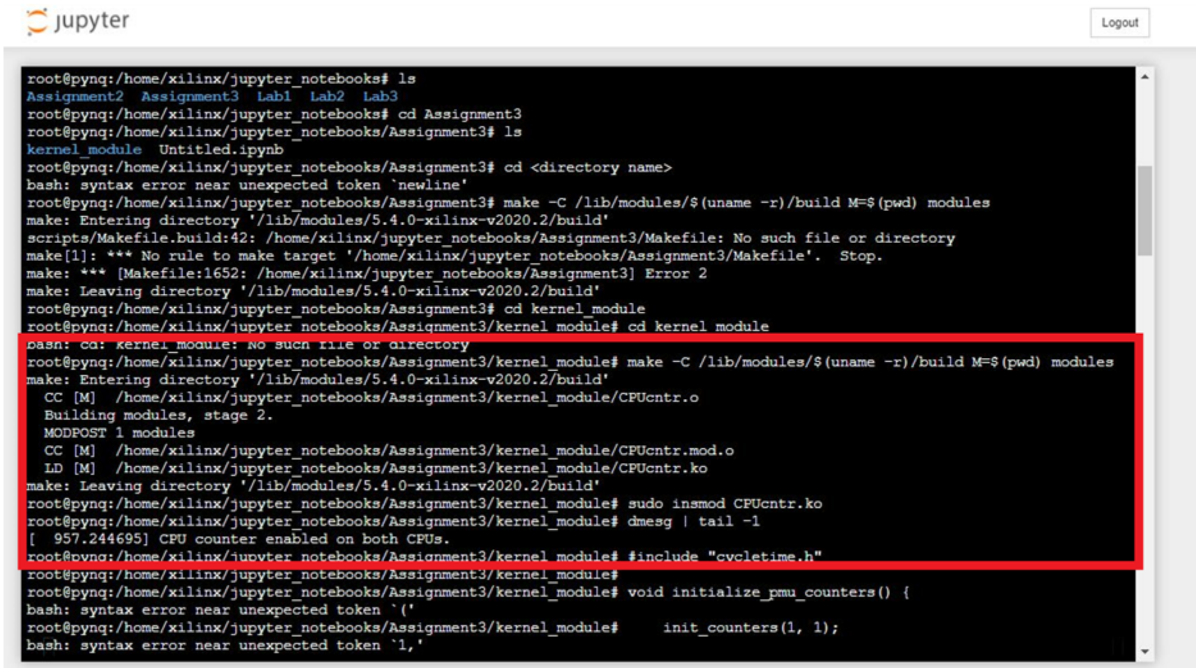
```
sudo insmod CPUcntr.ko
```

4. **Check the Module**: Use the `dmesg | tail -1` command to check that the module has been inserted:

```
dmesg | tail -1
```

terminal output:

-
-
-



Part A3.1: Access PMU from python

- Create a shared library object with two fuctions by wrapping the cycletime.h into a new shared opticed library (see Lab2)
  - One function to initialize the PMU counters
  - One function to get the cycle count
- Compile the shared library (see Lab2 if you've forgotten how to do this)
  Access the shared library functions using the ctypes
- module, **but don't wrap the function calls in a python function.**

Solve :

1. **Create a C file (let's call it `pmu_lib.c`)** that includes `cycletime.h` and implements the two functions. Here's a basic example:

```
#include "cycletime.h"

void initialize_pmu_counters() {
    init_counters(1, 1);
}

unsigned int get_cycle_count() {
    return get_cyclecount();
}
```

2. **Compile the shared library**. You can use `gcc` to compile the C file into a shared library. Here's the command you can use:

```
gcc -shared -o libpmu.so pmu_lib.c
```

This will create a shared library named `libpmu.so`.

3. **Access the shared library functions using the ctypes module in Python**.

```
import ctypes

# Load the shared library
libpmu = ctypes.CDLL('./libpmu.so')

# Now you can call the functions from the shared library
libpmu.initialize_pmu_counters()
print(libpmu.get_cycle_count())
```

terminal output:

```
root@pynq:/home/xilinx/jupyter_notebooks/Assignment3/kernel_module# gcc -shared -o libcycle.so cycletime.h
root@pynq:/home/xilinx/jupyter_notebooks/Assignment3/kernel_module# ls
CPUcntr.c   CPUcntr.mod    CPUcntr.mod.o  cycletime.h  Makefile       Module.symvers  Untitled.ipynb
CPUcntr.ko  CPUcntr.mod.c  CPUcntr.o      libcycle.so  modules.order  README
root@pynq:/home/xilinx/jupyter_notebooks/Assignment3/kernel_module# file libcycle.so
libcycle.so: GCC precompiled header (version 014) for C
```

Part A3.2: Comparing and Gathering Data

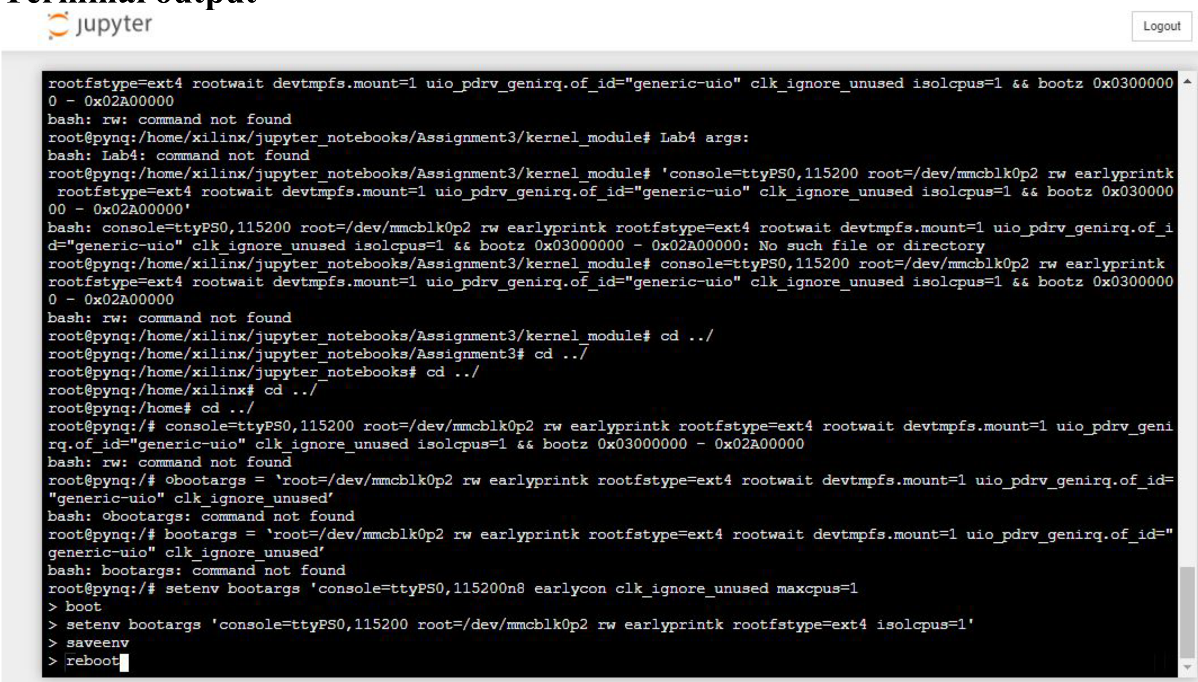
In this section, we are going to use psutil to monitor CPU usage in percent, and the time module and PMU counting to evaluate the recursive fibonacci sequence timing operations.

- Isolate CPU 1 by editing the bootargs (see lab work part 1)
- Insert the CPUcntr kernel object onto both cpus using the instructions from lab
- Write code to do the following using the recur_fibo function from lab
  - Initialize the cyclecounter
  - Get the 'before' time using the python time module
  - Get the 'before' cycle count
  - Run the recur_fibo function on a CPU 1
  - Get the 'after' cycle count
  - Get the 'after' time count using the python time module
  - Get the cycle count and the amount of time used
- Vary the number of terms from 1 to 30 **as you see fit** to compare the different execution times
  - Take multiple trials for each variation (i.e. get three cyclecounts for n=5, then get three cyclecounts for n=10, etc) and average the different tials.
  - The error for each 'n' will be the standard deviation from the mean which is the standard deviation of all the trials divided by the square root of the number of trials.
- Plot the average results for varying 'n' along with error bars of your measurments.
- In order to compare the timing module and PMU counting, we need them to be in the same units.
  - To get the CPU frequency, run cat /proc/cpuinfo in a new terminal or run lscpu
  - Use this frequency to convert the PMU output from clock counts to timing
  - Compare the timing of the PMU counter to the timing module

Solve Part A3.2: Comparing and Gathering Data

1. **Isolate CPU 1 by editing the bootargs**: You can do this by editing the bootargs in the U-Boot environment. This is usually done by setting the `isolcpus=1` bootarg.

Terminal output




```
rootfstype=ext4 rootwait devtmpfs.mount=1 uio_pdrv_genirq.of_id="generic-uio" clk_ignore_unused isolcpus=1 && bootz 0x0300000
0 - 0x02A00000
bash: rw: command not found
root@pynq:/home/xilinx/jupyter_notebooks/Assignment3/kernel_module# Lab4 args:
bash: Lab4: command not found
root@pynq:/home/xilinx/jupyter_notebooks/Assignment3/kernel_module# '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 0x030000
00 - 0x02A00000'
bash: console=ttyPS0,115200 root=/dev/mmcblk0p2 rw earlyprintk rootfstype=ext4 rootwait devtmpfs.mount=1 uio_pdrv_genirq.of_i
d="generic-uio" clk_ignore_unused isolcpus=1 && bootz 0x03000000 - 0x02A00000: No such file or directory
root@pynq:/home/xilinx/jupyter_notebooks/Assignment3/kernel_module# 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 0x0300000
0 - 0x02A00000
bash: rw: command not found
root@pynq:/home/xilinx/jupyter_notebooks/Assignment3/kernel_module# cd ../
root@pynq:/home/xilinx/jupyter_notebooks/Assignment3# cd ../
root@pynq:/home/xilinx/jupyter_notebooks# cd ../
root@pynq:/home/xilinx# cd ../
root@pynq:/home# cd ../
root@pynq:/# console=ttyPS0,115200 root=/dev/mmcblk0p2 rw earlyprintk rootfstype=ext4 rootwait devtmpfs.mount=1 uio_pdrv_geni
rq.of_id="generic-uio" clk_ignore_unused isolcpus=1 && bootz 0x03000000 - 0x02A00000
bash: rw: command not found
root@pynq:/# obootargs = 'root=/dev/mmcblk0p2 rw earlyprintk rootfstype=ext4 rootwait devtmpfs.mount=1 uio_pdrv_genirq.of_id=
"generic-uio" clk_ignore_unused'
bash: obootargs: command not found
root@pynq:/# bootargs = 'root=/dev/mmcblk0p2 rw earlyprintk rootfstype=ext4 rootwait devtmpfs.mount=1 uio_pdrv_genirq.of_id="
generic-uio" clk_ignore_unused'
bash: bootargs: command not found
root@pynq:/# setenv bootargs 'console=ttyPS0,115200n8 earlycon clk_ignore_unused maxcpus=1
> boot
> setenv bootargs 'console=ttyPS0,115200 root=/dev/mmcblk0p2 rw earlyprintk rootfstype=ext4 isolcpus=1'
> saveenv
> reboot
```

2. **Insert the CPUcntr kernel object onto both CPUs**: using the `insmod` command with the path to the `CPUcntr.ko` file.
3. **Write code to do the following using the recur_fibo function**:

```
import ctypes
import time
import psutil
import numpy as np
import matplotlib.pyplot as plt

# Load the shared library
libpmu = ctypes.CDLL('./libpmu.so')

def recur_fibo(n):
    if n <= 1:
        return n
    else:
        return(recur_fibo(n-1) + recur_fibo(n-2))

# Vary the number of terms from 1 to 30
for n in range(1, 31):
    times = []
    cycle_counts = []

    # Take multiple trials for each variation
    for _ in range(3):
        # Initialize the cyclecounter
        libpmu.initialize_pmu_counters()

        # Get the 'before' time and cycle count
```

```
        start_time = time.time()
        start_cycles = libpmu.get_cycle_count()

        # Run the recur_fibo function on a CPU 1
        psutil.Process().cpu_affinity([1])
        recur_fibo(n)

        # Get the 'after' cycle count and time
        end_cycles = libpmu.get_cycle_count()
        end_time = time.time()

        # Get the cycle count and the amount of time used
        cycle_counts.append(end_cycles - start_cycles)
        times.append(end_time - start_time)

    # Calculate the average and standard deviation
    avg_time = np.mean(times)
    avg_cycles = np.mean(cycle_counts)
    error_time = np.std(times) / np.sqrt(3)
    error_cycles = np.std(cycle_counts) / np.sqrt(3)

    # Plot the average results for varying 'n' along with error bars of
your measurements
    plt.errorbar(n, avg_time, yerr=error_time, fmt='o')
    plt.errorbar(n, avg_cycles, yerr=error_cycles, fmt='o')

plt.show()
```

4. **To get the CPU frequency**, you can run `cat /proc/cpuinfo` in a new terminal or run `lscpu`. You can use this frequency to convert the PMU output from clock counts to timing.

```
root@pynq:/home/xilinx/jupyter_notebooks/Assignment3/kernel_module# lscpu
Architecture:          armv7l
Byte Order:            Little Endian
CPU(s):                2
On-line CPU(s) list: 0,1
Thread(s) per core:    1
Core(s) per socket:    2
Socket(s):             1
Vendor ID:             ARM
Model:                 0
Model name:            Cortex-A9
Stepping:              r3p0
BogoMIPS:              650.00
Flags:                 half thumb fastmult vfp edsp neon vfpv3 tls vfpd32
root@pynq:/home/xilinx/jupyter_notebooks/Assignment3/kernel_module# lscpu
Architecture:          armv7l
Byte Order:            Little Endian
CPU(s):                2
On-line CPU(s) list: 0,1
Thread(s) per core:    1
Core(s) per socket:    2
Socket(s):             1
Vendor ID:             ARM
Model:                 0
Model name:            Cortex-A9
Stepping:              r3p0
BogoMIPS:              650.00
Flags:                 half thumb fastmult vfp edsp neon vfpv3 tls vfpd32
```

As we can see from figure the operation time and number of operation increase exponentially

In [20]:

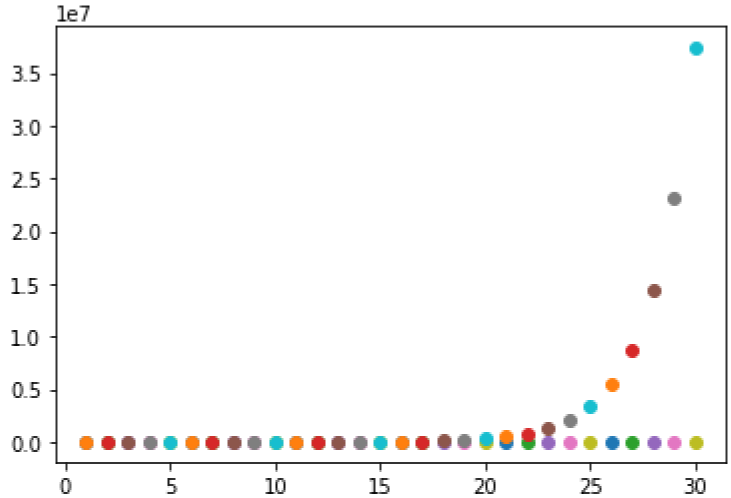
```
import ctypes

# Load the shared library
libpmu = ctypes.CDLL('./libpmu.so')

# Now you can call the functions from the shared library
libpmu.initialize_pmu_counters()
print(libpmu.get_cycle_count())
```

5395

In [1]:

```
import ctypes
import time
import psutil
import numpy as np
import matplotlib.pyplot as plt

# Load the shared library
libpmu = ctypes.CDLL('./libpmu.so')

def recur_fibo(n):
    if n <= 1:
        return n
    else:
        return(recur_fibo(n-1) + recur_fibo(n-2))

# Vary the number of terms from 1 to 30
for n in range(1, 31):
    times = []
    cycle_counts = []

    # Take multiple trials for each variation
    for _ in range(3):
        # Initialize the cyclecounter
        libpmu.initialize_pmu_counters()

        # Get the 'before' time and cycle count
        start_time = time.time()
        start_cycles = libpmu.get_cycle_count()

        # Run the recur_fibo function on a CPU 1
        psutil.Process().cpu_affinity([1])
        recur_fibo(n)

        # Get the 'after' cycle count and time
        end_cycles = libpmu.get_cycle_count()
        end_time = time.time()

        # Get the cycle count and the amount of time used
        cycle_counts.append(end_cycles - start_cycles)
        times.append(end_time - start_time)

    # Calculate the average and standard deviation
    avg_time = np.mean(times)
    avg_cycles = np.mean(cycle_counts)
    error_time = np.std(times) / np.sqrt(3)
    error_cycles = np.std(cycle_counts) / np.sqrt(3)
```

```
    # Plot the average results for varying 'n' along with error bars of your measuremen
    plt.errorbar(n, avg_time, yerr=error_time, fmt='o')
    plt.errorbar(n, avg_cycles, yerr=error_cycles, fmt='o')

plt.show()
```



In [ ]:

In [ ]: