Q-exam, Fall 2020, Architecture (CSCE 513)

1. Consider the following loop:

```
for (i=0; i<1024; i++)
      for (j=0; j < n; j++) {
            a[j*64 + i] = a[(j+1) * 64 + i] * a[i];
            a[i] = a[i] + b[j]
      }
```

   1) Is this loop parallel? Why or why not?
   2) Is there a potential strategy to execute the inner loop in multiple threads? How or why not?
   3) Is it possible to unroll the inner loop without the compiler having access to the value of n? Please explain your answer.

2. AMAT: In the system the memory has:
   - Separate L1 instruction and data caches, Hit Time = 1 cycle
   - 64KB L1 instruction cache with 5% miss rate, block size = 32B
   - 64KB L1 data cache with 10% miss rate, block size = 32B
   - 256KB L2 unified cache, Hit Time = 6 cycles, block size = 64B, miss rate = 50%
   - Main Memory Access time is 100 cycles for the first 128 bits and subsequent 128-bit chunks are available every 8 cycles
   - L1 caches are direct mapped, L2 is four-way associative
   - Assume there are no misses to main memory

   1) What is the average memory access time for instruction reference?
   2) What is the average memory access time for data reference?
   3) Assume the only memory reference instructions are loads(20%) and stores(5%). What percentage of total memory references are instruction references?
   4) What is the average memory access time? State any assumptions if needed.

3. The following loop is the so-called DAXPY loop (double-precision *aX* plus *Y*) and is the central operation in Gaussian elimination. The following code implements the DAXPY operation, *Y = aX + Y*, for a vector length 100. Initially, *R1* is set to the base address of array *X* and *R2* is set to the base address of *Y*:

```
      addi    x4,x1,#800 ; x1 = upper bound for X
foo:  fld     F2,0(x1)   ; (F2) = X(i)
      fmul.d  F4,F2,F0   ; (F4) = a*X(i)
      fld     F6,0(x2)   ; (F6) = Y(i)
      fadd.d  F6,F4,F6   ; (F6) = a*X(i) + Y(i)
      fsd     F6,0(x2)   ; Y(i) = a*X(i) + Y(i)
      addi    x1,x1,#8   ; increment X index
      addi    x2,x2,#8   ; increment Y index
      sltu    x3,x1,x4   ; test: continue loop?
      bnez    x3,foo     ; loop if needed
```

Assume the functional unit latencies as shown in the following table. Assume a one-cycle delayed branch that resolves in the ID stage. Assume that results are fully bypassed.

Instruction producing result	Instruction using result	Latency in clock cycles
FP multiply	FP ALU op	6
FP add	FP ALU op	4
FP multiply	FP store	5
FP add	FP store	4
Integer operations and all loads	Any	2

1) Assume a single-issue pipeline. Show how the loop would look both unscheduled by the compiler and after compiler scheduling for both floating-point operation and branch delays, including any stalls or idle clock cycles. What is the execution time (in cycles) per element of the result vector, Y, unscheduled and scheduled? How much faster must the clock be for processor hardware alone to match the performance improvement achieved by the scheduling compiler? (Neglect any possible effects of increased clock speed on memory system performance.)
2) Assume a single-issue pipeline. Unroll the loop as many times as necessary to schedule it without any stalls, collapsing the loop overhead instructions. How many times must the loop be unrolled? Show the instruction schedule. What is the execution time per element of the result?

Fall 2021 CSE Qualifying Exam
CSCE 531, Compilers

1. **Register Allocation**

   Consider the following program to compute the greatest common divisor of two numbers using Euclid's algorithm.

```
gcd(a, b)1:  LABEL start
          2:  IF a<b THEN next ELSE swap
          3:  LABEL swap
          4:  t := a
          5:  a := b
          6:  b := t
          7:  LABEL next
          8:  z := 0
          9:  b := b mod a
         10:  IF b = z THEN end ELSE start
         11:  LABEL end
         12:  RETURN a
```

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

```
gcd(a, b)1:  LABEL start
          2:  IF a<b THEN next ELSE swap
          3:  LABEL swap
          4:  t := a
          5:  a := b
          6:  b := t
          7:  LABEL next
          8:  z := 0
          9:  b := b mod a
         10:  IF b = z THEN end ELSE start
         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, and z.

   (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.

2. **Syntax-Directed Definition** Consider the following grammar for arithmetic expressions with constants, addition, and multiplication, where S is the start symbol and **c** is a numeric constant:

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

   (a) Show that the grammar is ambiguous.

   (b) Assume that + and * are associative and that, as usual, * has higher precedence that +. Rewrite the grammar to eliminate ambiguity, thus obtaining the standard LR (bottom-up) grammar for arithmetic expressions with constants, addition, and multiplication. (Use subscripts to indicate different occurrences of the same nonterminal in the same production.)

   (c) Write an attribute grammar by adding semantic rules to the grammar you just obtained that, given an input expression, produces an equivalent expression with the minimum number of parentheses. So the rules in effect remove unnecessary parentheses. Your resulting expression should be passed as a string attribute to S.output. Assume that the terminal **c** has a text attribute that contains the string representing the constant. Use '+' in your actions to denote string concatenation, and please surround string constants with double quotes. As for the previous part of this question, assume that + and * are associative operators, and that the usual precedence rules apply (* before +). Do not rearrange or alter the expression in any way other than by removing unnecessary parentheses.

Input	**Output**	**Comment**
2 + (3 + 4)	2 + 3 + 4	addition is associative
(2 * 3) * 4	2 * 3 * 4	multiplication is associative
2 + (3 * 4)	2 + 3 * 4	multiplication has precedence over addition
(2 + 3) * 4	(2 + 3) * 4	parentheses needed

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

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

   (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 2020 CSE Qualifying Exam
CSCE 551, Theory

1. Let $\Sigma := \{a, b, c\}$. For any string $w \in \Sigma^*$, let $r(w)$ be the string in Σ^* that results by removing from w every occurrence of "b" that immediately follows an occurrence of "a". For example,

   $$r(abc) = ac \qquad r(abb) = ab \qquad r(ac) = ac \qquad r(cb) = cb \qquad r(ba) = ba$$

   (Note that $r(w)$ may still have a "b" immediately following an "a".)

   For any language $L \subseteq \Sigma^*$, define

   $$r(L) := \{r(w) \mid w \in L\} .$$

   Show that if L is regular, then $r(L)$ is regular.

2. Let A be any language over some alphabet Σ. Suppose there is a computable function f such that, for any $x, y, z \in \Sigma^*$,

   $$f(x, y, z) = \begin{cases} 1 & \text{if } x, y, z \text{ are pairwise distinct and } A \cap \{x, y, z\} = \emptyset, \\ 0 & \text{otherwise.} \end{cases}$$

   Show that A is decidable. [Note that $f(x, x, x) = 0$ for all x, so just computing $f(x, x, x)$ does not tell you anything about A.]

3. Let G be a (simple, undirected) graph with n vertices and m edges. An *exact half vertex cover (ehvc)* of G is a set C of vertices of G such that *exactly* $m/2$ edges have at least one endpoint in C (the other $m/2$ edges having neither endpoint in C). Obviously, for an exact half vertex cover to exist, m must be even.

   Let EHVC be the following decision problem:

   > Instance: A graph G and a positive integer K.
   > Question: Does G have an ehvc of size $\leq K$?

   Clearly, EHVC is in NP. Show that EHVC is NP-complete by describing a polynomial reduction from VERTEX COVER to EHVC.

Fall 2020 Q-exam — CSCE 750 (Algorithms)

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

$$T(n) = 2T(n/2) + (1/2)T(n-1) + n$$

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. (Note: Implicit floors or ceilings in the recurrence do not affect the answer.)

2. **(Cutting a sheet)** Suppose you have a large, flat sheet of metal, measuring n units long and m units wide, where n and m are positive integers. Rectangles made of this particular metal can be very valuable for certain construction projects, but their value depends on the specific dimensions of the rectangle. Specifically, you are given a price array $P[1, \ldots, n][1, \ldots, m]$, in which $P[i, j]$ is a non-negative integer that represents the amount of money that a rectangle of metal i units long and j units wide can be sold for. (You should not assume anything about the relative values of different sizes of rectangles.) You have a saw that can make a complete horizontal cut or a complete vertical cut across any rectangle of the metal, forming two smaller rectangles. These cuts must be made at integer positions. There is no cost to use the saw. Your goal is to make a series of zero or more cuts, starting from the original $n \times m$ sheet, to maximize the total value of the resulting rectangles.

   Describe an algorithm, as efficient as possible, that takes as input the original dimensions n and m along with the price array $P[1, \ldots, n][1, \ldots, m]$, and outputs the maximum total value that can be obtained. **Explain** how and why your algorithm works, in enough detail to convince an intelligent but skeptical reader that it is correct.

3. **(Building a new road)** Consider a weighted directed graph $G_1 = (V, E_1)$ in which the vertices represent cities and the edges represent existing roads that directly connect pairs of cities. Weights on the edges represent the travel times along those roads. Thus, for any two vertices $s, t \in V$, the length of the shortest path in G_1 between s and t represents the total time needed to travel from s to t.

   Now suppose we are given a new weighted directed graph $G_2 = (V, E_2)$, for which $E_1 \cap E_2 = \emptyset$. Notice that the two graphs share the same vertex set V. Edges in E_2 represent new roads that might possibly be built in the future.

   Suppose only **one** new road from E_2 can be built. We are interested in determining, for a given pair of cities (s, t), the 'best' new road to build. That is, which one new road from E_2 would cause the length of the shortest path in G_1 from s to t to be reduced the most?

   Since the edge weights represent travel time, all of the weights in each of the graphs G_1 and G_2 are non-negative.

   Specifically, you should **describe** an algorithm that:

   (a) takes as input the two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ represented as adjacency lists, along with two vertices s and t, and

   (b) outputs an edge $e \in E_2$ that minimizes the length of the shortest path from s to t in the graph $(V, E_1 \cup \{e\})$. (If more than one edge leads to the minimum, your algorithm may return any of them.)

   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 should run in $O(|E_1| \log |V| + |E_2| \log |V|)$ time.