Question 1: Pipelining (40 points)

Above is the pipeline discussed in class, except with forwarding from the WB stage removed. Unconditional jumps are resolved in ID. Conditional branches are predicted not taken and resolved in EX. Register writes takes place in the first half and reads in the second half of the cycle.

Part (a) The processor is currently executing the code shown to the right. In the current cycle, the lw instruction is in the WB stage. Compute the control signals (marked 1-10 on the diagram) for this code sequence and enter your answers on the scantron. For this question, use the following key: a) 0, b) 1, c) 2, d) 3, e) don’t care/can’t know. (10 points)

```
func:  lw  $t0, 0($a0)   #11  2 cycle stall
       bne $t0, 0, A       # taken 80%           
       move $v0, $t0
       j   end

A:   lw  $t1, 0($t0)     #13
     li  $t0, 0
     bge $t1, $a1, skip  # taken 20%
     li  $t0, 1

skip:  add $t2, $t0, $t1
      add $t3, $t2, $a2
      add $v0, $t2, $t3
      sw  $t2, 0($a0)
      add $v1, $t3, $a3

end:  jr  $ra
```
Question 1: Pipelining (cont.)

For all of this page use the MIPS assembly code and the datapath on the previous page.

(b) Annotate the MIPS assembly to indicate all of the true data dependences. (5 points)

(c) Indicate (using #11 through #20 on the scantron) whether each instruction labeled with a number stalls relative to the immediately proceeding instruction. For each numbered instruction enter: (a) no new stall, (b) a 1-cycle stall, (c) a 2-cycle stall. (5 points)

(d) Calculate the average number of cycles the function func takes to run from the instruction fetch (IF) stage of the first instruction to the writeback (WB) of the last instruction. Explain. Feel free to use the grid at the end of this exam to aid in your computation, but you should put a full explanation here. (10 points)

(4 filling + 2 insts + 2 stalls) + 0.2 * (2 insts + 1 flush + 1 inst) + 0.8 * (2 flushes + 3 insts + 1 stall + 0.2 * 2 flushes + 0.8 * 1 inst + 6 insts + 2 stalls)

Alternative method:
bne not taken: 4 filling + 5 insts + 2 stalls + 1 flush = 12 cycles (20%)
bne taken, bge not taken: 4 filling + 12 insts + 5 stalls + 2 flushes = 23 cycles (80% * 80%)
bne taken, bge taken: 4 filling + 11 insts + 5 stalls + 4 flushes = 24 cycles (80% * 20%)

Overall is 20.96 cycles either way. Okay to leave answer as expression.

(e) Rewrite/reschedule the function func to make it faster. Your new code should produce identical results as the original code and you shouldn’t change the interface (e.g., the arguments/return value). (10 points)

```assembly
func:  lw $v0, 0($a0)   # load directly into $v0; avoids move
         # nothing can be done for stall here
     beq  $v0, 0, end     # flip branch; now fallthrough 80%
     lw  $t1, 0($v0)
     bge $t1, $a1, skip  # taken 20%
     add $v0, $t1, 1     # restructure to avoid adding 0 to $t0
skip:   add $t3, $v0, $a2
     add $v1, $t3, $a3   # hoisted to accept forwarded $t3
     sw  $v0, 0($a0)
     add $v0, $v0, $t3   # push down; avoid missing byapss stall
end:    jr  $ra
```

Also, if you noticed that the 'skip' branch could be replaced with an slt, you earned extra credit.
Question 2: Cache Analysis (25 points)

Consider the following loop nest along with a 128KB direct-mapped, write-back L1 cache with 64B blocks on a machine with 32-bit address spaces (both virtual and physical) and no hardware prefetching.

Assume that \( \text{sum}, i, j, \) and \( k \) are allocated to registers for the duration of the code. The \( \text{pow}(), \) \( \text{polynomial}() \), and \( \text{reduce}() \) functions do not perform any memory accesses. Assume that the memory instructions are executed in the order specified by the program and that addresses translation maps virtual addresses to the same physical addresses. You should also assume that arrays \( A \) and \( B \) are contiguous in memory.

```c
const int length = 24 * 1024;
int A[length], B[length]; // ints are 4 bytes big
int sum;

for (int i = 0 ; i < 4 ; i ++) {
    for (int j = 0 ; j < length ; j ++) {
        sum += polynomial(A[j], i + 1);
    }
    for (int k = 0 ; k < length ; k ++) {
        sum += reduce(B[k], pow(i, 2));
    }
}
```

**Part (a) Compute the number of misses for i=0, that is, for the first iteration of the outermost loop (and explain how you computed it!).** (10 Points)

You are accessing both arrays sequentially, and you are only accessing each array element once per outer loop iteration, so reuse/aliasing don't come into play at all for \( i = 0 \). 16 elements fit in a block (64B blocks / 4B integers), so you have a 1/16 miss rate for both \( A \) and \( B \), for a total of \( 2 \times 24 \times 1024 / 16 = 3072 \) misses. (# arrays * length of arrays / # elements per cache block)

If the array \( A \) doesn’t begin at a cache block boundary, you would have 1 additional miss, but that is negligible.
Question 2: Cache Analysis (cont.)

**Part (b)** Compute the number of misses for \( i = 1 \), that is, for the second iteration of the outermost loop (also **explain how you computed it**). (10 Points)

A and B combined are 1.5 times the size of the cache (96KB + 96KB = 192KB vs. 128KB). Because the arrays are laid out one after the other in memory, and because we access A in order before accessing B in order, we can think of them as a single large array that we access in order.

When \( i = 0 \), the last 2/3’s of B will kick out the first 2/3’s of A, so at the end of the iteration we’ll have the last 1/3 of A and all of B in the cache. This means that we’ll miss (at the 1/16\(^{th}\) rate) for the first 2/3’s of A. By touching this data, we’ll kick out the last 2/3’s of B. Since the last 1/3\(^{rd}\) of A and first 1/3\(^{rd}\) of B will still be in the cache, we’ll have no misses on those. But since A displaced the last 2/3\(^{rd}\)s of B we’ll have misses on that (again at the 1/16\(^{th}\) rate).

You'll therefore get 2/3\(^{rd}\)s as many misses as \( i = 0 \) or 2/3 * 3072 = 2048.

**Part (c)** Compute the number of misses for \( i = 1 \) assuming the cache is fully associative (instead of direct mapped). Assume the cache uses an LRU (Least Recently Used) replacement policy. **Explain.** (5 Points)

LRU means A\(_{1/3}\) will kick out A\(_{3/3}\) and A\(_{2/3}\) will kick out B\(_{1/3}\), and so on and so forth. We'll keep kicking out useful things and won't be able to exploit the reuse, so our number of misses will go back to 3072.
Question 3: Cache-aware Programming (25 points)

(a) Rewrite the following to optimize the cache performance of the code on a system with no support for prefetching and a single-level cache. The cache is a 2-way set associative 32KB cache with 32B blocks. Ints and floats are 4 byte data types. Assume that the hills array starts at an address divisible by 4096. Explain the transformations you applied and why. Your score will depend both on how much you optimize and how well you explain. (15 points)

```
#define NUM_ANT_TYPES 16
#define DIRECTIONS 8
#define NUM_HILLS_RECORDED 81000
struct anthill {
   // anthill is 128B
   float longitude, latitude; // 2 x 4B
   int tunnels[DIRECTIONS]; // 8 x 4B
   int ant_count[NUM_ANT_TYPES]; // 16 x 4B
   char loc_name[24]; // 24 x 1B
};

struct anthill hills[NUM_HILLS_RECORDED]; // filled in elsewhere
int total_ants[NUM_ANT_TYPES];
for (int i = 0 ; i < NUM_ANT_TYPES ; i ++) {
   total_ants[i] = 0;
   for (int j = 0 ; j < NUM_HILLS_RECORDED ; j ++) {
      total_ants[i] += hills[j].ant_count[i];
   }
}
```

// the work in the inner loop gets executed much more than the initialization of total_ants, so we should // focus on that. As written, the reuse of each hills data structure is significantly separated in time. We // would like to do loop inversion to increase locality in the hills data structure. This makes the access // pattern of total_ants worse, but since that array is only 16 elements, we don't care. To do the loop // inversion, we need to do loop fission first.

```
for (int i = 0 ; i < NUM_ANT_TYPES ; i ++) {
   total_ants[i] = 0;
}
for (int j = 0 ; j < NUM_HILLS_RECORDED ; j ++) { // loop interchange
   for (int i = 0 ; i < NUM_ANT_TYPES ; i ++) {
      total_ants[i] += hills[j].ant_count[i];
   }
}
```

// Also, would be beneficial to move the ant_count to be the first or // last element in the structure, so that it is all in one cache block.
Question 3: Cache-aware Programming (cont.)

(b) Apply tiling to the following code example to optimize cache behavior. 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. Explain your choice of tile size. (10 points)

```c
// Can be tiled either in one dimension or two. In both cases, we only
// need to make the tiles big enough that we benefit from spatial
// locality in both the A & B matrices. With 32B blocks and 8B types,
// each cache block holds 4 data items. If we knew for a fact that the
// matrices were aligned to a cache block boundary, 4 (or any multiple of
// 4) would be optimal. If the matrices do not start at cache block
// boundaries, a blocking factor of 4c yields c+1 misses (in one
// dimension), so we benefit from a higher c (to minimize miss rate).

// The blocking factor needs to be small enough to make sure the data
// doesn't get displaced before we finish the cache block. From a
// capacity perspective, the inner loop is touching 5/4 cache blocks per
// iteration, so TILE_SIZE <= CACHE_SIZE / (BLOCK_SIZE * 5/4) or 819.
// Conflicts are likely to be a problem long before this blocking factor
// so a number in the range of 20-40 is probably the best bet.

for (int i = 0 ; i < N ; i += TILE_SIZE) { // one-dimension version
    for (int j = 0 ; j < N ; j ++) {
        for (int ii = i ; ii < min(N, i + TILE_SIZE) ; ii ++) {
            B[ii][j] = A[j][ii];
        }
    }
}

// calls to the min function aren't necessary if N % TILE_SIZE == 0.

for (int i = 0 ; i < N ; i += TILE_SIZE) { // two-dimension version
    for (int j = 0 ; j < N ; j += TILE_SIZE) {
        for (int ii = i ; ii < min(N, i + TILE_SIZE) ; ii ++) {
            for (int jj = j ; jj < min(N, j + TILE_SIZE) ; jj ++) {
                B[ii][jj] = A[jj][ii];
            }
        }
    }
}
```
Multiple Choice (10 points)

Choose the best answer and answer on the provided scantron.

21) If everything else is equal, a higher CPI means better performance.  
A) TRUE    B) FALSE

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

23) How many bits is the block offset?  
a) 5  b) 6  c) 7  d) 8  e) 9

24) How many bits is the index?  
a) 5  b) 6  c) 7  d) 8  e) 9

25) 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

26) Which of the following is false, or select (e) if all of the statements are true:

a. Accessing multi-level page tables involves using a different portion of the physical page number as
   the index for each level of the page table. true if replace ‘physical’ with ‘virtual’

b. A page fault is when the desired page is not currently mapped into memory.

c. The page table base pointer can only be written by the operating system.

d. Virtual memory systems rely on both spatial and temporal locality for efficiency.

e. all of the statements are true.

27) When reading a whole track from a disk, what is likely to be the largest component of the read time:

A) rotational latency    B) circulation latency    C) transfer time
D) escape time

28) What kind of prefetching can be used to prefetch for the following code:

```c
for (int i = 1 ; i < (LENGTH – 1); i ++) {
    A[i] = B[i-1] * C[i+1]
}
```

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

29) for (int i = 0 ; i < NUM ; i ++) {
    for (int j = 0 ; j < NUM ; j ++) {
        elements[i][j] = starting_elements[i + j];  \hspace{1cm} (c) both
    }
}

The spatial locality should be obvious. There is a lot of temporal locality because we’re adding i and j, every element of ‘starting_elements’ except the first and last get touched at least twice and the NUM-1th element will get touched NUM times.

30) for (int z = 0 ; z < 4096; z += 16) {
    rays[z] = base[z] + flare[z]; \hspace{1cm} (d) neither
}

Since we’re doing a single pass through the array, there is no temporal locality. Since we’re accessing only every 16th element of rays, there will only spatial locality if sizeof(rays base type) * 16 < cache block size, so spatial locality is not definitely exhibited for rays. (For example, if rays is an array of doubles (8B type) and cache blocks are only 32 bytes (8B * 16 = 128 B is not < 32B)).