ELC 4312 Final Project: Hardware Performance Monitor (HPM)

Gabriel Yeung

December 9, 2025

GitHub Repository and Video Demonstration

Project Source Code: `https://github.com/gyeung743/ELC4312-Final-Project`
Demonstration Video: `https://youtu.be/7f2SLQ9a5eA`

Objective

The objective of this project was to design and integrate a Hardware Performance Monitor (HPM) into the MicroBlaze MCS Softcore SoC. Standard software profiling tools often introduce significant execution overhead, which skews performance data. By implementing a hardware-based monitor that "snoops" on the system bus, this project aims to measure cycle-accurate execution latency and bus utilization without modifying the processor's internal control logic or slowing down the software under test.

Design Method

I approached the design by creating a dedicated hardware peripheral that tracks system activity and a corresponding software driver to control it.

Hardware Implementation

- **Peripheral Logic:** I designed a custom SystemVerilog module, `hpm_peripheral.sv`, which contains a set of 32-bit counters. These counters track `Total Cycles` (incrementing every clock tick) and `Bus Accesses` (incrementing on memory strobes). The module exposes a memory-mapped interface allowing the CPU to start, stop, and reset the counters via software commands.
- **Bus Integration (The "Snoop" Architecture):** I modified the top-level system, `mcs_top_vanilla.sv`, to integrate the HPM. Originally, I attempted to use the MicroBlaze "Trace Bus" interface to count retired instructions. However, due to version mismatches with the IP core, these signals were unavailable. I successfully moved on to a "Bus Snooping" architecture. I wired the HPM probes to the `io_addr_strobe`, `io_read_strobe`, and `io_write_strobe` signals. This allows the monitor to count every time the CPU fetches an instruction or accesses data memory.
- **MMIO Mapping:** I updated `chu_io_map.svh` to assign the HPM to Slot 5 (`S5_HPM`) and modified `mmio_sys_vanilla.sv` to route the read/write signals to the new core.

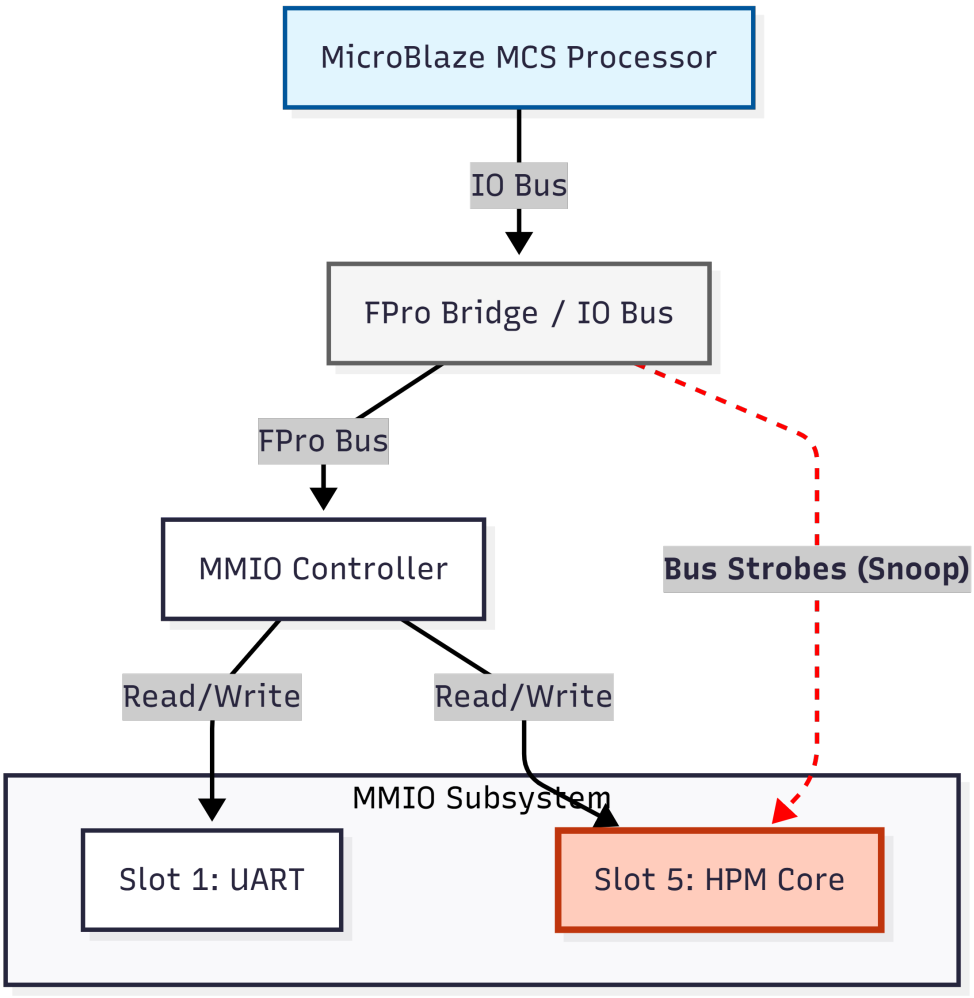

Figure 1: Hardware Block Diagram. The red dashed line indicates the HPM "snooping" on the Bus Strobes to monitor traffic without interrupting the CPU.

Software Driver & Application

- **Driver Class:** I developed a C++ class, `HpmCore`, to abstract the hardware registers. It provides simple methods like `start()`, `stop()`, and `print_stats()` so that benchmarking code remains clean and readable.
- **Benchmark Suite:** To demonstrate the utility of the HPM, I wrote two distinct stress tests in `main_hpm.cpp` to generate statistically significant data:
  1. **Arithmetic Benchmark:** A loop performing **1,000,000 iterations** of integer division (`sum / 3`) to test the CPU's computational throughput.
  2. **Memory Benchmark:** A loop performing **10,000,000 memory copies** to test the bandwidth of the local Block RAM (BRAM).

Experimental Results

The HPM successfully provided cycle-accurate timing for the benchmarks. The results displayed a significant architectural characteristic of the Softcore SoC.

Benchmark	Total Cycles	Time (us)	Time (s)	Cycles Per Op
Arithmetic (1M iterations)	425,063,024	4,250,630	**4.25 s**	$\approx$ **425**
Memory (10M operations)	130,500,004	1,305,000	**1.31 s**	$\approx$ **13**

Table 1: HPM Output Data (High Intensity Test)

The **Arithmetic Benchmark** was surprisingly slow, averaging 425 cycles per operation. This confirms that the vanilla MicroBlaze MCS lacks a hardware divider, which forces the compiler to use a slow software library routine for the division. In contrast, the **Memory Benchmark** averaged only 13 cycles per operation, which showed that the local BRAM allows for high-speed data access. This insight identifies the system as **Compute Bound** and not Memory Bound.

Testing and Conclusion

The implementation process required overcoming several significant challenges:

1. **Trace Bus Unavailability:** My initial design relied on the CPU's internal `TRACE_Valid_Instr` signal. When synthesis failed because the IP core version did not support these ports, I modified the design to monitor the external IO bus strobes instead. This successfully provided a proxy metric for system utilization.

2. **Software Build Issues:** I encountered a linker error stating "undefined reference to main" because the Vitis build system did not detect the new `main_hpm.cpp` file. I resolved this by deleting the `build` directory to force a CMake re-scan. Additionally, I had some ambiguity errors in the `uart.disp()` function due to type mismatches between `uint32_t` and `int`, which I fixed by explicit type casting in the driver.

3. **Memory Overflow:** Initially, the application exceeded the default 128KB memory limit. I resolved this by changing the optimization settings in the CMake configuration file to "-Os" and added the flags "-fno-exceptions -fno-rtti". This ensured that there was sufficient overhead for the C++ standard libraries and the benchmark arrays.

The final system is fully functional, which meets the requirement to build a custom SoC extension. The HPM provides a valuable tool for future profiling and is capable of distinguishing between compute-heavy and memory-heavy workloads with zero software overhead.