ECED 3403 – Computer Architecture

Quiz 1
2 June 2020
Open: 13h30 to 24h00

The following quiz is open book and open notes. State any assumptions made. Time allotted for quiz is 50 minutes, with an additional 15 minutes of overtime. Each question is worth 2 points. Comments have been removed from all code.

1. Show the contents of function `funcX()`'s stack frame (write each memory location on a separate line):

```
void funcX(unsigned int p0, int p1, int p2)
{
int x = 1, y = 2;
/* C-Statements */
}
```

After it is called with the following arguments:

```
funcX(4, -5, 8);
```

Analysis and answer:

A stack frame consists of the called function's automatics, the value of the caller's stack frame address (referred to as BP or back pointer or base pointer), the return address, and the arguments, pushed from right-to-left (in C/C++).

In this case (funcX()'s stack frame is highlighted in green):

Low memory		<- SP
	y (2)	
	x (1)	
	Caller's stack frame pointer	<- BP
	Return address	
	p0 (4)	
	p1 (-5)	
	p2 (8)	
	Top of caller's stack frame	
High memory		

2. Rewrite the following post-test loop with its pre-test equivalent

```
DO
      Statements
UNTIL
      Condition = FALSE
```

Analysis and answer:

A post-test loop is executed *at least once* while a pre-test loop is executed *zero or more times*. If we simply write:

```
WHILE Condition = TRUE DO
      Statements
```

We run into the possibility that `Condition` is `FALSE`, which means that the loop is not executed, although it should be.

To correct this, we need to force the pre-test loop to be executed at least once. This requires the addition of an extra loop control variable. For example:

```
DoOnce = TRUE
WHILE Condition = TRUE OR DoOnce = TRUE DO
      Statements
      DoOnce = FALSE
END WHILE
```

On exit, both `Condition` and `DoOnce` are `FALSE`.

3. The following C code has a potential runtime error. Where would the runtime error occur and why?

```
#define SIZE    10
int array[SIZE];
void init_array()
{
int i=0;
while (i<SIZE)
{
      i++;
      array[i] = 0;
}
}
```

Analysis and answer:

The array, array, has 10 integer elements and can accessed using subscripts:

Subscript	Element
[0]	First (0^{th})
[1]	
…	
[8]	
[9]	Last (9^{th})

The valid range of subscript values is 0 through 9. Anything outside this range could lead to a runtime error or the program exhibiting unexpected behaviours (why?).

The function, `init_array()`, start with `i` (the subscript) having a value of zero and stops when `i` is greater than or equal to `SIZE` (10); however, `i` is incremented before it is used as a subscript:

```
int i=0;
while (i<SIZE)
{
      i++;
      array[i] = 0;
}
```

Consequentially, when `i` is zero, the first element (`array[1]`) is assigned the value 0 and when `i` is 9, the 10th element is assigned the value 0.

Since C does not do bounds checking, the code produced by the compiler allows this to happen. Any data (or code) values in the location `array[10]` is assigned zero. This could result in a system or program runtime error.

Fixing the error is simply a case of swapping the two statements in the while loop:

```
while (i<SIZE)
{
      array[i] = 0;
      i++;
}
```

And yes, the fact that `array[0]` is not assigned a value of zero could also lead to unpredictable behaviour..

4. What is the output of the following C program? Explain why.

```
#include <stdio.h>

char a[] = "ABCD";

int main()
{
int *iptr;
iptr = (int*) a;
*iptr += 0x10203;
printf("%s\n", a);
getchar();
}
```

Analysis and answer:

To properly understand this program, we need to walk through it:

```
char a[] = "ABCD";
```

'a' is an array of five ASCII characters (or five bytes). The first character, 'A', is stored in the zeroth location (say 1000) and the subsequent values are stored in contiguous *byte* locations:

1000	A
1001	B
1002	C
1003	D
1004	NUL

```
int *iptr;
```

'iptr' is a pointer to a 32-bit (4-byte) integer. This means it takes the address of an integer, *not* the value of the integer.

```
iptr = (int*) a;
```

In this statement, the rvalue is:

```
(int*) a
```

Which we interpret as "the value of 'a' is cast (not "casted"), or to be treated as, the address of an integer. This is overriding C's type checking. By referring to 'a', we are referring to the *address* of the first character in 'a'.

Since 'a' has been cast as an integer pointer, it can be assigned to iptr as an lvalue.

'a' has the value 1000, therefore iptr now has the value 1000.

```
*iptr += 0x10203;
```

This statement should be interpreted as:

```
*iptr = *iptr + 0x10203;
```

'*iptr' means that the pointer is *dereferenced*. Dereferencing 'iptr' as an rvalue means we are dealing with a four byte integer starting at address 1000:

1000	A
1001	B
1002	C
1003	D

Or:

D	C	B	A

The rvalue then adds the value 0x00010203 to the integer value in location 1000:

D	C	B	A
00	01	02	03

Since these are ASCII value, we simply increase the "value" of the character by the amount specified:

D	C	B	A
00	01	02	03
D	D	D	D

The rvalue (“DDDD” or 0x44444444) is then assigned to the lvalue.

The lvalue, ‘*iptr’, is also dereferenced, but rather than taking the value from the dereferenced location (as with the rvalue), the dereferenced address refers to the location where the value is to be stored (1000):

1000	D
1001	D
1002	D
1003	D
1004	NUL

```
printf("%s\n", a);
```

In this statement, printf() is called with two arguments, the formatting string and ‘a’. The format string includes ‘%s’ which means treat the corresponding argument as an address of a string (i.e., zero or more characters in contiguous locations followed by a NUL). The argument ‘a’ is not the string, it is the address of the first character in the string (1000).

When executed, printf() displays DDDD.

Simply copying the program and then compiling and running it would produce the result “DDDD”. However, that does not answer the question *why*.

5. The following code is inefficient. Explain why. Then rewrite it so that it is more efficient.

```
Tp = GetToken();
Ip = Chk_Inst(Tp);
Dp = Chk_Dir(Tp);
Lp = Chk_Ptr(Tp);
IF Ip IS NOT EQUAL TO NULL THEN
DO
      Instruction code
END
IF Dp IS NOT EQUAL TO NULL THEN
DO
      Directive code
END
IF Lp IS NOT EQUAL TO NULL THEN
DO
      Label code
END
```

Analysis and answer:

The code is inefficient because `Chk_Dir(Tp)` and `Chk_Ptr(Tp)` are called, potentially unnecessarily. Subsequently, DP and LP are called, again, potentially unnecessarily.

The correction requires `Chk_Dir(Tp)` and `Chk_Ptr(Tp)` to be called only if necessarily; for example:

```
Tp = GetToken();
Ip = Chk_Inst(Tp);
IF Ip IS NOT EQUAL TO NULL THEN
DO
      Instruction code
END
ELSE
      Dp = Chk_Dir(Tp);
      IF Dp IS NOT EQUAL TO NULL THEN
      DO
            Directive code
      END
      ELSE
            Lp = Chk_Ptr(Tp);
            IF Lp IS NOT EQUAL TO NULL THEN
            DO
                  Label code
            END
      ENDIF
ENDIF
```

Using `ELSE` between each `IF` means that unnecessary calls can be avoided.

Putting the call into the `IF` is acceptable:

```
IF Ip = Chk_Inst(Tp) IS NOT EQUAL TO NULL THEN
```