Full Name:____

COMP2310/COMP6310, 2022

Final Exam

Semester 2, 2023

Instructions:

- The exam has a maximum score of 49 points.
- The problems vary in difficulty scale. The point value of each problem is indicated. Advice: Pile up the easy points quickly and then come back to the harder problems.
- Good luck!

1 (08):	
2 (08):	
3 (08):	
4 (05):	
5 (08):	
6 (12):	
TOTAL (49):	

Problem 1. (8 points):

This problem tests your understanding of how `for` loops in C relate to IA32 machine code
Consider the following IA32 assembly code for a procedure `foo()`:

```
foo:
        pushl %ebp
        movl %esp,%ebp
        movl 12(%ebp),%ecx
        xorl %eax,%eax
        movl 8(%ebp),%edx
        cmpl %ecx,%edx
        jle .L3
        .align 4
.L5:
        addl %edx,%eax
        decl %edx
        cmpl %ecx,%edx
        jg .L5
.L3:
        leave
        ret
```

Based on the assembly code above, fill in the blanks below in its corresponding C source code. (Note: you may only use symbolic variables x, y, i, and $result$, from the source code in your expressions below — do *not* use register names.)

Answer:

```
int foo(int x, int y)
{
  int i, result=0;
  for (i=x; i>y; i--) {
    result+=i;
  }
  return result;
}
```

Problem 2. (8 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.
- The only memory accesses are to the entries of the array `buffer`. Variables `i`, `j`, `cptr`, and `iptr` are stored in registers.

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%**

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%**

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%**

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

Problem 3. (8 points):

This problem tests your understanding of conflict misses. Consider the following transpose routine

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

void transpose(array dst, array src) {
  int i, j;

  for (i = 0; i < 2; i++) {
    for (j = 0; j < 2; j++) {
      dst[i][j] = src[j][i];
    }
  }
}
```

running on a hypothetical 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 cache that is direct mapped and write-allocate, with a block size of 8 bytes.
- Accesses to the `src` and `dst` arrays are the only sources of read and write misses, respectively.

A. Suppose the cache has a total size of 16 data bytes (i.e., the block size times the number of sets is 16 bytes) and that the cache is initially empty. Then for each `row` and `col`, indicate whether each access to `src[row][col]` and `dst[row][col]` is a hit (h) 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	m	**h**
row 1	**m**	**m**

`src` array		
	col 0	col 1
row 0	m	**m**
row 1	**m**	**m**

B. Repeat part A for a cache with a total size of 32 data bytes.

`dst` array		
	col 0	col 1
row 0	m	**h**
row 1	**m**	**h**

`src` array		
	col 0	col 1
row 0	m	**h**
row 1	**m**	**h**

Problem 4. (5 points):

Consider the following C program:

```
#include <sys/wait.h>

main() {
  int status;

  printf("%s\n", "Hello");
  printf("%d\n", !fork());

  if(wait(&status) != -1)
    printf("%d\n", WEXITSTATUS(status));

  printf("%s\n", "Bye");

  exit(2);
}
```

Recall the following:

- Function `fork` returns 0 to the child process and the child's process Id to the parent.
- Function `wait` returns `-1` when there is an error, e.g., when the executing process has no child.
- Macro `WEXITSTATUS` extracts the exit status of the terminating process.

What is a valid output of this program? *Hint: there are several correct solutions.*
One possible solution:

```
Hello
1
Bye
0
2
Bye
```

Process control

The next problem concerns the following four versions of the `tfgets` routine, a timeout version of the Unix `fgets` routine.

The `tfgets` routine waits for the user to type in a string and hit the return key. If the user enters the string within 5 seconds, the `tfgets` returns normally with a pointer to the string. Otherwise, the routine "times out" and returns a NULL string.

`tfgets`: Version A

```
void handler(int sig) {
  siglongjmp(env, 1);
}

char *tfgets(char *s, int size, FILE *stream) {
  pid_t pid;
  signal(SIGCHLD, handler);

  if (!sigsetjmp(env, 1)) {
    pid = fork();
    if (pid == 0) {
      return fgets(s, size, stream);
    }
    else {
      sleep(5);
      kill(pid, SIGKILL);
      wait(NULL);
      return NULL;
    }
  }
  else {
    wait(NULL);
    exit(0);
  }
}
```

`tfgets`: **Version B**

```
void handler(int sig) {
  wait(NULL);
  siglongjmp(env,1);
}

char *tfgets(char *s, int size, FILE *stream) {
  pid_t pid;

  signal(SIGUSR2, handler);
  if (sigsetjmp(env, 1) != 0)
    return NULL;
  if ((pid = fork()) == 0) {
    sleep(5);
    kill(getppid(), SIGUSR2);
    exit(0);
  }
  fgets(s, size, stream);
  kill(pid, SIGKILL);
  wait(NULL);
  return s;
}
```

`tfgets`: **Version C**

```
void handler(int sig) {
  wait(NULL);
  siglongjmp(env, 1);
}

char *
tfgets(char *s, int size, FILE *stream) {
  pid_t pid;
  str = NULL;
  signal(SIGCHLD, handler);

  if ((pid = fork()) ==  0) {
    sleep(5);
    exit(0);
  }
  else {
    if (sigsetjmp(env, 1) == 0) {
      str = fgets(s, size, stream);
      kill(pid, SIGKILL);
      pause();
    }
    return str;
  }
}
```

tfgets: Version D

```
void handler(int sig) {
  wait(NULL);
  siglongjmp(env, 1);
}

char *
tfgets(char *s, int size, FILE *stream) {
  pid_t pid;
  str = NULL;
  signal(SIGCHLD, handler);

  if ((pid = fork()) ==  0) {
    sleep(5);
    return NULL;
  }
  else {
    if (sigsetjmp(env, 1) == 0) {
      str = fgets(s, size, stream);
      kill(pid, SIGKILL);
      pause();
    }
    return str;
  }
}
```

Problem 5. (8 points):

This problem concerns the four versions of `tfgets` from the previous pages. Some of them are correct, and others are flawed because the author didn't understand basic concepts of concurrency and signaling. Circle the versions that are correct, in the sense that they return the input string if typed within 5 seconds, timeout after 5 seconds by returning NULL, and correctly reap their terminated children.

```
Version A          Version B          Version C          Version D
```

Note: The `pause` function sleeps until a signal is received and then returns.

Version B

Version C

A gets the parent and child confused.

D returns from the child instead of exiting.

Problem 6. (12 points):

The following problem concerns the way virtual addresses are translated into physical addresses.

- The memory is byte addressable.
- Memory accesses are to **1-byte words** (not 4-byte words).
- Virtual addresses are 16 bits wide.
- Physical addresses are 13 bits wide.
- The page size is 512 bytes.
- The TLB is 8-way set associative with 16 total entries.
- The cache is 2-way set associative, with a 4 byte line size and 16 total lines.

In the following tables, **all numbers are given in hexadecimal**. The contents of the TLB, the page table for the first 32 pages, and the cache are as follows:

TLB			
Index	Tag	PPN	Valid
0	09	4	1
	12	2	1
	10	0	1
	08	5	1
	05	7	1
	13	1	0
	10	3	0
	18	3	0
1	04	1	0
	0C	1	0
	12	0	0
	08	1	0
	06	7	0
	03	1	0
	07	5	0
	02	2	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	6	1	10	0	1
01	5	0	11	5	0
02	3	1	12	2	1
03	4	1	13	4	0
04	2	0	14	6	0
05	7	1	15	2	0
06	1	0	16	4	0
07	3	0	17	6	0
08	5	1	18	1	1
09	4	0	19	2	0
0A	3	0	1A	5	0
0B	2	0	1B	7	0
0C	5	0	1C	6	0
0D	6	0	1D	2	0
0E	1	1	1E	3	0
0F	0	0	1F	1	0

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	19	1	99	11	23	11	00	0	99	11	23	11
1	15	0	4F	22	EC	11	2F	1	55	59	0B	41
2	1B	1	00	02	04	08	0B	1	01	03	05	07
3	06	0	84	06	B2	9C	12	0	84	06	B2	9C
4	07	0	43	6D	8F	09	05	0	43	6D	8F	09
5	0D	1	36	32	00	78	1E	1	A1	B2	C4	DE
6	11	0	A2	37	68	31	00	1	BB	77	33	00
7	16	1	11	C2	11	33	1E	1	00	C0	0F	00

Part 1

A. The box below shows the format of a virtual address. Indicate (by labeling the diagram) the fields (if they exist) that would be used to determine the following: (If a field doesn't exist, don't draw it on the diagram.)

VPO	The virtual page offset
VPN	The virtual page number
TLBI	The TLB index
TLBT	The TLB tag

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

VPN: [15-9] VPO: [8-0]

TLBT: [15-10] TLBI: [9]

B. The box below shows the format of a physical address. Indicate (by labeling the diagram) the fields that would be used to determine the following:

PPO	The physical page offset
PPN	The physical page number
CO	The block offset within the cache line
CI	The cache index
CT	The cache tag

12	11	10	9	8	7	6	5	4	3	2	1	0

PPN: [12-9] PPO: [8-0]

CT: [12-5] CI: [4-2] CO: [1-0]

Part 2

For the given virtual address, indicate the TLB entry accessed, the physical address, and the cache byte value returned **in hex**. Indicate whether the TLB misses, whether a page fault occurs, and whether a cache miss occurs.

If there is a cache miss, enter "-" for "Cache Byte returned". If there is a page fault, enter "-" for "PPN" and leave parts C and D blank.

Virtual address: `1DDE`

A. Virtual address format (one bit per box)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

0001 1101 1101 1110

B. Address translation

Parameter	Value
VPN	0x**0E**
TLB Index	0x**0**
TLB Tag	0x**07**
TLB Hit? (Y/N)	**N**
Page Fault? (Y/N)	**N**
PPN	0x**1**

C. Physical address format (one bit per box)

12	11	10	9	8	7	6	5	4	3	2	1	0

0 0011 1101 1110

D. Physical memory reference

Parameter	Value
Byte offset	0x**2**
Cache Index	0x**7**
Cache Tag	0x**1E**
Cache Hit? (Y/N)	**Y**
Cache Byte returned	0x**F**