# ICS HW7

## T1

We use compilers to translate high-level languages (such as C/C++) into low-level languages (such as assembly language). Consider the following C code snippet:

```c
int x;
scanf("%d", &x);
int y = 2;
int z = y * 3 + 1;
for(int i = 0; i < z; i++) {
    y = i + 3;
    if(i < 0)
        x--;
    else
        x++;
    printf("%d\n", x);
}
printf("%d %d %d", x, y, z);
```

1. Compile the above code into LC-3 assembly code. For convenience, you may use INPUT [reg] to read an integer from the console into register [reg], and OUTPUT [reg] to print the integer value in [reg] to the console.

2. Are all statements in the resulting LC-3 assembly code necessary? Optimize this LC-3 assembly code to improve its efficiency as much as possible without changing its semantics.

3. what techniques may a compiler use to optimize its result?

## T2

In traditional arithmetic expressions (i.e. infix expressions), we often use parentheses to ensure the correct order of operations, which is not ideal for computer parsing. Using postfix expressions can effectively avoid the use of parentheses, thereby increasing convenience.

The rules for postfix expressions are: read the characters from left to right. If a number is read, push it onto the stack. If an operator is read, pop the topmost number(s) from the stack (equal to the number of operands the operator requires), perform the operation, and push the result back onto the stack. Repeat this process until the entire expression has been read. At this point, the single number remaining in the stack is the result of the expression.

For example, consider the postfix expression:

$$3\ 1\ -\ 4\ *$$

The steps to evaluate this postfix expression are:

1. Push the number $3$ onto the stack.
2. Push the number $1$ onto the stack.
3. Read the operator $-$, pop $1$ and $3$ from the stack, compute $3 - 1 = 2$, and push $2$ back onto the stack.
4. Push the number $4$ onto the stack.
5. Read the operator $*$, pop $4$ and $2$ from the stack, compute

$2 * 4 = 8$, and push 8 back onto the stack.

6. The entire expression has been read. The result is 8.

It is clear that this postfix expression is equivalent to the infix expression $(3 - 1) * 4$

1. Evaluate the following postfix expressions:

(1) $3\ 1\ 4\ 5 + \ll 6 - *$, where $\ll$ is the left shift operator

(2) $9\ 3\ 1\ -\ 3\ *\ +\ 10\ 2\ /\ +$

(3) $2\ 2\ 1 + * !\ 9 -$, where ! is the factorial operator

2. Convert the following infix expressions into postfix expressions:

(1) $(A + B) * (C - D)$

(2) $(A\ \&\&\ B)\ ||\ (C\ \&\&\ !\ D)$

(3) $(A\ +\ B\ *\ (C\ -\ D\ \hat{}\ E)\ /\ F)\ *\ G\ -\ H$

3. Suppose a postfix expression contains n numbers and m operators, where the i-th operator requires opt_i operands. What should be the quantitative relationship between n, m, and opt_i in a valid postfix expression?

## T3

1. When a function is called, a run-time stack is needed to store necessary information. What information must be saved when a function is called? Is it

necessary to save the values of all registers?

2. When using recursion to solve problems, a stack overflow error often occurs if the problem size is too large. Why is this? If we convert recursion to iteration and manually use a stack to store data, this situation does not occur. Why?

## T4

Consider the following two C code snippets:

(a)

```c
int a[]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int *b = a;
b[5] = 42;
printf("%d %d", a[5], b[5]);
```

(b)

```c
int a[]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int *b = malloc(10 * sizeof(int));
for (int i = 0; i < 10; i++) {
    b[i] = a[i];
}
b[5] = 42;
printf("%d %d", a[5], b[5]);
```

Questions:

1. What will be the output of each code snippet?

2. These two code snippets demonstrate two different ways of copying arrays. What are their respective advantages and disadvantages?

3. In code snippet (a), after b = a, can b point to a different array? Can a be

reassigned (e.g., a = b)?

## T5

As is well known, memory read/write speed is much slower than that of registers, so instructions that access memory incur a large time cost. Cache technology effectively alleviates this performance problem by inserting a memory called *cache* between registers and main memory.

Suppose a two-dimensional array $A[4][4]$ is stored in LC-3 memory space, each element is a 16-bit integer, and the array starts at memory address **x3000**.

Assume:

- Accessing cache takes **1 unit of time**
- Accessing memory takes **20 units of time**
- Ignore all time except array reads

1. Using the algorithm below to compute matrix multiplication **A × A** without cache, how many time units are required? (Ignore elements temporarily held in registers)

```
for(int i = 0; i < 4; i++){
    for(int j = 0; j < 4; j++){
        for(int k = 0; k < 4; k++){
            calc A[i][k]*A[k][j]
        }
    }
}
```

2. There is a cache whose address space is 4-bit, and each memory block is mapped to the cache line given by the **lower 4 bits** of its address.

On every memory access:

1. Check the corresponding cache line.
2. If the required data are present (hit), read them from cache.
3. If absent (miss), read from memory, then fill the cache line.

Assume the cache is initially empty.

Under these conditions, using the same algorithm to compute **A × A**, how many time units are required? (Again, ignore elements held in registers)

## T6

```c
#include<stdio.h>
struct Test{
    char c[3];
    int i;
}t;
int main(){
    scanf("%s",t.c);
    printf("%d\n",t.i);
    return 0;
}
```

Consider the C code above.

When the string **"abcdef"** is entered, what output does the program produce on your computer?

From this, deduce how the structure and the integer data are laid out in memory on your machine.