HW7 – Fun with Self-Modifying Code in PAL

The goal of this assignment is to practice more PAL by solving a tricky problem that will required in-depth understanding of many PAL instructions so you can use them in unique ways. Complete this assignment with your current partner.

Deliverables: Your solution program in a file named user1_user2.pal.

Your Task – Clear All Memory

Write a PAL program no longer than 100 lines that will erase (by erase I mean ``set to zero'') as many memory locations as possible, including the memory where the program is written. The number of lines is the number output in terminal before "end of file" when you assemble your program. You will use loops (and/or the entire memory space as one big loop given PAL's automatic continuing with instruction 0 after 7777) and so the number of lines executed will be much (much) higher than 100, the limit is on lines of actual PAL code in your .pal file. FYI, you shouldn't need anywhere close to 100 lines, so there is still plenty of room for whitespace and the comments you are required to include in your code.

Your grade will be determined by how many memory slots are not erased after your program either halts or has run for a reasonable amount of time, we'll say 10 seconds running in regular (not trace) mode on a lab computer using the PDP8 simulator. Your program does NOT need to stop itself with a HLT, and indeed the best solutions could not do so, you can instead use the timeout after 10 seconds as your HLT.

Scores:

- It is a reasonable task given the PAL we've learned in the class/reading so far to clear all but a few memory slots; if your program leaves 4 or fewer slots non-zero you will receive 8/10 points.
- It is not too much more difficult using auto-indexing (see below) and understanding of all PAL instructions to get down to all but 2 memory slots cleared; if your program can do this you will receive 9/10 points.
- It is tricky but doable to clear all but 1 memory slot; for this you will receive all 10/10 points.
- It is possible to clear all memory slots completely; I can show you some examples of solutions that do so if you are interested, though I will not post them anywhere for the sake of future 208 classes. If you come up with a solution that does this you will receive my being very impressed and 12/10 points.
- If your program leaves more than 4 slots uncleared, you will receive a variable amount of points based on your solution quality.
- **You still must comment your program** and use good style to receive the maximum possible points described above! Your program is only correct if it works AND you can explain how it works!

Hints:

- *** The most common mistake in the past *** was to see that all memory had value 0 after your program ran and thus conclude that you were done, when in fact one or more memory locations had not been modified at all by your program. A memory address is not "cleared" just because it happens to have the value 0; you cannot assume any default initial values in memory. You must actively somehow set the value at an address to 0 to "clear" it . The most obvious way to do this is to execute a DCA instruction (after ensuring the acc value is currently 0) to the memory address you wish to clear. There are multiple other ways to end up setting a value in memory to 0, and it's worth thinking about those.
- A corollary to the hint above is that executing uninitialized memory addresses as instructions means carrying out completely random instructions and accepting whatever consequences they have. If you haven't set the value at an address to 0 (or something else) explicitly, and the PC reaches that address, anything that can happen (in PAL), will happen (with some probability). I will test your program by first loading it, then filling all unused memory locations with random values.
- Use the red hexagon stop button to stop a program that is running in an infinite loop. Once you stop it the CPU window will re-expand so you can see the result.
    - *** Even more useful tip from another student: the window never minimizes when running in trace mode. Execution speed is much much slower, but that can be useful when debugging, and you can click on the red stop sign button anytime to pause at a specific place relatively accurately (within a few instructions).
- Double click on the BP column to the left of any instruction to set a breakpoint there. You can then run in either regular or trace mode and your program will pause at the breakpoint until you click the run button again, allowing you to see intermediate results.
- A PAL program that is not stopped simply keeps executing with the PC wrapping around from 7777 to 0. Remember that every sequence of 12 bits is some instruction so executing any memory slot does “something”, even if the result doesn’t actually change anything in the end. For example executing a cleared memory slot (value is just 0) runs the “AND 0” instruction which sets AC = AC AND value-at-mem-address-0. If the AC is already 0 or the value at address 0 is 0, this instruction has no effect.
- Your program will complete thousands of loops per second even if each loop executes all 10000 (octal) instructions, so no matter how complicated your solution, it should not need more than 10 seconds to reach a point where no more memory will be cleared (in standard run mode, trace is much slower).
- Make sure that if your program does keep running in an infinite loop, whatever non-zero instructions are left don’t re-set any memory values to be non-zero.
- The PC always starts with the value 200, so if you want your program to begin somewhere else, put a “JMP X” instruction at memory address 200, where X is the address of the first instruction in your actual program. This is only important when you first start your program, so if you later clear the memory and lose the JMP X instruction that’s just fine!

Auto-Indexing

- PAL treats memory slots 10-17 (octal) in a special manner. When you access these memory addresses using an indirect instruction, the value at that slot is incremented by 1 before taking the 2nd jump to the given address to get the actual value for the instruction. This is explained in the PDP8 pdf in chapter 3 (3-27 to 3-28). Here is an example:

Program using indirect access to normal memory slots (i.e. all except for 10-17):

```
*100
1
2
3
4

*176
100

*200
TAD I 176
TAD I 176
TAD I 176
HLT
```

After this program runs the AC would contain the value 3, because each of the TAD's does the same thing - gets the value 100 from mem address 176, then goes and gets the actual value to add from mem address 100, which is a 1.

However, if we change the program to use an indirect TAD to one of the special auto-index memory slots, the behavior is very different. The modified program is shown below:

```
*10
100

*100
1
2
3
4

*200
TAD I 10
TAD I 10
TAD I 10
HLT
```

After this program stops the AC would contain the value 9 (2+3+4) because each of the indirect TAD's does more than just the regular indirect load. The first "TAD I 10" first increments the value in memory address 10, making it 101, and then it goes and gets the actual value to load from this mem address 101, which is a 2. The second "TAD I 10" does the same thing, except the initial value in memory

address 10 is now 101, so it adds 1 to that making it 102, then goes and gets the value from memory address 102, which is a 3. Again the last TAD makes the value at address 10 a 103 and adds the value 4 from address 103 to the AC. You can see how if you made the above a loop that ran TAD I 10 over and over again, the value in 10 would be incremented each time and thus the actual value loaded would come from the next memory address. This works in the same way for any of the 6 memory access instructions. Using the auto-index memory in this way is different from using the ISZ instruction as it is doesn't perform the skip on zero part; instead the value at memory address 10 just keeps getting incremented, wrapping around to 0 after it reaches 7777. Like all memory addresses it is treated as an unsigned integer.

Think First…

- Spend some time just thinking about this problem and possible solutions. Start by writing a program that clears the first page of memory, or addresses 0-177. A simple algorithm for this would be to loop 200 (octal) times, each time using DCA (make sure the AC is 0 first) to clear the next memory slot (starting with the first slot 0). You could do this by using DCA I with an auto-indexed memory address, or by using DCA I to a normal memory slot and then ISZ to increment that normal slot each time in the loop.
- Expand your program to clearing "most" of the memory, even if not all but 4 or fewer slots. This just requires looping more times, and being careful that if you start to clear instructions from your program itself you don't cause cleared memory slots to be re-written with non-zero values.
- Now consider ways to reduce the number of instructions in your program, thus letting you clear more memory slots without clearing the important part of your program itself.
    - You might be able to get rid of your own looping entirely by using the fact that PAL automatically loops back around to execute the instruction at memory address 0 after it executes address 7777.
    - You might be able to use auto-indexing to increment values instead of ISZ and this may be simpler and/or use fewer instructions.
- Finally order your instructions and strategically place them in memory so that your actual program instructions are cleared last, and as many as possible of them are cleared before the program is so broken that it can't do anything anymore.

Please address any complaints to Jeff Ondich who originally came up with this problem, and/or Andy Exley who inspired me to use it.