CSPB 2400 - Park - Computer Systems

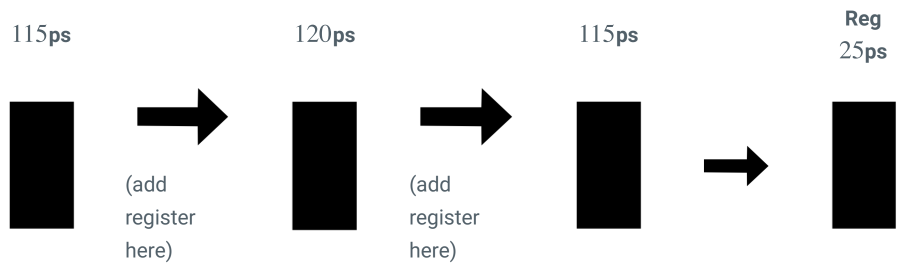
Started on	Wednesday, 10 April 2024, 12:36 PM
State	Finished
Completed on	Wednesday, 10 April 2024, 1:39 PM
Time taken	1 hour 3 mins
Marks	63.50/94.00
Grade	**6.76** out of 10.00 (**68**%)

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 2.67 GHz.



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

6.896551724138

You can use an expression if you like.

Your last answer was interpreted as follows: 6.896551724138

Correct answer, well done.
The cycle time of the fully pipelined design would depend on the largest stage delay (120) and one register delay. This results in a total cycle time of 145 with a resulting clock speed of 6.89655172414 GHz.

A correct answer is $\frac{200}{29}$, which can be typed in as follows: `200/29`

Question **2**

Partially correct

Mark 2.50 out of 5.00

Consider the following code sequence:

```
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. 1000 ✓
- ☑ c. &x ✗
- ☑ d. 3000 ✓
- ☐ e. 0
- ☑ f. &y ✗

Your answer is partially correct.

You have selected too many options.

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**

Complete

Not graded

Assume we have a processor with a single functional unit that handles memory operations (load/store), one functional unit that handles addition, one functional unit that handles multiply operations and one functional unit that handles divisions. Each functional unit can execute when the inputs to that functional unit are ready -- the performance is only limited data dependences, 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	2	2
Multiplication	2	1
Division	10	10
Memory load	2	1
Memory store	2	1

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

Operation	**Cycle Completed?**
`movl 4(%rsp), %eax`	1
`movl 8(%rsp), %edi`	2
`addl %eax, %edi`	4
`imull %edi, %eax`	5
`movl %edi, 8(%rsp)`	6

You can use an expression if that's useful.

Your last answer was interpreted as follows: 1

Your last answer was interpreted as follows: 2

Your last answer was interpreted as follows: 4

Your last answer was interpreted as follows: 5

Your last answer was interpreted as follows: 6

Incorrect answer.
Incorrect answer.

Incorrect answer.

Incorrect answer.

Incorrect answer.

Incorrect answer.

The results of the two **movl** instructions are needed by the **addl** instruction and must complete prior to **addl**.

The first **movl** instruction would finish at cycle 2. The second **movl** instruction could start 1 cycles later and would finish at cycle 3. Both **movl** instructions are finished by cycle 3.

The **addl** instruction thus starts at 3. The result of the **addl** is needed by the **imull** instruction and would be available at cycle 5.

The **imull** and last **movl** instruction can execute in parallel. The **imull** finishes at 7 and the **movl** finishes at 7.

Thus, the last instruction finishes at cycle 7.

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

A correct answer is 3, which can be typed in as follows: 3

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

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

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

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. Reducing procedure calls] ✓

b. Loop unrolling

c. Accumulating in temporary

d. Eliminating memory references

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**

Partially correct

Mark 2.50 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;
 }
```

Which optimization techniques have been used in by rewriting "func1" as "func3"?

```
void func3 (vec_ptr v, data_t *dest)
{
   long int i;
   long int l=vec_length(v)
   data_t *data = get_vec(v)
   for(i = 0; i < l; i++) {
      *dest = *dest + data[i];
   }
```

Please, choose all that apply. *You are penalized for incorrect choices*, so don't guess.

Select one or more:

- ☐ a. Accumulating in temporary
- [✓ b. Reducing procedure calls] ✓
- ☐ c. Re-association transformation
- ☐ d. Loop unrolling
- ☐ e. Eliminating memory references

Your answer is partially correct.

You have correctly selected 1.
The correct answers are: Eliminating memory references, 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. OP1 has a shorter pipeline than OP2] ✓
- c. There are more functional units that can execute OP1 than OP2
- d. There are more functional units that can execute OP2 than OP1
- e. OP2 has a shorter pipeline than OP1

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**

Incorrect

Mark 0.00 out of 5.00

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 registers] ✖ .

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 [of the number of available load units] ✖ .

Question **9**

Partially correct

Mark 2.50 out of 5.00

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

Your answer is partially correct.

You have correctly selected 1.
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;
    double f;
    int i;
    double d;
}
```

same miss rate ✓

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

lower 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;
    double f;
    int i;
    double d;
}
```

→ same miss rate,

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

→ lower miss rate,

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

→ lower miss rate

Question **11**

Incorrect

Mark 0.00 out of 8.00

You need to find the performance of some code on a machine with a 2048-byte direct-mapped data cache with 32-byte blocks.

Given the following definitions:

```
struct point_color {
  int c;
  int m;
  int y;
  int k;
};
struct point_color square[16][16];
int i,j;
```

And assuming the following:

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

Determine the cache performance of the following code:

```
for (i = 0; i < 16; i++) {
   for (j = 0; j < 16; j++) {
     square[j][i].c = 0;
     square[j][i].m = 0;
     square[j][i].y = 1;
     square[j][i].k = 0;
  }
}
```

1. There are are a total of 256 ✖ writes
2. A total of 64 ✖ writes miss in the cache.
3. The miss rate is 25 ✖ % (accurate to +/-0.5%)

Since the cache can 64 lines and each line holds two elements, The first iteration through the inner loop loads 16 pairs of elements, with 1 hit and 3 misses each. The second iteration through the inner loop with i = 1 has 4 hits each because those elements were loaded in the first iteration, with i = 0. So there are 7 hits and 1 miss for each 8 reads.

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 (j = 0; j < N; j++)
                for (i = 0; i < M; i++)
                        for (k = 0; k < K; k++)
                                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. 1
- [✓ b. K] ✗
- c. N
- d. M

Your answer is incorrect.

The correct answer is: 1

Question **13**

Correct

Mark 6.00 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 32-way set associative cache using 32-bit memory addresses and 32-byte cache blocks.

This cache has S= 1 ✓ sets, t= 27 ✓ tag bits, s= 0 ✓ set index bits and b= 5 ✓ block offset bits.

Question **14**

Correct

Mark 8.00 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 12 bits wide.
- . The cache is 8-way associative cache (E=8), with a 8-byte block size (B=8) 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	CT	CI	CI	CI	CI	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**

Correct

Mark 8.00 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 9 bits wide.
- . The cache is 2-way associative cache (E=2), with a 4-byte block size (B=4) and 2 sets (S=2).
- The cache contents are as shown below

Set #	**Way #0**	**Way #1**
0:	V=1;Tag=0x0f; Data = 0x42 0x8f 0x3b 0x61	V=1;Tag=0x22; Data = 0x67 0x6d 0x84 0x97
1:	V=1;Tag=0x0c; Data = 0xbb 0xc7 0x40 0x82	V=1;Tag=0x27; Data = 0x35 0xe8 0x71 0xb6

Assume that memory address **0x78** 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 0	✓
Cache tag (CT)	0x f	✓
Cache hit (Y/N)?	yes	✓
Cache byte returned	0x 42	✓

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 8ms (*i.e.* the seek and rotational delay sums to 8ms) and a throughput or transfer rate of 140MBytes/s, where a megabyte is measured as 1024^2.

A solid-state drive (SSD) has an average access time of 0.027ms and a throughput of 470MB/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 110MB file.

How many "movies per second" can be processed by the hard disk drive? 1 / (8E-3 + (110 / 140))

Your last answer was interpreted as follows: $\frac{1}{0.008+\frac{110}{140}}$

Correct answer, well done.
Exactly!

How many "movies per second" can be processed by the SSD disk drive? 1 / (0.027E-3 + (110 / 470

Your last answer was interpreted as follows: $\frac{1}{2.7e-5+\frac{110}{470}}$

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 / (8E-3 + (512 / (140 * 1

Your last answer was interpreted as follows: $\frac{1}{0.008+\frac{512}{140\cdot1024^2}}$

Correct answer, well done.
Close enough!

How many "IOPS" can be processed by the SSD disk drive? 1 / (0.027E-3 + (512 / (47

Your last answer was interpreted as follows: $\frac{1}{2.7e-5+\frac{512}{470\cdot1024^2}}$

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 110MB file is $T_{\text{fs}} = T_{\text{seek}} + (110/\text{throughput})$. The speed in files per second is $\frac{1}{T_{\text{fs}}}$.

For the disk, this is $\frac{1}{\frac{8}{1000}+\frac{110}{140}} = \frac{1750}{1389} = 1.25989920806.$

For the ssd, this is $\frac{1}{\frac{\frac{27}{1000}}{1000}+\frac{110}{470}} = \frac{47000000}{11001269} = 4.27223441223.$

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 $\frac{1io}{8ms*\frac{1s}{1000ms}} = 125 = 125.0.$

For the ssd this is $\frac{1io}{0.027ms*\frac{1s}{1000ms}} = \frac{1000000}{27} = 37037.037037.$

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 $\frac{1750}{1389}$, which can be typed in as follows: `1750/1389`

A correct answer is $\frac{47000000}{11001269}$, which can be typed in as follows: `47000000/11001269`

A correct answer is 125, which can be typed in as follows: `125`

A correct answer is $\frac{1000000}{27}$, which can be typed in as follows: `1000000/27`

Question **17**

Partially correct

Mark 2.00 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 ✓
Row 1	Hit ✗	Miss ✗

src array

	Col 0	Col 1
Row 0	Miss	Hit ✓
Row 1	Hit ✗	Miss ✗