Bit masking:

0110 **AND** 0100 == 0100 (detects that bit_2 == 1)
0110 **AND** 1011 == 0010 (resets bit_2 == 0)
0110 **OR** 0100 == 0110 (sets bit_2 == 1)

Check that bit_7 in register R0 is 1, LC4 assembly language:

```
  LIM DR1 b010000000    ;---  R1 <== mask for bit_7
  ALU SR0 SR1 DR1 AND   ;---  R0 AND mask  ==> R1
                        ;---  if bit_7 was 1, R1 > 0
  ALU SR2 SR2 DR2 SUB   ;---  0 ==> R2
  ALU SR2 SR1 DR2 SUB   ;---  0 - R1 ==> R2
                        ;---  IF R2 < 0, bit_7 was 1:
  BRR CR2 AR7           ;---      go to THEN (R7 has address)
  ...                   ;---  ELSE:
                        ;---      code for bit_7 == 0
```

Check that bit_7 in register R0 is 1, LC3 assembly language:

```
  LD R1, mask           ;---  R1 <== mask
  AND R1, R0, R1        ;---  R1 <== R0 AND mask
                        ;---  IF bit_7 was on, R1 > 0
  BRp Bit_7_On          ;---      jump to THEN part of IF
                        ;---  ELSE:
                        ;---      code for bit_7 == 0
  ...
Bit_7_On:
  ...
mask:
  .FILL x0080           ;--- mask for bit_7
```

Check bit_15 (Ready) in keyboard status register, LC3 assembly language:

```
loop:
  LDI R1, kbsr          ;---  R1 <== kbsr
  BRn ready             ;---  IF R1[15] == 1, jump to ready
  BRnzp loop            ;---  ELSE, loop
ready:                  ;---  ready
  LDI R0, kbdr          ;---  get data from keyboard
  ...
kbsr:
  .FILL xFE00           ;--- pointer to keyboard status reg.
  .FILL xFE02           ;--- pointer to keyboard data reg.
```

TRAPs

Call the OS function (TRAP x21) that prints a character on the display. TRAP x21 expects that R0 contains the ASCII code of the character to be displayed. This program displays "EH?":

```
    LEA R7, chars      ;--- R7 <=== address of 'E'
    LDR R0, R7, #0     ;--- R0 <=== MEM[ R7 ] == 'E'
    TRAP x21           ;--- jump to TRAP function
    ADD R7, R7, #1     ;--- R7 <=== address of 'H'
    LDR R0, R7, #0
    TRAP x21
    ADD R7, R7, #1     ;--- R7 <=== address of '?'
    LDR R0, R7, #0
    TRAP x21
    ...
  chars:
  .FILL x0045     ;--- ASCII for 'E'
  .FILL x0048     ;--- ASCII for 'H'
  .FILL x003F     ;--- ASCII for '?'
```

Initialize the Vector Table entry for TRAP x21. Suppose the TRAP x21 code is in memory at address x0325. The memory cell at address x0021 gets the address to jump to when we want to execute the TRAP x21 function (i.e., MEM[x0021] gets the address x0325):

```
    LD R2, jumpAddress
    LD R3, VTaddress
    STR R2, R3, #0       ;--- x0325 ===> MEM[ x0021 ]
    ...
  jumpAddress:
    .FILL x0325     ;--- address of TRAP function's code
  VTaddress:
    .FILL x0021     ;--- address of TRAP's VT slot
```

The TRAP x21 function sends the data it receives in R0 to the display data register, then returns to the caller. Recall that the TRAP call loads R7 with the address immediately after the TRAP instruction.

```
(at address x0325):
    STI R0, ddr
    JMP R7
  ddr:
    .FILL xFE06     ;--- address of Display Data Register
```

Note that in this code we did not check whether or not the display was ready. We should have read the display status register (at address xFE04) and checked whether bit_15 == 1 before sending new data to the display data register.

Turing Machines

A Turing Machine (TM) consists of a Finite State Machine (FSM) and a read/write tape with a read/write head positioned at one cell of the tape. Each cell of the tape contains a single symbol. The FSM starts in some initial state, START. At the tick of a clock, the FSM changes its current state. It then reads the symbol in the tape cell where the R/W head is positioned; writes some symbol to the cell, and moves the R/W head one cell to the left or right. The actions of the FSM are determined by which state it is currently in and what symbol it sees in the current tape cell.

To describe a FSM we write down the rules for what it does in each of its states and what state it starts in. To fully describe a TM, we also need to describe the content of its tape cells and which cell the R/W is positioned on, just before it starts operating.

In describing a FSM, for each of its states, we must define what its actions will be for every possible input symbol it might see. The set of such symbols is the machine's "symbol set" or "alphabet".

We might adopt the convention that if there are no rules specified for a particular state, then the machine will halt when it gets into that state. Or, we could adopt the convention that if it writes a particular symbol, the FSM halts. The latter would correspond well to an implementation of the FSM as an electronic circuit in which one wire, when set to logic 1, would turn off the machine's clock and/or power supply. We usually adopt the convention that havnig no rules for a state means it halts if it ever enters that state.

An "instantaneous description" of a TM consists of a description of the machine's current state and the current content of its tape.

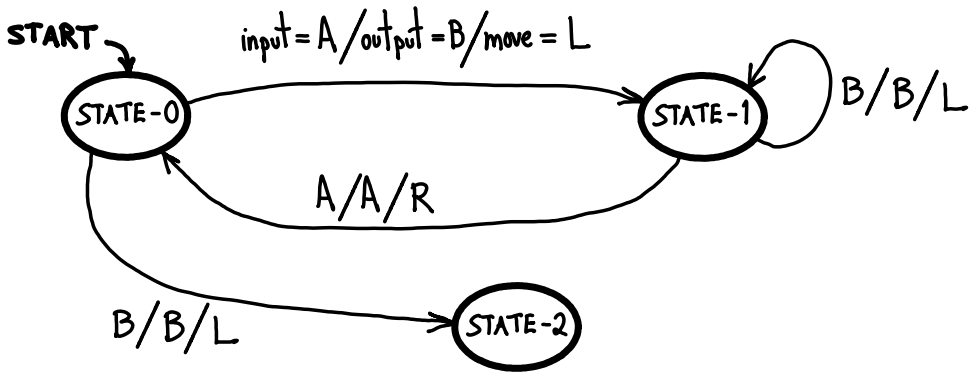
Here is a complete description of a TM:

it starts in state STATE-0;
if state = STATE-0 and input = A, write B, move LEFT, change to STATE-1;
if state = STATE-0 and input = B, write A, move LEFT, change to STATE-2;
if state = STATE-1 and input = A, write A, move RIGHT, change to STATE-0;
if state = STATE-1 and input = B, write B, move LEFT, change to STATE-1;

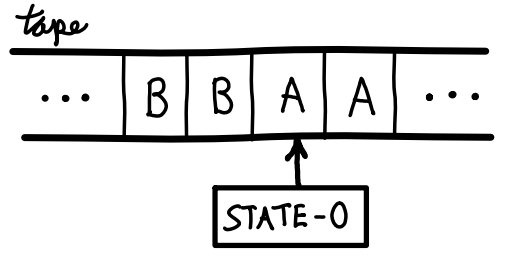
its R/W head is initially positioned on a cell containing a A, with B's in all the cells to the left and A's in all the cells to the right.

We can give the same information as a "state-transition" diagram and an "instantaneous" description:

State-Transition Diagram



initial Instantaneous Description



We can also give a description of the same TM in an encoded form. We make codes for each state and symbol, and a pattern for each rule:

Codes:

State	code	Symbol	code	Move	code
STATE-0	1	A	1	LEFT	1
STATE-1	11	B	11	RIGHT	11
STATE-2	111				

Rules:

current-state	input	output	move	next-state
1	1	11	1	11
1	11	11	1	111
11	1	1	11	1
11	11	11	1	11

Because there are no rules for STATE-2 (111), we assume the machine halts in STATE-2. By adding a few "0"s, we can encode the entire table as a single bit string:

0000	1	0	1	0	11	0	1	0	11	00
	1	0	11	0	11	0	1	0	111	00
	11	0	1	0	1	0	11	0	1	00
	11	0	11	0	11	0	1	0	11	0000

By following the rule patterns, we can identify each rule and each part of a rule. We can also encode the machine's intial tape and initial state. The instantaneous description of a tape containing alternating A's and B's with the R/W head reading the cell containing the third A from the left and the machine in state STATE-0 is,

```
1000 1 0 11 0 1 0 11 00 1 0 11 0 1 0 11 0000 1 0000
```

The "1000" indicates the left extent of the encoded part of the machine's tape. We make the assumption that farther to the left (or right) the tape is blank. Each encoded tape cell is demarcated with "0", and the content of the cell is shown as "1" (for A) or "11" (for B). The location of the R/W head is indicated by "00", which we can assume means the head is reading the symbol to the right of this indicator. The current state code is bracketed by "0000". The entire rule table and instantaneous description can be written as a single binary string,

```
1000101101011001011010110000100000000101011010110010110110101110011010101101001101101101011000
```

One interesting aspect of this is that any question we might want to ask about a machine plus its input can be thought of as a question about integers. In particular, one very interesting question is, Which integers encode TM's and their input such that the encoded TM started in the situation indicated by the encoded instantaneous description will eventually halt.

We can output a 1 if the TM would halt, and output a 0 otherwise. That makes it into a function that maps the integers onto the set {0, 1}. This function is called the "Halting Problem Function."

The Halting Problem Function is similar to another question about integers, "Which integers are divisible by 2?" We can change this question into a function by saying the answer is either 1 if the integer is even, and 0 otherwise. It is easy to show a TM that can compute this function: just have the input integer encoded in binary and look at the rightmost bit: if it is 0, the integer is even and output a 1 on the tape; otherwise, output a 0.

Turing showed that there is no TM that can compute the Halting Problem Function. There exist (at least mathematically), functions that no TM can compute. Thus, we can never write a program to determine before we try it, whether another program reading some input data will get stuck forever in a loop. Even if we try running the program, we cannot be sure the machine might not eventually halt at some long time in the future; so, our testing method cannot terminate unless the program we are testing does.

Finally, given an instance of a TM and its input, the Halting Problem can be formulated as a logical hypothesis, e.g., "This TM reading this input x will halt." We can then ask whether this is a true statement. Hilbert's aim was to show that any logical statements could be "mechanically" shown to be either true or false. If Hilbert were right, we could use the method to prove any Halting Problem statement. Which would be the same thing as saying the Halting Problem can be computed. Because we already know that is impossible, Hilbert's problem is also not computable. This is called "reducing Hilbert's problem to the Halting Problem."