Lec-4-HW-1-selfModify

Self-modifying code puzzler

Write an LC4 program which alters its own first instruction by changing only the opcode:

instr_0.opcode <== instr_0.opcode + A

The notation, "instr_0.opcode", means the opcode field of the instruction, "instr_0". An instruction's opcode field is the four left-most, high-order bits of the 16-bit instruction. Write your program in machine code (0s and 1s) and assembly code, e.g., "ADD".

An initial state is shown at right. The PC is pointing to the location containing instr_0. instr_0 is about to be fetched and executed. In this example, data A is in location x1234 - 2 = x1232. However, write your code so that it will work no matter where in memory the code and data are. For instance, it should also work if A is in xFFF0 and instr_0 is in xFFF2. (In that case, the PC = xFFF2.)

Data A and instr_0 are both 16-bit words. Because the opcode is only 4-bits, the value of A has to be a small. Let us assume A is positive; so, 0 < A < 16. That is, only the low-order four bits of A can be non-zero. A will have to be altered so that the addition only affects the opcode bits of instr_0.

---------- LC4 Instructions and Semantics ----------

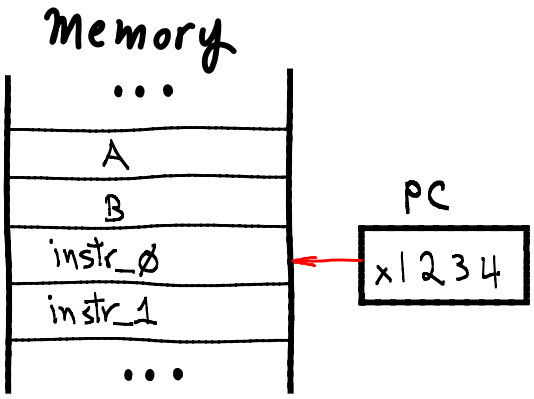
```
ALU SR3 SR4 DR4 ADD   //--- R3 + R4 ===> R4
LIM DR4  x-4          //--- R4 <=== SEXT9( IR[8:0] )
LDR DR2 AR4           //--- R2 <=== DMEM[ R4 ]
STR SR2 AR4           //--- R2 ===> DMEM[ R4 ]
LEA DR3               //--- R3 <=== PC
BRR CR6 AR4           //--- if R6 < 0, PC <=== R4
```

SEXT9() converts a 9-bit integer to a 16-bit integer with the same value. E.g., here is +3 extended:

000000011 ===> SEXT9 ===> 0000000000000011

Negative values have their bits flipped, and 1 added. E.g., here is -3 sign extended:

111111101 ===> SEXT9 ===> 1111111111111101



ALU operations are ADD (000), AND (001), NOT (010), INC (011), SUB (100), iOR (101), NOR (110), DEC (111). INC adds 1; DEC subtracts 1; iOR is inclusive OR. For NOT, INC, and DEC, both source registers are the same register.

HINT: You can shift bits to the left by adding a number to itself. E.g.,
001101 + 01101 = 011010

----------Instruction Formats ----------
Examples are shown ("x" means, "don't care whether bit is 0 or 1").

```
ALU  SR2  SR3  DR0  ADD
0000 010  011  000  000

ALU  SR2  SR2  DR0  INC
0000 010  010  000  011

LIM  DR2   x3
0001 010  00000011

LDR  DR2  AR6
0010 010  110  xxxxxx

STR  SR2  AR6
0100 010  110  xxxxxx

LEA  DR2
1000 010  xxxxxxxxx

BRR  SR2  AR3
1111 010  011  xxxxxx
```

LC4 Assembly Language Conventions

S means "Source": "SR2" reads "Source Register 2"
D means "Destination": "DR3" reads "Destination Register 3"
A means "Address": "AR4" reads "Address Register 4"
C means "Condition": "CR5" reads "Condition Register 5"

"ALU SR2 SR3 DR4 ADD" can be read as,
"ALU Source data from R2 and Source data from R3 goes to Destination R4 after ADDing"

"LDR DR6 AR2" can be read as,
"LoaD-via-Register to Destination R6 using Address in R2"

Advantages of this style are (1) the usage of the instruction's bit fields are displayed in the assembly language for that instruction, (2) the assembly language bit fields correspond left-to-right to the instruction's bits from high-order to low-order, and (3) this assembly language looks different from LC3's assembly language, making it easier to tell which machine the instruction are meant for.

More Hints

1. It is easier to work backwords and forwards on this at the same time. The last thing your program has to do is write your modified instr_0 back to its original location in memory. Let's suppose the modified bits of instr_0 are in register R3. Suppose further that you have the address of the memory location where instr_0 was originally in register R1. (We'll work out how that all came to be true later.) For now, given those two assumptions, what is the last instruction of your program?

2. Given (1), it must be that we loaded R1 before the last instruction. Why not take care of that first thing? What we need is at least some starting address in R1. How can you get an address into R1? The only thing around that already has an address in it is the PC. So, we should get the PC's value into R1. What instruction loads the PC to a register? Note that the first instruction of our program is instr_0; so, we are figuring out what instr_0 should be.

3. If instr_0 loaded the PC's value to R1, to which memory location does R1 refer to? Recall that the PC is incremented before being loaded to R1; so, we need to adjust R1 to point to instr_0. How would we do that? This is the instruction after instr_0.

4. Now that our program has the right address in R1, we need get instr_0 into a register so we can manipulate its bits. Suppose we put it into R2. What instruction would accomplish that?

5. With instr_0's bits in R2, we need to get A into R3. Change R1 to point to data A's location, and read A into R3.

6. The next part is to get the low-order bits of R3 shifted to be the high-order bits. Use the hint about using ADD to shift bits. Then, add the content of R2 and R3 to change the opcode.

7. Readjust R1 to instr_0's location and write the result.