**Full Name:**_____

# EECS 213 Fall 2010
# Final Exam

## 1. (10 points):

Given a 32-bit address machine with a 2048 byte cache, fill in for each block size (B) and lines per set (E) as specified below the number of cache sets (S), tag bits (t), set index bits (s), and block offset bits (b). Give answers in decimal.

| B | E | S | t | s | b |
|---|---|---|---|---|---|
| 8 | 4 | *64* | *23* | *6* | *3* |
| 8 | 256 | *1* | *29* | *0* | *3* |
| 16 | 1 | *128* | *21* | *7* | *4* |
| 16 | 128 | *1* | *28* | *0* | *4* |
| 32 | 1 | *64* | *21* | *6* | *5* |
| 32 | 4 | *16* | *23* | *4* | *5* |

## 2. (10 points):

Describe the sequence of events in this program and its output. Assume the necessary includes are done and all functions return normally.

```
pid_t pid, cid;

void handler1(int sig) {
  printf("zip");
  fflush(stdout); /* Flushes the printed string to stdout */
  kill(cid, SIGUSR1);
}

void handler2(int sig) {
  printf("zap");
  exit(0);
}

main() {
  signal(SIGUSR1, handler1);
  pid = getpid();
  if ((cid = fork()) == 0) {
    signal(SIGUSR1, handler2);
    kill(pid, SIGUSR1);
    while(1) {};
  }
  else {
    int status;
    if (wait(&status) > 0) {
      printf("zoom");
    }
  }
}
```

**Comment [CKR1]:** Common mistake: saying this sends a signal. It does not. It just registers a handler for a signal.

**Comment [CKR2]:** Common mistake: saying this kills a process. kill() sends a signal. The signal may or may not be SIGTERM.

*Output is "zipzapzoom"*

*The parent sets its handler for SIGUSR1 to handler1, sets pid to its own id, forks, sets cid to the child id, and waits for the child to exit.*
*The child sets its handler for SIGUSR1 to handler2, sets cid to 0, sends SIGUSR1 to the parent, and begins an endless loop.*
*handler1 in parent prints "zip" and sends SIGUSR1 to child.*
*handler2 in child prints "zap" and exits child.*
*The wait in parent returns the child id, parent prints "zoom" and exits.*

**Comment [CKR3]:** The major mistake some people made was just marking what different parts of the code did. This is way too vague and unclear about what happens in which process and when.

## 3. (12 points):

Consider an allocator that uses an implicit free list, where the layout of each memory block is

- 32 bit header, with size and use bits
- the payload
- 32 bit footer, with size and use bits

The size of each memory block is a multiple of eight bytes, including the header and footer. The low-order bit 0 of the size is set to 1 if the block is allocated, 0 if free.

Implement the code as indicated by the comments. Make use of previous results and the function `size()` when possible.

**C note:** pointer arithmetic is valid only on pointers to actual types, not `void *`.

```
void *p = malloc(20);

void *hp = ((int *) p) - 1;

void *fp = ((char *) p) + size(hp) - 8;

int used = (* (int *) hp) & 1;

void *pp = ((char *) hp) - size(((int *) hp) - 1);

/* return the  size in the header pointed to by hp */

int size(void *hp) { return (*(int *) hp) & (~7);   }
```

**Comment [CKR4]:** must cast to get a value or do pointer arithmetic. Casting back to void * is automatic.

Way too many people tried to do arithmetic or dereferencing with void pointers. You can't do anything with a void pointer except pass it around and cast it to something else.

**Comment [CKR5]:** note use of char when dealing with bytes and int when dealing with whole words

**Comment [CKR6]:** 4 if you add size to hp

**Comment [CKR7]:** just 1 also valid though that was an oversight on my part.

**Comment [CKR8]:** shortest way to specify all 1's except last 3 bits; also independent of word size

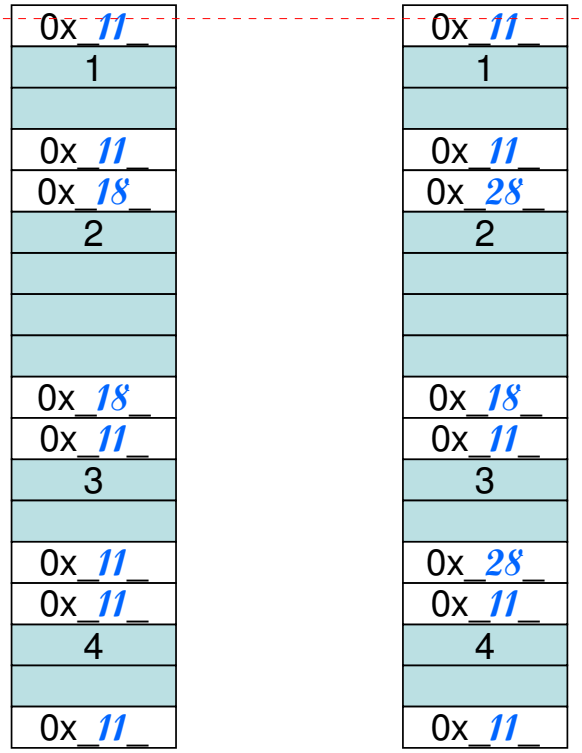
Common mistakes: dividing by 2 or using modulo or masking by 1, all of which return something other than the size.

Another mistake was using a mask like 0xFFFE which would zero out higher order bits of a large block.

~1 got credit but is less robust than ~7.

## 4. (10 points)

Assume the heap structure of the previous problem. In the diagrams below, each rectangle is a 32-bit word. On the left, block 2 is free, the rest are in use. Fill in the header values in hex. On the right, show the heap after block 3 is freed, and coalescing is done.




**Comment [CKR9]:** Most common mistakes were putting either the number of words or the number of bits. Address arithmetic internally is always in bytes.

When numbers were wrong, I gave credit if headers = footers, used blocks were odd and free blocks were even, the right headers and footers changed, and the size in the new header and footer was the sum of the sizes in merged blocks.

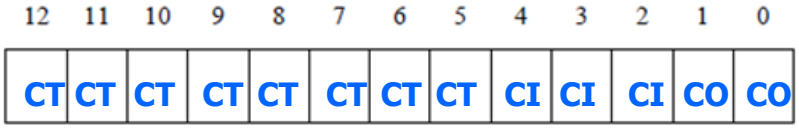
## 5. (10 points)

Given:

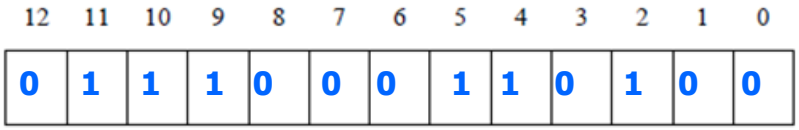
- Memory accesses are to **1-byte words** (not 4-byte words).
- Physical addresses are 13 bits wide.
- The cache below:

| 2-way Set Associative Cache | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Index | Tag | Valid | Byte 0 | Byte 1 | Byte 2 | Byte 3 | Tag | Valid | Byte 0 | Byte 1 | Byte 2 | Byte 3 |
| 0 | 09 | 1 | 86 | 30 | 3F | 10 | 00 | 0 | 99 | 04 | 03 | 48 |
| 1 | 45 | 1 | 60 | 4F | E0 | 23 | 38 | 1 | 00 | BC | 0B | 37 |
| 2 | EB | 0 | 2F | 81 | FD | 09 | 0B | 0 | 8F | E2 | 05 | BD |
| 3 | 06 | 0 | 3D | 94 | 9B | F7 | 32 | 1 | 12 | 08 | 7B | AD |
| 4 | C7 | 1 | 06 | 78 | 07 | C5 | 05 | 1 | 40 | 67 | C2 | 3B |
| 5 | 71 | 1 | 0B | DE | 18 | 4B | 6E | 0 | B0 | 39 | D3 | F7 |
| 6 | 91 | 1 | A0 | B7 | 26 | 2D | F0 | 0 | 0C | 71 | 40 | 10 |
| 7 | 46 | 0 | B1 | 0A | 32 | 0F | DE | 1 | 12 | C0 | 88 | 37 |

Clearly label the address diagram below with the fields for the cache offset (**CO**),cache index (**CI**) and cache tag (**CT**).




Fill in the bits for this address. : 0x**0E34**




For this address, fill in the table below. Use hex where indicated. Indicate whether a cache miss occurs. If so, enter "-" for "Byte returned".

| Parameter | Value |
|---|---|
| Cache Offset | 0x___*C*___ |
| Cache Index | 0x___*5*___ |
| Cache Tag | 0x___*71*__ |
| Cache Hit? (Y/N) | *Yes* |
| Byte returned | 0x___*CB*__ |

## Problem 6. (10 points):

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 640x480 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:

```
struct pixel {
   char r;
   char g;
   char b;
   char a;
};

struct pixel buffer[480][640];

register int i, j;
register char *cptr;
register int *iptr;
```

Assume: sizeof(char) = 1, sizeof(int) = 4, buffer begins at memory address 0, the cache is initially empty. Variables i, j, cptr, and iptr are stored in registers, so the only memory accesses are to the array buffer.

A. What percentage of the writes in the following code will miss in the cache?

```
for (j=0; j < 640; j++) {
      for (i=0; i < 480; i++){
            buffer[i][j].r = 0;
            buffer[i][j].g = 0;
            buffer[i][j].b = 0;
            buffer[i][j].a = 0;
      }
}
```

Miss rate for writes to buffer: __25__ %

**Comment [CKR10]:** because each initial write causes 4 bytes to be read, so the next 3 are in-cache.

B. What percentage of the writes in the following code will miss in the cache?

```
char *cptr;
cptr = (char *) buffer;
for (; cptr < (((char *) buffer) + 640 * 480 * 4); cptr++)
*cptr = 0;
```

Miss rate for writes to buffer: ___25__ %

**Comment [CKR11]:** the same thing happens

C. What percentage of the writes in the following code will miss in the cache?

```
int *iptr;
iptr = (int *) buffer;
for (; iptr < (buffer + 640 * 480); iptr++)
*iptr = 0;
```

Miss rate for writes to buffer: _100__ %

**Comment [CKR12]:** each write needs 4 new bytes

D. Which code (A, B, or C) should be the fastest? __C___

**Comment [CKR13]:** 4 bytes are stored at once; the number of cache misses is the same; you can't compare rates across different storage sizes.