KTH Information and Communication Technology

IS1200/IS1500

Introduction to DTEK-V Hardware Performance Counters
2024-10-04
v1.1

Introduction

Hardware performance counters are units inside the processor that collect information about different states in the processors that may influence the performance of the system. By performance, we often talk about how fast (execution time) a system runs an application, but it could also mean how energy-efficient a system is. These counters are often built into the hardware and can be accessed through special control and status registers (CSRs).

This document outlines and introduces you to the hardware performance counters on the DTEK-V board.

Hardware Performance Counters

The DTEK-V board has a total of nine different hardware performance counters which continuously monitor the state and events of the system. These counters can be both written to and read from.

Each counter is mapped to a specific CSR register. Some hardware counters exist on all RISC-V based systems that support them and thus have a dedicated name, such as, for example, the **`mcycle`** register, which counts the number of elapsed clock cycles. Other hardware performance counters, such as collecting the total number of data-cache misses, are specific to a particular architecture and are mapped to the **`mhpmcounterX`** registers, where **`X`** is a number corresponding to the performance counter you want to collect.

Each hardware counter is 64-bit wide, enabling collection of a large number of events. However, since each register in the system only is 32-bit wide, the most-significant 32-bit of a counter has the name suffixed with the constant **h**. For example, to read the least significant 32-bits of the cycle counter, one must read from **mcycle**, while reading the most significant 32-bits of the cycle counter requires to ready from **mcycleh**. Similar reasoning holds for the **mhpmcounters/h**.

Note: *Unless your application runs for a long time, it is often enough to only work with the lower 32-bits of the counter.*

In real systems, the number of hardware counters is often very limited, and one can only select to measure a limited set of events. For example, an ARM Cortex-A5, can only count two events at a single time. However, the DTEK-V board has no such limit and can count all nine counters simultaneously.

The following table overviews all the different performance counters, and the remaining chapters describes them more in details.

Name	Description
`mcycle` `mcycleh`	Counts the number of clock cycles that elapsed.
`minstret` `minstreth`	Counts the number of instructions that have been retired.
`mhpmcounter3` `mhpmcounter3h`	Counts the number of memory instructions that have been retired.
`mhpmcounter4` `mhpmcounter4h`	Counts the number of times an instruction-fetch resulted in a I-cache miss.
`mhpmcounter5` `mhpmcounter5h`	Counts the number of times an memory operation resulted in a D-cache miss.
`mhpmcounter6` `mhpmcounter6h`	Counts the number of stalls the CPU experienced due to I-cache misses.
`mhpmcounter7` `mhpmcounter7h`	Counts the number of stalls the CPU experienced due to D-cache misses.
`mhpmcounter8` `mhpmcounter8h`	Counts the number of stalls that the CPU experienced due to data hazards that could not be solved by forwarding.
`mhpmcounter9` `mhpmcounter9h`	Counts the number of stalls that the CPU experienced due to expensive ALU operations.

Clock Cycle Counter (mcycle / mcycleh)

The clock cycle counter counts up with every riding edge of the processor clock. Since the DTEK-V board runs at 30 MHz, it counts $30*10^6$ times per second. The intention of using the clock cycle counter is for timing reasons, such as measuring the time it takes to execute a particular function or parts of a code.

Similar functionality can, in theory, be accomplished by using the memory-mapped timer, but since accessing the timer is performed through the system bus (which can take a longer time to access), the clock cycle timer is much faster (the read occurs in the merged IF-stage of the processor).

Accessing the clock cycle counter involves reading from the **`mcycle`** (and **`mcycleh`**) CSR register.

Instruction Counter (minstret / minstreth)

The instruction counter counts up every time the processor finishes executing an instruction (that is, the instruction is retired). Since the DTEK-V processor is an in-order scalar processor, it can retire at most 1 instruction/clock cycle, which translates to 30 million operations per second (30 MOPS).

The intention of using the instruction counter is to find out how many instructions a certain, performance-critical part of the program consumed. More importantly, by measuring both the instruction- and clock cycle, we can compute the important metric IPC (# instructions executed / cycle).

Accessing the instruction counter involves reading from the **minstret** (and ***minstreh***) CSR register. In RISC-V assembly, for example, the following code can be used to measure the number of instructions something consumed to perform some action.

Memory Instruction Counter (mhpmcounter3 / mhpmcounter3h)

The memory instruction counter counts up every time the processor retires a memory (load or store) instruction. The DTEK-V processor can read or written to the memory during every clock cycle.

The intention behind this counter is to understand how much of your application uses memory in order to reflect whether a particular code is compute- or memory-bound[1]. Furthermore, by combining the memory instruction counter with the data cache counter, we can measure the data-cache miss (or hit) ratio.

Accessing the instruction counter involves reading from the **mhpmcounter3** (and *mhpmcounter3h*) CSR register.

I-Cache Miss Counter (mhpmcounter4 / mhpmcounter4h)

The instruction cache miss counter counts up every time the processor issues a fetch instruction but misses in the instruction cache. Since the processor can issue one fetch every cycle, the theoretical number of misses is $3.75*10^6$ per second but is very likely to be much larger (since satisfying a miss often takes longer than 8 clock cycle).

The intention behind using this counter is to understand how well your code utilizes the instruction cache. Typically, instruction cache hit rates are in the order of 90%+, and having lower than that indicates that you might want to restructure the code. More importantly, we can combine the instruction cache miss counter with the minstret counter to calculate the derived metric instruction cache hit rate.

[1] For more information you can consult the course literature or visit (for example) https://en.wikipedia.org/wiki/CPU-bound or https://en.wikipedia.org/wiki/Memory-bound_function

Accessing the instruction counter involves reading from the mhpmcounter4 (and mhpmcounter4h) CSR register.

D-Cache Miss Counter (mhpmcounter5 / mhpmcounter5h)

The data cache miss counter counts up every time the processor issues a load or store instruction but misses in the data cache. Since the processor can issue one load or store every cycle, the theoretical number of misses is $3.75*10^6$ per second but is very likely to be much larger (since satisfying a miss often takes longer than 8 clock cycle).

The intention behind using this counter is to understand how well your code utilizes the data cache. Typically, data cache hit rates are in the order of 90%+, and having lower than that indicates that you might want to restructure the code. More importantly, we can combine the instruction cache miss counter with the memory instruction counter to estimate the derived metric data cache hit rate.

Accessing the instruction counter involves reading from the mhpmcounter5 (and mhpmcounter5h) CSR register.

I-Cache Stall Counter (mhpmcounter6 / mhpmcounter6h)

The instruction cache stall counter counts up every time the processor is stalled due to an instruction cache miss waiting to be satisfied. The intention behind using this counter is to understand the cost of missing in the instruction cache and, hence, having to go to external memory to satisfy the miss. You can combine this metric with the cycle counter to understand how often your processor is stalling or use the instruction cache miss rate counter to estimate the cost of missing in the instruction cache.

Accessing the instruction counter involves reading from the mhpmcounter6 (and mhpmcounter6h) CSR register.

D-Cache Stall Counter (mhpmcounter7 / mhpmcounter7h)

The data cache stall counter counts up every time the processor is stalled due to a data cache miss waiting to be satisfied. The intention behind using this counter is to understand the cost of missing in the data cache and, hence, having to go to external memory to satisfy the miss. You can combine this metric with the cycle counter to understand how often your processor is stalling or use the d cache miss rate counter to estimate the cost of missing in the instruction cache.

Accessing the instruction counter involves reading from the mhpmcounter7 (and mhpmcounter7h) CSR register.

Data Hazard Stall Counter (mhpmcounter8 / mhpmcounter8h)

The data hazard stall counter counts up every time the processor has to stall due to an ongoing data hazard that cannot be solved by forwarding. The intention behind using this counter is to gauge how well forwarding works for your application, as well as estimate the cost that hazards have on your architecture. You can combine measuring the data hazard stalls with the clock cycle counter to understand the overall impact it has on your application.

Accessing the instruction counter involves reading from the mhpmcounter8 (and mhpmcounter8h) CSR register.

ALU Stall Counter (mhpmcounter9 / mhpmcounter9h)

The ALU stall counter counts up every time the processor has to stall due to an expensive ALU operation taking place. This, more specifically, applies to the division operation, which consumes many more cycles than other operations in the system. The intention behind using this counter is to gauge the impact divisions have on your overall system performance, and it can be combined with the clock cycle counter to understand the overall impact it has on your application.

Accessing the instruction counter involves reading from the mhpmcounter9 (and mhpmcounter9h) CSR register.

Accessing the CSR Registers in C or ASM

There are two ways of accessing the hardware counters through the CSR registers. One is by writing assembler code (ASM), and the other is by inlining assembler code in C. Both work equally well, and it is up to personal preference which to use.

Clearing and Reading CSRs in Assembler

Suppose you have written a function, foo, that you would like to profile in order to understand how long time it took to execute. To perform this, you would first have to: **(i)** clear the **mcycle** counter, **(ii)** call the foo function, and **(iii)** read the **mcycle** counter and print it.

The code for doing (i)-(iii) in assembler is shown below:

```
// Clear the mcycle CSR by writing 0 to it
csrw mcycle, x0

// Call the foo function
jal foo

// Read mcycle into a0 (argument a0)
csrr a0, mcycle

// Insert code to print out the value of t0 to know how long time foo consumed
jal print_dec
```

Clearing and Reading CSRs in C

Similar to the previous example, suppose you have a function, foo, that you would like to profile in order to understand how long time it took to execute in C. To perform this, you would first have to **(i)** clear the **mcycle** counter, **(ii)** call the foo function, and **(iii)** read the **mcycle** counter and print it.

The code for doing (i)-(iii) in C is shown below:

```
unsigned int foo_time; // Create a variable called foo_time

// Clear the mcycle CSR by writing 0 to it
asm volatile ("csrw mcycle, x0");

// Call the foo function
foo();

// Read the mcycle value into foo_time
asm("csrr %0, mcycle" : "=r"(foo_time) );

// Print out the value of foo_time (requires print_dec in time4riscv.zip)
print("\nTime for foo() was: ");
print_dec(foo_time);
print("\n");
```

Note: *above examples only showed the 32-bit version. If foo takes longer than ~2 minutes to execute, you would have to read the* `mcycleh` *counter as well and merging it with* `mcycle` *to obtain a 64-bit value.*

Note: *you do not have to clear the mcycle counter, but then you have to record it at the beginning of the profiling session and subtract it from the end (that is, you print out the difference in mcycle between before and after the foo function).*

An Important Note about Stalls

You will notice that several hardware counters measure events relative to stalling. For example, we have the data hazard stall counters as well as the I-cache stall counter. It is important to understand that one type of stall does not exclude another, which, in essence, means that multiple types of stalls can occur in the processor at the same time. This also means that summing all individual stalls in the system will result in a sum larger than the time it took to run your application. It is important to take this into consideration when doing performance analysis.

Performance Metrics and Derived Metrics

Multiple metrics are needed to understand the performance of a system. Some of these metrics are called derived metrics, which means that they have been obtained by combining multiple other metrics together. The following is a small list of important metrics, how they are calculated, and their meaning:

Metric	Calculation	Usage
Execution time (s)	$\frac{mcycle}{30\,000\,000}$	Execution time in seconds (s)
CPI (Cycles/Instr)	$\frac{mcycle}{minstret}$	Indication of processor performance. Lower is better. Cannot be lower than 1 CPI.
IPC (Instr/Cycle)	$\frac{minstret}{mcycle}$	Indication of processor performance. Higher is better. Cannot be higher than 1 IPC.
D-cache miss ratio	$\frac{mhpmcounter5}{mhpmcounter3}$	Fraction of data cache misses. Lower is better. Often <20%.
D-cache hit ratio	$1 - \frac{mhpmcounter5}{mhpmcounter3}$	Fraction of data cache hits. Higher is better. Oftne >80%.
I-cache miss ratio	$\frac{mhpmcounter4}{minstret}$	Fraction of instruction cache misses. Lower is better. Often <5%.
I-cache hit ratio	$1 - \frac{mhpmcounter4}{minstret}$	Fraction of instruction cache hits. Higher is better. Often >95%.
ALU-stall ratio	$\frac{mhpmcounter9}{mcycle}$	Fraction of all cycles that were ALU stalls.
Memory Intensity	$\frac{mhpmcounter3}{minstret}$	Fraction of instructions that were memory.
Hazard-stall ratio	$\frac{mhpmcounter8}{mcycle}$	Fraction of all cycles were due to hazard stalls.
Cache misses	$mhpmcounter4 + mhpmcounter5$	Sum of all cache misses

Version history

1.0 – First published version. 2024-10-10.

1.1 – Fixed error in I-cache hit/miss calculation (2024-11-26)