Problem 1. Software Pipelining
Consider the following loop

for(i = 0; i < n; i++) {
    X[i+1] = X[i] * 2
    Y[i+1] = X[i+1] + Y[i]
}

For this problem, assume that ADD/MUL takes 2 clocks, while LD/ST takes 1 clock. The machine has two MEM units that can execute a LD and ST, and two ALU units that can execute ADD and MUL in one clock. The machine can auto-increment address registers.

1. Draw the data dependence graph by showing four types of nodes, LD, ADD, MUL, and ST.

Assembly code (assuming R0 and R1 are properly initialized pointers corresponding to X and Y, respectively):

1: LD R2 *(R0++)
2: MUL R3, R2, $2
3: ST *(R0), R3
4: LD R4, *(R0)
5: LD R5, *(R1++)
6: ADD R6, R4, R5
7: ST *(R1) R6
Grading note: the autoincrement can be interpreted as a read/write, and additional edges may have been present in the student’s solution (e.g. a self loop on node 1), and this was also accepted.

2. Label the edges of the dependency graph according to the type of dependency (true dependency, anti-dependency or output dependency)

   All edges are RAW dependencies (true), but other answers were accepted depending on the interpretation of the autoincrement or alternative assembly translation.

3. What is the lower bound on the initiation interval?
   4 due to the cycle length of 4 and iteration difference of 1

   Some students only considered resource constraints, and as it wasn’t made explicit in this problem, an answer of $\lceil 2.5 \rceil$ was also accepted.
Problem 2. Dependency analysis and parallelization
Consider the following program:

```c
for(i = 1; i < n; i++) {
    for(j = i; j < 2 * i; j++) {
    }
}
```

1. Draw the iteration space for this loop.

2. What are the possible data dependencies in this loop? A data dependency is something of the form “read/write A[i], read/write B[j]”.
   - read A[i-1, j], write A[i, 2*j]
   - write A[i, 2*j], read A[i-1, j] (across iterations)
   - write A[i, 2*j], write A[i, 2*j] (across iterations)

3. Is this loop nest 1 or 2-d parallelizable without loop transformations? You do not need to show the exact solution to the equations, but justify your answer.
   This loop cannot be be parallelized as written. We see a dependence across i iterations, as iteration i=1, j=1 writes A[1, 2], while iteration i=2, j=2 reads A[1, 2].
Problem 3. Global Instruction Scheduling

Assume you have a statically scheduled machine that can only issue one operation every clock. All operations have a latency of one clock cycle, with the exception of its memory load operation, which has a latency of three clock cycles. Consider the following locally scheduled program:

Assume that only \( r \) is live at the end of the program. Each branch in the flow graph is labeled with the probability that it is taken dynamically. To answer the following, you may apply any of the code motions discussed in class, but no other optimizations.

1. Is this the best globally scheduled code that can be generated given that \( p = 0.1 \)? If not, provide the improved code along with its expected execution time.
Path 1: $L0 \rightarrow L1 \rightarrow L3$
$4 + 3 + 1 = 8$ instructions at $0.9 \times 0.99 = 0.891$ probability

Path 2: $L0 \rightarrow L1 \rightarrow L2 \rightarrow L3$
$4 + 3 + 4 + 1 = 12$ instructions at $0.9 \times 0.01 = 0.009$ probability

Path 3: $L0 \rightarrow L2 \rightarrow L3$
$4 + 4 + 1 = 9$ instructions at 0.1 probability

The expected execution time is $8 \times 0.891 + 12 \times 0.009 + 9 \times 0.1 = 8.136$ clock cycles.

2. Repeat part 1 given that $p = 0.9$.

The code cannot be further improved, as we cannot speculatively execute $w = a$ with $w$ being live in $L3$.

3. What is the percentage improvement in execution time from the original program for your programs in part 1 and 2, if one was provided?

Path 1: $L0 \rightarrow L1 \rightarrow L3$
$2 + 6 + 1 = 9$ instructions
When $p = 0.9$, 0.099 probability, otherwise 0.891

Path 2: $L0 \rightarrow L1 \rightarrow L2 \rightarrow L3$
$2 + 6 + 4 + 1 = 13$ instructions
When $p = 0.9$, 0.001 probability, otherwise 0.009 probability

Path 3: $L0 \rightarrow L2 \rightarrow L3$
$2 + 4 + 1 = 7$ instructions
When $p = 0.9$, 0.9 probability, otherwise 0.1 probability

There is no improvement for $p = 0.9$ and $9 \times 0.891 + 13 \times 0.009 + 7 \times 0.1 = 8.836$ clock cycles for a $\approx 8\%$ improvement for $p = 0.1$
Problem 4. More Software Pipelining
Consider the following dependence graph for a single iteration of a loop, with resource constraints:

![Dependence Graph](image)

All instructions take 1 cycle each.

1. What is the bound on the initiation interval $T$ according to the precedence and resource constraints for this program?

   4 due to data dependence in cycle Max (cycle length/ iteration diff) = $4/1 = 4$

   4 due to resource constraints as well.

2. What is the minimum initiation interval? Show a modulo reservation table for an optimal software pipelined schedule. Also show the code and schedule for an iteration in the source loop.

   Modulo reservation table:

<table>
<thead>
<tr>
<th>A</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>B</td>
</tr>
</tbody>
</table>

   Single iteration: A, nop, nop, B, nop, nop, nop, nop, nop, C, D, nop, E

3. Can the scheduling algorithm described in class produce the optimal schedule for this loop?

   Yes
Problem 5. Affine Transforms. Apply affine transform to find the largest degree of outermost loop parallelism. Show the transformed code, marking the loops that are parallelizable.

```c
for (i = 1; i <= N; i++) {
    for (j = 1; j <= N; j++) {
        A[i,j] = X[i,j] + X[i,j];
    }
}
for (i = 1; i <= N; i++) {
    for (j = 1; j <= N; j++) {
        for (k = 1; k <= N; k++) {
            B[i,k] = B[i,k] + Y[j];
        }
    }
}
for (i = 2; i <= N; i++) {
    for (j = 1; j <= N; j++) {
        C[i,j] = A[i-1,j] * A[i-1,j];
    }
}
for (i = 1; i <= N; i++) {
    D[i] = C[2,i] + B[2,i];
}
```

Solution:

```c
for (pi = 0; pi <= N; pi++) { // parallelizable
    for (pj = 1; pj <= N; pj++) { // parallelizable
        if (pi >= 1 && pi <= N) {
            A[pi,pj] = X[pi,pj] + X[pi,pj];
        }

        if (pi >= 0 && pi <= N - 1) {
            for (j = 1; j <= N; j++) {
                B[pi+1,pj] = B[pi+1,pj] + Y[j];
            }
        }

        if (pi >= 1 && pi <= N - 1) {
        }

        if (pi == 1) {
            D[pj] = C[pi+1,pj] + B[pi+1,pj];
        }
    }
}
```