# CSCI 2021, Sprint 2015, Homework Assignment III

**Problem 0:**
Clearly label your assignment with the time of your recitation section (8:00, 9:05, 10:10, 11:15, 12:20, 1:25, 2:30). This will help us turn back your graded assignments more efficiently.

The next problem concerns the following C code. This program reads a string on standard input and prints an integer in hexadecimal format based on the input string it read.

```
#include <stdio.h>

/* Read a string from stdin into buf */
int evil_read_string()
{
    int buf[2];

    scanf("%s",(char *)buf);
    return buf[1];
}

int main()
{
    printf("0x%x\n", evil_read_string());
}
```

Here is the corresponding machine code on a Linux/x86 machine:

```
08048414 <evil_read_string>:
 8048414:  55                    push   %ebp
 8048415:  89 e5                 mov    %esp,%ebp
 8048417:  83 ec 14              sub    $0x14,%esp
 804841a:  53                    push   %ebx
 804841b:  83 c4 f8              add    $0xfffffff8,%esp
 804841e:  8d 5d f8              lea    0xfffffff8(%ebp),%ebx
 8048421:  53                    push   %ebx                  address arg for scanf
 8048422:  68 b8 84 04 08        push   $0x80484b8            format string for scanf
 8048427:  e8 e0 fe ff ff        call   804830c <_init+0x50>  call scanf
 804842c:  8b 43 04              mov    0x4(%ebx),%eax
 804842f:  8b 5d e8              mov    0xffffffe8(%ebp),%ebx
 8048432:  89 ec                 mov    %ebp,%esp
 8048434:  5d                    pop    %ebp
 8048435:  c3                    ret

08048438 <main>:
 8048438:  55                    push   %ebp
 8048439:  89 e5                 mov    %esp,%ebp
 804843b:  83 ec 08              sub    $0x8,%esp
 804843e:  83 c4 f8              add    $0xfffffff8,%esp
 8048441:  e8 ce ff ff ff        call   8048414 <evil_read_string>
 8048446:  50                    push   %eax                  integer arg for printf
 8048447:  68 bb 84 04 08        push   $0x80484bb            format string for printf
 804844c:  e8 eb fe ff ff        call   804833c <_init+0x80>  call printf
 8048451:  89 ec                 mov    %ebp,%esp
 8048453:  5d                    pop    %ebp
 8048454:  c3                    ret
```

## Problem 1:

This problem tests your understanding of the stack discipline and byte ordering. Here are some notes to help you work the problem:

- `scanf("%s", buf)` reads an input string from the standard input stream (stdin) and stores it at address `buf` (including the terminating '`\0`' character). It does **not** check the size of the destination buffer.
- `printf("0x%x", i)` prints the integer i in hexadecimal format preceded by "`0x`".
- Recall that Linux/x86 machines are little endian.
- You will need to know the hex values of the following characters:

| Character | Hex value | Character | Hex value |
|---|---|---|---|
| 'd' | 0x64 | 'v' | 0x76 |
| 'r' | 0x72 | 'i' | 0x69 |
| '.' | 0x2e | 'l' | 0x6c |
| 'e' | 0x65 | '\0' | 0x00 |
| | | 's' | 0x73 |

A. Suppose we run this program on a Linux/x86 machine, and give it the string "`dr.evil`" as input on stdin.

Here is a template for the stack, showing the locations of `buf[0]` and `buf[1]`. Fill in the value of `buf[1]` (in hexadecimal) and indicate where `ebp` points just **after** `scanf` returns to `evil_read_string`.

```
                   |<- buf[0]->|<-buf[1] ->|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

What is the 4-byte integer (in hex) printed by the `printf` inside main?

0x________________________

B. Suppose now we give it the input "`dr.evil.lives`" (again on a Linux/x86 machine).

(a) List the contents of the following memory locations just **after** `scanf` returns to `evil_read_string`. Each answer should be an unsigned 4-byte integer expressed as 8 hex digits.

`buf[0] = 0x`________________________

`buf[3] = 0x`________________________

(b) Immediately **before** the `ret` instruction at address `0x08048435` executes, what is the value of the frame pointer register `%ebp`?

`%ebp = 0x`________________________

You can use the following template of the stack as *scratch space*. *Note:* this does **not** have to be filled out to receive full credit.

```
               <- buf[0] -><- buf[1] ->
--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--
```

## Buffer overflow

The next problem concerns the following C code, excerpted from Dr. Evil's best-selling autobiography, "World Domination My Way". He calls the program *NukeJr*, his baby nuclear bomb phase.

```
/*
 * NukeJr - Dr. Evil's baby nuke
 */
#include <stdio.h>

int overflow(void);
int one = 1;

/* main - NukeJr's main routine */
int main() {
  int val = overflow();

  val += one;
  if (val != 15213)
    printf("Boom!\n");
  else
    printf("Curses! You've defused NukeJr!\n");
  _exit(0); /* syscall version of exit that doesn't need %ebp */
}

/* overflow - writes to stack buffer and returns 15213 */
int overflow() {
  char buf[4];
  int val, i=0;

  while(scanf("%x", &val) != EOF)
    buf[i++] = (char)val;
  return 15213;
}
```

## Buffer overflow (cont)

Here is the corresponding machine code for NukeJr when compiled and linked on a Linux/x86 machine:

```
08048560 <main>:
 8048560:       55                pushl  %ebp
 8048561:       89 e5             movl   %esp,%ebp
 8048563:       83 ec 08          subl   $0x8,%esp
 8048566:       e8 31 00 00 00    call   804859c <overflow>
 804856b:       03 05 90 96 04    addl   0x8049690,%eax        # val += one;
 8048570:       08
 8048571:       3d 6d 3b 00 00    cmpl   $0x3b6d,%eax          # val == 15213?
 8048576:       74 0a             je     8048582 <main+0x22>
 8048578:       83 c4 f4          addl   $0xfffffff4,%esp
 804857b:       68 40 86 04 08    pushl  $0x8048640
 8048580:       eb 08             jmp    804858a <main+0x2a>
 8048582:       83 c4 f4          addl   $0xfffffff4,%esp
 8048585:       68 60 86 04 08    pushl  $0x8048660
 804858a:       e8 75 fe ff ff    call   8048404 <_init+0x44>  # call printf
 804858f:       83 c4 10          addl   $0x10,%esp
 8048592:       83 c4 f4          addl   $0xfffffff4,%esp
 8048595:       6a 00             pushl  $0x0
 8048597:       e8 b8 fe ff ff    call   8048454 <_init+0x94>  # call _exit

0804859c <overflow>:
 804859c:       55                pushl  %ebp
 804859d:       89 e5             movl   %esp,%ebp
 804859f:       83 ec 10          subl   $0x10,%esp
 80485a2:       56                pushl  %esi
 80485a3:       53                pushl  %ebx
 80485a4:       31 f6             xorl   %esi,%esi
 80485a6:       8d 5d f8          leal   0xfffffff8(%ebp),%ebx
 80485a9:       eb 0d             jmp    80485b8 <overflow+0x1c>
 80485ab:       90                nop
 80485ac:       8d 74 26 00       leal   0x0(%esi,1),%esi
 80485b0:       8a 45 f8          movb   0xfffffff8(%ebp),%al    # L1: loop start
 80485b3:       88 44 2e fc       movb   %al,0xfffffffc(%esi,%ebp,1)
 80485b7:       46                incl   %esi
 80485b8:       83 c4 f8          addl   $0xfffffff8,%esp
 80485bb:       53                pushl  %ebx
 80485bc:       68 80 86 04 08    pushl  $0x8048680
 80485c1:       e8 6e fe ff ff    call   8048434 <_init+0x74>    # call scanf
 80485c6:       83 c4 10          addl   $0x10,%esp
 80485c9:       83 f8 ff          cmpl   $0xffffffff,%eax
 80485cc:       75 e2             jne    80485b0 <overflow+0x14> # goto L1
 80485ce:       b8 6d 3b 00 00    movl   $0x3b6d,%eax
 80485d3:       8d 65 e8          leal   0xffffffe8(%ebp),%esp
 80485d6:       5b                popl   %ebx
 80485d7:       5e                popl   %esi
 80485d8:       89 ec             movl   %ebp,%esp
 80485da:       5d                popl   %ebp
 80485db:       c3                ret
```

## Problem 2:

This problem uses the NukeJr program to test your understanding of the stack discipline and byte ordering. Here are some notes to help you work the problem:

- Recall that Linux/x86 machines are Little Endian.
- The `scanf("%x", &val)` function reads a whitespace-delimited sequence of characters from `stdin` that represents a hex integer, converts the sequence to a 32-bit `int`, and assigns the result to `val`. The call to `scanf` returns either 1 (if it converted a sequence) or EOF (if no more sequences on `stdin`).

  For example, calling `scanf` four times on the input string `"0 a ff"` would have the following result:

  - 1st call to `scanf`: `val=0x0` and `scanf` returns 1.
  - 2nd call to `scanf`: `val=0xa` and `scanf` returns 1.
  - 3rd call to `scanf`: `val=0xff` and `scanf` returns 1.
  - 4th call to `scanf`: `val` is unchanged and `scanf` returns EOF.

A. After the `subl` instruction at address `0x804859f` in function `overflow` completes, the stack contains a number of objects which are shown in the table below. Determine the address of each object as a byte offset from `buf[0]`.

| Stack object | Address of stack object |
|---|---|
| return address | `&buf[0] +` ________ |
| old %ebp | `&buf[0] +` ________ |
| buf[3] | `&buf[0] +` ________ |
| buf[2] | `&buf[0] +` ________ |
| buf[1] | `&buf[0] + 1` |
| buf[0] | `&buf[0] + 0` |

B. What input string would defuse NukeJr by causing the call to `overflow` to return to address `0x8048571` instead of `804856b`? *Notes: (i) Your solution is allowed to trash the contents of the* `%ebp` *register. (ii) Fill in each blank with a one or two digit hex number.*

`Answer: "0  0  0  0  ___  ___  ___  ___  ___  ___  ___  ___ "`

## Problem 3:

Fill in the following tables. Each table describes the actions that are taken at each stage of the execution for an instruction. Your answer should follow the conventions of section 4.3 in the textbook.

| | `leave` Desribed in problem 4.48 in the textbook |
|---|---|
| Fetch | |
| Decode | |
| Execute | |
| Memory | |
| Write back | |
| PC Update | |

| | `iaddl V, rB` Described in problem 4.47 in the textbook |
|---|---|
| Fetch | |
| Decode | |
| Execute | |
| Memory | |
| Write back | |
| PC Update | |

| | `jmp *D(rA)`<br>`An indirect jump similar to what is described in Section 3.6.3.`<br>`The jump destination is stored in memory location rA + D.` |
|---|---|
| Fetch | |
| Decode | |
| Execute | |
| Memory | |
| Write back | |
| PC Update | |

The following code segment correspond to the next two questions.

```
L5:
        mrmovl (%ebx),%eax      (In 1)
        rmmovl %eax,(%ecx)      (In 2)
        irmovl $4,%edi          (In 3)
        subl %edi,%edx          (In 4)
        addl %edi,%ecx          (In 5)
        xorl %eax,%edi          (In 6)
        andl %edx,%edx          (In 7)
        jg L5                   (In 8)
```

## Problem 4:

Consider a low-cost pipelined processor based on the structure shown in Figure 4.41, with no bypassing paths. In other words, the processor in this problem handles all data hazards by stalling? The branch predictor always predicts that a branch is *not* taken. Fill in the following tables with which execution stage each instruction is in, for each clock cycle. The columns correspond to clock cycles. The rows correspond to individual instructions. Instructions can appear multiple time in the table, because these instructions are part of a loop. For example, the existing entries in the table indicate that `In 1` is in fetch stage in cycle 1, in decode stage in cycle 2, in execution stage in cycle 3 in memory stage in cycle 4 and in writeback stage in cycle 5.

| Iteration 1 | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instr | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| Iteration 1 | | | | | | | | | | | | | | | | | | | | |
| In 1 | F | D | E | M | W | | | | | | | | | | | | | | | |
| In 2 | | | | | | | | | | | | | | | | | | | | |
| In 3 | | | | | | | | | | | | | | | | | | | | |
| In 4 | | | | | | | | | | | | | | | | | | | | |
| In 5 | | | | | | | | | | | | | | | | | | | | |
| In 6 | | | | | | | | | | | | | | | | | | | | |
| In 7 | | | | | | | | | | | | | | | | | | | | |
| In 8 | | | | | | | | | | | | | | | | | | | | |
| Iteration 2 | | | | | | | | | | | | | | | | | | | | |
| In 1 | | | | | | | | | | | | | | | | | | | | |
| In 2 | | | | | | | | | | | | | | | | | | | | |
| In 3 | | | | | | | | | | | | | | | | | | | | |
| In 4 | | | | | | | | | | | | | | | | | | | | |
| In 5 | | | | | | | | | | | | | | | | | | | | |
| In 6 | | | | | | | | | | | | | | | | | | | | |
| In 7 | | | | | | | | | | | | | | | | | | | | |
| In 8 | | | | | | | | | | | | | | | | | | | | |
| Iteration 3 | | | | | | | | | | | | | | | | | | | | |
| In 1 | | | | | | | | | | | | | | | | | | | | |
| In 2 | | | | | | | | | | | | | | | | | | | | |
| In 3 | | | | | | | | | | | | | | | | | | | | |
| In 4 | | | | | | | | | | | | | | | | | | | | |
| In 5 | | | | | | | | | | | | | | | | | | | | |
| In 6 | | | | | | | | | | | | | | | | | | | | |
| In 7 | | | | | | | | | | | | | | | | | | | | |
| In 8 | | | | | | | | | | | | | | | | | | | | |

The throughput of a pipeline is typically measure in terms of cycle per instruction (CPI). CPI is the average number of instructions that complete execution per clock cycle. The performance of a pipeline is function of the CPI and the clock rate.

1. The effective throughput of this pipeline in executing this code segment = _______________;
2. If the clock operates at 1GHz, the instruction-per-second rate achieved by the pipeline on this code = _______________.

## Problem 5:

Consider a pipelined processor with bypassing paths as shown in Figure 4.52 and a perfect branch predictor. Fill in the following tables with which execution stage each instruction is in, during each clock cycle. The columns correspond to clock cycles. The rows correspond to individual instructions. Instructions can appear multiple time in the table, because these instructions are part of a loop. For example, the existing entries in the table indicate that `In 1` is in fetch stage in cycle 1, in decode stage in cycle 2, in execution stage in cycle 3 in memory stage in cycle 4 and in writeback stage in cycle 5. Marks all the bypassings that occur during the execution.

| Iteration 1 | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instr | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| Iteration 1 | | | | | | | | | | | | | | | | | | | | |
| In 1 | F | D | E | M | W | | | | | | | | | | | | | | | |
| In 2 | | | | | | | | | | | | | | | | | | | | |
| In 3 | | | | | | | | | | | | | | | | | | | | |
| In 4 | | | | | | | | | | | | | | | | | | | | |
| In 5 | | | | | | | | | | | | | | | | | | | | |
| In 6 | | | | | | | | | | | | | | | | | | | | |
| In 7 | | | | | | | | | | | | | | | | | | | | |
| In 8 | | | | | | | | | | | | | | | | | | | | |
| Iteration 2 | | | | | | | | | | | | | | | | | | | | |
| In 1 | | | | | | | | | | | | | | | | | | | | |
| In 2 | | | | | | | | | | | | | | | | | | | | |
| In 3 | | | | | | | | | | | | | | | | | | | | |
| In 4 | | | | | | | | | | | | | | | | | | | | |
| In 5 | | | | | | | | | | | | | | | | | | | | |
| In 6 | | | | | | | | | | | | | | | | | | | | |
| In 7 | | | | | | | | | | | | | | | | | | | | |
| In 8 | | | | | | | | | | | | | | | | | | | | |
| Iteration 3 | | | | | | | | | | | | | | | | | | | | |
| In 1 | | | | | | | | | | | | | | | | | | | | |
| In 2 | | | | | | | | | | | | | | | | | | | | |
| In 3 | | | | | | | | | | | | | | | | | | | | |
| In 4 | | | | | | | | | | | | | | | | | | | | |
| In 5 | | | | | | | | | | | | | | | | | | | | |
| In 6 | | | | | | | | | | | | | | | | | | | | |
| In 7 | | | | | | | | | | | | | | | | | | | | |
| In 8 | | | | | | | | | | | | | | | | | | | | |

The throughput of a pipeline is typically measure in terms of cycle per instruction (CPI). CPI is the average number of instructions graduated per clock cycle. The performance of a pipeline is function of the CPI and the clock rate.

1. The effective throughput of this pipeline in executing this code segment = _______________;
2. If the clock operates at 1GHz, the instruction per second achieved by the pipeline on this code = _______________.

## Problem 6:

Consider a pipeline with a complex ALU that is able to support multiplication and divide instructions. Multiplication instructions take 4 cycles to execute; and divide instructions take 6 clock cycles to complete. The ALU generates an ExCmp signal. ExCmp is set at the end of the final cycle of a long computation. For instance if a multiplication starts on cycle 0, ExCmp is set by the end of cycle 3. All other ALU operations still complete in only one cycle as before, and so set ExCmp to 1 by the end of the first cycle when the inputs are presented. Describe changes necessary to the pipeling control logic to incorprate such an ALU.

**Problems 0, 3, and 5 should be submitted for grading.**