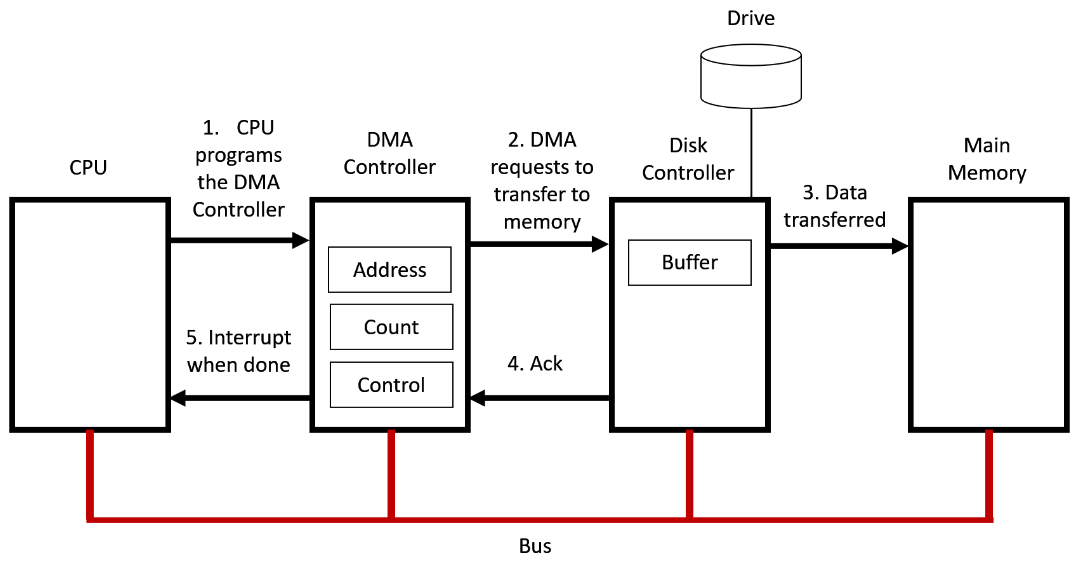
Homework 08 Answer Key

Problem 1 - 100 points - Describe Direct Memory Access and the 5-step process of the CPU performing a DMA request.

Allows I/O devices to directly read/write main memory. DMA engine contains registers written by CPU:

- Memory address to place data
- # of bytes
- I/O device #, direction of transfer
- unit of transfer, amount to transfer per burst



Problem 2 - 100 points - Define Cache Coherence and state and describe the two types of Cache Coherency.

Cache Coherence is the consistency of shared resource data that ends up stored in multiple local caches.

Write Through: write to cache and memory at the same time.

- Can be very slow
- Works well if the store frequency is << 1 memory write per cycle.

Write Back: write to cache only. Write the cache block to memory during a page swap

- Need a "dirty" bit for each cache block
- Greatly reduce the memory bandwidth requirement
- Control can be complex

Problem 3 - 50 points - Describe and differentiate between coarse-grained, fine-grained, and simultaneously multithreading

Fine-grain multithreading

- Switch threads after each cycle
- Interleave instruction execution
- If one thread stalls, others are executed

Coarse-grain multithreading

- Only switch on long stall (e.g., L2-cache miss)
- Simplifies hardware, but doesn't hide short stalls (eg, data hazards)

Simultaneous Multithreading

- In multiple-issue dynamically scheduled processor
- Schedule instructions from multiple threads
- Instructions from independent threads execute when function units are available
- Within threads, dependencies handled by scheduling and register renaming

Problem 3 - 250 points - Consider the following code segment, with the values MAX = 10^4 and WSIZE = 4 and the following data:

- Read from x, w, or y array takes 20 cycles (accounts for lw add)
- Write to a register takes 1 cycle.
- Multiply takes 10 cycles (accounts for writing to the register)
- Each addition takes 5 cycles (accounts for writing to the register)
- Write to y takes 20 cycles
- Each comparison takes 5 cycles
- L1 cache hits require 5 cycles
- Cache miss penalty is 10 cycles

Code segment:

```
C/C++
for( i = 0; i < MAX; ++i ){
    t = 0;
    for( j = 0; j < WSIZE; ++j ){
        t += x[i+j]*y[j] + w[i];
        y[i] += t*i*j;
    }
}
```

A) Without any modifications, how many cycles does the code segment take?

i=0 -> Once - 1 cycle

i < MAX -> 5 cycles each -> 5 * 10^4 cycles

++i -> 5 cycles each -> 5 * 10^4 cycles

t=0 -> 1 cycle -> 1 * 10^4 cycles

j=0 -> 1 cycle -> 1 * 10^4 cycles

j < WSIZE -> 5 cycles per loop -> 4 internal loops -> 5 * 4 * 10^4 cycles = 20* 10^4 cycles

++j -> 5 cycles per loop -> 4 internal loops -> 5 * 4 * 10^4 cycle s= 20* 10^4 cycles

i+j -> 5 cycles per loop -> 4 internal loops -> 5 * 4 * 10^4 cycles = 20* 10^4 cycles

x[i+j] -> 20 cycles for a read -> 4 internal loops -> 20 * 4 * 10^4 cycles = 80* 10^4 cycles

y[j] -> 20 cycles for a read -> 4 internal loops -> 20 * 4 * 10^4 cycles = 80* 10^4 cycles

w[i] -> 20 cycles for a read -> 4 internal loops -> 20 * 4 * 10^4 cycles = 80* 10^4 cycles

x[i+j]*y[j] -> 10 cycles for a multiply -> 4 internal loops -> 10 * 4 * 10^4 cycles = 40* 10^4 cycles

t += x[i+j]*y[j] + w[i] ->Two Additions take 10 total cycles -> 4 internal loops -> 10 * 4 * 10^4 cycles = 40* 10^4 cycles

t*i*j; -> Two multiples take 20 total cycles -> 4 internal loops -> 20 * 4 * 10^4 cycles = 80* 10^4 cycles

y[i] += t*i*j -> Read from y, Add and a write to y take a total of (20 + 5 + 20) cycles -> 4 internal loops -> 45 * 4 * 10^4 cycles = 180* 10^4 cycles

Total: (5+5+1+1+20+20+20+80+80+80+40+40+80+180)* 10^4 + 1 cycles = 572*10^4 + 1 cycles = 652*10^4+1 = 6,520,001

B) Discuss the opportunities for reduction of cycle times and determine the improvement in performance when implemented in SIMD with 32 processors, assuming strong scaling.

The case where y[i] is written at the end of the loop poses a unique challenge with optimization, one that our understanding of Computer Architecture will help us with. Since we change y[i], we must account for the first four loops where the value is updated, and we must reload into a register. Once we pass i >= WSIZE, we no longer need to update those values, but we need to check every cycle anyway.

There are two possible approaches to this code, and we will calculate and see which approach is better, given our architecture:

```C/C++
// Case 1 - Cascading if/else
if( i == 1 ){
    j0 = y[0];
}
else if( i == 2 ){
    j1 = y[1];
}
else if( i == 3 ){
    j2 = y[2];
}
else if( i == 4 ){
    j3 = y[3];
}
// Case 2 - One if and four loads
if( i < WSIZE ){
    j0 = y[0];
    j1 = y[1];
    j2 = y[2];
    j3 = y[3];
 }
```

For case 1, in the first loop it will make one comparison and one read (5 + 20). The second loop will perform two comparisons and one read (10 + 20), The third loop will perform three comparisons and one read (15 + 20), the fourth comparison will perform four comparisons and a read (20 + 20). The first four cases will take 25+30+35+40 = 130. The remaining 10^4 - 4 loops will perform four comparisons.(10^4 - 4)*20 cycles. Therefore, the final result is 20*10^4 + 50 cycles

For case 2, in the first four loops, there will be one comparison and four reads (4 * (5+80)). So the first four cases will take 340 cycles. and the rest will be one comparison (10^4 - 4)*5. Therefore, the final result is 5*10^4 + 320.

We now see that Case 2 is better *in the long run*. In the short term, for the first four cases, Case 1 provides an improvement of 130 compared to 340 cycles. But in the long run, we gain an improvement of 20,050 / 5320 by using case 2 since we are only performing one branch operation for the rest of the loops. Through our understanding of Computer Architecture, we demonstrate that taking a short-term performance penalty can equate to a long-term performance gain.

We also do not need to initialize j0-j4 at the onset because they are properly initialized in the first iteration of the for loop before they are used. So there is no save to register penalty necessary

This makes the final improvement in code equal to:

```C/C++
int j0, j1, j2, j3, t;

for( i = 0; i < MAX; ++i ){
    w_val = w[i];
    if( i < MAX ){
        j0 = y[0];
        j1 = y[1];
        j2 = y[2];
        j3 = y[3];
    }
    t = 0;
    j_intermediate = 0;
    t += x[i]*j0 + w_val;
    t += x[i+1]*j1 + w_val;
    j_intermediate += t*i;
    t += x[i+2]*j2 + w_val;
    j_intermediate += 2*t*i;
    t += x[i+3]*j3 + w_val;
    j_intermediate += 3*t*i;
    y[i] += j_intermediate;
}
```

We can also load the sections of x, w, and y into the cache to reduce their hits to 5 cycles, with a penalty of 10 cycles fora miss every loop:

i=0 -> Once - 1 cycle

i < MAX -> 5 cycles each -> 5 * 10^4 cycles

++i -> 5 cycles each -> 5 * 10^4 cycles

i < MAX comparisons: 5* 10^4 + 320 cycles (previously calculated on previous page)

t=0 -> 1 cycle -> 1 * 10^4 cycles

i+1, i+2, and i+3 for the internal loop -> 15 cycles each cycle -> 15 * 10^4 cycles

Load w and x into the cache:

w[i] -> 5 cycles for a read -> 1 operation each loop -> 5 * 10^4 cycles

x[VAL] -> 5 cycles for a read -> 4 operations in each loop -> 20 * 10^4 cycles x[VAL]*jVAL -> 10 cycles for a multiply -> 4 per loop -> 10 * 4 * 10^4 cycles = 40* 10^4 cycles

t += x[VAL]*jVAL + w_val ->Two Additions take 10 total cycles -> 4 per loop -> 10 * 4 * 10^4 cycles = 40* 10^4 cycles

j_intermediate += t*i - One multiply and one add -> 1 per loop = (5+10)*10^4

j_intermediate += 2*t*i - Two multiplies and one add -> 1 per loop = (5+20)*10^4

j_intermediate += 3*t*i - Two multiplies and one add -> 1 per loop = (5+20)*10^4

y[i] += j_intermediate; - Read from y is 5 cycles, add is 5 cycles, and write to y is 5 cycles -> -> 1 per loop -> 15*10^4

Cache miss penalty of 10 cycles every loop -> 10 * 10^4 cycles

Total Unparalleled: (5+5+5+1+15+5+20+40+40+15+25+25+15+10)*10^4+320+1 cycles = 226*10^4 + 321 =2,260,321

Strong Scaling: 2,260,321 cycles / 32 processors = 70,635.03 cycles. => ~70,636 cycles.

Overall Improvement ~= 6,520,001 / 70,636 = 92.3x

Note, if the students did not do the load into cache, they would have 20 instead of 5 for all the reads/writes to x, w, and y, but no cache miss. Therefore, the result would be:

- w[i] -> 20 cycles for a read -> 1 operation each loop -> 20 * 10^4 cycles (increase of 15)
- x[VAL] -> 20 cycles for a read -> 4 operations in each loop -> 80 * 10^4 cycles (increase of 60)
- y[i] += j_intermediate; - Read from y is 20 cycles, add is 5 cycles, and write to y is 20 cycles -> -> 1 per loop -> 45*10^4 (increase of 30)
- No cache miss, so there would be a reduction of a cache miss penalty (decrease of 10)
- The increase is 15+60+30-10 = 95.
- The parallel jumps to 226 + 95 = **321**
- The overall would be **321*10^4 + 321 =3,210,321**
- Parallel would become 3,210,321 cycles / 32 processors = 100,322.53 cycles. => ~100,323 cycles
- Improvement would become 6,520,001 / 100,323 = 65x

For the C coding, see the link on the Canvas page