Turing Machines

CSE 211 (Theory of Computation)

Atif Hasan Rahman

Assistant Professor
Department of Computer Science and Engineering
Bangladesh University of Engineering & Technology

Adapted from slides by

Dr. Muhammad Masroor Ali

Extensions to the Basic Turing Machine

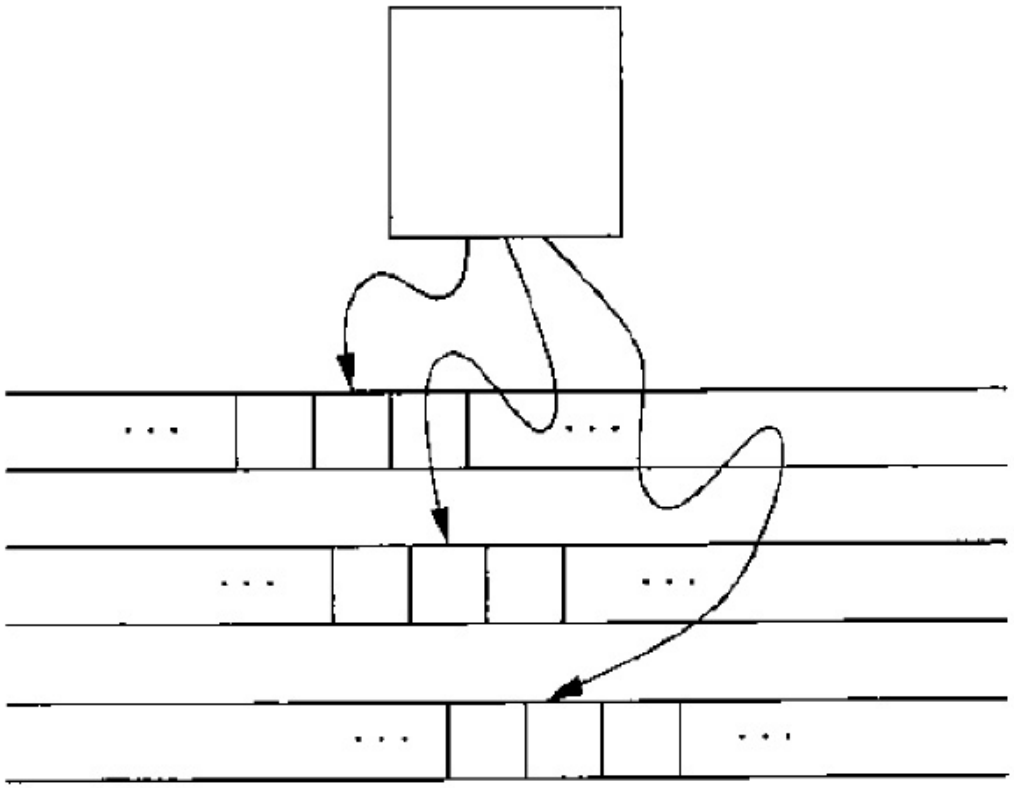
- We shall see certain computer models that. are related to Turing machines and have the same language-recognizing power as the basic model of a TM with which we have been working.
- One of these, the multitape Turing machine, is important because it is much easier to see how a multitape TM can simulate real computers (or other kinds of Turing machines), compared with the single-tape model we have been studying.
- Yet the extra tapes add no power to the model, as far as the ability to accept languages is concerned.

Extensions to the Basic Turing Machine

- We then consider the nondeterministic Turing machine, an extension of the basic model that is allowed to make any of a finite set of choices of move in a given situation.
- This extension also makes “programming” Turing machines easier in some circumstances, but adds no language-defining power to the basic model.

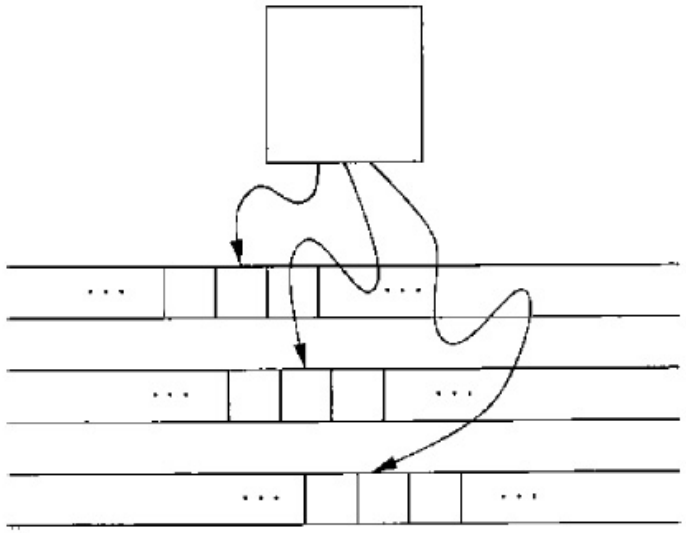
Multitape Turing Machines

A multitape TM is as suggested in the figure.



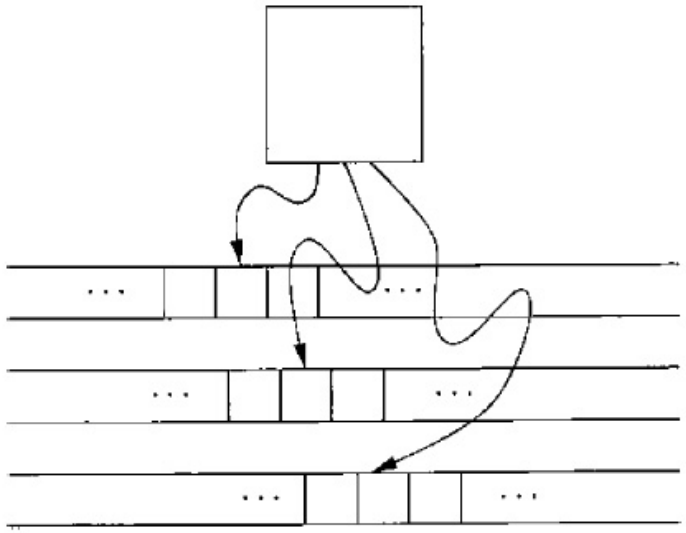
A multitape Turing machine

Multitape Turing Machines



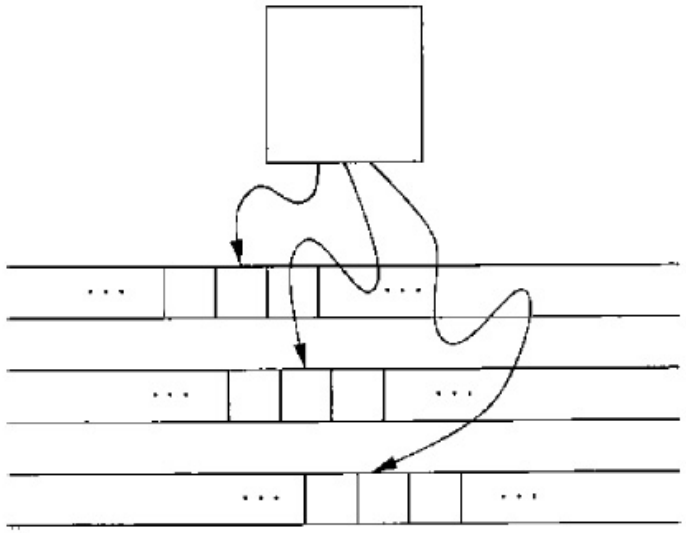
- The device has a finite control (state), and some finite number of tapes.
- Each tape is divided into cells, and each cell can hold any symbol of the finite tape alphabet.

Multitape Turing Machines



- As in the single-tape TM, the set of tape symbols includes a blank.
- The tape symbols has a subset called the input symbols, of which the blank is not a member.
- The set of states includes an initial state and some accepting states.

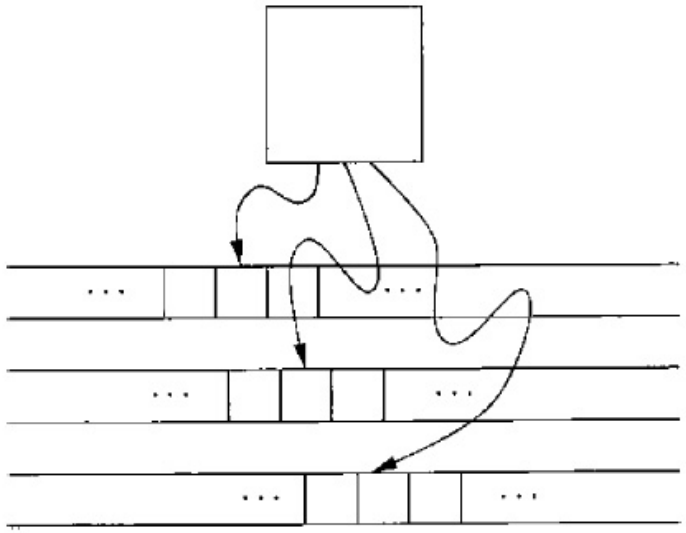
Multitape Turing Machines



Initially:

1. The input, a finite sequence of input symbols, is placed on the first tape.
2. All other cells of all the tapes hold the blank.
3. The finite control is in the initial state.

Multitape Turing Machines



4. The head of the first tape is at the left end of the input.
5. All other tape heads are at some arbitrary cell.
   - Since tapes other than the first tape are completely blank, it does not matter where the head is placed initially.
   - All cells of these tapes “look” the same.

Multitape Turing Machines

- A move of the multitape TM depends on the state and the symbol scanned by each of the tape heads.
- In one move, the multitape TM does the following:
  1. The control enters a new state, which could be the same as the previous state.
  2. On each tape, a new tape symbol is written on the cell scanned.
     Any of these symbols may be the same as the symbol previously there.
  3. Each of the tape heads makes a move, which can be either left, right, or stationary.
     The heads move independently, so different heads may move in different directions, and some may not move at all.

Multitape Turing Machines

- Formal notation of transition rules, is a straightforward generalization of the notation for the one-tape TM.
- Additionally, directions are now indicated by a choice of L, R, or S.
- For the one- tape machine, we did not allow the head to remain stationary, so the S option was not present.
- Multitape Turing machines, like one-tape TM's, accept by entering an accepting state.

Equivalence of One-Tape and Multitape TM's

- The recursively enumerable languages are defined to be those accepted by a one-tape TM.
- Surely, multitape TM's accept all the recursively enumerable languages, since a one-tape TM is a multitape TM.
- However, are there languages that are not recursively enumerable, yet are accepted by multitape TM's?
- The answer is "no,".
- We prove this fact by showing how to simulate a multitape TM by a one-tape TM.

Equivalence of One-Tape and Multitape TM's

- The recursively enumerable languages are defined to be those accepted by a one-tape TM.
- Surely, multitape TM's accept all the recursively enumerable languages, since a one-tape TM is a multitape TM.
- However, are there languages that are not recursively enumerable, yet are accepted by multitape TM's?
- The answer is "no,".
- We prove this fact by showing how to simulate a multitape TM by a one-tape TM.

Equivalence of One-Tape and Multitape TM's

Theorem

Every language accepted by a multitape TM is recursively enumerable.

PROOF: The proof is suggested by figure.

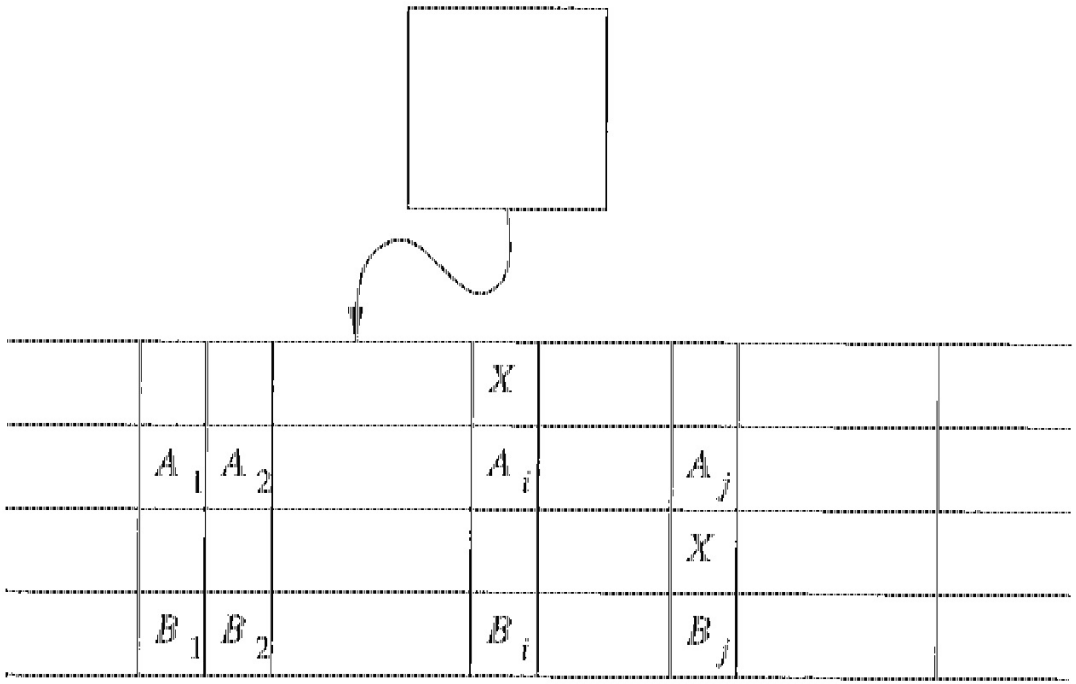
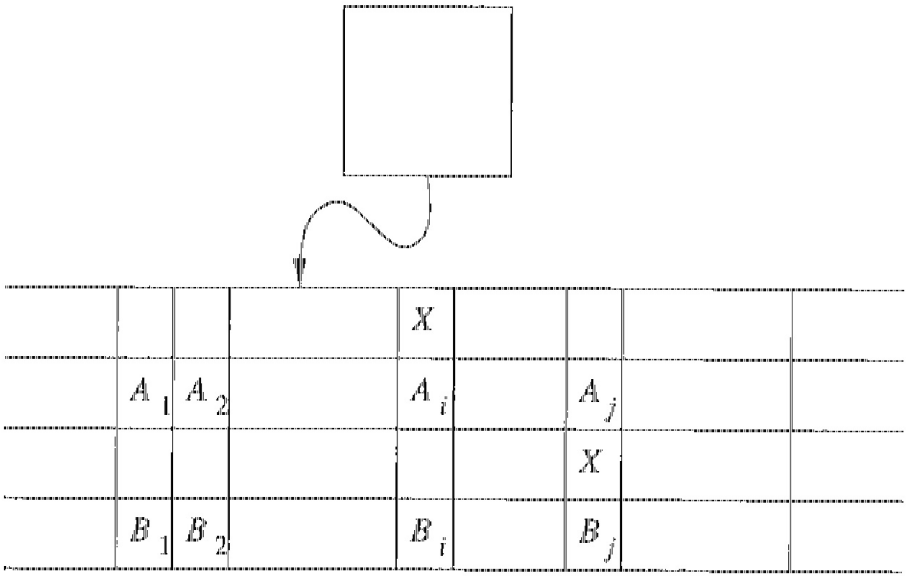


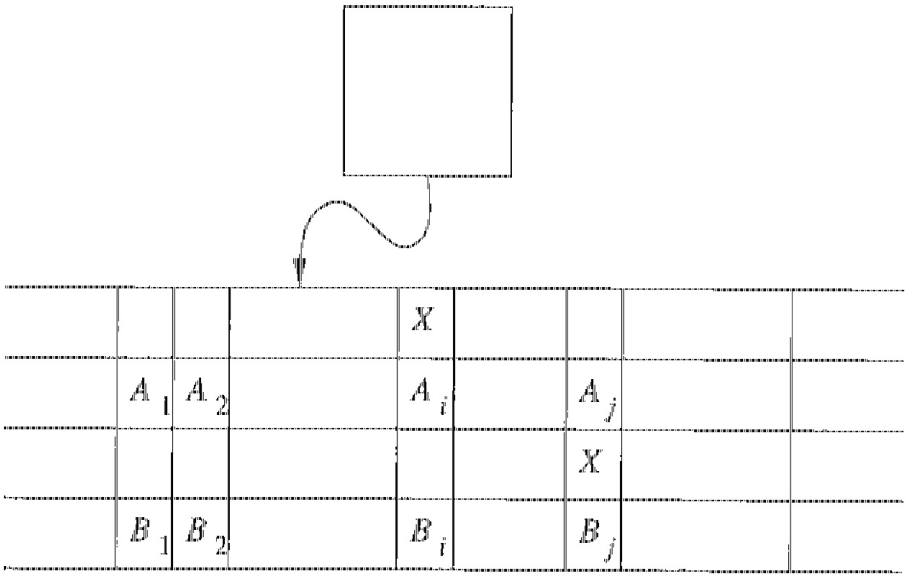
Figure 8.17: Simulation of a two-tape Turing machine by a one-tape Turing machine

Equivalence of One-Tape . . . TM's



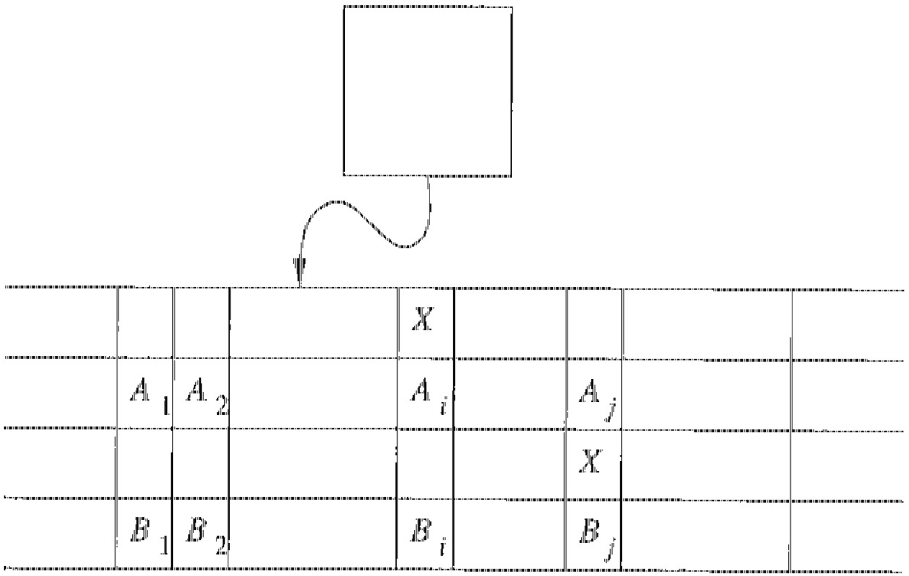
- Suppose language L is accepted by a k-tape TM M.
- We simulate M with a one-tape TM N whose tape we think of as having $2k$ tracks.

Equivalence of One-Tape . . . TM's



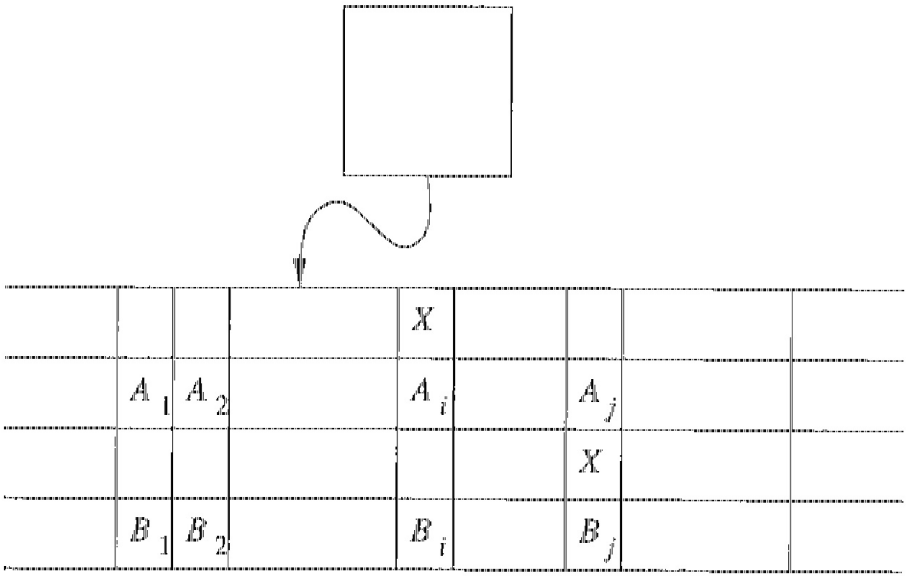
- Half these tracks hold the tapes of M.
- The other half of the tracks each hold only a single marker that indicates where the head for the corresponding tape of M is currently located.
- The TM in figure assumes $k = 2$.

Equivalence of One-Tape . . . TM's



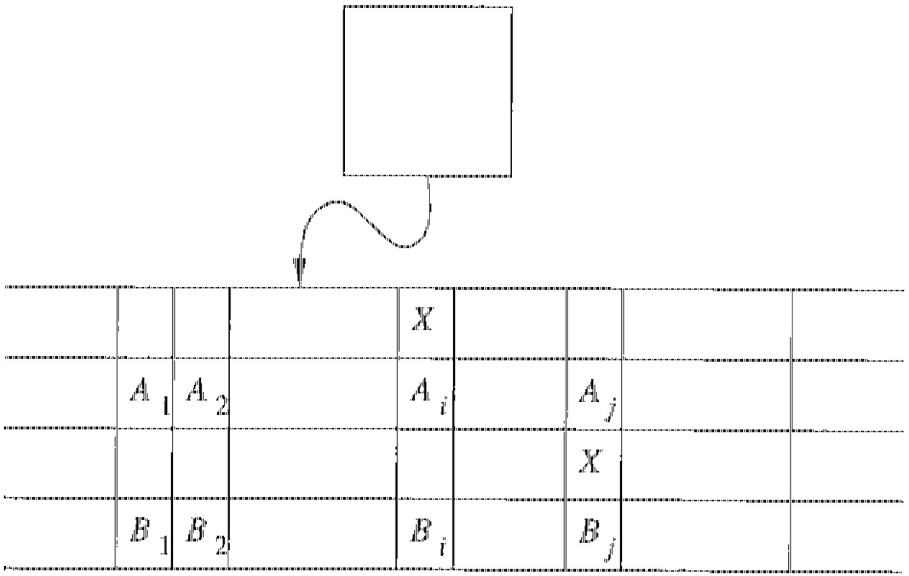
- The second and fourth tracks hold the contents of the first and second tapes of *M*.
- Track 1 holds the position of the head of tape 1, and track 3 holds the position of the second tape head.

Equivalence of One-Tape . . . TM's



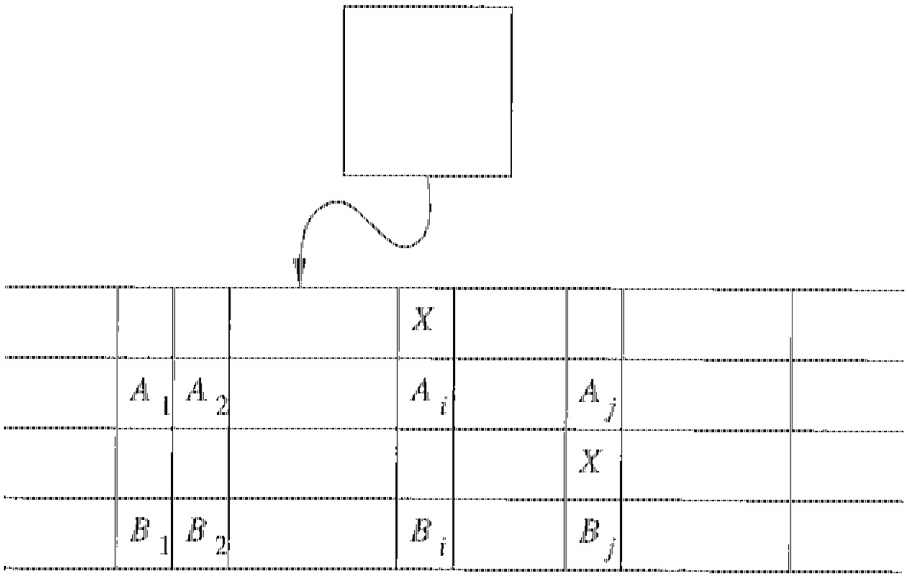
- To simulate a move of M, N's head must visit the k head markers.
- So that N not get lost, it must remember how many head markers are to its left at all times.
- That count is stored as a component of N's finite control.

Equivalence of One-Tape . . . TM's



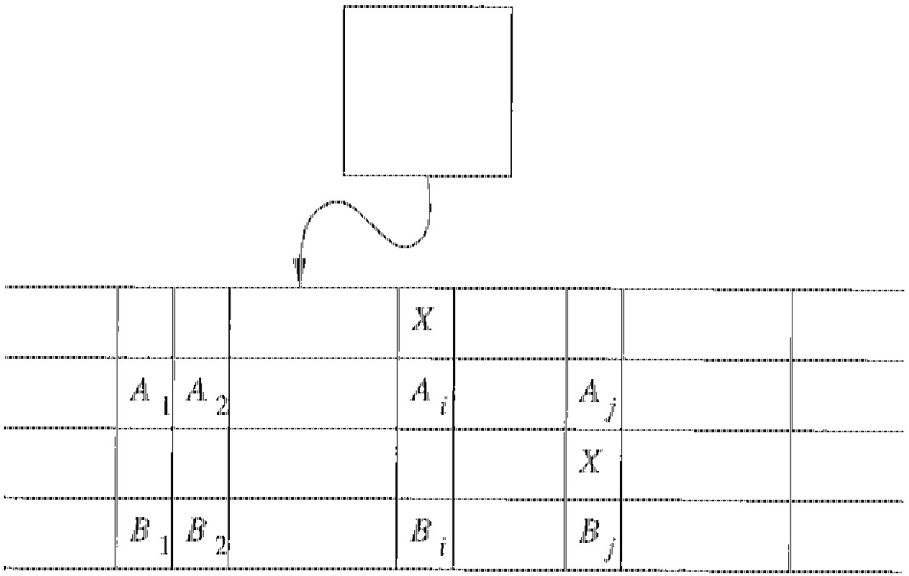
- After visiting each head marker and storing the scanned symbol in a component of its finite control, *N* knows what tape symbols are being scanned by each of *M*'s heads.
- *N* also knows the state of *M*, which it stores in *N*'s own finite control.
- Thus, *N* knows what move *M* will make.

Equivalence of One-Tape . . . TM's



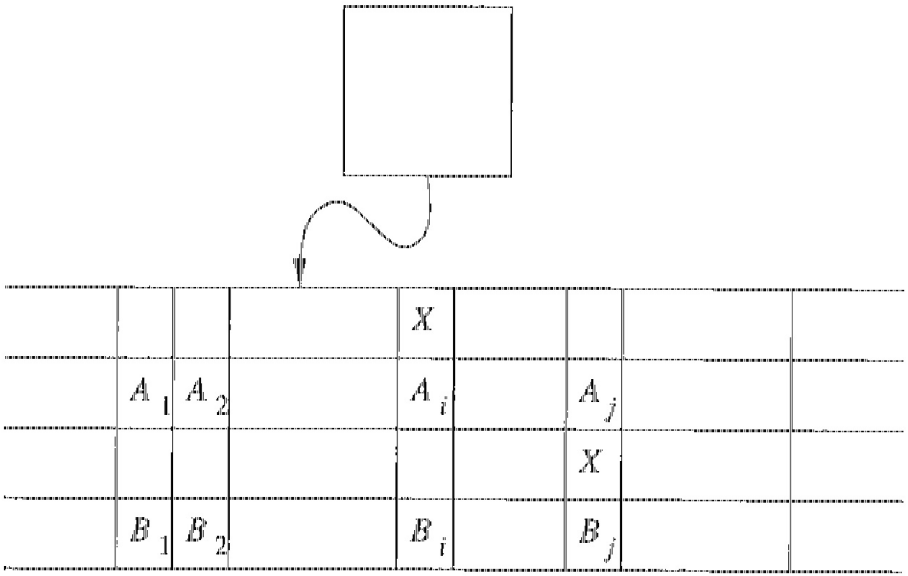
- *N* now
  - revisits each of the head markers on its tape,
  - changes the symbol in the track representing the corresponding tapes of *M*,
  - and moves the head markers left or right, if necessary.

Equivalence of One-Tape . . . TM's



- Finally, *N* changes the state of *M* as recorded in its own finite control.
- At this point, *N* has simulated one move of *M*.

Equivalence of One-Tape . . . TM's



- We select as *N*'s accepting states all those states that record *M*'s state as one of the accepting states of *M*.
- Thus, whenever the simulated *M* accepts, *N* also accepts, and *N* does not accept otherwise.

Running Time to Simulate Multitape TM's

Theorem

The time taken by the one-tape TM N to simulate n moves of the k-tape TM M is $O(n^2)$

PROOF:

- After n moves of M the tape head markers cannot have separated by more than $2n$ cells
- If N starts at the leftmost marker, it has to move no more than $2n$ cells right to find all the head markers
- Again, moving head markers left or right requires no more than $2n$ moves left plus at most $2k$ moves to reverse direction and write X in the cell to the right, if needed
- The number of moves by is no more than $4n + 2k$
- Since k is a constant, this is $O(n^2)$

Nondeterministic Turing Machines

- A *nondeterministic* Turing has a transition function δ such that for each state q and tape symbol X, $\delta\ (q, X)$ is a set of triples

$$\{(q_1, Y_1, D_1), (q_2, Y_2, D_2), \ldots, (q_k, Y_k, D_k)\}$$

  where k is any finite integer.
- The NTM can choose, at each step, any of the triples to be the next move.
- It cannot, however, pick a state from one, a tape symbol from another, and the direction from yet another.

Nondeterministic Turing Machines

- The language accepted by an NTM M is defined in the expected manner, in analogy with the other nondeterministic devices, such as NFA's and PDA's.
- That is, M accepts an input w
  - if there is any sequence of choices of move that leads from the initial ID with w as input,
  - to an ID with an accepting state.
- The existence of other choices that do not lead to an accepting state is irrelevant, as it is for the NFA or PDA.

Nondeterministic Turing Machines

- The NTM's accept no languages not accepted by a deterministic TM (DTM).
- The proof involves showing that for every NTM M_N, we can construct a DTM M_D that explores the ID's that M_N can reach by any sequence of its choices.
- If M_D finds one that has an accepting state, then M_D enters an accepting state of its own.
- M_D must be systematic, putting new ID's on a queue, rather them a stack, so that after some finite time M_D has simulated all sequences of up to k moves of for $k = 1, 2, \ldots$.

Nondeterministic Turing Machines

Theorem

If M_N is a nondeterministic Turing machine, then there is a deterministic Turing machine M_D such that $L(M_N) = L(M_D)$.

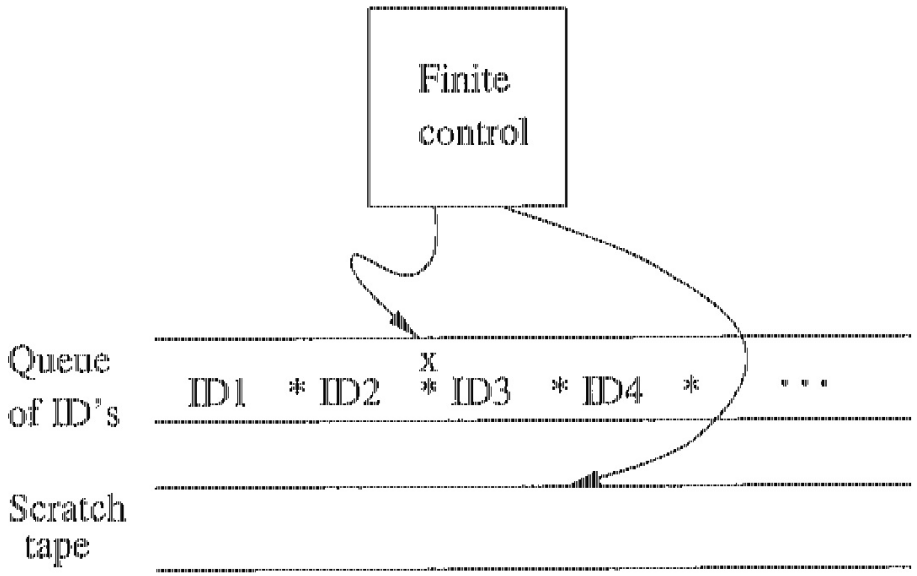
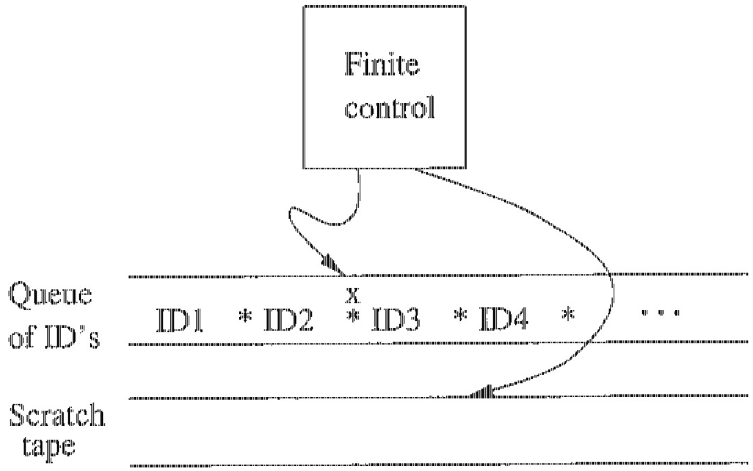


Figure 8.18: Simulation of an NTM by a DTM

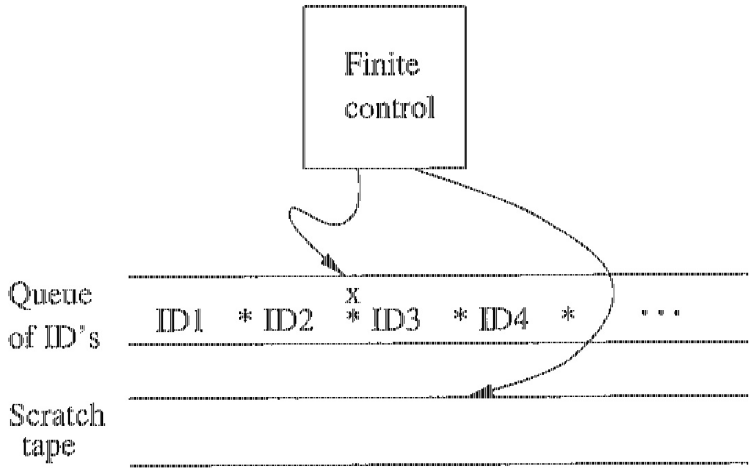
PROOF: M_D will be designed as a multitape TM, sketched in figure.

Nondeterministic Turing Machines



- The first tape of M_D holds a sequence of ID's of M_N, including the state of M_N.
- One ID of M_N is marked as the "current" ID, whose successor ID's are in the process of being discovered.
- In figure, the third ID is marked by an x along with the inter-ID separator, which is the $*$.
- All ID's to the left of the current one have been explored and can be ignored subsequently.

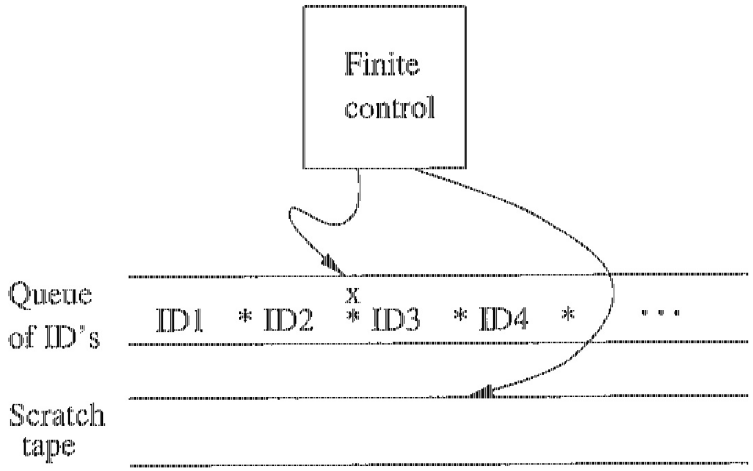
Nondeterministic Turing Machines



To process the current ID, M_D does the following:

1. M_D examines the state and scanned symbol of the current ID.
   - Built into the finite control of M_D is the knowledge of what choices of move M_N has for each state and symbol.
   - If the state in the current ID is accepting, then M_D accepts and simulates M_N no further.

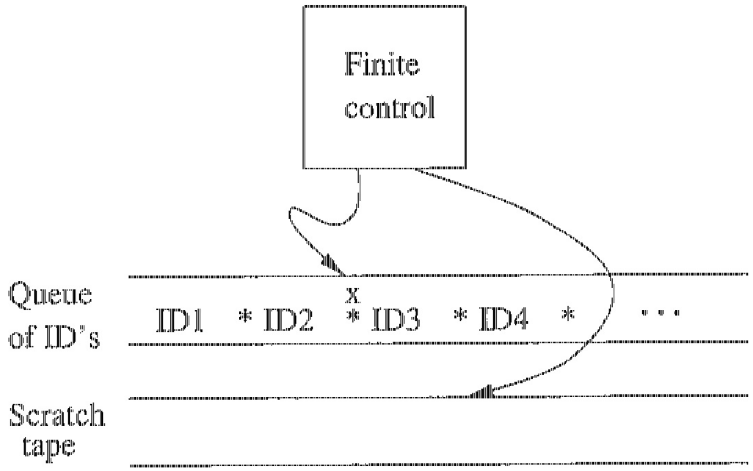
Nondeterministic Turing Machines



To process the current ID, M_D does the following:

2. However, if the state is not accepting, and the state symbol combination has k moves.
   - Then M_D uses its second tape to copy the ID and then make k copies of that ID at the end of the sequence of ID's on tape 1.

Nondeterministic Turing Machines



To process the current ID, M_D does the following:

3. M_D modifies each of those k ID's according to a different one of the k choices of move that M_N has from its current ID.
4. M_D returns to the marked, current ID, erases the mark, and moves the mark to the next ID to the right.
   - The cycle then repeats with step (1).

Nondeterministic Turing Machines

- Suppose, m is the maximum number of choices M_N has in any configuration
- Then there is one initial ID of M_N, at most m IDs that M_N can reach after one move, at most m^2 IDs M_N can reach after two moves, and so on
- After n moves, M_N can reach at most $1 + m + m^2 + \cdots + m^n$ IDs
- This number is at most nm^n IDs
- Since M_D explores IDs in a breadth first manner, if M_N reaches an accepting state, M_D will eventually accept. So,

$$L(M_N) = L(M_D)$$

- But M_D may need exponentially more moves compared to M_N

Restricted Turing Machines

- We have seen seeming generalizations of the Turing machine that do not add any language-recognizing power.
- Now, we shall consider some examples of apparent restrictions on the TM that also give exactly the same language-recognizing power.

Turing Machines With Semi-infinite Tapes

- We have allowed the tape head of a Turing machine to move either left or right from its initial position.
- It is only necessary that the TM's head be allowed to move within the positions at and to the right of the initial head position.
- In fact, we can assume the tape is *semi-infinite*, that is, there are no cells to the left of the initial head position.
- A TM with a semi-infinite tape can simulate one whose tape is, like our original TM model, infinite in both directions.

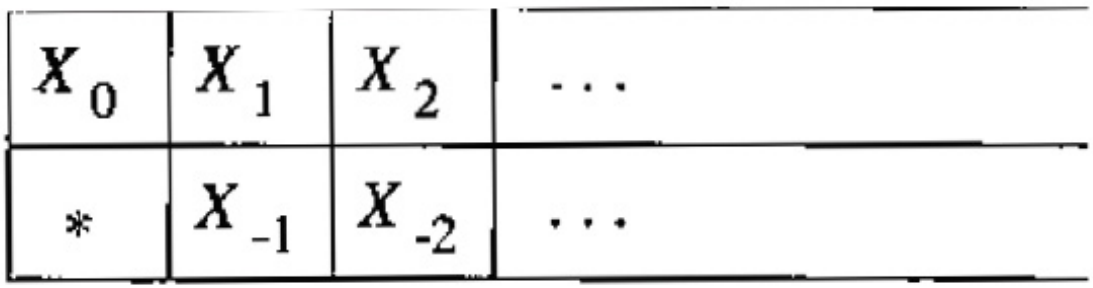
Turing Machines With Semi-infinite Tapes

X_0	X_1	X_2	$\cdots$
*	X_{-1}	X_{-2}	$\cdots$

A semi-infinite tape can simulate a two-way infinite tape

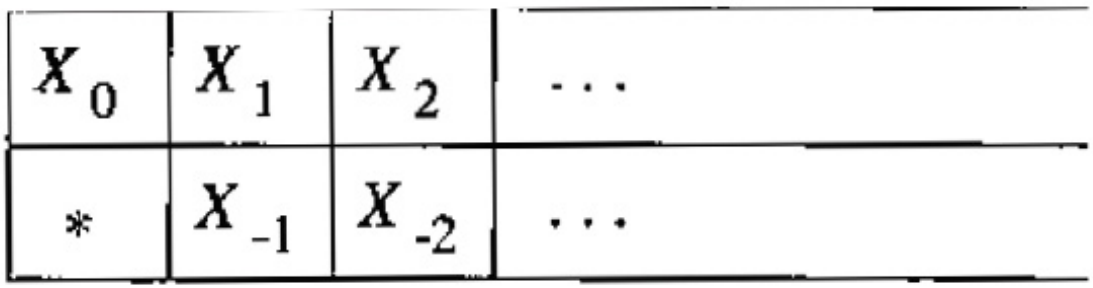
- The trick behind the construction is to use two tracks on the semi-infinite tape.
- The upper track represents the cells of the original TM that are at or to the right of the initial head position.
- The lower track represents the positions left of the initial position, but in reverse order.

Turing Machines With Semi-infinite Tapes



- The upper track represents cells $X_0, X_1, \ldots,$ where X_0 is the initial position of the head.
- X_1, X_2, and so on, are the cells to its right.
- Cells X_{-1}, X_{-2}, and so on, represent cells to the left of the initial position.

Turing Machines With Semi-infinite Tapes

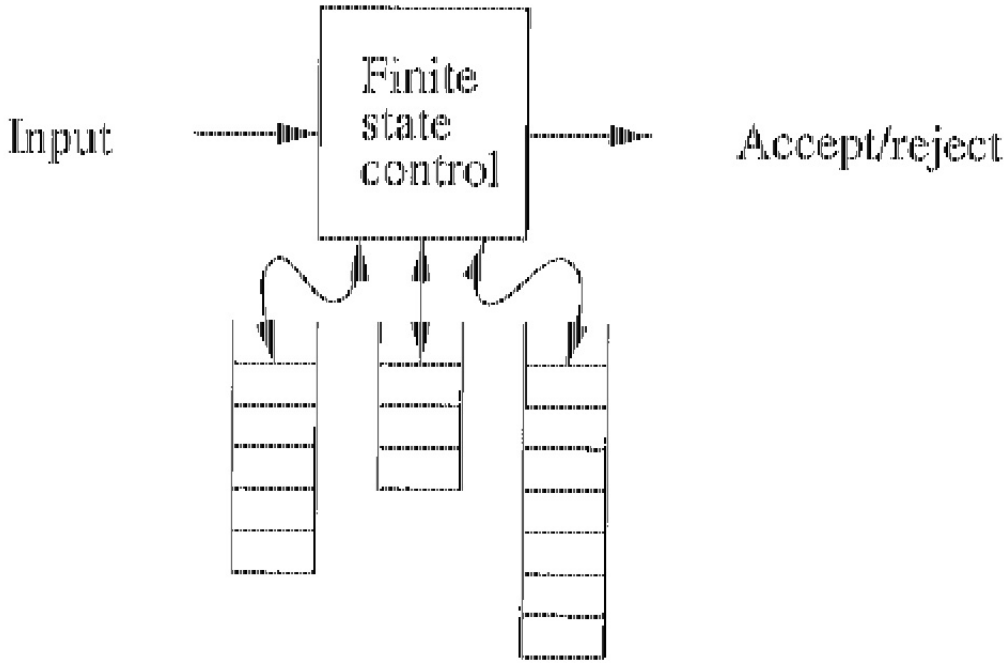


- Notice the $*$ on the leftmost cell's bottom track.
- This symbol serves as an endmarker and prevents the head of the semi-infinite TM from accidentally falling off the left end of the tape.

Multistack Machines

- A Turing machine can accept languages not accepted by any PDA with one stack.
- But if we give the PDA two stacks, then it can accept any language that a TM can accept.
- A k-stack machine is a deterministic PDA with k stacks.
- If a language L is accepted by a Turing machine, then L is accepted by a two-stack machine.

Multistack Machines



- It has a finite control.
- It obtains its input, like the PDA does, from an input source, not from the tape, as the TM does.
- It has finite stack alphabet, which it uses for all its stack.

Counter Machines

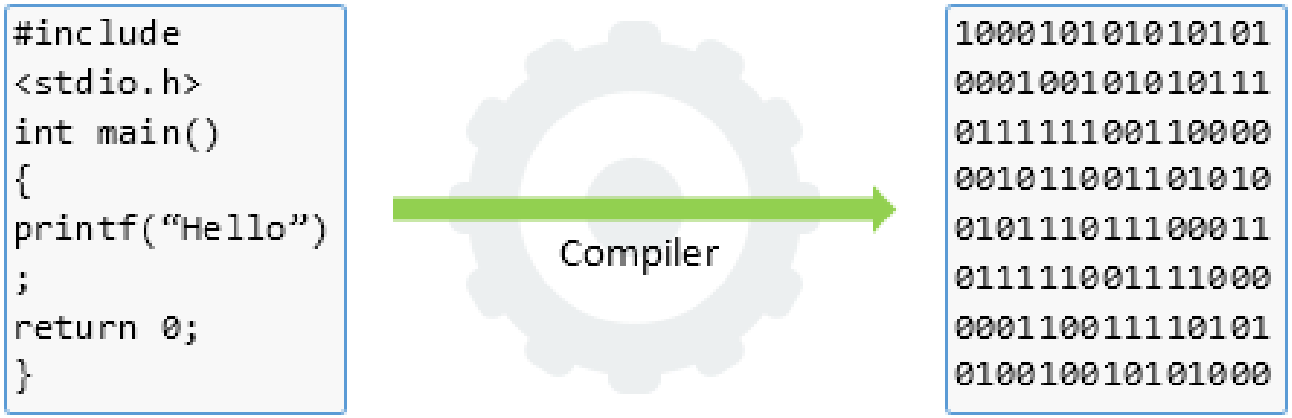
- The counter machine has the same structure as the multistack machine but in place of each stack is a counter.
- Counters hold any nonnegative integer, but we can only distinguish between zero and nonzero counters.
- That is, the move of the counter machine depends on its state, input symbol, and which, if any, of the counters are zero.
  a. Change state.
  b. Add or subtract 1 from any of its counters, independently. However, a counter is not allowed to become negative, so it cannot subtract 1 from a counter that is currently 0.
- Every recursively enumerable language is accepted by a two counter machine.

Turing Machines and Computers

- Let us compare the Turing machine and the common computer that we use daily.
- Since the notion of "a common computer" is not well defined mathematically, the arguments in this section are informal.
- A computer can simulate a Turing machine.
- A Turing machine can simulate a computer, and can do so in an amount of time that is at most some polynomial in the number of steps taken by the computer.

Computers

- Compiler converts programs to executable codes.

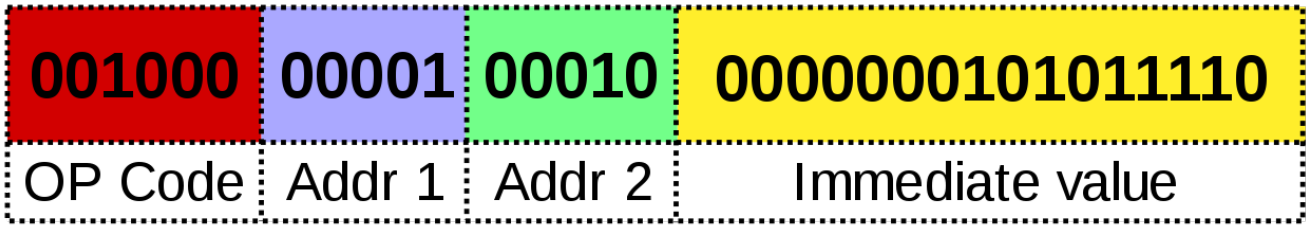


Instruction types

- Arithmetic and logic operations
  - Add, subtract, multiply, or divide, etc. (possibly setting zero, carry flags in a status register)
  - And, or, not, etc.
- Data handling and memory operations
  - Copy data from a memory location to a register, or vice versa
  - Read and write data from hardware devices
- Control flow operations
  - Jump to another location in the program and execute instructions there
  - Jump to another location if a certain condition holds
  - Call another block of code

Instructions

MIPS32 Add Immediate Instruction



Equivalent mnemonic: **addi $r1, $r2, 350**

Control flow

- C code

```
if (i == j)
    f = g+h;
else
    f = g-h;
```

Register assignment
- f ($s0), g ($s1), h ($s2), i ($s3), j ($s4)

- MIPS assembly code

```
    bne $s3, $s4, Else # go to Else if i !=j
    add $s0, $s1, $s2  # f = g + h
    j   Exit           # go to Exit
Else: sub $s0, $s1, $s2  # f = g - h
Exit:
```

Assembler calculates addresses

- Conditional branch: bne, beq
- Unconditional branch: j

Control flow

- Example Source Statement

```
sum = 0;
for (i = 0; i < 100; i++)
    sum += a[i];
```

- Translation into MIPS instructions, assuming `sum`, `i`, and the starting address of `a` are in `$2`-`$4`, respectively.

```
    and   $2,$2,$0    # sum = 0;
    or    $3,$0,$0    # i = 0;
Loop:
    sll   $5,$3,2     # tmp = i*4;
    addu  $5,$5,$4    # tmp = tmp + &a;
    lw    $5,0($5)    # load a[i] into tmp;
    addu  $2,$2,$5    # sum += tmp;
    addiu $3,$3,1     # i++;
    slti  $5,$3,100   # test i < 100
    bne   $5,$0,Loop  # if true goto Loop;
```

Simulating a Computer by a Turing Machine

Some assumptions on how a typical computer operates.

- First, we shall suppose that the storage of a computer consists of an indefinitely long sequence of words, each with an address. We shall not put a limit on the length of a given word. Addresses will be assumed to be integers 0,1,2, and so on.
- Also, in a real computer, there would be a limit on the number of words in "memory," but we shall assume there is no limit to the number of words.
- We assume that the program of the computer is stored in some of the words of memory. These words each represent a simple instruction, as in the machine or assembly language of a typical computer.

Simulating a Computer by a Turing Machine

- We assume that each instruction involves a limited (finite) number of words, and that each instruction changes the value of at most one word.
- A typical computer has registers, which are memory words with especially fast access. Often, operations such as addition are restricted to occur in registers. We shall not make any such restrictions, but will allow any operation to be performed on any word.

Simulating a Computer by a Turing Machine

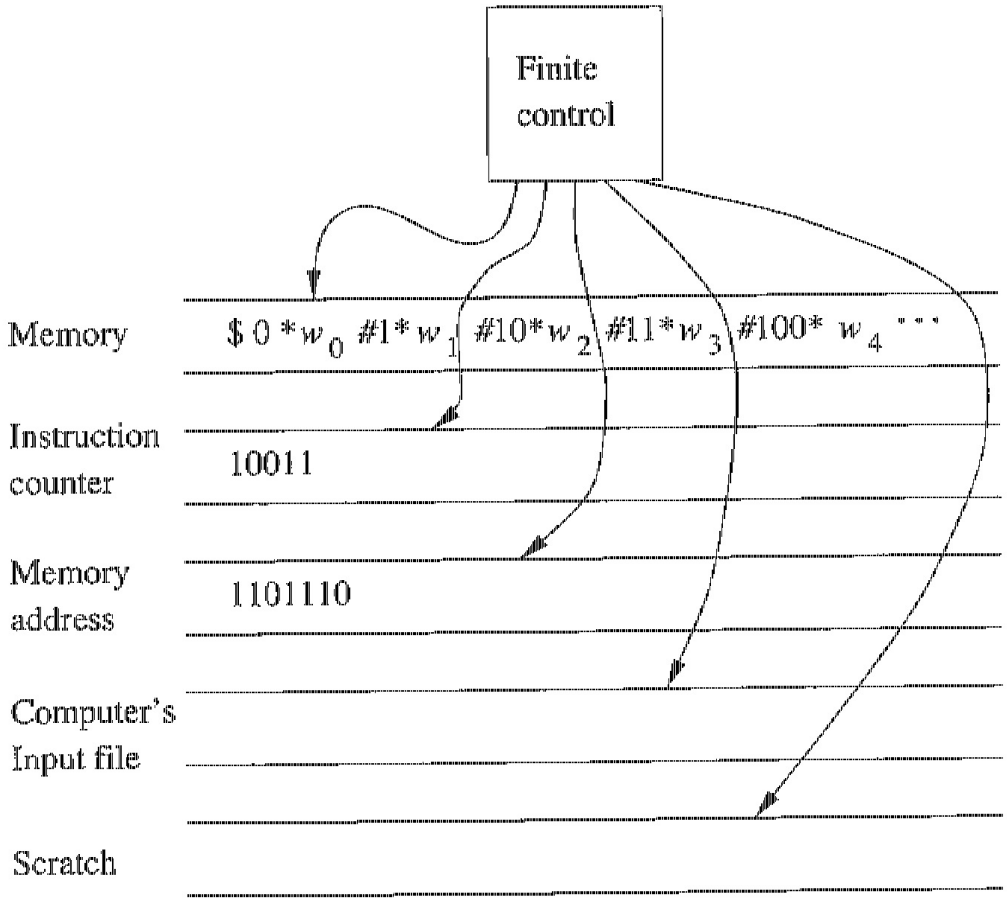


Figure 8.22: A Turing machine that simulates a typical computer

- This TM uses several tapes. It could be converted to a one-tape TM.

Simulating a Computer by a Turing Machine

- The first tape represents the entire memory of the computer.
- Addresses of memory words, in numerical order, alternate with the contents of those memory words.
- Both addresses and contents are written in binary.
- The marker symbols * and # are used to make it easy to find the ends of addresses and contents.
- Another marker, $, indicates the beginning of the sequence of addresses and contents.

Simulating a Computer by a Turing Machine

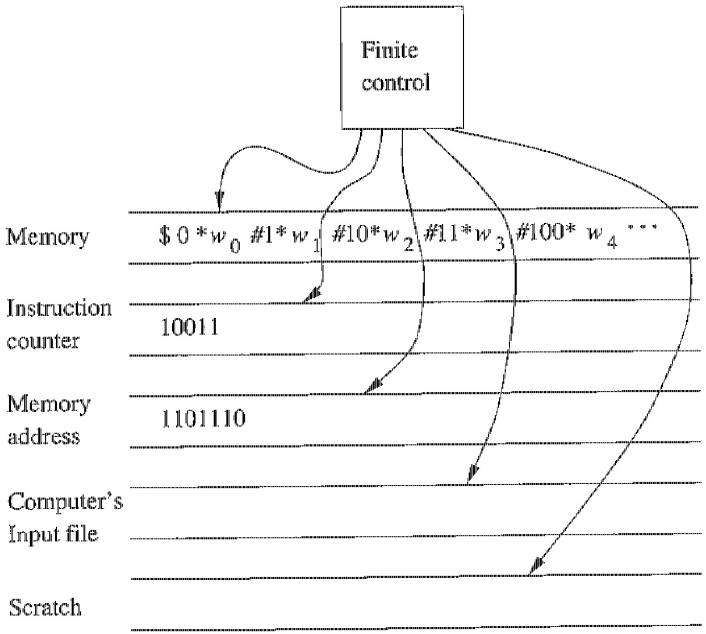
- The second tape is the “instruction counter.”
- This tape holds one integer in binary, which represents one of the memory locations on tape 1.
- The value stored in this location is the next computer instruction to be executed.

Simulating a Computer by a Turing Machine

- The third tape holds a “memory address” or the contents of that address after the address has been located on tape 1.
- To execute an instruction, the TM must find the contents of one or more memory addresses that hold data.
- First, the desired address is copied onto tape 3 and compared with the addresses on tape 1, until a match is found.
- The contents of this address is copied onto the third tape and moved to wherever it is needed.

Simulating a Computer by a Turing Machine

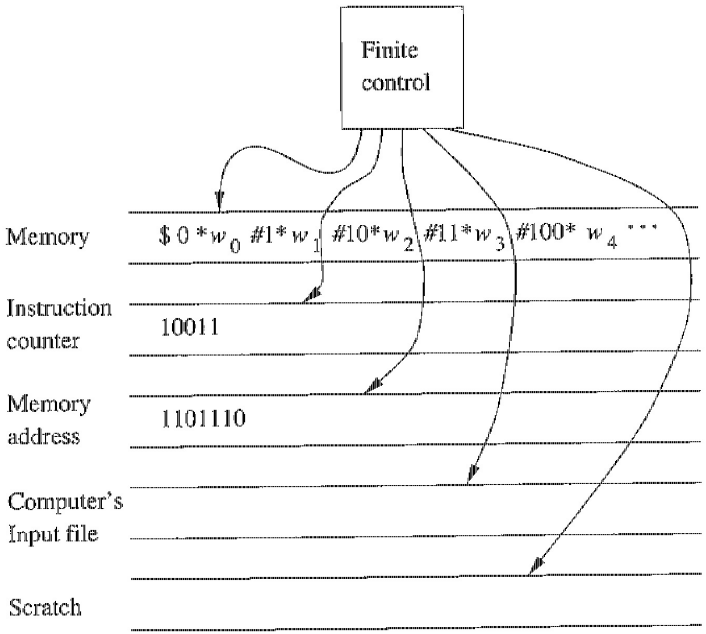
Simulation of the *instruction cycle* of a computer:



- Search the first tape for an address that matches the instruction number on tape 2.
- Start at the $ on the first tape, and move right, comparing each address with the contents of tape 2.
- The comparison of addresses on the two tapes is easy.

Simulating a Computer by a Turing Machine

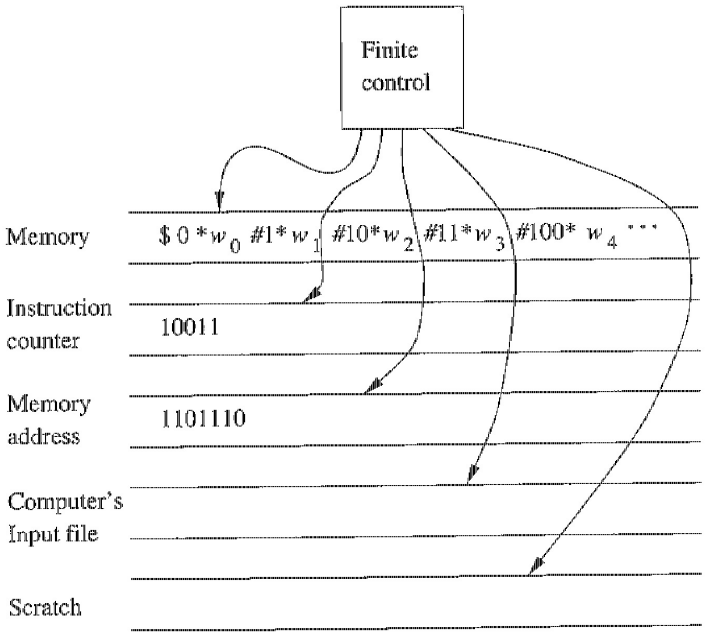
Simulation of the *instruction cycle* of a computer:



- When the instruction address is found, examine its value.
- Let us assume that when a word is an instruction, its first few bits represent the action to be taken (e.g., copy, add, branch).
- The remaining bits code an address or addresses that are involved in the action.

Simulating a Computer by a Turing Machine

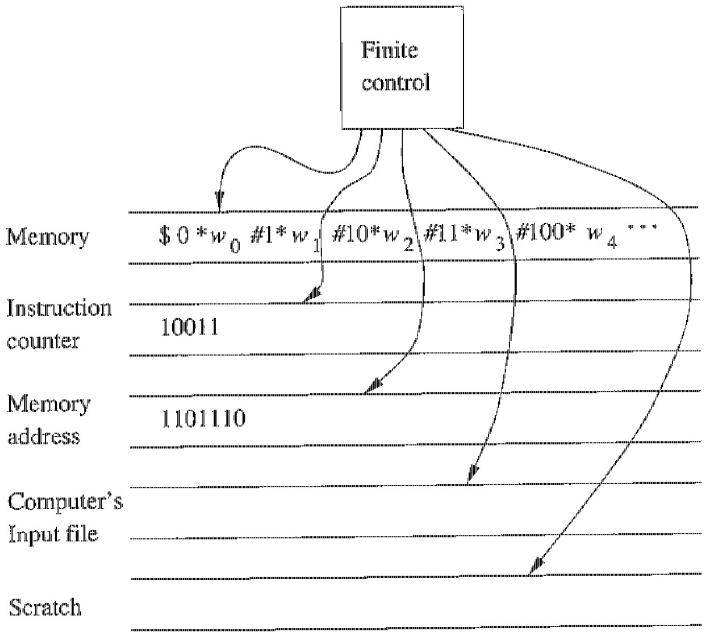
Simulation of the *instruction cycle* of a computer:



- If the instruction requires the value of some address, then that address will be part of the instruction.
- Copy that address onto the third tape, and mark the position of the instruction using a second track of the first tape.
- Now, search for the memory address on the first tape and copy its value onto tape 3.

Simulating a Computer by a Turing Machine

Simulation of the *instruction cycle* of a computer:



- Execute the instruction or the part of the instruction involving this value.
- Afer performing the instruction, and determining that the instruction is not a jump, add 1 to the instruction counter on tape 2 and begin the instruction cycle again.

Sample Instruction Executions

Copy

- We get the second address from the instruction, find this address by putting it, on tape 3 and searching for this address on Tape 1.
- When we find the second address, we copy the value into the space reserved for the value of that address.
- If more space is needed for the new value, or the new value uses less space than the old value, change the available space by shifting over.
- As a special case, the address may not yet appear on the first tape. In this case, we find the place and store both the address and the new value there.

Sample Instruction Executions

Shifting over

i. Copy, onto a scratch tape the entire nonblank tape to the right of where the new value goes.
ii. Write the new value, using the correct amount of space for value.
iii. Recopy the scratch tape onto tape 1, immediately to the right of the new value.

Sample Instruction Executions

Addition

- Go back to the instruction to locate the other address. Find this address on tape 1.
- Perform a binary addition of the value of that address aud the value stored on tape 3.
- By scanning the two values from their right ends, a TM can perform a ripple-carry addition.
- Should more space be needed for the result, use the shifting over technique.

Sample Instruction Executions

Jump

- Simply copy tape 3 to Tape 2 and begin the instruction cycle again.

Comparison of Running Times

- We shall use the TM not only to examine what can be computed at all, but what can be computed with enough efficiency to be used in practice.
- The dividing line between the tractable from the intractable is generally held to be between what can be computed in polynomial time and what requires more than any polynomial running time.
- Thus, we need to assure ourselves that if a problem can be solved in polynomial time on a typical computer, then it can be solved in polynomial time by a Turing machine, and conversely.

The issue of multiplication as a computer instruction

- The problem is that we have not put a limit on the number of bits that one computer word can hold.
- If the computer were to start with a word holding integer 2, and were to multiply by itself for n consecutive steps, then the word would hold the number 2^{2^n}. This number requires $2^n + 1$ bits to represent.
- So the time the Turing machine takes to simulate these n instructious would be exponential in n.

The issue of multiplication as a computer instruction

- One approach is to insist that words retain a fixed maximum length.
- We shall take a more liberal stance: the computer may use words that grow to any length, but one computer instruction can only produce a word that is one bit longer than the longer of its arguments.
- Addition is allowed, since the result can only be one bit longer than the maximum length of the addends.

The issue of multiplication as a computer instruction

- Multiplication is not allowed, since two m-bit words can have a product of length $2m$.
- However, we can simulate a multiplication of m-bit integers by a sequence of m additions, interspersed with shifts of the multiplicand one bit left.
- Thus, we can still multiply arbitrarily long words, but the time taken by the computer is proportional to the square of the length of the operands.

Comparison of Running Times

Theorem

If a computer

i. Has only instructions that increase the maximum word length by at most 1
ii. Has only instructions that a multitape TM can perform on words of length k in $O(k^2)$ steps or less,

then the Turing machine described can simulate n steps of the computer in $O(n^3)$ of its own steps.

Comparison of Running Times

PROOF:

- That program stored on the first tape may be long, but it is fixed and of constant length independent of n, the number of instruction steps the computer executes.
- Thus, there is some constant *c* that is the largest of the computer's words and addresses appearing in the program.
- There is also a constant *d* that is the number of words occupied by the program.

Comparison of Running Times

- Thus, after executing n steps, the computer cannot have created any words longer than $c + n$.
- And it cannot have created or used any addresses that are longer than $c + n$ bits either.
- Each instruction creates at most one new address that gets a value, so the total number of addresses after n instructions have been executed is at most $d + n$.
- Each address-word combination requires at most $2(c + n) + 2 = 2(c + n + 1)$ bits, including the address, the contents, and two marker symbols to separate them
- The total number of TM tape cells occupied after n instructions have been simulated is at most $2(c + n + 1) \times (d + n)$.
- As c and d are constants, this number of cells is $O(n^2)$.

Comparison of Running Times

- Each of the fixed number of lookups of addresses involved in one computer instruction can be done in $O(n^2)$ time.
- Since words are $O(n)$ in length, our second assumption tells us that the instructions themselves can each be carried out by a TM in $O(n^2)$ time.
- Shifting-over involves copying at most $O(n^2)$ data from tape 1 to the scratch tape and back again.
- Thus, shifting over also requires only $O(n^2)$ time per computer instruction.

Comparison of Running Times

- So, the TM simulates one step of the computer in $O(n^2)$ of its own steps.
- Thus, n steps of the computer can be simulated in $O(n^3)$ steps of the multi-tape Turing machine.
- A computer can be simulated for n steps by a one-tape Turing machine, using at most, $O(n^6)$ steps of the Turing machine.