École Polytechnique Fédérale de Lausanne

Data-dependent timing of instructions on Apple's M1

by Ioan Alexandru Țifui

Master Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

PhD. Stud. Luca Di Bartolomeo
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

June 9, 2023

Abstract

In this thesis, we analyze the data-dependent timing of instructions on Apple's M1 Silicon processor. Our objective is to find instructions that may have different cycle latencies depending on their input operands. Previous research has been done on various ARM processors, but none targeted the M1. From Armv8.4 onwards, ARM offers support for enabling the DIT flag which ensures data-independent timing for a subset of instructions. In their official documentation, ARM claims that even without the DIT flag being set the timing of the instructions in question is very often not correlated to the values of the input registers. We aim to investigate and benchmark the behavior of those instructions when the DIT flag is not set, as well as of the instructions for which DIT does not offer any guarantees. We managed to identify SDIV and UDIV as having data-dependent latency and FDIV and FSQRT as having variable, but data-independent latency. Our findings highlight the risk that UDIV and SDIV possess when used in sensitive contexts such as cryptographic implementations, as they make the system prone to side-channel attacks.

Contents

Chapter 1

Introduction

Many cryptographic implementations rely on the assumption that the instructions they execute are not dependent on the values of the source registers. To meet this assumption and reduce the possibility of timing side-channel attacks, hardware vendors equipped their CPUs with features that ensure data-independent timing for a subset of instructions, such as Data Operand Independent Timing (DOIT) on Intel [9] and Data Independent Timing (DIT) on ARM [1]. This thesis analyzes the data-independent timing of Aarch64 instructions on Apple's M1 Silicon processor. We aim to find instructions that may take a different number of cycles to execute depending on the operands.

In the past, the data dependency of ARM32 instructions has been studied on different processors such as the Cortex-M3, the Cortex-M4 [2] and the Cortex-A7 [3]. While it is true that people measured the timing of instructions on M1 in an attempt to reverse-engineer the microarchitecture [10], no experiments addressing the data dependency timing have been conducted so far. In this thesis, we take into consideration the approaches used for the other CPUs, especially regarding the sampling methods, but the main challenge was coming up with a reliable setup to measure the timing. The latency of each instruction is heavily dependent on the microarchitecture, which provides different mechanisms to support out-of-order execution. As we try to avoid this kind of optimization at all costs during our experiment, we had to find different solutions to prevent it from interfering with our measurements.

Before dealing with the hurdles introduced by out-of-order execution, finding a method to get reliable information about the cycle duration of an instruction was difficult by itself. We took as inspiration the work of Dougall Johnson [10] who built a setup for measuring the timing of instructions on M1. His infrastructure was built for macOS and we had to adapt it for Asahi Linux. This proved to be more difficult than expected due to the lack of documentation from Apple and the unstable nature of Asahi.

Chapter 2

Background

As mentioned in the previous chapter, to get reliable results about the latency of instructions we have to take into consideration the microarchitectural details. The Apple M1 Silicon has 4 Firestorm cores optimized for performance and 4 Icestorm cores optimized for efficiency. We ran our experiments both on Icestorm and Firestorm cores as we expected particular differences in cycle latency. We used `taskset` to set the affinity of the measurement process to the target core. This also prevents the scheduling algorithm from switching the core while running, thus allowing for more reliable results.

The M1 is a superscalar processor, which means that there are multiple execution units that can execute similar instructions in parallel. For instance, when reverse-engineering the microarchitecture [10], the Asahi team discovered there are 6 Arithmetic Logic Units (ALU) on Firestorm and 3 such units on Icestorm. Along with other features such as Issue Queues and the Reorder Buffer, multiple execution units allow for out-of-order execution which brings massive improvements to the performance of the CPU. In our experiment, as we need to consecutively run the same instruction multiple times to filter the noise, we want to avoid these microarchitectural optimizations to get accurate results. To do so we need to create dependencies between consecutive instructions. Most superscalar processors implement register renaming, which is a process of mapping the same logical register tag to different physical registers. This way the effects of Write after Write (WAW) and Write After Read(WAR) dependencies are nullified. However, Read After Write (RAW) dependencies pose problems to out-of-order execution even after register renaming. Therefore, in our setup we need to create RAW dependencies, by ensuring that each instruction produces a register that would be consumed by the next one.

Apart from defeating the out-of-order execution, we had to find a way of communicating with the system registers to get accurate performance counters. As discussed in detail in the next chapter, we based our setup on the work of Dougall Johnson [10] who did something similar for macOS and we adapted it to Asahi Linux. The biggest challenge was accessing the system registers needed for measuring CPU cycles of instructions, as they cannot be accessed directly from the userspace, so we were required to write a custom kernel module. The job was made more difficult by the lack of official documentation from Apple. We had to rely on the reverse-engineering work of the Asahi team and use a trial-and-error approach in our implementation to make it work.

Our target instructions were data processing instructions operating on integers, floating points, and vector registers. We did not consider conditional instructions and branches as their timing is very dependent on how speculative execution is implemented, this topic being out of the scope of the thesis. We also did not consider instructions operating on memory such as loads and stores because of similar reasons, as the memory translation implementation determines the latency of such instructions. For parsing the instructions we made use of the collection of Aaron Graham [7] of all the Aarch64 instructions along with their encodings.

Chapter 3

Timing Measurement Setup

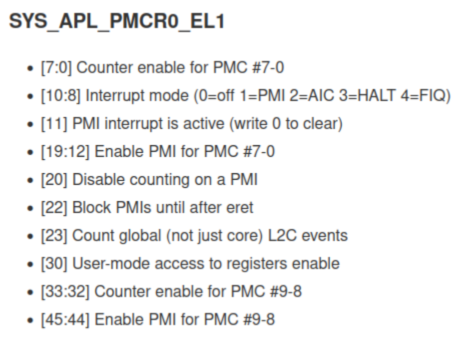
3.1 Performance monitoring

To measure the latency of instructions we used the Performance Monitoring Unit (PMU) [4] which is a piece of hardware that offers monitoring capabilities of various events related to performance. The results of performance monitoring can be read from dedicated system registers called Performance Monitor Counters (PMCs).

Researching the documentation of the Asahi Linux team on M1's system registers [5] we find that the PMC0 register is responsible for counting CPU cycles. We need to configure the Performance Monitor Control Register (PMCR), which dictates the behavior of the PMU, to enable the PMC0 register and set a Performance Monitoring Interrupt (PMI) for it. After that, we need to select the event that the PMU should monitor (in our case counting cycles) by writing a custom value in the Performance Monitors Event Select Register (PMESR). When such an event is triggered, the PMU will transfer the control flow to an Interrupt Service Routine (ISR) that will run the performance analysis.

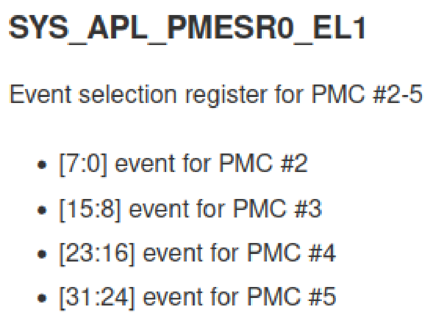
Figure 3.1 shows the bit encoding of the PMCR system register. We are most interested in the 30th bit which specifies whether or not we have access to the register from user mode. By default, this bit is set to 0, which means that we need to write a kernel module to handle read and write operations on this register. We took as inspiration the kernel module written by Dougall Johnson [10] for macOS and adapted it for Linux.

Johnson writes the value `0x3003400ff4ff` into PMCR0 to enable all PMC registers and their PMI and set the interrupt mode to Fast Interrupt Request (FIQ) which allows for fast and high-priority interrupt handling mechanism. Although we are not interested in all the PMC registers, we stuck to this value as more information about the execution of instructions proved to be helpful when debugging the sampling procedure. Next, we need to set the value of PMESR, which encoding is


SYS_APL_PMCR0_EL1

- [7:0] Counter enable for PMC #7-0
- [10:8] Interrupt mode (0=off 1=PMI 2=AIC 3=HALT 4=FIQ)
- [11] PMI interrupt is active (write 0 to clear)
- [19:12] Enable PMI for PMC #7-0
- [20] Disable counting on a PMI
- [22] Block PMIs until after eret
- [23] Count global (not just core) L2C events
- [30] User-mode access to registers enable
- [33:32] Counter enable for PMC #9-8
- [45:44] Enable PMI for PMC #9-8

Figure 3.1: PMCR bits [5]


SYS_APL_PMESR0_EL1

Event selection register for PMC #2-5

- [7:0] event for PMC #2
- [15:8] event for PMC #3
- [23:16] event for PMC #4
- [31:24] event for PMC #5

Figure 3.2: PMESR bits [5]

described in Figure 3.2. Again, Johnson provides an elegant script that writes the corresponding value to the register considering the desired metrics, but in our case `0x0202020202020202` will be enough to just account for the cycles.

3.2 Measurement script

Now that we presented how to use the PMU infrastructure we can discuss the actual approach taken for measuring the latency of a piece of code. We `mmap` a region in memory and make it both writable and executable (essentially we disable the Data Execution Prevention (DEP) protection). This is needed as we are going to write our target assembly into that memory region at runtime and also execute it to measure its latency.

We treat the memory region as a function, so we surround it with a standard prologue and epilogue to preserve the state of the registers after our measurement is done. Then we place any initialization code needed for ensuring custom values of the source registers. We then write `0x3003400ff4ff` into PMCR0 and call `isb` which flushes the pipeline. This is needed for setting the PMCR0, but as a side effect it removes any uncommitted instructions from the pipeline that could interfere with the code we want to measure. To eliminate the noise we unroll our code 100 times. To prevent any optimizations, we also loop back to the beginning of the unrolled code 100 times. Later we will have to divide our results by 10000 because of that. The branch we use for our loop will be taken care of by the Branch Target Buffer so it should not skew our results. Finally, we disable all the PMCs by writing `0x3000400ff403` into PMCR0.

As discussed in chapter 2, one challenge when running the timing measurements is avoiding out-of-order execution. The M1 has many computational units and can decide to execute instructions in parallel if there are no RAW dependencies among them. As we want to avoid such optimizations, a solution is to create RAW dependencies, by placing the destination register among the source registers. Our instructions will have the following form: `opcode rd, rd, rs1, rs2, rs3, ...`

where by `rd` we denote the destination register and by `rs_i` we denote the i^{th} source register. The only problem is that because we execute the instruction 10000 times, we are going to have a different value of the first source register at each iteration. To avoid this problem, we took a slightly different approach by first saving the result of the instruction into a different register, and then at the end of each iteration adding 2 EORs (exclusive or instruction on ARM) that would create a RAW dependency chain on one of the source registers, but at the same time would preserve its value. The measured code sequence will now look like this:

```
opcode rd, rs1, rs2, rs3, ...

EOR rs1, rs1, rd

EOR rs1, rs1, rD
```

Here `rd` and `rs_i` have the same meaning as above, and `rD` is the register that was used to compute the operation in the initialization code and should have the same value as `rd`. The two EOR instructions add an overhead of 2 cycles for integer registers and 4 cycles for floating-point and vector registers that we subtracted from the reported result. To avoid any errors we also measured the 2 EORs individually every time before measuring the target assembly and we always got consistent results.

Chapter 4

Sampling Values

We divided our inputs into 5 classes, considering as inspiration the sampling method used in [2]. Additionally, we considered 3 individual values (-1, 0, and 1) which are values frequently used. In the end, the 8 classes of input are:

1. minus one (`0xFF` on each byte of the register)
2. zero
3. one
4. small number ($\leq 2^{register\,bit\,length/2}$)
5. big number ($> 2^{register\,bit\,length/2}$)
6. small power of 2 ($\leq 2^{register\,bit\,length/2}$)
7. big power of 2 ($> 2^{register\,bit\,length/2}$)
8. top heavy (has bits set only on the upper half)

For each source register of each instruction, we randomly sampled inputs from all 8 classes. We also shuffled the register tags of the source registers and used custom values such as `xrz`, `wrz`, `x30`, and `w30`. The last two refer to the Link Register (LR), which is used for storing the return address before entering a function.

Because the zero register always reads zero, and the writes to it are ignored, we got very different timings when using it as an operand for almost all instructions. This could not be considered a true data dependency as the zero register represents an actual different register in the register file, so we decided to ignore it. We also could not find any correlation between the source register tags or their order, and the recorded latency of instructions.

Each register has an associated bit length which can be 32, or 64 in the case of integer and regular floating-point instructions and 8, 16, 32, 64, or 128 in the case of vector and advanced floating-point instructions. For each register, we sampled values that would fit on all the possible bit lengths less or equal to the register's bit length, starting with 32. For instance, for `V0` we sampled values of 32, 64, and 128 bits. On average, we used approximately 2 different bit lengths for each register.

Therefore, for each instruction, we considered on average $(2 * 8)^a$ different test cases, where a represents the arity of the instruction. Of course, this is just an approximation, as there are instructions that take as arguments registers of different bit lengths (ex: UMADDL, UMSUBL) and also instructions that allow for a single specific bit length combination for their operands (ex: cryptographic instructions such as SHA1C, SHA1P, SHA1M).

Chapter 5

Results

For most of the instructions, we did not find any data-dependent latency. For FDIV we discovered that the instruction takes 8 cycles when the operands are 32-bit registers and 10 cycles when the operands are 64-bit registers. Similarly, FSQRT takes 10 cycles when the source register is 32-bit wide and 13 cycles in the 64-bit case. The same results are reported on both Firestorm and Icestorm. However, these results cannot be interpreted as data dependencies, as the 32-bit registers involve working with single precision, whereas 64-bit registers involve double precision. We expect more precise computation to be more time expensive, but even if that would not be the case, we can make an argument for considering single-precision division and double-precision division as two different instructions. The same is true for computing the square root.

We managed to find data-dependent timings for unsigned and signed integer division instruction, namely UDIV, and SDIV. Our results are displayed in the tables from Appendix A. The recorded latencies of the two instructions are more consistent on Firestorm, where they can take 7, 8, or 9 cycles. On Icestorm the values of the recorded cycles vary from 7 to 21 for both UDIV and SDIV.

A common feature of both Firestorm and Icestorm is that the divisions take 7 cycles when the denominator is 0 or 1, regardless of the bit size of the registers. Another observation is that the difference between the smallest latency recorded (which is always 7 cycles) and the largest one is bigger when working with 64-bit registers. For Icestorm this difference is more than double in the 64-bit case (where the largest latency is 21 cycles and the difference is 14 cycles) than in the 32-bit case (where the largest latency is 13 cycles and the difference is 6 cycles). Also, on Icestorm the value of 8 cycles seems to be the rule, as the only case in which a division takes 7 cycles is when the denominator is either 0 or 1.

UDIV and SDIV are susceptible to side-channel attacks. As a consequence of the Icestorm CPU cores being optimized for efficient energy consumption and not for performance, the set of possible values for divisions' latency is much larger than in the case of Firestorm, which makes an attack easier to perform and increases its success probability. However, this does not imply that Firestorm is not prone to side-channel attacks, as even the slightest correlation between data and instruction latency can prove to be enough for a successful attack.

Chapter 6

Conclusions

The data-dependent timing of integer division instructions seems to be a known issue. UDIV and SDIV are not present on the list of instructions for which the DIT flag ensures data-independent timing. Neither are their x86 correspondents on Intel's DOIT list. Intel acknowledged [6] that the use of div instructions might lead to side-channel issues. In [8] there are several mitigations proposed to combat the potential side channel resulting from the variable latency of division instructions on Intel. Among them, we can find: extending the ISA to enable and disable the early exit of div instructions, implementing the div algorithm in software using instructions that take constant time, and forcing the CPU to execute the division in parallel with another division which is guaranteed to take the longest amount of cycles. A successful timing side-channel attack based on an ARM architecture (ARM Cortex-M4F) has been implemented in [11]. The root cause of the attack was the variable latency of the DIV instruction used in an implementation of the Number Theoretic Transform (NTT) algorithm.

Our thesis confirms that even without setting the DIT flag, there is no correlation between the input registers and the timing of the vast majority of assembly instructions on M1. UDIV and SDIV are the only exceptions we found, and we should raise attention to the risk they possess when used in cryptographic implementations. Moreover, even if they are technically data-independent, we found that FDIV and FSQRT have variable latency depending on the bit length of the registers. However, this has limited security implications, as an attacker is usually assumed to know the precision at which floating-point computations are performed.

Chapter 7

Future Work

The software we built for M1 can easily be ported to other ARM CPUs with minimal modifications involving the interaction with the system registers responsible for performance monitoring. We can also adapt the sampling program to Aarch32, the other major ARM flavor. That being said, an interesting future research direction would be to perform the same experiments on other ARM processors, prior to Armv8.4 where support for the DIT flag was introduced. ARM-based embedded systems are notorious for their energy efficiency. Due to our findings about the latency of integer division instruction on Icestorm, a unit also optimized for power consumption, we have good reasons to believe that such processors might have instructions that present some sort of data-dependent timing.

Our program for register value sampling can be extended to other RISC ISAs as well. As RISC-V is an architecture that gains popularity, especially in the case of embedded systems, we can use a similar approach to investigate potential side-channel attacks in RISC-V CPUs caused by the data-dependent latency of instructions. Previously, RISC-V introduced constant timing for multiplication instructions, as they are used frequently in cryptographic implementations [12]. However, previous research on ARM architectures [2] [3] showed that other instructions than multiplications can have variable latency as well, so an exhaustive analysis of the whole instructions set would still be important.

Appendix A

Latencies recorded for UDIV and SDIV

As described in chapter 4, we divided the numerator and denominator into 8 different classes, such that "small" and "big" refer to values smaller, and greater than $2^{register\,bit\,length/2}$ respectively, "small-pow-2" and "big-pow-2" refer to powers of 2 smaller, and greater than $2^{register\,bit\,length/2}$ respectively, and "topheavy" refers to values that have bits set only in the upper half.

denominator / numerator	zero	one	minus one	small	big	small pow 2	big pow 2	topheavy
zero	7	7	7	7	7	7	7	7
one	7	7	7	7	7	7	7	7
minus-one	7	7	8	8	8	7	7	7
small	7	7	7	8	7	7	7	7
big	7	7	7	8	7	7	7	7
small-pow-2	7	7	7	7	7	7	7	7
big-pow-2	7	7	7	8	7	7	7	7
topheavy	7	7	7	7	7	7	7	7

Table A.1: Latency of UDIV in CPU cycles on Firestorm for 32-bit registers

denominator / numerator	zero	one	minus one	small	big	small pow 2	big pow 2	topheavy
zero	7	7	7	7	7	7	7	7
one	7	7	7	7	7	7	7	7
minus-one	7	7	9	9	9	7	7	7
small	7	7	7	8	7	7	7	7
big	7	7	7	9	9	7	7	7
small-pow-2	7	7	7	7	7	7	7	7
big-pow-2	7	7	7	9	7	7	7	7
topheavy	7	7	7	7	7	7	7	7

Table A.2: Latency of UDIV in CPU cycles on Firestorm for 64-bit registers

denominator / numerator	zero	one	minus one	small	big	small pow 2	big pow 2	topheavy
zero	7	7	7	7	7	7	7	7
one	7	7	7	7	7	7	7	7
minus-one	7	7	7	7	7	7	7	7
small	7	7	7	7	7	7	7	7
big	7	7	7	8	7	7	7	7
small-pow-2	7	7	7	7	7	7	7	7
big-pow-2	7	7	7	8	7	7	7	7
topheavy	7	7	7	7	7	7	7	7

Table A.3: Latency of SDIV in CPU cycles on Firestorm for 32-bit registers

denominator / numerator	zero	one	minus one	small	big	small pow 2	big pow 2	topheavy
zero	7	7	7	7	7	7	7	7
one	7	7	7	7	7	7	7	7
minus-one	7	7	7	7	7	7	7	7
small	7	7	7	8	7	7	7	7
big	7	7	7	9	9	7	7	7
small-pow-2	7	7	7	7	7	7	7	7
big-pow-2	7	7	7	9	7	7	7	7
topheavy	7	7	7	7	7	7	7	7

Table A.4: Latency of SDIV in CPU cycles on Firestorm for 64-bit registers

denominator / numerator	zero	one	minus one	small	big	small pow 2	big pow 2	topheavy
zero	7	7	8	8	8	8	8	8
one	7	7	8	8	8	8	8	8
minus-one	7	7	8	10	8	11	8	13
small	7	7	8	8	8	8	8	8
big	7	7	8	10	8	10	8	12
small-pow-2	7	7	8	8	8	8	8	8
big-pow-2	7	7	8	11	8	10	8	10
topheavy	7	7	8	8	8	8	8	8

Table A.5: Latency of UDIV in CPU cycles on Icestorm for 32-bit registers

numerator \ denominator	zero	one	minus one	small	big	small pow 2	big pow 2	topheavy
zero	7	7	8	8	8	8	8	8
one	7	7	8	8	8	8	8	8
minus-one	7	7	8	14	8	21	13	20
small	7	7	8	8	8	8	8	12
big	7	7	8	14	8	18	12	20
small-pow-2	7	7	8	8	8	8	8	9
big-pow-2	7	7	8	8	8	10	8	10
topheavy	7	7	8	8	8	8	8	8

Table A.6: Latency of UDIV in CPU cycles on Icestorm for 64-bit registers

numerator \ denominator	zero	one	minus one	small	big	small pow 2	big pow 2	topheavy
zero	7	7	8	8	8	8	8	8
one	7	7	8	8	8	8	8	8
minus-one	7	7	8	8	8	8	8	8
small	7	7	10	8	8	8	8	9
big	7	7	13	9	8	11	8	12
small-pow-2	7	7	9	8	8	8	8	8
big-pow-2	7	7	10	10	8	10	8	10
topheavy	7	7	8	8	8	8	8	8

Table A.7: Latency of SDIV in CPU cycles on Icestorm for 32-bit registers

denominator / numerator	zero	one	minus one	small	big	small pow 2	big pow 2	topheavy
zero	7	7	8	8	8	8	8	8
one	7	7	8	8	8	8	8	8
minus-one	7	7	8	8	8	8	8	8
small	7	7	14	8	8	8	8	12
big	7	7	21	14	8	14	9	20
small-pow-2	7	7	8	8	8	8	8	8
big-pow-2	7	7	10	8	8	10	8	10
topheavy	7	7	8	8	8	8	8	8

Table A.8: Latency of SDIV in CPU cycles on Icestorm for 64-bit registers

Bibliography

[1] URL: https://developer.arm.com/documentation/ddi0601/2020-12/AArch64-Registers/DIT--Data-Independent-Timing.

[2] URL: https://pure.tue.nl/ws/portalfiles/portal/47038543.

[3] URL: https://fmt.ewi.utwente.nl/media/258_attachment_1.pdf.

[4] URL: https://developer.arm.com/documentation/ka004659/latest/.

[5] AsahiLinux. *HW:arm system registers.* URL: https://github.com/AsahiLinux/docs/wiki/HW%3AARM-System-Registers.

[6] By. *Configuring workloads for microarchitectural and side Channel Security.* URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/securing-workloads-against-side-channel-methods.html.

[7] CAS-Atlantic. *Aarch64-encoding/aarch64*$_o ps.CSV at master ůCas-Atlantic/AARCH64-encoding$. URL: https://github.com/CAS-Atlantic/AArch64-Encoding/blob/master/AArch64_ops.csv.

[8] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. "Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors". In: *2009 30th IEEE Symposium on Security and Privacy.* 2009, pp. 45–60. DOI: 10.1109/SP.2009.19.

[9] *Data Operand Independent Timing ISA guidance.* URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html.

[10] Dougallj. *Dougallj/applecpu at 2db5cd3b7abdb2227cfee26fa064649e4c61492b.* URL: https://github.com/dougallj/applecpu/tree/2db5cd3b7abdb2227cfee26fa064649e4c61492b.

[11] Robert Primas, Peter Pessl, and Stefan Mangard. "Single-Trace Side-Channel Attacks on Masked Lattice-Based Encryption". In: *Cryptographic Hardware and Embedded Systems – CHES 2017.* Ed. by Wieland Fischer and Naofumi Homma. Cham: Springer International Publishing, 2017, pp. 513–533. ISBN: 978-3-319-66787-4.

[12] Riscv. *Constant time requirements in the specification. · issue 64 · RISCV/RISCV-crypto.* URL: https://github.com/riscv/riscv-crypto/issues/64.