REF 5

Using the same compression technique as in the proof of Theorem 4.1.40, we can prove an analogous result for the space compression.

Theorem 4.1.42 (Linear space compression theorem) *For any function $s(n) > n$ and any real $\varepsilon > 0$ we have $Space(s(n)) = Space(\varepsilon s(n))$.*

Theorem 4.1.42 allows us to define

$$\mathbf{PSPACE} = \bigcup_{k=0} Space(n^k)$$

as the class of all languages that can be decided by MTM with a polynomial space bound.

4.2 Random Access Machines

Turing machines are an excellent computer model for studying fundamental problems of computing. However, the architecture of Turing machines has little in common with that of modern computers and their programming has little in common with programming of modern computers. The most essential clumsiness distinguishing a Turing machine from a real sequential computer is that its memory is not immediately accessible. In order to read a memory far away, all intermediate cells also have to be read. This difficulty is bridged by the **random access machine** model (RAM), introduced and analysed in this section, which has turned out to be a simple but adequate abstraction of sequential computers of the von Neumann type. Algorithm design methodologies for RAM and sequential computers are basically the same. Complexity analysis of algorithms and algorithmic problems for RAM reflect and predict the complexity analysis of programs to solve these problems on typical sequential computers. At the same time, surprisingly, if time and space requirements for RAM are measured properly, there are mutually very efficient simulations between RAM and Turing machines.

4.2.1 Basic Model

The memory of a RAM (see Figure 4.15a) consists of a data memory and a program memory. The data memory is an infinite **random access array of registers** $R_0, R_1, R_2, \ldots$ each of which can store an arbitrary integer. The register R_0 is called the **accumulator**, and plays a special role. The **program memory** is also a random access array of registers $P_0, P_1, P_2, \ldots$ each capable of storing an instruction from the instruction set shown in Figure 4.15b. A **control unit** (also called ALU, for 'arithmetical logical unit') contains two special registers, an **address counter** AC and an **instruction counter** IC. In addition, there are **input** and **output** units.

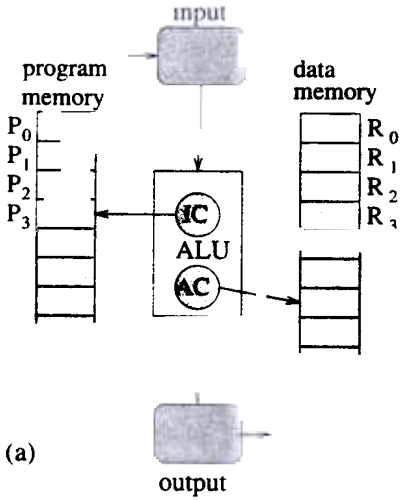
At the beginning of a computation all data memory and control unit registers are set to 0, and a program is stored in the program memory. A **configuration** of a RAM is described by a i-tuple $(i, i_1, n_{i_1}, \ldots, i_m, n_{i_m})$, where i is the content of IC, $i_1, \ldots, i_m$ are the addresses of the registers used up to that moment during the computation, and n_{i_k} is the current content of the register R_{i_k}.

The operand of an instruction is of one of the following three types:

$=i$	a constant i;
i	an address, referring to the register R_i,
$*i$	an indirect address; referring to the register $R_{c(R_i)}$

where $c(R_i)$ denotes the contents of the register R_i. (In Figure 4.15 R_{op} means i, if the operand has the form $=i$; R_{op} means R_i, if the operand is of the form i; R_{op} stands for $R_{c(R_i)}$, if the operand has the form $*i$.)

A computation of a RAM is a sequence of computation steps. Each step leads from one configuration



(a)

READ	operand	{ input ⟶ R_{op} }
WRITE	operand	{ R_{op} ⟶ output }
LOAD	operand	{ R_{op} ⟶ R_0 }
STORE	operand	{ R_0 ⟶ R_{op} }
ADD	operand	{ $R_0 + R_{op} \rightarrow R_0$ }
SUB	operand	{ $R_0 - R_{op} \rightarrow R_0$ }
MULT	operand	{ $R_0 * R_{op} \rightarrow R_0$ }
DIV	operand	{ $R_0 / R_{op} \rightarrow R_0$ }
JUMP	label	{ go to label }
JZERO	label	{ if $R_0 = 0$, then go to label }
JGZERO	label	{ if $R_0 > 0$, then go to label }
HALT		
ACCEPT		
REJECT		

(b)

Figure 4.15 Random access machine

to another. In each computational step a RAM executes the instruction currently contained in the program register $P_{c(IC)}$. In order to perform a nonjump instruction, its operand is stored in AC, and through AC the data memory is accessed, if necessary. The READ instruction reads the next input number; the WRITE instruction writes the next output number. The memory management instructions (LOAD and STORE), arithmetical instructions and conditional jump instructions use the accumulator R_0 as one of the registers. The second register, if needed, is specified by the contents of AC. After a nonjump instruction has been performed, the content of IC is increased by 1, and the same happens if the test in a jump instruction fails. Otherwise, the label of a jump instruction explicitly defines the new contents of IC.

A computation of a function is naturally defined for a RAM. The arguments have to be provided at the input, and a convention has to be adopted to determine their number. Either their number is a constant, or the first input integer determines the total number of inputs, or there is some special number denoting the last input.[7] Language recognition requires, in addition, an encoding of symbols by integers.

Figure 4.16 depicts RAM programs to compute two functions: (a) $f(n) = 2^{2^n}$ for $n > 0$; (b) F_n – the nth Fibonacci number. In both cases n is given as the only input. Fixed symbolic addresses, like N, i, F_{i-1}, F_i, *aux* and *temp*, are used in Figure 4.16 to make programs more readable. Comments in curly brackets serve the same purpose.

The instruction set of a RAM, presented in Figure 4.15, is typical but not the only one possible. Any 'usual' microcomputer operation could be added. However, in order to get relevant complexity results in the analysis of RAM programs, sometimes only a subset of the instructions listed in Figure 4.15 is allowed – namely, those without multiplication and division. (It will soon become clear why.) Such a model is usually called a RAM$^+$. To this new model the instruction SHIFT, with the semantics $R_0 \leftarrow \lfloor R_0/2 \rfloor$, is sometimes added.

Figure 4.17 shows how a RAM$^+$ with the SHIFT operation can be used to multiply two positive integers x and y to get $z = x \cdot y$ using the ordinary school method. In comments in Figure 4.17 k

[7]For example, the number 3 can denote the end of a binary vector.

while:

body:

(a)

Figure 4.16

1:
2:
3:
4:
5:
6:
7:
8:
9:
10:

Figure 4.17

stands for t
the algorith
SHIFT oper
If we u
time step a
complexity
$T_u(n) = \mathcal{O}(1$
because jus
out is to cc
instruction
one (multip
The second
perform an
of all the nu
needed for

```
        READ    N       {N ← n}
        LOAD    =2
while:  STORE   temp    {temp ← 2^(2^n − N)}
        LOAD    N
        JGZERO  body    {while N > 0 do}
        WRITE   temp
        HALT
body:   SUB     =1
        STORE   N       {N ← N − 1}
        LOAD    temp
        MULT    0       {R0 ← temp^2}
        JUMP    while
```

(a)

```
        READ    N       {N ← n}
        LOAD    =1
        STORE   i       {i ← 1}
        STORE   F_{i-1}
        STORE   F_i
while:  SUB     N
        JZERO   print   {while i < N do}
        LOAD    F_i
        STORE   aux
        ADD     F_{i-1}
        STORE   F_i     {F_i^new ← F_i + F_{i-1}}
        LOAD    aux
        STORE   F_{i-1} {F_{i-1}^new ← F_i}
        LOAD    i
        ADD     =1
        STORE   i
        JUMP    while
print:  WRITE   F_i
        HALT
```

(b)

Figure 4.16 RAM programs to compute (a) $f(n) = 2^{2^n}$; (b) F_n, the nth Fibonacci number.

```
 1:  READ    0    {R0 ← x}
 2:  STORE   x1   {x1 ← x}
 3:  READ    0    {R0 ← y}
 4:  STORE   y1   {y1 ← ⌊y / 2^k⌋}
 5:  SHIFT
 6:  STORE   y2   {y2 ← ⌊y / 2^(k+1)⌋}
 7:  ADD     y2   {R0 ← 2⌊y / 2^(k+1)⌋}
 8:  SUB     y1   {R0 ← 2⌊y / 2^(k+1)⌋ − ⌊y / 2^k⌋}
 9:  JZERO   13   {if the k-th bit of y is 0}
10:  LOAD    z    { zero at the start}
11:  ADD     x1
12:  STORE   z    {z ← x · (y mod 2^k)}
13:  LOAD    x1
14:  ADD     x1
15:  STORE   x1
16:  LOAD    y2
17:  JZERO   19   {if ⌊y / 2^k = 0⌋}
18:  JUMP    4
19:  WRITE   z
20:  HALT
```

Figure 4.17 Integer multiplication on RAM$^+$

stands for the number of cycles performed to that point. At the beginning $k = 0$. The basic idea of the algorithm is simple: if the kth right-most bit of y is 1, then $x2^k$ is added to the resulting sum. The SHIFT operation is used to determine, using the instructions numbered 4 to 9, the kth bit.

If we use complexity measures like those for Turing machines, that is, one instruction as one time step and one used register as one space unit, the **uniform complexity measures**, then the complexity analysis of the program in Figure 4.16, which computes $f(n) = 2^{2^n}$, yields the estimations $T_u(n) = \mathcal{O}(n) = \mathcal{O}(2^{\log n})$ for time and $S_u(n) = \mathcal{O}(1)$ for space. Both estimations are clearly unrealistic, because just to store these numbers one needs time proportional to their length $\mathcal{O}(2^n)$. One way out is to consider only the RAM$^+$ model (with or without the shift instruction). In a RAM$^+$ an instruction can increase the length of the binary representations of the numbers involved at most by one (multiplication can double it), and therefore the uniform time complexity measure is realistic. The second more general way out is to consider the **logarithmic complexity measures**. The time to perform an instruction is considered to be equal to the sum of the lengths of the binary representations of all the numbers involved in the instruction (that is, all operands as well as all addresses). The space needed for a register is then the maximum length of the binary representations of the numbers stored

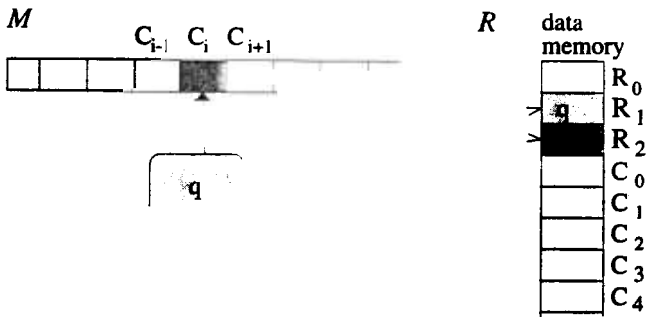


Figure 4.18 Simulation of a TM on a RAM

in that register during the program execution plus the length of the address of the register. The logarithmic space complexity of a computation is then the sum of the logarithmic space complexities of all the registers involved. With respect to these logarithmic complexity measures, the program in Figure 4.16a, for $f(n) = 2^{2^n}$, has the time complexity $T_l(n) = \Theta(2^n)$ and the space complexity $S_l(n) = \Theta(2^n)$, which corresponds to our intuition. Similarly, for the complexity of the program in Figure 4.17, to multiply two n-bit integers we get $T_u(n) = \Theta(n)$, $S_u(n) = \Theta(1)$, $T_l(n) = \Theta(n^2)$, $S_l(n) = \Theta(n)$, where the subscript u refers to the uniform and the subscript l to the logarithmic measures. In the last example, uniform and logarithmic measures differ by only a polynomial factor with respect to the length of the input. In the first example the differences are exponential.

4.2.2 Mutual Simulations of Random Access and Turing Machines

In spite of the fact that random access machines and Turing machines seem to be very different computer models, they can simulate each other efficiently.

Theorem 4.2.1 *A one-tape Turing machine $\mathcal{M}$ of time complexity $t(n)$ and space complexity $s(n)$ can be simulated by a RAM$^+$ of uniform time complexity $\mathcal{O}(t(n))$ and space complexity $\mathcal{O}(s(n))$, and with the logarithmic time complexity $\mathcal{O}(t(n)\lg t(n))$ and space complexity $\mathcal{O}(s(n))$.*

Proof: As mentioned in Section 4.1.3, we can assume without loss of generality that $\mathcal{M}$ has a one-way infinite tape. Data memory of a RAM$^+$ $\mathcal{R}$ simulating $\mathcal{M}$ is depicted in Figure 4.18. It uses the register R_1 to store the current state of $\mathcal{M}$ and the register R_2 to store the current position of the head of $\mathcal{M}$. Moreover, the contents of the jth cell of the tape of $\mathcal{M}$ will be stored in the register R_{j+2}, if $j \geq 0$.

$\mathcal{R}$ will have a special subprogram for each instruction of $\mathcal{M}$. This subprogram will simulate the instruction using the registers $R_0 - R_2$. During the simulation the instruction LOAD *2, with indirect addressing, is used to read the same symbol as the head of $\mathcal{M}$. After the simulation of an instruction of $\mathcal{M}$ is finished, the main program is entered, which uses registers R_1 and R_2 to determine which instruction of $\mathcal{M}$ is to be simulated as the next one. The number of operations which $\mathcal{R}$ needs to simulate one instruction of $\mathcal{M}$ is clearly constant, and the number of registers used is larger than the number of cells used by $\mathcal{M}$ by only a factor of 2. This gives the uniform complexity time and space estimations. The size of the numbers stored in registers (except in R_2) is bounded by a constant, because the alphabet of $\mathcal{M}$ is finite. This yields the $\mathcal{O}(s(n))$ bound for the logarithmic space complexity. The logarithmic factor for the logarithmic time complexity $\lg t(n)$, comes from the fact that the number representing the head position in the register R_2 may be as large as $t(n)$. □

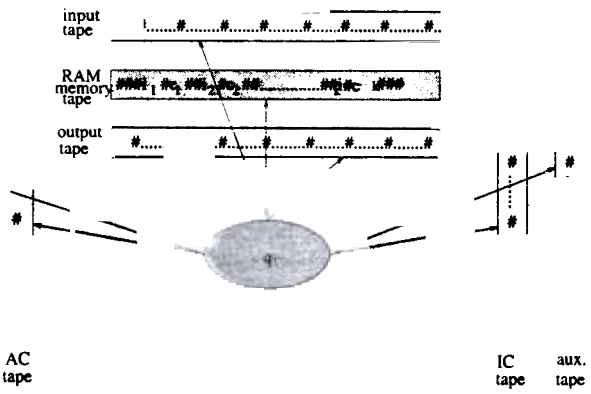


Figure 4.19 Simulation of a RAM on a TM

It is easy to see that the same result holds for a simulation of MTM on RAM$^+$, except that slightly more complicated mapping of k tapes into a sequence of memory registers of a RAM has to be used.

Exercise 4.2.2 *Show that the same complexity estimations as in Theorem 4.2.1 can be obtained for the simulation of k-tape MTM on RAM$^+$.*

The fact that RAM can be efficiently simulated by Turing machines is more surprising.

Theorem 4.2.3 *A RAM$^+$ of uniform time complexity $t(n)$ and logarithmic space complexity $s(n) \leq t(n)$ can be simulated by an MTM in time $\mathcal{O}(t^4(n))$ and space $\mathcal{O}(s(n))$. A RAM of logarithmic time complexity $t(n)$ and logarithmic space complexity $s(n)$ can be simulated by an MTM in time $\mathcal{O}(t^3(n))$ and space $\mathcal{O}(s(n))$*

can be
ith the

Proof: If a RAM$^+$ has uniform time complexity $t(n)$ and logarithmic space complexity $s(n) \leq t(n)$, then its logarithmic time complexity is $\mathcal{O}(t(n)s(n))$ or $\mathcal{O}(t^2(n))$, because each RAM$^+$ instruction can increase the length of integers stored in the memory at most by one, and the time needed by a Turing machine to perform a RAM$^+$ instruction is proportional to the length of the operands.

We show now how a RAM$^+$ $\mathcal{R}$ with logarithmic time complexity $t(n)$ and logarithmic space complexity $s(n)$ can be simulated by a 7-tape MTM $\mathcal{M}$ in time $\mathcal{O}(t^2(n))$. From this the first statement of the theorem follows

$\mathcal{M}$ will have a general program to pre-process and post-process all RAM instructions and a special group of instructions for each RAM instruction. The first read-only input tape contains the inputs of $\mathcal{R}$, separated from one another by the marker #. Each time a RAM instruction is to be simulated, the second tape contains the addresses and contents of all registers of $\mathcal{R}$ used by $\mathcal{R}$ up to that moment in the form

$$\#\#\#i_1\#c_1\#\#i_2\#c_2\#\#i_3\#\ldots\#\#i_{k-1}\#c_{k-1}\#\#i_k\#c_k\#\#\#,$$

where # is a marker; $i_1, i_2, \ldots, i_k$ are addresses of registers used until then, stored in binary form; and c_i is the current contents of the register R_{i_j}, again in binary form. The accumulator tape contains

the current contents of the register R_0. The AC tape contains the current contents of AC, and the IC tape the current value of IC. The output tape is used to write the output of $\mathcal{R}$, and the last tape is an auxiliary working tape (see Figure 4.19).

The simulation of a RAM instruction begins with the updating of AC and IC. A special subprogram of $\mathcal{M}$ is used to search the second tape for the register $\mathcal{R}$ has to work with. If the operand of the instruction has the form '$=j$', then the register is the accumulator. If the operand has the form 'j', then j is the current contents of AC, and one scan through the second tape, together with comparison of integers i_k with the number j written on the AC tape, is enough either to locate j and c_j on the second tape or to find out that the register R_j has not been used yet. In the second case, the string $\#\#j\#0$ is added at the end of the second tape just before the string $\#\#\#$. In the case of indirect addressing, '$*j$', two scans through the second tape are needed. In the first, the register address j is found, and the contents of the corresponding register, c_j, are written on the auxiliary tape. In the second the register address c_j is found in order to get c_{c_j}. (In the case j or c_j is not found as a register address, we insert on the second tape a new register with 0 as its contents.)

In the case of instructions that use only the contents of the register stored on the second tape, that is, WRITE and LOAD or an arithmetic instruction, these are copied on either the output tape or the accumulator tape or the auxiliary tape.

Simulation of a RAM instruction for changing the contents of a register R_i found on the second tape is a little bit more complicated. In this case $\mathcal{M}$ first copies the contents of the second tape after the string $\#\#i\#c_i\#$ on the auxiliary tape, then replaces $c_i\#$ with the contents of the AC tape, appends $\#$, and copies the contents of the auxiliary tape back on to the second memory tape. In the case of arithmetical instructions, the accumulator tape (with the content of R_0) and the auxiliary tape, with the second operand, are used to perform the operation. The result is then used to replace the old contents of the accumulator tape.

The key factor for the complexity analysis is that the contents of the tapes can never be larger than $\mathcal{O}(s(n))$. This immediately implies the space complexity bound. In addition, it implies that the scanning of the second tape can be done in time $\mathcal{O}(t(n))$. Simulations of an addition and a subtraction also require only time proportional to the length of the arguments. This provides a time bound $\mathcal{O}(t^2(n))$. In the case of multiplication, an algorithm similar to that described in Figure 4.17 can be used to implement a multiplication in $\mathcal{O}(t^2(n))$ time. (Actually the SHIFT instruction has been used only to locate the next bit of one of the arguments in the constant time.) This is easily implementable on a TM. A similar time bound holds for division. This yields in total a $\mathcal{O}(t^3(n))$ time estimation for the simulation of a RAM with logarithmic time complexity $t(n)$. □

Exercise 4.2.4 *Could we perform the simulation shown in the proof of Theorem 4.2.3 without a special tape for IC?*

4.2.3 Sequential Computation Thesis

Church's thesis concerns basic idealized limitations of computing. In this chapter we present two quantitative variations of Church's thesis: the **sequential computation thesis** (also called the **invariance thesis**) and the **parallel computation thesis**. Both deal with the robustness of certain quantitative aspects of computing: namely, with mutual simulations of computer models.

Turing machines and RAM are examples of computer models, or computer architectures (in a modest sense). For a deeper understanding of the merits, potentials and applicability of various