# TinyRAM Architecture Specification v0.991

Eli Ben-Sasson Alessandro Chiesa Daniel Genkin Eran Tromer Madars Virza
Technion MIT Technion Tel Aviv University MIT
eli@cs.technion.ac.il alexch@csail.mit.edu danielg3@cs.technion.ac.il tromer@cs.tau.ac.il madars@csail.mit.edu

SCIPR Lab scipr-lab.org

August 14, 2013

#### Abstract

We describe the TinyRAM architecture: a RISC random-access machine with a Harvard architecture and word-addressable random-access memory.

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 by arithmetic circuits, resulting in efficient verification using SCIPR's algorithmic and cryptographic mechanisms.

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

# Contents

1	Introduction	3
2	Architecture Overview	5
3	Accepting Computations	6
4	Instructions	7
5	Assembly Language	12
6	Preamble	13
7	Binary Encoding Of Instructions	14
$\mathbf{R}$	oferences	16

#### 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 a Harvard architecture and word-addressable random-access memory. It 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<sup>+</sup>13] in order to express correctness of nondeterministic computations in the setting of verifiable delegation of computations. For a discussion on the engineering choices behind the design of TinyRAM, see [BCG<sup>+</sup>13]; in this document, we focus on a precise specification of TinyRAM.

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). 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.<sup>2</sup>

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

<sup>&</sup>lt;sup>1</sup> 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.

<sup>&</sup>lt;sup>2</sup>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 0.991) is parametrized by two integers: the word size, denoted W, and the number of registers, denoted K. (At times, to avoid confusion, we explicitly denote this by using the notation TinyRAM<sub>W.K.</sub>.) The state of the machine consists of the following.

- The *program*, denoted **P**. Following the Harvard architecture paradigm, **P** is stored in a separate, read-only, address space (i.e., different from the read-write data address space).
- The program counter, denoted pc; it consists of W bits.
- K general-purpose registers, denoted  $r0, r1, \ldots, r(K-1)$ ; each register consists of W bits.
- The (condition) flag, denoted flag; it consists of a single bit.
- Memory, which is a linear array of  $2^W$  words of W bits each.
- 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.)

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.

The program **P** is a sequence of *instructions*. We shall specify the available instructions in Section 4; briefly, the instruction set of TinyRAM includes simple load and store instructions for accessing random-access memory, as well as simple integer, shift, logical, compare, move, and jump instructions. In particular, TinyRAM programs 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" by TinyRAM programs.

At every time step, TinyRAM fetches and executes the pc-th instruction of  $\mathbf{P}$ .<sup>3</sup> The only input to  $\mathbf{P}$  is via the two input tapes, and the 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.)

See Figure 1 below for a diagram of TinyRAM.

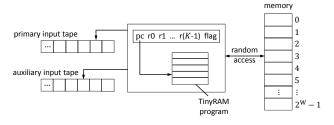


Figure 1: A diagram of TinyRAM. The word size is W and the number of registers is K.

<sup>&</sup>lt;sup>3</sup>We defined pc as a W-bit string. We use pc to also denote the corresponding integer between 0 and  $2^W - 1$ .

# 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 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.

The set of accepting computations is  $\mathcal{L} = \bigcup_{W,K} \mathcal{L}_{W,K}$ , where  $\mathcal{L}_{W,K}$  is the set of triples  $(\mathbf{P}, x, T)$  where  $\mathbf{P}$  is a TinyRAM<sub>W,K</sub> 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.<sup>5</sup>

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.

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

<sup>&</sup>lt;sup>4</sup>See more details about the answer instruction in Section 4.

 $<sup>^{5}</sup>$ 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".

<sup>&</sup>lt;sup>6</sup>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 27 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 increments pc (the program counter) by 1 (modulo  $2^W$ ) and does not modify flag. 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  $\{0,\ldots,K-1\}$ ;  $\mathbf{r}i$  is the i-th register, and  $[\mathbf{r}i]$  the W-bit string currently stored in it; similarly for  $\mathbf{r}j$  and  $[\mathbf{r}j]$ . 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  $[\mathbf{r}i]_{\mathbf{u}}$  the unsigned integer encoded by the contents of  $\mathbf{r}i$  (i.e.,  $\sum_{k=0}^{W-1} a_i 2^i$  if  $\mathbf{r}i$  stores the W-bit string  $a_{W-1}\cdots a_0$ ) and by  $[\mathbf{r}i]_{\mathbf{s}}$  the signed integer encoded by the contents of  $\mathbf{r}i$  (i.e., using two's complement,  $-a_{W-1}2^{W-1} + \sum_{i=0}^{W-2} a_i 2^i$  if  $\mathbf{r}i$  stores the W-bit string  $a_{W-1}\cdots a_0$ ). The notations  $[\mathbf{r}j]_{\mathbf{u}}$ ,  $[\mathbf{r}j]_{\mathbf{s}}$  and  $[A]_{\mathbf{u}}$ ,  $[A]_{\mathbf{s}}$  are similarly defined. When these notations are used, arithmetic is performed over the integers. 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	tion operands		ds	effects	flag	notes
mnemonic						
and	ri	$\mathtt{r} j$	A	compute bitwise AND of $[rj]$ and $[A]$ and store result in $ri$	result is $0^W$	
or	ri	$\mathtt{r} j$	A	compute bitwise OR of $[rj]$ and $[A]$ and store result in $ri$	result is $0^W$	
xor	ri	$\mathtt{r} j$	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	$\mathtt{r} j$	A	compute $[rj]_{u} + [A]_{u}$ and store result in $ri$	overflow	
sub	ri	$\mathtt{r} j$	A	compute $[rj]_{u} - [A]_{u}$ and store result in $ri$	borrow	
mull	ri	$\mathtt{r} j$	A	compute $[rj]_u \times [A]_u$ and store least significant bits of result in $ri$	overflow	
umulh	ri	$\mathtt{r} j$	A	compute $[rj]_u \times [A]_u$ and store most significant bits of result in $ri$	overflow	
smulh	$[ri  rj  A]$ compute $[rj]_s \times [A]_s$ and store most significant bits of result in $ri$				over/underflow	
udiv	ri	$\mathtt{r} j$	A	compute quotient of $[rj]_{\rm u}/[A]_{\rm u}$ and store result in $ri$	$[A]_{\mathbf{u}} = 0$	
umod	ri	$\mathtt{r} j$	A	compute remainder of $[rj]_{\rm u}/[A]_{\rm u}$ and store result in $ri$	$[A]_{\mathbf{u}} = 0$	
shl	ri	$\mathtt{r} j$	A	shift $[rj]$ by $[A]_u$ bits to the left and store result in $ri$	MSB of $[rj]$	
shr	ri	$\mathtt{r} j$	A	shift $[rj]$ by $[A]_u$ bits to the right and store result in $ri$	LSB of $[rj]$	
cmpe	ri	$\overline{A}$		none ("compare equal")	[ri] = [A]	
cmpa	ri	A		none ("compare above", unsigned)	$[ri]_{\mathrm{u}} > [A]_{\mathrm{u}}$	
cmpae	ri	A		none ("compare above or equal", unsigned)	$[ri]_{\mathrm{u}} \geq [A]_{\mathrm{u}}$	
cmpg	ri	A		none ("compare greater", signed)	$[ri]_{s} > [A]_{s}$	
cmpge	ri	A		none ("compare greater or equal", signed)	$[ri]_{s} \geq [A]_{s}$	
mov	ri	$\overline{A}$		store $[A]$ in $\mathbf{r}i$		
cmov	ri	A		if flag = 1, store [A] in $\mathbf{r}i$		
jmp	A			set pc to [A]		
cjmp	A			if flag = 1, set pc to [A] (else increment pc as usual)		
cnjmp	A			if flag = 0, set pc to [A] (else increment pc as usual)		
store	A	ri		store $[ri]$ at memory address $[A]_{u}$		
load	ri	$\stackrel{\scriptscriptstyle{-}}{A}$		store the content of memory address $[A]_{\mathbf{u}}$ into $\mathbf{r}i$		
read	ri			if the $[A]_{\mathbf{u}}$ -th tape has remaining words then consume the next word,		(1)
		store it in $\mathbf{r}i$ , and set flag = 0; otherwise store $0^W$ in $\mathbf{r}i$ and set flag = 1				
answer	stall or halt (and the return value is $[A]_{u}$ )					(2)
(1) All but t	1	irst t	wo ta	apes are empty: if $[A]_{\mathbf{u}} \notin \{0,1\}$ then store $0^W$ in $\mathbf{r}i$ and set flag = 1.	1	. /

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.

(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.

Bit operations. These are standard bit operations on registers.

- and: The instruction and ri rj A stores in ri the bitwise AND of [rj] and [A]. Moreover, flag is set to 1 if the result is  $0^W$  and to 0 otherwise.
- or: The instruction or ri rj A stores in ri the bitwise OR of [rj] and [A]. Moreover, flag is set to 1 if the result is  $0^W$  and to 0 otherwise.
- xor: The instruction xor ri rj A stores in ri the bitwise XOR of [rj] and [A]. Moreover, flag is set to 1 if the result is  $0^W$  and to 0 otherwise.
- not: The instruction not  $\mathbf{r}i$  A stores in  $\mathbf{r}i$  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  $\mathbf{r}i$   $\mathbf{r}j$  A stores in  $\mathbf{r}i$  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  $G=[\mathbf{r}j]_{\mathbf{u}}+[A]_{\mathbf{u}}$ . Moreover, flag is set to  $G_W$ , where  $G_W$  is the MSB of G.
- sub: The instruction sub  $\mathbf{r}i$   $\mathbf{r}j$  A stores in  $\mathbf{r}i$  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  $G=[\mathbf{r}j]_{\mathbf{u}}+2^W-[A]_{\mathbf{u}}$ . Moreover, flag is set to  $1-G_W$ , where  $G_W$  is the MSB of G.
- mull: The instruction mull  $\mathbf{r}i$   $\mathbf{r}j$  A stores in  $\mathbf{r}i$  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  $[\mathbf{r}j]_{\mathbf{u}}\times [A]_{\mathbf{u}}$ . Moreover, flag is set to 1 if  $[\mathbf{r}j]_{\mathbf{u}}\times [A]_{\mathbf{u}}\not\in U_W$  and to 0 otherwise.
- umulh: The instruction umulh  $\mathbf{r}i$   $\mathbf{r}j$  A stores in  $\mathbf{r}i$  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  $[\mathbf{r}j]_{\mathbf{u}}\times [A]_{\mathbf{u}}$ . Moreover, flag is set to 1 if  $[\mathbf{r}j]_{\mathbf{u}}\times [A]_{\mathbf{u}}\not\in U_W$  and to 0 otherwise.
- smulh: The instruction smulh  $\mathbf{r}i$   $\mathbf{r}j$  A stores in  $\mathbf{r}i$  the W-bit string  $a_{W-1}\cdots a_0$  obtained as follows:  $a_{W-1}$  is the sign of  $[\mathbf{r}j]_{\mathrm{s}}\times [A]_{\mathrm{s}}$  and  $a_{W-2}\cdots a_0$  are the W-1 most significant bits of the absolute value of  $[\mathbf{r}j]_{\mathrm{s}}\times [A]_{\mathrm{s}}$ . Moreover, flag is set to 1 if  $[\mathbf{r}j]_{\mathrm{s}}\times [A]_{\mathrm{s}}\not\in S_W$  and to 0 otherwise.
- udiv: The instruction udiv  $\operatorname{ri} rj A$  stores in  $\operatorname{ri}$  the W-bit string  $a_{W-1}\cdots a_0$  obtained as follows. If  $[A]_{\operatorname{u}}=0$ , then  $a_{W-1}\cdots a_0=0^W$ . If  $[A]_{\operatorname{u}}\neq 0$ , then  $a_{W-1}\cdots a_0$  is the binary encoding of Q where Q is the unique integer such that  $[\operatorname{r} j]_{\operatorname{u}}=[A]_{\operatorname{u}}\times Q+R$  for some integer  $R\in\{0,\ldots,[A]_{\operatorname{u}}-1\}$ . Moreover, flag is set to 1 if and only if  $[A]_{\operatorname{u}}=0$ .

<sup>&</sup>lt;sup>7</sup>An equivalent definition of the mull instruction is the following: "The instruction mull  $\mathbf{r}i$   $\mathbf{r}j$  A stores in  $\mathbf{r}i$  the W-bit string  $a_{W-1}\cdots a_0$  obtained as follows:  $a_{W-1}$  is the sign of  $[\mathbf{r}j]_{\mathbf{s}}\times [A]_{\mathbf{s}}$  and  $a_{W-2}\cdots a_0$  are the W-1 least significant bits of the absolute value of  $[\mathbf{r}j]_{\mathbf{s}}\times [A]_{\mathbf{s}}$ . Moreover, flag is set to 1 if  $[\mathbf{r}j]_{\mathbf{u}}\times [A]_{\mathbf{u}}\not\in U_W$  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  $[A]_u = 0$ , then  $a_{W-1} \cdots a_0 = 0^W$ .

If  $[A]_{\mathbf{u}} \neq 0$ , then  $a_{W-1} \cdots a_0$  is the binary encoding of R where R is the unique integer in  $\{0, \ldots, [A]_{\mathbf{u}} - 1\}$  such that  $[\mathbf{r}j]_{\mathbf{u}} = [A]_{\mathbf{u}} \times Q + R$  for some integer Q. Moreover, flag is set to 1 if and only if  $[A]_{\mathbf{u}} = 0$ .

**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 [rj] by  $[A]_u$  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 [rj].

shr: The instruction shr ri rj A stores in ri the W-bit string obtained by shifting [rj] by  $[A]_u$  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 [rj].

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 [ri] = [A] and to 0 otherwise.

cmpa: The instruction cmpa ri A sets flag to 1 if  $[ri]_u > [A]_u$  and to 0 otherwise.

cmpae: The instruction cmpae ri A sets flag to 1 if  $[ri]_u \ge [A]_u$  and to 0 otherwise.

cmpg: The instruction cmpg ri A sets flag to 1 if  $[ri]_s > [A]_s$  and to 0 otherwise.

cmpge: The instruction cmpge ri A sets flag to 1 if  $[ri]_s \ge [A]_s$  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: The instruction store A ri stores [A] into address  $[ri]_u$  of memory.

load: The instruction load ri A stores the W-bit string at address  $[A]_u$  of memory into ri.

**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  $\mathbf{r}i$  A stores in  $\mathbf{r}i$  the next W-bit word on the  $[A]_{\mathbf{u}}$ -th tape, if any. More precisely, if the  $[A]_{\mathbf{u}}$ -th tape has remaining words then consume the next word, store it in  $\mathbf{r}i$ , and set flag = 0; otherwise (if there are no remaining input words on the  $[A]_{\mathbf{u}}$ -th tape) store  $0^W$  in  $\mathbf{r}i$  and set flag = 1.

Because TinyRAM only has two input tapes, all but the first two tapes are assumed to be empty. Specifically, if  $[A]_u$  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  $[A]_{\rm u}$ . 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  $\mathbf{P}$  is a text file consisting of a sequence of lines (separated by CR, LF or CR/LF). The text file is structured as follows. The first line contains the string

"; TinyRAM V=1.00 W=
$$W$$
 K= $K$ "

where 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  $[x]_u \equiv a \mod 2^W$ .
- 4. Optional whitespace.
- 5. An optional comment starting with a semicolon (";") and lasting until the end of the line.

Instructions are (implicitly) numbered sequentially, starting with 0 and ignoring non-instruction lines. 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<sup>+</sup>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** TinyRAM<sub>W,K</sub> program if it starts with the following instructions (where, for readability, we have made explicit the implicit instruction numbers):

- 0. store 0, r0
- 1. mov r0, 1
- 2. read r1, 0
- 3. cimp 7
- 4. add r0, r0, 1
- 5. store r0, r1
- 6. jmp 2
- 7. store 1, r0

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, and finally storing in address 1 the address of where the last input word was stored).<sup>8</sup> Afterwards, since an n-word input is stored in addresses  $2, 3, \ldots, n+1$ , 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 1 contains the value n+1.)<sup>9</sup>

<sup>&</sup>lt;sup>8</sup>Let us explain the code in Definition 6.1 in somewhat more detail. First of all, Instruction 0 is only a technicality (needed even if we require, as we do, that memory is initialized to "all zeros"), 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 1 the current value of r0, which holds address n+1 (where n is the number of words in the primary input) which is the address at which the last word was stored.

<sup>&</sup>lt;sup>9</sup> 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

Recall that 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  $6 + 2 \cdot \lceil \log_2 K \rceil \le W$ ; 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 six binary fields, all using the little-endian convention.

- Field #1. This field stores the instruction's opcode, which consists of  $5 = \lceil \log_2 27 \rceil$  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  $0^{W-6-2|K|}$  so that the first six 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 encoded using 2W bits. See Table 2 for details on opcodes and field assignments.

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

Table 2: Binary encoding of 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  $\lceil \log_2 K \rceil$ , and  $\langle \cdot \rangle$  denotes the binary representation of the argument, which is |K| bits long for fields #3 and #4, and W bits long for field #6.

# References

- [BCG<sup>+</sup>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, 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.