Notes on Random Access Machine (RAM) Model

Eric Ruppert

September 13, 2025

1 Definition of RAM Model

The RAM model is meant to describe architectures like the ones used in most modern computers (simplified and with unbounded space). The memory consists of an array of registers $R[0], R[1], \ldots$. There are also read-only input registers $I[0], I[1], \ldots$ used to store the input to a programme. Each register stores a natural number. Assume that, initially, $R[i] = 0$ for all i.

A programme for a RAM is similar to assembly language. It is a sequence of instructions with line numbers. (See Example 1 below.) Each instruction can do one of the following things:

- Write a constant into a register. E.g., $R[7] \leftarrow 5$.
- Copy the value from one register to another. E.g. $R[4] \leftarrow R[17]$ or $R[3] \leftarrow I[4]$.
- Either of the above can be done with indirect addressing. E.g., $R[R[2]] \leftarrow 7$ or $R[R[1]] \leftarrow I[R[1]]$.
- Add or subtract two registers and write the result in a third. E.g., $R[3] \leftarrow R[6] + R[9]$. Since registers can only store natural numbers; if the result of subtraction is negative, 0 is stored instead.
- Divide a register's value by 2 (like a bit shift to the left). E.g., halve R[8] (has the effect $R[8] \leftarrow \lfloor R[8]/2 \rfloor$).
- Unconditional jump. E.g., jump to line 6.
- Conditional jump. E.g., jump to line 8 if $R[9] > 0$. The conditions can be of the form $R[j] = 0$ or $R[j] > 0$.
- Halt the programme.

After completing the instruction on line ℓ, the RAM executes line $\ell + 1$ next, unless line ℓ is a jump statement.

Example 1. Here is a RAM algorithm that multiplies the numbers in $I[0]$ and $I[1]$ and leaves the answer in $R[0]$.

```
 1: R[1] ← I[0]
 2: R[2] ← I[1]
 3: jump to line 14 if R[2] = 0                    ▷ start of loop; exit when R[2] = 0
 4: ▷ loop invariant: at this point, I[0] · I[1] = R[0] + R[1] · R[2]
 5: R[3] ← R[2]
 6: halve R[3]
 7: R[3] ← R[3] + R[3]
 8: R[3] ← R[2] − R[3]                             ▷ lines 5–8 just set R[3] = R[2] mod 2
 9: jump to line 11 if R[3] = 0                    ▷ skip line 10 if R[2] is even
10: R[0] ← R[0] + R[1]
11: R[1] ← R[1] + R[1]
12: halve R[2]
13: jump to line 3                                 ▷ end of loop
14: halt
```

There are two plausible cost models for measuring time complexity of a RAM programme.

- Unit cost model: each instruction takes 1 unit of time.
- Bit cost model: the time to perform an instruction is proportional to the number of bits in the binary representation of the values that the instruction operates on.

For simplicity, we usually use the unit cost model. For most algorithms, if the input is of length n, the values stored in each register will be polynomial in n, so they have length $O(\log n)$, and thus the two cost measures only differ by a factor of $O(\log n)$ anyway.

Any programme you write in C or Java or Python gets compiled into machine language so that your computer can execute it. Machine language instructions are similar to the RAM instructions listed above. (The processor in your computer might have a slightly different instruction set than the RAM model, but it is not hard to see how to implement any of the instructions in your processor's instruction set using RAM instructions. Moreover, that implementation of your machine's instruction will take a constant number of steps on the RAM.) This means that any programme you write in any high-level language can be implemented to run on a RAM.

2 Simulating a Turing machine with a RAM

It is fairly easy to see how to simulate a Turing machine with a RAM. We first write a simulation of the TM in pseudocode. Assume the input string is initially in a vector of tape characters characters $A[1..n]$. We use an array $delta[q, a]$ indexed by states and characters to store the transition function of the TM. Each entry stores the triple $\delta(q, a)$.

```
high ← n                                          ▷ rightmost head position in vector A
A[0] ← ▷                                          ▷ add left-end marker on simulated tape
head ← 1                                          ▷ position of simulated TM on its tape
q ← q_0                                           ▷ state of simulated TM
while q is not a halting state do
    ⟨q', a', dir⟩ ← delta[q, A[head]]             ▷ look up TM's next move
    A[head] ← a'                                  ▷ update character on simulated tape
    q ← q'                                        ▷ update simulated state
    if dir = LEFT then                            ▷ move head left
        head ← head − 1
    else                                          ▷ move head right
        head ← head + 1
        if head > high then                       ▷ visiting a new simulated tape square
            append a blank character to end of A
            high ← high + 1
```

If this pseudocode is compiled into machine language using the RAM instructions, it will simulate the TM. Moreover, each step of the TM will be simulated using $O(1)$ RAM instructions, so if the TM takes t steps, the RAM will take $O(t)$ steps.

3 Simulating a RAM with a Turing Machine

We describe how to use a Turing Machine with five tapes to simulate a given RAM programme. It uses the following five tapes.

- Input tape: Stores the values initially stored in the input registers. This tape is never modified. We assume the input is written on this tape in the form $I[0]\#I[1]\#I[2]\#\cdots\#I[k]$, where each natural number $I[j]$ is written in binary. Thus the characters written on this tape are 0,1,#.
- Memory tape: stores a series of pairs (i, v) indicating that the value stored in $R[i]$ is v. Both i and v are written in binary. Thus the characters written on this tape are $0, 1, (,)$, X and comma. (The character X will be used in the simulation.)
- Address tape: used to store a memory address in binary.
- Value tape: used to store a value, written in binary, read from a memory address or to be written into a memory address.
- Work tape: to store a second value (useful for doing arithmetic like addition).

For each line of the RAM programme, we build a portion of the TM, using a few states. We describe, at a high level, how the TM accomplishes various RAM instructions.

To **lookup the value in** $I[j]$, the TM writes j (in binary) on the address tape and moves to the left end of the input tape. Then, it scans across the input tape, decrementing the number on the address tape each time it sees a # character on the input tape. When the value written on the address tape reaches 0, it copies the value found on the input tape on to the value tape.

To **lookup the value in** $R[j]$, the TM writes j (in binary) on the address tape and moves to the left end of the memory tape. Then, it scans across the memory tape. Each time it reaches a new pair (i, v), it checks character-by-character whether i on the memory tape matches j on the

address tape. If a match is found, then it copies v onto the value tape. If no match is found when it reaches a blank character on the memory tape, then it writes the value 0 on the value tape.

To **write the value on the value tape into** $R[j]$, the TM writes j onto the address tape and moves to the left end of the memory tape. Then, it scans across the memory tape. Each time it reaches a new pair (i, v), it checks character-by-character whether i on the memory tape matches j on the address tape. If a match is found, then it deletes the pair by overwriting it with X characters. When it reaches the blank symbol, it writes (j, v) at the end of the memory tape by copying the contents of the address and value tape onto the memory tape.

Indirect addressing is handled similarly. For example, reading $R[R[j]]$ is done by reading $R[j]$ and then copying the value from the value tape onto the address tape, and then reading the value $R[R[j]]$.

Addition and subtraction is done by writing the two operands onto the value tape and work tape and using the algorithm you learned in grade one to add or subtract the two numbers bit by bit. **Halving** just removes the last bit from the end of a value.

Thus each instruction (other than jumps) can be implemented using $O(1)$ states of the TM. When the simulation of instruction on line ℓ of the RAM programme is complete, the TM transitions into the first state used to simulate line $\ell + 1$ of the RAM programme. **Jump statements** instead go the the first state used to simulate the target line (after checking the condition, if any). Any RAM programme has a finite number of lines, so the number of states in the TM that simulates the RAM programme is also finite.

3.1 Time Overhead of Simulation

It remains to bound the number of steps needed for the simulation. Suppose a RAM programme takes t steps on an input of length n. (Here, we assume n is the total number of bits needed to represent all of the input values $I[0], I[1], \ldots, I[k]$ in binary.)

We need to bound the length of the tapes. We first bound the length of the binary representation of any index or value written into a register by the RAM programme. Let c be the length of the maximum constant value that appears in the RAM code (e.g., 5 and 42 are constants in the instruction $R[5] \leftarrow 42$).

Claim 1. *All indices used and all values written during the first r steps of the RAM programme are at most $c + n + r$ bits long.*

Proof. We prove the claim by induction on r. The claim is vacuously true for $r = 0$. Let $r > 0$. Assume the claim is true for the first $r - 1$ steps.

A value used as an index or written into memory in the rth step of the RAM programme is one of the following.

- A value read from an input register (at most n bits long).
- A constant that appears in the RAM programme (at most c bits long).
- A value written into a register in the first $r - 1$ steps (at most $c + n + r - 1$ bits long, by the induction hypothesis).
- The result of an arithmetic operation (add, subtract, halve) on values written during the first $r-1$ steps. The operands are at most $c+n+r-1$ bits long, by the induction hypothesis. Only addition can result in a longer number, and the result of an addition is at most one bit longer than its longest operand, so it is at most $c + n + r$ bits long.

This completes the proof of the claim. □

If the RAM runs for t steps, it writes at most t pairs on the memory tape and each pair is $O(n+t)$ bits long, by the claim; since c is a constant. Thus the TM uses $O(t(n+t))$ squares on each tape. It follows that the TM takes $O(t(n+t))$ steps to simulate each instruction of the RAM programme. So simulating the entire RAM programme takes $O(t^2(n+t))$ steps.

In particular, if the RAM programme runs in polynomial time, so does the TM simulation of it. (And if we convert the multitape TM simulation to a single-tape TM, it would still run in polynomial time.)

Thus, any programme that runs on your computer can be implemented on a RAM and then that RAM can be simulated on a TM (with some overhead). So, there is a programme that solves a problem in polynomial time on your computer if and only if there is a polynomial-time Turing machine for the problem.

(Actually, this assumes your computer has as much memory as needed by the programme so that it never runs out of memory; you can imagine attaching another hard drive to it whenever it uses up all of its memory.)