B.1. Simulation of a RAM by a Turing machine.

Theorem 1.1 *Let L be a language over some alphabet Σ. If there is a RAM that accepts L, then there is a Turing machine that also accepts L.*

proof. Let P be any RAM program constructed according to the syntax of Note 5. Our aim is to construct a Turing machine M that correctly simulates the operation of P on all inputs $x \in \Sigma^*$. It is convenient to allow M to be a machine with multiple tapes; we know from Note 4 that M could, in turn, be simulated by a one-tape machine. We shall not attempt to present a formal description of M in terms of states, tuples, etc. Instead, the proposed machine M will be divided into a number of functional components, and the operation of each of these described informally. Each of these functional components will be sufficiently simple that we shall be left in no doubt that the machine M could, in principle, be written down quite formally. Our growing experience with Turing machines will assure us that the details could be supplied on request.

The Turing machine M has an *input tape*, a *storage tape*, and a number of *work tapes*. The input tape holds the input word $x \in \Sigma^*$, and simulates the input stream of the RAM in a straightforward manner. The storage tape of M records the current state of the RAM in a format shortly to be described. The work tapes provide temporary storage for addresses and operands, and for performing simple arithmetic computations. The storage tape and work tapes are initially blank, but the first action of the machine M is to write a dollar symbol, `$`, onto the leftmost square of the storage tape. It will become apparent that the storage tape now contains an encoding of the zero function, which is the initial state of the RAM.

The storage tape of M, at a typical instant in the simulation, has the following format:

$$\texttt{\$\#}a_1 : v_1 \texttt{\#}a_2 : v_2 \texttt{\#}a_3 : v_3 \texttt{\#} \cdots \texttt{\#}a_m : v_m \text{ƀ ƀ ƀ} \cdots, \qquad (*)$$

where a_i and v_i are integers represented as signed binary numbers. Roughly, each pair $a_i : v_i$ appearing on the storage tape can be interpreted as an assertion that the register with address a_i has content v_i. If the storage tape contains no assertion about a particular register, then that register is deemed to contain zero. If there are a number of contradictory assertions about a particular register, then the rightmost assertion takes priority. More formally, the storage tape $(*)$ specifies a state $s : \mathbb{Z} \to \mathbb{Z}$ of the RAM, the function s being defined as follows. Let a be any integer. If $a \neq a_i$ for all i in the range $1 \leq i \leq m$, then $s(a) = 0$. Otherwise, $s(a) = v_j$, where j is the largest index for which $a_j = a$.

The storage tape of M is required to support two operations:

(E) Given an integer a, evaluate $s(a)$. That is, determine the content of a register given its address.

(U) Given two integers v and a, modify the storage tape so that it becomes a representation of the new state $s' = \text{update}(s, v, a)$. That is, assign the value v to the register with address a.

Both operations are straightforward to implement as subroutines within M.

First, consider operation (E). Suppose the address a is presented as a signed binary number on a designated work tape of M, and $s(a)$ is to be returned on another designated work tape. The machine M scans right along the storage tape until it encounters a blank symbol. It then scans left along the storage tape looking for the first occurrence of the substring `#`a`:` on the tape. If the dollar symbol, `$`, is encountered before the substring is found, then `0` is written to the result tape. Otherwise the head is shifted to the square immediately to the right of the substring just located, and the signed binary number appearing there is copied to the result tape. We shall refer to this entire procedure as subroutine (E).

Operation (U) is even more straightforward to implement. Suppose the integer address a and integer value v are presented on designated worktapes of M. The machine M scans right along the storage tape until it finds the first blank symbol. It then continues scanning to the right, copying the string `#`a`:`v to the storage tape as it proceeds. We shall refer to this procedure as subroutine (U).

Note that subroutine (E) searches the storage tape from *right* to *left*, and that subroutine (U) always adds new pairs to the *right* of all existing pairs. Thus, when a new pair `#`a`:`v is added to the storage tape, all existing pairs of the form `#`a`:`v' (i.e., referring to the same address a) are rendered inaccessible. Subroutine (U) thus achieves the effect of *overwriting* the previous content of the register with address a.

Having described the use made by M of the storage tape, we are now in position to describe how M may simulate each instruction of the RAM program P, and hence the program itself. We consider each instruction type in turn. (Refer to NOTE 5.)

(a) `accept`: M immediately accepts.

(b) `reject`: M immediately halts without accepting.

(c) `read` L: M reads a symbol from the input tape and converts it to an integer code v, which is written to a designated work tape. (Recall that the RAM has an internal code in the set $\{1, 2, \ldots, |\Sigma|\}$ for each symbol of the input alphabet Σ. The blank symbol has code 0, meaning 'end-of-input'.) At the same time the head scanning the input tape is moved right one square in preparation for the next `read` instruction. The operand L is now evaluated in the context of the current state s of the RAM to yield an address a; this address also is written to a designated worktape. The evaluation of L,

if it is of the form "k, will employ subroutine (E). Finally, the storage tape is updated using subroutine (U). The storage tape now contains a representation of the new state $s' = \text{update}(s, v, a)$ of the RAM.

1. (d)L `:=` $R_1 \circ R_2$: The simulating machine proceeds as follows. First, the operands R_1 and R_2 are evaluated in context s, and the results v_1 and v_2 stored on two of the work tapes. The evaluation of operand R_i involves zero, one, or two applications of subroutine (E), depending on whether R_i has the form k, 'k, or "k. The machine M then computes $v_1 \circ v_2$, and stores the result, v, on a designated work tape. (Note that the four arithmetic operators, `+`, `-`, `*`, and `div`, can be implemented as Turing machine subroutines.) Next, L is evaluated in context s to yield an integer address a, which is stored on a designated work tape. The evaluation of L may involve a further application of subroutine (E). Finally, the storage tape is updated using subroutine (U). The storage tape now contains a representation of the new state $s' = \text{update}(s, v, a)$ of the RAM.

(e) `if` $R_1 \circ R_2$ `goto` λ: Using subroutine (E), the operands R_1 and R_2 are evaluated in context s, and the results v_1 and v_2 stored on two of the work tapes. M then computes $v_1 \circ v_2$, and exits to different states according to whether the result is true or false. (Note that the four relational operators, `=`, `<>`, `<=`, and `<`, can be implemented as Turing machine subroutines.)

Using the constructions described in paragraphs (a)–(e) above, each instruction in the RAM program P can be translated into a Turing machine subroutine. Each subroutine can be considered, graphically, as a collection of states with associated transitions. Each subroutine has one entry point (state), and up to two exits (transitions from states): (a) and (b) have no exits, (c) and (d) have one, and (e) has two (corresponding to the branch condition being true or false).

The machine M is now simply obtained by forming the disjoint union of the subroutines corresponding to all the instructions in the program P, and then gluing together the entry points and exit transitions of the subroutines so that the instructions of P are simulated in the correct sequence. □