C.1. Simulation of a Turing machine by a RAM. We saw, in Appendix B, that any language that is accepted by a RAM is also accepted by a Turing machine. Here we prove that the converse holds: any language that is accepted by a Turing machine is also accepted by a RAM. In fact we demonstrate rather more. Recall that a *three-register RAM* is a random access machine, as defined in Note 5, but having just three registers, with addresses -1, 0, and 1. The *state* of a three register RAM is thus a function from $\{-1, 0, 1\}$ to $\mathbb{Z}$. To avoid referencing non-existent registers we place a severe restriction on the form of *l_values* and *r_values* that may occur in a program for a three-register RAM: the only *l_values* allowed are `'-1`, `'0`, and `'1`; and the only *r_values* allowed are `'-1`, `'0`, `'1`, and signed decimal constants. Note that 'indirect addressing' is forbidden.

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

proof. Let $M = (Q, \Gamma, \Sigma, \text{ƀ}, q_I, q_F, \delta)$ be a (one-tape) Turing machine that accepts the language L. We shall construct a three-register RAM program P that simulates M.

During its simulation of M, the RAM maintains an encoding of the contents of M's tape; the encoding scheme employed is the following. Let $m = |\Gamma| + 1$, and assign to each symbol in Γ a distinct internal code which is an integer in the range 0 to $m - 2$. We insist that the blank symbol receives code 0, and that elements of Σ receive codes in the range $1, \ldots, |\Sigma|$; otherwise the assignment of codes to symbols is arbitrary. The number $m - 1$ is reserved as the code for a special 'end-of-tape symbol' whose purpose will become apparent in due course.

Now fix attention on the tape of M at some instant during a computation. Assume that the tape squares are numbered in sequence, starting at zero. For each natural number i let a_i be the internal code of the symbol appearing in tape square i, and let a_{-1} be $m-1$ (the end-of-tape symbol). Also let k be the sequence number of the currently scanned tape square. Then the tape contents of M are encoded as three integers which are stored in the registers of the RAM as indicated below.

$$\begin{aligned} \text{content of register } -1: &\quad l = a_{k-1} + a_{k-2}m + a_{k-3}m^2 + \cdots + a_{-1}m^k; \\ \text{content of register } 0: &\quad s = a_k; \\ \text{content of register } 1: &\quad r = a_{k+1} + a_{k+2}m + a_{k+3}m^2 + \cdots . \end{aligned}$$

Note that r is a well defined integer, despite being specified by a infinite series; to see this, recall that the internal code of the blank symbol is zero, and that there can be only a finite number of non-blank symbols on M's tape. Note also that the

three registers of the RAM between them provide a complete description of the tape contents: the digits of l and r, when expressed as numbers in base m, yield the internal codes of the symbols appearing to the left and right of the tape head, while s is the internal code of the scanned symbol.

We now consider the flow of control in the program P that performs the simulation. Let the states of M be $q_0, q_1, q_2, \ldots, q_{|Q|-1}$, where q_0 is the initial state. The top-level structure of P is presented in Figure 2. Here, as elsewhere in this note, the notation $\langle n \rangle$ is used to denote the decimal representation of the number n. (Thus, if M were a 186-state machine, `state`$\langle |Q| - 1 \rangle$ would stand for the string `state185`.) Aside from some preliminary code concerned with initialization, it will

$$
\begin{array}{rl}
 & [\![\text{code to read the input word and initialise registers}]\!] \\
\texttt{state0:} & [\![\text{code to simulate transitions from state } q_0]\!] \\
\texttt{state1:} & [\![\text{code to simulate transitions from state } q_1]\!] \\
\texttt{state2:} & [\![\text{code to simulate transitions from state } q_2]\!] \\
 & \vdots \\
\texttt{state}\langle |Q| - 1 \rangle\texttt{:} & [\![\text{code to simulate transitions from state } q_{|Q|-1}]\!]
\end{array}
$$

Figure 2: Stepwise refinement: deciding the state.

be seen that the program P is formed from a series of *blocks*, each block dealing with transitions from a single state. We shall consider these blocks first, returning at the end to deal with the task of initialization. The block corresponding to the final state of M is special, consisting of a single `accept` instruction. Each of the other blocks is constructed according to a fixed template; a typical instance—the block corresponding to state q_0—is shown in Figure 3. What we see in Figure 3 is a primitive case-statement whose limbs are selected according to the scanned symbol. The state q and scanned symbol s having been determined, the code within each limb of the case-statement now has the job of implementing the transition itself. If no transition is defined for the pair (q, s), then the limb consists of a single `reject` instruction. Otherwise, suppose $\delta(q, s) = (q', s', L)$. (The case of a right shifting transition is handled in an analogous manner.) The first action is to simulate the overwriting of the current symbol of M: this is handled by a single instruction which simply assigns the internal code for s' to register 0. The next action is to simulate the left shift of the tape head, which is achieved by the code presented in Figure 4. (Observe that if the head of M drops off the end of the tape, then the end-of-tape code appears in register 0. This condition is trapped at the next cycle of the simulation.) The reader should check that the register

```
                if '0 = 0 goto pair0X0
                if '0 = 1 goto pair0X1
                if '0 = 2 goto pair0X2
                        ⋮
                if '0 = ⟨m − 2⟩ goto pair0X⟨m − 2⟩
                reject
      pair0X0:  ⟦scanned symbol has internal code 0 (i.e., is ƀ)⟧
      pair0X1:  ⟦scanned symbol has internal code 1⟧
      pair0X2:  ⟦scanned symbol has internal code 2⟧
                        ⋮
pair0X⟨m − 2⟩:  ⟦scanned symbol has internal code m − 2⟧
```

Figure 3: Stepwise refinement: deciding the symbol.

contents after execution of this code fragment are

$$\begin{aligned}
\text{content of register } -1: &\quad l' = a_{k-2} + a_{k-3}m + a_{k-4}m^2 + \cdots + a_{-1}m^{k-1};\\
\text{content of register } 0: &\quad s' = a_{k-1};\\
\text{content of register } 1: &\quad r' = a_k + a_{k+1}m + a_{k+2}m^2 + \cdots ;
\end{aligned}$$

as we should expect. The final action of the RAM in simulating a single transition

```
'1   :=  '1 * ⟨m⟩
'1   :=  '1 + '0
'0   :=  '-1 + 0
'-1  :=  '-1 div ⟨m⟩
'-1  :=  '-1 * ⟨m⟩
'0   :=  '0 - '-1
'-1  :=  '-1 div ⟨m⟩
```

Figure 4: Stepwise refinement: shifting left.

of M is to jump unconditionally to the instruction labelled `state`$\langle i \rangle$, where i is the index of the next state.

It only remains to deal with the code for input and initialization. If the input loop is arranged in the obvious way, the head of the simulated machine ends up scanning the first blank symbol. However a second loop incorporating a left shift will return the tape head to the leftmost square in readiness for the simulation proper. The necessary code is presented in Figure 5. This completes the description of the program P. □

```
                '-1 := ⟨m − 1⟩
   next_char:   read '0
                if '0 = 0 goto end_of_input
                '-1 := '-1 * ⟨m⟩
                '-1 := '-1 + '0
                if 0 = 0 goto next_char
end_of_input:   '1 := 0
       shift:   if '-1 = ⟨m − 1⟩ goto state0
                ⟦code to shift head left: see Figure 4⟧
                if 0 = 0 goto shift
```

Figure 5: Stepwise refinement: input and initialization.

EXERCISE (Hard! The kind that's usually marked with a $*$) Does the theorem remain true if 'three-register RAM' is replaced by 'two-register RAM'?