TIP 0004: Drop U32 Table

TIP	0004
authors:	Alan Szepieniec and Ferdinand Sauer
title:	Drop U32 Table
status:	superseded
created:	2022-09-24
issue tracker:	https://github.com/TritonVM/triton-vm/issues/8
pdf:	tip-0004.pdf

Abstract. For the purpose of U32 operations, this note proposes an alternative to relying on a separate table with corresponding table relations. It describes how the `split` instruction is powerful enough to simulate all other U32 operations, thus giving rise to a simpler design.

Introduction

The current design achieves U32 operations by first delegating it to the u32 coprocessor and then proving that the evolution that occurs (non-deterministically) in the Processor Table also occurs (but deterministically) in the U32 Table. While this design enables proving the correct computation of U32 operations without blowing up the cycle count, the additional Table and Table Relations constitute a formidable engineering task with equally formidable design complexity.

This note proposes an alternative way to prove the correct computation of U32 operations. It relies on the `split` instruction as the only operation not native to finite fields. To verify the bounded size of the resulting integers, their bit representations are nondeterministically guessed. All U32 operations can be simulated as intrinsics or pseudoinstuction whose expansion makes use of `split` and divining bits.

Split

The `split` instruction sends the opstack `_ a` to `_ lo hi` where `lo` and `hi` are both 32-bit integers and `a == (hi << 32) + lo`. This relation is enforced with the aid of two primitive arguments.

1. If `hi` is all ones, then `lo` must be zero. This constraint arises from the field whose "largest" element is `0xffffffff00000000`, and is enforced by the transition constraints $2^{32} \cdot \mathsf{st}_0^\star + \mathsf{st}_1^\star - \mathsf{st}_0$ and $((\mathsf{st}_0^\star - \texttt{0xffffffff}) \cdot \mathsf{hv}_0 - 1) \cdot \mathsf{st}_1^\star$. The helper variable hv_0 is the inverse-or-zero of $\mathsf{st}_0^\star - \texttt{0xffffffff}$.
2. Both `hi` and `lo` must be 32-bit integers. This fact is enforced by a table-lookup argument. Looking these values up in the U32 Op Table suffices. However, this note tries to eliminate the U32 Op Table and so must provide an alternative way to establish the values' bounded size.

Bounded Size

The following program verifies that the argument `a` consists of at most `t` bits. The arguments are `a` and `t`, both supplied on the stack. Furthermore, it consumes the argument.

instruction	stack	comment
`assert_bounded_size:`		`_ a t` –> `_` if `a` consists of no more than `t` bits
`dup0`	`_ a t`	
`push 0`	`_ a t t`	
`eq`	`_ a t t 0`	
`skiz`	`_ a t t==0`	
`call remove_divined_bit`	`_ a t`	jumps to (2)
`pop`	`_ a t`	(1)
`push 0`	`_ a`	
`eq`	`_ a 0`	
`assert`	`_ a==0`	crashes if `a != 0`
`return`	`_`	graceful return
`remove_divined_bit:`		(2)
`push -1`	`_ a t`	
`add`	`_ a t -1`	
`swap1`	`_ a t-1`	
`divine`	`_ t-1 a`	divine least significant bit of `a`
`dup0`	`_ t-1 a b`	
`dup0`	`_ t-1 a b b`	
`dup0`	`_ t-1 a b b b`	
`mul`	`_ t-1 a b b b b`	
`eq`	`_ t-1 a b b b*b`	
`assert`	`_ t-1 a b b==b*b`	crashes if `b` is not a bit
`push -1`	`_ t-1 a b`	
`mul`	`_ t-1 a b -1`	
`add`	`_ t-1 a -b`	
`push 2`	`_ t-1 a-b`	
`inv`	`_ t-1 a-b 2`	
`mul`	`_ t-1 a-b 2^-1`	
`swap1`	`_ t-1 a>>1`	
`dup0`	`_ a>>1 t-1`	
`push 0`	`_ a>>1 t-1 t-1`	
`eq`	`_ a>>1 t-1 t-1 0`	
`skiz`	`_ a>>1 t-1 t-1==0`	
`return`	`_ a>>1 t-1`	jumps to (1)
`recurse`	`_ a>>1 t-1`	jumps to (2)

Using this program as a subprocedure, it is possible to assert the bounded size of the two products of `split`. Specifically, the following pseudoinstruction is functionally equivalent to `split` but additionally asserts that the two returned integers are no more than 32 bits in size.

instruction	stack
`split`	`_ c`
`dup1`	`_ hi lo`
`push 32`	`_ hi lo hi`
`call assert_bounded_size`	`_ hi lo hi 32`
`dup 0`	`_ hi lo`
`push 32`	`_ hi lo lo`
`call assert_bounded_size`	`_ hi lo lo 32`

U32 Operations

The following pseudo-instructions simulate a selection of standard u32 operations using `split` as a building block. It is assumed that the produced integers' bounds are asserted.

`lte`

instruction	stack
`push -1`	`_ b a`
`mul`	`_ b a -1`
`add`	`_ b -a`
`split`	`_ (b-a)`
`push 0`	`_ lo hi`
`eq`	`_ lo hi 0`
`swap1`	`_ lo hi==0`
`pop`	`_ hi==0 lo`
`-`	`_ hi==0`

Where `hi * (1 << 32) + lo == BFieldElement(b-a)`.

`lt`

instruction	stack
`push 1`	`_ b a`
`add`	`_ b a 1`
`lte`	`_ b a+1`
`-`	`_ a+1<=b`

`div`

instruction	stack
`divine`	`_ d n`
`dup2`	`_ d n q`
`dup1`	`_ d n q d`
`mul`	`_ d n q d q`
`dup2`	`_ d n q d*q`
`swap1`	`_ d n q d*q n`
`push -1`	`_ d n q n d*q`
`mul`	`_ d n q n d*q -1`
`add`	`_ d n q n -d*q`
`dup3`	`_ d n q r`
`dup1`	`_ d n q r d`
`lt`	`_ d n q r d r`
`assert`	`_ d n q r r<d`
`swap2`	`_ d n q r`
`pop`	`_ d r q n`
`swap2`	`_ d r q`
`pop`	`_ q r d`
`-`	`_ q r`

Where `n = q*d + r` and `0 <= r < d`.

Conclusion

It is possible to simulate U32 operations without relying on a U32 Table, but only at the expense of a longer execution trace in the Processor Table. The `split` operation has a relatively straightforward arithmetization (as was already known) but the need to assert that the resulting integers have a bounded size generates a significant overhead – some 1200 cycles.

Given access to a black box `split` operation that also asserts bounds on the produced integers, standard u32 operations like comparison or division are relatively straightforward.

While the simpler design resulting from dropping one table and its table relation arguments is a worthwhile goal, this note raises the question what cost a simpler design is allowed to generate. Simultaneously, it suggests that the current instruction set might not be conducive to dropping the U32 Table and, accordingly, that a modification to it might be more appropriate.