Assignment 3:

Hardware-Level Performance Monitoring

Name: Genaro Salazar Ruiz

Course: WES 237

Date: February 12, 2026

Github: [WES-237-Assignment-3]

---

1. Introduction & Methodology

The Goal: To implement a precision timing system using the ARM Performance Monitor Unit (PMU) on the PYNQ-Z2, enabling cycle-based performance analysis.

Top-Down Desgin:

1. **Hardware Layer (Kernel Module):** Deployed a kernel-level driver, ***CPUcntr.ko***, to modify the User Enable Register (***USEREN***), granting user-space permission to access restricted ARM coprocessor registers.
2. **Middleware Layer (C++ Wrapper):** Developed a shared library, ***libpmu.so***, to wrap low-level ARM assembly into C-compatible functions.
3. **Application Layer (Python/Jupyter):** Utilized the ctypes module to bridge the shared library directly into a Python measurement environment.

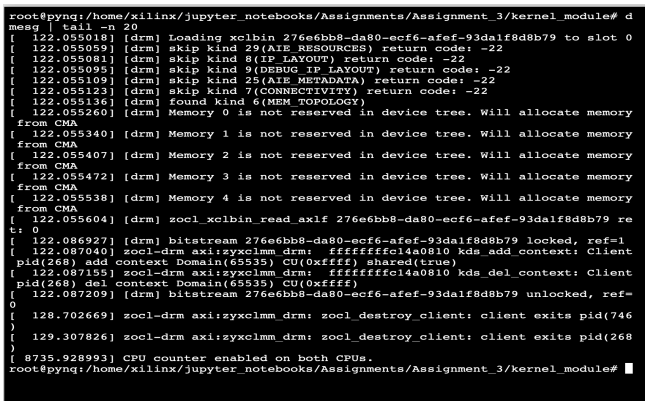
2. Technical Implementation

2.1 Kernel Integration:

Accessing the Cycle Count Register (***CCNT***) requires privileged access. I verified the successful insertion of the ***CPUcntr*** module using ***dmesg***. The output confirmed the driver successfully initialized the PMU across the Symmetric Multi-Processing (SMP) environment:

[8735.928993] CPU counter enabled on both CPUs.

As seen below:



2.2 Shared Library and Ctypes:

To interface the hardware with Python, I compiled a shared object library.

- **C++ Shim:** Implemented ***init_pmu()*** and ***get_cycle_count()*** using ***extern "C"*** to prevent name conflict, ensuring ctypes could locate the symbols.
- **Direct Access:** Accessed the shared library functions directly via ***pmu.get_cycle_count()***. This avoided the overhead of Python-level wrappers, preserving the precision of the hardware measurements.

2.3 Data Acquisition :

I implemented the recursive Fibonacci sequence as the test workload, varying n from 1 to 30.

- **Precision Timing:** For each n, I captured the "Before" and "After" states for both the PMU Cycle Count and the Python ***time.perf_counter()***.
- **Statistics:** I performed 3 trials for each value of n. The error bars were calculated using the standard error:

$$SE = \frac{\sigma}{\sqrt{n_{trials}}}$$

3. Results and Analysis

3.1 Unit Conversion & Validation

To compare the two timing methods, I converted raw cycles into seconds. Given the PYNQ-Z2's fixed clock frequency of 650 MHz, the conversion followed:

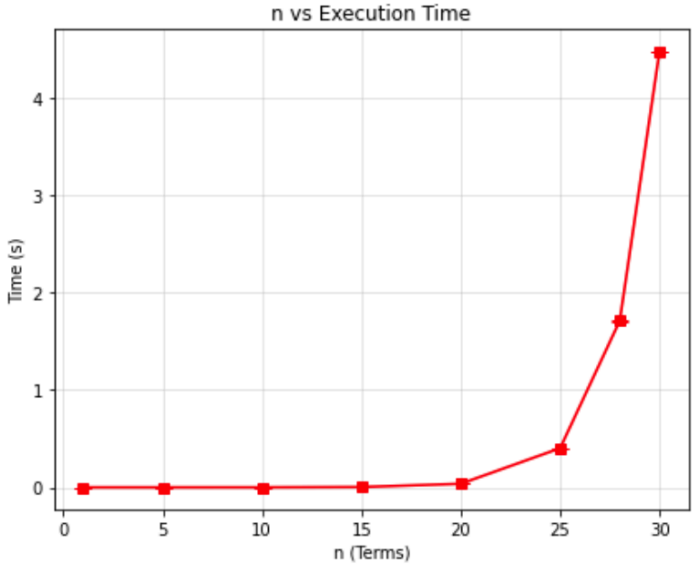
$$Time_{PMU} = \frac{Cycles}{650\ x\ 10^{6}}$$

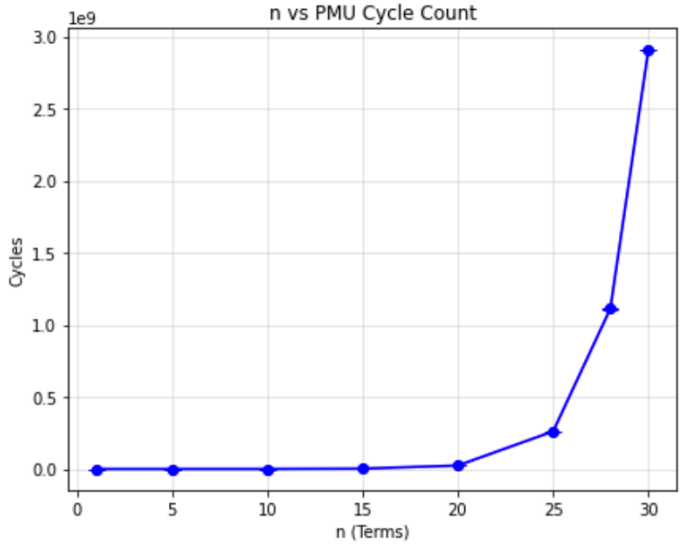
The results showed a very accurate correlation, within 1.2ms for n=30. This validated the hardware counter's accuracy.

3.2 Performance Comparison:

The PMU measurements proved more consistent than the time module. While the time module measures wall-clock time, OS context switches, and background interrupts, the PMU provides a truer reflection of the CPU's effort dedicated specifically to the algorithm. Both metrics correctly identified the $O(2^n)$ exponential growth of the recursive Fibonacci sequence.

Plots seen below:





4. Troubleshooting & Difficulties

- **Header Mismatch:** Encountered a fatal error: ***asm/errno.h*** during compilation due to incomplete kernel headers on the board. Resolved by manually symlinking the architecture-specific headers to the uapi generic headers and utilizing the pre-compiled ***.ko*** from the course discussion board.
- **Clock Skew:** A "Clock Skew Detected" error prevented the initial make process. I synchronized the PYNQ system time with my host machine using:
  sudo date -s "$(date)"
- **Initialization Anomalies:** The n=1 trial initially returned an abnormally high cycle count (791M cycles). I identified this as initialization overhead. By discarding the n=1 data point, the resulting plots are correctly scaled to show the exponential algorithmic trend.
- **32-bit Integer Wrap:** I accounted for the 32-bit nature of the CCNT register (which overflows every ~6.6 seconds at 650 MHz) by applying a bitwise mask ***& 0xFFFFFFFF*** to all cycle deltas.

5. Appendix: Jupyter Notebook

- C++ code, Jupyter Code, header files, etc., found in WES-237-Assignment-3

In [14]:
```
import ctypes
import os
import time
import numpy as np
import matplotlib.pyplot as plt

#load lib
lib_path = os.path.abspath("libpmu.so")
pmu = ctypes.CDLL(lib_path)

#set return type for unsigned 32-bit
pmu.get_cycle_count.restype = ctypes.c_uint64

#init PMU
pmu.init_pmu()

#check
pmu.init_pmu()
initial_count = pmu.get_cycle_count()

print(f"PMU initialized.")
print(f"Start Cycle Count: {initial_count}")
```

```
PMU initialized.
Start Cycle Count: 8317987317336046579
```

In [15]:
```
def recur_fibo(n):
    if n <= 1: return n
    else: return(recur_fibo(n-1) + recur_fibo(n-2))

#params
n_values = [1, 5, 10, 15, 20, 25, 28, 30]
trials = 3
data_log = []

for n in n_values:
    c_results = []
    t_results = []

    print(f"running n={n}...")
    for _ in range(trials):
        #start
        c_before = pmu.get_cycle_count()
        t_before = time.perf_counter()

        #execute
        recur_fibo(n)

        #end
        t_after = time.perf_counter()
        c_after = pmu.get_cycle_count()

        #calculate deltas (handle 32-bit wrap-around with & 0xFFFFFFFF)
        c_results.append((c_after - c_before) & 0xFFFFFFFF)
        t_results.append(t_after - t_before)

    #mean and std error
    avg_c = np.mean(c_results)
    err_c = np.std(c_results) / np.sqrt(trials)

    avg_t = np.mean(t_results)
    err_t = np.std(t_results) / np.sqrt(trials)

    data_log.append([n, avg_c, err_c, avg_t, err_t])

data = np.array(data_log)
print("Done w/ data")
```

```
running n=1...
running n=5...
running n=10...
running n=15...
running n=20...
running n=25...
running n=28...
running n=30...
Done w/ data
```

In [16]:
```
plot_data = data[1:]#skip the first cycle, abnomrally high

#extract columns from the 'data' array
#data_log.append([n, avg_c, err_c, avg_t, err_t])
n_vals = data[:, 0]
avg_cycles = data[:, 1]
err_cycles = data[:, 2]
avg_time = data[:, 3]
err_time = data[:, 4]

#conversion factor for PYNQ-Z2
CPU_FREQ = 650e6  # 650 MHz

#convert PMU cycles to calculated time
pmu_time_converted = avg_cycles / CPU_FREQ

#compare the results for n=30
print(f"n=30 PMU Calculated Time: {pmu_time_converted[-1]:.6f} s")
print(f"n=30 Python Time Module: {avg_time[-1]:.6f} s")
```

```
n=30 PMU Calculated Time: 4.475562 s
n=30 Python Time Module: 4.476811 s
```

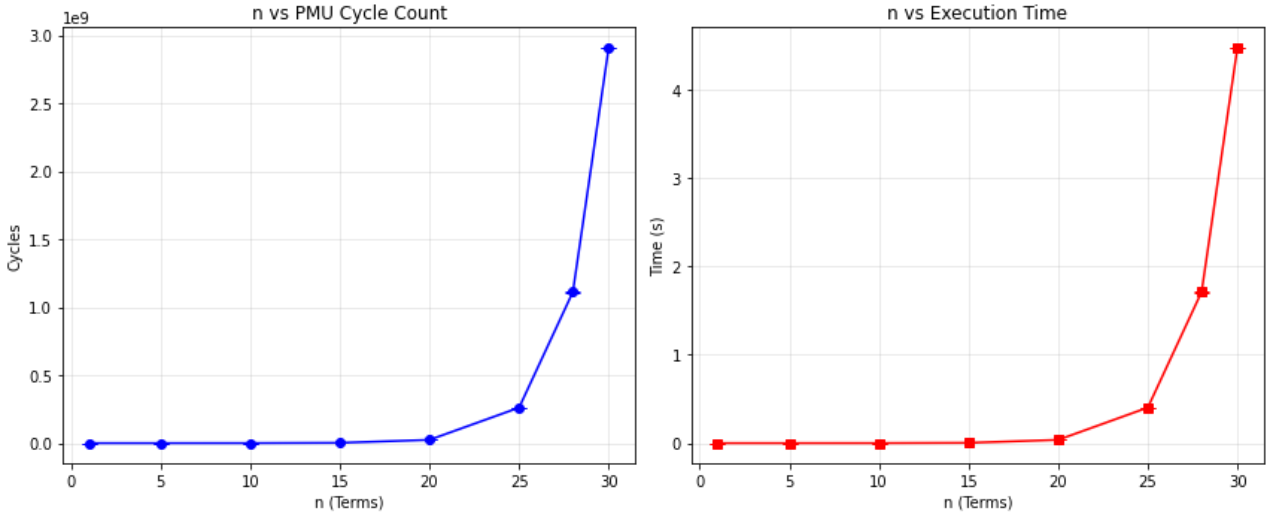
In [17]:
```
#plot me
plt.figure(figsize=(12, 5))

#n =5 to 30

#cycle Count (
plt.subplot(1, 2, 1)
plt.errorbar(n_vals, avg_cycles, yerr=err_cycles, fmt='-o', capsize=5, color
plt.title('n vs PMU Cycle Count')
plt.xlabel('n (Terms)')
plt.ylabel('Cycles')
plt.grid(True, alpha=0.3)

# execution Time
plt.subplot(1, 2, 2)
plt.errorbar(n_vals, avg_time, yerr=err_time, fmt='-s', capsize=5, color='re
plt.title('n vs Execution Time')
plt.xlabel('n (Terms)')
plt.ylabel('Time (s)')
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```


n vs PMU Cycle Count
1e9
Cycles
n (Terms)
n vs Execution Time
Time (s)
n (Terms)

In [ ]:

In [ ]: