Department of Computer Science
Ashoka University

Programming Language Design and Implementation (PLDI): CS-1319-1

Assignment Type: Individual

Assignment - 6: `Optimization, Register Assignment and Code Generation` *Marks: 100*
Assign Date: *December 04, 2023* Submit Date: *23:55, December 18, 2023*

This is a bonus assignment. There will be no extension

As a bonus assignment, this is not counted in the 100% of assignment credit or in the assignment component. The marks scored here will be considered for overall add-on (with capping at 100%).

Assignment 2 through 5 dealt with coding for nanoC *translator. In contrast this assignment is a problem workout.*

Consider the following program to test `495` as the 3-digits Kaprekar's Constant:

```
// IO Library header -- as defined in Assignment 5
int printStr(char *s);
int printInt(int n);
int readInt(int *eP);

// Swap two integers
void swap(int *a, int *b) {
    int t;
    t = *a;
    *a = *b;
    *b = t;
}

// Works for 3 digit numbers only
int kt(int n) {
    int p;  // Previous number
    int d1; // Largest digit
    int d2; // Second largest digit
    int d3; // Smallest digit
    int m;  // Next number

    p = n;  // Remember current number

    // Extract digits in sorted order
    d1 = n % 10;
    n = n / 10;
    d2 = n % 10;
    n = n / 10;
    if (d1 < d2)
        swap(&d1, &d2);
    d3 = n % 10;
    if (d2 < d3) {
        swap(&d2, &d3);
        if (d1 < d2)
            swap(&d1, &d2);
    }

    // Check digits to debug
    printInt(d1);
    printInt(d2);
    printInt(d3);
```

```
    printStr("\n");

    // Compute the diff of largest and smallest
    // three digit numbers with the given digits
    m = (d1 - d3) * 99;

    // Check for the fixed point
    if (m == p)
        return m; // Should return 495 if n != 0
    else
        return kt(m); // Continue search for fixed point
}

int main() {
    int n;
    int m;

    while (1) {
        n = readInt(0);
        m = kt(n);
        printStr("Constant = ");
        printInt(m);
        printStr("\n");
    }

    return 0;
}
```

1. Translate the above program to three address codes:
   (a) Show the Global Symbol Table with the symbol name, data type, category, and size. Mark appropriate parent / child symbol table pointers to build the tree of symbol tables. **[6]**
   Use the type expression ptr(int) for `int*` type.
   (b) Generate the array of quad codes starting at index 100. **[4 + 9 + 6 = 19]**
      i. For function `swap`.
      ii. For function `kt`.
      iii. For function `main`. String constants need handling as constants in static area.
   (c) Show the Symbol Tables for functions `swap`, `kt` and `main` showing the symbol name, data type, category, size, and offset. Mark appropriate parent / child pointers to build the tree of symbol tables.
   Also, show the Table of constants with offset, if needed, and discuss how you would use it in Q1b. **[2 + 4 + 2 + 2 = 10]**
   (d) Discuss how would you handle IO library functions `printStr`, `printInt`, and `readInt` as they are given only as prototype (not with the body). How would the symbol tables of these functions look like (what information would you maintain)? What stage of the code build process would use these information? **[3 + 2 = 5]**
2. Peephole optimize the code of function `kt` with multiple iterations as needed: **[10 + 3 = 13]**
   - propagating copies and removing dead code

Given Code	**Optimized Code**
`x = a + b` `y = x`	`y = a + b`

   - folding constants

Given Code	**Optimized Code**
`x = 3 + 1`	`x = 4`

   - short-cutting jump-to-jump

Given Code **Optimized Code**

```
Given Code              Optimized Code
[110]: goto 113         [110]: goto 119
        ...
[113]: goto 119
```

- eliminating jump-over-jump

```
Given Code                     Optimized Code
[110]: if a > b goto 112       [110]: if a <= b goto 120
[111]: goto 120                [112]: x = 0
[112]: x = 0
```

- applying algebraic simplification

```
Given Code              Optimized Code
    x = p + 0               x = p
```

- applying strength reduction

```
Given Code              Optimized Code
    a = 4 * i               a = i << 2
```

Renumber the peephole optimized quads from 100.

3. Construct the Control Flow Graph (CFG) for `kt`. **[2 + 10 = 12]**

   (a) Identify the leader quads

   (b) Construct the basic blocks and build the CFG. If the control from a basic block falls through to a `goto`, link directly to the target of the `goto`

4. For every block, optimize for local common sub-expression, copy propagation and dead-code elimination within the block.

   Regenerate the array of quads and redraw the CFG after local optimization. **[8 + 2 = 10]**

5. Mark the set of live variables at every program point. That is, every point in every basic block and at the entry and exit of every basic block. Using liveness information, optimize for global common sub-expression, copy propagation and dead-code elimination across the blocks. **[5 + 10 = 15]**

6. Using 4 registers ($r_0, r_1, \cdots, r_3$) generate the target code by Sethi-Ullman algorithm. You may assume any appropriate simple assembly language. **[10]**