Lec-4-HW-1-selfModify-b

An extension of our prior assignment, Lec-3-HW-2-selfModify-a. We will use more of the LC3's instruction set. Note, when referring to numbers expressed in hex, we will use, e.g., "x1234" or "0x1234" or "h1234" interchangeably. When it is obviously hex notation, we may use "1234". Also, LC3 assembly language uses "#" for decimal notation, so "#10" is ten, or binary "1010", or "b1010".

Self-modifying code puzzler (recap)

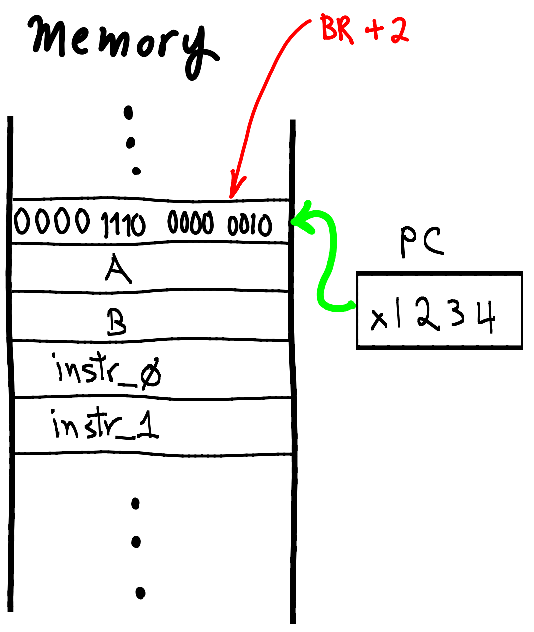
Write a program, **P**, in LC3 ISA machine code which alters its first two instructions by adding integers A and B to those instruction's opcodes:

instr_0.opcode <== instr_0.opcode + A

instr_1.opcode <== instr_1.opcode + B

"instr_0", is the first instruction after the data items A and B, as shown at right. "instr_1" immediately follows it.

The code to do this should work correctly regardless of where the program is located in memory when executing. Assume the PC initially contains the address of P's first instruction, which is just above A. In the example at right, it is in the memory cell whose address is x1234. That happens to be a BRnzp instruction, an unconditional branch.

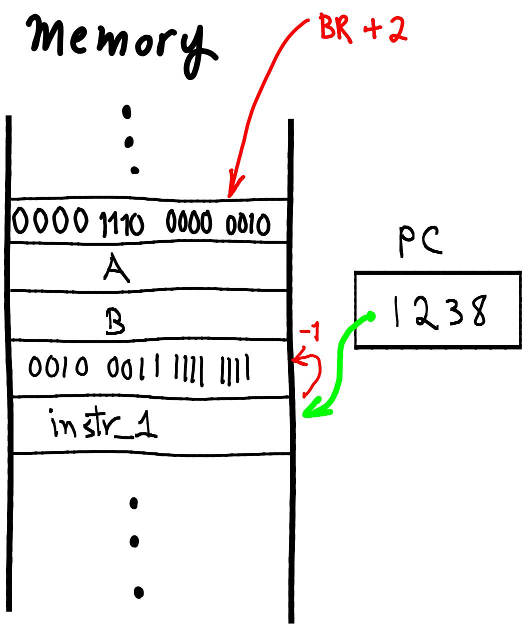


New Stuff

We saw that the first job needed was to get the address of instr_0, somehow. We can put instructions in that refer to that instruction by using PC-relative addressing, and calculating the offset needed. For instance, we saw that the following would get instr_0 into a register:

```
LD R1, #-1      (or,   0010 001 1 1111 1111,  or  23FF )
```

In this case, when instr_0 is executed, the PC would have (1237+1). The offset is -1; so, the LD is from address 1237, which contains instr_0.



Note on PennSim.jar

Bits in memory are translated as if they were meant to be instructions. However, the offset is not shown, rather the memory address calculated from the PC+offset is shown. In our present case PennSim.jar displays,

```
LD R1, x1237
```

for instr_0.

We will want to refer to instr_1 as well. We could use the same trick: simply make instr_1 be,

```
LD R2, #-1      ( or  0010 010 1 1111 1111,  or x25FF )
```

However, later on we will want to store our altered instructions back in their original locations. We could calculate offsets at that point, but there is an easier way. Let's get the address of instr_0 into a register first, then later we can always use that register as a base in forming our addresses. There is an instruction that can do that,

```
LEA R3, #-1    ( or 1110 011 1 1111 1111,  or E7FF )
```

which does the same address calculation that LD does, but stores the address into the register instead of accessing memory.

(Problem 1) Alter your solution to the last homework (or create from scratch) to use LEA as instr_0, and then always use that register, R3, e.g., for any loads or stores using base-offset addressing mode (LDR, STR).

(Problem 2)

The constants, A and B, should be in the range 1 to 7, i.e., in the set

{ 0000 0000 0000 0001, ..., 0000 0000 0000 0111 }

We noticed last HW that these need to be shifted left to line up with the opcodes of instr_0 and instr_0, respectively. We also discovered that that can be done by adding. Suppose A has been loaded to R4. If we do,

```
ADD R4, R4, R4      (0001 100 100 000 100,    or   x1904)
```

R4 will contain A left-shifted one bit. So, we could get the shift we want by writing 12 ADDs in our program. However, that's boring. It would be much more fun to write a loop instead.

In fact, we have just the instruction we need for that, e.g.,

```
BRnzp #-10    (  or   0000 111 1 1111 0110,  or  0FF6 )
```

will jump backward 10 addresses in memory. And,

```
BRz #-10       (  or  000 100 1 1111 0110,  or 09F6 )
```

will jump back 10, if the prior instruction that loaded a register wrote a zero. If we loaded, e.g., R5, with 0000 0000 0000 1100 (or x000C, which is #12), we could use R5 as a counter, and test it at the end of the loop to see when the counter reached 0:

```
ADD R5, R5, #-1     ( or 0001 101 101 1 1 1111,  or 1B7F )
BRp  #-10             ( or 0000 001 1 1111 0110,  or 07F6 )
```

which jumps back until zero is stored in R5. Assuming the loop's instructions started 10 instructions back in memory, this is what we would want.

Rewrite your solution to use loops to do all bit shifting.