CMPT 295

Name: Junchen Li

Student ID: 301385486

Date: 2020/3/18

Assignment 7

Objectives:

* Another look at recursion in x86-64 assembly code
* Designing and evaluating instruction sets (ISA)

Submission:

* Submit your document called **Assignment\_7.pdf**, which must include your answers to all of the questions in Assignment 7.
  + Add your full name and student number at the top of the first page of your document **Assignment\_7.pdf**.
* When creating your assignment, first include the question itself and its number then include your answer, keeping the questions in its original numerical order.
* **If you hand-write your answers (as opposed to using a computer application to write them):** When putting your assignment together, do not take photos (no .jpg) of your assignment sheets! Scan them instead! Better quality -> easier to read -> easier to mark!
* Submit your assignment **Assignment\_7.pdf** electronically on CourSys.

Due:

* Thursday, March 19 at 3pm.
* Late assignments will receive a grade of 0, but they will be marked in order to provide the student with feedback.

Marking scheme:

This assignment will be marked as follows:

* + All questions will be marked for correctness.

The amount of marks for each question is indicated as part of the question.

A solution will be posted after the due date.

1. [7 marks] Another look at recursion in x86-64 assembly code

Consider the following recursive implementation of the function factorial fact written in x86-64 implementation:

1 # fact(n): Buggy version

2 .globl fact

3 # n in edi

4 fact:

5 cmpl $1, %edi

6 jg endif

7 movq $1, %rax

8 ret

9 endif:

10 decl %edi

11 pushq %rdi

12 call fact

13 imulq (%rsp), %rax

14 leaq 8(%rsp), %rsp

15 popq %rdi

16 ret

Refresher:

Mathematically, the formula for a factorial is as follows.

If *n* is an integer greater than or equal to 1, then

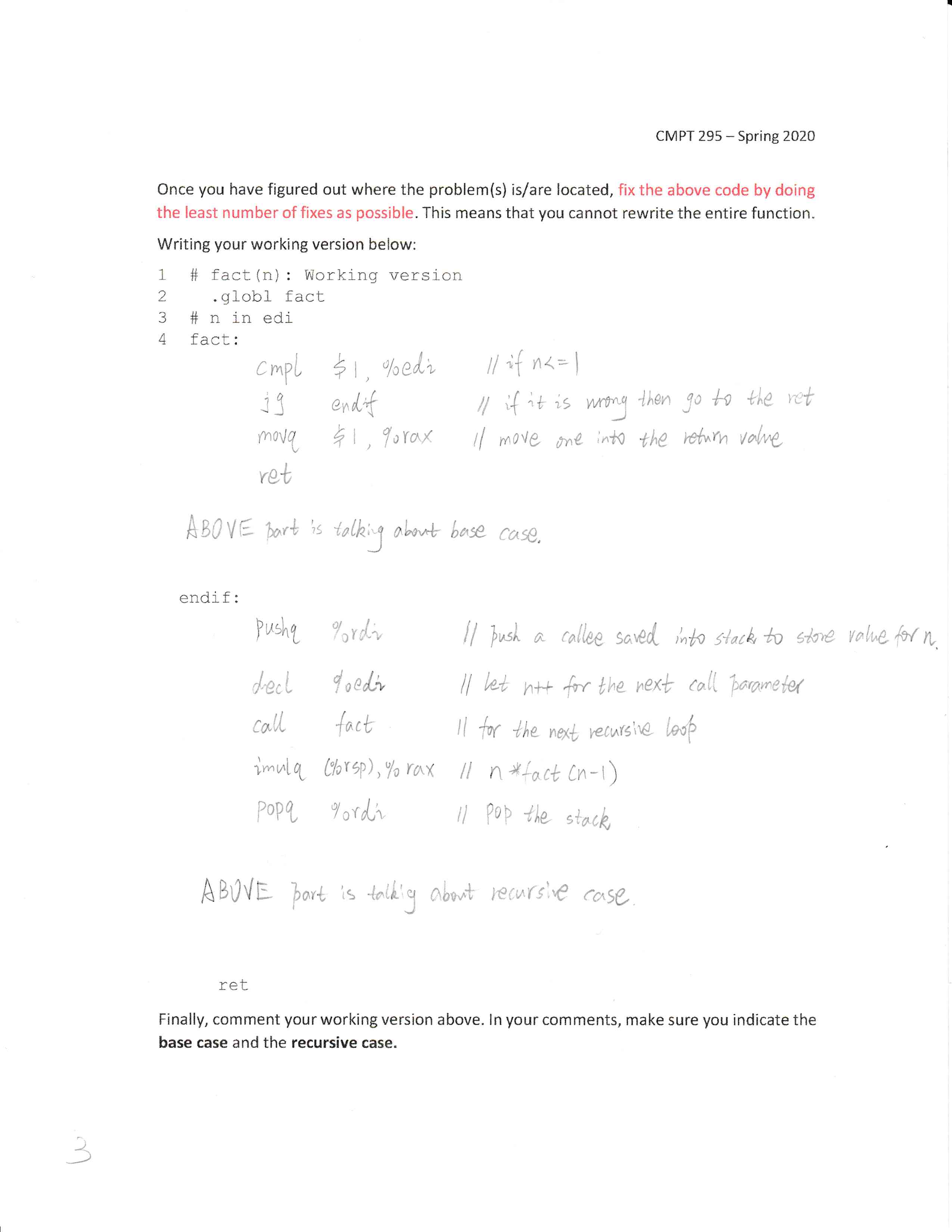
*n* ! = *n* ( *n* - 1)( *n* - 2)( *n* - 3) ... (3)(2)(1)

If *n* = 0, then *n* ! = 1.

Download the files main.c, fact.s and makefile. When we compile and execute the above function fact, we get a segmentation fault error. Verify that this is indeed the case.

Your task is to debug the function so that it produces the expected results.

You are to do so by hand tracing it with the test case n= 4. As you are hand tracing the function, you may want to draw its stack diagram and its register table as this may help you visualize what is happening with the code. You do not have to hand in your stack diagram/register table.



[13 marks] Designing and evaluating instruction sets (ISA)

Note: What you need to do in this question is highlighted in this blue colour.

Instruction Set 1 – x295

**Description of ISA**

During our lectures 22, 23 and 24, we specified an instruction set architecture (ISA) called **x295**, with the following components:

* Memory model of the computer
  + Size of external memory (RAM): 212 × 16
  + Memory address: 12 bits
  + Word size: 16 bits
  + Number of registers: 0
* Instruction set
  + Maximum number of instructions: 16
  + Opcode size: 4 bits (24 = 16)
  + Operand Model:
    - Memory (only) – only memory locations are operands, holding values (no registers are used as operands)
    - 3 operands
    - Operand order: Dest, Src1, Src2
  + Memory addressing mode: Direct
  + Instructions (so far):
    - ADD a,b,c Meaning: M[c] <- M[a] + M[b]
    - SUB a,b,c Meaning: M[c] <- M[a] - M[b]
    - MUL a,b,c Meaning: M[c] <- M[a] \* M[b]
  + Template:

opcode

Src2 (12 bits)

XXXX

Src1 (12 bits)

Dest (12 bits)

XXXX

This template can be used to form all three instructions.

* Data size : 16 bit

**Evauation of ISA**

We wrote the following C program using the instructions in this **x295** instruction set:

|  |  |
| --- | --- |
| **C program** | **x295 program** |
| z = (x + y) \* (x – y) | ADD x, y, tmp1  SUB x, y, tmp2  MUL tmp1, tmp2, z  (where tmp1 and tmp2 are memory addresses holding temporary results) |

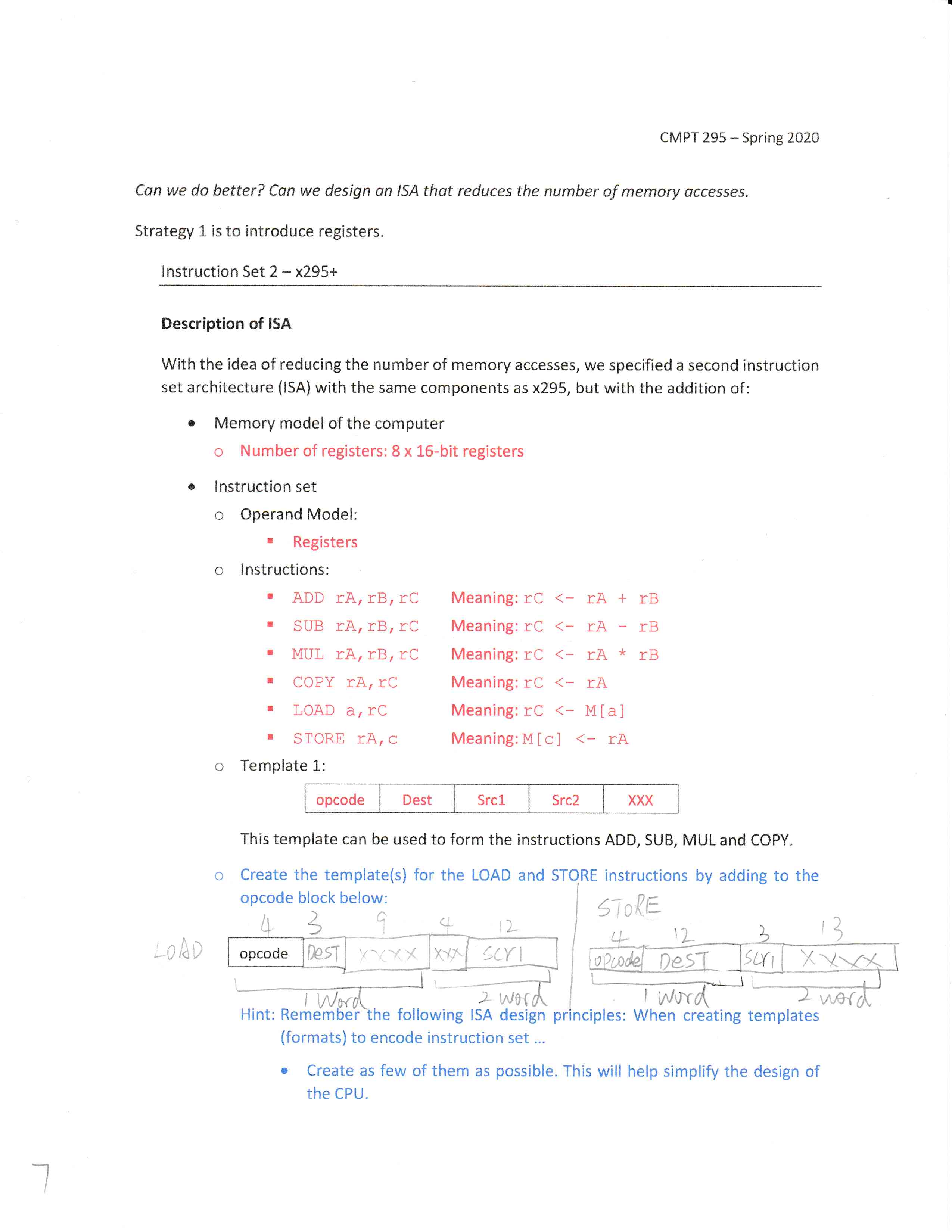
1. In order to evaluate our instruction set, we used the metric called memory traffic, i.e., we counted the number of memory accesses our program (written in x295) made during its execution. In other words, we counted how many time the execution of our program required a word (16 bits) to be read from or written to memory.

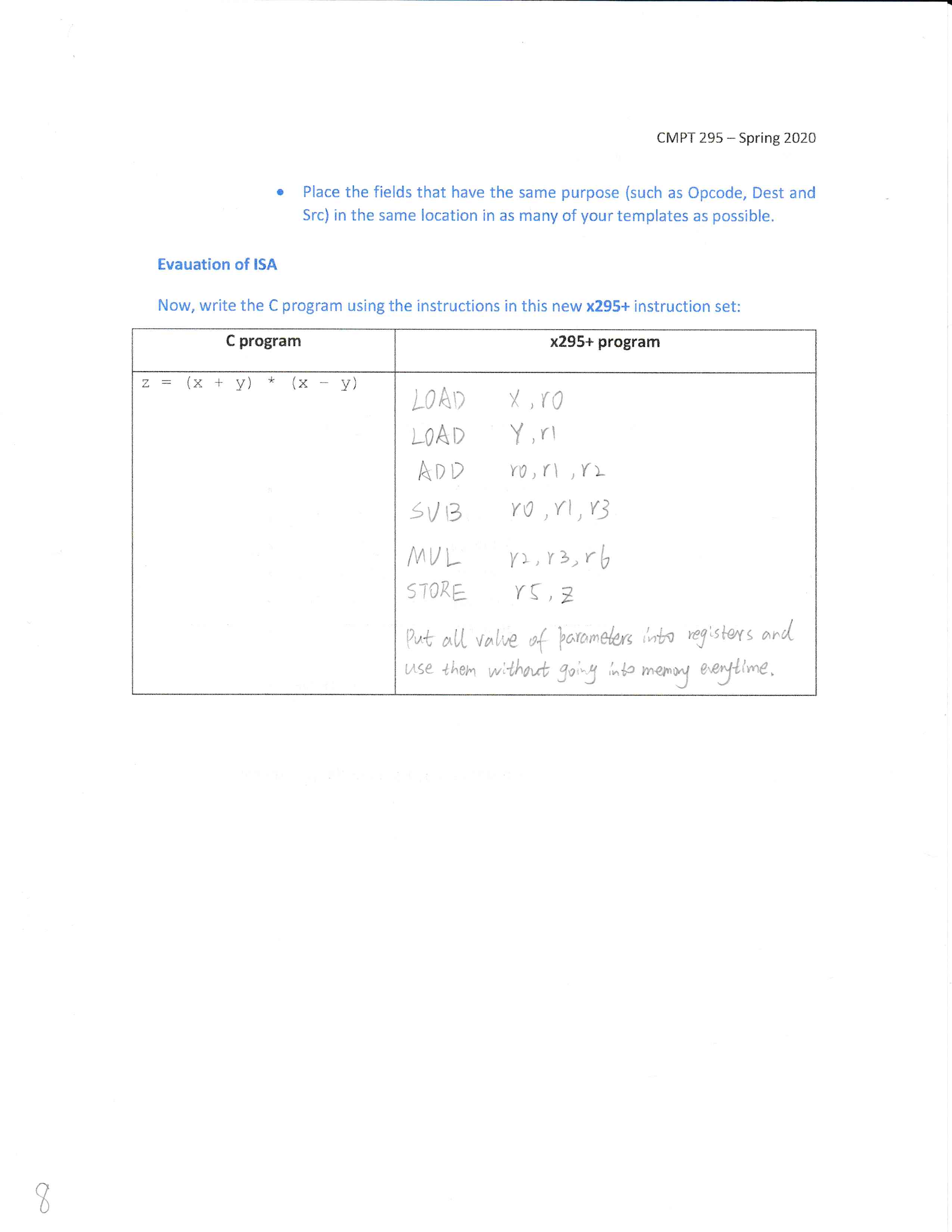
Note that we first did ascertain the fact that we were able to completely write the above C program using the instructions found in our **x295** instruction set and hand tracing it allowed to ascertain that it would produce the expected result (e.g., test case: x = 3, y = 2, expected result: 5).

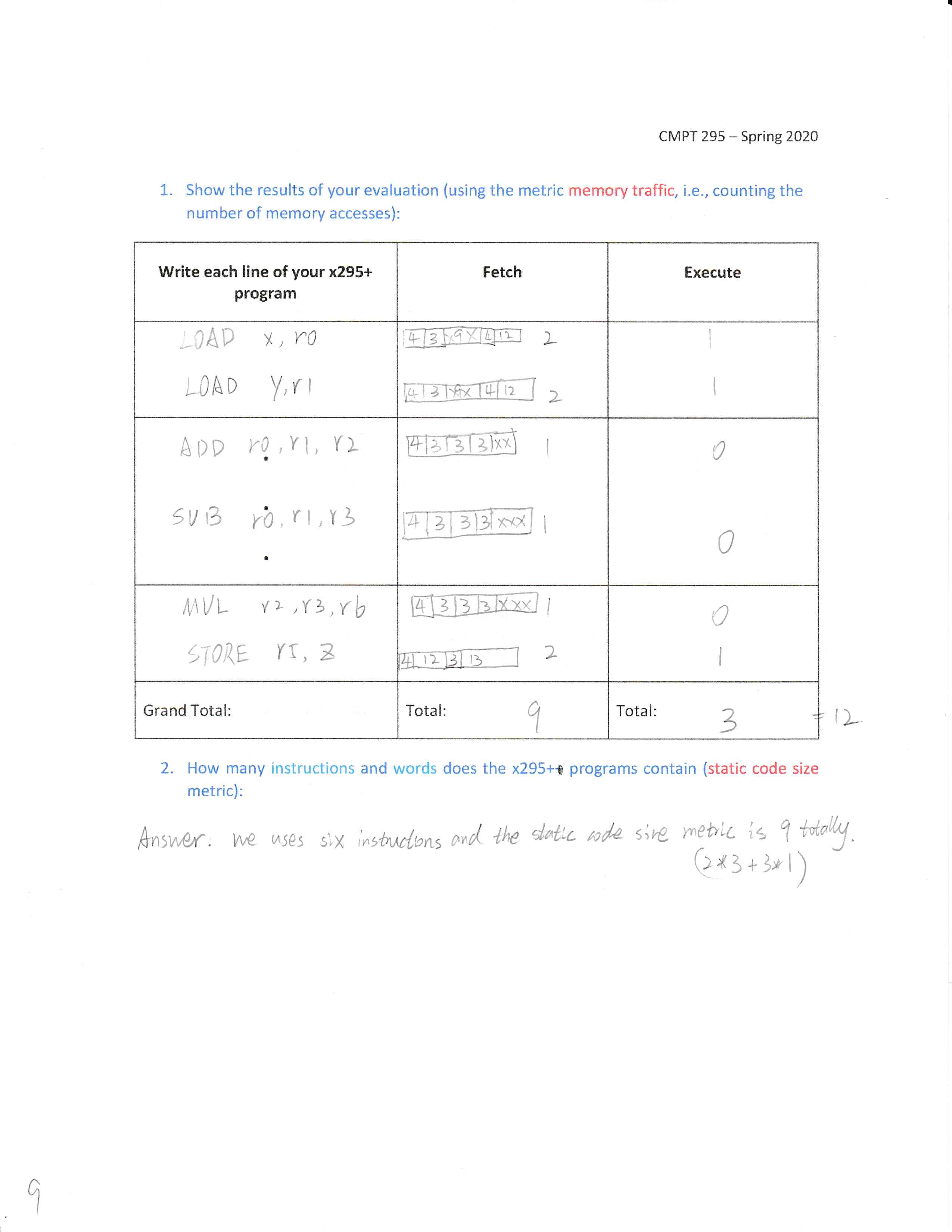
The table below show the results of our evaluation:

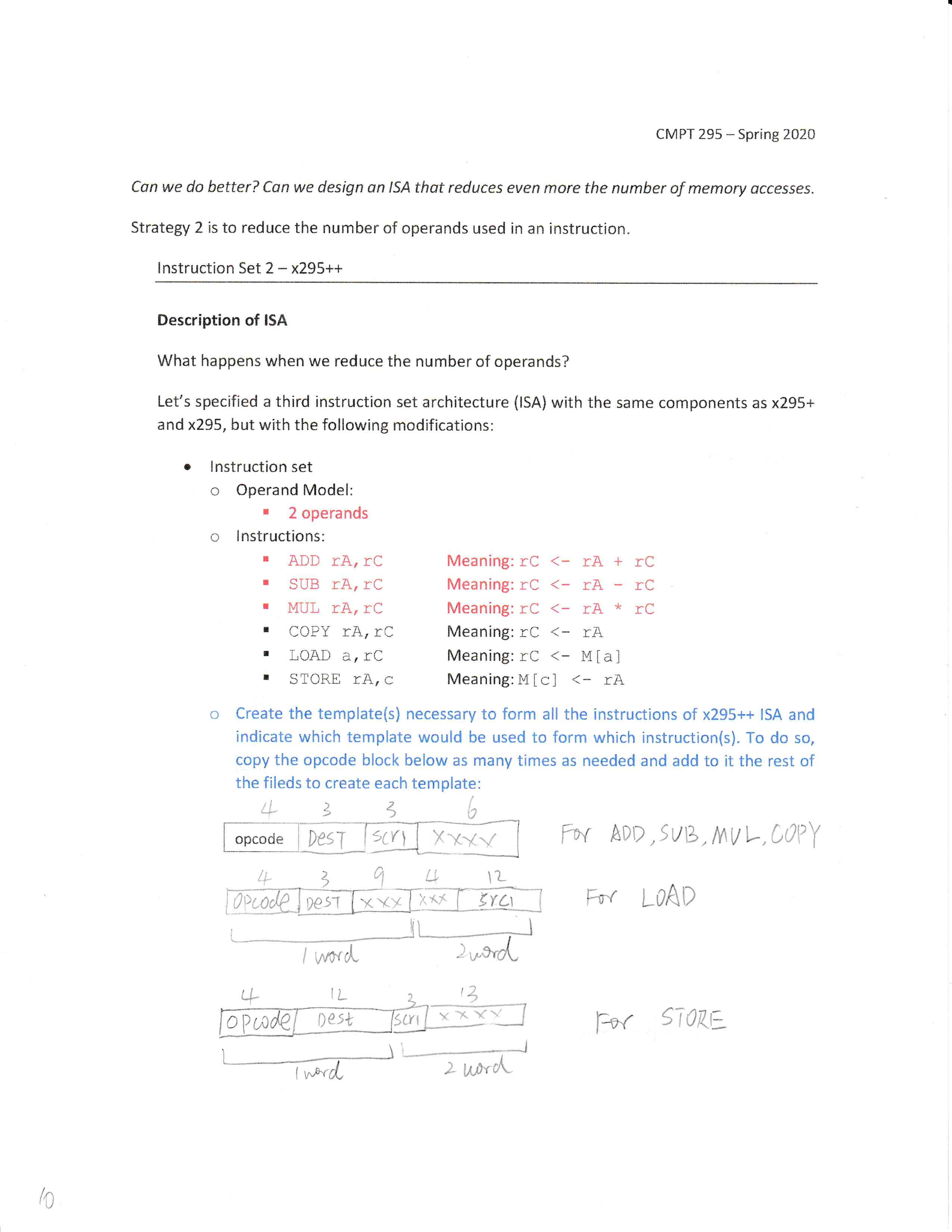
|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions of x295 program** | **Fetch** | **Execute** | |
| ADD x, y, tmp1 | 3  since the binary encoding of the ADD instruction is 3-word wide | 2  since the ADD instruction requires the value of two operands read from memory | 1  since the ADD instruction writes its result to memory |
| SUB x, y, tmp2 | 3  since the binary encoding of the SUB instruction is 3-word wide | 2  since the SUB instruction requires the value of two operands read from memory | 1  since the SUB instruction writes its result to memory |
| MUL tmp1, tmp2, z | 3  since the binary encoding of the MUL instruction is 3-word wide | 2  since the MUL instruction requires the value of two operands read from memory | 1  since the MUL instruction writes its result to memory |
| Grand Total: 18 | Total: 9 | Total: 6 + 3 | |

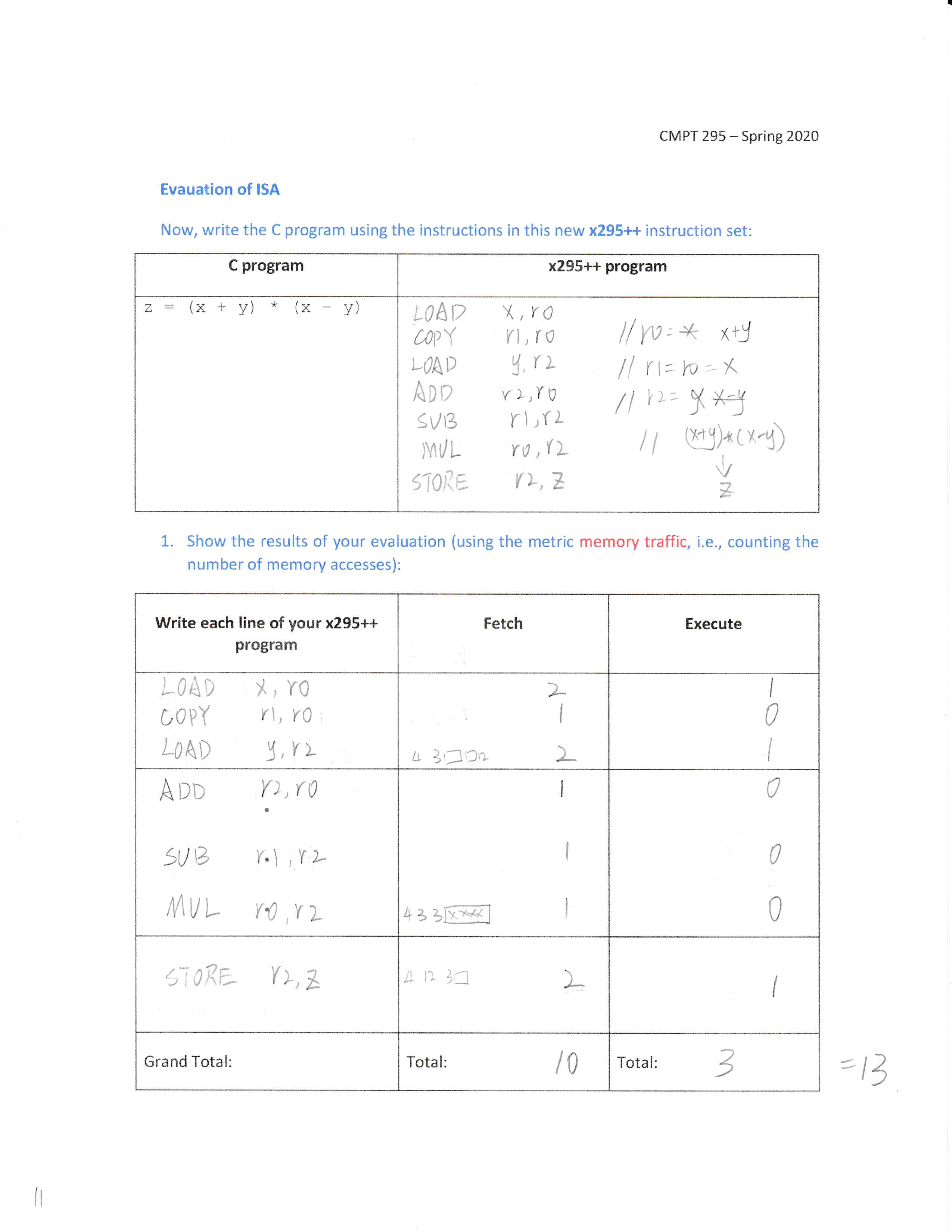
1. We also evaluated our instruction set using the metric called static code size:
   * The code size of our x295 program is 3 instructions
   * Since each instruction is 3-word long, the code size of our x295 program is also 9 words.

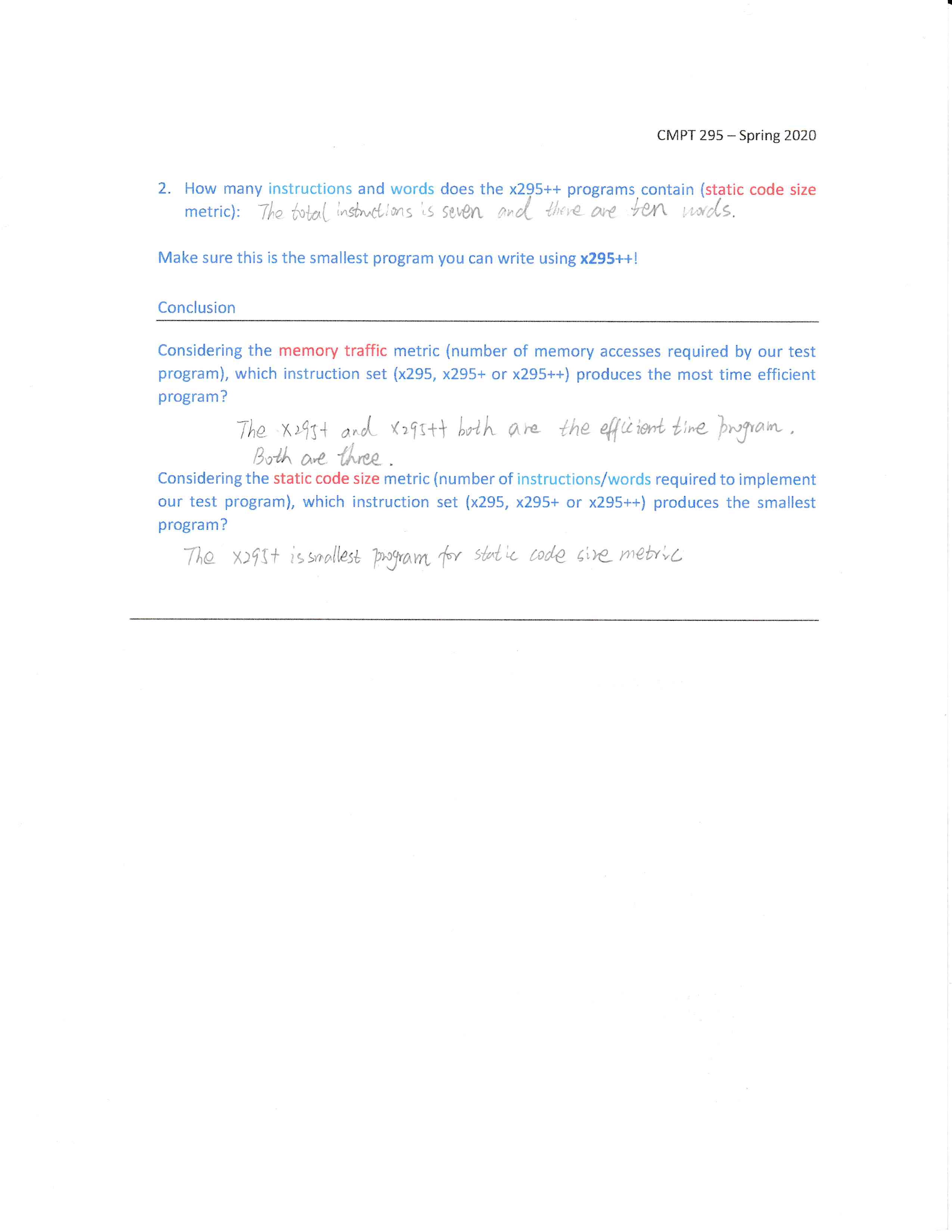




**

**

**

**