Question 1: Pipelining (40 points)

Above is the pipeline discussed in class.

**Part (a)** The processor is currently executing the code shown to the right. In the current cycle, the `addi` instruction is in the writeback stage. Assume that the branch is not taken. Compute the control signals for the following instruction and enter your answers on the scantron. For all questions, use the following key: a) 0, b) 1, c) 2, d) 3, e) don’t care/can’t know. (10 points)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>PC</th>
<th>instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00010000</td>
<td>addi</td>
<td>$1, $0, 768</td>
</tr>
<tr>
<td>…</td>
<td></td>
<td>bne $1, $2, C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lw $3, 16($1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>add $2, $2, $3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ori $2, $2, 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sw $2, 16($1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Func. Unit</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU/Adder</td>
<td>4ns</td>
</tr>
<tr>
<td>Register File</td>
<td>3ns</td>
</tr>
<tr>
<td>Cache</td>
<td>5ns</td>
</tr>
</tbody>
</table>

20 ns \(=\) ICACHE + RF_READ + ALU + DCACHE + RF_WRITE \(=\) 5 + 3 + 4 + 5 + 3 = 20ns

and, what is the largest speedup that can be achieved by pipelining shown above (e.g., consider when there is a large number of instructions executed and there are no stalls and flushes)? Explain. (5 points)

Max speedup = \(\frac{4}{1}\) Old time/new time = 20ns / 5ns

Longest pipeline stages are IF and MEM 5ns each, so our shortest clock period is 5ns. If there is a large number of instructions, then pipeline fill is negligible, so we complete 1 inst/cycle, just like the single cycle.
For all of this page consider the MIPS assembly code below. Corresponding C code is shown on the right.

```
loop:
    sll $t1, $t0, 2
    add $t1, $a0, $t1
    lw $t2, 0($t1)
    lw $t3, 8($t2)
    bne $t3, $t7, skip
    add $v0, $v0, 1

skip:
    add $t0, $t0, 1
    slt $t1, $t0, $a1
    bne $t1, $0, loop
```

**Part (d)** Assume this loop iterate many times, and the if statement is executed 50% of the time. Compute the average number of cycles that this loop takes per iteration. Explain your answer for partial credit. *Assume conditional branches are resolved in the EX stage.* (10 points)

If the branch isn't taken, it's 9 instructions + 1 lw-lw stall + 1 lw-bne stall + 2 cycles for the loop branch flush for a total of 13 cycles.

If the branch is taken, it's 8 instructions + the same two stalls + 2 cycles for the taken branch flush + 2 cycles for the loop branch flush for a total of 14 cycles.

Since the branch is taken 50% of the time, it's 13.5 cycles / iteration on average.

**Part (e)** Re-schedule/re-write the loop to make it faster. Faster code will achieve more points, but your answer must fit in the space below. (5 points)

Hoist independent instructions from the code after the if to avoid stall cycles after loads.

```
loop:  sll $t1, $t0, 2
        add $t1, $a0, $t1
        lw $t2, 0($t1)
        lw $t3, 8($t2)
        slt $t1, $t0, $a1  # hoist
        bne $t3, $t7, skip
        add $v0, $v0, 1

skip:  bne $t1 $0, loop
```

**Part (c)** Annotate the MIPS assembly to indicate all of the true data dependences, including those that span loop iterations. (5 points)
Consider the data path shown above.

**Part (f)**
In what pipeline stage are jump register (jr) instructions resolved? *Explain how you know.* Feel free to highlight parts of the data path to support your answer. (5 points)

Jump register is resolved in the EX stage. This can be seen from the path that moves a register value to the program counter is latched in the ID/EX latch before reaching the PC.
Question 2: Cache Analysis (25 points)

Part (a) For a 32KB 2-way set-associative, write-back cache with 64B blocks on a machine with 32-bit address spaces (both virtual and physical) and no hardware prefetching, compute the average number of misses per inner-loop iteration for the following loop nest (and explain how you computed it!). Assume everything is in registers, except the data structures A, B, and C. Ignore address translation for this question. (20 points)

```
const int N = 37, M = 2196;
double A[N][M], C[N]; // floats are 4 bytes, doubles are 8 bytes big
struct annotate { int index; float offset; double scaling; };
struct annotate B[M];

for (int i = 0 ; i < N ; i ++) {
    for (int j = 0 ; j < M ; j += 2) {
    }
}
```

0 + \(\frac{1}{4}\) + \(\frac{1}{2}\) = \(\frac{3}{4}\) misses per iteration.

There is a potential for a small rate of aliasing misses, because C[i], A[i][j], and B[j] could map to the same (2-way associative) set, which couldn’t hold all 3 cache blocks at the same time. With 256 sets, however, the probability that all 3 blocks map to the same set is relatively small.

Part (b) How would the miss rate change if we inverted the loops (shown below)? Explain. (5 points)

```
for (int j = 0 ; j < M ; j += 2) {
    for (int i = 0 ; i < N ; i++) {
    }
}
```

0 + \(\frac{1}{4}\) + 0 = \(\frac{1}{4}\) misses per iteration.

Now has temporal locality. \(~0\) misses per iteration.

Has spatial locality. Furthermore, this whole array is only 37*8B = 296B, much smaller than the cache. Because this data will remain in the cache across iterations, we’ll have misses in the first outer loop iteration, then hits resulting in \(~0\) misses per inner loop iteration.

No temporal locality. No spatial locality within an outer loop iteration, but because N is small, you can fit a whole column of A in the cache at a time (along with the C array): N=37, so we need 37 cache blocks to hold the column, while our cache has 512 blocks. Because M is not a power of 2, these blocks are unlikely to alias, so when I load A[0][2], A[0][0] is likely still in my cache, meaning that (in general) I’ll use all 4 even elements of this cache block that I brought in, for a \(\frac{1}{4}\) misses per inner loop iteration.
Question 3: Cache-aware Programming (25 points)

Part (a) Rewrite the following code to optimize its cache performance on a system with no support for prefetching and a single-level cache. The cache is a 2-way set associative 16KB cache with 32B blocks. Floats are 4 byte data types. (15 points)

```
#define N 100000
float A[N], B[N], total_sum = 0;

for (int i = 0 ; i < N ; i += 20) {
    for (int j = 0 ; j < N ; j += 20) {
        total_sum += A[i] * B[j];
    }
}
```

Neither array access exhibits spatial locality, because the loops are incrementing by 20 and 20x4B (floats) = 80B (stride) > 32B (block size).

There is temporal locality in the accesses to A[i] in the inner loop and reuse of the same values in B[j] across outer loop iterations, but the current loop structure prevents exploiting any temporal locality in B, because 100000/20 = 5000 blocks of B are accessed and only 512 fit in the cache at a time. So by the time we get to the end of the inner loop, we've kicked out the blocks from the beginning of the loop.

To enable reuse, we can apply tiling so that each element of B that is brought in is used many times before we kick it out. Applying tiling in this way alleviates the bandwidth requirement of having to bring the block in a bunch of times. This is important in this loop because there is not a lot of computation to be done for each cache block brought in (i.e., we're only using one of the elements in the cache block).

```
const int T = 20*x; // tile size; make sure it is a multiple of 20
for (int i = 0 ; i < N ; i += T) {
    for (int j = 0 ; j < N ; j += T) {
        for (int ii = i ; ii < (i+T) ; ii += 20) {
            for (int jj = j ; jj < (j+T) ; jj += 20) {
                total_sum += A[ii] * B[jj];
            }
        }
    }
}
```

The tile size should be selected so that tiles of both A and B fit in the cache: 2*x*32B <= 16KB. We've prefetched in addition to tiling, so that we can reduce the time to bring the tile in. Because the prefetches extend past the end of the tile a little, we may want to back off the maximum size tile.
Question 3: Cache-aware Programming (cont.)

Part (b) Rewrite the following code to optimize its cache performance on a system with no support for prefetching and a single-level cache. Assume the L1 cache is a 2-way set associative 32KB cache with 32B blocks. Double precision floating point numbers are 8 byte data types. (10 points)

```c
const int N = 10000;
double A[N], B[N], C[N], D[N], E[N];
for (int i = 0 ; i < N ; i ++) {
    C[i] = k * (D[i] + B[i]);
}
for (int i = 0 ; i < N ; i ++) {
    E[i] = A[i] / E[i];
    D[i+1] = (D[i] + C[i]);
}
```

Current code is inefficient because we go through the arrays A, C, and D in both loops. We can reduce the number of misses for array A by moving the first statement of the second loop to the first loop. The last statement has a WAR dependence on the 2nd statement so it is trickier.

```c
for (int i = 0 ; i < N ; i ++) {
    C[i] = k * (D[i] + B[i]);
    E[i] = A[i] / E[i];
}
for (int i = 0 ; i < N ; i ++) {
    D[i+1] = (D[i] + C[i]);
}
```

The dependance in the last statement can be mitigated by shifting that loop by one iteration relative to the first loop. To do this, we peel off the first iteration of the first loop and the last iteration of the second loop and merge the remains. Now we make a single pass through all of the arrays.

```c
    B[0] = A[0] * A[0];
    C[0] = k * (D[0] + B[0]);
    E[0] = A[0] / E[0];

    for (int i = 1 ; i < N ; i ++) {
        C[i] = k * (D[i] + B[i]);
        E[i] = A[i] / E[i];
        D[i] = (D[i-1] + C[i-1]);
    }
    D[N] = (D[N-1] + C[N-1]);
```
Multiple Choice (10 points)

Choose the best answer and **answer on the provided scantron**.

For a 16KB 4-way set-associative, write-back cache with 64B blocks on a machine with 32-bit address spaces (both virtual and physical): *(for questions 11-12)*

11) How many bits is the block offset?  
   a) 3  
   b) 4  
   c) 5  
   d) 6  
   e) 7

12) How many bits is the index?  
   a) 3  
   b) 4  
   c) 5  
   d) 6  
   e) 7

13) Which of the following is false, or select (e) if all of the statements are true:
   a. The page table base pointer can only be written by the operating system.
   b. A page fault is when the desired page is not currently mapped into memory.
   c. Accessing multi-level page tables involves using a different portion of the virtual page number as the index for each level of the page table.
   d. Virtual memory systems rely on both spatial and temporal locality for efficiency.
   e. **all of the statements are true.**

14) If everything else is equal, a higher clock period means better performance.  
   A) TRUE  
   B) FALSE

15) When reading a single random sector from a disk, what is likely to be the smallest component of the read time:
   A) circulation latency  
   B) transfer time  
   C) transition time  
   D) seek time  
   E) rotational latency

16) Pipelining improves:  
   A) **Throughput, but not latency**  
   B) Latency, but not throughput  
   C) **Both latency and throughput**  
   D) Neither latency nor throughput

17) Assuming the array B is large, what kind(s) of prefetching are likely to effective for prefetching accesses to B?

   ```c
   for (int i = 0 ; i < LENGTH ; i ++) {
       A[i] = B[C[i]]
   }
   ```

   A) hardware prefetching  
   B) **software prefetching**  
   C) either hardware or software prefetching  
   D) neither hardware nor software prefetching

18) In real machines, the least recently used policy works because it takes advantage of:

   A) **spatial and temporal locality in programs**  
   B) spatial locality in programs (not temporal)  
   C) temporal locality in programs (not spatial)  
   D) neither spatial nor temporal locality

8
Multiple Choice, cont.

For the following pieces of code, indicate which forms of locality are definitely exhibited for the accesses to the bolded data structure:

- a) temporal locality
- b) spatial locality
- c) both
- d) neither

19) for (int i = 0 ; i < NUM ; i ++) {
    for (int j = 0 ; j < NUM ; j ++) {
        A[i][j] = initial_value[i];
    }
}

20) for (int i = 0 ; i < NUM ; i ++) {
    x += flights[i]->cost;
}