1. [2] Consider the C program below. Assume that all functions return normally

```c
main() {
    if (fork() == 0) {
        if (fork() == 0) {
            printf("3");
        } else {
            pid_t pid; int status;
            if ((pid = wait(&status)) > 0) {
                printf("4");
            }
        }
    } else {
        if (fork() == 0) {
            printf("1");
            exit(0);
        }
        printf("2");
    }
    printf("0");
    return 0;
}
```

Out of the 5 outputs listed below, circle only the valid outputs (can be one or more or none) of this program. Assume that all processes run to normal completion.

A. 2030401  B. 1234000  C. 2300140  D. 2034012  E. 3200410

**Solution:** By the end, there are 4 different processes. Each of them prints one of 1, 2, 3, and 4. We will refer to the 4 processes by these numbers. Note that process 1 exits immediately after printing 1, but the others do not exit after printing their number, meaning that each of the other 3 processes will also execute the printf("0") at the end. So, we know that the digits 1234000 will be printed in some order. There are a few constraints. Each of the digits 2,3,4 must be paired with a later 0. Also, process 4 waits for process 3 to finish before printing 4, so a 3 and its corresponding 0 must come before 4. Other than that, the digits may appear in any order. A, C, and E all follow these constraints and all are valid outputs of the program. B is not because there must be a 3 and a 0 before the 4. D is not because there are not enough 0's.
2. Consider the C program below. Assume that all functions return normally.

```c
int main () {
    if (fork() == 0) {
        if (fork() == 0) {
            printf("3");
        } else {
            pid_t pid; int status;
            if ((pid = wait(&status)) > 0) {
                printf("4");
            }
        }
    } else {
        printf("2");
        exit(0);
    }
    printf("0");
    return 0;
}
```

What are the possible output(s) of that program?

32040  23040  30240  30420  30402

Because of the wait(), the child (printing 40) must wait for the grandchild (printing 30). The parent (printing 2) can be interleaved anywhere in the sequence of 3040.
3. Consider a computer system that has a cache with 256 blocks. Each block can store 16 bytes. What will be the value stored in the TAG field of the cache block that holds the memory block containing the address 0x3CFBCF (Note: The address provided tells you that the length of an address here is 24 bits)

The address (24 bits in length) will be divided as TAG SET OFFSET

block size = 16 bytes \(\rightarrow\) \(2^4\) bytes \(\rightarrow\) we need 4 bits for the offset field.

(i) [3] if it is a direct-mapped cache

Direct-mapped cache means each set has only one block. We have a total of 256 blocks \(\rightarrow\) we have 256 sets \(\rightarrow\) \(2^8\) sets \(\rightarrow\) we need 8 bits for the set field.
This means the tag will be \(24 - (8 + 4)\) = 12 bits.
From the address given above, the TAG is: 0x3CF

(ii) [3] if it is a 16-way set-associative cache

Here the set contains 16 blocks \(\rightarrow\) number of sets is \(256/16 = 2^8/2^4 = 2^4\) sets \(\rightarrow\) we need 4 bits for the set field.
This means the tag field needs \(24 - (4 + 4)\) = 16 \(\rightarrow\) from the address specified in the problem the tag then is: 0x3CFB

(iii) [3] if it is fully associative

A fully associative cache has only one set \(\rightarrow\) we do not need any bits for the set field \(\rightarrow\) the tag will be: 0x3CFBC
4. [1] What is the output of this program?

```c
int val = 10;

void handler(sig)
{
    val += 7;
    return;
}

int main()
{
    int pid;
    signal(SIGCHLD, handler);
    if ((pid = fork()) == 0) {
        val -= 3;
        exit(0);
    }
    waitpid(pid, NULL, 0);
    printf("val = %d\n", val);
    exit(0);
}
```

The output is: `val = 17`

The reason for this is that after the fork(), there are two copies of `val`. The child process does `val -= 3` while the parent waits till the child exits then executes the handler. The handler adds 7 to `var` (the parent’s copy). Finally the parents prints its copy!

5. [6] In case of page fault, the disk needs to be accessed and the page table updated, then the cache is accessed. Therefore, in this problem, whenever there is a page fault, we will leave the physical address and cache hit/miss entries blank.

The correct order is:
- From the virtual address extract the virtual page number (VPN) (see below)
- Access the page table, using VPN, to get the physical page number (PPN) or find it is a page fault.
- With PPN, add the offset and form the total physical address
- Divide this obtained physical address into TAG, Index (i.e. set), and Offset (as indicated below)
- Using the index, access the corresponding set in the cache
- Compare the tag with the tags in the set
- If one of them matches, it is a cache hit.
<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
<th>Page Fault</th>
<th>Cache Hit/Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0A7F</td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>0x08AB</td>
<td>0x5AB</td>
<td>No</td>
<td>Miss</td>
</tr>
<tr>
<td>0x1019</td>
<td>0x019</td>
<td>No</td>
<td>Hit</td>
</tr>
<tr>
<td>0x101B</td>
<td>0x01B</td>
<td>No</td>
<td>Hit</td>
</tr>
</tbody>
</table>

**At the virtual address part:**
The total address is 16 bits → 4 hexadecimal digits
A page is 256 bytes in size → offset is 8 bits → 2 hexadecimal digits
This means that from the 4 digits of the virtual address, the two hexadecimal digits on the left (i.e. the most significant two digits) are the virtual page number (VPN) and the rest are the offset.

**At the physical address part:**
The physical address is 12 bits → 3 hexadecimal digits
Those 12 bits will be divided into 3 parts: TAG, Set (index), and offset, in that order from left to right.
Block size = 4 bytes → 2 bits (the least significant two bits)
The set size = associativity * block size = 2 * 4 = 8 bytes
Total cache size = 64 bytes
Number of sets in the cache = total cache size / set size = 64/8 = 8 → 3 bits for the index
The rest of the 12 bits (i.e. the 7 most significant bits) represent the tag.