Directions:

• This homework includes a Gradiance quiz and some written questions. The written part should be submitted via Gradescope. No late days can be applied to the Gradiance quiz.

• You may use up to two of your remaining late days for this assignment, for a late deadline of March 3, 2023 at 11:59pm.

• This assignment is an individual assignment. You are allowed to discuss the homework with others, but you must write the solution individually. If you look up any material in the textbook or online, you should cite it appropriately.
Problem 1. Global Instruction Scheduling

Assume you have a machine with a statically scheduled processor 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:

\[
\begin{align*}
\text{d} &= \ast a; \\
\text{nop;} \\
\text{nop;} \\
\text{if (d } \geq 0) \text{ goto L1;}
\end{align*}
\]

1. Is this the best globally scheduled code that can be generated given that \( p = 0.9 \)? If not, provide the improved code along with its expected execution time. **Note:** for an if branch instruction, assume that if the branch corresponding to “goto L” is taken, it costs two cycles; otherwise it costs only one.

In the original program, the number of clock cycles and the probability for each path are shown as follows:

- \( L0 \rightarrow L1 \rightarrow L4, p = 0.09, \text{num\_cycles} = 8 \)
- \( L0 \rightarrow L1 \rightarrow L3 \rightarrow L4, p = 0.81, \text{num\_cycles} = 13 \)
- \( L0 \rightarrow L2 \rightarrow L4, p = 0.1, \text{num\_cycles} = 10 \)
The expected execution time is $8 \times 0.09 + 13 \times 0.81 + 10 \times 0.1 = 12.25$ cycles.

We can get a better schedule by pushing the instructions in L4 upwards to fill in the idle points when the program is waiting for the result of a memory load.

The number of clock cycles and the probability for each path are shown as follows:

- L0 → L1 → L4, $p = 0.09$, num_cycles = 7
- L0 → L1 → L3 → L4, $p = 0.81$, num_cycles = 11
- L0 → L2 → L4, $p = 0.1$, num_cycles = 9

The expected execution time is $7 \times 0.09 + 11 \times 0.81 + 9 \times 0.1 = 10.44$ cycles.

2. Repeat part 1, but instead assume that $p = 0.1$.

In the original program, the number of clock cycles and the probability for each path are shown as follows:

- L0 → L1 → L4, $p = 0.09$, num_cycles = 8
- L0 → L1 → L3 → L4, $p = 0.01$, num_cycles = 13
- L0 → L2 → L4, $p = 0.9$, num_cycles = 10

The expected execution time is $8 \times 0.09 + 13 \times 0.01 + 10 \times 0.9 = 9.85$ cycles.

With the following optimized program:
the number of clock cycles and the probability for each path are shown as follows:

- L0 → L1 → L4, \( p = 0.09 \), \( \text{num\_cycles} = 7 \)
- L0 → L1 → L3 → L4, \( p = 0.01 \), \( \text{num\_cycles} = 14 \)
- L0 → L2 → L4, \( p = 0.9 \), \( \text{num\_cycles} = 8 \)

The expected execution time is \( 7 \times 0.09 + 14 \times 0.01 + 8 \times 0.9 = 7.97 \) cycles.
**Problem 2.** Software Pipelining

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

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

   Both the $B$–$C$–$D$–$B$ and $E$–$F$–$E$ cycles impose a bound of 5 due to data dependence, as computed by dividing cycle length by iteration difference. The left resource imposes a bound of 5 and the right resource 4. So overall, $T \geq 5$. 

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.

The minimum initiation interval is 5. Modulo reservation table:

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

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

3. Can the scheduling algorithm described in class produce the optimal schedule for this loop? If not, show the modulo reservation table and code schedule generated by the algorithm.

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 the \{B, C, D\} SCC immediately after A (in the first available spot), eliminating the ability interleave the \{E, F\} SCC in-between.

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

Here, we are stuck, since we cannot schedule F 3 clocks after E. However, the algorithm will be able to schedule instructions with an initiation interval of 6. Modulo reservation table:

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

Single iteration: A, B, C, nop, E, D, nop, F
Consider the following do-all loop program:

for (i = 0; i < 1000; i++)
    \( C[i] = (A[i] - b) \times A[i]; \)

One iteration of the loop can be written in assembly as:

1. LD R5, 0(R1++) // R1 = &A[i]; R5 = *R1; R1++
2. SUB R6, R5, R2 // R2 = b; R6 = R5 - R2
3. 
4. MUL R6, R6, R5 // R6 = R6 \times R5
5. 
6. ST R6, 0(R3++) // R3 = &C[i]; *R3 = R6; R3++

Optimize this program using software pipelining for a machine with the following specifications:

- The processor can issue at most one LD instruction, one ST instruction, and one arithmetic instruction within the same clock cycle.
- Each arithmetic operation has a two-cycle latency, but can be pipelined.
- The processor supports auto-incrementing addressing and hardware loop operations.

Answer the questions below using your optimized program:

1. What is the minimum initiation interval? 2

2. Write down the pipelined portion of the loop using actual registers. Annotate each instruction with the iteration it's associated with (e.g., \( i, i+1, i+2 \), etc.). Resolve any anti-dependencies across loop iterations by unrolling the loop as described in class.

\[
\begin{align*}
\text{ST } & R6, 0(R3++) (i) & \text{MUL } & R16, R16, R15 (i+1) & \text{LD } & R5, 0(R1++) (i+3) \\
\text{ST } & R16, 0(R3++) (i+1) & \text{MUL } & R26, R26, R25 (i+2) & \text{LD } & R15, 0(R1++) (i+4) \\
\text{ST } & R26, 0(R3++) (i+2) & \text{MUL } & R6, R6, R5 (i+3) & \text{LD } & R25, 0(R1++) (i+5) \\
\end{align*}
\]

Here's the full program for reference:

![The full program for reference]
3. How many more registers does software pipelining require compared to the unpipelined code? Think of a way we can reduce the number of registers used without decreasing the throughput. (You do not have to show the generated code.) *Hint*: check the live ranges for all the duplicated registers.

We need 4 more registers (R15, R16, R25, R26). There are two ways to reduce registers:

(a) Notice that R25 isn’t strictly needed, since the live ranges of R5 and R25 do not overlap. We can remove the need for R25 by unrolling the loop 6 times rather than 3 with the following register assignment:

i. R5 – R6
ii. R15 – R16
iii. R5 – R26
iv. R15 – R6
v. R5 – R16
vi. R15 – R26

In general, we can reduce register pressure by unrolling \( n \) times, where \( n \) is the least common multiple of \( \lceil \text{lifetime}_r / T \rceil \) for all registers \( r \). This is in contrast with the algorithm in lecture 8, which suggests unrolling \( \max_r \lceil \text{lifetime}_r / T \rceil \) times.

(b) Alternatively, we can change the program a bit to break up the live range of R6:

LD R5, 0(R1++)
SUB R6, R5, R2
MUL R6, R6, R5 → MUL R7, R6, R5
ST R6, 0(R3++) → ST R7, 0(R3++)

Because of this, we now only need to unroll twice and create another copy each of R5 and R6. Compared to the original program, this also adds 3 registers (R7 and the two copies).
Problem 4. Dependency Analysis and Parallelization

Consider the C-style program:

```c
for (i = k+2; i < n; i++) {
    for (j = 2*i+m; j < 6*n-i; j++) {
        X[i, 4*j-2] = X[i, 4*j+1] + Y[i, j]
        Y[i+1, j-3] = X[i, 2*j] + Y[i, j]
    }
}
```

Assuming $0 \leq m \leq k \ll n$ and that $X$ and $Y$ are non-overlapping arrays, answer the following questions.

1. Draw the iteration space for this loop. *Hint:* The axes should be situated such that program execution first goes from left to right, and next from bottom to top.

![Iteration Space Diagram](image)

2. What are the data dependencies in this loop? Categorize each as a true dependency, output dependency, or an anti-dependency. Include all dependency, even if they can be ruled out using simple tests like the GCD test. *Hint:* a data dependency is something of the form “read/write $A[i]$, read/write $B[j]$.”

   (a) True dependency: write $X[i, 4j - 2]$, read $X[i, 4j + 1]$
   (b) True dependency: write $X[i, 4j - 2]$, read $X[i, 2j]$
   (c) True dependency: write $Y[i + 1, j - 3]$, read $Y[i, j]$
   (d) Output dependency: write $Y[i + 1, j - 3]$, write $Y[i + 1, j - 3]$ (optional)
   (e) Output dependency: write $X[i, 4j - 2]$, write $X[i, 4j - 2]$ (optional)
3. Formulate the data dependence tests for the given loop nest.

Bounds equations:

\[ k + 2 \leq i < n, \quad 2i + m \leq j < 6n - i, \]
\[ k + 2 \leq i' < n, \quad 2i' + m \leq j' < 6n - i'. \]

Unique indices test:

\[(i, j) \neq (i', j')\]

(a) write \(X[i, 4j - 2]\), read \(X[i, 4j + 1]\):
\[i = i' \land 4j - 2 = 4j' + 1\]

(b) write \(X[i, 4j - 2]\), read \(X[i, 2j]\):
\[i = i' \land 4j - 2 = 2j'\]

(c) write \(Y[i + 1, j - 3]\), read \(Y[i, j]\):
\[i + 1 = i' \land j - 3 = j'\]

(d) write \(Y[i + 1, j - 3]\), write \(Y[i + 1, j - 3]\)
\[i + 1 = i' + 1 \land j - 3 = j' - 3\]

(e) write \(X[i, 4j - 2]\), write \(X[i, 4j - 2]\)
\[i = i' \land 4j - 2 = 4j' - 2\]

Combine each of each of the conditions in (a)–(e) the bounds conditions and unique indices test. If the equations have integral solutions, then there is a data dependence.

(a) can be easily ruled out using the GCD test. (d) and (e) can also be ruled out as they conflict with the unique indices test.

4. Is the loop nest 1-d parallelizable or 2-d parallelizable without loop transformations (or not parallelizable at all)? If not parallelizable, provide an example pair of loop indices that cause the conflict.

Suppose \(k = 0\) and \(m = 0\).

The inner loop is not parallelizable, as the write to \(X[i, 4j - 2]\) and read from \(X[i, 2j]\) can overlap for fixed \(i\) indices (this is dependency (b) above). As an example, iteration \((i=2, j=10)\) will write to \(X[2, 38]\), while \((i'=2, j'=19)\) will read from \(X[2, 38]\).

The outer loop is not parallelizable either due to dependency (c). As an example, \((i=9, j=25)\) will write to \(Y[10, 22]\), while \((i'=10, j'=22)\) will read from \(Y[10, 22]\).

Therefore, this loop nest is not parallelizable at all.
Problem 5. Affine Transforms

Apply affine transform to find the largest degree of outermost loop parallelism. Show the transformed code, and mark the loops that are parallelizable. Assume $N \geq 100$.

\begin{verbatim}
for (i = 1; i <= N; i++) {
    for (j = 1; j <= N; j++) {
        A[i,j] = A[i-1,j] + X[i,j];
    }
}
for (i = 4; i <= N; i++) {
    for (j = i+1; j <= N; j++) {
        for (k = 1; k <= N; k++) {
            B[i,k] = B[i-1,k] + Y[j];
        }
    }
}
for (i = 1; i <= N; i++) {
    C[i] = A[3,i];
}
\end{verbatim}

Solution:

\begin{verbatim}
for (pj = 1; pj <= N; pj++) { // parallelizable
    for (pi = 1; pi <= N; pi++) {
        if (pi >= 4) {
            for (j = pi+1; j <= N; j++) {
                B[pi,pj] = B[pi-1,pj] + Y[j];
            }
        }
        if (pi == 3) {
            C[pj] = A[pi,pj];
        }
    }
}
\end{verbatim}

This uses the following index mappings:

- For $A$ loop: \[
\begin{bmatrix}
    p_i \\
    p_j
\end{bmatrix} = \begin{bmatrix}
    1 & 0 \\
    0 & 1
\end{bmatrix}
\begin{bmatrix}
    i_A \\
    j_A
\end{bmatrix},
\]

- For $B$ loop: \[
\begin{bmatrix}
    p_i \\
    p_j
\end{bmatrix} = \begin{bmatrix}
    1 & 0 \\
    0 & 1
\end{bmatrix}
\begin{bmatrix}
    i_B \\
    k_B
\end{bmatrix},
\]

- For $C$ loop: \[
\begin{bmatrix}
    p_i \\
    p_j
\end{bmatrix} = \begin{bmatrix}
    0 \\
    1
\end{bmatrix}
i_C + \begin{bmatrix}
    3 \\
    0
\end{bmatrix}.
\]

There is only one degree of outermost loop parallelism.
Other acceptable solutions:

1. It’s acceptable for the B loop to not be fused together with A and C, since there is no dependency between the second loop and the other two loops.

2. Noticing that B and A/C loops are completely independent, another way to do the transform is to execute B completely in parallel with A/C:

   ```
   for (pk = 1; pk <= 2; pk++) { // parallelizable
       for (pj = 1; pj <= N; pj++) { // parallelizable
           for (pi = 1; pi <= N; pi++) {
               if (pk == 1) {
               }
               if (pk == 2 && pi >= 4) {
                   for (j = pi+1; j <= N; j++) {
                       B[pi,pj] = B[pi-1,pj] + Y[j];
                   }
               }
               if (pk == 1 && pi == 3) {
                   C[pj] = A[pi,pj];
               }
           }
       }
   }
   ```

This uses the following processor mapping:

- For A loop:
  \[
  \begin{bmatrix}
  p_i \\
  p_j \\
  p_k
  \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix}
  i_A \\
  j_A
  \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix},
  \]

- For B loop:
  \[
  \begin{bmatrix}
  p_i \\
  p_j \\
  p_k
  \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix}
  i_B \\
  k_B
  \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 2 \end{bmatrix},
  \]

- For C loop:
  \[
  \begin{bmatrix}
  p_i \\
  p_j \\
  p_k
  \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} i_C + \begin{bmatrix} 3 \\ 0 \\ 1 \end{bmatrix}.
  \]

Caveats:

1. Notice that the j loop in the B loop nest is not actually parallelizable, since two independent iterations of j can write to the same location in memory.

   (You can reduce the j loop to the last iteration of j since that’s the only one that sticks, but that’s not something that can be automatically done using an affine transform.)