Question 4 Tradeoffs among Addressing Modes

Program 1

Register contains address of the start of the array. To support the variability in element sizes in the two arrays (8 bits and 64 bits), we can use **scaled indexing mode**. This allows us to specify a base address (in the register) and an index (which can be incremented in a loop) along with a scale factor (1 for 8-bit elements and 8 for 64-bit elements, assuming byte addressing) to compute the effective address of each element in the array. An example implemention would require two loops, one for each array, 1 register that each hold the base address of the current array/loop of interest and a register to hold the index which is incremented/decremented in each iteration of the respective loops and also to set/reset before starting the second loop. The use of loops in general minimizes the number of instructions while scaled indexing mode allows us to handle the different element sizes.

Note: One might use autoincrement for the first loop but clarification with professor indicated that answers should focus only on 1 addressing mode per program.

Program 2

Register contains address of the start of singular integer array. Assuming we know how many bits/bytes each integer in the array occupies (e.g., 32 bits/4 bytes), we can use the **scaled indexing mode**, where the base address is first stored in a register and an index value stored in another (which can be incremented in a loop) along with a scale factor equal to the size of each integer (4 for 32-bit integers, assuming byte addressing) is used to compute the effective address of each integer in the array.

Program 3

A register contains the value of the variable `p`. `p` here is a pointer whose value is an address of an allocated memory location. We can use **register indirect addressing mode** to access and update the value pointed to by `p` (use value in register which is an address as the effective address). We would be able to store a value to the memory location pointed to by `p` using just a single instruction like `STR GPR[1], [GPR[0]]`, assuming `GPR[0]` holds the value of `p` and `GPR[1]` holds the new value to be stored.

Program 4

A register contains the value of the variable `p`. `p` here is a pointer to a pointer to a pointer (i.e., triple indirection). For this, we would use **register indirect addressing mode** in multiple steps to get the desired value. Note that we begin with `GPR[0]` containing the value of `p`, which is the address of the second pointer.

```
// GPR[0] contains value of first pointer i.e. address of second pointer
LDR GPR[0] [GPR[0]]; // GPR[0] now contains value of second pointer i.e.
address of third pointer
LDR GPR[0] [GPR[0]]; // GPR[0] now contains value of third pointer i.e.
address of actual variable
STR GPR[1] [GPR[0]]; // Store new value from GPR[1] into the actual
variable
```

Why not use memory indirect addressing mode?
Consider `STR GPR1 [[GPR[0]]]` as suggested syntax in lecture notes. Doing so is memory indirect addressing since we are using `M[R0]` as the address to be accessed. But this operation would overwrite the third pointer rather than the actual variable. As clarified in class with the professor, adding an additional `[]` while would achieve the desired effect, is not valid syntax.

Ideally, the best way to reduce instruction count is to use both addressing modes.

```
LDR GPR[0] [[GPR[0]]] // Use M[GPR[0]] aka value in second pointer as
address, dereferencing yields third pointer.
STR GPR[1] [GPR[0]]   // Store new value from GPR[1] into the address
specified by GPR[0] (third pointer)
```