# Note 5: Random Access Machines

Over the years, many different formulations of the notion of effective computability have been proposed. These formulations differ widely in appearance; nevertheless, whenever a new formulation has been proposed, it has always turned out to be equivalent to all previous ones. The failure of anybody to find a reasonable model of computation which properly extends the notion of effective computability provides what is perhaps the most convincing empirical evidence in support of the Church-Turing thesis.

In this note, we consider a model of computation very different from the Turing machine, namely the *Random Access Machine*, or *RAM*. The significance of the RAM is that it provides a relatively simple basis for computation which is not far removed from real computers. Thus, in introducing the RAM, we bring into play all the intuition about computation that we have gained from our experience.

As its name suggests, the main feature of the RAM is its ability to access data *by address*, rather than merely *sequentially* as in a Turing machine. (The relationship between the two models can be likened to the relationship between the *array* and the *doubly-linked list* as data-structuring techniques.) Despite the apparently less restricted nature of computation on a RAM, we shall see that the RAM and Turing machine are of equivalent power.

## 1 Syntax of RAM programs

The syntax of a RAM program is presented below, in Backus-Naur form.

$$\begin{aligned}
\textit{program} &= \textit{instruction program} \mid \textit{instruction}\\
\textit{instruction} &= [\,\textit{label} :]\ \big(\texttt{accept}\\
&\qquad\quad \mid \texttt{reject}\\
&\qquad\quad \mid \texttt{read}\ \textit{l\_value}\\
&\qquad\quad \mid \textit{l\_value}\ \texttt{:=}\ \textit{r\_value arithmetic\_op r\_value}\\
&\qquad\quad \mid \texttt{if}\ \textit{r\_value relational\_op r\_value}\ \texttt{goto}\ \textit{label}\big)\\
\textit{l\_value} &= \texttt{'}\textit{integer} \mid \texttt{"}\,\textit{integer}\\
\textit{r\_value} &= \textit{integer} \mid \texttt{'}\textit{integer} \mid \texttt{"}\,\textit{integer}\\
\textit{arithmetic\_op} &= + \mid - \mid \star \mid \texttt{div}\\
\textit{relational\_op} &= \texttt{=} \mid \texttt{<>} \mid \texttt{<=} \mid \texttt{<}\\
\textit{label} &= \textit{alphanumeric\_sequence}.
\end{aligned}$$

The non-terminal *integer* is intended to stand for an arbitrary signed decimal number. Naturally enough, we insist that no two instructions are assigned the same label, and that every conditional jump refers to an existent label.

We shall assign a precise meaning to RAM programs in the following section. Roughly speaking, however, an unquoted integer denotes a *constant*, an integer prefixed by a single quote is to be interpreted as a *direct address*, and an integer prefixed by a double quote is to be interpreted as an *indirect address*. Armed with this clue, you should already be able to guess the meaning of many, if not all the instruction types.

## 2 Semantics of RAM programs

We assign meaning to a RAM program by introducing the notion of a *state* of a random access machine. Informally, a RAM has an infinite number of registers, each containing an arbitrary integer. The registers are indexed by the integers; the index of a register is sometimes called its *address*. Before execution of a program, all registers contain zero. The instructions of a program are thought of as being numbered in sequence, starting at zero. A particular instruction of the program is picked out by an *instruction counter*, which is a natural number; this instruction counter is initialized to zero. Execution of the program proceeds in a sequence of steps. In a single step, the instruction indicated by the instruction counter is executed, as a result of which the state of the machine and the instruction counter are updated. The state and instruction counter together play the role of configuration in the Turing machine model.

Formally, the *state* of a RAM is a function $s : \mathbb{Z} \to \mathbb{Z}$, which defines the content $s(i)$ of each register $i$. The function $s$ has a finite description, being zero on all but a finite subset of $\mathbb{Z}$. The initial state of a RAM (before the program executes) is the zero function. Before describing the effect of executing each instruction type, it is convenient to introduce some terminology and notation. If $R$ is an *r_value* and $s$ a state, then the *result of evaluating $R$ in context $s$* is an integer value $v$ given by

$$v = \begin{cases} k, & \text{if } R = k; \\ s(k), & \text{if } R = \texttt{'}k; \\ s(s(k)), & \text{if } R = \texttt{"}k. \end{cases}$$

If $L$ is an *l_value* and $s$ a state, then the *result of evaluating $L$ in context $s$* is an integer address $a$ given by

$$a = \begin{cases} k, & \text{if } L = \texttt{'}k; \\ s(k), & \text{if } L = \texttt{"}k. \end{cases}$$

Finally, if $s$ is a state, $v$ an integer value, and $a$ an integer address, then $\mathrm{update}(s, v, a)$ denotes the new state $s' : \mathbb{Z} \to \mathbb{Z}$ given by

$$s'(i) = \begin{cases} v, & \text{if } i = a; \\ s(i), & \text{otherwise.} \end{cases}$$

Informally, the new state $s'$ agrees with the old state $s$ at all points except $a$, where $s'(a) = v$ independent of the value of $s(a)$.

We are now in a position to assign a meaning to each instruction by specifying how that instruction modifies the state and instruction counter. Let $s$ denote the state before execution of the instruction in question, and $s'$ denote the state afterwards.

(a) `accept`: The RAM halts, and accepts its input.

(b) `reject`: The RAM halts, and rejects its input.

(c) `read` $L$: The input to a RAM is a stream of integers. The next value, say $v$, is removed from the stream. Let $a$ be the result of evaluating $L$ in context $s$. Then $s'$ becomes $\mathrm{update}(s, v, a)$, and the instruction counter is incremented by one.

(d) $L$ `:=` $R_1 \circ R_2$: Let $a$, $v_1$, and $v_2$ be the results of evaluating $L$, $R_1$, and $R_2$ in context $s$, and let $v = v_1 \circ v_2$.[1] Then $s'$ becomes $\mathrm{update}(s, v, a)$, and the instruction counter is incremented by one.

[1]The operator $\star$ denotes integer multiplication and `div` denotes integer division. Thus `div` takes two integers $v_1$ and $v_2$, with $v_2 > 0$, and yields the unique integer $v$ satisfying $0 \leq v_1 - vv_2 < v_2$; if $v_2 \leq 0$ the program halts and rejects.

(e) `if` $R_1 \circ R_2$ `goto` $\lambda$: Let $v_1$ and $v_2$ be the results of evaluating $R_1$ and $R_2$ in context $s$. If $v_1 \circ v_2$ is false, the instruction counter is incremented by one. If $v_1 \circ v_2$ is true, the instruction counter is set to the index of the instruction labelled by $\lambda$. In either case the state is unchanged, i.e., $s' = s$.

After executing the (syntactically) last instruction of a program, the instruction counter may no longer contain a meaningful value; in that case the RAM halts and rejects.

A random access machine $M$ of the form described above can be viewed as a language recognizer. Let $\Sigma$ be a finite input alphabet, and associate the symbols of $\Sigma$ with the numbers $1, 2, 3, \ldots, |\Sigma|$. Then a word $x \in \Sigma^n$ can be presented to the RAM as a sequence of $n$ positive numbers (encoding elements of $\Sigma$) followed by 0 (which can be thought of as an end-of-input marker or blank symbol). The *language $L(M)$ recognized by $M$* is then the set of words $x \in \Sigma^*$ on which $M$ halts and accepts. It is a straightforward task to extend the model to encompass *transducers* by adding a instruction of the form '`write` *r_value*' to the repertoire of instructions.

## 3 Example: recognizing palindromes

```
                 '1 := 2
  next_symbol :  read "1
                 if "1 = 0 goto end_of_input
                 '1 := '1 + 1
                 if 0 = 0 goto next_symbol
 end_of_input :  '1 := '1 - 1
                 '0 := 2
         loop :  if '1 <= '0 goto yes
                 if "0 <> "1 goto no
                 '0 := '0 + 1
                 '1 := '1 - 1
                 if 0 = 0 goto loop
          yes :  accept
           no :  reject
```

Figure 1: A RAM program for recognizing palindromes

The RAM program in Fig. 1 above recognizes the language of palindromes over $\{a, b\}$, where $a$ is encoded as 1, and $b$ as 2. The $n$ symbols of the input are read into registers 2 to $n + 1$, which can be thought of as constituting an $n$-element array. Registers 0 and 1 are used to implement indices, $i$ and $j$ say, into this array. Initially, $i = 2$ and $j = n + 1$. At each iteration, the array elements indexed by $i$ and $j$ are compared. If these elements are found to be unequal then the input was *not* a palindrome and the program halts and rejects. Otherwise the index $i$ is incremented, and $j$ decremented. If the pointers cross (i.e, $j$ becomes less than or equal to $i$) then the input *was* a palindrome and the program halts and accepts.

## 4 Redundancy

The RAM model described in this note contains a fair number of redundant features. It is not too difficult to show that the arithmetic operators `+`, `*`, and `div` can be removed without affecting the class of languages which can be recognized. Likewise, the relational operators =, <>, and < are redundant. More surprisingly, as we shall see later, it is possible to make do with a fixed, finite set of registers, and dispense with indirect addressing entirely.

## 5 Simulation of a RAM by a Turing machine

Nobody has yet conceived of a procedure which could reasonably be described as effective, but which could not be expressed in a high-level programming language such as Java or C. Our failure to find such an object can be regarded as empirical evidence that every effective procedure could, in principle, be expressed in such a high-level language. This claim should accord with your own experience over several years of programming in different languages and environments.

You will also be aware that any program written in a high-level language can be translated into machine code, as long as we ignore the limitations inherent in a *bounded* word-size and *bounded* address-space. This, of course, is exactly the job of a compiler, and we have definite evidence that such things exist. Now the RAM model clearly has at least the power of a conventional machine code, but without the restrictions implied by bounded word-size or address-space. We thus have convincing empirical evidence that any effective procedure can be expressed as a RAM program.

Our experience with Turing machines, on the other hand, is much more limited, and we may be less convinced that every effective procedure can be described by a Turing machine. The purpose of this section is to demonstrate that Turing machines are at least as powerful as RAMs, and hence to provide convincing empirical evidence in support of the Church-Turing Thesis. The claim is made precise in the following theorem.

**Theorem 1.** *Let $L$ be a language over some alphabet $\Sigma$. If there is a RAM that recognizes $L$, then there is a Turing machine that also recognizes $L$.*

*Proof.* Let $P$ be any RAM program. 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 already know 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:

$$\texttt{\$}\,\texttt{\#}a_1\texttt{:}v_1\texttt{\#}a_2\texttt{:}v_2\texttt{\#}a_3\texttt{:}v_3\texttt{\#}\cdots\texttt{\#}a_m\texttt{:}v_m\,\text{b̸}\text{b̸}\text{b̸}\cdots, \tag{1}$$

where $a_i$ and $v_i$ are integers represented as signed binary numbers. Roughly, each pair $a_i\,\texttt{:}\,v_i$ 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 (1) 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 $s' = \mathrm{update}(s, v, a)$. 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 `#`$a\,\texttt{:}\,v$ 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 `#`$a\,\texttt{:}\,v$ is added to the storage tape, all existing pairs of the form `#`$a\,\texttt{:}\,v'$ (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.

(a) `accept`: $M$ immediately halts and accepts.

(b) `reject`: $M$ immediately halts and rejects.

(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 $\Sigma$. 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 $s' = \mathrm{update}(s, v, a)$ of the RAM.

(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 $s' = \mathrm{update}(s, v, a)$ of the RAM.

(e) `if` $R_1 \circ R_2$ `goto` $\lambda$: 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. □

## 6 Simulation of a Turing machine by a RAM

We have seen that any language that is recognized by a RAM is also recognized by a Turing machine. We now prove that the converse holds: any language that is recognized by a Turing machine is also recognized by a RAM. In fact we demonstrate rather more. A *three-register RAM* is a random access machine, as defined earlier, 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 2.** *Let $L$ be a language over some alphabet $\Sigma$. If there is a Turing machine that recognizes $L$, then there is a three-register RAM that also recognizes $L$.*

*Proof.* Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ be a (standard, one-tape) Turing machine that recognizes 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 $\Gamma$ 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 $\Sigma$ 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.

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

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, 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`.)

```
                        [[code to read the input word and initialize registers]]
state0          :       [[code to simulate transitions from state q_0]]
state1          :       [[code to simulate transitions from state q_1]]
state2          :       [[code to simulate transitions from state q_2]]
                                    ⋮
state⟨|Q| − 1⟩  :       [[code to simulate transitions from state q_{|Q|−1}]]
```

Figure 2: Deciding the state

Aside from some preliminary code concerned with initialization, it will 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 blocks corresponding to the two halting states $q_{\text{accept}}$ and $q_{\text{reject}}$ of $M$ are special, and consist of a single `accept` or `reject` 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. 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

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

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.

You should check that the register 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.

```
   '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: Shifting left

At this point the RAM tests whether register 0 contains the special end-of-tape symbol $m - 1$; if so, the TM has attempted to move off the end of its tape, so the RAM undoes the above head-shifting sequence (but not the symbol overwriting). The final action of the RAM in simulating a single transition of $M$ is to jump unconditionally to the instruction labelled state$\langle i \rangle$, where $i$ is the index of the next state $q'$.

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 $P$ and hence the proof of the theorem. □

```
                 '$-1 := \langle m-1 \rangle$
    next_char  : read '0
                 if '$0 = 0$ goto end_of_input
                 '$-1 :=$ '$-1 * \langle m \rangle$
                 '$-1 :=$ '$-1 +$ '0
                 if $0 = 0$ goto next_char
 end_of_input  : '$1 := 0$
        shift  : if '$-1 = \langle$m$-1\rangle$ goto state0
                 ⟦code to shift head left: see Fig. 4⟧
                 if $0 = 0$ goto shift
```

Figure 5: Input and initialization

**Exercise:** [Hard!] Does the theorem remain true if "three-register RAM" is replaced by "two-register RAM"?

## 7 Conclusions

We have seen that the class of languages recognizable by Turing Machines is exactly the same as the class of languages recognizable by Random Access Machines.[2] Since we should readily believe that a RAM is capable of simulating any conceivable computer, this fact constitutes very strong evidence to support the Church-Turing Thesis. Moreover, the situation is exactly analogous for many other models that were developed to formalize the notion of computation: all of them have turned out to be equivalent to the Turing Machine in computational power. Thus the class of *Turing-recognizable* languages (i.e., languages recognized by TMs) is an extremely robust class. Note also that the simulations between Turing Machines and RAMs presented above preserve the halting property: i.e., the simulating machine halts whenever the original one does. Thus we get the same robustness for the class of *Turing-decidable* languages (i.e., languages recognized by a TM that halts on all inputs). For historical reasons, Turing-recognizable languages are often called *recursively enumerable*, and Turing-decidable languages are often called *recursive*.

[2]Actually, putting Theorems 1 and 2 together, we have seen that the apparently much more limited three-register RAM is capable of simulating an arbitrary RAM.