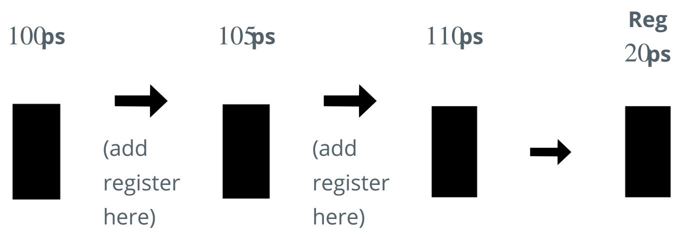
Started on	Friday, 13 November 2020, 3:11 PM
State	Finished
Completed on	Friday, 13 November 2020, 4:24 PM
Time taken	1 hour 13 mins
Grade	**74.70** out of 100.00

Question **1**

Correct

Mark 5.00 out of 5.00

We have a combinatorial logic function that can be decomposed into three steps each with the indicated delay with a resulting clock speed of 2.99GHz.



Assume we further pipeline this logic by adding two additional registers. What would be the resulting clock speed in GHz?

1000/130

You can use an expression if you like.

Your last answer was interpreted as follows: $\frac{1000}{130}$

Correct answer, well done.
The cycle time of the fully pipelined design would depend on the largest stage delay (110) and one register delay. This results in a total cycle time of 130with a resulting clock speed of 7.692307692311GHz.

A correct answer is $\frac{100}{13}$, which can be typed in as follows: `100/13`

Question **2**

Consider the following code sequence in C:

Correct

Mark 5.00 out of 5.00

```
x = 1000; y = 3000;
*q = y;
*p = x;
t1 = *q;
```

What are the possible values of **t1** given all possible values of **p** and **q** including the possibility they have the same value.

Select all values that apply, but you will be penalized for incorrect choices, so don't guess.

Select one or more:

- [ ] a. 4000
- [ ] b. &x
- [ ] c. 0
- [ ] d. &y
- [x] e. 1000 ✓
- [x] f. 3000 ✓

Your answer is correct.

When we execute:

t1 = *q;

t1 is set to the value contained in the location pointed to by **q**.

Most of the time, we would expect **t1** to equal the value of **y**. However, in the case where **q** and **p** both point to the same location, the value that **q** points to is overwritten to be the value of **x** when the following line is executed:

*p = x;

The correct answers are: 1000, 3000

Question **3**

Correct

Mark 5.00 out of 5.00

Consider the following functions:

```
int min(int x, int y) { return x < y ? x : y; }
int max(int x, int y) { return x < y ? y : x; }
void incr(int *xp, int v) { *xp += v; }
int square(int x) { return x*x; }
```

Assume x equals 10 and y equals 100. The following code fragment calls these functions.

```
for (i = max(x, y) - 1; i >= min(x, y); incr(&i, -1))
    t += square(i);
```

The function **min** is called 91 ✓ times.

The function **max** is called 1 ✓ times.

The function **incr** is called 90 ✓ times.

The function **square** is called 90 ✓ times.

Question **4**

Partially correct

Mark 1.00 out of 6.00

Assume we have a processor with a single functional unit that handles memory operations (load/store), one functional unit that handles addition, another functional unit that handles multiply operations and yet another functional unit that handles divisions. Each functional unit can execute when the inputs to that functional unit are ready -- the performance is only limited by data dependencies, pipeline latencies and pipeline issue rates.

The following table shows the number the latency and issue cycles for each operation:

Operation	Latency	Issue
Add / Sub	3	2
Multiplication	3	1
Division	15	15
Memory load	4	2
Memory store	4	2

How many cycles would the following 64-bit Intel code sequence take? In other words, at what cycle would the each instruction complete?

Operation	Cycle Completed?
`movl 4(%rsp), %eax`	4
`addl %edi, %eax`	8
`movl 8(%rsp), %ebp`	8
`addl %eax, %esi`	11
`imull %edi, %eax`	15
`movl %edi, 8(%rsp)`	15

You can use an expression if that's useful.

Your last answer was interpreted as follows: 4

Your last answer was interpreted as follows: 8

Your last answer was interpreted as follows: 8

Your last answer was interpreted as follows: 11

Your last answer was interpreted as follows: 15

Your last answer was interpreted as follows: 15

Your answer is partially correct.

The results of the first **movl** instruction is used by the **addl** instruction and must complete prior to that **addl**.

The first **movl** instruction would finish at cycle 4.

The first **addl** instruction thus starts at 4 and finishes at 7.

The second **movl** instruction can start at 2 and would finish at 6.

The second **addl** instruction can start when the first add instruction finishes, cycle 7, because it depends on **%eax**. It finishes at cycle 10.

The **imull** instruction depends on the value of **%eax** from the first **add** instruction. Thus, the **imull** can start at 7 and finishes at 10.

The last store or **movl** instruction doesn't depend on any value produced by the preceding instructions and thus can start after the second **movl** finishes at time 6. The store will complete at time 10.

Thus, the last instruction finishes at cycle 10.

A correct answer is 4, which can be typed in as follows: `4`

A correct answer is 7, which can be typed in as follows: `7`

A correct answer is 6, which can be typed in as follows: `6`

A correct answer is 10, which can be typed in as follows: `10`

A correct answer is 10, which can be typed in as follows: `10`

A correct answer is 10, which can be typed in as follows: `10`

Question **5**

Correct

Mark 5.00 out of 5.00

A programmer is trying to improve the performance of a program and transforms the loop

```
for(i = 0; i < vec_length(v); i++) {
   *dest = *dest + data[i];
}
```

to the form

```
int l = vec_length(v);
for(i = 0; i < l; i++) {
   *dest = *dest + data[i];
}
```

Select from the list below the name of the optimization that best identifies this code transformation.

Select one:

a. Eliminating memory references

[✓ b. Reducing procedure calls] ✓

c. Accumulating in temporary

d. Loop unrolling

Your answer is correct.

This is an example of removing procedure calls in the loop body (the condition check).

The correct answer is: Reducing procedure calls

Question **6**

Incorrect

Mark 0.00 out of 5.00

A programmer attempts to improve the performance of the "func1" below:

```
void func1 (vec_ptr v, data_t *dest)
{
   long int i;
   for(i = 0; i < vec_length(v); i++) {
      data_t val;
      get_vec_element(v, i , &val);
      *dest = *dest + val
   }
```

by adding a "get_vec" such as the one below:

```
data_t *get_vec(vec_ptr v)
 {
   return v->data;
 }
```

and then rewrites "func1" using the return value of "get_vec".

What class of optimization techniques is being used in this process for performance improvement?

Select one:

a. Reducing procedure calls

[✓ b. Eliminating unneeded memory references] ✗

c. Accumulating in temporary

d. Loop unrolling

Your answer is incorrect.

The correct answer is: Reducing procedure calls

Question **7**

Correct

Mark 6.00 out of 6.00

Consider the following code:

```
for (; i < length; i++) {
   acc = acc OP data[i];
}
```

Assume the performance of the loop is measured with different operations **OP1** and **OP2** and the execution time (in CPE) for **OP1** is less than that of **OP2.** Which of the following statements are true?

Select one:

- a. It is possible to unroll the loop with OP1 but not the loop with OP2
- b. There are more functional units that can execute OP2 than OP1
- c. OP2 has a shorter pipeline than OP1
- d. There are more functional units that can execute OP1 than OP2
- [✓ e. OP1 has a shorter pipeline than OP2] ✓

Your answer is correct.

Because the loop has a single accumulator and dependence from one iteration to the next, multiple function units can not improve the performance.

The correct answer is: OP1 has a shorter pipeline than OP2

Question **8**

Correct

Mark 6.00 out of 6.00

Assume you have the following code

```
/* Accumulate in temporary */
void inner4(vec_ptr u, vec_ptr v, data_t *dest)
{
  long int i;
  int length = vec_length(u);
  data_t *udata = get_vec_start(u);
  data_t *vdata = get_vec_start(v);
  data_t sum = (data_t) 0;
  for (i = 0; i < length; i += 4) {
     sum = sum + udata[i] * vdata[i]
               + udata[i+1] * vdata[i+1]
               + udata[i+2] * vdata[i+2]
               + udata[i+3] * vdata[i+3];
  }
  for (i = 0; i < length; i++) {
     sum = sum + udata[i] * vdata[i];
  }
 *dest = sum;
}
```

which uses 4-way loop unrolling. Measurements for this function with the x86-64 architecture shows it achieves a CPE of 2.0 for integer data but a CPE of 3.0 for single and double precision floating point.

Assuming the model of the Intel i7 architecture used in class (one branch unit, two arithmetic, one load, one store unit) the performance of this loop with any arithmetic operation can not get below 2.0 CPE because of [the number of available load units] ✓ .

The program author used loop unrolling in an attempt to speed up the code, but the unrolled version was no faster than the original. Other than the limit factor you selected above, the mostly likely reason this occurs is because [the loop overhead was not the limiting factor] ✓ .

Question **9**

Correct

Mark 6.00 out of 6.00

Assume you have the following code

```
/* Accumulate in temporary */
void inner4(vec_ptr u, vec_ptr v, data_t *dest)
{
  long int i;
  int length = vec_length(u);
  data_t *udata = get_vec_start(u);
  data_t *vdata = get_vec_start(v);
  data_t sum = (data_t) 0;
  for (i=0; i < length; i++) {
     sum = sum + udata[i] * vdata[i];
  }
 *dest = sum;
}
```

and you modify the code to the form below.

```
/* Accumulate in temporary */
void inner4(vec_ptr u, vec_ptr v, data_t *dest)
{
  long int i;
  int length = vec_length(u);
  data_t *udata = get_vec_start(u);
  data_t *vdata = get_vec_start(v);
  data_t sum = (data_t) 0;
  data_t sum2 = (data_t) 0;
  for (i = 0; i < length-1; i += 2) {
     sum = sum + udata[i] * vdata[i];
     sum2 = sum2 + udata[i+1] * vdata[i+1];
  }
  for (; i < length; i++) {
     sum = sum + udata[i] * vdata[i];
  }
 *dest = sum + sum2;
}
```

What type of optimizations is being applied?

Pick all optimizations the programmer performed that play a significant role in performance.

Select one or more:

- a. Inlining
- [✓ b. Loop unrolling] ✓ Correct!
- c. Strength reduction
- d. Instruction pipelining
- [✓ e. Parallel accumulators] ✓
- f. Common subexpression elimination
- g. Machine independent optimization

Your answer is correct.

This is an example of loop unrolling and multiple accumulators.

The correct answers are: Loop unrolling, Parallel accumulators

Question **10**

Correct

Mark 6.00 out of 6.00

A programmer is trying to improve the performance of a code that uses the following data structure:

```
struct datatype {
   double f;
    int i;
    char c;
    double d;
    char cc;
} data[N];
```

The code goes through the **data** array repeatedly updating each element of the struct and the size of the array (N) is very large. For each of the following struct variants, indicate if it would have a larger cache miss rate, lower cache miss rate or the same.

```
struct datatype {
   char c, cc;
    int i;
   double f;
    double d;
}
```

lower miss rate ✓

```
struct datatype {
   char c, cc;
    double f;
    int i;
    double d;
}
```

same miss rate ✓

```
struct datatype {
   double f;
    int i;
    char c;
    char cc;
    double d;
}
```

lower miss rate ✓

Your answer is correct.

The cache miss rate is directly related to the size of the struct. The original struct takes 32 bytes of storage due to alignment constraints.

For example, the form

```
struct datatype {
   double f;
    int i;
    char c;
    char cc;
    double d;
}
```

takes 24 bytes, so the miss rate will be less. The form

```
struct datatype {
   char c, cc;
   double f;
    int i;
    double d;
}
```

takes 32 bytes and would have the same miss rate.

The correct answer is:

```
struct datatype {
   char c, cc;
    int i;
   double f;
    double d;
}
```

→ lower miss rate,

```
struct datatype {
   char c, cc;
    double f;
    int i;
    double d;
}
```

→ same miss rate,

```
struct datatype {
   double f;
    int i;
    char c;
    char cc;
    double d;
}
```

→ lower miss rate

Question **11**

Correct

Mark 8.00 out of 8.00

You are writing a new 3D game that you hope will earn you fame and fortune. You are currently working on a function to blank the screen buffer before drawing the next frame. The screen you are working with is a 640 × 480 array of pixels. The machine you are working on has a 64 KB direct-mapped cache with 4-byte lines.

The C structures you are using are as follows:

```
struct pixel {
  char r;
  char g;
  char b;
  char a;
};
struct pixel buffer[480][640];
int i, j;
```

```
char *cptr;
int *iptr;
```

And assuming the following:

- **sizeof(int) == 4** and **sizeof(char) == 1**
- **buffer** begins at memory address 0
- The cache is initially empty
- The only memory accesses are to the entries of the array **buffer**, with the variables **i, j, cptr and iptr** being stored in registers

Determine the cache performance of the following code:

```
int *iptr = (int *) buffer;
for( ; iptr < (int*) &buffer[480][640]; iptr++) {
   *iptr = 0;
}
```

The miss rate is 100 ✓ % (accurate to +/-0.5%)

In this code each loop iteration zeros the entire 4-byte structure by writing a 4-byte integer zero. Thus, although there are only 640 × 480 writes, each of these writes misses. Thus, the miss rate is 100%.

This problem, in comparison to a similar problem using a **char*** demonstrates that miss rates can be changed by performing more or fewer loads. This code has the same total number of misses as code using a **char*** to initialize the array and although the miss rate is higher, this code would probably be faster (because fewer loads are being executed).

Question **12**

Incorrect

Mark 0.00 out of 6.00

Consider the following functions:

```
int AveragePixel(int pixel[M][N][K])
{
        int i, j, k, avg = 0 ;

        for (k = 0; k < K; k++)
                for (i = 0; i < M; i++)
                        for (j = 0; j < N; j++)
                                avg += pixel[i][j][k];
        return avg/(M*N*K);
}
```

What is the stride number for the reference patterns of the above function?

Select one:

- a. M*N
- b. N
- c. 1
- d. M

[✓ e. N*K] ✗

f. K

Your answer is incorrect.

The inner loop is over **j** which is used to index the second dimension. Each change in index in the second dimension involves a stride of K array items.

The correct answer is: K

Question **13**

Partially correct

Mark 1.50 out of 6.00

Determine the number of cache sets (S), tag bits (t), set index bits (s), and block offset bits (b) for a 1024-byte cache using 64-bit memory addresses, 32-byte cache blocks and a 8-way associative design.

The cache has

- S=
  64
  sets,

  Your last answer was interpreted as follows: 64

- t=
  32
  tag bits for each block,

  Your last answer was interpreted as follows: 32

- s=
  6
  set index bits and

  Your last answer was interpreted as follows: 6

- b=
  5
  block offset bits

  Your last answer was interpreted as follows: 5

Your answer is partially correct.

Each cache block is 32 bytes and you need 5 to address the individual bits within the 32-byte cache block.

The total cache size is 1024 bytes and it follows a 8-way set associative design. Each "set" in the cache is composed of 8 blocks of 32 bytes each, or $8*32 = 256$ bytes. This means the total number of sets in the cache would be $1024/(8*32)$ or 4 total sets.

The address size is 64, but 5 bits are used to identify the byte within a cache block and 2 bits are used to identify the specific set in which the block can be found. The remaining $64-5-2 = 57$.

A correct answer is 4, which can be typed in as follows: 4

A correct answer is 57, which can be typed in as follows: 57

A correct answer is 2, which can be typed in as follows: 2

A correct answer is 5, which can be typed in as follows: 5

Question **14**

Correct

Mark 6.00 out of 6.00

Assume the following:

- . The memory is byte addressable.
- . Memory accesses are to 1-byte words (not to 4-byte words).
- . Addresses are 12 bits wide.
- . The cache is 4-way associative cache (E=4), with a 16-byte block size (B=16) and 16 sets (S=16).

The following figure shows the format of an address (one bit per box). Indicate (by labeling the diagram) the fields that would be used to determine the following:

CO - The cache block offset

CI - The cache set index

CT - The cache tag

11	10	9	8	7	6	5	4	3	2	1	0
CT	CT	CT	CT	CI	CI	CI	CI	CO	CO	CO	CO
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

A cache with this configuration could store a total of 1024 ✓ bytes of memory (ignoring the tags and valid bits).

Question **15**

Partially correct

Mark 3.20 out of 8.00

Assume the following:

- . The memory is byte addressable.
- . Memory accesses are to 1-byte words (not to 4-byte words).
- . Addresses are 10 bits wide.
- . The cache is 2-way associative cache (E=2), with a 8-byte block size (B=8) and 8 sets (S=8).
- The cache contents are as shown below

Set #	Way #0	Way #1
0:	V=1;Tag=0x4; Data = 0xb2 0x4c 0xb7 0x30 0x82 0x95 0x7d 0x7e	V=1;Tag=0xa; Data = 0x97 0x68 0xc3 0x13 0x9d 0x9f 0x23 0xb5
	V=1;Tag=0x9; Data =	V=1;Tag=0x6; Data =

1:	0x8e 0xb7 0xf2 0x21 0x58 0x14 0xba 0x72	0x65 0x99 0x24 0x8e 0x19 0x5f 0x77 0xed
2:	V=1;Tag=0x7; Data = 0x4f 0x4a 0x31 0x73 0xf4 0x95 0x11 0xb6	V=1;Tag=0x5; Data = 0xf7 0x57 0x79 0xb6 0x09 0xa2 0x7d 0xf8
3:	V=1;Tag=0x4; Data = 0x2c 0x32 0xcd 0x04 0x93 0x81 0x6a 0x8a	V=1;Tag=0xb; Data = 0xac 0xab 0x55 0x43 0xce 0xbb 0x28 0xd7
4:	V=1;Tag=0xc; Data = 0x46 0x2a 0xc3 0x7e 0x8f 0x83 0x3a 0x1b	V=0;Tag=0x8; Data = -- -- -- -- -- -- -- --
5:	V=1;Tag=0xd; Data = 0x31 0x40 0x2f 0xae 0x49 0xf5 0x27 0x1b	V=1;Tag=0xa; Data = 0x95 0x90 0xb5 0xd8 0x4f 0xca 0x51 0x92
6:	V=1;Tag=0x1; Data = 0x63 0x04 0xab 0x2b 0x0e 0x49 0x41 0x3e	V=1;Tag=0x2; Data = 0x3e 0x29 0x63 0xf4 0xf6 0xd7 0xbe 0x11
7:	V=1;Tag=0xe; Data = 0xd9 0x61 0xcd 0x92 0xbc 0xd3 0x26 0x03	V=1;Tag=0x2; Data = 0xc0 0xac 0x89 0x9b 0xba 0xa8 0xc8 0xe9

Assume that memory address **0x304** has been referenced by a load instruction. Indicate the cache entry accessed and the cache byte value returned **in hex** . Indicate whether a cache miss occurs. If there is a cache miss, enter "-" for the "Cache Byte Returned". For values that need a hexidecimal value, do not enter leading zeros even if leading zeros are shown in the value above.

Cache block Offset (CO)	0x 0	✗
Cache set index (CI)	0x 3	✗
Cache tag (CT)	0x 18	✗
Cache hit (Y/N)?	no	✓
Cache byte returned	0x -	✓

Question **16**

Correct

Mark 5.00 out of 5.00

At a high level, you can model the time to access a hard disk drive as the "access time" and the "transfer time" and a solid state disk can be modeled in a similar way.

Assume that a specific hard disk drive has an average access time of 16ms (*i.e.* the seek and rotational delay sums to 16ms) and a throughput or transfer rate of 200MBytes/s, where a megabyte is measured as 1024^2.

A solid-state drive (SSD) has an average access time of 0.032ms and a throughput of 430MB/s -- the access time serves the same role as the combined "seek" and "rotational" delay of a disk and represents the time to talk to the SSD and the overhead time for the non-volatile memory to be accessed.

Big Reads

> Some applications, such as playing back a movie, read large files -- they do a single "seek" or access to the beginning of the file and then read a large collection of blocks. Assume that a movie playing application does a single "access" and then reads a 115MB file.

How many "movies per second" can be processed by the hard disk drive?

1/((16/1000)+(115/200))

Your last answer was interpreted as follows: $\frac{1}{\frac{16}{1000}+\frac{115}{200}}$

Correct answer, well done.
Exactly!

How many "movies per second" can be processed by the SSD disk drive?

1/(0.000032+ 115/430)

Your last answer was interpreted as follows: $\frac{1}{3.2e{-}5+\frac{115}{430}}$

Correct answer, well done.
Close enough!

Each answer should be accurate to within 5% "movies per second" and you can use algebraic expressions if you like.

Random I/O

Many other applications are limited by "IOPS", or the "random I/O operations per second". An example of this would be a database that needs to seek to a part of the disk and read a small amount of data of 512 bytes repeatedly.

How many "IOPS" can be processed by the hard disk drive?

1/((16/1000)+((512/1024)

Your last answer was interpreted as follows: $\frac{1}{\frac{16}{1000}+\frac{\frac{512}{1024}}{\frac{1024}{200}}}$

Correct answer, well done.
Close enough!

How many "IOPS" can be processed by the SSD disk drive?

1/(0.000032+ ((512/1024)

Your last answer was interpreted as follows: $\frac{1}{3.2e{-}5+\frac{\frac{512}{1024}}{\frac{1024}{430}}}$

Correct answer, well done.
Close enough!

Each answer should be accurate to within one "IOPS" and you can use algebraic expressions if you like.

The time to seek to and read a 115MB file is $T_{\text{fs}} = T_{\text{seek}} + (115/\text{throughput})$. The speed in files per second is $\frac{1}{T_{\text{fs}}}$.

For the disk, this is $\frac{1}{\frac{16}{1000}+\frac{115}{200}} = \frac{1000}{591} = 1.69204737733$.

For the ssd, this is $\frac{[illegible]}{\frac{4}{\frac{125}{1000}+\frac{115}{430}}} = \frac{[illegible]}{179709} = 3.73868309322.$

There's a relatively small (3-10x) difference in the number of movies-per-second that can be served, related to the difference in throughput.

The number of IOPS is $1/T_{\text{seek}}$.

For the disk this is $\frac{1io}{16ms * \frac{1s}{1000ms}} = \frac{125}{2} = 62.5.$

For the ssd this is $\frac{1io}{0.032ms * \frac{1s}{1000ms}} = 31250 = 31250.0.$

There's a relatively large (~500x) difference in the number of movies-per-second that can be served, related to the difference in latency.

A correct answer is $\frac{1000}{591}$, which can be typed in as follows: `1000/591`

A correct answer is $\frac{671875}{179709}$, which can be typed in as follows: `671875/179709`

A correct answer is $\frac{125}{2}$, which can be typed in as follows: `125/2`

A correct answer is 31250, which can be typed in as follows: `31250`

Question **17**

Correct

Mark 6.00 out of 6.00

In linear algebra, the transpose of a matrix is an operator which flips a matrix over its diagonal elements. Transposing the rows and columns of a matrix is an important problem in signal processing and scientific computing applications. It is also interesting from a locality point of view because its reference pattern is both row-wise and column-wise.

For example, consider the following transpose routine:

```
typedef int array[2][2];

void transpose1(array dst, array src)
{
  int i, j;
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 2; j++) {
      dst[j][i] = src[i][j];
    }
  }
}
```

Assume this code runs on a machine with the following properties:

- sizeof(int) == 4.
- The src array starts at address 0 and the dst array starts at address 16 (decimal).
- There is a single L1 data cache that is direct-mapped, write-through, and write-allocate, with a block size of 8 bytes.
- The cache has a total size of 32 data bytes and the cache is initially empty.
- Accesses to the src and dst arrays are the only sources of read and write misses, respectively.

For each row and col, indicate whether the access to **src[row][col]** and **dst[row][col]** is a hit or a miss (m). For example, reading **src[0][0]** is a miss and writing **dst[0][0]** is also a miss.

dst array

	Col 0	Col 1
Row 0	Miss	Hit ✓

src array

	Col 0	Col 1
Row 0	Miss	Hit ✓

Row 1	Miss ✓	Hit ✓	Row 1	Miss ✓	Hit ✓