TinyRAM Architecture Specification v2.000

Eli Ben-Sasson
StarkWare
eli@starkware.co

Alessandro Chiesa
UC Berkeley
alexch@berkeley.edu

Daniel Genkin
University of Michigan
genkin@umich.edu

Eran Tromer
Tel Aviv University and Columbia University
tromer@cs.tau.ac.il

Madars Virza
MIT
madars@mit.edu

SCIPR Lab
scipr-lab.org

March 30, 2020

Abstract

We describe the TinyRAM architecture, a simple RISC random-access machine with byte-addressable random-access memory and input tapes. TinyRAM comes in two variants: one following the Harvard architecture and one following the von Neumann architecture.

The Succinct Computational Integrity and Privacy Research (SCIPR) project constructs mechanisms for proving correct execution of TinyRAM programs, and TinyRAM is designed for efficiency in this setting.

In particular, TinyRAM strikes a balance between two opposing requirements:

(1) Sufficient expressibility to support short and efficient assembly code when compiling from high-level programming languages, and

(2) Small instruction set, with instructions that are simple to verify via arithmetic circuits, resulting in efficient verification using SCIPR's algorithmic and cryptographic mechanisms.

This specification greatly benefited from comments by former SCIPR Lab members:
Ohad Barta, Lior Greenblatt, Shaul Kfir, Michael Riabzev, Gil Timnat, and Arnon Yogev.

Contents

1 Introduction **3**

2 Architecture Overview **5**

3 Accepting Computations **7**

4 Instructions **8**

5 Assembly Language **13**

6 Preamble **14**

7 Binary Encoding Of Instructions **15**

References **17**

1 Introduction

The need to efficiently express *correctness of nondeterministic computations* arises in various applications that utilize proof systems for achieving certain security properties. For instance, this need arises in zero-knowledge proofs, probabilistically-checkable proofs, and others.

We describe the TinyRAM architecture, which is a random-access machine designed to be a convenient tool to efficiently express correctness of nondeterministic computations. TinyRAM is a reduced instruction set computer (RISC) with byte-addressable random-access memory; it comes in two variants: one following the Harvard architecture (where data and program lie in separate address spaces, and the latter is read-only), and one following the von Neumann one (data and program lie in the same read-write address space).

TinyRAM strikes a balance between two opposing goals:

- Having an architecture that is expressive enough to allow for short and efficient assembly code obtained by compiling programs written in high-level programming languages; and
- Having an architecture that is minimalistic enough to allow for efficient reductions from the correctness of program computations to arithmetic circuit satisfiability (and other algebraic constraint satisfaction problems).

TinyRAM was introduced by Ben-Sasson et al. [BCG$^+$13] in order to express correctness of nondeterministic computations in the setting of verifiable delegation of computations. For a discussion on the engineering choices and motivation behind the design of TinyRAM, see [BCG$^+$13]. Originally defined as a Harvard architecture (where the code is fixed), TinyRAM was extended in subsequent papers [BCTV14b, BCTV14a] into a von Neumann architecture (where the code is stored in RAM and can self-modify). In this document, we focus on a precise specification of TinyRAM.

Versioning. Version 2.000 of the TinyRAM specification adds the von Neumann architecture variant, switches to byte-level addressing, changes the binary opcode representation, and changes the preamble code. This specification is meant to be consistent with *libsnark* [SCI], but may have minor deviations from the aforementioned papers, e.g., in the preamble code and initial register state.

Illustrative application: succinctly verifying nondeterministic computations. We describe a simple example that motivates the need for expressing the correctness of nondeterministic computations in security applications.

Consider two parties, Alice and Bob, who respectively own inputs x and w. Alice wishes to learn the correct output of an algorithm A on input (x, w), but does not want to incur the cost of computing the algorithm's output $z := A(x, w)$.[1] Specifically, Alice is only willing to run in time that is proportional to the length of her own input (i.e., $|x|$) and the length of the output (i.e., $|z|$) but is not willing to run in time that is proportional to Bob's input (i.e., $|w|$) nor to the time needed to compute the algorithm's output z.

If Alice trusts Bob (and Bob is indeed honest), then Alice's efficiency requirement can be easily met as follows: Alice sends x to Bob, then Bob computes $z = A(x, w)$ and sends z to Alice. Alice thus learns z without incurring the cost of computing z.

But what if Alice does not trust Bob? Can the efficiency requirements still be met?

In such a case, Bob needs to *convince* Alice that his claimed computation's output $\tilde{z}$ does equal the correct output z. In other words, after learning x from Alice, Bob wants to convince Alice of the statement "there exists w such that $\tilde{z} = A(x, w)$". Using terminology from Theoretical Computer Science, such a statement is *nondeterministic* in the sense that Bob's input w is not fixed by the statement but rather is existentially quantified: Alice only cares that there exists some choice for Bob's input that correctly produces the claimed output $\tilde{z}$. (The input w is sometimes called a "witness" because it witnesses the fact that the output $\tilde{z}$ is a legitimate output of the computation.)

Crucially, Bob must use a *very efficient method* to convince Alice that the aforementioned nondeterministic statement holds, because Alice is not willing to compute z from scratch. (In particular, Bob cannot simply send w to Alice as "proof" that $\tilde{z} = A(x, w)$.) So Bob needs to use a cryptographic tool that is known as a *proof system with succinct verification*, which is a proof system that enables one party (the *prover*) to convince another one (the *verifier*) of the truth of a nondeterministic statement, while requiring the other party to invest resources that are proportional only to the nondeterministic statement's size.[2]

Being able to formally express nondeterministic statements is crucial for using in practice such a proof system, and TinyRAM can be used to efficiently express nondeterministic statements.

[1] An important special case of this setting is when w is the empty string; in such a case Alice wishes to enlist Bob's help in computing the output z of the algorithm A when given her input x.

[2] Note that, in the example mentioned above, the statement size is indeed proportional only to $|x|$ and $|z|$, as well as the size of the description of A, but is not proportional to $|w|$, or the time to compute z.

2 Architecture Overview

TinyRAM (version 2.000) is parametrized by two integers: the *word size*, denoted W and required to be a power of 2 and divisible by 8, and the *number of registers*, denoted K. When we wish to be precise, we explicitly denote this by using the notation $\mathsf{TinyRAM}_{W,K}$.

The *state* of the machine consists of the following.

- The *program counter*, denoted pc; it consists of W bits.
- K general-purpose *registers*, denoted $\mathtt{r}0, \mathtt{r}1, \ldots, \mathtt{r}(K-1)$; each register consists of W bits.
- The *(condition) flag*, denoted flag; it consists of a single bit.
- *Memory*, which is a linear array of 2^W bytes. When storing or loading blocks of multiple bytes, we use the little-endian convention (i.e., the least-significant byte is at the lowest address). We say that a block is *aligned* to the a-th byte if its least-significant byte is at address a.
- Two *tapes*, each containing a string of W-bit words; each tape is read-only in one direction. One tape is for a *primary input* x and the other tape is for an *auxiliary input* w. We treat the primary input as given, and the auxiliary input as nondeterministic advice. (See discussion about TinyRAM accepting computations in Section 3.)

At each step, the machine executes an *instruction*, which changes the machine's state. We specify the available instructions in Section 4; briefly, the instruction set of TinyRAM includes simple load and store instructions for accessing random-access memory (both as byte or word blocks), as well as simple integer, shift, logical, compare, move, and jump instructions. In particular, TinyRAM can efficiently implement control flow, loops, subroutines, recursion, and so on. Complex instructions, such as floating-point arithmetic, are not directly supported and can be implemented "in software".

TinyRAM has two variants, depending on where instructions to be executed reside.

Harvard architecture TinyRAM (hvTinyRAM). In hvTinyRAM, following the Harvard architecture paradigm, a *program* $\mathbf{P}$ is stored in a separate, read-only, address space (i.e., different from the read-write data address space) simply as a sequence of instructions. In such a case, the *initial state* of the machine is as follows: the contents of pc, all general-purpose registers, flag, and memory are all 0; the content of one tape defines the primary input and that of the other tape defines the auxiliary input. Then, at every time step, the machine fetches and executes the pc-th instruction of $\mathbf{P}$;[3] every instruction increments pc by 1 (unless it explicitly modifies pc). The machine's only input is via the two input tapes, and its only output is via an `answer` instruction (which also terminates execution) that has a single argument A, representing the return value. The return value $A = 0$ by default means "accept". (If pc is not an integer in $\{0, \ldots, L-1\}$, where L is the number of instructions in $\mathbf{P}$, then the instruction `answer` 1 is fetched as default.)

Von Neumann architecture TinyRAM (vnTinyRAM). In vnTinyRAM, following the von Neumann architecture paradigm, the program $\mathbf{P}$ is initially stored in the same, read-write, address space as data. In particular, memory is not initialized to "all zeros" but, rather, its initial contents are defined by $\mathbf{P}$ (and may, more generally, include both instructions and initial data). In memory, any TinyRAM instruction is represented via a double word (see Section 7 for the binary encoding).

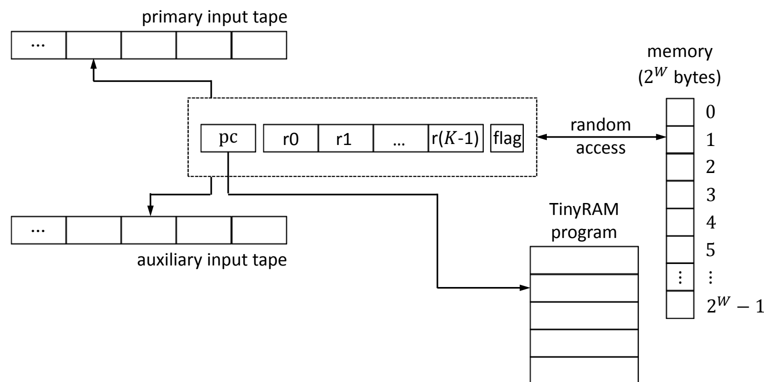
[3] We defined pc as a W-bit string. We use pc to also denote the corresponding integer between 0 and $2^W - 1$.

Thus, in vnTinyRAM, the *initial state* of the machine is as follows. The contents of all general-purpose registers and flag are all 0. The content of one tape defines the primary input and that of the other tape defines the auxiliary input. The initial contents of pc is 0, so that execution begins with the instruction encoded by the double word aligned to the 0-th byte in memory.

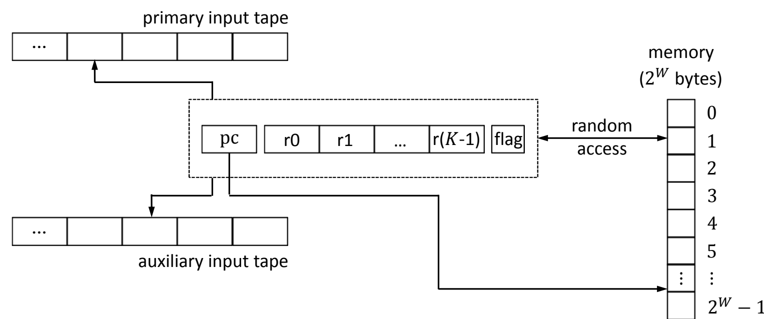
Then, at every time step, the machine fetches and executes the instruction encoded by the double word aligned to the $[\mathsf{pc}]_{2W}$-th byte in memory, where $[\mathsf{pc}]_{2W}$ denotes the unsigned integer given by pc rounded down to a multiple of $2W/8$ (i.e, setting its $\log_2 W - 2$ least-significant bits set to 0); every instruction increments pc by $2W/8$ (unless it explicitly modifies pc).

The machine's only input is via the two input tapes and initial memory, and (as in hvTinyRAM) its only output is via an `answer` instruction (which also terminates execution) that has a single argument A, representing the return value. Again, $A = 0$ by default means "accept".

See Figure 1 below for a diagram of the two variants of TinyRAM.



(a) A diagram of hvTinyRAM.



(b) A diagram of vnTinyRAM.

Figure 1: A diagram of the two variants of TinyRAM: the Harvard architecture variant (hvTinyRAM), and the von Neumann architecture variant (vnTinyRAM). In both diagrams, the word size is W and the number of registers is K. The main difference between the two diagrams is where the TinyRAM program resides: for hvTinyRAM, the program resides in a separate address space; while, for vnTinyRAM, the program resides in the same address space as data. (In particular, in the case of von Neumann case, a program may *modify* its own code.)

3 Accepting Computations

In Section 1 we stated that TinyRAM is designed to be a convenient tool for expressing the correctness of nondeterministic computations. We now formalize what are the "good" nondeterministic computations on a TinyRAM machine.

We first introduce the notion of acceptance for a TinyRAM computation: a computation on TinyRAM is *accepting* (i.e., "good") if execution halts with the instruction `answer` 0. More precisely, fix a word size W and number of registers K, let $\mathbf{P}$ be a $\mathsf{TinyRAM}_{W,K}$ program, and let x and w be strings of W-bit words. We say that $\mathbf{P}(x,w)$ **accepts in** T **steps** if $\mathbf{P}$, with x as primary input and w as auxiliary input, executes the instruction `answer` 0 in step T.[4]

The **set of accepting computations** is $\mathcal{L} = \cup_{W,K} \mathcal{L}_{W,K}$, where $\mathcal{L}_{W,K}$ is the set of triples $(\mathbf{P}, x, T)$ where $\mathbf{P}$ is a $\mathsf{TinyRAM}_{W,K}$ program, x is a string of W-bit words, and T is a time bound, such that there exists a string w of W-bit words for which $\mathbf{P}(x,w)$ accepts in T steps.[5]

Now, given a nondeterministic computation, it is straightforward to encode it into a triple $(\mathbf{P}, x, T)$ such that $(\mathbf{P}, x, T) \in \mathcal{L}$ if and only if the nondeterministic computation is correct. For instance, consider the statement "there is some w such that $\tilde{z} = A(x,w)$" that we mentioned in Section 1; then consider the TinyRAM program $\mathbf{P}$ that works as follows: given as input $((A, x, \tilde{z}), w))$, compute $z = A(x,w)$, and halt with `answer` 0 if $\tilde{z} = z$ and otherwise halt with `answer` 1. Clearly, if one is convinced that $(\mathbf{P}, (A, x, \tilde{z}), T) \in \mathcal{L}$, for some T, then one is also convinced that $A(x,w)$ outputs $\tilde{z}$ within T steps for some choice of w.[6]

Thus, the above is the precise sense in which we mean that TinyRAM is a convenient tool for expressing the correctness of nondeterministic computations.

---

[4]See more details about the `answer` instruction in Section 4.

[5]We thus model all nondeterministic choices via a witness that is given as auxiliary input to the machine. In particular, reading the next W-bit word from the auxiliary input is a "nondeterministic instruction".

[6]Using time bounds is necessary for avoiding running into problems with the undecidability of the halting problem.

4 Instructions

The instruction set of TinyRAM consists of 29 instructions. Each instruction is specified via an *opcode* and up to three *operands*. An operand can be a *register name* (i.e., an integer between 0 and $K-1$) or an *immediate value* (i.e., a W-bit string). Unless stated otherwise, every instruction does not modify flag and increments pc (the program counter) by i (modulo 2^W), where $i = 1$ for hvTinyRAM and $i = 2W/8$ for vnTinyRAM. Generally, the first operand is the destination register of the computation performed by the instruction, and the other operands (if any) specify arguments to the instruction. Finally, all instructions take one cycle of the machine to execute.

We now proceed to describe the instruction set of TinyRAM. The instructions are summarized in Table 1 below; following the table we describe each instruction in more detail.

Notations. In the following, i and j are integers in $\{0, \ldots, K-1\}$; $\mathtt{r}i$ is the i-th register, and $[\mathtt{r}i]$ the W-bit string currently stored in it; similarly for $\mathtt{r}j$ and $[\mathtt{r}j]$. Also, A denotes either an immediate value or a register name; $[A]$ denotes its value (i.e., the immediate value itself or the W-bit string currently stored in the register). At times we also need to consider unsigned and signed integers represented by a W-bit string. With this in mind, we denote by $[\mathtt{r}i]_{\mathrm{u}}$ the *unsigned* integer encoded by the contents of $\mathtt{r}i$ (i.e., $\sum_{k=0}^{W-1} a_i 2^i$ if $\mathtt{r}i$ stores the W-bit string $a_{W-1} \cdots a_0$) and by $[\mathtt{r}i]_{\mathrm{s}}$ the *signed* integer encoded by the contents of $\mathtt{r}i$ (i.e., using two's complement, $-a_{W-1}2^{W-1} + \sum_{i=0}^{W-2} a_i 2^i$ if $\mathtt{r}i$ stores the W-bit string $a_{W-1} \cdots a_0$). The notations $[\mathtt{r}j]_{\mathrm{u}}, [\mathtt{r}j]_{\mathrm{s}}$ and $[A]_{\mathrm{u}}, [A]_{\mathrm{s}}$ are similarly defined. When these notations are used, arithmetic is performed over the integers. We also need notation for values addressing aligned words in memory (which is always addressed in terms of bytes): we thus denote by $[\mathtt{r}i]_{\mathrm{w}}$ the unsigned integer, with value in the set $\{0, \ldots, 2^W - 1\}$, obtained by rounding $[\mathtt{r}i]_{\mathrm{u}}$ down to a multiple of $W/8$ (equivalently, the integer whose binary representation is $[\mathtt{r}i]$ with the $\log_2 W - 3$ least-significant bits set to 0). Finally, we use MSB and LSB to respectively denote the most-significant (left-most) and least-significant (right-most) bit of a binary string.

instruction mnemonic	operands	effects	flag	notes
`and`	ri rj A	compute bitwise AND of [rj] and $[A]$ and store result in ri	result is 0^W	
`or`	ri rj A	compute bitwise OR of [rj] and $[A]$ and store result in ri	result is 0^W	
`xor`	ri rj A	compute bitwise XOR of [rj] and $[A]$ and store result in ri	result is 0^W	
`not`	ri A	compute bitwise NOT of $[A]$ and store result in ri	result is 0^W	
`add`	ri rj A	compute $[\mathtt{r}j]_\mathrm{u} + [A]_\mathrm{u}$ and store result in ri	overflow	
`sub`	ri rj A	compute $[\mathtt{r}j]_\mathrm{u} - [A]_\mathrm{u}$ and store result in ri	borrow	
`mull`	ri rj A	compute $[\mathtt{r}j]_\mathrm{u} \times [A]_\mathrm{u}$ and store least significant bits of result in ri	overflow	
`umulh`	ri rj A	compute $[\mathtt{r}j]_\mathrm{u} \times [A]_\mathrm{u}$ and store most significant bits of result in ri	overflow	
`smulh`	ri rj A	compute $[\mathtt{r}j]_\mathrm{s} \times [A]_\mathrm{s}$ and store most significant bits of result in ri	over/underflow	
`udiv`	ri rj A	compute quotient of $[\mathtt{r}j]_\mathrm{u}/[A]_\mathrm{u}$ and store result in ri	$[A]_\mathrm{u} = 0$	
`umod`	ri rj A	compute remainder of $[\mathtt{r}j]_\mathrm{u}/[A]_\mathrm{u}$ and store result in ri	$[A]_\mathrm{u} = 0$	
`shl`	ri rj A	shift [rj] by $[A]_\mathrm{u}$ bits to the left and store result in ri	MSB of [rj]	
`shr`	ri rj A	shift [rj] by $[A]_\mathrm{u}$ bits to the right and store result in ri	LSB of [rj]	
`cmpe`	ri A	none ("compare equal")	$[\mathtt{r}i] = [A]$	
`cmpa`	ri A	none ("compare above", unsigned)	$[\mathtt{r}i]_\mathrm{u} > [A]_\mathrm{u}$	
`cmpae`	ri A	none ("compare above or equal", unsigned)	$[\mathtt{r}i]_\mathrm{u} \geq [A]_\mathrm{u}$	
`cmpg`	ri A	none ("compare greater", signed)	$[\mathtt{r}i]_\mathrm{s} > [A]_\mathrm{s}$	
`cmpge`	ri A	none ("compare greater or equal", signed)	$[\mathtt{r}i]_\mathrm{s} \geq [A]_\mathrm{s}$	
`mov`	ri A	store $[A]$ in ri		
`cmov`	ri A	if $\mathsf{flag} = 1$, store $[A]$ in ri		
`jmp`	A	set pc to $[A]$		
`cjmp`	A	if $\mathsf{flag} = 1$, set pc to $[A]$ (else increment pc as usual)		
`cnjmp`	A	if $\mathsf{flag} = 0$, set pc to $[A]$ (else increment pc as usual)		
`store.b`	A ri	store the least-significant byte of [ri] at the $[A]_\mathrm{u}$-th byte in memory		
`load.b`	ri A	store into ri (with zero-padding in front) the $[A]_\mathrm{u}$-th byte in memory		
`store.w`	A ri	store [ri] at the word in memory that is aligned to the $[A]_\mathrm{w}$-th byte		
`load.w`	ri A	store into ri the word in memory that is aligned to the $[A]_\mathrm{w}$-th byte		
`read`	ri A	if the $[A]_\mathrm{u}$-th tape has remaining words then consume the next word, store it in ri, and set $\mathsf{flag} = 0$; otherwise store 0^W in ri and set $\mathsf{flag} = 1$	$\longleftarrow$	(1)
`answer`	A	stall or halt (and the return value is $[A]_\mathrm{u}$)		(2)

(1) All but the first two tapes are empty: if $[A]_\mathrm{u} \notin \{0,1\}$ then store 0^W in ri and set $\mathsf{flag} = 1$.
(2) `answer` causes a stall (i.e., not increment pc) or a halt (i.e., the computation stops); the choice between the two is undefined.

Table 1: Summary of the TinyRAM instruction set. Where "flag" is specified, flag is set to 1 if the predicate holds and to 0 otherwise. Below, we describe each instruction in more detail.

Bit operations. These are standard bit operations on registers.

`and`: The instruction `and` $\mathtt{r}i$ $\mathtt{r}j$ A stores in $\mathtt{r}i$ the bitwise AND of $[\mathtt{r}j]$ and $[A]$.
Moreover, flag is set to 1 if the result is 0^W and to 0 otherwise.

`or`: The instruction `or` $\mathtt{r}i$ $\mathtt{r}j$ A stores in $\mathtt{r}i$ the bitwise OR of $[\mathtt{r}j]$ and $[A]$.
Moreover, flag is set to 1 if the result is 0^W and to 0 otherwise.

`xor`: The instruction `xor` $\mathtt{r}i$ $\mathtt{r}j$ A stores in $\mathtt{r}i$ the bitwise XOR of $[\mathtt{r}j]$ and $[A]$.
Moreover, flag is set to 1 if the result is 0^W and to 0 otherwise.

`not`: The instruction `not` $\mathtt{r}i$ A stores in $\mathtt{r}i$ the bitwise NOT of $[A]$.
Moreover, flag is set to 1 if the result is 0^W and to 0 otherwise.

Integer operations. These are various unsigned and signed integer operations. In each case, the condition flag is set to 1 if an arithmetic overflow or an error (such as divide by zero) occurs, and is set to 0 otherwise. (Below, we shall specify, for each operation, the predicate that sets the flag.)

In the sequel, U_W is the set of integers $\{0, \ldots, 2^W - 1\}$; these are the 2^W integers that can be encoded by W-bit strings. Similarly, S_W is the set of integers $\{-2^{W-1}, \ldots, 0, 1, \ldots, 2^{W-1} - 1\}$; these are the 2^W integers that can be encoded, via two's complement, by W-bit strings.

`add`: The instruction `add` $\mathtt{r}i$ $\mathtt{r}j$ A stores in $\mathtt{r}i$ the W-bit string $a_{W-1} \cdots a_0$ obtained as follows: $a_{W-1} \cdots a_0$ are the W least significant bits of $G = [\mathtt{r}j]_\mathrm{u} + [A]_\mathrm{u}$.
Moreover, flag is set to G_W, where G_W is the MSB of G.

`sub`: The instruction `sub` $\mathtt{r}i$ $\mathtt{r}j$ A stores in $\mathtt{r}i$ the W-bit string $a_{W-1} \cdots a_0$ obtained as follows: $a_{W-1} \cdots a_0$ are the W least significant bits of $G = [\mathtt{r}j]_\mathrm{u} + 2^W - [A]_\mathrm{u}$.
Moreover, flag is set to $1 - G_W$, where G_W is the MSB of G.

`mull`: The instruction `mull` $\mathtt{r}i$ $\mathtt{r}j$ A stores in $\mathtt{r}i$ the W-bit string $a_{W-1} \cdots a_0$ obtained as follows: $a_{W-1} \cdots a_0$ are the W least significant bits of $[\mathtt{r}j]_\mathrm{u} \times [A]_\mathrm{u}$.
Moreover, flag is set to 1 if $[\mathtt{r}j]_\mathrm{u} \times [A]_\mathrm{u} \notin U_W$ and to 0 otherwise.[7]

`umulh`: The instruction `umulh` $\mathtt{r}i$ $\mathtt{r}j$ A stores in $\mathtt{r}i$ the W-bit string $a_{W-1} \cdots a_0$ obtained as follows: $a_{W-1} \cdots a_0$ are the W most significant bits of $[\mathtt{r}j]_\mathrm{u} \times [A]_\mathrm{u}$.
Moreover, flag is set to 1 if $[\mathtt{r}j]_\mathrm{u} \times [A]_\mathrm{u} \notin U_W$ and to 0 otherwise.

`smulh`: The instruction `smulh` $\mathtt{r}i$ $\mathtt{r}j$ A stores in $\mathtt{r}i$ the W-bit string $a_{W-1} \cdots a_0$ obtained as follows: a_{W-1} is the sign of $[\mathtt{r}j]_\mathrm{s} \times [A]_\mathrm{s}$ and $a_{W-2} \cdots a_0$ are the $W - 1$ most significant bits of the absolute value of $[\mathtt{r}j]_\mathrm{s} \times [A]_\mathrm{s}$.
Moreover, flag is set to 1 if $[\mathtt{r}j]_\mathrm{s} \times [A]_\mathrm{s} \notin S_W$ and to 0 otherwise.

`udiv`: The instruction `udiv` $\mathtt{r}i$ $\mathtt{r}j$ A stores in $\mathtt{r}i$ the W-bit string $a_{W-1} \cdots a_0$ obtained as follows.
If $[A]_\mathrm{u} = 0$, then $a_{W-1} \cdots a_0 = 0^W$.
If $[A]_\mathrm{u} \neq 0$, then $a_{W-1} \cdots a_0$ is the binary encoding of Q where Q is the unique integer such that $[\mathtt{r}j]_\mathrm{u} = [A]_\mathrm{u} \times Q + R$ for some integer $R \in \{0, \ldots, [A]_\mathrm{u} - 1\}$.
Moreover, flag is set to 1 if and only if $[A]_\mathrm{u} = 0$.

[7] An equivalent definition of the `mull` instruction is the following: "The instruction `mull` $\mathtt{r}i$ $\mathtt{r}j$ A stores in $\mathtt{r}i$ the W-bit string $a_{W-1} \cdots a_0$ obtained as follows: a_{W-1} is the sign of $[\mathtt{r}j]_\mathrm{s} \times [A]_\mathrm{s}$ and $a_{W-2} \cdots a_0$ are the $W - 1$ least significant bits of the absolute value of $[\mathtt{r}j]_\mathrm{s} \times [A]_\mathrm{s}$. Moreover, flag is set to 1 if $[\mathtt{r}j]_\mathrm{u} \times [A]_\mathrm{u} \notin U_W$ and to 0 otherwise."

`umod`: The instruction `umod` ri rj A stores in ri the W-bit string $a_{W-1} \cdots a_0$ obtained as follows.
If $[A]_\mathrm{u} = 0$, then $a_{W-1} \cdots a_0 = 0^W$.
If $[A]_\mathrm{u} \neq 0$, then $a_{W-1} \cdots a_0$ is the binary encoding of R where R is the unique integer in $\{0, \ldots, [A]_\mathrm{u} - 1\}$ such that $[\mathtt{r}j]_\mathrm{u} = [A]_\mathrm{u} \times Q + R$ for some integer Q.
Moreover, flag is set to 1 if and only if $[A]_\mathrm{u} = 0$.

Shift operations. These are left and right (logical) shift operations.

`shl`: The instruction `shl` ri rj A stores in ri the W-bit string obtained by shifting $[\mathtt{r}j]$ by $[A]_\mathrm{u}$ bits to the left. The vacant positions (obtained after the shift) are filled with 0's.
Moreover, flag is set to the most significant bit of $[\mathtt{r}j]$.

`shr`: The instruction `shr` ri rj A stores in ri the W-bit string obtained by shifting $[\mathtt{r}j]$ by $[A]_\mathrm{u}$ bits to the right. The vacant positions (obtained after the shift) are filled with 0's.
Moreover, flag is set to the least significant bit of $[\mathtt{r}j]$.

Compare operations. These are various compare operations. Each of these instructions do not modify any registers; instead, the result of the comparison is stored in the condition flag.

`cmpe`: The instruction `cmpe` ri A sets flag to 1 if $[\mathtt{r}i] = [A]$ and to 0 otherwise.

`cmpa`: The instruction `cmpa` ri A sets flag to 1 if $[\mathtt{r}i]_\mathrm{u} > [A]_\mathrm{u}$ and to 0 otherwise.

`cmpae`: The instruction `cmpae` ri A sets flag to 1 if $[\mathtt{r}i]_\mathrm{u} \geq [A]_\mathrm{u}$ and to 0 otherwise.

`cmpg`: The instruction `cmpg` ri A sets flag to 1 if $[\mathtt{r}i]_\mathrm{s} > [A]_\mathrm{s}$ and to 0 otherwise.

`cmpge`: The instruction `cmpge` ri A sets flag to 1 if $[\mathtt{r}i]_\mathrm{s} \geq [A]_\mathrm{s}$ and to 0 otherwise.

Move operations. These are standard move and conditional move operations.

`mov`: The instruction `mov` ri A stores $[A]$ in ri.

`cmov`: The instruction `cmov` ri A stores $[A]$ in ri if flag $= 1$. (If flag $= 0$, ri is not changed.)

Jump operations. These are standard jump and conditional jump operations. Each of these instructions do not modify any registers or the condition flag, but only modify the program counter.

`jmp`: The instruction `jmp` A stores $[A]$ in pc.

`cjmp`: The instruction `cjmp` A stores $[A]$ in pc if flag $= 1$. (If flag $= 0$, pc is incremented as usual.)

`cnjmp`: The instruction `cnjmp` A stores $[A]$ in pc if flag $= 0$. (If flag $= 1$, pc is incremented as usual.)

Memory operations. These are simple load and store operations where the address in memory is determined either by an immediate value or the contents of a register. These are the *only* addressing modes in TinyRAM. (In particular, the common "base+offset" addressing mode is not supported.)

`store.b`: The instruction `store.b` A ri stores the least-significant byte of $[\mathtt{r}i]$ at the $[A]_\mathrm{u}$-th byte in memory.

load.b: The instruction `load.b` ri A stores into ri (with zero-padding in front) the $[A]_{\mathrm{u}}$-th byte in memory.

store.w: The instruction `store.w` A ri stores [ri] at the word in memory that is aligned to the $[A]_{\mathrm{w}}$-th byte.

load.w: The instruction `load.w` ri A stores into ri the word in memory that is aligned to the $[A]_{\mathrm{w}}$-th byte.

Input operation. This is the instruction to access contents to one of the two input tapes; the 0-th tape is used for the primary input and the 1-th tape is used for an auxiliary input.

read: The instruction `read` ri A stores in ri the next W-bit word on the $[A]_{\mathrm{u}}$-th tape, if any. More precisely, if the $[A]_{\mathrm{u}}$-th tape has remaining words then consume the next word, store it in ri, and set flag $= 0$; otherwise (if there are no remaining input words on the $[A]_{\mathrm{u}}$-th tape) store 0^W in ri and set flag $= 1$.
Because TinyRAM only has two input tapes, all but the first two tapes are assumed to be empty. Specifically, if $[A]_{\mathrm{u}}$ is not 0 or 1, then we store 0^W in ri and set flag $= 1$.

Answer operation. This instruction signifies that the program has finished the computation and thus no additional operations are allowed.

answer: The instruction `answer` A causes the machine to stall (i.e., not increment pc) or halt (i.e., the computation stops) with return value $[A]_{\mathrm{u}}$. The choice between stall or halt is undefined. A return value of 0 is used to indicate that the program accepted (see Section 3).

5 Assembly Language

A TinyRAM *program* **P** is written in the TinyRAM *assembly language*, which we now describe. (The syntax is inspired by the Intel x86 syntax.)

A TinyRAM *program* **P** is a text file consisting of a sequence of lines (separated by CR, LF or CR/LF). The text file is structured as follows. If a program is for hvTinyRAM, the first line contains the string

"`; TinyRAM V=2.000 M=hv W=`W `K=`K"

if instead the program is for vnTinyRAM, the first line contains the string

"`; TinyRAM V=2.000 M=vn W=`W `K=`K".

Above, W is the word size in decimal representation and K is the number of registers in decimal representation. Each subsequent line contains the following, in sequence:

1. Optional whitespace.
2. An optional *label* followed by "`:`". This defines the label as referring to the first instruction following it (if any).
   A label must match the regular expression "`_[0-9a-zA-Z_]+`". In particular, a label must start with an underscore (to distinguish the label from an immediate value and registers).
3. An optional instruction, consisting of an *instruction mnemonic*, followed by its *operands* (if any). The instruction mnemonic is separated from the first operand by whitespace; and subsequent operands are separated by a comma ("`,`") surrounded by optional whitespaces. Registers are specified as "`r`" followed by the register number in decimal, e.g., "`r0`", "`r12`". An immediate operand may be written as an integer in decimal representation, or as a label; an integer a (which may be negative) represents the W-bit word x such that $[x]_{\mathrm{u}} \equiv a \bmod 2^W$.
4. Optional whitespace.
5. An optional *comment* starting with a semicolon ("`;`") and lasting until the end of the line.

Every instruction has an implicit number. In hvTinyRAM, instructions are numbered sequentially starting with 0 (ignoring non-instruction lines), and there can be at most 2^W instructions. In vnTinyRAM, instructions are numbered sequentially, starting with 0, in steps of $W/4$ (so to represent properly aligned-addresses).

A label may be defined at most once. Labels given as operands must be defined, and are resolved to the number of the instruction following the label definition.

6 Preamble

In the context of succinctly verifying nondeterministic computations [BCGT13, BCG+13], we require TinyRAM programs to start with a specific preamble given below. This is is needed for technical reasons, to improve the efficiency of reducing accepting computations (see Section 3) to circuit satisfiability (and other related problems).

Definition 6.1. *We say that* **P** *is a* **proper** $\mathsf{TinyRAM}_{W,K}$ *program if it starts with the instructions:*

$\mathtt{I}_0$. `store.w` 0, `r0`

$\mathtt{I}_1$. `mov` `r0`, 2^{W-1}

$\mathtt{I}_2$. `read` `r1`, 0

$\mathtt{I}_3$. `cjmp` I_7

$\mathtt{I}_4$. `add` `r0`, `r0`, inc

$\mathtt{I}_5$. `store.w` `r0`, `r1`

$\mathtt{I}_6$. `jmp` I_2

$\mathtt{I}_7$. `store.w` 2^{W-1}, `r0`

where, above,

- $\mathtt{I_i} = 1 \cdot i$ *and* $\mathsf{inc} = 1$ *for* hvTinyRAM*, and*
- $\mathtt{I_i} = 2W/8 \cdot i$ *and* $\mathsf{inc} = W/8$ *for* vnTinyRAM*.*

In other words, we only consider TinyRAM programs working as follows. First, the program stores 0^W in address 0^W, and after that the program reads *all* of the primary input into memory (reading one word at a time, each time storing the word into the next available address starting from address $2^{W-1} + \mathsf{inc}$, and finally storing in address 2^{W-1} the address of where the last input word was stored).[8] Afterwards, since an n-word input is stored in addresses $2^{W-1}+\mathsf{inc}, \ldots, 2^{W-1}+n\cdot\mathsf{inc}$, when the program wants to access a word of the primary input, it can do so by reading the suitable address in memory. (The program can learn the length of the input because address 2^{W-1} contains the value $2^{W-1} + n \cdot \mathsf{inc}$.)[9]

Remark 6.2. In order to work correctly, a proper program should only be given inputs that are at most 2^{W-1} words in the case of hvTinyRAM and $2^{W+2}/W$ words in the case of vnTinyRAM. Furthermore, in the case of vnTinyRAM, one should only consider programs that are at most $2^{W+1}/W$ instructions, or else some instructions stored in the "lower half of memory" are overwritten by the preamble.

[8] Let us explain the code in Definition 6.1 in somewhat more detail. First of all, Instruction 0 is only a technicality, and the "interesting" part of the code is Instructions 1 through 7, which are the instructions responsible for reading the primary input. Register `r0` stores a pointer to the last available address, while register `r1` holds the current word read from the primary input tape (i.e., tape 0). Instruction 2 reads the next primary input word from the tape; if there is one, then it is stored in register `r1` and the condition flag `flag` is not set; otherwise `flag` is set. Instruction 3 checks if `flag` is set. If `flag` is not set, the program proceeds to increase the counter `r0` and then store in memory the new word from the input; then the program jumps back to Instruction 2 in order to try to read a new word from tape 0. Otherwise, if `flag` is set, the program jumps to Instruction 7 in order to store in address 2^{W-1} the current value of `r0`, which holds address $2^{W-1} + n \cdot \mathsf{inc}$ (where n is the number of words in the primary input) which is the address at which the last word was stored.

[9] A program may, later, reuse these memory addresses. However, the primary input tape is fully consumed and cannot be read again.

7 Binary Encoding Of Instructions

An instruction is specified via an opcode and up to three operands. (See Section 4.) We now describe the binary encoding of an instruction. The binary encoding assumes that $6 + 2 \cdot \lceil \log_2 K \rceil \leq W$; this is the case for natural choices of K and W. (E.g., $K, W = 16$ or $K, W = 32$ both work.)

An instruction is encoded via the following six binary fields.

- *Field #1.* This field stores the instruction's opcode, which consists of $5 = \lceil \log_2 29 \rceil$ bits. (See Table 2 for the opcodes.)
- *Field #2.* This field is a bit that is 0 if A is a register name and 1 if A is an immediate value.
- *Field #3.* This field stores a register name operand, which consists of $\lceil \log_2 K \rceil$ bits. It is all 0's when not used. This is the name of the instruction's destination register (i.e., the one to be modified) if any.
- *Field #4.* This field stores a register name operand, which consists of $\lceil \log_2 K \rceil$ bits. It is all 0's when not used. This is the name of a register operand (if any) that will not be modified by the instruction.
- *Field #5.* This field consists of padding with any sequence of $W - 6 - 2|K|$ bits, so that the first five fields fit exactly in a string of W bits.
- *Field #6.* This is either the name of another register (which is not modified by the instruction) or an immediate value, as determined by field #2. The length of this field is W bits (which is the maximum between the length of a register name and of an immediate value).

Overall the instruction is thus encoded using $2W$ bits, by concatenating fields 1 through 6 (in this order) and taking the MSB-to-LSB representation of each field. For example, if we take $K, W = 16$, a valid encoding of the instruction `add r3 r7 1234` is the following $2W = 32$ bits:

$$\underbrace{00100}_{\texttt{add}}\ \underbrace{1}_{\text{3rd arg is imm.}}\ \underbrace{0011}_{\texttt{r3}}\ \underbrace{0111}_{\texttt{r7}}\ \underbrace{00}_{\text{padding}}\ \underbrace{0000010011010010}_{1234}\ .$$

See Table 2 for details on opcodes and field assignments. Any sequence of $2W$ bits beginning with an opcode that does not appear in Table 2 is assumed to encode the instruction `answer` 1.

instruction mnemonic	operands			binary encoding (in six fields) #1	#2	#3	#4	#5	#6
`and`	ri	rj	A	00000	0/1	$\langle i \rangle$	$\langle j \rangle$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`or`	ri	rj	A	00001	0/1	$\langle i \rangle$	$\langle j \rangle$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`xor`	ri	rj	A	00010	0/1	$\langle i \rangle$	$\langle j \rangle$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`not`	ri	A		00011	0/1	$\langle i \rangle$	$*^{\|K\|}$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`add`	ri	rj	A	00100	0/1	$\langle i \rangle$	$\langle j \rangle$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`sub`	ri	rj	A	00101	0/1	$\langle i \rangle$	$\langle j \rangle$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`mull`	ri	rj	A	00110	0/1	$\langle i \rangle$	$\langle j \rangle$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`umulh`	ri	rj	A	00111	0/1	$\langle i \rangle$	$\langle j \rangle$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`smulh`	ri	rj	A	01000	0/1	$\langle i \rangle$	$\langle j \rangle$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`udiv`	ri	rj	A	01001	0/1	$\langle i \rangle$	$\langle j \rangle$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`umod`	ri	rj	A	01010	0/1	$\langle i \rangle$	$\langle j \rangle$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`shl`	ri	rj	A	01011	0/1	$\langle i \rangle$	$\langle j \rangle$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`shr`	ri	rj	A	01100	0/1	$\langle i \rangle$	$\langle j \rangle$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`cmpe`	ri	A		01101	0/1	$*^{\|K\|}$	$\langle i \rangle$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`cmpa`	ri	A		01110	0/1	$*^{\|K\|}$	$\langle i \rangle$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`cmpae`	ri	A		01111	0/1	$*^{\|K\|}$	$\langle i \rangle$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`cmpg`	ri	A		10000	0/1	$*^{\|K\|}$	$\langle i \rangle$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`cmpge`	ri	A		10001	0/1	$*^{\|K\|}$	$\langle i \rangle$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`mov`	ri	A		10010	0/1	$\langle i \rangle$	$*^{\|K\|}$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`cmov`	ri	A		10011	0/1	$\langle i \rangle$	$*^{\|K\|}$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`jmp`	A			10100	0/1	$*^{\|K\|}$	$*^{\|K\|}$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`cjmp`	A			10101	0/1	$*^{\|K\|}$	$*^{\|K\|}$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`cnjmp`	A			10110	0/1	$*^{\|K\|}$	$*^{\|K\|}$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`store.b`	A	ri		11010	0/1	$\langle i \rangle$	$*^{\|K\|}$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`load.b`	ri	A		11011	0/1	$\langle i \rangle$	$*^{\|K\|}$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`store.w`	A	ri		11100	0/1	$\langle i \rangle$	$*^{\|K\|}$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`load.w`	ri	A		11101	0/1	$\langle i \rangle$	$*^{\|K\|}$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`read`	ri	A		11110	0/1	$\langle i \rangle$	$*^{\|K\|}$	$*^{W-6-2\|K\|}$	$\langle A \rangle$
`answer`	A			11111	0/1	$*^{\|K\|}$	$*^{\|K\|}$	$*^{W-6-2\|K\|}$	$\langle A \rangle$

Table 2: Binary encoding of the TinyRAM instructions. The opcode in field #1 is written MSB-first. Field #2 is 0 if A is a register name, and 1 if A is an immediate value. Also, $|K|$ denotes $\lceil \log_2 K \rceil$; $\langle \cdot \rangle$ denotes the binary representation of the argument (which is $|K|$ bits long for fields #3 and #4, and W bits long for field #6); and $*^m$ denotes any binary string of m bits.

References

[BCG+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Proceedings of the 33rd Annual International Cryptology Conference*, CRYPTO '13, pages 90–108, 2013.

[BCGT13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems. In *Proceedings of the 4th Innovations in Theoretical Computer Science Conference*, ITCS '13, pages 401–414, 2013.

[BCTV14a] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In *Proceedings of the 34th Annual International Cryptology Conference*, CRYPTO '14, pages 276–294, 2014. Extended version at `http://eprint.iacr.org/2014/595`.

[BCTV14b] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *Proceedings of the 23rd USENIX Security Symposium*, Security '14, pages 781–796, 2014. Extended version at `http://eprint.iacr.org/2013/879`.

[SCI] SCIPR Lab. libsnark: a C++ library for zkSNARK proofs.