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, 2 registers that each hold the base address of the respective arrays and a register to hold the index which is incremented in each iteration of the loop and also 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.

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 **based addressing mode**, where the base address is first stored in a register and a fixed offset which is the number of byte per integer (assuming byte addressing) is added to the base address to compute the effective address of each integer. Note that we would use **post-indexed addressing** in particular, using the address in register first and then updating it with the newly computed effective address after each access (base + offset). An example implementation would require a loop, a register to hold the base address of the array and an offset value (4 bytes for 32-bit integers) that is added to the base address in each iteration of the loop. Similarly, the use of loops minimizes the number of instructions while based addressing mode allows us to handle the automatic increment of the address after each access.

Program 3

A register contains the value of the variable `p`. `p` here is a pointer with memory already allocated to it, in other words, the value of `p` is an address. 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 **memory indirect addressing mode**. `GPR[0]` yields value stored in `p` aka the first pointer aka the address of the second pointer. Performing `MEM[GPR[0]]` yields the second pointer aka address of the third pointer. Performing `MEM[MEM[GPR[0]]]` yields the third pointer aka address of the actual variable. Finally, performing `MEM[MEM[MEM[GPR[0]]]]` would yield the value of the actual variable. To update the value of the actual variable, we would perform `MEM[MEM[MEM[GPR[0]]]] = new_value`. If nested memory indirect addressing is not supported, we can simply overwrite the value in the register with the intermediate pointer values in sequence. For example, first perform `LDR GPR[0] [GPR[0]]` to get the second pointer, then `LDR GPR[0] [GPR[0]]` again to get the third pointer, and finally `STR GPR[1] [GPR[0]]` to update the actual variable.