HACK Questions

Answering Question 1:

Beginning with Basic Operations

Question 1: Simple Addition
Write a program that adds the values in `RAM[0]` and `RAM[1]` and stores the result in `RAM[3]`.
Correct!

Question 2: Check for Positivity
Create a program that checks if the value in `RAM[2]` is positive. If positive, store `1` in `RAM[4]`, otherwise store `0`.

Question 3: Increment Value
Write a program that increments the value in `RAM[5]` by 1 and stores the result back in `RAM[5]`.

Introducing Conditional Logic

Question 4: Conditional Sign Inversion
Develop a program that:
1. Checks if the value in `R AM[6]` is positive.
2. If it is, multiply it by -1 and store the result back in `RAM[6]`.
----remember 0 is not positive!

Question 5: Absolute Value Calculation
Create a program that calculates the absolute value of the number in `RAM[7]` and stores the result in `RAM[8]`.

Working with Bitwise Operations

Question 6: Bitwise AND
Write a program that performs a bitwise AND operation between the values in `RAM[9]` and `RAM[10]` and stores the result in `RAM[11]`.

Question 7: Bitwise OR
Develop a program that performs a bitwise OR operation between the values in `RAM[12]` and `RAM[13]` and stores the result in `RAM[14]`.

Combining Concepts

Question 8: Addition and Conditional Negation
Write a program that:
1. Adds `RAM[0]` to `RAM[1]` and stores the result in `RAM[2]`.
2. If the result in `RAM[2]` is positive, multiplies it by -1 and stores it back in `RAM[2]`.

Question 9: Addition and Bitwise OR
Create a program that:
1. Adds `RAM[0]` to `RAM[1]` and stores the result in `RAM[2]`.
2. Performs a bitwise OR of `RAM[2]` with `RAM[1]` and stores the result in `RAM[2]`.

Culminating in Your Exam Question

Exam-like Question: Complete Operation
Your code should:
1. Add `RAM[0]` to `RAM[1]`, then store the result in `RAM[2]`.
2. If `RAM[2]` is positive, then multiply `RAM[2]` by −1.
3. Perform a bitwise OR of `RAM[2]` with `RAM[1]` and store the result in `RAM[2]`.

Answering Question 2:

Question 1: Basic Output to Screen (Easy)
Write a Hack assembly program that turns the top-left pixel of the screen black.

Question 2: Screen Addressing (Moderate)
Write a Hack assembly program that turns the entire first row of the screen black.

Question 3: Screen Manipulation (Intermediate)
Write a Hack assembly program that creates a vertical stripe pattern on the screen, with every other column being black.

Question 4: Keyboard Input (Moderately Challenging)
Write a Hack assembly program that turns the screen black when the 'A' key is pressed and white when the 'B' key is pressed.

Question 5: Combined Keyboard and Screen (Challenging)
Write a Hack assembly program that, upon pressing the 'S' key, draws a black square starting from the top-left corner of the screen, covering a quarter of the screen.

Question 6: Timed Screen Changes (More Challenging)
Write a Hack assembly program that automatically toggles the screen color from black to white and back every second, without any keyboard input.

Question 7: Keyboard Sequences (Advanced)
Write a Hack assembly program that waits for a sequence of key presses 'X', 'Y', 'Z', in that order, and then fills the screen with a diagonal line from the top left to the bottom right.

Question 8: Checkerboard Pattern (Similar to Attached)
Write a Hack assembly program that fills the screen with a checkerboard pattern. Each square should be 8x8 pixels.

Question 9: Dynamic Checkerboard Pattern (Very Challenging)
Adapt the program from Question 8 to change the checkerboard pattern to its opposite when the 'C' key is pressed and held, as described in the provided document.

Question 10: Specific Key Sequence Detection (Similar to Q2 Attached)
Write a Hack assembly program that turns the screen black only after the specific key sequence 'O' followed by 'M' is entered, with no other keys pressed in between, as described in the provided document.

Answering Question 3:

—---

1. Arithmetic Operations (add, sub, neg)
****Easy:****
- Write HACK assembly code for adding the top two values on the stack.
- Implement subtraction (`sub`) in HACK assembly using the top two stack values.

****Intermediate:****
- Create an assembly sequence that adds two constants, say 7 and 3, and stores the result back on the stack.
- Write assembly code to subtract a constant value, e.g., 4, from a value at the top of the stack.

****Hard:****
- Develop an assembly routine that performs multiple additions and subtractions sequentially on stack values.
- Create a complex arithmetic operation combining `add`, `sub`, and `neg` on multiple stack values.

—---

### 2. Logical Commands (eq, gt, lt, and, or, not)
****Easy:****
- Implement the `not` operation in HACK assembly.
- Write assembly code for the `and` operation using two stack values.

****Intermediate:****
- Develop assembly code for the `or` operation on two stack values.
- Implement the `eq` command comparing two constants pushed onto the stack.

****Hard:****
- Write a complex assembly sequence that combines `lt`, `eq`, and `gt` to compare multiple pairs of stack values.
- Create an assembly routine that uses a combination of `and`, `or`, and `not` in a logical decision-making process.

—---

### 3. Segment Handling (push/pop on different segments)
****Easy:****
- Write assembly code for `push constant 10`.
- Implement `pop temp 2` in HACK assembly.

****Intermediate:****
- Develop assembly code for `push argument 5` assuming a known base address for the argument segment.
- Create an assembly sequence for `pop this 1` with a known base address for the `this` segment.

****Hard:****
- Write a complex assembly routine that involves pushing and popping across multiple segments like `local`, `argument`, `this`, and `that`.
- Implement an assembly code sequence that manages `push` and `pop` operations across `pointer` and `temp` segments in a complex algorithm.

—---

### 4. Function Call and Implementation
****Easy:****
- Translate a simple VM function call, e.g., `call Foo.bar 0`, into HACK assembly.

****Intermediate:****
- Implement a HACK assembly sequence for a function call with local variables, e.g., `call Compute.sum 3`.

****Hard:****
- Write a detailed assembly code for a complex function call like `call Main.compute 5`, which involves multiple operations before and after the call.
- Develop an assembly routine representing a nested function call scenario, where one function calls another, e.g., `call Main.init 2` followed by `call Main.process 4`.

—---

### Exam-Level Questions
- Implement a complete HACK assembly code that simulates a series of VM commands, including arithmetic, logical operations, and function calls. The sequence should reflect a real-world scenario, like a simple computation or decision-making process.

- Write a comprehensive assembly routine that involves handling multiple segments, arithmetic/logical operations, and function call handling, reflecting the complexity of an actual VM program.

Write HACK assembly code that directly translates the Hack VM instruction `call Main.process 3`. Assume that the current state of RAM represents the Hack VM state. Your code should modify RAM and the program counter exactly as `call Main.process 3` would. Assume the standard memory map from Hack VM to assembly, an unused label `auto$100`, and the assembly code for `Main.process` starts with `(call$Main.process)`.

—--

Answering Question 4:

### 1. Left Shift Operation
Easy:
- Write a Hack assembly program that performs a single left shift operation on the word in RAM[0] and stores the result in RAM[2].

Intermediate:
- Implement a Hack assembly program that performs the left shift operation on RAM[0] three times, storing the result in RAM[2].

Hard:
- Develop a program as described in your prompt: "RAM[2] ← RAM[0] ≪ RAM[1]". Handle cases where RAM[1] is greater than 1 but less than 16.

Exam Level:
- Enhance the left shift program to include error handling or boundary conditions, such as if RAM[1] is negative or greater than 15.

### 2. Left Rotation Operation
Easy:
- Implement a single left rotation on the word in RAM[0] and store the result in RAM[2].

Intermediate:
- Create a Hack assembly program to perform left rotation on RAM[0] a total of 2 times, storing the result in RAM[2].

Hard:
- Write a program for the specified task: "RAM[2] ← left rotation of RAM[0] for RAM[1] times".

Exam Level:
- Modify the left rotation program to optimize it for situations where RAM[1] is a large number.

### 3. Right Rotation Operation
Easy:
- Write a Hack assembly program that performs a single right rotation operation on the word in RAM[0].

Intermediate:
- Develop a program for performing right rotation on RAM[0] a total of 2 times.

Hard:
- Implement the described task: "Perform right rotation on RAM[0] for RAM[1] times and store in RAM[2]".

Exam Level:
- Create an optimised version of the right rotation program for cases where RAM[1] ≥ 16.

### 4. Right (Logical) Shift
Easy:
- Perform a single right logical shift on RAM[0] and store the result in RAM[2].

Intermediate:
- Implement a right logical shift operation on RAM[0] a total of 3 times, storing the result in RAM[2].

Hard:
- Create a program for "RAM[2] ← RAM[0] >> RAM[1]" for up to 15 shifts.

Exam Level:
- Extend the right shift program to handle cases where RAM[1] is greater than 15.

### 5. The Collatz Conjecture
Easy:
- Write a program that performs one step of the Collatz process on RAM[0] and stores the result in RAM[32].

Intermediate:
- Develop a program that executes the Collatz process for 5 steps starting with RAM[0], storing each result sequentially in RAM starting from RAM[32].

Hard:
- Implement the full Collatz conjecture program as described in your prompt, starting with RAM[0] and storing each step's result in memory starting from RAM[32].

Exam Level:
- Optimise the Collatz conjecture program for large inputs, focusing on efficient bitwise operations for division and multiplication.