Chapter 12

"Zicntr" and "Zihpm" Counters

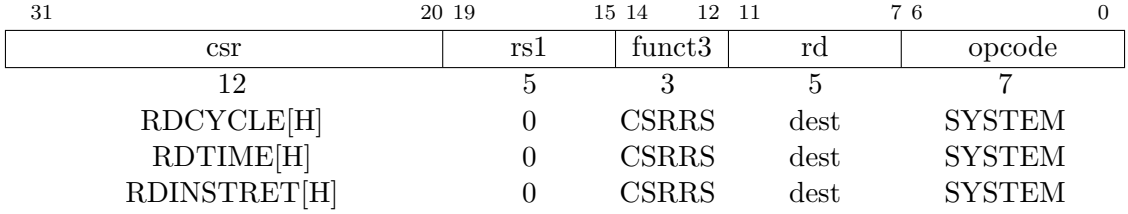
RISC-V ISAs provide a set of up to thirty-two 64-bit performance counters and timers that are accessible via unprivileged XLEN-bit read-only CSR registers `0xC00`–`0xC1F` (when XLEN=32, the upper 32 bits are accessed via CSR registers `0xC80`–`0xC9F`). These counters are divided between the "Zicntr" and "Zihpm" extensions.

12.1 "Zicntr" Standard Extension for Base Counters and Timers

The Zicntr standard extension comprises the first three of these counters (CYCLE, TIME, and INSTRET), which have dedicated functions (cycle count, real-time clock, and instructions retired, respectively). The Zicntr extension depends on the Zicsr extension.

We recommend provision of these basic counters in implementations as they are essential for basic performance analysis, adaptive and dynamic optimization, and to allow an application to work with real-time streams. Additional counters in the separate Zihpm extension can help diagnose performance problems and these should be made accessible from user-level application code with low overhead.

Some execution environments might prohibit access to counters, for example, to impede timing side-channel attacks.



31 20	19 15	14 12	11 7	6 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
RDCYCLE[H]	0	CSRRS	dest	SYSTEM
RDTIME[H]	0	CSRRS	dest	SYSTEM
RDINSTRET[H]	0	CSRRS	dest	SYSTEM

For base ISAs with XLEN≥64, CSR instructions can access the full 64-bit CSRs directly. In particular, the RDCYCLE, RDTIME, and RDINSTRET pseudoinstructions read the full 64 bits of the `cycle`, `time`, and `instret` counters.

The counter pseudoinstructions are mapped to the read-only `csrrs rd, counter, x0` *canonical form, but the other read-only CSR instruction forms (based on CSRRC/CSRRSI/CSRRCI) are also legal ways to read these CSRs.*

For base ISAs with XLEN=32, the Zicntr extension enables the three 64-bit read-only user-level counters to be accessed in 32-bit pieces. The RDCYCLE, RDTIME, and RDINSTRET pseudoinstructions provide the lower 32 bits, and the RDCYCLEH, RDTIMEH, and RDINSTRETH pseudoinstructions provide the upper 32 bits of the respective counters.

We required the counters be 64 bits wide, even when XLEN=32, as otherwise it is very difficult for software to determine if values have overflowed. For a low-end implementation, the upper 32 bits of each counter can be implemented using software counters incremented by a trap handler triggered by overflow of the lower 32 bits. The sample code given below shows how the full 64-bit width value can be safely read using the individual 32-bit width pseudoinstructions.

The RDCYCLE pseudoinstruction reads the low XLEN bits of the `cycle` CSR which holds a count of the number of clock cycles executed by the processor core on which the hart is running from an arbitrary start time in the past. RDCYCLEH is only present when XLEN=32 and reads bits 63–32 of the same cycle counter. The underlying 64-bit counter should never overflow in practice. The rate at which the cycle counter advances will depend on the implementation and operating environment. The execution environment should provide a means to determine the current rate (cycles/second) at which the cycle counter is incrementing.

RDCYCLE is intended to return the number of cycles executed by the processor core, not the hart. Precisely defining what is a "core" is difficult given some implementation choices (e.g., AMD Bulldozer). Precisely defining what is a "clock cycle" is also difficult given the range of implementations (including software emulations), but the intent is that RDCYCLE is used for performance monitoring along with the other performance counters. In particular, where there is one hart/core, one would expect cycle-count/instructions-retired to measure CPI for a hart.

Cores don't have to be exposed to software at all, and an implementor might choose to pretend multiple harts on one physical core are running on separate cores with one hart/core, and provide separate cycle counters for each hart. This might make sense in a simple barrel processor (e.g., CDC 6600 peripheral processors) where inter-hart timing interactions are non-existent or minimal.

Where there is more than one hart/core and dynamic multithreading, it is not generally possible to separate out cycles per hart (especially with SMT). It might be possible to define a separate performance counter that tried to capture the number of cycles a particular hart was running, but this definition would have to be very fuzzy to cover all the possible threading implementations. For example, should we only count cycles for which any instruction was issued to execution for this hart, and/or cycles any instruction retired, or include cycles this hart was occupying machine resources but couldn't execute due to stalls while other harts went into execution? Likely, "all of the above" would be needed to have understandable performance stats. This complexity of defining a per-hart cycle count, and also the need in any case for a total per-core cycle count when tuning multithreaded code led to just standardizing the per-core cycle counter, which also happens to work well for the common single hart/core case.

Standardizing what happens during "sleep" is not practical given that what "sleep" means is not standardized across execution environments, but if the entire core is paused (entirely clock-gated or powered-down in deep sleep), then it is not executing clock cycles, and the cycle count shouldn't be increasing per the spec. There are many details, e.g., whether clock cycles required to reset a processor after waking up from a power-down event should be counted, and these are considered execution-environment-specific details.

Even though there is no precise definition that works for all platforms, this is still a useful facility for most platforms, and an imprecise, common, "usually correct" standard here is better than no standard. The intent of RDCYCLE was primarily performance monitoring/tuning, and the specification was written with that goal in mind.

The RDTIME pseudoinstruction reads the low XLEN bits of the `time` CSR, which counts wall-clock real time that has passed from an arbitrary start time in the past. RDTIMEH is only present when XLEN=32 and reads bits 63–32 of the same real-time counter. The underlying 64-bit counter increments by one with each tick of the real-time clock, and, for realistic real-time clock frequencies, should never overflow in practice. The execution environment should provide a means of determining the period of a counter tick (seconds/tick). The period should be constant within a small error bound. The environment should provide a means to determine the accuracy of the clock (i.e., the maximum relative error between the nominal and actual real-time clock periods).

On some simple platforms, cycle count might represent a valid implementation of RDTIME, in which case RDTIME and RDCYCLE may return the same result.

It is difficult to provide a strict mandate on clock period given the wide variety of possible implementation platforms. The maximum error bound should be set based on the requirements of the platform.

The real-time clocks of all harts in a single user application must be synchronized to within one tick of the real-time clock.

As with other architectural mandates, it suffices to appear "as if" harts are synchronized to within one tick of the real-time clock, i.e., software is unable to observe that there is a greater delta between the real-time clock values observed on two harts.

The RDINSTRET pseudoinstruction reads the low XLEN bits of the `instret` CSR, which counts the number of instructions retired by this hart from some arbitrary start point in the past. RDINSTRETH is only present when XLEN=32 and reads bits 63–32 of the same instruction counter. The underlying 64-bit counter should never overflow in practice.

Instructions that cause synchronous exceptions, including ECALL and EBREAK, are not considered to retire and hence do not increment the `instret` *CSR.*

The following code sequence will read a valid 64-bit cycle counter value into `x3`:`x2`, even if the counter overflows its lower half between reading its upper and lower halves.

```
again:
    rdcycleh    x3
    rdcycle     x2
    rdcycleh    x4
    bne         x3, x4, again
```

Figure 12.1: Sample code for reading the 64-bit cycle counter when XLEN=32.

12.2 "Zihpm" Standard Extension for Hardware Performance Counters

The Zihpm extension comprises up to 29 additional unprivileged 64-bit hardware performance counters, `hpmcounter3`–`hpmcounter31`. When XLEN=32, the upper 32 bits of these performance counters are accessible via additional CSRs `hpmcounter3h`–`hpmcounter31h`. The Zihpm extension depends on the Zicsr extension.

In some applications, it is important to be able to read multiple counters at the same instant in time. When run under a multitasking environment, a user thread can suffer a context switch while attempting to read the counters. One solution is for the user thread to read the real-time counter before and after reading the other counters to determine if a context switch occurred in the middle of the sequence, in which case the reads can be retried. We considered adding output latches to allow a user thread to snapshot the counter values atomically, but this would increase the size of the user context, especially for implementations with a richer set of counters.

The implemented number and width of these additional counters, and the set of events they count, is platform-specific. Accessing an unimplemented or ill-configured counter may cause an illegal instruction exception or may return a constant value.

The execution environment should provide a means to determine the number and width of the implemented counters, and an interface to configure the events to be counted by each counter.

For execution environments implemented on RISC-V privileged platforms, the privileged architecture manual describes privileged CSRs controlling access by lower privileged modes to these counters, and to set the events to be counted.

Alternative execution environments (e.g., user-level-only software performance models) may provide alternative mechanisms to configure the events counted by the performance counters.

It would be useful to eventually standardize event settings to count ISA-level metrics, such as the number of floating-point instructions executed for example, and possibly a few common microarchitectural metrics, such as "L1 instruction cache misses".