Learning Objectives

1. Analyzing run-time on a variety of pipelines.
2. Practice with scheduling code to mitigate stalls and flushes.
3. Loop unrolling

Test Preparation

In this section, we’re going to focus on solving some software optimization problems relating to running code on pipelined processors. We won’t provide all of the solutions to these problems in class. We want to encourage you to solve these on your own (they are practice for the exam). We’ll post solutions next week.
Loop Unrolling

Consider the following code:

```c
def fib(int n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

which could be written in assembly as the following:

```assembly
li $t0, 0   # i = 0
li $t1, 1   # pow = 1
bge $t0, $a1, loop_done
loop:       mul $t1, $t1, $a0   # pow *= in
            add $t0, $t0, 1
            blt $t0, $a1, loop
loop_done:
```

1. If this loop were to execute on the 5-stage pipeline discussed in class, how many cycles would it take from the fetch of the first instruction to the writeback of the last instruction if the loop iterated 10 times? (to simplify this, pretend that the `bge` and `blt` execute as a single machine instruction). Assume that branches are predicted as `not-taken` and resolved in the EX stage.

2. If you knew that the value of `N ($a1)` was always going to be even, how could you re-write the assembly code to make it run faster?

3. Compute the number of cycles your new loop will take given the same conditions as above.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pipelining without forwarding

Compute the average time per iteration for the following loop for a MIPS 5-stage pipeline that predicts branches as not-taken, resolves branches in the EX stage, and implements no forwarding. Assume that a register read in ID will receive the correct value if the same register is being written in that cycle in WB. To get any credit, you must document your computation.

```
bit_reverse_loop:
srl  $t1, $a0, $t0
and  $t1, $t1, 1
sll  $t1, $t1, $t2
or   $v0, $v0, $t1
sub  $t2, $t2, 1
add  $t0, $t0, 1
blt  $t0, 32, bit_reverse_loop
```

Loop Unrolling

Consider the following code:

```
for (int i = 0 ; i < N ; i ++) {
    A[i] ++;
}
```

(a) Compute the average number of cycles per iteration for this loop, on a deeply-pipelined machine with the following characteristics: 1) the load-to-use latency is 4 (meaning that the result of a load can’t be used by the subsequent 3 instructions without stalling), and 2) the machine includes a dynamic branch predictor and BTB so that the loop back-edge branch can be predicted correctly with a 0-cycle taken branch penalty. Assume that $N$ is very large.

(b) Assume that $N$ is a multiple of four and rewrite this code using the loop unrolling optimization. Loop unrolling replicates the loop body by some factor—four in this case—to provide more independent instructions with which to schedule the code. When the copies are made, it is generally necessary to rename some of the temporary registers to avoid artificial dependences on certain registers. Schedule your code so as to avoid as many stalls as possible and compute the new average number of cycles per iteration (recognizing that the new iterations do 4 times as much work as the original loop) for the same processor as is discussed in part (a).
Substituting Arithmetic Operations for Control Flow

Consider the two following implementations of a population count function:

```c
unsigned pop_count(unsigned in) { 
    unsigned count = 0;
    for (; in != 0 ; in = in >> 1) {
        if (in & 1) {
            count ++;
        }
    }
    return count;
}
```

The above version uses control flow. The below version uses arithmetic on the logical operations.

```c
unsigned pop_count(unsigned in) { 
    unsigned count = 0;
    for (; in != 0 ; in = in >> 1) {
        count += (in & 1);
    }
    return count;
}
```

(a) Estimate the relative performance of these two versions of the code on a 5-stage MIPS pipeline with forwarding that predicts branches as not taken, resolves branches in the EX stage, and resolves unconditional jumps in the ID stage. Assume that each bit of the parameter to the function has a 50% chance of being set.

(b) How would the relative performance change if the pipeline implemented 2-bit dynamic branch prediction with a BTB in the IF stage (giving a 0 cycle taken branch penalty for both predicted taken branches and unconditional jumps)? Assume branches are still resolved in the EX stage, so that the misprediction penalty is 2 cycles. State any assumptions that you made.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>