Homework 1 — Microarchitecture Timing

ECE/CSC: Microarchitecture Security & Reverse Engineering (Fall 2025)

Instructor: Dr. Samira Ajorpaz

Release: **August 24, 2025** Due: **September 2, 2025 (11:59 PM ET)**

1 Timing

The most fundamental operation of microbenchmarking is timing. This is complicated on modern processors which dynamically reorder instructions. X86 provides timing instructions in the form of `RDTSC` and `RDTSCP`. These read from the timestamp counter, a counter that ticks up at a constant rate of 1 tick/0.5 ns (or 2 GHz) on Artemisia, regardless of CPU frequency. This rate varies across CPU architectures, and pre-Haswell (Intel CPUs before 2014) counted clock cycles instead of wall-clock time. The difference between them is that `RDTSCP` serializes the instruction execution - it ensures instructions before it finish executing before instructions after it execute - whereas RDTSC doesn't. This is important, as the CPU can execute RDTSC pretty much anywhere due to out-of-order execution, whereas RDTSCP enforces exactly where it executes.

Take the following code block:

```
RDTSC
MOV %RDX, [%RCX]
RDTSC
```

This stores the contents of the memory location stored at %RCX into %RDX and the `RDTSC`s surrounding it grab the timestamp; we can subtract the second timestamp from the first to get the timing of this memory access. Due to out-of-order execution, the CPU can reorder these any way it sees fit. This means execution order could potentially be, for example, `MOV` → `RDTSC` → `RDTSC` instead of what we want it to be, `RDTSC` → `MOV` → `RDTSC`. Taking the difference between the `RDTSC`s, we would not obtain a meaningful value since the `MOV` would not have executed between them. This is where `RDTSCP` comes in - it enforces the program ordering of the instructions in the CPU; this also known as *serialization*. We can rewrite the code like this:

```
RDTSCP
MOV %RDX, [%RCX]
RDTSCP
```

With this, we ensure that `RDTSCP` → `MOV` → `RDTSCP` is the only possible execution and thus we can time this instruction accurately. On x86, serialization without grabbing the timestamp is often accomplished with the `CPUID` instruction.

2 Compiler Considerations

2.1 Compiler Intrinsics

Instead of manually writing assembly, compilers provide intrinsics that allow you to stay in C. The intrinsic for `RDTSCP` on GCC is `unsigned __int64 __rdtscp (unsigned int * mem_addr)`. You can call this in C code by adding this line at the top: `#include <immintrin.h>`.

A full list of compiler intrinsics is located here:
https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

2.2 Compilation Options

Compilers will generate assembly that doesn't follow code exactly due to optimizations. To get the assembly maximally close to the source code, use the compiler flag `-O0`, which disables most optimizations. However, if you have a read to a variable that isn't ever used, `-O0` will still remove that read. In this case, mark the variable as `volatile` in the source code, which tells the compiler not to optimize out reads and writes to that variable.

2.3 Writing Assembly in C

Often, you will have to write assembly in C because the compiler isn't generating the specific assembly that you want. This can be done with an `asm` block. For example,

```
asm volatile(
"MOV %eax, [%ebx]\n"
"MOV %ecx, [%edx]\n"
)
```

2.4 Variable alignment

Often you will be measuring something related to the memory system such as the cache or TLB. A cache line on x86 is 8 bytes while a page (by default) is 4096 bytes. It is sometimes desirable to have a variable be aligned so that it is in its own cache line or own page. This

is accomplished with variable alignment. An example is as follows:

```
char __attribute__((aligned(4096))) x;
```

This aligns a 1-byte value to a 4096 (page) boundary, which puts it on its own page.

3 Studies

3.1 Cache Timing

The timestamp counter gives us enough resolution to differentiate between a cache hit and a miss. To guarantee the eviction of a cache line, we can use `CLFLUSH`. However, due to out-of-order execution, this instruction can execute anywhere in the dynamic instruction stream. To fix this, we use `MFENCE` which ensures memory operations complete before continuing on execution. We recommend pairing MFENCE with CLFLUSH when trying to ensure a line is not in the cache. The compiler intrinsic for `MFENCE` is `_mm_mfence()` and for `CLFLUSH` is `_mm_clflush()`.

Exercise 1. Time a cache hit and miss, report the results. Repeat 1M times and plot a distribution of data collected for misses and hits.

3.2 AVX Timing

AVX is a SIMD unit, executing the same instruction on multiple data. On Sapphire Rapids, there are 3 AVX versions - AVX (16 byte registers), AVX2 (32 byte registers), and AVX512 (64 byte registers). An example program is below, using AVX2.

```
#include <immintrin.h>
int main() {
    __m256 evens = _mm256_set_ps(2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0);
    __m256 odds = _mm256_set_ps(1.0, 3.0, 5.0, 7.0, 9.0, 11.0, 13.0, 15.0);
    volatile __m256 result;

    result = _mm256_mul_ps(evens, odds);
}
```

`_mm256_set_ps` creates a virtual register that will eventually load the given 4 byte values in, and `_mm256_mul_ps` will do a pairwise multiplication of the two given virtual registers.

Exercise 2. Time any AVX2 operation.

Exercise 3. Time any AVX512 operation.

Exercise 4. Time an AVX512 multiplication, wait for 100 ms (for example with `usleep`), and time another AVX512 multiplication. Is there a timing difference? Repeat 1M times and plot a distribution of data.

Exercise 5. Time an AVX multiplication, wait for 100 ms, and time another AVX multiplication. Is there a timing difference? Repeat 1M times and plot a distribution of data.

4 Resources for Example Codes Tested on Artemisia Server and Further Readings

- GateBleed repository - https://github.com/jkalya/gatebleed
- GateBleed paper - https://arxiv.org/abs/2507.17033
- Intel Compiler Intrinsics - link
- Intel Software Developer Manual - link

Connecting to Artemisia Server:

- Using the NCSU student credentials, you should must first connect via the **NC State University VPN client**.
- Install **Cisco Secure Client 5.x**:
  - Windows: Download for Windows
  - macOS: Download for macOS
  - Linux: Download for Linux
- After installation:
  - Connect to `vpn.ncsu.edu`
  - Change the default group to **8-Workshop-Temp**
  - Enter your username and password (as provided).
- Once VPN is connected, use any SSH client:
  - `ssh <username>@artemisia.ece.ncsu.edu`