

How can I measure clock cycles per instruction (CPI) using TRACE32?

2025-08-21 - Comments (0) - Tracing

There are mainly two methods to obtain **CPI** (clock cycles per instruction) or **MIPS** (Millions of Instructions Per Second) in TRACE32.

BenchMark Counters

The first method is **BMC (BenchMark Counters)**, which in the case of Arm processors correspond to the PMU (Performance Monitoring Unit). BMC is supported by the TRACE32 BMC.* command group. The results are directly obtained from the core counters, which makes BMC the recommended primary method for measuring CPI or MIPS.

In addition, BMC often provides counters for cache hits and misses, which may also be useful in the context of CPI analysis. To check whether BMC is supported for your processor architecture, refer to the *Processor Architecture Manual* for your target in [main.pdf](#). Generic BMC commands are documented in the [General Commands Reference Guide B](#).

Example for Arm Cortex-A72

BMC.state

BMC.AutoInit ON

BMC.CLOCK 1000MHz

BMC.PMN0.EVENT INST_RETIRED

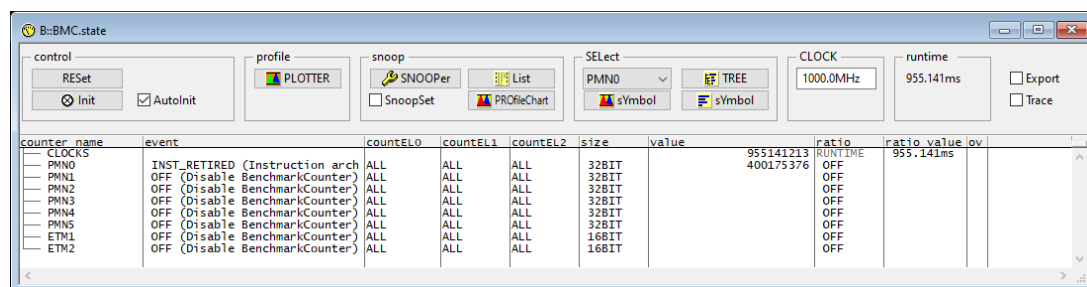
BMC.PMN0.RATIO X/TIME

BMC.Init

Go

WAIT 1.s

Break



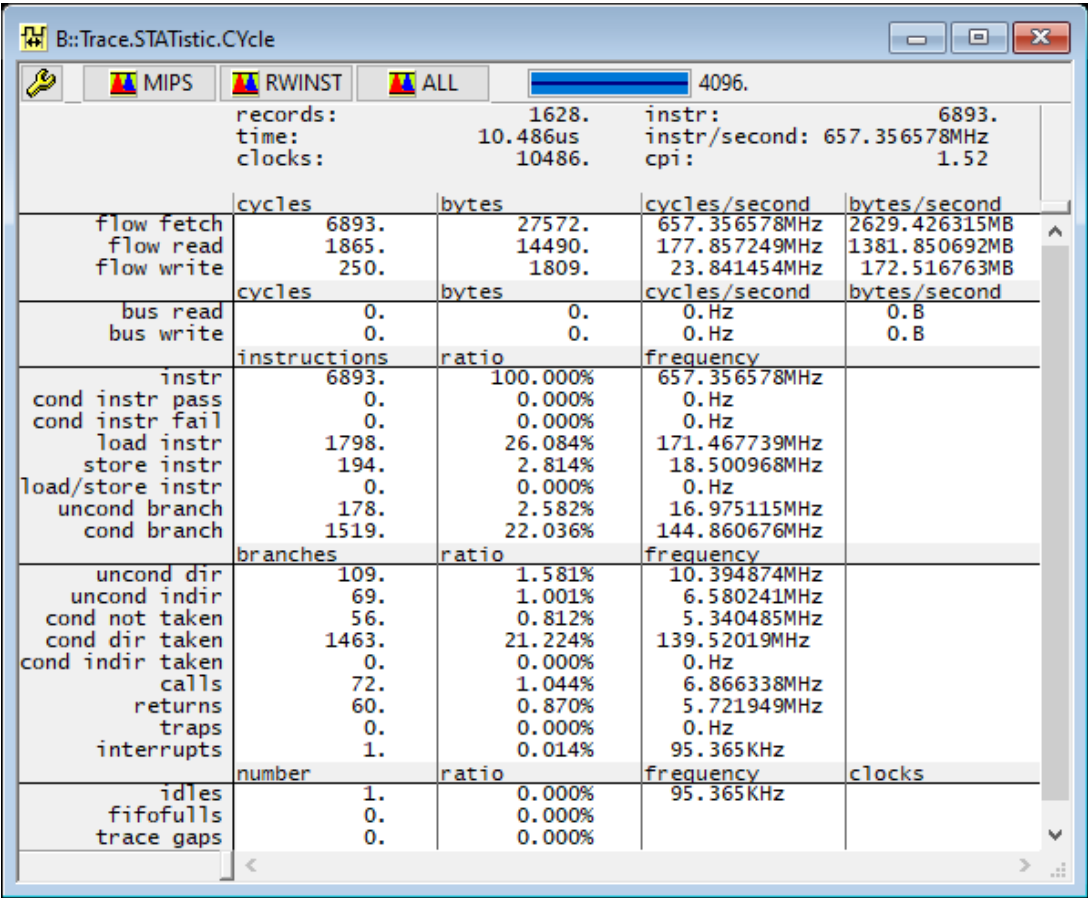
| counter_name | event | countE0 | countE1 | countE2 | size | value | ratio | runtime | ratio_value | ov |
|--------------|--------------------------------|---------|---------|---------|-------|-----------|-------|-----------|-------------|----|
| CLOCKS | | | | | | | | 955.141ms | 955.141ms | |
| PMN0 | INST_RETIRED (Instruction arch | ALL | ALL | ALL | 32BIT | 400175376 | OFF | | | |
| PMN1 | OFF (Disable BenchmarkCounter) | ALL | ALL | ALL | 32BIT | | OFF | | | |
| PMN2 | OFF (Disable BenchmarkCounter) | ALL | ALL | ALL | 32BIT | | OFF | | | |
| PMN3 | OFF (Disable BenchmarkCounter) | ALL | ALL | ALL | 32BIT | | OFF | | | |
| PMN4 | OFF (Disable BenchmarkCounter) | ALL | ALL | ALL | 32BIT | | OFF | | | |
| PMN5 | OFF (Disable BenchmarkCounter) | ALL | ALL | ALL | 32BIT | | OFF | | | |
| ETM1 | OFF (Disable BenchmarkCounter) | ALL | ALL | ALL | 16BIT | | OFF | | | |
| ETM2 | OFF (Disable BenchmarkCounter) | ALL | ALL | ALL | 16BIT | | OFF | | | |

Using Trace-Based Measurements

The second method is to use **trace-based measurements** to obtain **CPI** or **MIPS** values. This approach is mainly useful for metrics not covered by BMC counters. CPI can be

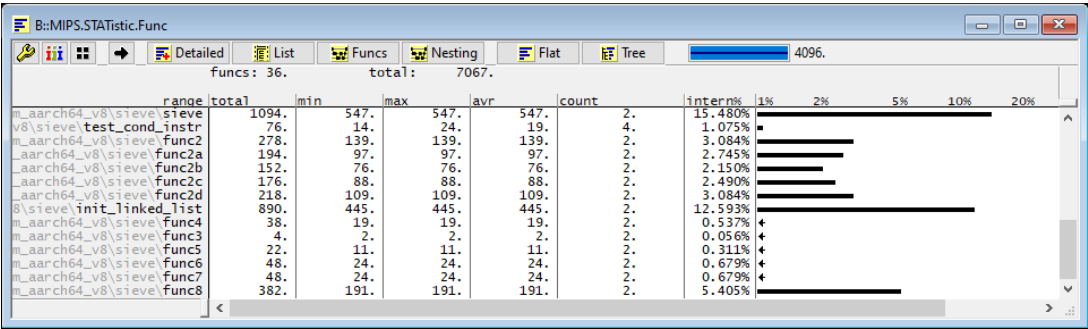
statistically analyzed from trace data using commands such as **Trace.STATistic.CYcle**, which provides average clocks per instruction calculated from traced instructions and cycles.

Trace.STATistic.CYcle



The MIPS.* commands can additionally be used to analyze the **MIPS**. The system can be analyzed under different aspects: workload per task, workload per high-level language line, workload per specified functional group etc. The following command displays for instance a numerical analysis per function.

MIPS.STATistic.Func



Refer for more information about the **MIPS.*** command group to the [General Commands Reference Guide M](#).

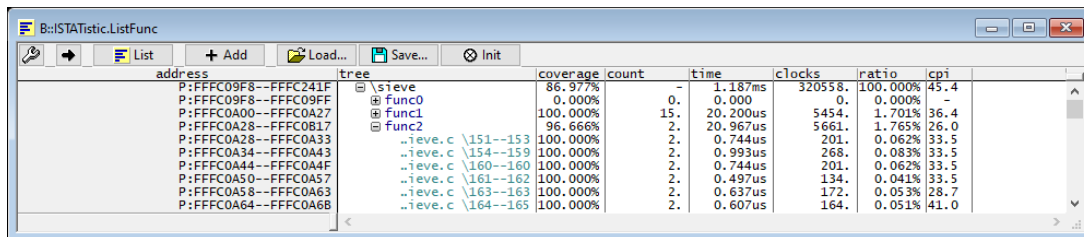
Moreover, you can use the `ISTATistic.*` command group to obtain detailed information for individual instructions when a program trace is available. However, it should be noted that for “high-end” cores this method may not provide cycle-accurate results for every single instruction.

Example:

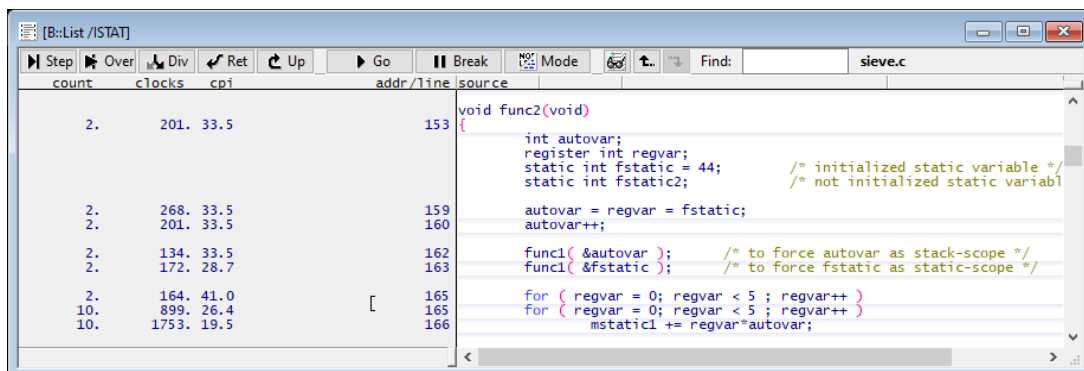
`ISTATistic.ADD`

`ISTATistic.ListFunc`

`List /ISTAT`



| address | tree | coverage | count | time | clocks | ratio | cpi |
|----------------------|---------------------|----------|-------|----------|---------|----------|------|
| P:FFFC09F8--FFFC241F | \sieve | 86.977% | - | 1.187ms | 320558. | 100.000% | 45.4 |
| P:FFFC09F8--FFFC09FF | func0 | 0.000% | 0. | 0.000 | 0. | 0.000% | - |
| P:FFFC0A00--FFFC0A27 | func1 | 100.000% | 15. | 20.200us | 5454. | 1.701% | 36.4 |
| P:FFFC0A28--FFFC0B17 | func2 | 96.666% | 2. | 20.967us | 5661. | 1.765% | 26.0 |
| P:FFFC0A28--FFFC0A33 | ..sieve.c \151--153 | 100.000% | 2. | 0.744us | 201. | 0.062% | 33.5 |
| P:FFFC0A34--FFFC0A43 | ..sieve.c \154--159 | 100.000% | 2. | 0.993us | 268. | 0.083% | 33.5 |
| P:FFFC0A44--FFFC0A4F | ..sieve.c \160--160 | 100.000% | 2. | 0.744us | 201. | 0.062% | 33.5 |
| P:FFFC0A50--FFFC0A57 | ..sieve.c \161--162 | 100.000% | 2. | 0.497us | 134. | 0.041% | 33.5 |
| P:FFFC0A58--FFFC0A63 | ..sieve.c \163--163 | 100.000% | 2. | 0.637us | 172. | 0.053% | 28.7 |
| P:FFFC0A64--FFFC0A6B | ..sieve.c \164--165 | 100.000% | 2. | 0.607us | 164. | 0.051% | 41.0 |



| count | clocks | cpi | addr/line | source |
|-------|--------|------|-----------|--|
| 2. | 201. | 33.5 | 153 | void func2(void) |
| | | | | { |
| | | | | int autovar; |
| | | | | register int regvar; |
| | | | | static int fstatic = 44; /* initialized static variable */ |
| | | | | static int fstatic2; /* not initialized static variable */ |
| | | | | autovar = regvar = fstatic; |
| | | | | autovar++; |
| 2. | 268. | 33.5 | 159 | |
| 2. | 201. | 33.5 | 160 | |
| | | | | func1(&autovar); /* to force autovar as stack-scope */ |
| 2. | 134. | 33.5 | 162 | |
| 2. | 172. | 28.7 | 163 | func1(&fstatic); /* to force fstatic as static-scope */ |
| | | | | for (regvar = 0; regvar < 5 ; regvar++) |
| 2. | 164. | 41.0 | 165 | for (regvar = 0; regvar < 5 ; regvar++) |
| 10. | 899. | 26.4 | 165 | mstatic1 += regvar*autovar; |
| 10. | 1753. | 19.5 | 166 | |

Refer for more information about the `ISTATistic.*` command group to the [General Commands Reference Guide I](#).

Note

The trace results must not include errors, such as `FIFOFULLs` or `FLOWERRORs`, in order to provide reliable results. Refer for more information to [HARDERRORs](#), [FIFOFULLs](#) and [FLOWERRORs in the trace](#).