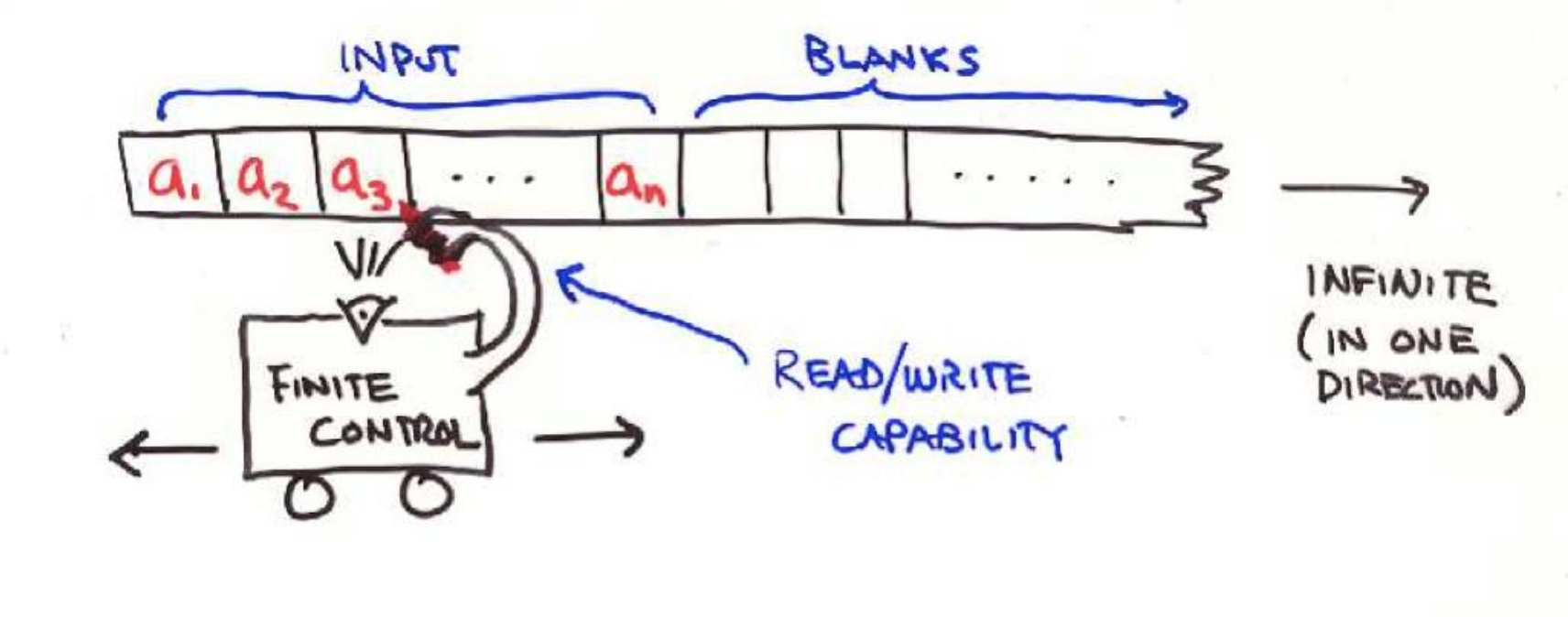
Universal Turing Machines & Church-Turing Thesis & Undecidability

TM recap



- DFA with (infinite) tape.
- One move: read, write, move, change state.

Transition Function

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

current state

symbol scanned

new state

symbol to write

direction to move on tape

$\delta(q,a) = (p, b, L)$

from state q, on reading a:

go to state p

write b

move head Left

Example/Refresher

TM that adds two unary numbers.

7 + 4:	$0000000#0000	initial tape contents
= 11:	$00000000000	final tape contents

Strategy?

- go right to first blank, turning # into 0
- back up one cell, erase 0
- return to first cell

Example/Refresher

7 + 4: $\underline{0}000000\#0000$ initial tape contents

Strategy?

- go right to first blank, turning # into 0
  $\delta(q_0,0) = (q_0,0,R)$
  $\delta(q_0,\#) = (q_0,0,R)$
- back up one cell, erase 0
  $\delta(q_0,B) = (q_1,B,L)$
  $\delta(q_1,0) = (q_2,B,L)$
- return to first cell
  $\delta(q_2,0) = (q_2,0,L)$
  $\delta(q_2,\$) = (q_{halt},\$,R)$

TM programming tricks

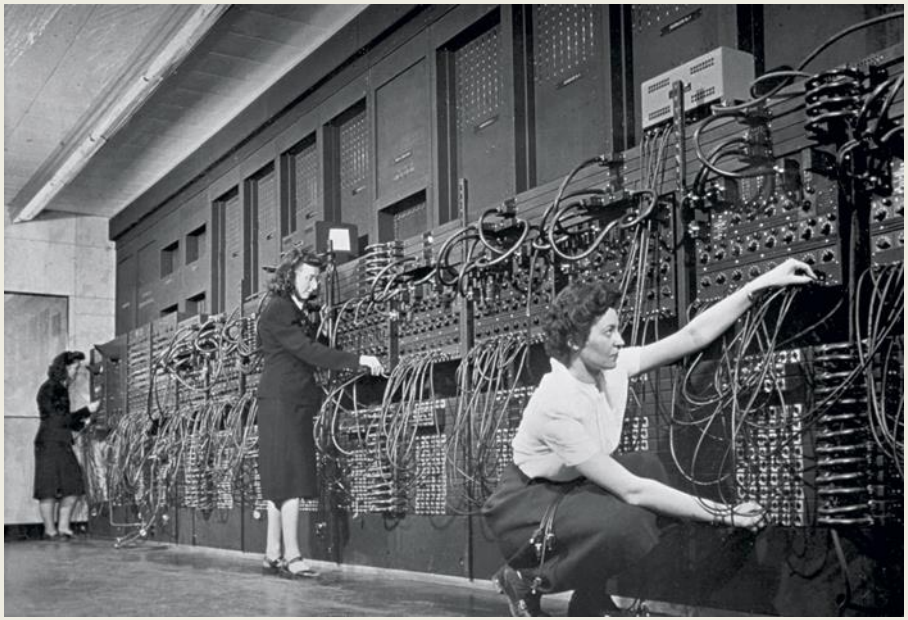
- checking off symbols
- shifting over
- using finite control memory
- subroutine calls

TM "extensions"

- 2-way infinite tape
- multiple tracks
- multiple tapes

Special purpose machines?

- Different DFA for different languages (duh)
- Different TMs for different languages, functions.
- Early computer programming was no different



Von Neumann Architecture

- stored-program computer
  - programs can be data!
  - program-as-data determines subcircuits to employ
- fetch-decode-execute cycle
- hence, one computer can behave like any

http://idiomzero.blogspot.com/2010/07/8-anecdotes-about-john-von-neumann.html

Original Idea was due to Turing

"I know that in or about 1943 or '44 von Neumann was well aware of the fundamental importance of Turing's paper of 1936 ... Von Neumann introduced me to that paper and at his urging I studied it with care. Many people have acclaimed von Neumann as the "father of the computer" (in a modern sense of the term) but I am sure that he would never have made that mistake himself. He might well be called the midwife, perhaps, but he firmly emphasized to me, and to others I am sure, that the fundamental conception is owing to Turing— in so far as not anticipated by Babbage ... "

- Stan Frankel – Los Alamos

Universal TM

- A *single* TM M_u that can compute anything computable!
- Takes as input
  - the ***description*** of some *other* TM *M*
  - data *w* for *M* to run on
- Outputs
  - the results of running *M*(*w*)

Need to make precise what the *description* of a TM is

Coding of TMs

- Show how to represent every TM as a natural number
- *Lemma*: If L over alphabet {0,1} is accepted by some TM M, then there is a one-tape TM M' that accepts L, such that
  - $\Gamma = \{0,1,B\}$
  - states numbered 1, ..., k
  - q_1 is the unique start state
  - q_2 is the unique halt/accept state
  - q_3 is the unique halt/reject state
- So, to represent a TM, we need only list its set of transitions – everything else is implicit by above

Listing Transition

- Use the following order:

  $\delta(q_1,0)$, $\delta(q_1,1)$, $\delta(q_1,B)$, $\delta(q_2,0)$, $\delta(q_2,1)$, $\delta(q_2,B)$,...

  ... $\delta(q_k,0)$, $\delta(q_k,1)$, $\delta(q_k,B)$.

- Use the following encoding:

  111 t_1 11 t_2 11 t_3 11 ... 11 t_{3k} 111

where t_i is the encoding of transition *i* as given on the next slide.

Encoding a transition

Recall transition looks like $\delta(q,a) = (p, b, L)$

So, encode as

<state> 1 <input> 1 <new state> 1 <new-symbol> 1 <direction>

where

- state q_i represented by 0^i
- 0, 1, B represented by 0, 00, 000
- L, R, S represented by 0, 00, 000

$\delta(q_3,1) = (q_4, 0, R)$ represented by 0001001000010100

q_3 1 q_4 0 R

Typical TM code:

111010100001001001101001000001010 11.....11.....11.....111

- Begins, ends with 111
- Transitions separated by 11
- Fields within transition separated by 1
- Individual fields represented by 0s

TMs are (binary) numbers

- Every TM is encoded by a unique element of N
- Convention: elements of N that do not correspond to any TM encoding represent the "null TM" that accepts nothing.
- Thus, every TM is a number, and vice versa
- Let <*M*> mean the number that encodes *M*
- Conversely, let M_n be the TM with encoding *n*.

Universal TM M_u

Construct a TM M_u such that

$L(M_u) = \{ \langle M \rangle \# w \mid M \text{ accepts } w \}$

Thus, M_u is a stored-program computer.

It reads a program $\langle M \rangle$ and executes it on data w

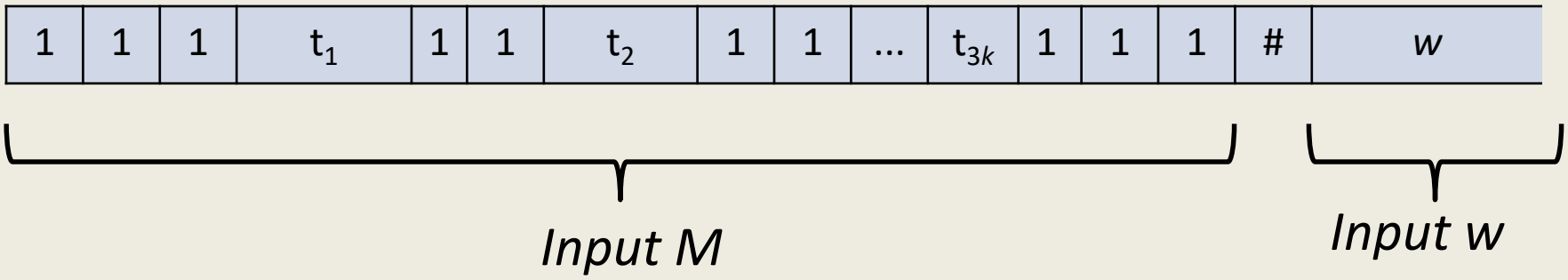
M_u simulates the run of M on w

A single TM captures the notion of "computable" !!

How M_u works

3 tapes

- Tape 1: holds input *M* and *w*; never changes
- Tape 2: simulates *M*'s single tape
- Tape 3: holds *M*'s current state



Universal TM M_u

Phase 1: Check if <*M*> is a valid TM on tape 1

- No four 1's in a row
- Three initial, ending 1's
- substring 110^i10^j1 doesn't appear twice
- appropriate number of 0's between 1's in transition codes: 1100001010000010000 1...

  (0000 does not encode a 0,1,or B to write)
- could check that transitions are in right order, and form a complete set (but not necessary)
- etc.

If not valid, then halt and reject

Phase 2: Set up

- copy *w* to tape 2, with head scanning first symbol
- write 0 on tape 3 indicating *M* is in start state q_1

Tape 1

111010100001001001101001000001010 11.....111 # 100110

Code for *M*

Tape 2

$100110

Current contents of *M*'s tape

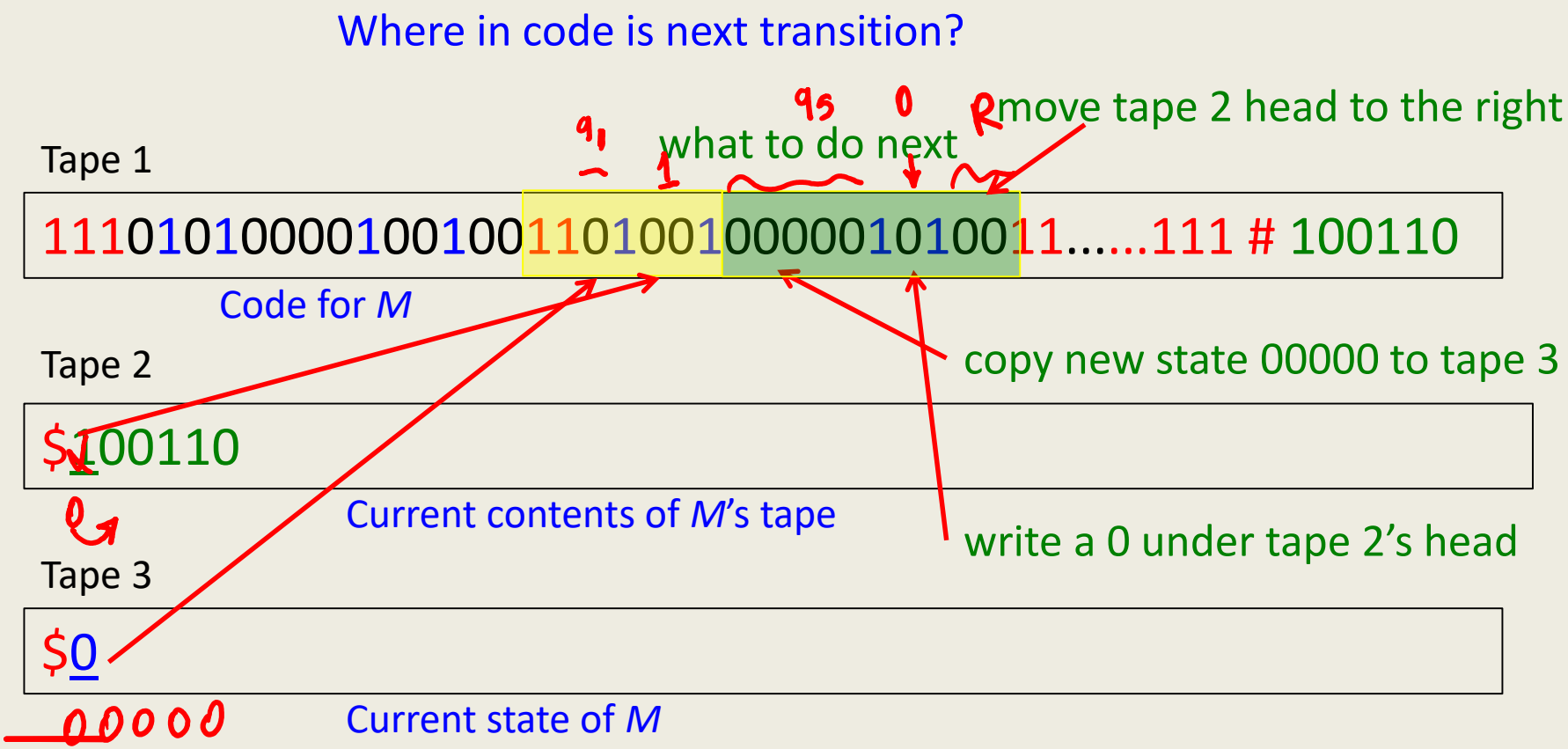
Tape 3

$0

Current state of *M*

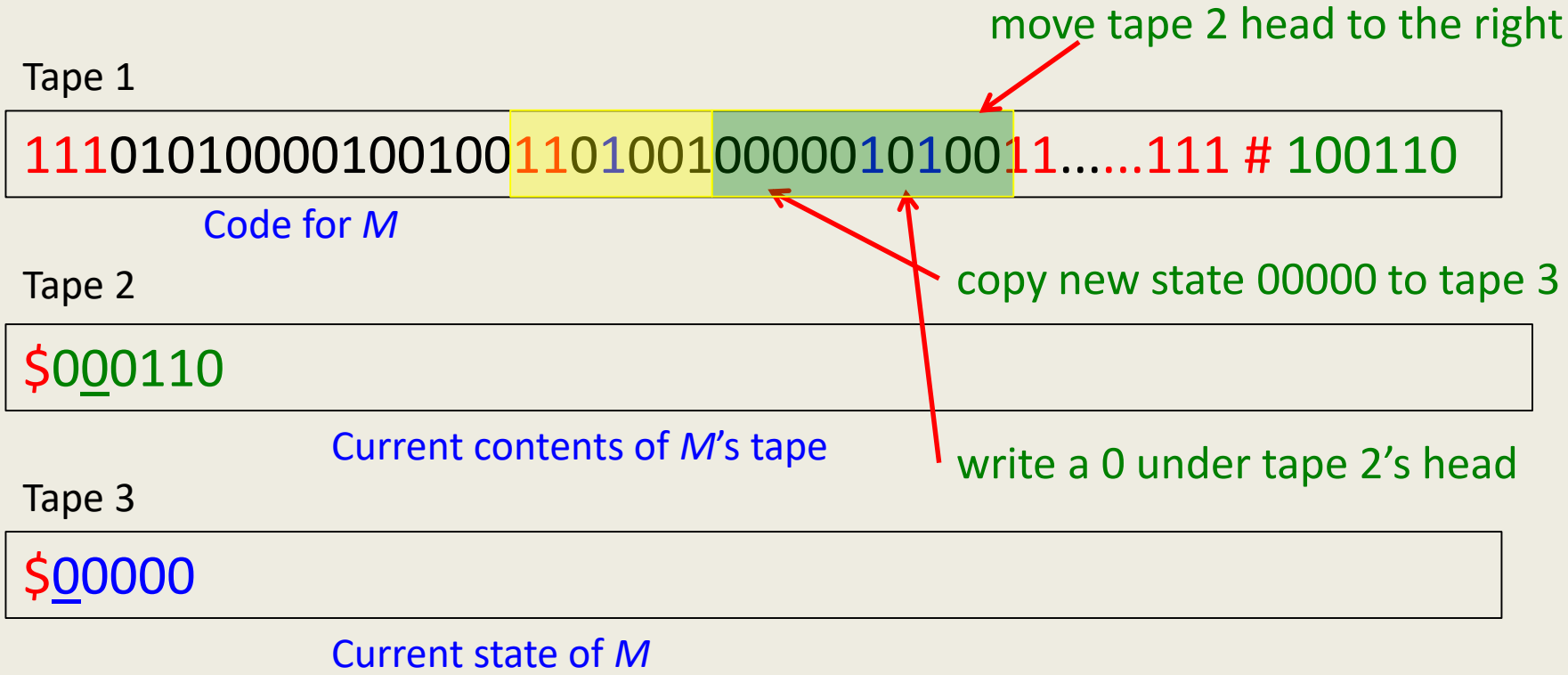
If at any time, Tape 3 holds 00 (or 000), then halt and accept (or reject)

Phase 3: Repeatedly simulate steps of *M*



If tape 3 holds 0^i and tape 2 is scanning 1, then search for substring 110^i1001 on tape 1

Phase 3: After the single move



Check if 00 or 000 is on tape 3; if so, halt and accept or reject

Otherwise, simulate the next move by searching for pattern.

In this example, the next pattern = 1100000101

Exercise

- Show how UTM on input <M>#w#t where t is a binary number simulates M on w for t time steps
- Is quite useful in simulating
  - Multiple machines in parallel
  - Dovetailing
  - Etc

Towards "real" computers: RAMs

Random Access Machine:

- finite number of arithmetic registers
- infinite number of memory locations
- instruction set (next page)
- program instructions written in continuous block of memory starting at location 1 and all registers set to 0.

RAM instruction set

Instruction	Meaning
Add X, Y	Add contents of register X and Y, and place result in register X
LOADC X, num	Place constant num in register X
LOAD X, M	Put contents of memory loc M into register X
LOADI X, M	Indirect addressing: put value(value(M)) into register X
STORE X, M	Copy contents of reg X into mem location M
JUMP X, M	If register X = 0, then next instruction is at memory location M (otherwise, next instruction is the one following the current one, as usual)
HALT	Halt (duh)

TMs can simulate RAMs

- Can write a “TM-interpreter” of RAM code Thus, no more TM programming.
- Actual simulation has low overhead (though memory is not random-access).

TM tapes

- Instruction-location tape
  - stores memory location where next instruction is
  - initially contains only “1”
- Register tape
  - stores register numbers and their contents, as follows: # <reg-num> # <contents> # .. etc.
  - Example: suppose R1 has 11, and R4 has 101, and all other registers are empty. Then register tape:

$	#	1	,	1	1	#	1	0	0	,	1	0	1	#	.	.	.

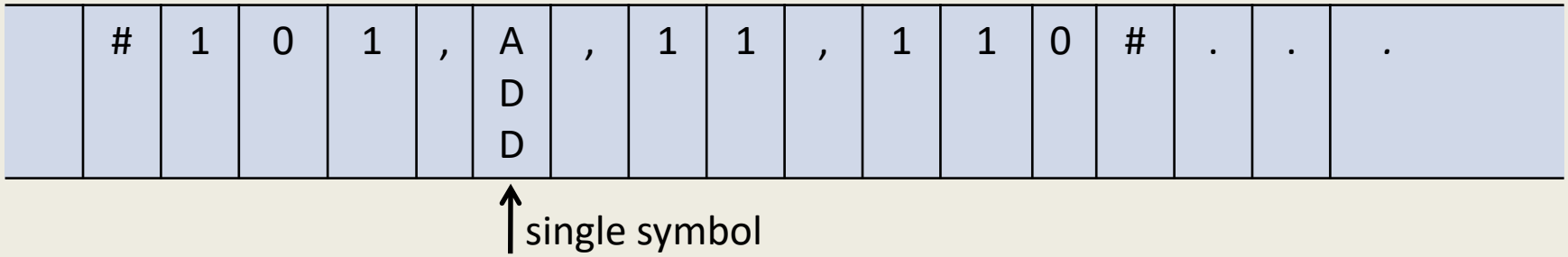
TM tapes

- Memory tape – similar to register tape, but can hold numbers, OR instructions:

numbers: # <mem-location> , <value> # ...

instructions:

example: mem location 101 holds ADD 3,6



- 5 work tapes

TM setup

- Blank register tape
- Memory tape holds RAM program, starting at memory location 1. No other data stored.
- 1 on instruction-location tape

TM step overview

(many TM steps for each RAM step)

- Read instruction-location tape
- search memory tape for the instruction
- execute the instruction, changing register and memory tapes as needed
- update the location-instruction tape

In other words, it goes through a fetch-decode-execute cycle

Example

- Suppose instruction location tape holds only:

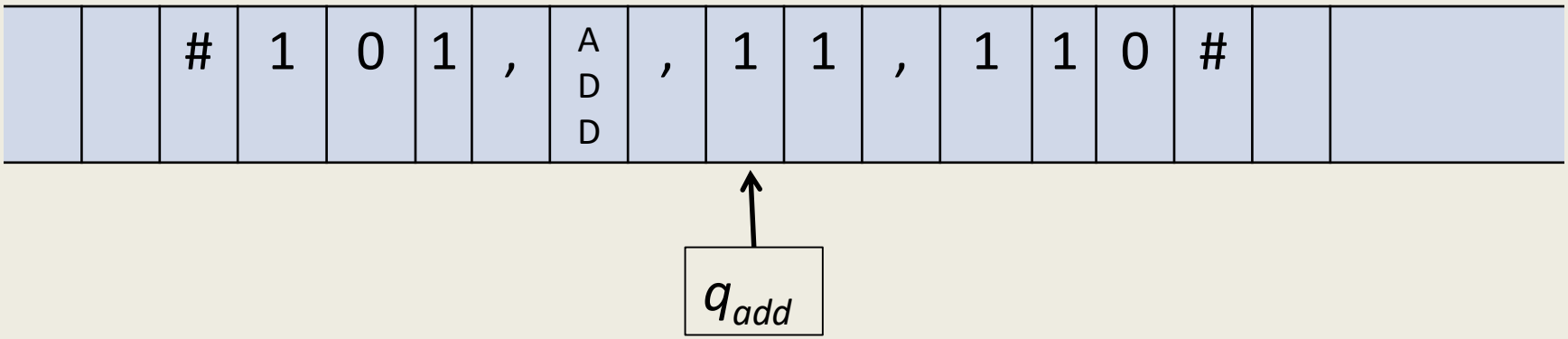
$	1	0	1													

- Scan memory tape, looking for "# 1 0 1 ," Suppose it finds

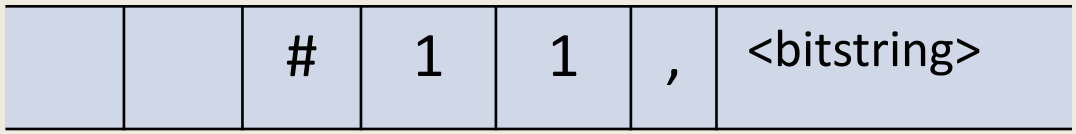
.	.	#	1	0	1	,	A D D	,	1	1	,	1	1	0	#		

- It finds "ADD" following "," and switches to special state q_{add} to handle the addition

Example (cont.)



- first argument is in register 11 so search register tape for:



- then copy <bitstring> to worktape 1
- similarly, search for, find, place value of register 110 onto worktape 2

Example (cont.)

- Now go to subroutine to add worktape 1 + worktape 2, place results on worktape 3.
- Result must go back into register 11
- Search register tape again for

		#	1	1	,	<bitstring>

- Replace <bitstring> with new value copied from worktape 3, shifting as necessary
- Add 1 to instruction-location tape

RAM simulation

- *MANY* details left out
- Other types of instructions are similar
- Number of steps to simulate RAM?
- Delicate issue.... does RAM actually have constant-time access to *infinite* memory?
- Can show (beyond this course) for "reasonable" time model on a RAM, if T(n) steps are required, then on a TM, only $T(n)^2$ steps. ($T(n)^3$ if RAM has mult. and div.)

Church-Turing thesis

- TMs capture notion of "computable"
- Evidence
  - RAM computer
  - general recursive functions (Gödel & Herbrand)
    - constant/projection/successor/composition/recursion
  - λ-calculus (Church) for defining functions (CS 421)
  - general string-rewriting-system
    - unrestricted grammar, productions of form $\alpha \rightarrow \beta$ for any α and β
  - attempts to extend TMs

All give you exactly the TM-computable functions

Undecidability

- Can a Turing Machine Compute anything?

No

- Some problems are undecidable!
- E.g. Halting Problem

Halting Problem

- Given a program *M* and a string *w, does M* halt when started on *w?*

No Algorithm to Solve This Problem

- Any program can be computed by a Turing machine *M.*

$$L_{Halt} = \{ \langle M \rangle \# w \mid \text{Turing Machine } M \text{ halts on input } w \}$$

Halting Problem

$$L_{Halt} = \{< M > \#w \mid \text{Turing Machine } M \text{ halts on input } w\}$$

- *Theorem*: L_{Halt} is not recursive i.e. undecidable.

- *Recursive Languages* *(also called "decidable")*

  = {L | there is a TM M' that halts for all w in Σ^*

  and such that L(M') = L }

Halting Problem

$$L_{Halt} = \{< M > \#w \mid \text{Turing Machine } M \text{ halts on input } w\}$$

- *Theorem*: L_{Halt} is not recursive i.e. undecidable.
- *Proof: By Contradiction*

Suppose there is some Turing Machine M_0 that halts for all x *in* Σ^* and accepts if x *in* L_{Halt}

Counterexample: $<M_{bad}>\#w_{bad}$

There is some program (TM) M_{bad} and string w_{bad} such that M_0 accepts $<M_{bad}>\#w_{bad}$ even if M_{bad} does not halt on w_{bad} and rejects $<M_{bad}>\#w_{bad}$ even if M_{bad} halts on w_{bad} .

Halting Problem

$$L_{Halt} = \{< M > \#w \mid \text{Turing Machine } M \text{ halts on input } w\}$$

- *Theorem*: L_{Halt} is not recursive i.e. undecidable.
- *Proof: By Contradiction*

M_{bad} :

Takes input <M>.
Runs M_0 on input <M>#<M>
If M_0 accepts,
    go to infinite loop
else
    halt.

w_{bad} : $<M_{bad}>$

$<M_{bad}>\#<M_{bad}>$

Case 1: M_0 accepts $<M_{bad}>\#w_{bad}$ i.e. halts
M_{bad} on input $<M_{bad}>$ goes to infinite loop
→ M_0 is wrong!

Case 2: M_0 rejects $<M_{bad}>\#w_{bad}$ i.e. doesn't halt
M_{bad} on input $<M_{bad}>$ halts → M_0 is wrong!