ECED 3403 - Knowledge Survey
Questions and solutions

Larry Hughes, PhD
9 May 2018

1. What is the function of an ALU?

The **ALU** or **Arithmetic and Logic Unit** is part of the **Central Processing Unit** (**CPU**). The ALU performs, as its name suggestions, the arithmetic and logic functions required by the CPU to execute an instruction.

2. In signed arithmetic, why is 0xC7A3 considered to be negative?

When data is considered to be, or treated as, a negative quantity, the most significant or leftmost bit is set (i.e., has a value of 1). Inspecting 0xC7A3 (a data value represented in hexadecimal format), shows that the leftmost bit (in bit 15) is set:

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

Therefore, 0xC7A3 is considered to be a negative number in signed arithmetic.

3. How are characters (such as 'A') stored in memory?

All data (and instructions for that matter) are stored in memory as binary values. Characters, such as 'A' have an internal binary (i.e., numeric) value, determined by the machine's character code (such as **ASCII**)[1].

The binary value of 'A' in ASCII is:

7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1
4				1			

Or 0x41.

Since the ASCII alphabet values are sequential and contiguous, 'B' and 'C' have the values of 0x42 and 0x43, respectively.

Since the internal value of a character is a number, it can be treated as such, allowing operations such as addition or subtraction (this might be prohibited in strongly type-checked languages other than C). For example:

[1] ASCII or **American Standard Code for Information Interchange** is a 7-bit binary computer code (from 0000000 through 1111111) that is the international standard for creating a set of 128 characters, including upper- and lower-case letters of English and other alphabets and various special, numeric and control characters and symbols. The ASCII format allows the translation of a character into a machine-readable numeric form, providing a "universal language" that enables otherwise incompatible systems to exchange data. ASCII files are readable by most computer systems. Source: *Segen's Medical Dictionary*. (2011). Retrieved May 9 2018 from https://medical-dictionary.thefreedictionary.com/ASCII

```
char ch;
for (ch='A'; ch<='Z'; ch++)
    printf("%c ", ch);
```

In the above example, ch is stored internally as a binary number; however, printf() converts the numeric value to a printable character for display, in this case, 'A' through 'Z'.

4. Where is a return address stored?

When one routine calls another, control is transferred to the second routine (in C, a function). When the second routine finishes, it returns to the statement (or instruction) following the call. The location where control is to return is referred to as the **return address**.

In most machines, the return address is stored on the active **stack** as part of the called routine's **stack frame**. When the called routine is finished, the return address is taken from the stack frame, allowing control to return to the calling routing.

5. What is the purpose of the NUL character in a C/C++ string?

The **NUL** character is an ASCII character (0x00 or '\0'). It is used in C to indicate the end of a strong, meaning it is always the last character in the string; for example:

```
char *sptr;
sptr = string;
while (*sptr != '\0')          /* Or simply *sptr */
    printf("%c", *sptr++);
```

NUL is not the same as **NULL** (and not only because of the spelling). NUL refers to a character (8 bits) and NULL refers to a null or zero address pointer value. The number of bits required to represent NULL depends on how addresses are supported on the machine; common sizes are 16-bit (0x0000) or 32-bit (0x00000000).

6. At a minimum, what should be stored when an interrupt occurs?

An **interrupt** or **exception** is an event (external or internal to the machine) that interrupts the currently executing program. In most, but not all, cases, control is to return to the program, meaning that the location where execution is to resume should be stored, typically on the stack (see question 4; an interrupt can be thought of as an implicit subroutine call).

However, since the executing program may be in the middle of a task, it is also necessary to save its **state** to allow the task to resume as if the interrupt never occurred. Most machine store other CPU information such the task's **status register** (or **program status register**, **flags register**, or **condition control register**) and, potentially, other **registers**. A good practice is to explicitly store any registers that are to be used by the **interrupt service routine (**ISR) or **interrupt handler**.

7. How can a number be multiplied by 16 on a machine without a multiplication instruction and without using addition?

Numbers are stored on machines as a binary or base-2 numbers. Each bit (binary digit, a 0 or 1) represents an increasing power of two; starting from the rightmost (least significant) bit: 2^0, 2^1, and so on; for example:

Bit position 'n'	7	6	5	4	3	2	1	0
2^n	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Value	128	64	32	16	8	4	2	1

Shifting a bit from one position to another multiples (left shift) or divides (right shift) the number by 2; for example:

Original number		Shift (<< - left or multiply; >> - right or divide)	Result	
0000.0010	2	0000.0010 << 1	0000.0100	4
0001.0100	20	0001.0100 >> 1	0000.1010	10
0000.1111	15	0000.1111 << 1	0001.1110	30

Shifting left more than once will result in the value of the number increasing by 2 each time. Multiplying by 16 (or 2^4), requires four left shifts; for example, starting with the number 3 (0000.0011):

Result of shift	Value	Action	Multiply by	
0000.0011	2+1 = 3	Original value		
0000.0110	4+2 = 6	<< 1	2^1	× 2
0000.1100	8 + 4 = 12	<< 1	2^2	× 4
0001.1000	16 + 8 = 24	<< 1	2^3	× 8
0011.0000	32 + 16 = 48	<< 1	2^4	× 16

*8. Answer the question (in **bold**) in main():*

This question refers to the **scope** or **visibility** of an entity (i.e., variable, constant, or, depending on the language, function).

An entity is visible or has scope in the **block** (i.e., declarations and statements in the module or inside a pair of **braces**, '**{…}**') in which it is declared. Broadly speaking, this is classified into local (i.e., entities that are known only within the block) and global (i.e., those declared at the "highest level").

When an entity is encountered, it is necessary to look outwards to find its first declaration. The location of the declaration dictates its scope. In the example, there are two x1s, one global (visible to all) and one local (visible inside func1() only).

When the compiler encounters an entity (x1 in this example), it searches for its first (or "nearest") declaration. Inside func1(), the **lvalue** x1 on line 5 refers to the declaration of x1 on line 4. When func1() is executed, x1 is assigned the value of data.

The variable x1 inside func1() "exists" for as long as the function is running (i.e., from when control passes to func1() until the execution of func1() ends). For this reason, variables declared inside a block are termed **automatic** because they are "created" (and "removed") automatically when the block is executing. (Automatic variables are stored on the current stack as part of the subroutine's **stack frame**.)

In other words, the value assigned to x1 exists as long as func1() is executing.

Values assigned to globals, such as x1, exist as long as the program is running. Other than in the case of an automatic declaration, they are visible to all functions. Therefore, on line 9 inside func2(), x1 refers to the global x1.

To solve this problem, it is necessary to "play computer" and trace through its execution. When func2() is called, x1 is assigned the value 5. Next, func1() is called, and x1 is assigned the value 3.

1	int x1;	Global x1	
2	void func1(int data)		Local x1
3	{		Local x1
4	int x1;		Local x1
5	x1 = data;		Local x1
6	}		Local x1
7	void func2(int data)	Global x1	
8	{	Global x1	
9	x1 = data;	Global x1	
10	}	Global x1	
11	main()	Global x1	
12	{	Global x1	
13	func2(5);	Global x1	
14	func1(3);	Global x1	
15	/* **What is the value of x1?** */	Global x1	
16	}	Global x1	

In short, the value of x1 on line 15 is the value of global x1 which was assigned when func2() was called with the value 5. The call to func1() does not affect the value of x1 because the x1 in func1() is local.

9. What is a memory-mapped device?

Most (if not all) devices are associated with device **registers** (not to be confused with CPU registers). The number of registers a device has depends, not surprisingly, on the device itself. A generic device has three registers:

Control. A register (or set of registers) that allows a program to program a device to perform in a specific way. The control register is common to most devices.

Status. A register (or possibly registers) indicating the status of the device; for example, whether it has completed a task. The device may or may not have a status register. Often an interrupt is deemed to be the indication of a device status change (for example, on a timer register).

Data. A register (or registers) to handle the device's input or output, or both. Again, whether the device has data registers depends on what the device does.

These registers are accessible either through special memory (sometimes referred to as **device memory**) or are **memory mapped** (the device registers appear as one of the machine's directly accessible **primary memory**). Device memory requires the machine to have special instructions that access it, whereas memory-mapped registers can be accessed using the machine's existing memory-access instructions (such as **load**, **store**, or **move** – again, depending on the machine).

10. What does overflow indicate?

Overflow occurs in **signed arithmetic** when two numbers with the same sign are added (or subtracted) and the resulting sign is different; for example:

Expression	Result	Overflow?
1000.0000 + 0111.1111 (-128 + 127)	1111.1111 (-1)	Overflow not possible since signs are different
1000.0000 + 1111.1111 (-128 + -1)	0111.1111 (+127)	Overflow has occurred, the resulting sign bit is different from that of the two operands.
0111.1111+0000.0001 (127 + 1)	1000.0000 (-128)	Overflow has occurred, the resulting sign bit is different from that of the two operands.
0000.0001+0000.1111 (1 + 15)	0001.0000 (16)	Although signs are the same, the resulting sign is the same as that of the two operands, so overflow has not occurred.

Overflow is *not* the same as **carry**. Carry is used in **unsigned arithmetic** (or multi-byte/word signed arithmetic) to indicate a carry or borrow has occurred.

Overflow and carry are typically found in the CPU's status register as the **V** and **C** bits, respectively. We will discuss the status register and its bits later in the course.

11. What is stored in an interrupt vector?

Interrupts and exceptions have **interrupt service routines** (**ISR**) (essentially a subroutine) which contains instructions to handle the exception. Calls to an ISR are not explicit (as with a subroutine call), they are implicit (see question 6). This means that the CPU needs to know the address of the ISR.

These addresses are stored in an array of memory locations (typically in low- or high-memory) referred to as **interrupt vectors**. An individual vector contains at a minimum the address of the ISR. Some machines include other information in the vector.

12. What is a heap?

The heap is an area of memory used for the **dynamic allocation** of memory at **run-time** (as opposed to **compile-time**). A routine can call a memory-allocation function such as **malloc()** to request a block of memory. Malloc() (and **free()**) are supported by the underlying memory management routines, which are responsible for supplying contiguous blocks of memory at run-time.

The allocated memory cannot come from the global memory, the stack, or the instruction memory as this is already in use. Most systems allow running programs to access a third "type" of data memory at run-time referred to as the **heap** and managed by the memory manager. Management includes both allocating memory to the caller, but performing "**garbage collection**", ensuring that blocks of contiguous **fragmented memory** (i.e., freed blocks of previously allocated memory) are turned in single blocks of contiguous memory for reallocation.

Traditionally, memory is viewed as follows (note that in ECED 3403 and ECED 4402, "low" memory is at the top and "high" memory is at the bottom):

Low memory	**Program memory**	Static, does not change in size during the execution of the program
	Global memory	Static, does not change in size during the execution of the program. Global constants do not change, whereas global variables can.
…	**Heap**	Memory allocated by the memory manager at run-time. Functions such as malloc() access blocks of memory in the heap, while functions like free() returns it.
High memory	**Stack**	The stack "grows" from high memory to low memory. A stack push decrements the stack pointer, while a pull increments it. The initial stack top should be in the highest possible memory location.

In a machine without memory management, the stack and heap start at opposite ends of the available memory and grow towards each other. Neither should be allowed to "stray" into the program or global memory. This problem can be addressed using hardware memory protection, for example, dividing memory in segments or pages, or both.

13. What is a symbol table?

Assemblers translate **assembly-language records** into machine executable instructions (often stored in **object modules** for the **linkage editor**). Some records have **labels**, while some instructions have operands that refer to labels. The assembler parses each record in the assembly language file, putting label and their addresses into a **symbol table** (operands that refer to labels can require slightly different treatment, this will be discussed in class).

A symbol table typically contains three "parts": the label, the value associated with the label (typically an address or an equated value), and a type. This will be discussed in class.

Compilers that translate statements into object modules (in some cases, compilers produce assembly-language records that are supplied to an assembler). Declarations are often stored in symbol tables and used in much the same way as done with an assembler.

14. What does a dataflow diagram represent?

A dataflow diagram is a graphical tool for showing the flow of data into, out of, and within a system and its entities. We will discuss dataflow diagrams in class as required.

15. When funcX() is called, a segmentation fault occurs. What caused it?

A **segmentation fault** (or **memory access violation**) can occur when a program attempts to access a memory location it does not have access to. This can be a problem when using pointers.

Pointer variables, like all variables have four "parts":

Name: A unique (to its scope) name; in this example, ptr.

Type: A type. In this example, ptr is defined as a pointer ('*') to an integer (int).

Location: A location, somewhere in memory, where it is stored. In this example, ptr is an automatic in funcX(). (One can argue that a variable has a fifth "part", notably an **address**; for the time being, it can be consider synonymous with the location.)

Value: The value of a variable is assigned when the variable is an **lvalue**. A pointer must be assigned the address a memory location before being used; two common ways are:

a) The address of an existing variable; for example:

```
ptr = &data;  /* Assuming data is declared as in integer */
```

b) From the memory manager, using a tool such as malloc():

```
ptr = (int*) malloc(sizeof(int));
```

Note that assigning an address to ptr is not the same as assigning a value to the location where ptr points (by dereferencing ptr as an lvalue):

```
*ptr = value;
```

The process of dereferencing means that the value of ptr (an address) is used as the address to store the expression on the right of the '='s (the **rvalue**). This is equivalent to **indirect addressing**.

In the following example, ptr is defined as a pointer ('*') to an integer (int) – it has a name, type, and location – but it does not have a value (i.e., an address). The assignment in the statement on line 4 has ptr being dereferenced – to an unknown address. If the funcX() has not been given access to this memory, a memory violation (i.e., segmentation error) will occur. This is a **run-time** error: the compiler accepted it as a syntactically correct program and the linker dutifully linked it to the necessary supporting object modules (except malloc() apparently). It wasn't until run-time that the error occurred.

```
1  int *funcX(int value)
2  {
3  int *ptr;
4  *ptr = value;
5  return ptr;
6  }
```

As CEs, it is important to understand the difference between an **error** and a **fault**. An error is the end-result of a fault. The error was the segmentation error. The fault occurred in the original design, failing to assign ptr a value.

Note that returning ptr is perfectly acceptable: funcX() is of type int* (meaning it returns a pointer to an integer) and ptr is of type int* (meaning it is a pointer to an integer).

16. What is a program counter?

A program consists of a series of instructions stored in memory. The CPU must **fetch** each instruction in order to **decode** and **execute** it. The fetch step requires the CPU to maintain state, in this case the address of the next instruction to fetch, the address is held in a register referred to as the **Program Counter** or **PC**. (On some machine, the program counter is referred to as the **Instruction Pointer** or **IP**.)

In sequential instructions, the PC is incremented by the size of the instruction. In conditional instructions, the PC may be incremented by the size of the instruction or it may be assigned the value of the target address (for example, the else-statement or the end-if statement).

When a subroutine is called, the value of incremented PC is saved as the **return address** (for example, on the stack) and the address of the subroutine is assigned to the PC.

17. How many bytes of storage would the following structure occupy?

Fundamental types (**char**s, **short**s, etc.) are stored in memory. Memory size is typically expressed in terms of bytes. Not surprisingly, each fundamental type is associated with a specific number of bytes: char (1 byte); short (2 bytes); int (4 bytes, but can be considered as 2 bytes on some machines); and **double**s (twice the size of an int).

Aggregate types (structures and arrays) are groupings of fundamental or aggregate types (for example, an array of ints or an array of structures). The size of an aggregate type is determined by the total number of bytes in the fundamental types used to create the new type.

A C-structure, **struct**, is an aggregate type and consists of one or more fundamental or aggregate types referred to as **fields**. The size of the structure is therefore the sum of the size of each field

struct node {	**Comments**	**Number of bytes**
char name[8];	- name is an array '[]' of 8 chars, or 8-bytes	8
short age;	- age is a short, or 2-bytes	2
char shoesize;	- shoesize is a char, or one byte	1
struct node *next;	- next is a pointer '*' to a structure of type node. The size of a pointer is machine or compiler dependent. If we assume 16-bit pointers, next would require 2-bytes.	2
};	**Total**	13

The fact that the total is 13 raised some interesting points (not because it is 13, but because 13 is an odd number. We will discuss this in more detail in class.

18. In a machine that only supports addition, how is subtraction performed?

Some architectures only support addition or, if subtraction is supported, it is done using the **method of complements** which involves taking the ones-complement of the subtrahend, adding it to the minuend, and adding '1' to the result to give the difference. The process of taking the ones-complement of the subtrahend and adding one to the result ensures a twos-complement result (note that all numbers begin as two-complement numbers); for example:

Expression	**Binary**	**Complement of subtrahend**	**Result of addition**	**Result plus 1**	**Difference**	
3	0000.0011		0000.0011	0000.0000	0000.0001	3
-2	-0000.0010	1111.1101	+1111.1101	+0000.0001	(1)	-2
?			0000.0000	0000.0001		1
-5	1111.1011		1111.1011	0000.0100	0000.0101	-5
- -10	-1111.0110	0000.1001	+0000.1001	+0000.0001	(5)	- -10
?			0000.0100	0000.0101		5

19. What is a Harvard architecture?

Programs consists of instructions and data stored in **primary memory**. While this memory is often discussed or thought-of as a single, contiguous entity, some machines divide the primary memory into two, one part for instructions and the other for data. Machines with separate instruction and data memory are referred to as having a **Harvard architecture**.

Machines in which instructions and data share the same **address space** are referred to as having a **Princeton** or **von Neumann architecture**.

There are arguments for both types of architecture.

20. In a function call, where are the arguments stored?

Arguments to a function are typically stored on the current stack as part of the callee's stack frame (built, in part, by the calling subroutine). We will be discussing subroutine calls and the stack frame in more detail in class.

21. Rewrite the following code fragment using a single if-else:

```
if (a > b)
    funcA();
else
if (a == b)
    funcB();
else
    funcA();
```

In this case, funcB() is called only if the condition "a==b" is true (i.e., non-zero); in all other cases, funcA() is called. The code could be rewritten as:

```
if (a == b)
    funcB();
else
    funcA();
```

There were some poorly coded solutions to this problem. Keeping things simple makes life much easier to understand.

22. Name the three loop structures.

- Pre-test, performed zero or more times (e.g., a C while loop)
- Post-test, performed one or more times (e.g., a C do-while loop)
- Deterministic, with an explicit starting value, increment, and termination condition (e.g., a C for loop).

23. What are the values of 'a' and 'b' when this loop terminates?

```
while (a < 10 && b == 0)
{
   /* Instructions to change 'a' and 'b' */
}
```

When a loop terminates the condition for staying in the loop is no longer true. A straightforward way of determining this is to apply de Morgan's rules by complementing the loop condition. In this case:

Expression	**Complement**
a < 10	a >= 10
b == 0	b != 0
&&	||

The complemented conditional expression is therefore:

```
a >= 10 || b != 0
```

It is very good programming practice to put the complement of the loop as a comment at the end of the loop as a reminder as to why the code is at that point (this is extremely useful when a loop spans more than one page):

```
while (a < 10 && b == 0)
{
   /* Instructions to change 'a' and 'b' */
}
/* a >= 10 || b != 0 */
```

The same coding habit should also be used with if-then-else statement. The program's speed doesn't change, it just makes it easier for the poor person maintaining your software. Remember, it could be you!

24. What does a state diagram (or state machine) represent?

A state diagram is a graphical tool used in the design step of solving a problem. We will discuss state diagrams in more detail in class.

25. List the four main parts of a computer.

A generic computer has four "parts":

CPU. The **Central Processing Unit** (CPU) where instructions are decoded and executed. The CPU typically contains at least one **ALU**, an **instruction decoder**, and a **register file**.

Memory. Memory holds programs consisting of instructions and data (global, stack, and heap). This memory is usually referred to as **primary memory**. Memory which holds programs or data that are not active in the primary memory is referred to as **secondary memory**. Secondary memory is usually some form a storage device such as a disk.

Bus. A typical bus consists three parts: address lines (carrying addresses from the CPU to primary memory or devices), **bi-directional data lines** (carrying data between primary memory and the CPU; the data lines also carry instructions from primary memory to the CPU); and **control lines**, indicating whether the CPU wants to **read** from or **write** to a memory location.

Devices. Devices connect to the computer, allowing it to access the outside world (or the outside world to access it).