Machine generated alternative text:
Abstraction• 
Implementation: 
Assumptions: 
• SPstoredinRAM[01 
Stack machine 
stack 
17 
SP 
o 
1 
2 
256 
12 
5 
258 
12 
5 
push constant 17 
Logic: 
•sp 
Hack assembly: 
• Stack base addr 256 
257 
258 
@17 
@SP 
@SP 
// 0=17 
// •sp.D 
o 
1 
2 
256 
257 
258 
stack 
12 
5 
17 
259 
12 
5 
17 

C:\users\shiraf\AppData\Local\Temp\msohtmlclip1\02\clip_image002.png

SP=RAM[0]

SP starts out at 256

With @x, A= RAM[x] while M=RAM[RAM[x]]

And M also = RAM[A]? What if A changes? Then M wouldn't change, because only @x can change the A and M values

NO it is M that relies on A!! Only A can change the M value

@x affects A; A then = x. But it can be changed after that, until the next @

A affects M; M then = RAM[A]. But it can be changed after that, until the next @

M always becomes RAM[A] after

The Rules

1. @x makes A=x and M=RAM[x]
2. A and M can then be reassigned

The real question is, is M pass by reference? Like, if A=2 and M=RAM[2] which happens to be 5, then does M++ change 1)only its own value, to 6 2)the value of its own value and of RAM[2], both to 6, or 3)the value of 2, to 3, making it RAM[3]? Okay I don't think it does 3.

Also, does changes in M change A?

Ok hopefully this answers something:

C:\users\shiraf\AppData\Local\Temp\msohtmlclip1\02\clip_image001.png

Machine generated alternative text:


Machine generated alternative text:
Register RAM[I] has the value 0. 
Correct 
The value of the A register is 1, so the assignment in the second line assigns A-I (which 
equals 0) to RAM[I] 

Machine generated alternative text:
The next instruction to be executed is instruction 1. 
Correct 
The JEQ jump directive is activated, so the next instruction will be the value stored in A. 

Machine generated alternative text:
Correct 
The first line sets the value of register Ato 1 

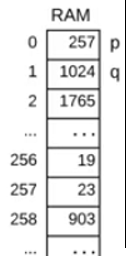
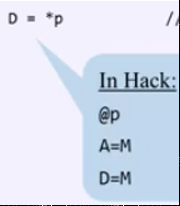
Sooo…

The Rules cont.:

1. When M is reassigned, the value in the RAM it was pointing to is also reassigned.
2. They told me at some point that M changes three things. Let me make sure that third thing is to set M itself equal to a new value, which is no longer a RAM. It loses its pointer status once you reassign a constant to it.

Ok, so שמעון's words:

 "When we say A=M, this will cause A to become the contents of the selected memory register which is register number 0. And therefore, A will now contain 257. Now remember that A is an adverse register. So once we put a certain value into A, like 257. Automatically, we're going to select register 257 from the RAM. So this will become the selected register which in the Hack Assembly Language is represented by the letter M. And therefore, when we next say, D=M, we're going to get the value of this register. The currently selected register number 257 and the value happens to be 23."



My summation:

When you do @p, this happens:

* 1. A=p
  2. M becomes a pointer to RAM[p]

When you do A=M, this happens in order:

* 1. A's value becomes the value in RAM[0], which is 257
  2. M becomes a pointer to RAM[257]

When you do D=M, this happens:

* 1. D's value becomes the value in the RAM that M points to.

OK, so whenever you have M on the right side it is treated as the value in the registor it points to

When M is on the left side, it just

My rule understanding:

M is a pointer. Unless assigned otherwise, in which case it just becomes a regular variable.

A is always changed when there is an A-Instruction (@x)

M is always affected when A is changed, because it is then set as a pointer to RAM[A]. If M changes, RAM[A] changes. As as it does that, though, M no longer points to the same place. So the next time you use M, it will have the value of whatever you assigned to it, probably a constant from D or something. I think you don't generally assign stuff to M, however, unless you need an extra variable to work with. M loses its pointer status until A is changed again.

A, on the other hand, ALWAYS affects M. M does not affect A. (Unless you are actually using M to assign something to A!)

D is never a pointer, so anytime you assign M to D, it will get a COPY OF THE VALUE IN THE REGISTER THAT M POINTS TO

**SO.**

Version 1

* 1. **A and D always contain constants.**
  2. **Whenever there is an A-Instruction (@x), A is assigned the constant value x**
  3. **Whenever A is assigned the constant value x(@x or A=x), M is assigned to point to RAM[x]**
  4. **Whenever A or D are assigned M (A=M or D=M), the computer sends a copy of the constant in the RAM to which M points. In the case of A (A=M), M is then changed to point to RAM[A].**
  5. **Whenever M is assigned x (M=x), it is really assigned a pointer to RAM[x].**
  6. **Whenever M is assigned a computation y which contains itself (M=D+M, etc.), the computer computes y with a copy of the constant in the RAM to which M currently points and then assigns M to point to RAM[y]**

 Version 2

* 1. A and D always contain constants.
  2. Whenever there is an A-Instruction (@x), A is assigned the constant value x.
  3. Whenever A is assigned x (@x or A=x), M is consequently assigned to point to RAM[x].
  4. Whenever M is assigned x (M=x), x is really being assigned to the RAM to which M points.  M itself doesn't change - it still points to the same RAM.
  5. Whenever M appears in a computation (A=M, D=M, M=M+1, etc.), the computer really assigns a copy of the constant in the RAM to which M points.  (In the case of A (A=M), M is consequently changed to point to RAM[A].)

Version 3

A and D always contain constants.

Whenever there is an A-Instruction (@x), A is consequently assigned the constant value of x. (if x is a RAM address label like SP, A is assigned the constant that is contained in that RAM.)

Whenever A is assigned a constant x (@x or A=x), M is consequently assigned to point to RAM[x].

Whenever M appears in a computation (A=M, D=M, M=M+1, etc.), the computer really assigns a copy of the constant in the RAM to which M points. In the case of A (A=M), M is consequently changed to point to RAM[A].

Whenever x is assigned to M (M=x), x is really being assigned to the RAM to which M points. M itself doesn't change - it still points to the same RAM address.

 Version 4

Ok final list

* 1. A and D always contain constants.
  2. Whenever there is an A-Instruction (@x), A is consequently assigned the constant value of x. (if x is a RAM address label like SP, A is assigned the constant that is contained in that RAM.)
  3. Whenever A is assigned a constant x (@x or A=x), M is consequently assigned to point to RAM[x].
  4. Whenever M appears in a computation (A=M, D=M, M=M+1, etc.), the computer really works with a copy of the constant in the RAM to which M points. In the case of A (A=M), M is consequently changed to point to RAM[A].
  5. Whenever x is assigned to M (M=x), x is really being assigned to the RAM to which M points. M itself doesn't change - it still points to the same RAM address.