CSE Qualifying Exam, Fall 2021, Architecture (513)

1. Consider the following loop:

```
LOOP:
LD F2, 0(R1)
MULT F2, F2, F4
SD F2, 0(R1)
LD F2, 12(R1)
MULT F2, F2, F4
SD F2, 12(R1)
ADDI R1,R1,4
BLT R1,R2,LOOP
```

   a. Does this loop contain a loop carried dependency? Why or why not?

   b. Assume all floating-point instructions (those that operate on F-registers) have a latency of 4 cycles, meaning that there must be 4 cycles separating an instruction that produces a result with another instruction that uses it as input.

      Can you unroll and schedule this loop to eliminate stalls? Are there any complications when doing this? Assume the iteration count is a multiple of the unroll factor.

2. What is the arithmetic intensity of the loop nest below? What is its expected performance if its host machine has a memory bandwidth of 12.8 GB/s and a peak computational throughput of 10 Gflops/s?

   Assume:
   (1) the data type of the res, mat, and vec arrays are double
   (2) the vec array fits in on-chip cache, so you don't need to include accesses to this array in the calculation

```
for (int i=0;i<n;i++) {
  res[i] = 0.0;
  for (int j=0;j<m;j++)
    res[i] += mat[i][j] * vec[j];
}
```

3. Assume a CPU core has a clock rate of 5 GHz, a base CPI of 2.1, and has a 2-level cache with the following characteristics:
    - L1 hit time = 0 s, L1I cache miss rate = 5%, L1D miss rate = 2%
    - L2 hit time = 5 ns, L2 miss rate = 2%, L2 miss time = 20 ns
    - No penalty for write hits or misses

    a. What is its actual CPI? State any assumptions you make.

    b. What is the overall speedup achieved from reducing the L2 miss rate by a factor of 2?

Fall 2021 CSE Qualifying Exam
CSCE 531, Compilers

1. **Syntax-Directed Transformation of Syntax Trees**

   Consider the following grammar for expressions:

$$\begin{array}{lcl} E & ::= & E + E \\ E & ::= & \textbf{num} \end{array}$$

   (a) Describe with a simple English sentence which expressions are generated by the grammar. Show that the grammar is ambiguous.

   (b) Convert the grammar into a left-recursive unambiguous grammar with exactly two productions.

   (c) Recall that *reducing* a syntax tree means replacing any node in the tree that has only one child with that child (which may be empty). A *fully reduced* syntax tree has no nodes with only one child. Draw a fully reduced syntax trees for the expression 1+2+3, where the numbers are values of the **num** token, using the unambiguous grammar from the previous step.

   (d) Add actions to the grammar to produce an abstract syntax tree. Use the following (common) convention: **PlusExp** corresponds to E, **NumExp**(.) corresponds to the value of the token **num**, and terminals and nonterminals on the right-hand of a production are indicated by $\$i$, as usual. Draw the abstract syntax tree for 1+2+3.

   (e) Rewrite the grammar from part (b) to eliminate left recursion. Your grammar must have three productions. Draw the syntax tree for 1+2+3. Draw the fully reduced syntax tree for 1+2+3.

   (f) The syntax trees for 1+2+3 given by the unambiguous grammars with and without left recursion are quite different. In general, while the grammar with left recursion produces trees that reflect expression structure well, the grammar without left recursion does not. Add actions to its rules that construct abstract syntax trees like the ones constucted for the unambiguous grammar with left recursion. This is more easily done in a functional language that supports anonymous functions (using lambda notation); if you use a C-style language, you will need pointers and holes.

2. **Liveness Analysis and Register Allocation**

   Consider the following program.

```
f(a,b)1:  LABEL begin
       2:  IF a<b THEN lab1 ELSE lab2
       3:  LABEL lab1
       4:  x := a + b
       5:  b := a + x
       6:  LABEL lab2
       7:  y := b − a
       8:  a := a/y
       9:  z := a − b
      10:  IF z<0 THEN end ELSE begin
      11:  LABEL end
      12:  RETURN a
```

(a) Compute *succ(i), gen(i), and kill(i)* for each instruction in the program. For your convenience, an example of the table to be filled is provided next to the program.

```
f(a,b)1:  LABEL begin
       2:  IF a<b THEN lab1 ELSE lab2
       3:  LABEL lab1
       4:  x := a + b
       5:  b := a + x
       6:  LABEL lab2
       7:  y := b − a
       8:  a := a/y
       9:  z := a − b
      10:  IF z<0 THEN end ELSE begin
      11:  LABEL end
      12:  RETURN a
```

i	$succ[i]$	$gen[i]$	$kill[i]$
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			

(b) Calculate *in* and *out* for every instruction in the program. Show your work in tabular form. Use of fixed-point iteration is recommended.

(c) Draw the (register-)interference graph for a, b, t, x, y, and z. Also show the interference table (with columns for statement number, kill set, and intereferes with set) that you used to build the interference graph.

(d) Make a three-coloring of the interference graph.

(e) Explain how one could modify the program to use only two registers. You do not need to provide a solution; only describe the approach that you would take.

3. **Predictive (LL(1)) Parsing** Consider the following grammar for postfix expressions:

$$\begin{array}{lcl} E & ::= & EE+ \\ E & ::= & EE* \\ E & ::= & \mathbf{c} \end{array}$$

(a) Eliminate left-recursion in the grammar.

(b) Do left-factorization of the grammar produced in part (a).

(c) Calculate *Nullable*, *FIRST* for every production, and *FOLLOW* for every non-terminal in the grammar produced in part (b).

(d) Make an LL(1) parse table for the grammar produced in part (b).

Fall 2021 CSE Qualifying Exam
CSCE 551, Theory

1. Let Σ be any alphabet. For any languages $L_1, L_2 \subseteq \Sigma^*$, define $L_1 \diamondsuit L_2$ to be the set of all strings obtained by overlapping a string from L_1 followed by a string from L_2. (The "overlap" could be the empty string.). Formally,

$$L_1 \diamondsuit L_2 := \{xyz \mid x, y, z \in \Sigma^* \text{ and } xy \in L_1 \text{ and } yz \in L_2\} .$$

   Show that if L_1 and L_2 are regular, then $L_1 \diamondsuit L_2$ is regular. If your proof involves a correct construction, then you do not need to prove that it is correct.

2. Let Σ be some alphabet. Say that a language $L \subseteq \Sigma^*$ is *length-dependent* iff, for all strings $x, y \in \Sigma^*$, if $x \in L$ and $|x| = |y|$, then $y \in L$. Define the language LD_{TM} as follows:

$$\text{LD}_{\mathsf{TM}} := \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is length-dependent}\} .$$

   (a) Show that LD_{TM} is undecidable by giving a mapping reduction from A_{TM} to LD_{TM} (i.e., $A_{\mathsf{TM}} \leq_m \text{LD}_{\mathsf{TM}}$).

   (b) Show that LD_{TM} is not Turing-recognizable by giving a mapping reduction from $\overline{A_{\mathsf{TM}}}$ to LD_{TM} (i.e., $\overline{A_{\mathsf{TM}}} \leq_m \text{LD}_{\mathsf{TM}}$, or equivalently, $A_{\mathsf{TM}} \leq_m \overline{\text{LD}_{\mathsf{TM}}}$).

   If you give a correct reduction, then you do not need to prove that it is correct. Getting either (a) or (b) correct is worth 70%. Getting both correct is worth 100%.

3. Let $G = (V, E)$ be a graph. An *almost-clique* in G is a set $C \subseteq V$ of vertices such that all pairs of distinct vertices in C are adjacent except for one pair. That is, an almost-clique is a clique except with exactly one edge missing.

   Let ALMOST-CLIQUE be the following decision problem:

   > Instance: A graph G and an integer $K \geq 2$.
   > Question: Does there exist an almost-clique in G with K or more vertices?

   ALMOST-CLIQUE is clearly in NP. Show that ALMOST-CLIQUE is NP-complete by giving a polynomial reduction from CLIQUE to ALMOST-CLIQUE.

   [If your reduction is correct, you do not need to show that it is correct.]

Fall 2021 Q-exam — CSCE 750 (Algorithms)

1. **(Solving a Recurrence)** Let $T(n)$ be any positive-valued function defined for all integers $n \geq 1$ by the following recurrence:

$$T(n) = n + \sum_{i=1}^{\lfloor \log_3 n \rfloor} T(\lfloor n/3^i \rfloor) \ .$$

   Find an expression $f(n)$, as simple as possible, such that $T(n) = \Theta(f(n))$. Use the substitution method to **prove** that your answer is correct.

2. **(Counting Short Paths in a DAG)** You are given a directed acyclic graph (dag) $G = (V, E)$, two vertices $s, t \in V$, and a positive integer $k \leq |E|$. You want to find the number of directed paths from s to t of length at most k (where the length of a path is the number of edges along it).

   Describe an algorithm that takes as input a dag $G = (V, E)$ with $|V| = n$ and $|E| = m$ (given in adjacency list form with vertex array $V[1 \ldots n]$) and a positive integer $k \leq m$ and returns the number of paths of length $\leq k$ from $V[1]$ to $V[n]$. You can assume that the vertices in the array $V[1 \ldots n]$ are arranged so that every edge $(V[i], V[j]) \in E$ satisfies $i < j$. (That is, all edges go "forward" from lower indices to higher ones.)

   Explain how and why your algorithm works, in enough detail to convince an intelligent but skeptical reader that it is correct.

   For full credit, your algorithm must run in time $O(k(n+m))$, assuming all addition operations take $O(1)$ time.

3. **(Binomial Trees)** In a nonempty rooted tree T, the *degree* of a node x in T is the number of x's children. (So all leaves have degree 0, etc.). The *degree* of T is the degree of its root.

   A nonempty, rooted, ordered tree T is a *binomial tree* if the following two conditions hold:

   - if $d \geq 0$ is the degree of T, then the children of the root (there are d of them) have degrees $d-1, d-2, \ldots, 0$ in left-to-right order;
   - all the subtrees of the root of T are binomial trees.

   Prove the following for binomial trees T of degree $d \geq 0$ and root r:

   (a) T has depth d and size 2^d.

   (b) The shape of T is uniquely determined by d, that is, any two binomial trees of degree d have exactly the same shape.

   (c) If $d > 0$, then removing the leftmost child of r results in a binomial tree of degree $d-1$.

   (d) For all $0 \leq i \leq d$, the number of nodes of depth i in T is $\binom{d}{i}$, where $\binom{d}{i} = \frac{d!}{i!(d-i)!}$ is the "d choose i" binomial coefficient.

   HINT for Part (d): Use part (c) and the defining recurrence for binomial coefficients: for all integers $d > 0$ and integers i,

$$\binom{0}{0} = 1 \ , \qquad \binom{d}{i} = \begin{cases} \binom{d-1}{i-1} + \binom{d-1}{i} & \text{if } 0 \leq i \leq d, \\ 0 & \text{otherwise.} \end{cases}$$