Turing Machine II

Zeyu Mi

2020-12-9

Adapted From:
IALC 8.3.1,8.3.2,8.4.1,8.4.2,8.6.1,8.6.2

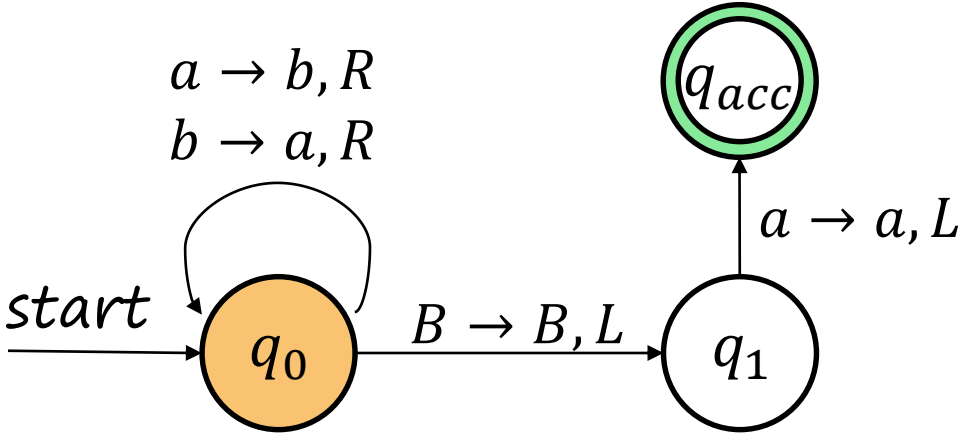
Review

- Computers are complex.
- Automata is a mathematical model of a computing device
  - Clean and simple
  - DFA: Five tuples $(Q, \Sigma, \delta, q_0, F)$
  - Turing Machine: Seven tuples, $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$

Review

- A **Turing Machine** consists of:
  - Q:A finite set of states
  - Σ: A finite set of input symbols
  - Γ: The complete set of tape symbols. $\Sigma \subset \Gamma$
  - δ: A transition function. $Q \times \Sigma \rightarrow Q \times \Gamma \times \{L, R\}$
  - q_0: The start state. $q_0 \in Q$
  - B: The blank symbol. $B \in \Gamma \wedge B \notin \Sigma$.
  - F: A set of final or accepting states. $F \subseteq Q$

Review



…	ℬ	ℬ	a	a	b	ℬ	ℬ	ℬ	…

The Language of a TM

- Similar to DFA, the language of a TM $\boldsymbol{M} = (\boldsymbol{Q}, \boldsymbol{\Sigma}, \boldsymbol{\Gamma}, \boldsymbol{\delta}, \boldsymbol{q_0}, \boldsymbol{B}, \boldsymbol{F})$ is set of strings that it accepts, or:

$$L(\boldsymbol{M}) = \{w | w \in \Sigma^* \wedge p \in F \wedge q_0 w \vdash^* \alpha p \beta\}$$

- For a certain language L, if there exists a TM $\boldsymbol{M}$ such that $L = L(\boldsymbol{M})$, then we call L is a **recursively enumerable language(递归可枚举语言)**, or ***RE***.

How we investigate computability?

2.1 Discussion on Problems

Part1. Intro & Set theory(I) : Basics & Formal Language.

Part2. Set Theory(II): Axiom system & Cardinality.

Part3. Capture Structures: Binary Relation & Function

2.2 Discussion on Computation

Part1. Turing Machine Basics.

Part2. Variants of Turing Machine. Church-Turing Thesis.

2.3 Discussion on computability

Part1. The Language of Turing Machine. R & RE

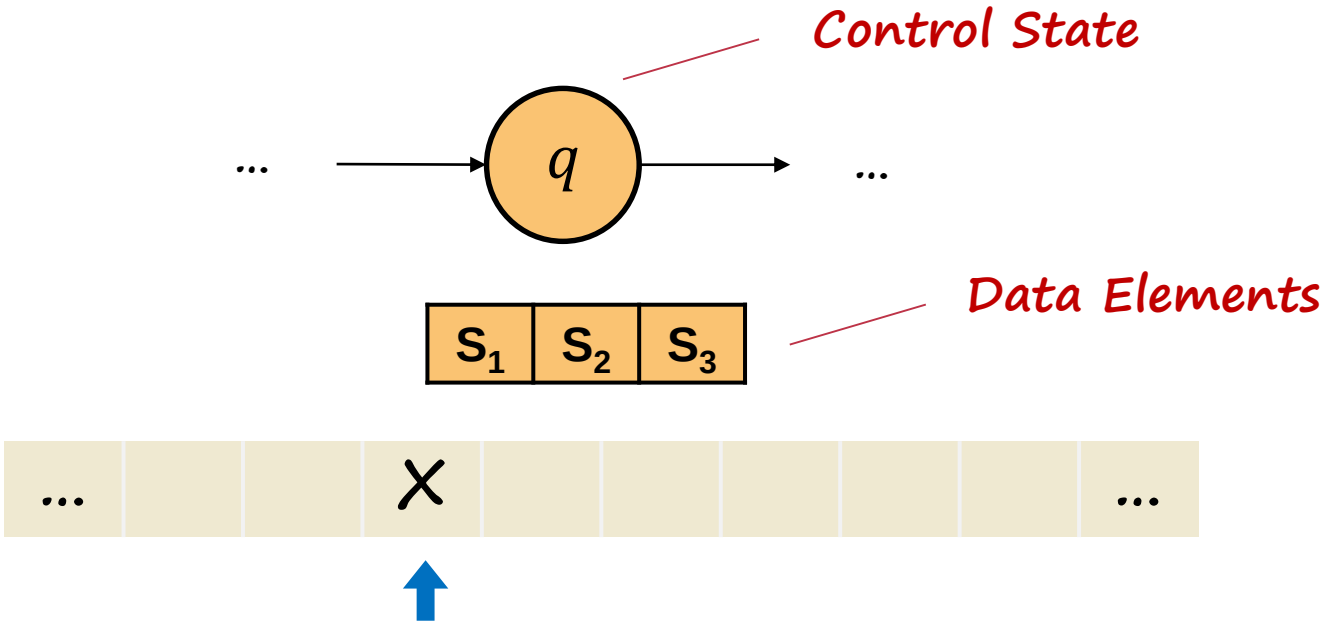
Part2. Undecidability.

Remember the Execution State

- **It's common in our daily programming that using some variables to remember the execution state.**
  - A certain char we've seen in processing a string.
  - The biggest number we've found in finding the maximum.
  - Did we end a loop normally or by break.
  - Or many states combined together.

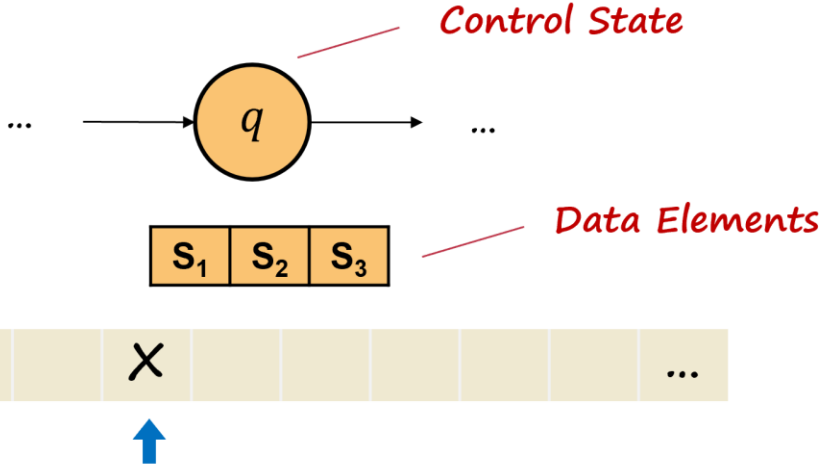
Can we simply apply this programming techniques into TM?

Remember the Execution State



TM with storage

- We can use the state to hold a **finite** amount of data.
- The technique requires no extension to the TM model, we just think the state as tuple now.
  - $[q, S_1, S_2, S_3]$ is the current state.
- Regarding states this way allows us to describe transitions in a more systematic way

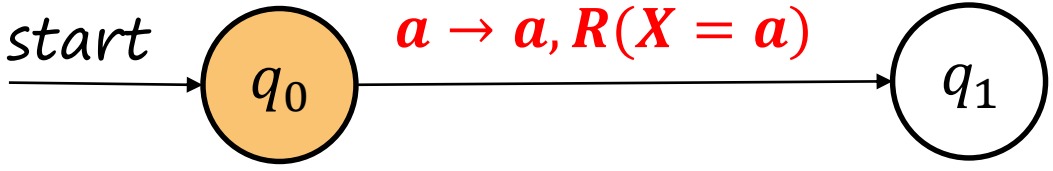


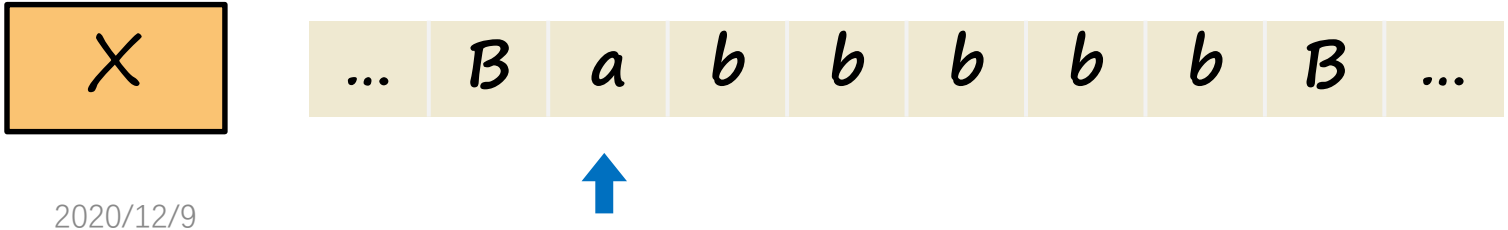
Example: $\{ab^* + ba^*\}$

- We can remember the first symbol we've seen, and check whether the remaining symbols are different from it.
  - $\boldsymbol{q_0}$: Read the first symbol, save it into data elements.
  - $\boldsymbol{q_1}$: Then scan the remaining symbols, we continue this process if every symbol is different from the symbol stored , and stop and accept when we meet a blank symbol.
- The data elements can be $\boldsymbol{a}, \boldsymbol{b}$ or $\boldsymbol{B}$, so the actual states of this TM are $\{q_0, q_1\} \times \{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{B}\}$

Example: $\{ab^* + ba^*\}$

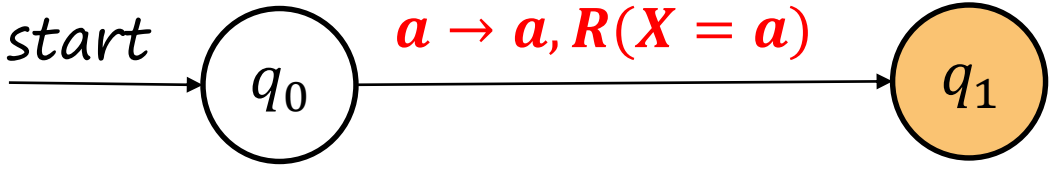
Pseudo TM (ab*) – Phase 1





Example: $\{ab^* + ba^*\}$

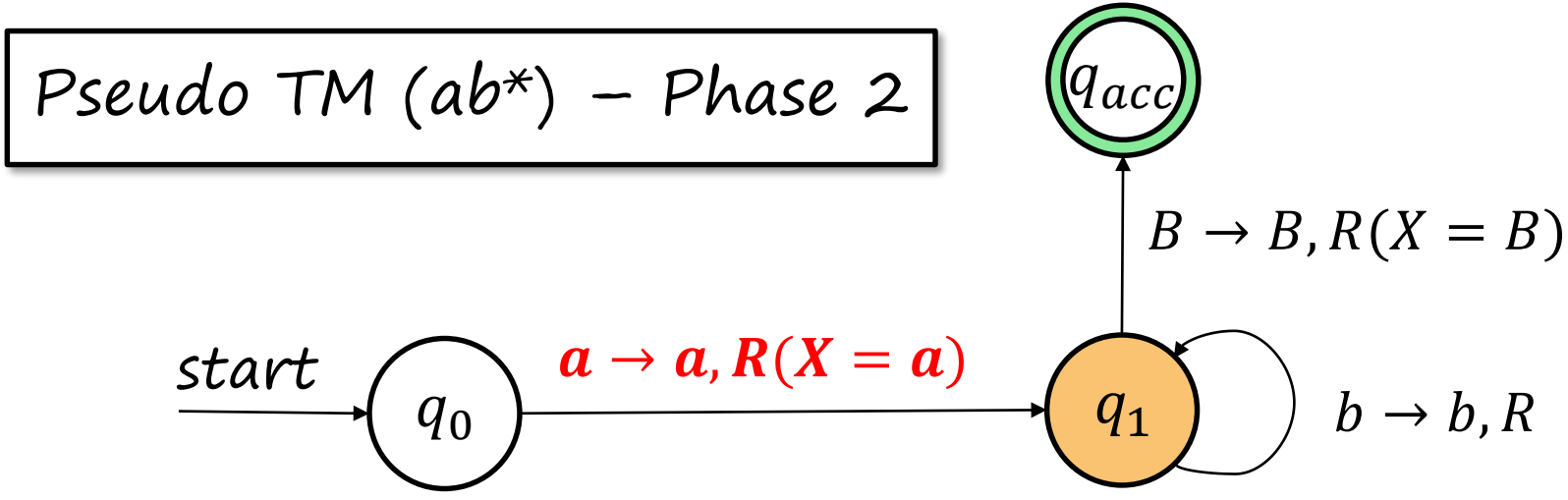
Pseudo TM (ab*) – Phase 1



X=a

...	B	a	b	b	b	b	b	B	...

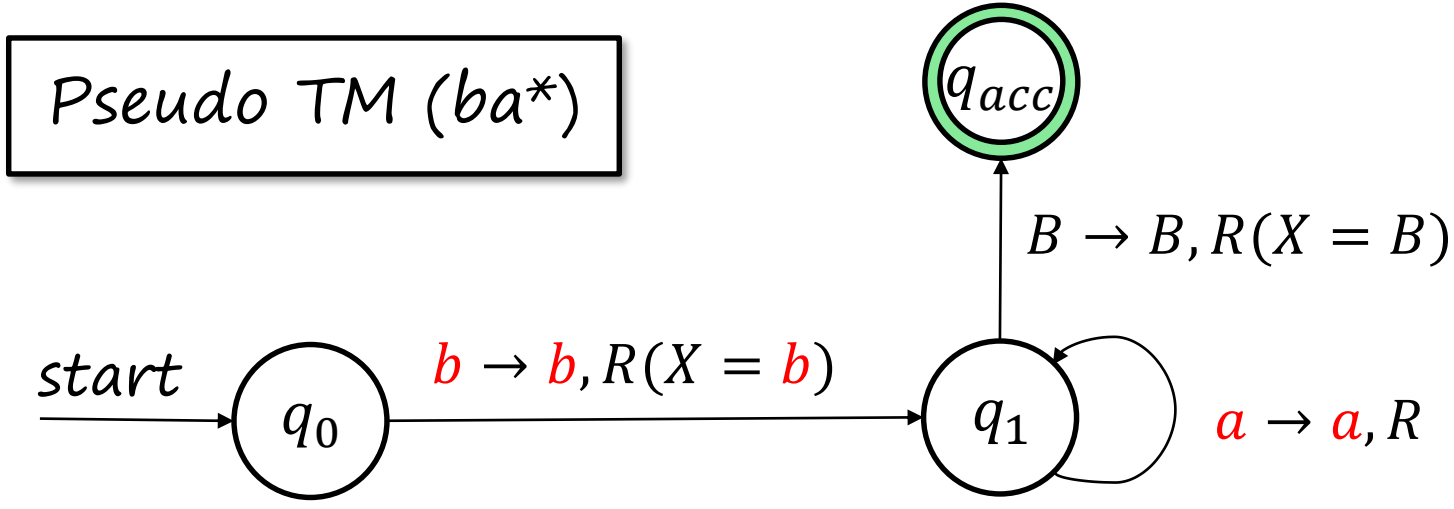
Example: $\{ab^* + ba^*\}$

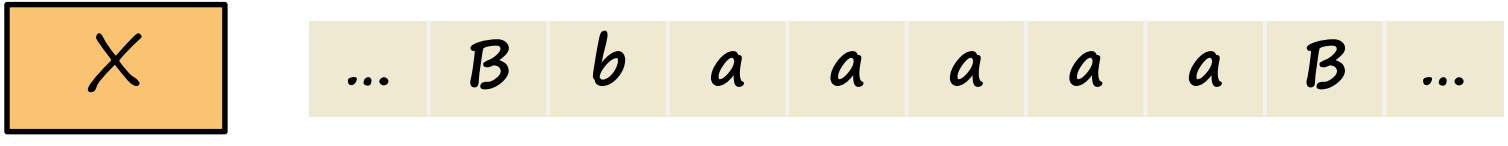


X=a

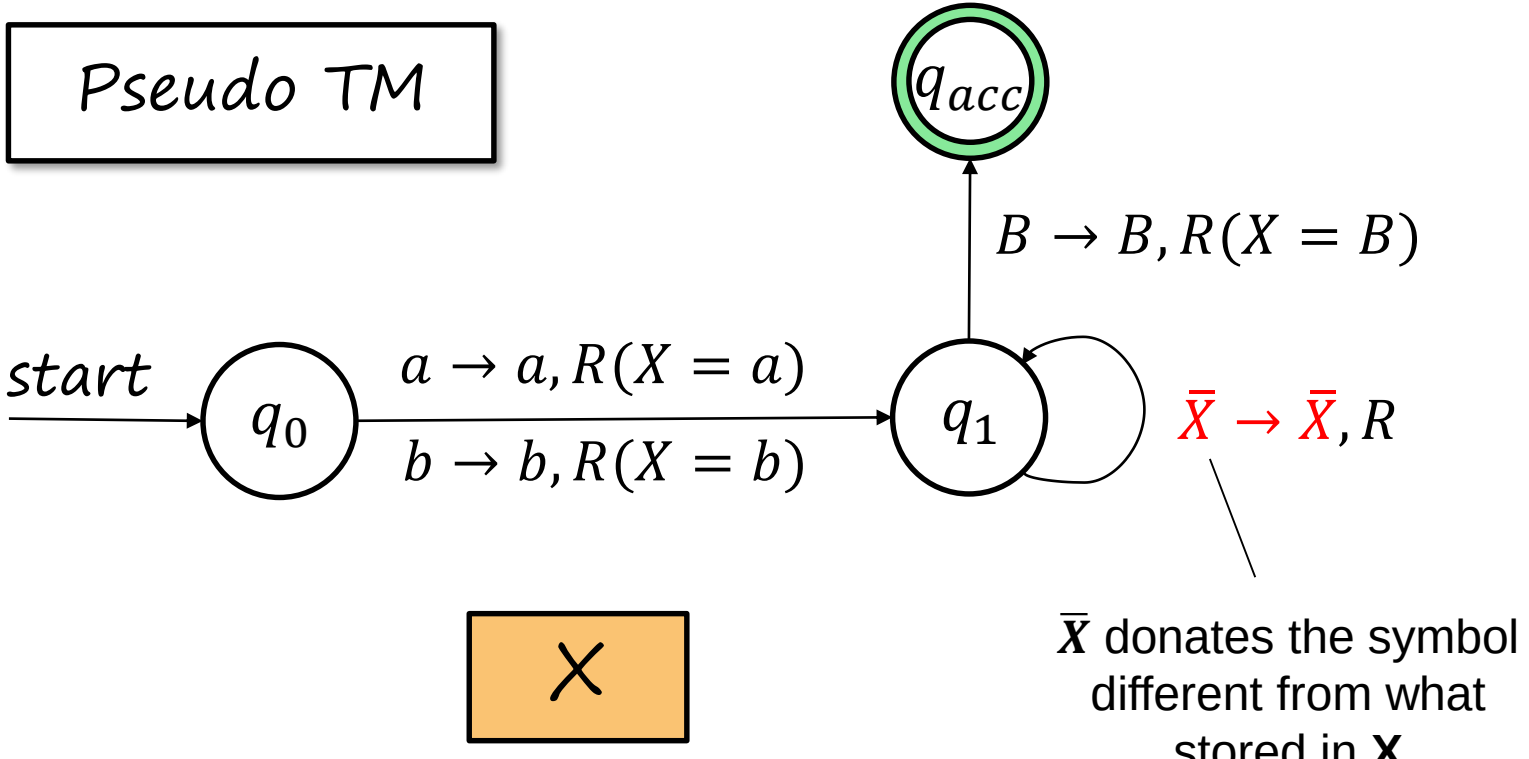
...	B	a	b	b	b	b	b	B	...

Example: $\{ab^* + ba^*\}$

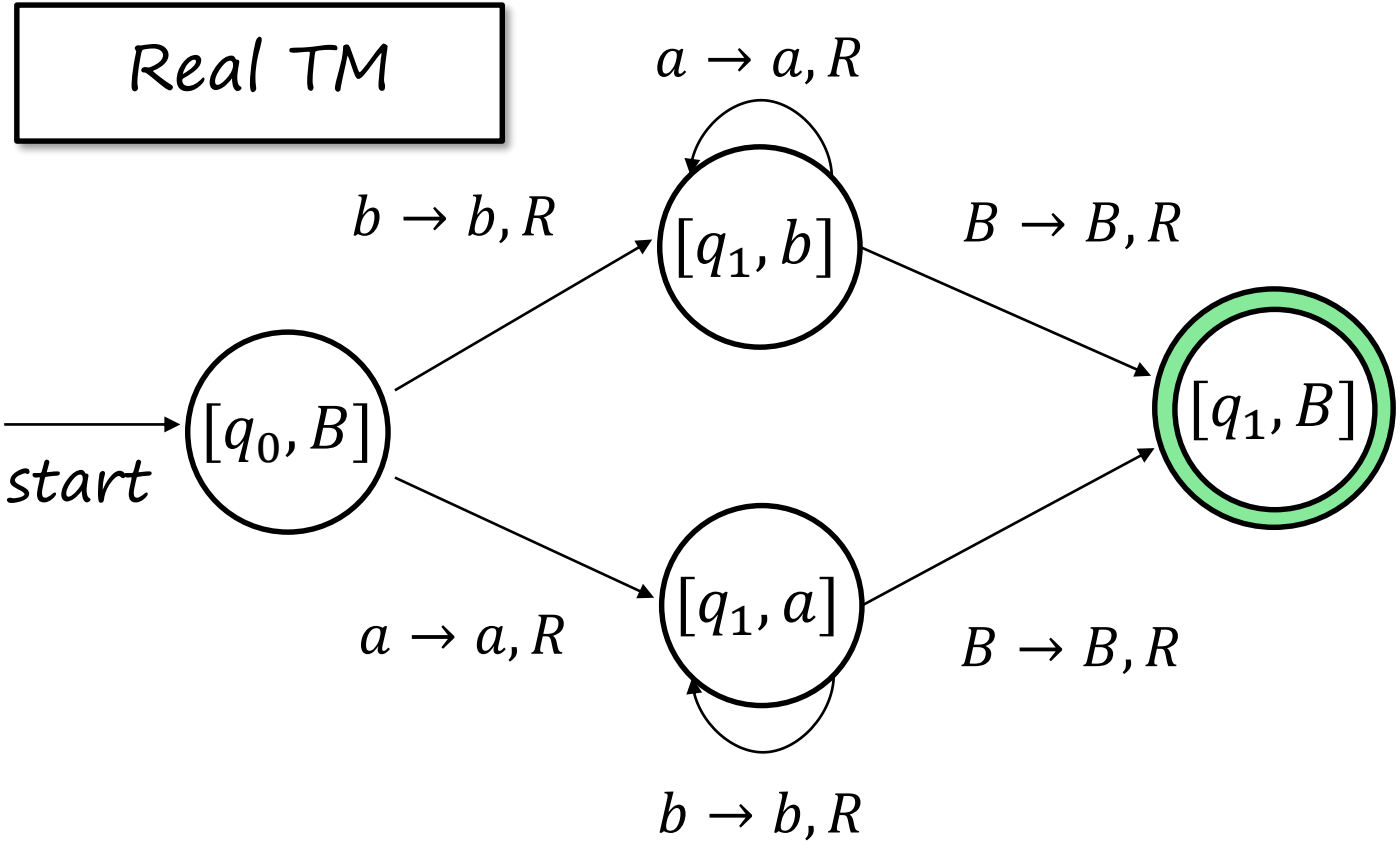




Example: $\{ab^* + ba^*\}$



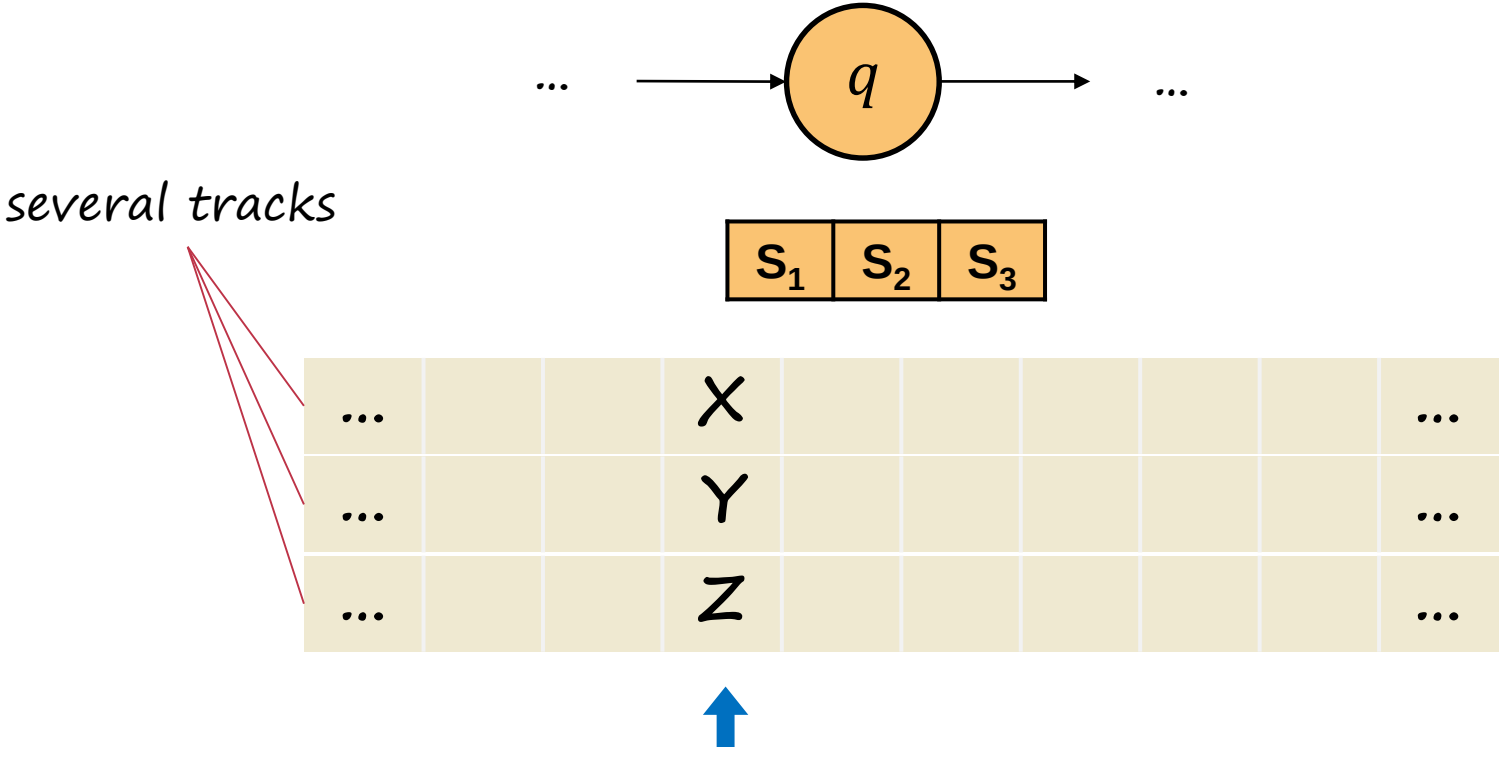
Example: $\{ab^* + ba^*\}$



Remember the Execution State

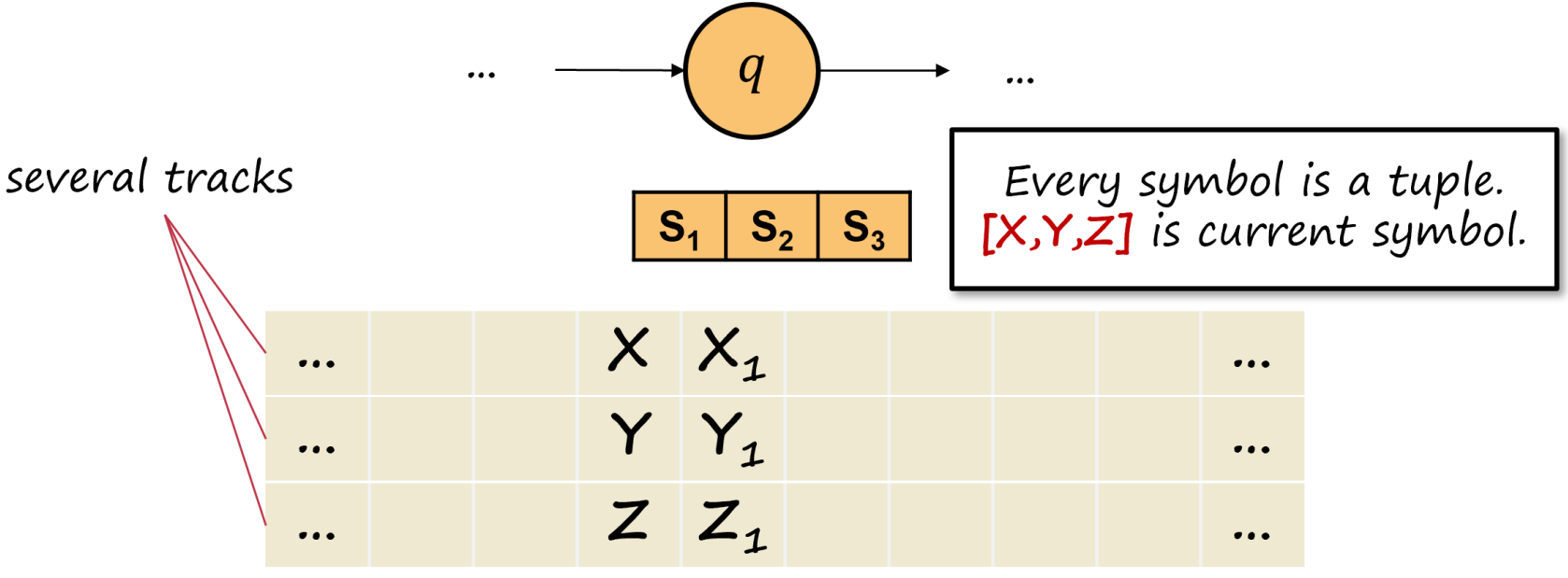
- We may need to remember the state for every single elements in an array, which means we may need to remember infinite states.
  - A bit array denotes whether an element has certain property.
- But the technique above can only hold finite many data in states

Remember the Execution State

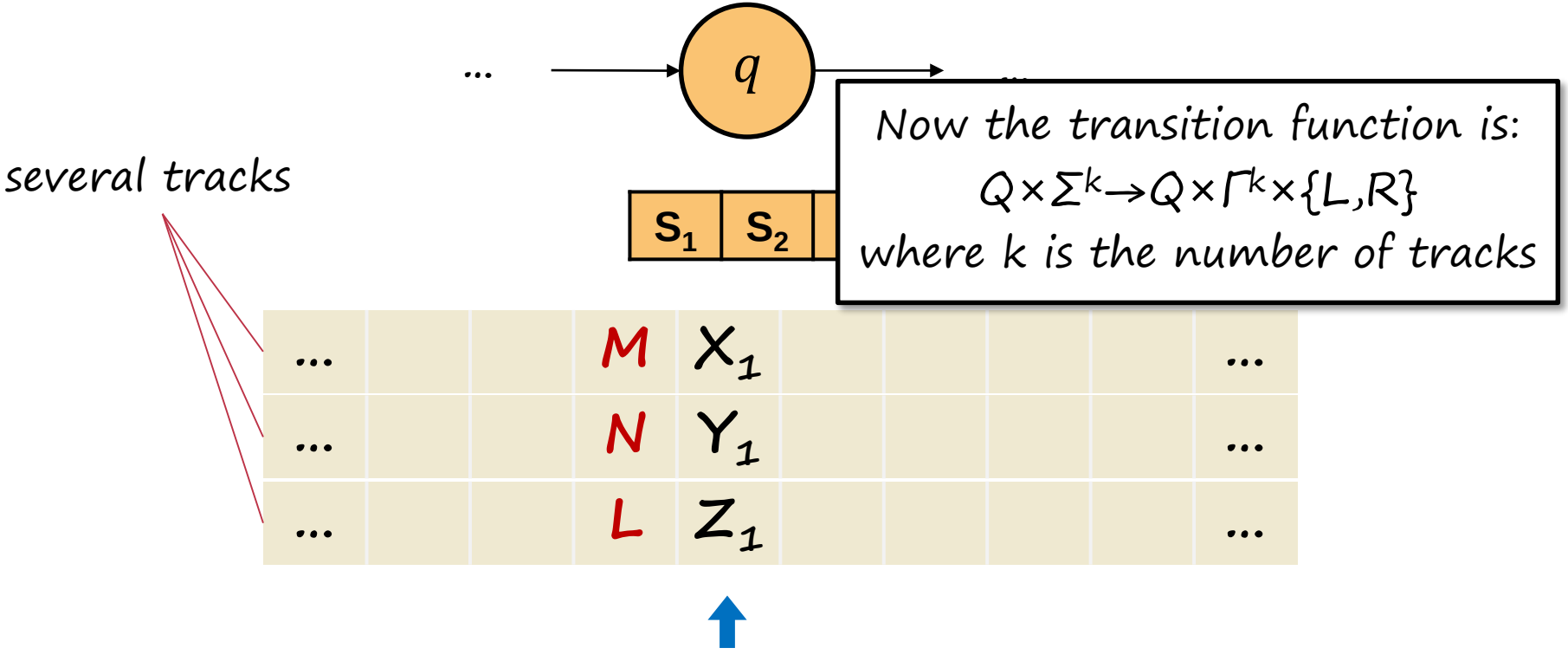


Multitrack

- We can think of the tape of a Turing machine as composed of several tracks. Each track can hold one symbol, and the tape alphabet of the TM consists of tuples.



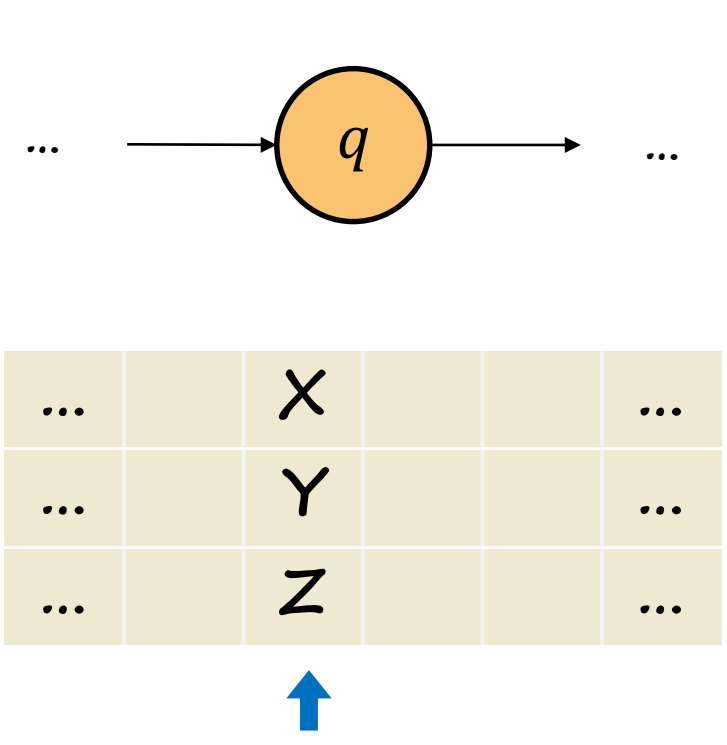
Multitrack


...
q
...
Now the transition function is:
$Q \times \Sigma^k \rightarrow Q \times \Gamma^k \times \{L,R\}$
where k is the number of tracks
several tracks
S_1
S_2
... M X_1 ...
... N Y_1 ...
... L Z_1 ...

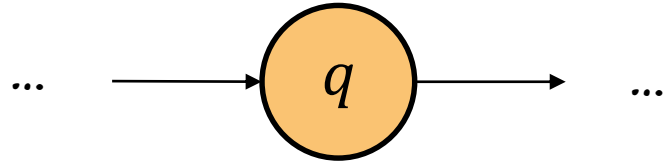
Multitrack

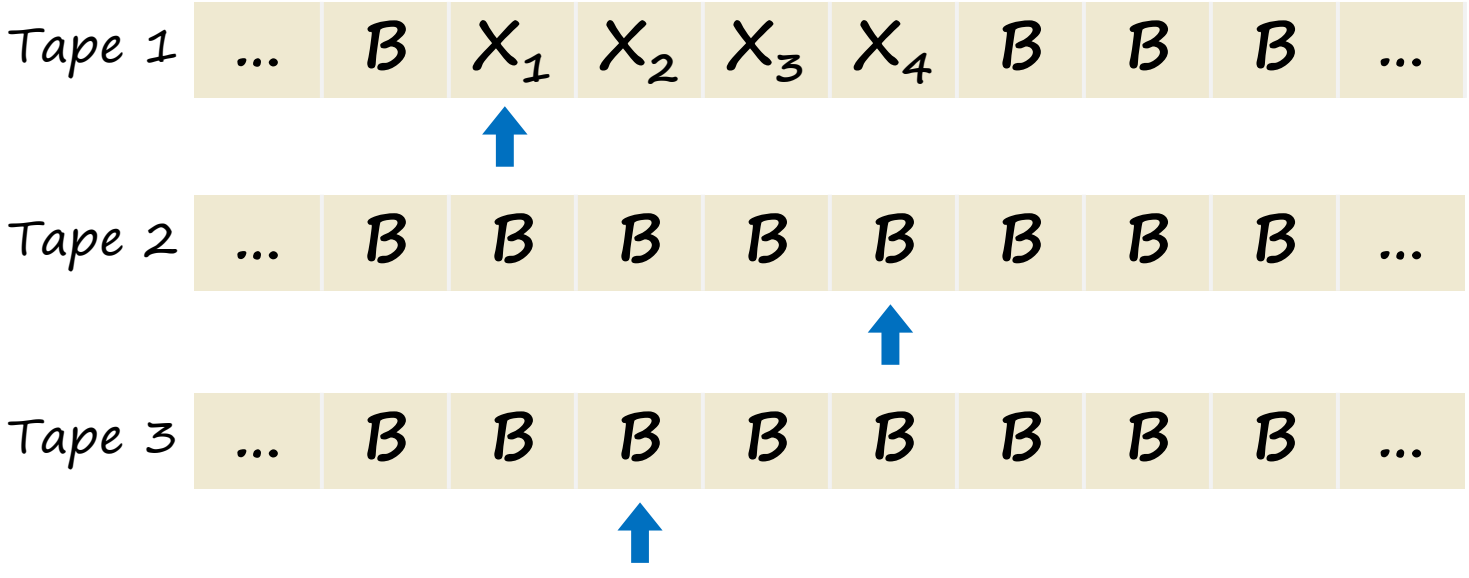
- Multitrack also **does not extend** what the Turing machine can do. It is simply a way to view tape symbols and to imagine that they have a useful structure.
  - Actually, we can use a single alphabet Γ' such that there's a bijection between Γ' and $\Gamma \times \Gamma \ldots \times \Gamma$. So we can replace the tuple in $\Gamma \times \Gamma \ldots \times \Gamma$ with a symbol in Γ' in all places.

Multitrack and Multi-tape

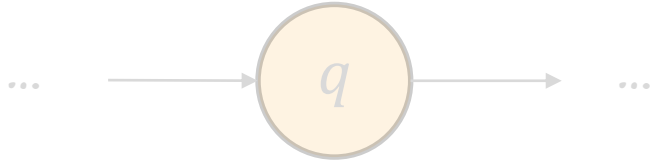


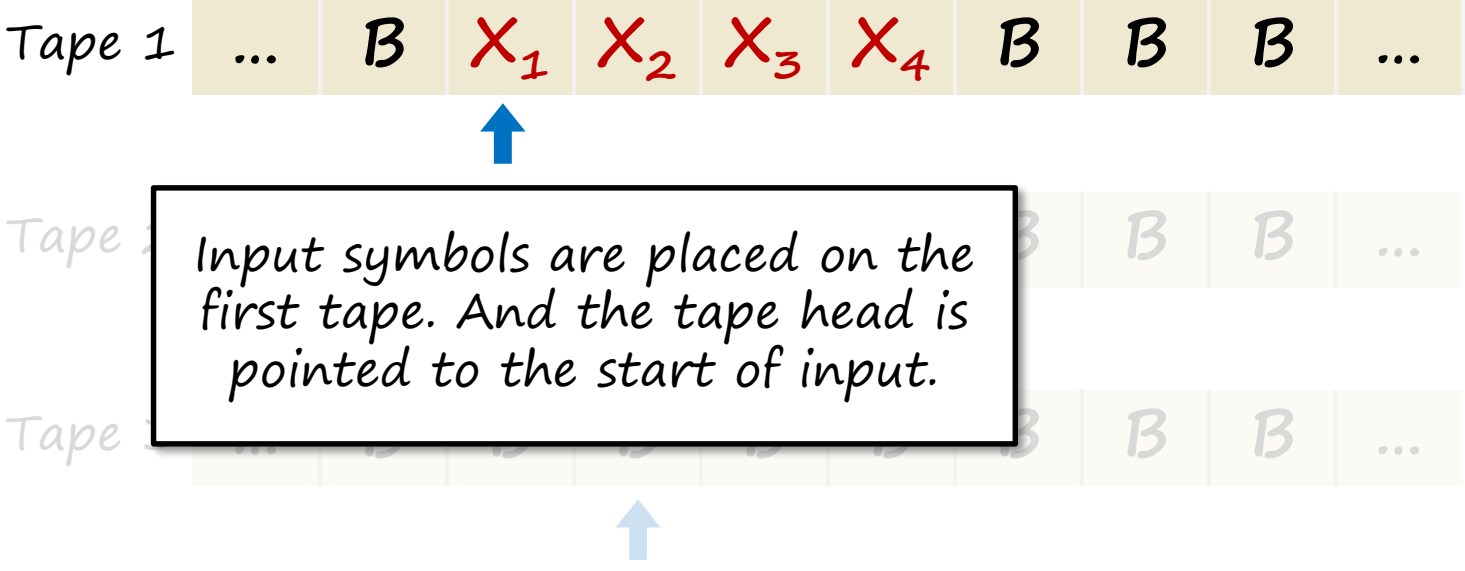
Multi-tape Turing Machine



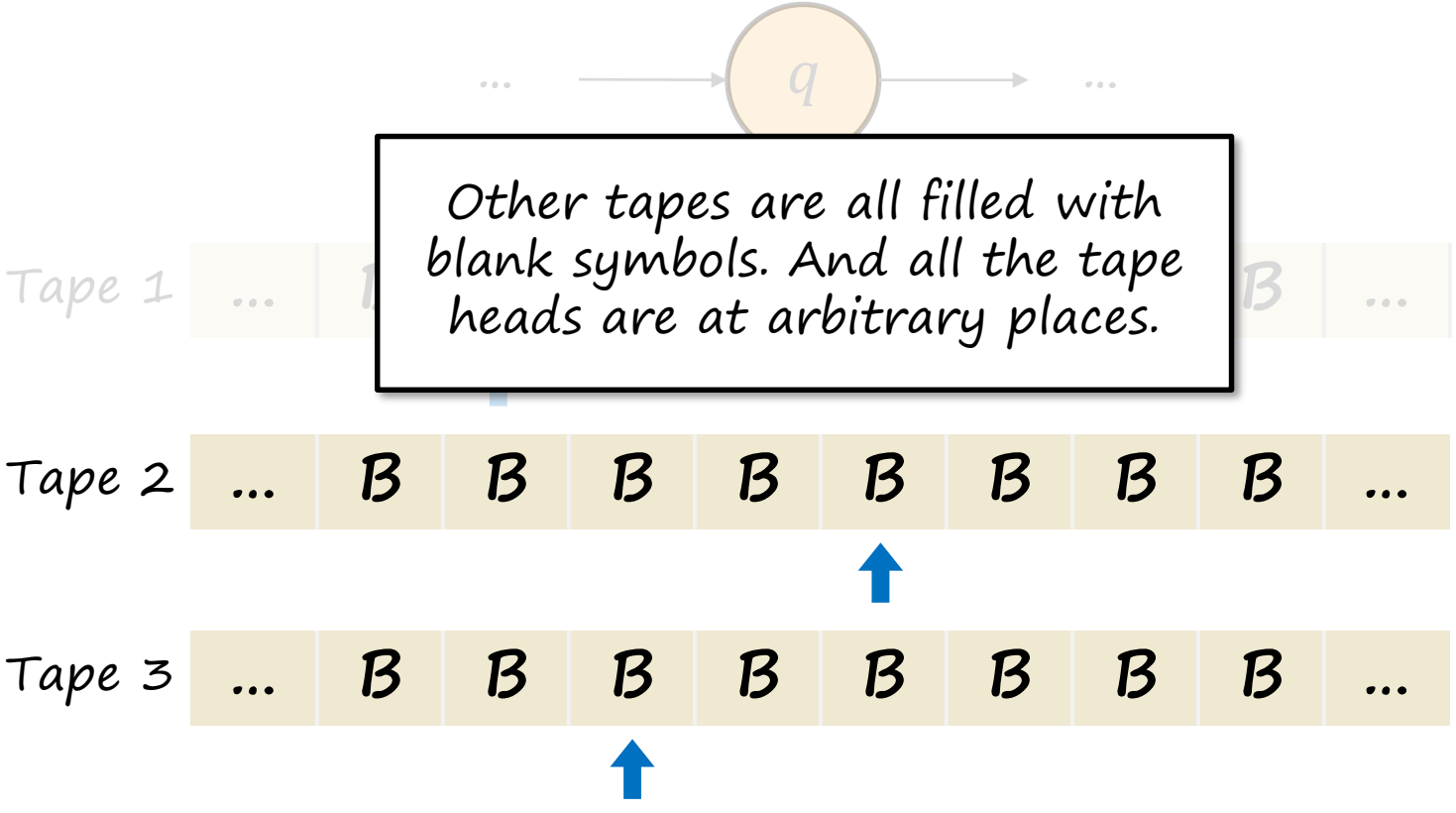


Multi-tape Turing Machine

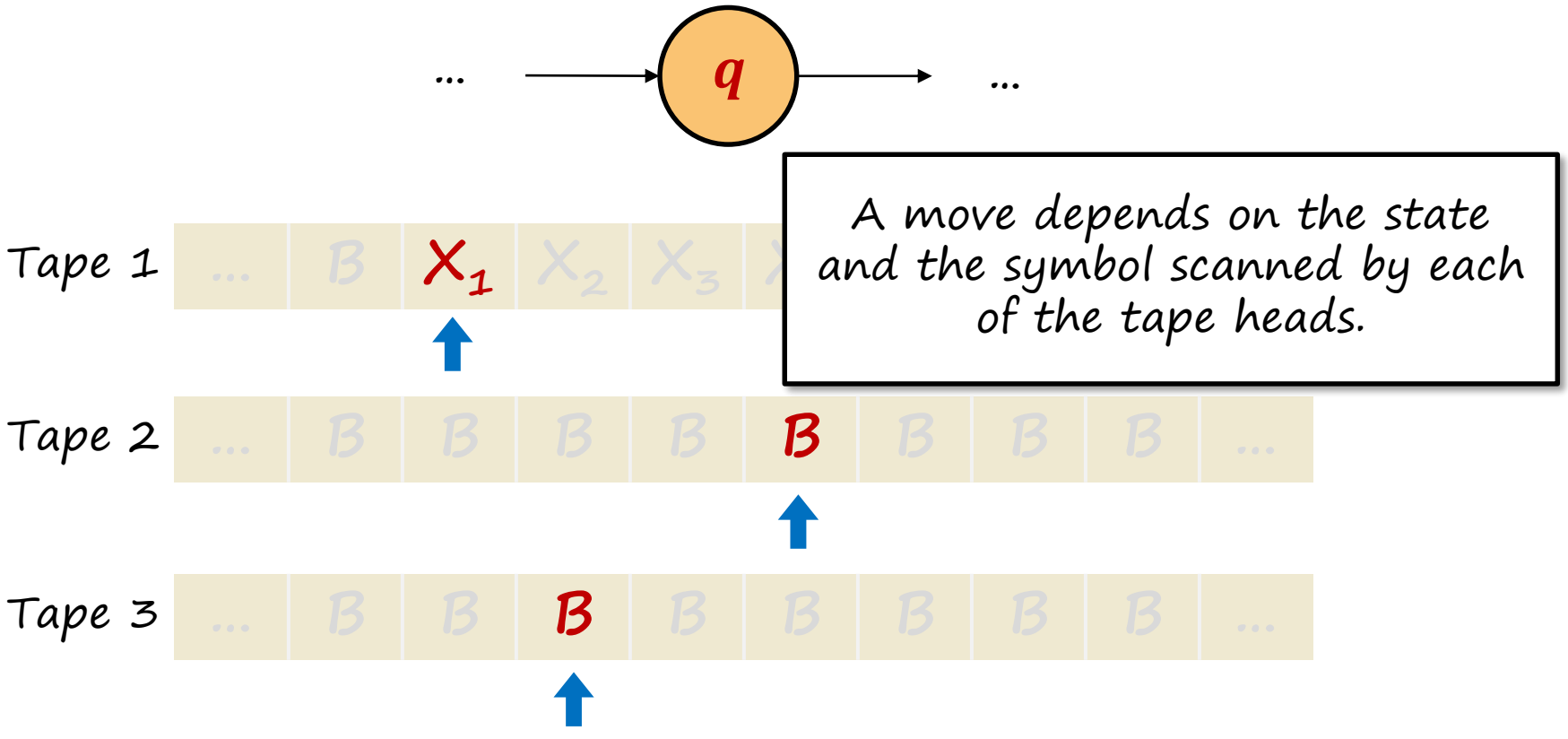




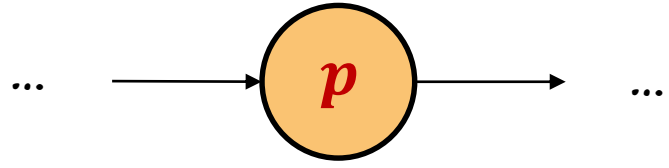
Multi-tape Turing Machine

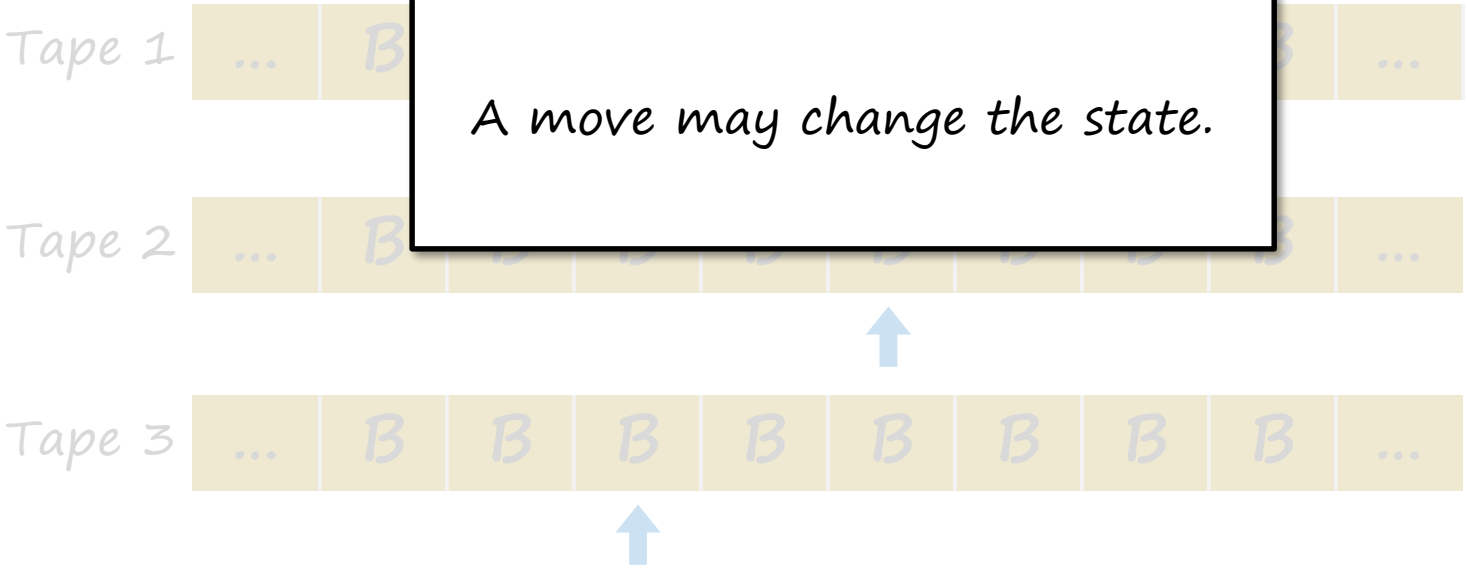


Multi-tape Turing Machine



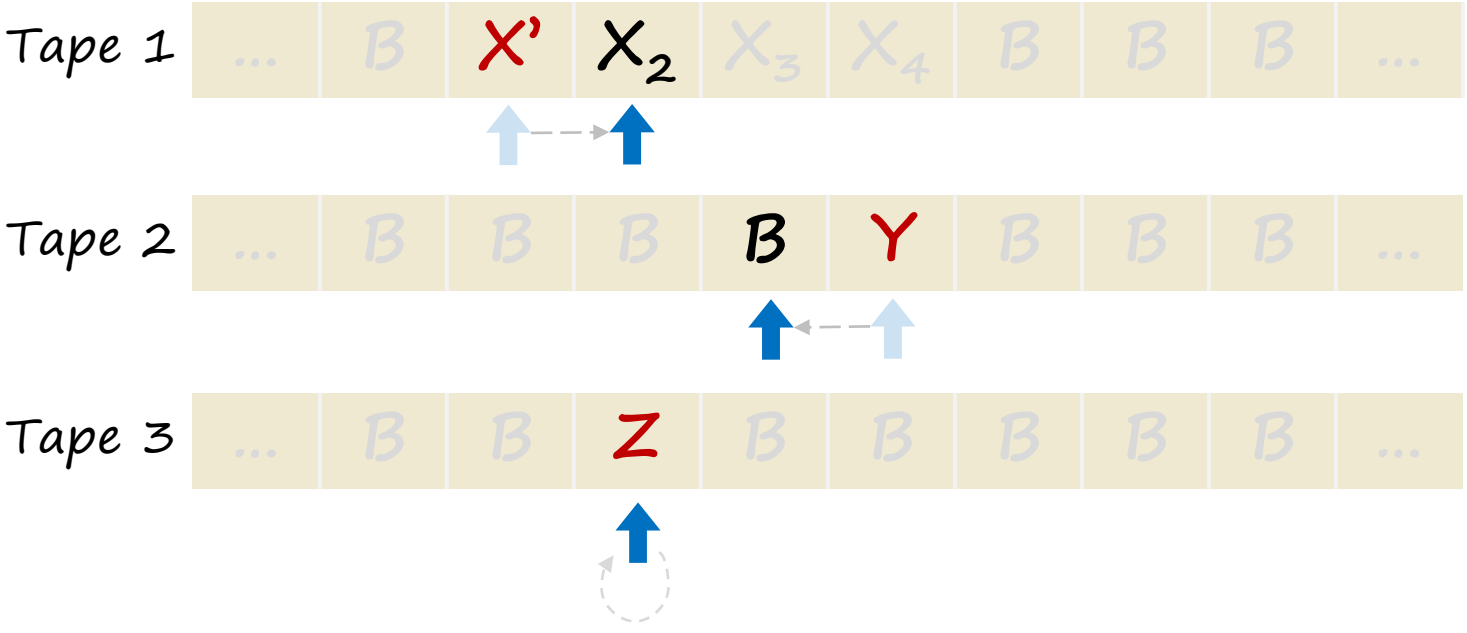
Multi-tape Turing Machine





Multi-tape Turing Machine

On each tape, a symbol may be written on the current cell, and the tape heap move to left, right or stationary.



Multi-tape Turing Machine

- A ***k*-tape Turing Machine** can also be donated as a seven tuple $(\boldsymbol{Q}, \boldsymbol{\Sigma}, \boldsymbol{\Gamma}, \boldsymbol{\delta}, \boldsymbol{q_0}, \boldsymbol{B}, \boldsymbol{F})$
- But now the transition function is:

$$\delta: Q \times \Sigma^k \rightarrow Q \times \Gamma^k \times \{L, R, S\}^k$$

k symbols scanned by all tapes

k symbols written into current cells of all tapes

k moving directions for all tape heads. **L, R and S donate left, right and stationary.**

Computation Capability

- For any certain type of automata (e.g. DFA,TM), the computation capability of it can be regarded as all the languages the type of automata can accept.

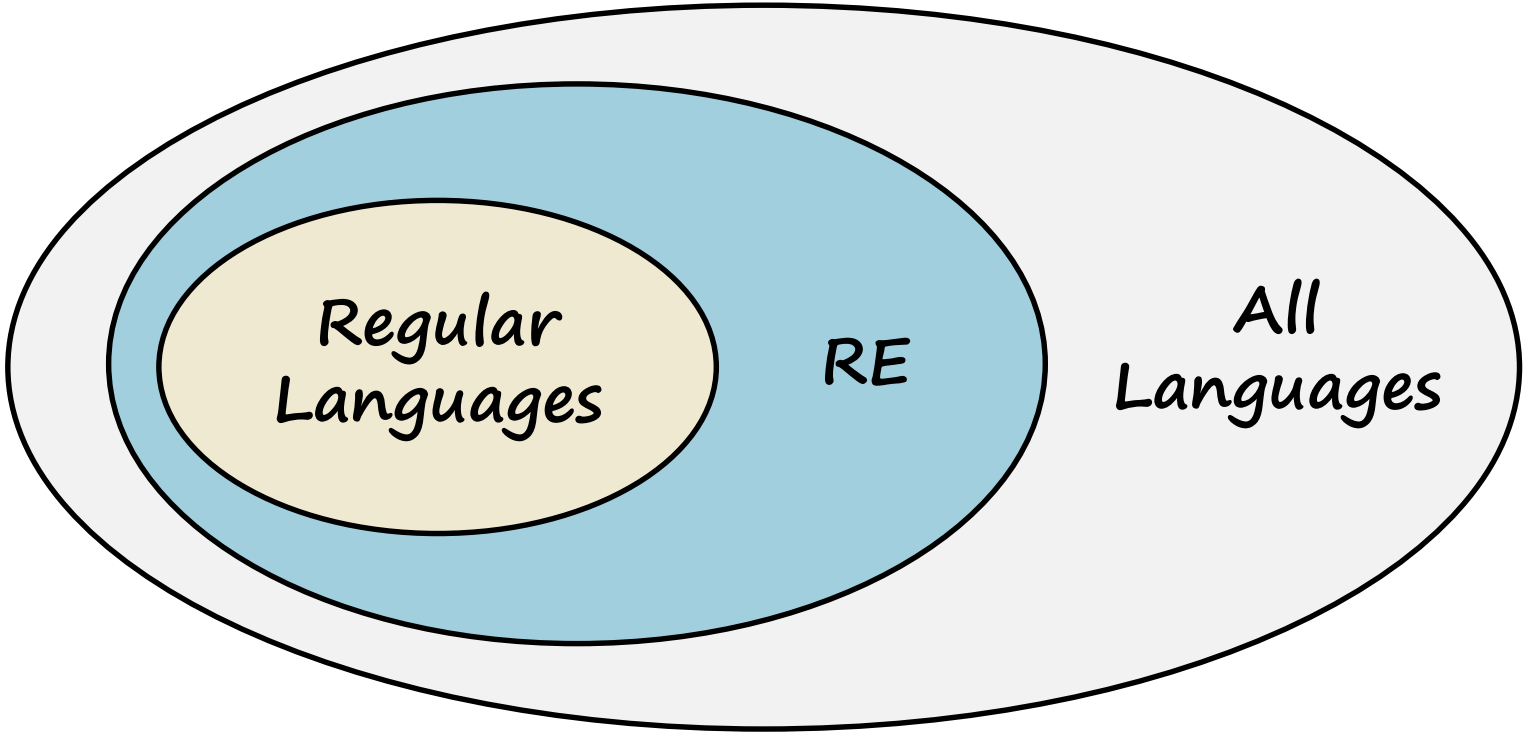
$$P(T) = \{L(A) | A \ is \ a \ T\},$$
where T can be DFA , Turing Machine, etc.

- For DFA, the capability is all the regular languages. For Turing Machine, it is all the *RE* languages.
  - $P(DFA) = \{L | L \ is \ a \ Regular \ Language\}$
  - $P(TM) = \{L | L \ is \ a \ RE \ Language\}$

Computation Capability

- For two types of automata T_1 and T_2
  - If $P(T_1) = P(T_2)$, we say T_1 and T_2 have same capability.
  - If $P(T_1) \subset P(T_2)$, we say T_2 have greater capability than T_1.
- DFA and TM:
  - For any language $\boldsymbol{L}$ accepted by a DFA ($\boldsymbol{L} \in \boldsymbol{P(DFA)}$), there's a TM $\boldsymbol{M}$ such that $L(\boldsymbol{M}) = \boldsymbol{L}$ ($\boldsymbol{L} \in \boldsymbol{P(TM)}$), so it means that $\boldsymbol{P(DFA) \subseteq P(TM)}$
  - And we know there's some languages can be accepted by TM but can not be accepted by DFA(like $\{1^n2^n\}$), so we have $P(DFA) \subset P(TM)$, which means **TM have greater computation capability than DFA**
  - Moreover, $\boldsymbol{P(ST-TM) = P(MT-TM)}$ (Why?)

Computation Capability



Computer and Turing Machine

- What we care about most is that whether Turing Machine can complete what a computer can do.
  - We even hope TM has greater capability than a computer.
- A real computer has **memory limitations**: you have a finite amount of RAM, a finite amount of disk space, etc.
- But we'll focus on an **idealized computer** which is like a regular computer, but with unlimited RAM and disk space. It functions just like a regular computer, but never runs out of memory.

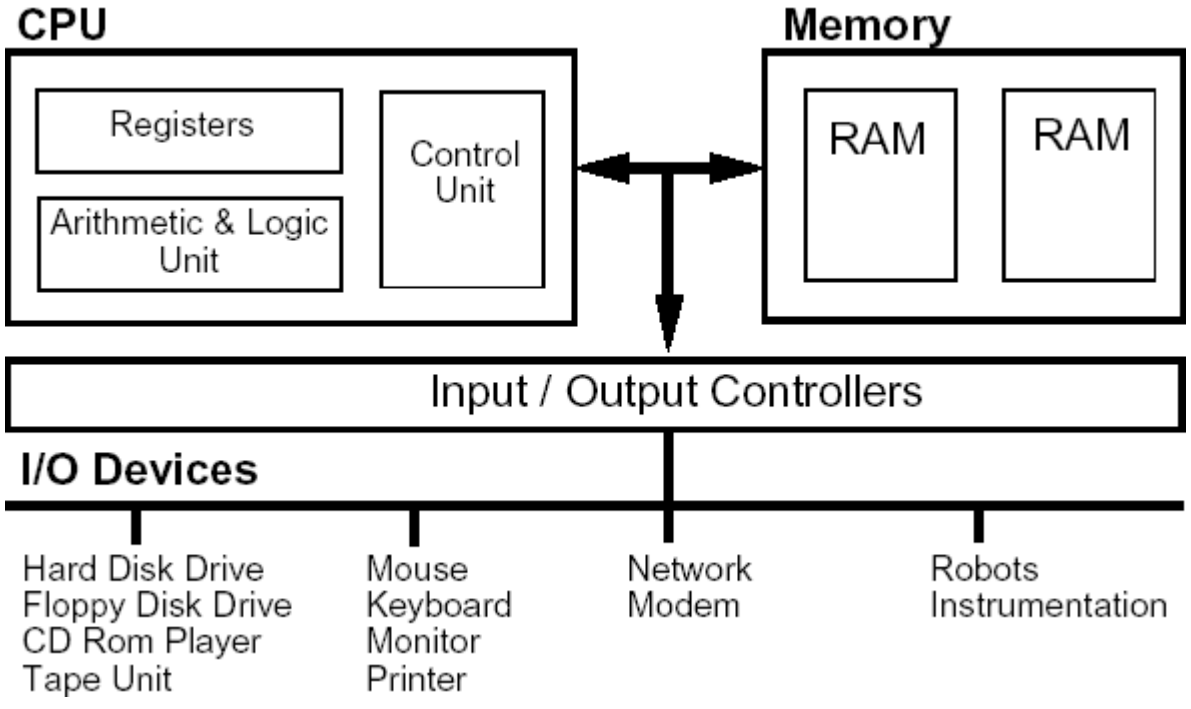
Computers Simulate TMs

- It’s obvious that we can simulate a TM with a computer since we can describe it in a programmable way.
  - How about a programming lab for writing a TM?
- But the tape of TM is infinite, we can only simulate all TMs in an idealized computer with infinite memory
  - Actually, if a computer has finite memory, there’s no chance to simulate all TMs.

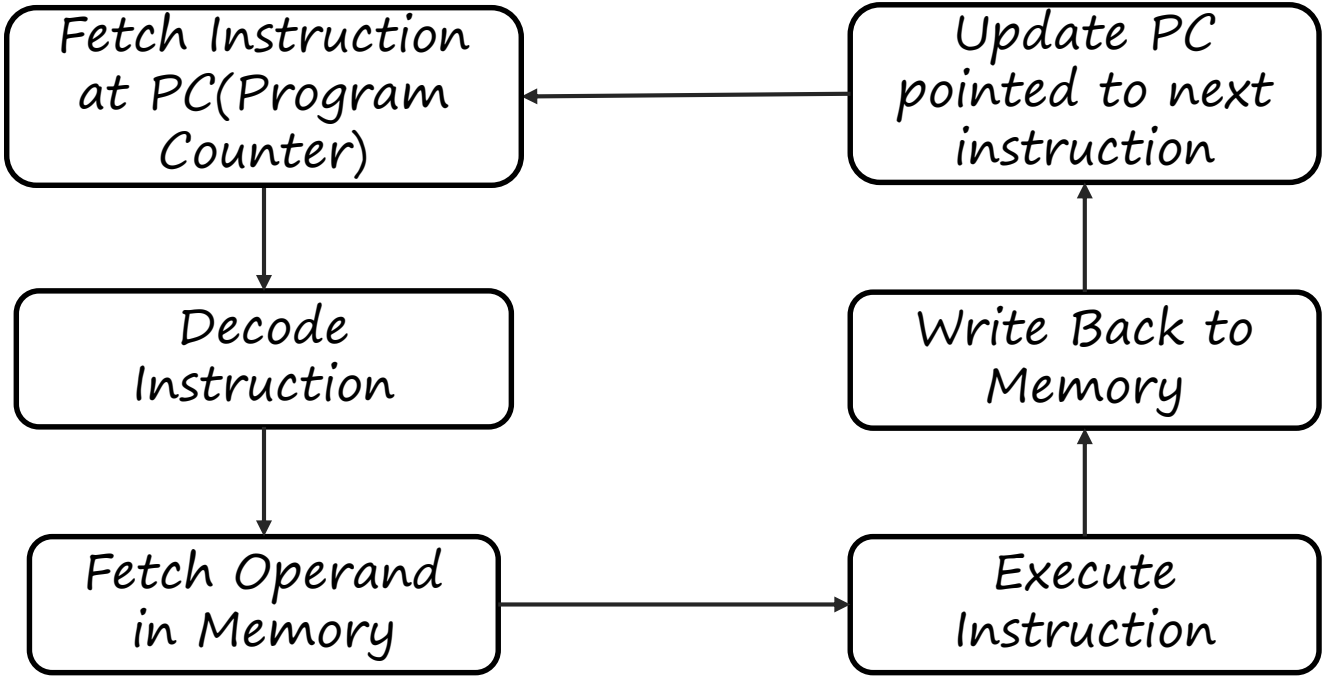
TMs Simulate Computers

- The more interesting question is that whether a TM can simulate an idealized computer.
- But what is the specific definition of an idealized computer or even a general computer?
  - No rigorous mathematical definition, we can only define it by ourselves.

Computer Architecture



Perform by Executing Instructions



CSAPP 4.3.1 Instruction Cycle

Assumptions

- The storage of a computer consists of a sequence of ***words***, each with an ***address***. A word can be of any length. Addresses will be assumed to be integers 0,1,2 and so on instead of align them to 32 or 64 bit. We use $\boldsymbol{M[n]}$ to donate the nth word in memory.
- There's no other registers except a PC.
- Program(code) of the computer is stored in some of the words of memory. Each word represents a simple instruction like:
  - Zero(n): $M[n] = 0$
  - Successor(n): $M[n] = M[n] + 1$
  - Mov(m, n): $M[n] = M[\mathrm{m}]$
  - Jump(m,n,q): if $M[m] == M[n]$, then set PC to q.

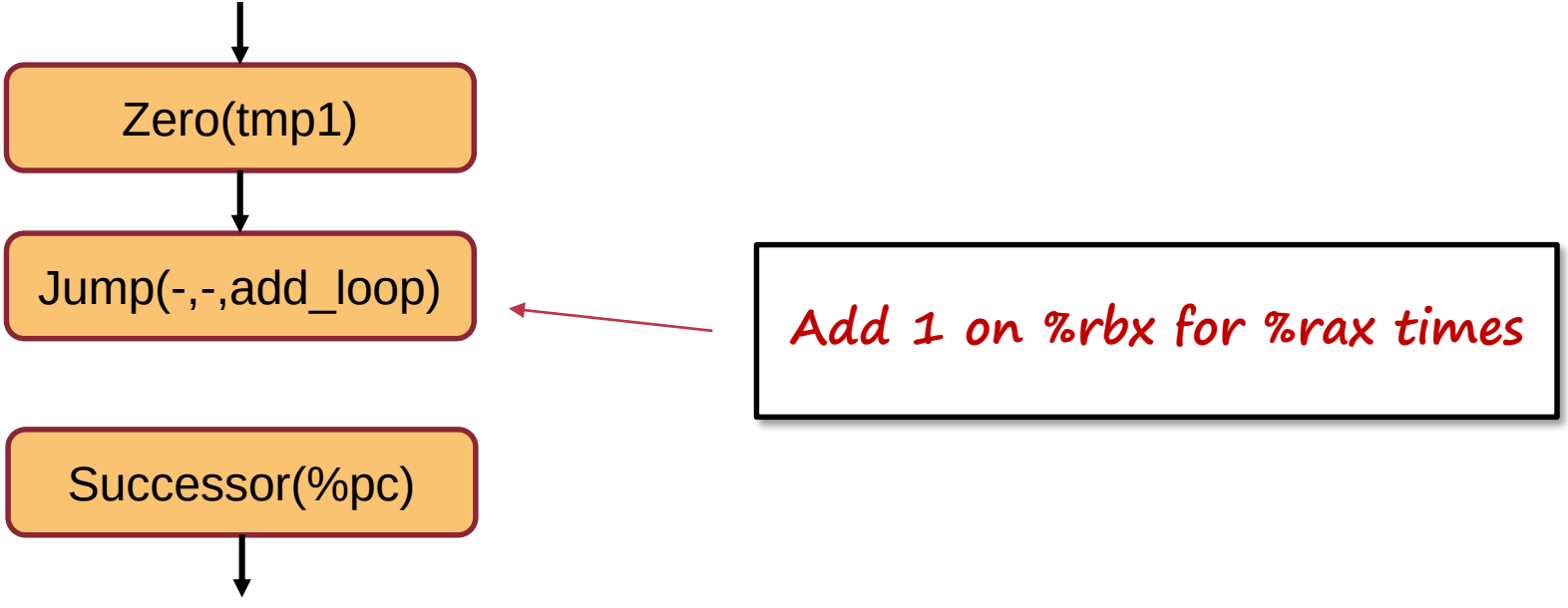
Actually these four instructions are enough for all computation!

To simplify real instructions

- **First, map each computer state with a word region(no overlap between every two region)**
  - M[0]~M[MEM_MAX] for memory
  - M[%pc], M[%rax], M[%rbx]… for registers
  - …
- **Moreover, we can use the tape to store extra temporal state**
  - M[tmp1], M[tmp2], M[tmp3]…
- **And, we should write down useful constants**
  - M[$zero]=0, M[$reg_max] = 0xFFFFFFFF …
- **Now, encode real instructions to simple instructions**

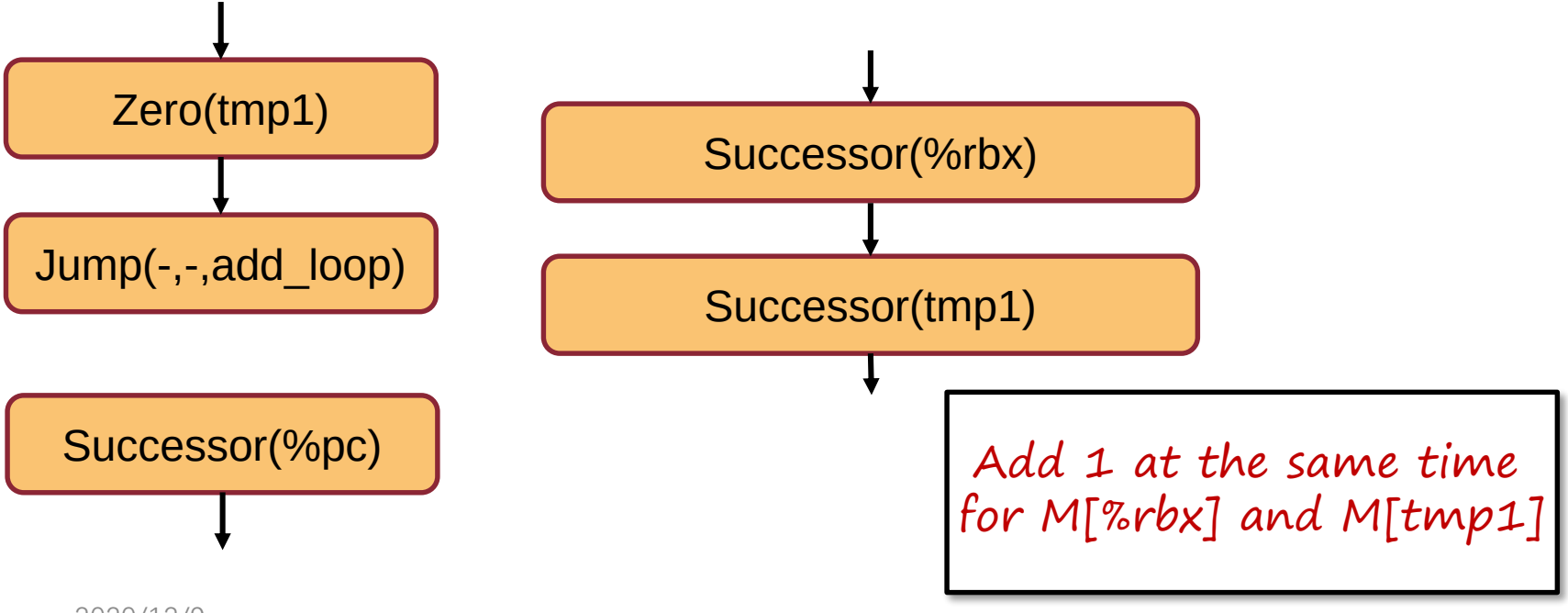
Example

- **addq %rax, %rbx**



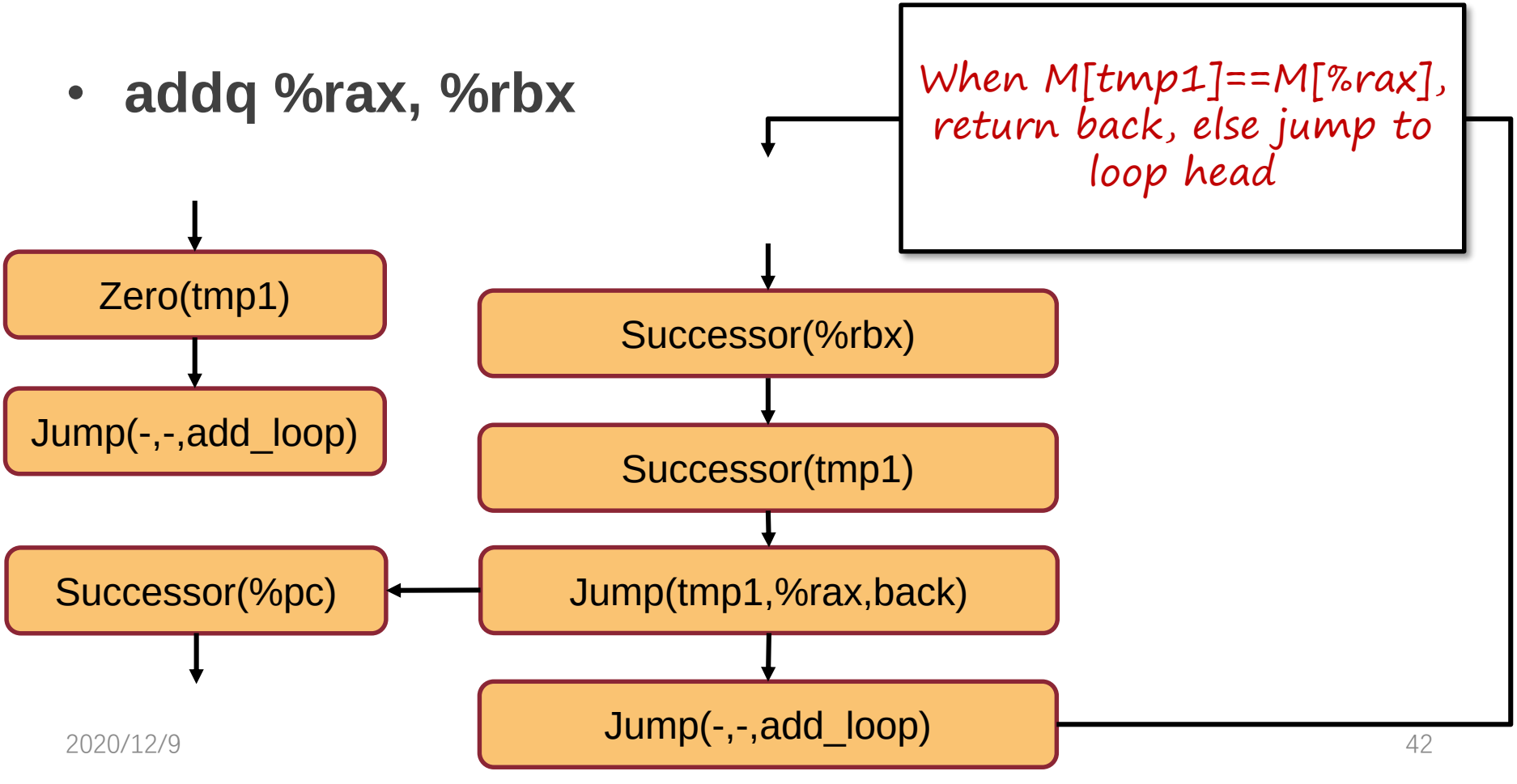
Example

- **addq %rax, %rbx**



Example

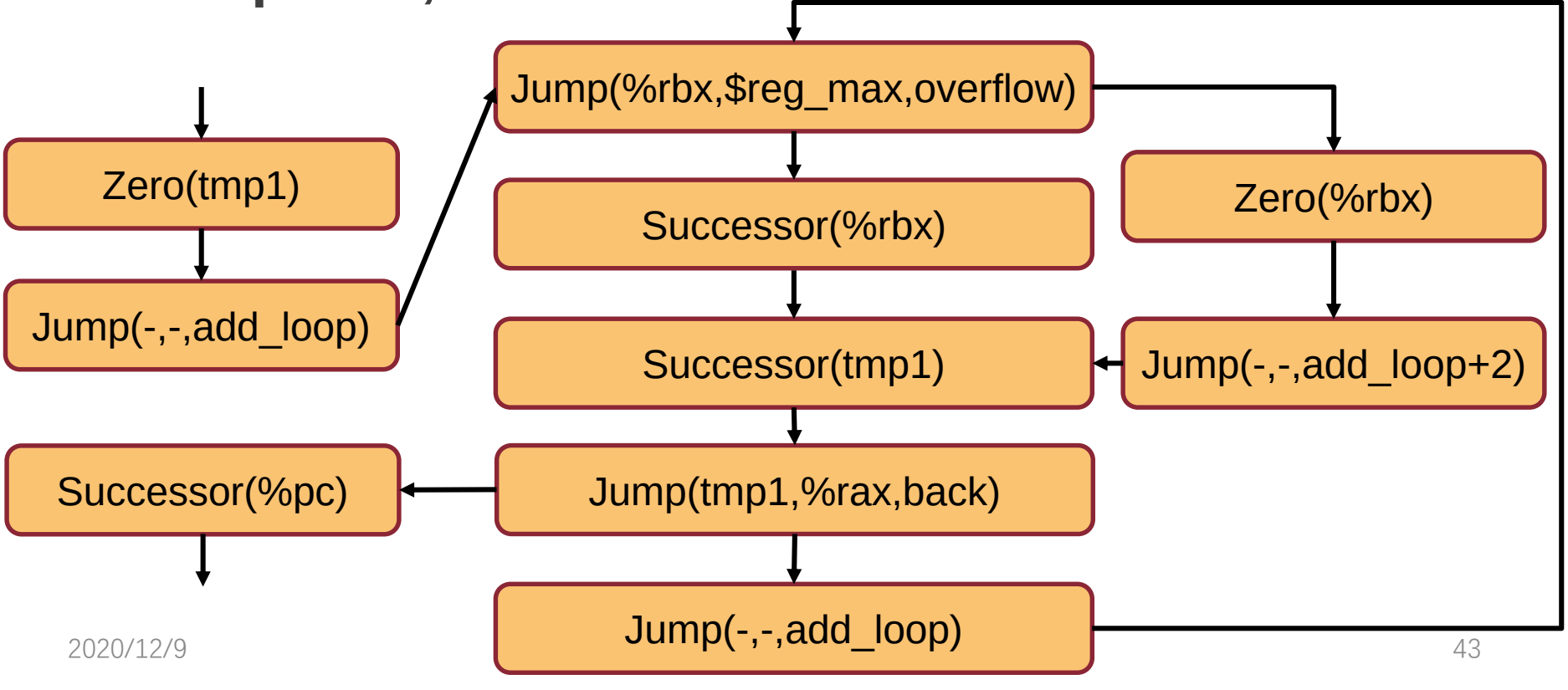
- **addq %rax, %rbx**



Example

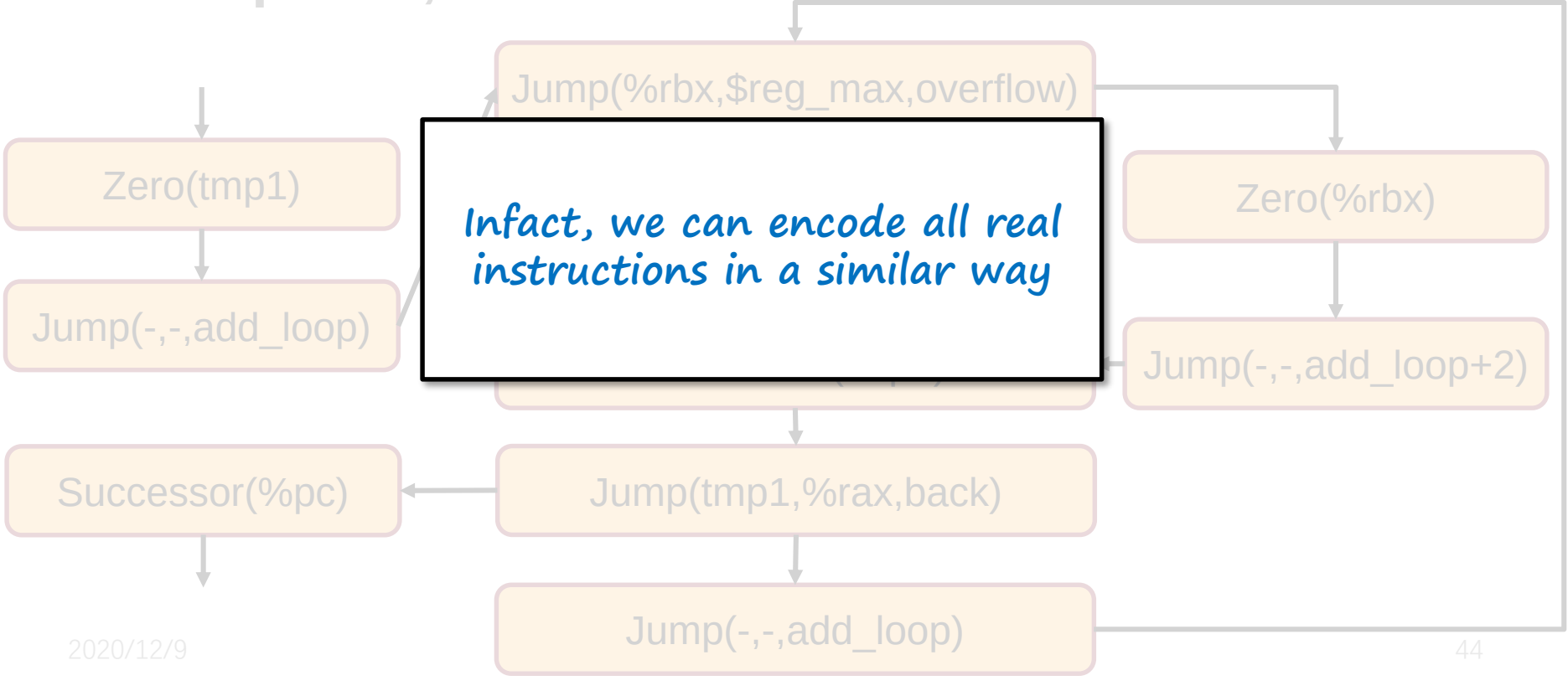
- **addq %rax, %rbx**

Don't forget overflow!



Example

- addq %rax, %rbx

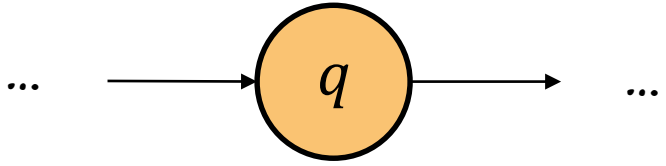


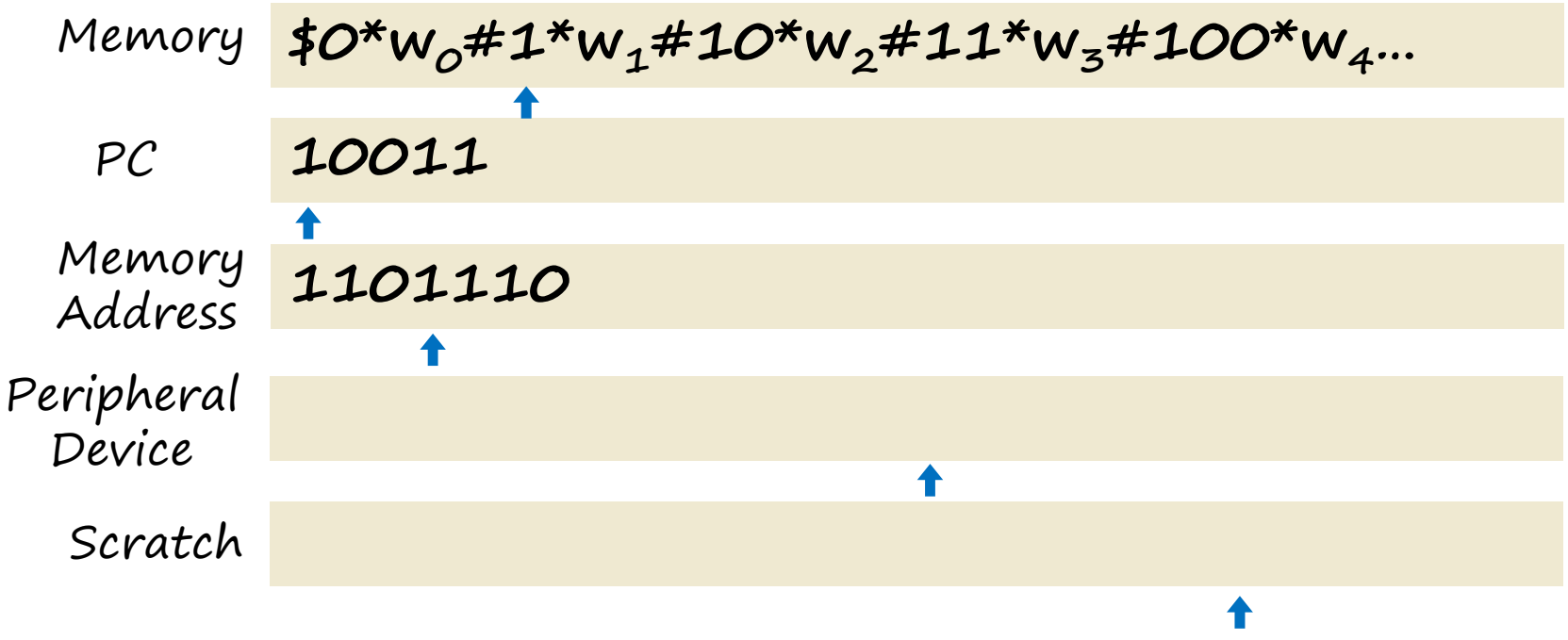
Infact, we can encode all real instructions in a similar way

Idealized computer & Real computer

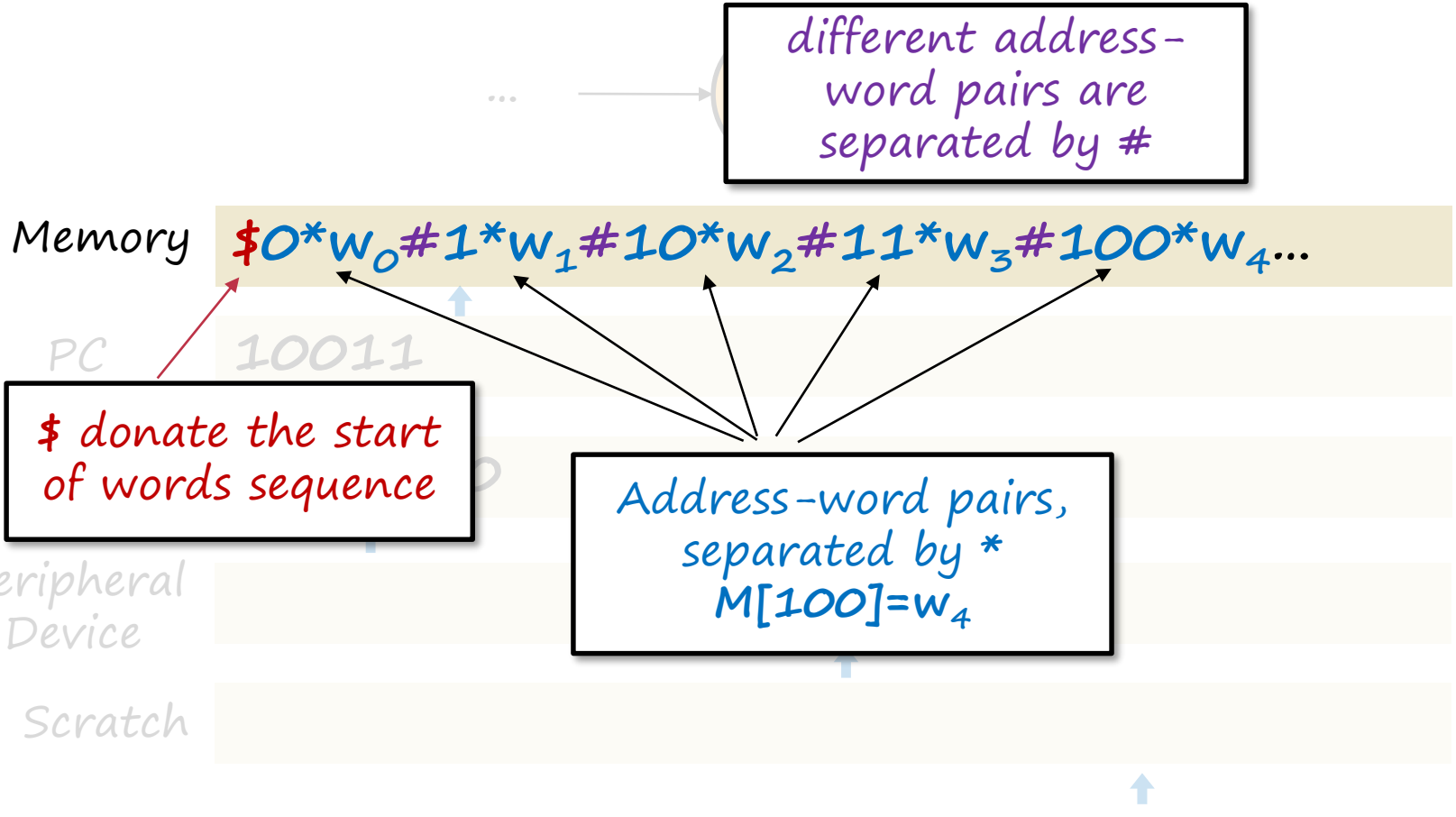
- **Estabilish a connection**
  - Computer memory & Turing machine tape region
  - Register & Turing machine tape region
  - Real instructions & Simple instructions
- **Next, we focus on idealized computers**
  - States on Tapes
  - Simple instructs(Zero, Successor, Mov, Jump)

A TM Simulates Idealized Computers.

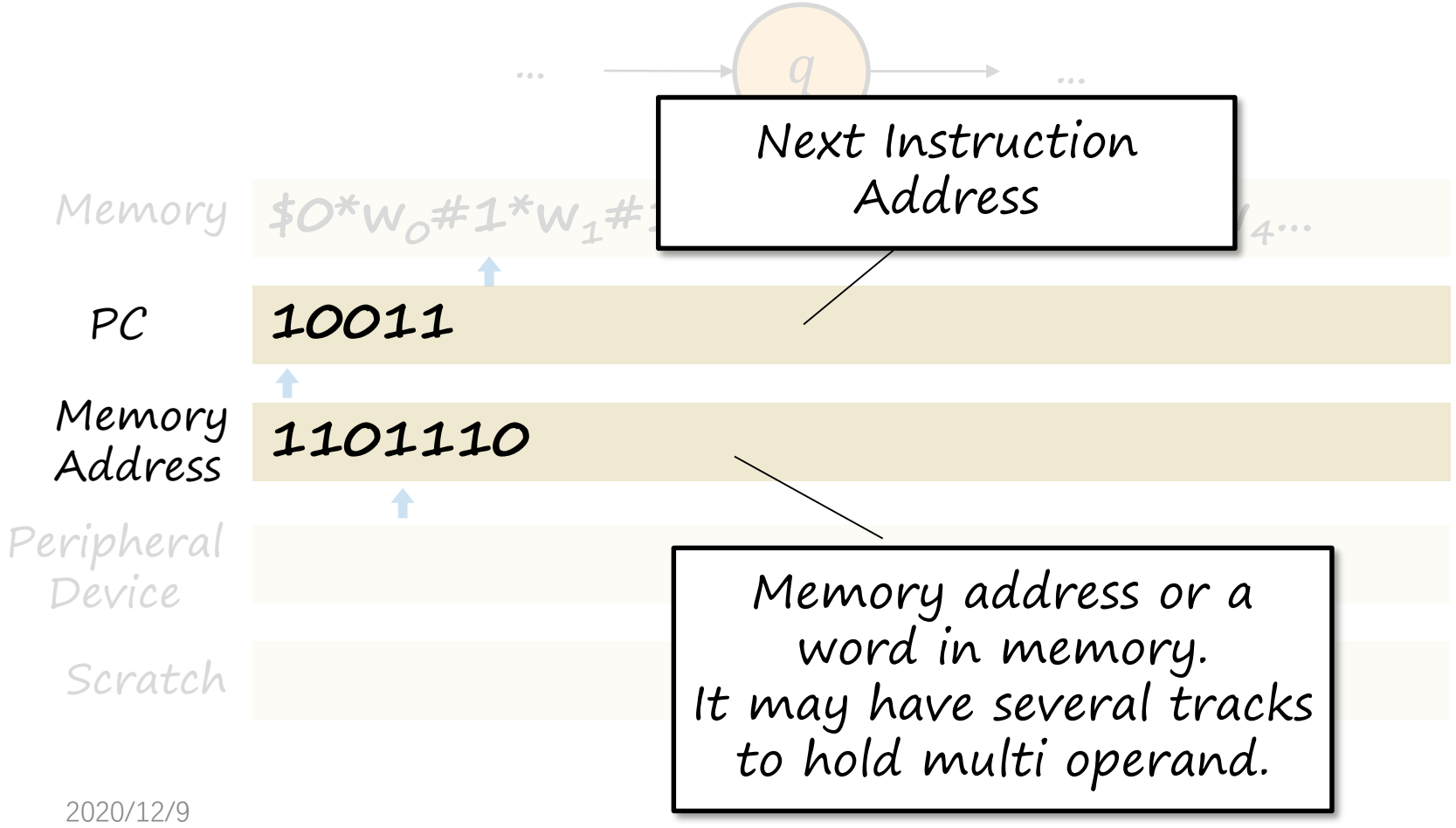




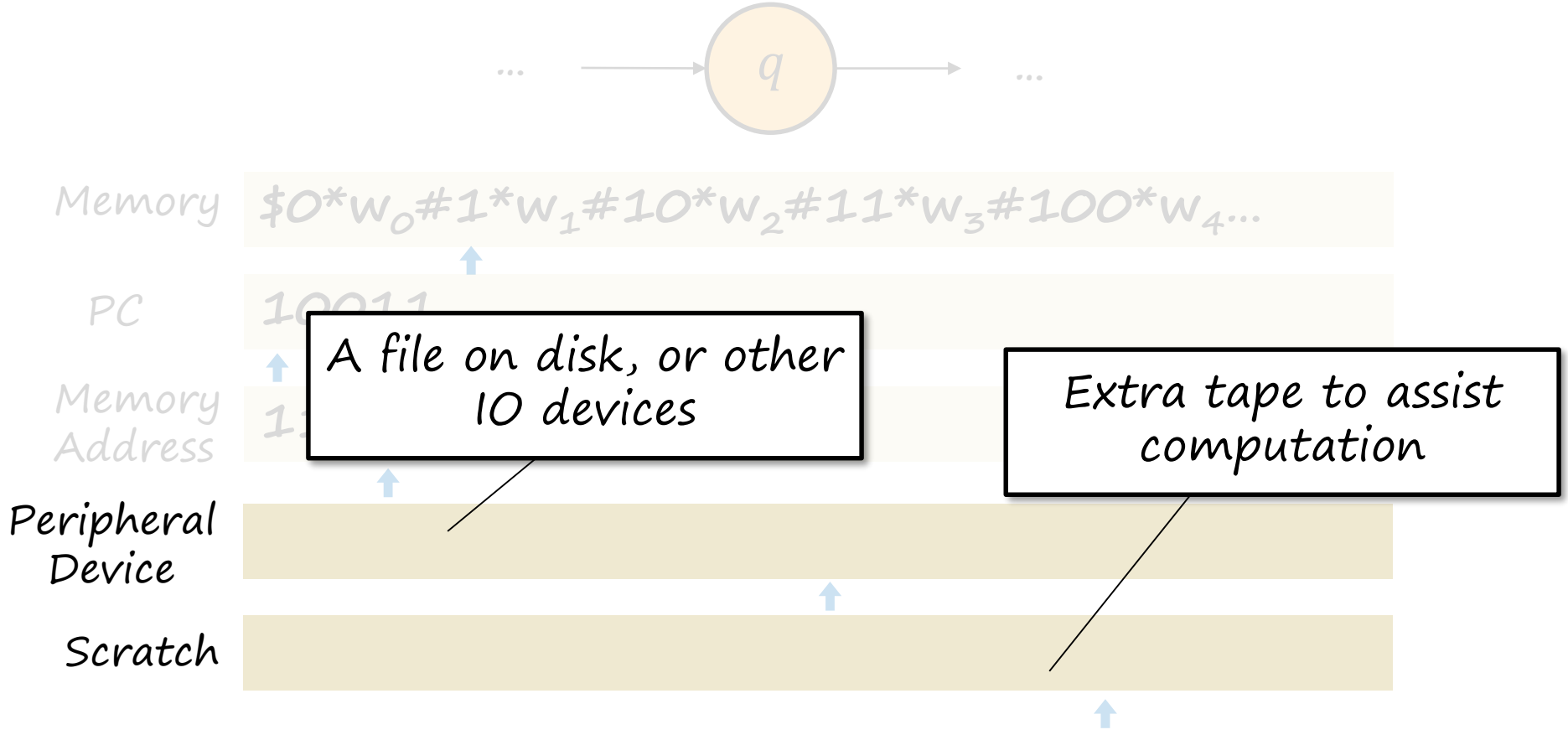
A TM Simulates Idealized Computers.



A TM Simulates Idealized Computers.



A TM Simulates Idealized Computers.



A TM Simulates Idealized Computers.

Memory $\$0*w_0\#1*w_1\#10*w_2\#...\#10011*w_{19}\#...$

PC 10011

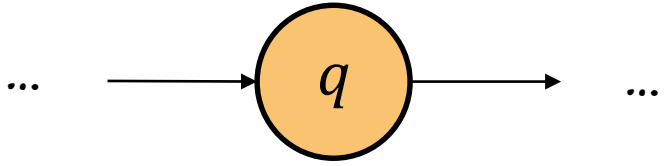
Memory Address

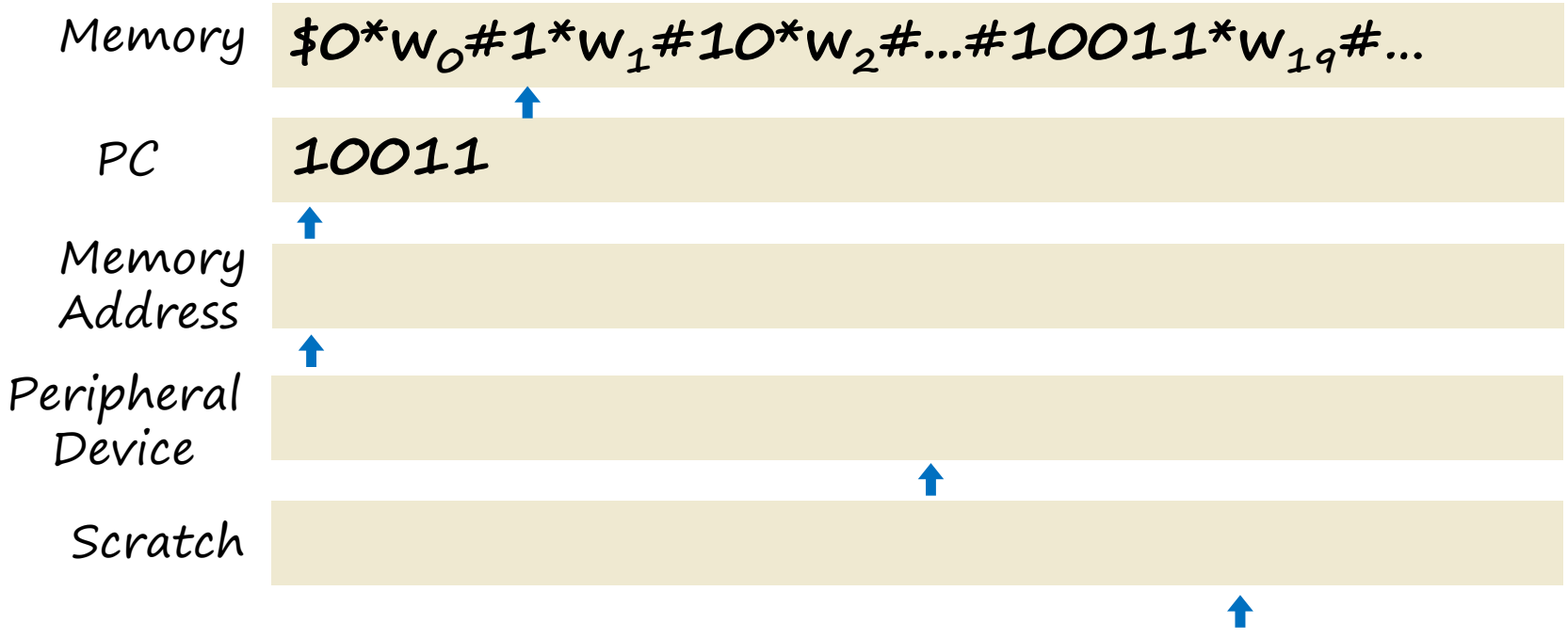
Peripheral Device

Scratch

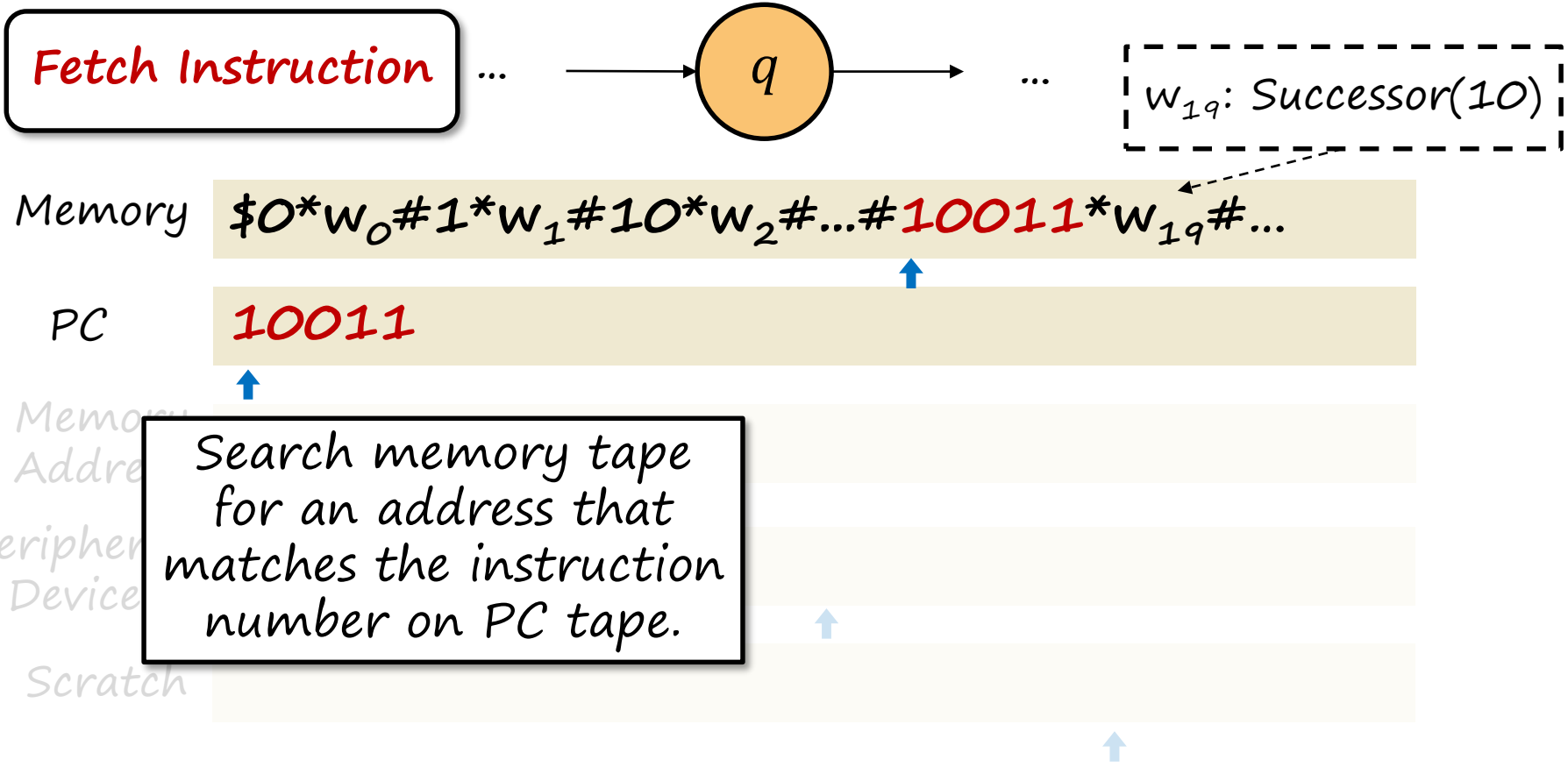
Now, we let the computer take a step forward

A TM Simulates Idealized Computers.

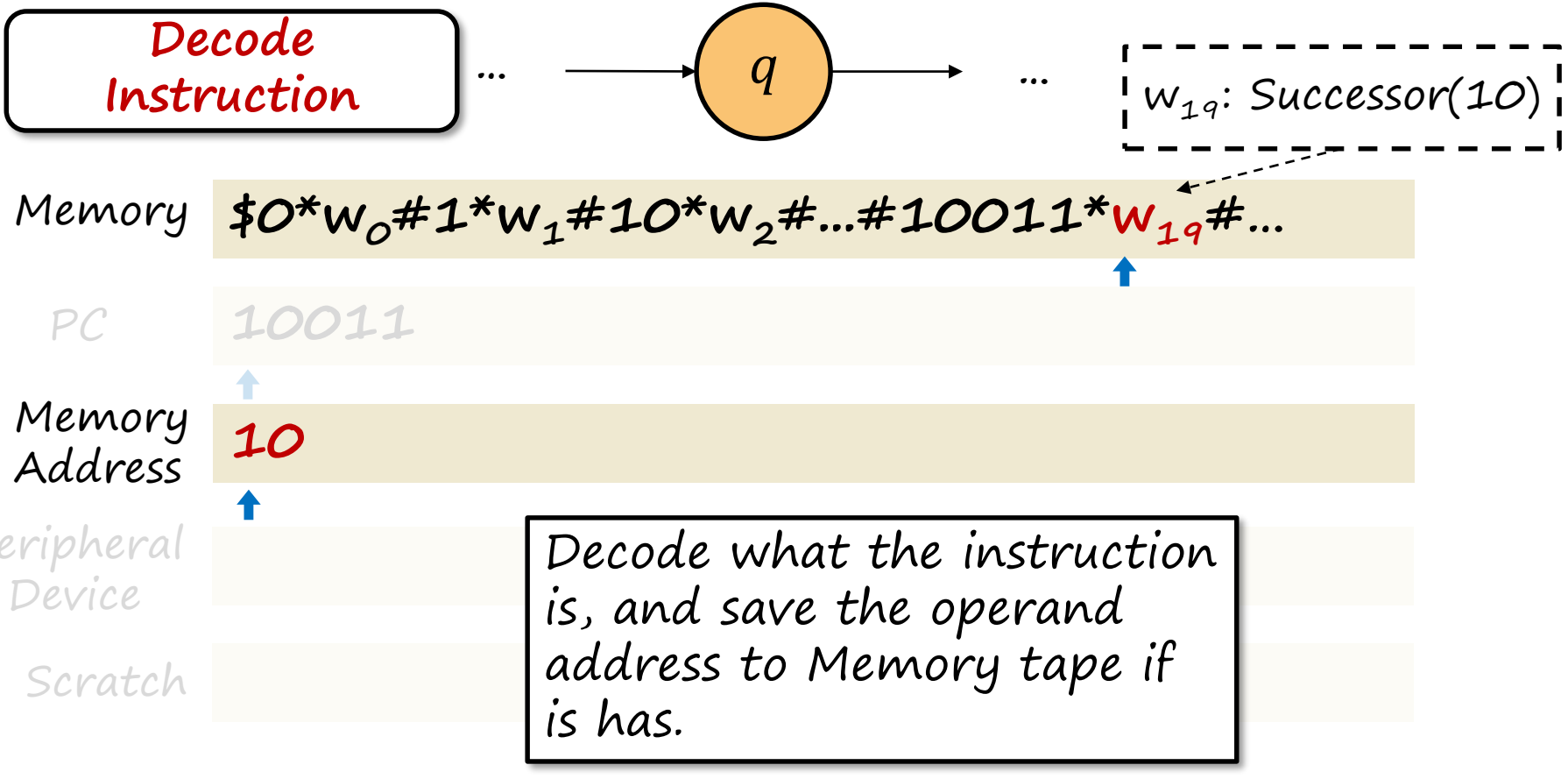




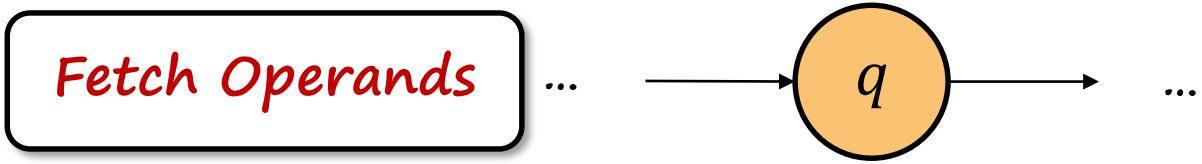
A TM Simulates Idealized Computers.



A TM Simulates Idealized Computers.



A TM Simulates Idealized Computers.



Memory $\$0*w_0\#1*w_1\#10*w_2\#...\#10011*w_{19}\#...$

PC 10011

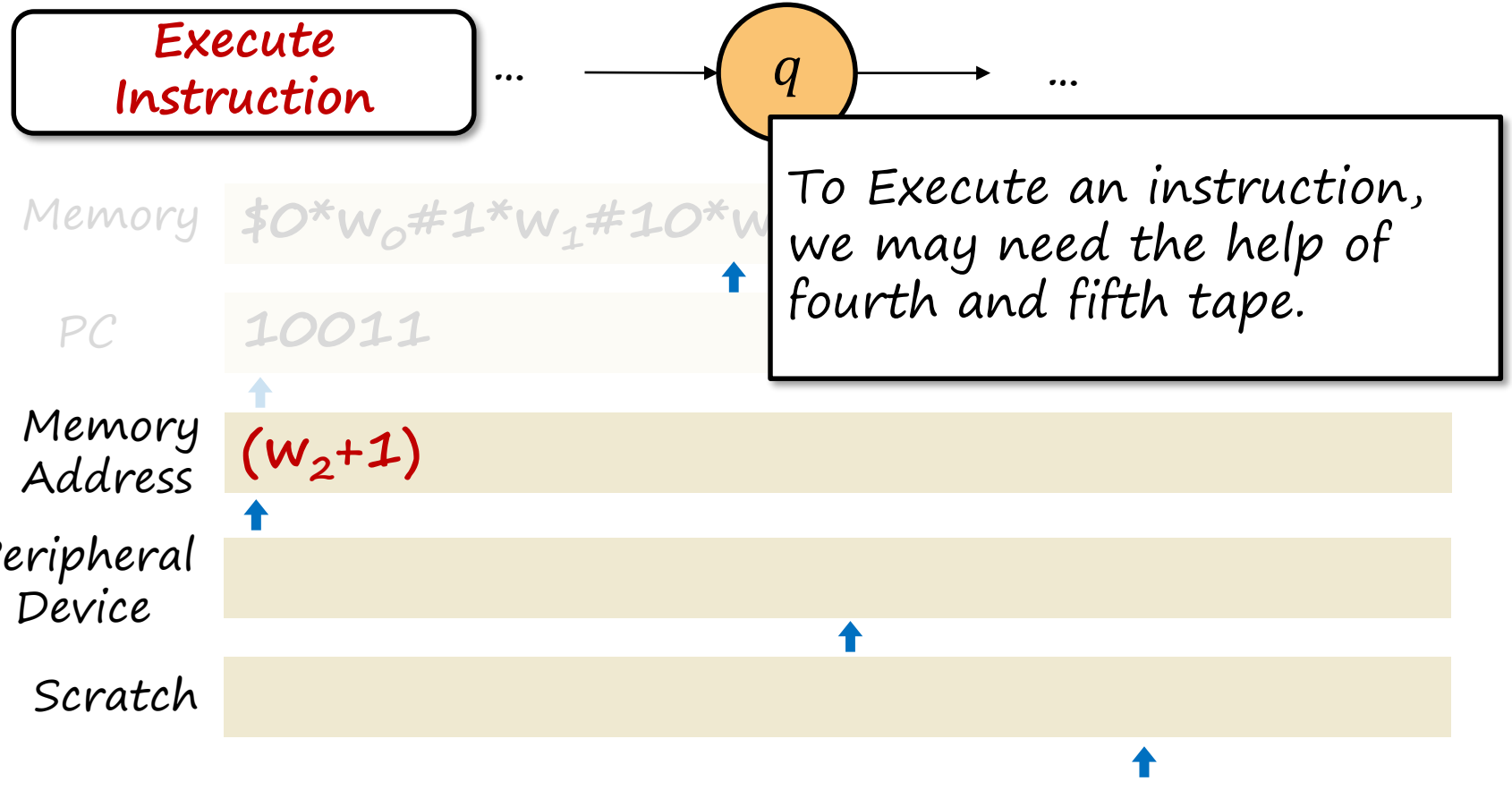
Memory Address w_2

Peripheral Device

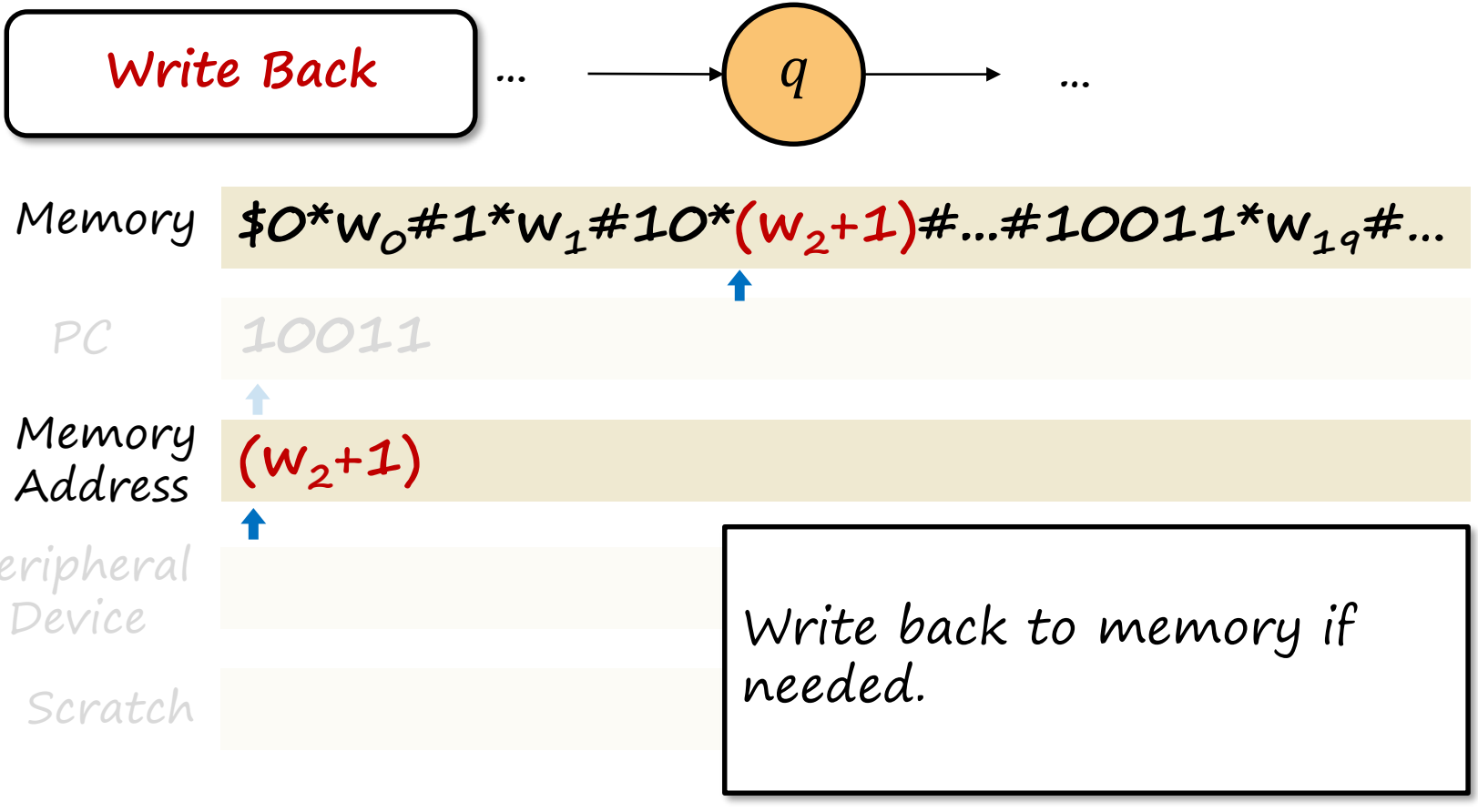
Scratch

Search memory for address that matches the address in third tape, copy its word into third tape.

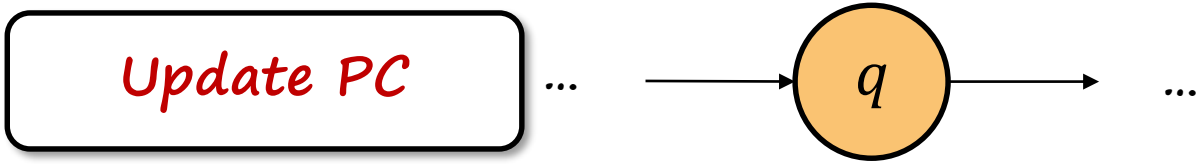
A TM Simulates Idealized Computers.



A TM Simulates Idealized Computers.



A TM Simulates Idealized Computers.



Memory $\$0*w_0\#1*w_1\#10*w_2\#...\#10011*w_{19}\#...$

PC 10100

Memory Address

Peripheral Device

Scratch

If it is not jump instruction, update PC pointed to next instruction.

TMs and Computers

- We’ve informally proved that a TM can simulate computers by simulating its instruction cycle.
  - It may require a leap of faith to believe since we do not prove it rigorously.
- So we know TM and computer can simulate each other, which means that TM has same computation capability with computer.

$$TM \approx Computer$$

Effective Computation

- An ***effective method of computation*** is a form of computation with the following properties:
  - The computation consists of a set of steps.
  - There are fixed rules governing how one step leads to the next.
  - Any computation that yields an answer does so in finitely many steps.
  - Any computation that yields an answer always yields the correct answer.
- This is not a formal definition. Rather, it's a set of properties we expect out of a computational system.

Church-Turing Thesis

> Every effective method of computation is either equivalent to or weaker than a Turing machine.

- This is not a **theorem** but a falsifiable scientific hypo**thesis**. But it has been thoroughly tested! So we have strong faith in its correctness.
- This means Turing Machine can model any “computation”.

~~What problems can a computer solve?~~

~~What languages can a computer recognize?~~

What languages can TM recognize?

Next

Effective Computation on TM

- Decidable, recognizable. R & RE language
- The properties of R & RE languages.
- The universal TM.