Problem 1. Software Pipelining

Consider a simple loop below

```c
for(i = 1; i<n; ++i) {
    B[i] = 2 * A[i];
}
```

Assume LD and ST take 1 clock, ADD and MUL take 2 clocks. The machine has two MEM units that can execute one LD and one ST, and two ALU units that can execute ADD and MUL in one clock. The machine can autoincrement address registers.

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

First we write down the assembly code corresponding to this loop (assuming R0 and R1 are properly initialized pointers):

1: LD *(R0++), R2
2: ADD R2, $1, R3
3: ST R3, *(R1)
4: LD *(R1), R4
5: MUL $2, R4, R5
6: ST R5, *(R1++)
The dependency graph, annotated, is:

Note that we do not need to include output dependencies across iterations. There is also no WAR dependency between $A[i-1]$ and $A[i]$ because they refer to different locations.

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

3. What is the lower bound on the initiation interval?

Based on precedence constraints, the lower bound is 4, because we have a cycle of length 4 with an iteration difference of 1. Based on resources, each iteration requires 4 MEM and 2 ALU, so the lower bound is 2. Overall, the lower bound is 4.
Problem 2. Data Dependency Analysis

Consider the loop nest below:

```c
for(i = 1; i<n; i++) {
    for(j = i+1; j<m; j++) {
        Z[i, j] = 2 * Y[2*i+1];
    }
}
```

1. Draw the iteration space for this loop.

   If $n < m$, the iteration space is:

   ![Iteration Space Diagram 1](image1)

   Otherwise, the iteration space is:

   ![Iteration Space Diagram 2](image2)

   Either solution was acceptable.
2. What are the possible data dependencies in this loop? Hint: a data dependency is something of the form “read/write A[i], read/write B[j]”.

- read Y[2*j], write Y[2*j]
- write Y[2*j], read Y[2*i+1]
- write Y[2*j], write Y[2*j] (across iterations)
- write Z[i,j], write Z[i,j] (across iterations)

3. What are the data dependency equations that must be satisfied for the loop to be parallelizable?

Given iterations $i, j$ and $i', j'$, we want to check if the data dependencies are actually possible (Slide 14 of Lecture 9):

- $0 \leq i < n$
  $0 \leq i' < n$
  $i + 1 \leq j < m$
  $i' + 1 \leq j' < m$
  $2j = 2j'$

- $0 \leq i < n$
  $0 \leq i' < n$
  $i + 1 \leq j < m$
  $i' + 1 \leq j' < m$
  $2j = 2i' + 1$

- $0 \leq i < n$
  $0 \leq i' < n$
  $i + 1 \leq j < m$
  $i' + 1 \leq j' < m$
  $2j = 2j'$
  $j \neq j'$
\[ 0 \leq i < n \\
0 \leq i' < n \\
i + 1 \leq j < m \\
i' + 1 \leq j' < m \\
\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} i' \\ j' \end{bmatrix} \\
\begin{bmatrix} i \\ j \end{bmatrix} \neq \begin{bmatrix} i' \\ j' \end{bmatrix} \]

(It was ok to omit the loop boundary constraints in each data dependency equation, as they are the same.)

4. 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.

The inner loop writes to \(Y[2j]\) and reads from \(Y[2i+1]\). One is even the other is odd. Therefore there is no dependency between these two. There is also no carried dependency inside the inner loop (it is a do-all loop), so the inner loop is indeed parallelizable. The outer loop is not: multiple threads can write to the same \(Y[2j]\) different values at different \(i\) iterations (write-write dependency). This also means the loop nest is not 2d parallelizable.

5. Is this loop nest 1 or 2-d parallelizable with loop transformations? Again, you do not need to show the exact solution, but justify your answer.

Split the loop the writes \(Z\) from the one that writes \(Y\). This is allowed because there is no dependency from writing \(Y\) and reading \(Y\). Then the \(Z\) loop becomes 2d parallelizable.
Problem 3. Global Instruction Scheduling

SLOWMO is a statically scheduled machine that can only issue one operation every clock. Its memory load operation (denoted by the * dereference operator) has a latency of three clocks; all other operations have a latency of one clock. Consider the following locally scheduled program:

Assume that \( r \) is the only live variable at the end of the program. Each branch in the flow graph is labeled with the likelihood that it is taken dynamically.

1. Assuming that \( p = 0.01 \), what is the best globally scheduled code that you can generate for the above program? You may apply any of the code motions discussed in class and introduce extra nops to fit the schedule. What is the expected execution time of the scheduled program?

Label L3 is the only block which can benefit from global instruction scheduling. However, because it is so rarely executed most changes will have a net increase on the overall expected runtime of the block. For \( p = 0.01 \), we can effectively move code upwards from L3 to L1 (since L3 will be executed 99% of the time if L1 is executed), but not to L0 or L2. The best schedule is as follows:
We can calculate the expected execution time of the optimized program by adding up the number of instructions on each path multiplied by the probability of executing that path.

Path 1: $L_0 \rightarrow L_1 \rightarrow L_4$
2 + 3 + 1 = 6 instructions
$.01 \times .01 = .0001$ probability

Path 2: $L_0 \rightarrow L_1 \rightarrow L_3 \rightarrow L_4$
2 + 3 + 7 + 1 = 13 instructions
$.01 \times .99 = .0099$ probability

Path 3: $L_0 \rightarrow L_2 \rightarrow L_2' \rightarrow L_3 \rightarrow L_4$
2 + 2 + 3 + 7 + 1 = 15 instructions
$.99 \times .01 = .0099$ probability

Path 4: $L_0 \rightarrow L_2 \rightarrow L_4$
2 + 2 + 1 = 5 instructions
$.99 \times .99 = .9801$ probability

The expected instruction count is 5.1783 (a speedup of .2% over the original program).

2. Repeat part 1 for $p = 0.99$.

Similarly for $p = .99$ we can only move code upwards from block L3 to L2 without incurring a net increase in the programs execution time. The best schedule is as follows:
The expected execution time of the new program is as follows:

Path 1: \( L_0 \rightarrow L_1 \rightarrow L_4 \)
\[ 2 + 1 + 1 = 4 \text{ instructions} \]
\[ .99 \times .99 = .9801 \text{ probability} \]

Path 2: \( L_0 \rightarrow L_1 \rightarrow L_1' \rightarrow L_3 \rightarrow L_4 \)
\[ 2 + 1 + 4 + 6 + 1 = 14 \text{ instructions} \]
\[ .99 \times .01 = .0099 \text{ probability} \]

Path 3: \( L_0 \rightarrow L_2 \rightarrow L_3 \rightarrow L_4 \)
\[ 2 + 4 + 6 + 1 = 13 \text{ instructions} \]
\[ .01 \times .99 = .0099 \text{ probability} \]

Path 4: \( L_0 \rightarrow L_2 \rightarrow L_4 \)
\[ 2 + 4 + 1 = 7 \text{ instructions} \]
\[ .01 \times .01 = .0001 \text{ probability} \]

The expected instruction count is 4.1884 (a speedup of .5% over the original program).

3. Find the expected execution time of the scheduled program in part 1 and part 2 under \( p = 0.40 \). Are they better than the original program?

The probability of paths being executed are 0.16, 0.24, 0.24, 0.36. The expected instruction count for part 1 is 9.48 and for part 2 is 9.64. The expected instruction count for original program is 9.4. The original program is marginally better than the extreme optimizations.
Problem 4. More Software Pipelining

Consider the following dependence graph for a single iteration of a loop, with resource constraints:

Instructions A, D and E take 2 clock cycles and B and C 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>E</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>B</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
</tr>
</tbody>
</table>

Single iteration:
3. Can the scheduling algorithm described in class produce the optimal schedule for this loop? Discuss whether and how the original algorithm should be improved.

The scheduling algorithm described in class will backtrack only within strongly connected components, and schedule greedily across them. Therefore it will not produce an optimal schedule: it will pack D in the middle of the table (in the first available spot), leaving no space for E.

To obtain the optimal schedule, one needs to allow multilevel backtracking. This considerably slows down the algorithm. It was also acceptable to argue that it is not wise to backtrack as this slows down the algorithm and we are ok with a little approximation to avoid very large scheduling times.