Assignment 2

Question:
What does it mean when a function does not return a value? How do you state that in a program? How can then the function (or more precisely, the procedure) perform anything useful?

Answer:
That a function doesn't return a value (indicated by **void <FUNCTION NAM>**) means that no return is to be expected. A function that doesn't return a value can still be useful if it for instance:

- Works with outputs (for instance printing to console [printf])
- Modifies a value through pointers
- Checks data and throws exceptions if needed
- And more…

Question:
How did you implement the side effect that is needed to make print_number behave correctly?

Answer:
We used a counter that was compared in each iteration to the global COLUMNS to determine when we needed to print out a line break. The counter increases by 1 when a prime number is found and is reseted after line break is printed.

Assignment 3

Question:
How did you represent the marking of 'prime' and 'not a prime' in the memory array?

Answer:
By using 0 and 1 as false and true.

Question:
Which are the main steps in the algorithm? How have you implemented these steps?

Answer:
The main steps are looping through each value and jumping that many steps forward in the array and setting each of those spots to false (0, not prime). If the value is false (0), we continue to the next iteration.
.
Question:
What is the largest prime number that you can print within 2 seconds of computation? What is the largest number you can print within 10 seconds? Is it the same for print_prime.c, sieves.c, and sieves-heap.c? Why or why not?

Answer:
The results vary due to the algorithm and different limits.
Sieves-heap handles larger numbers than **sieves.c**, while both **sieves.c** and **sieves-help.c** is faster than **print_prime.c**. However, **sieves-heap.c** hits a limit where greater input ends up as a slower process.

	print_prime.c	**sieves.c**	**sieves-heap.c**
2 sec	30773	MAXING OUT	-
10 sec	77839	MAXING OUT	-

Assignment 4

Question:
Explain how you get the pointer addresses to the two char arrays (text1 and text2) and the counter variable (count) in function work().

Answer:
The global pointers text1 and text2 are holding the addresses. These are passed as arguments via work() to copycodes() where they are saved as new pointers. The counter on the other hand doesn't have a global pointer. Instead it is defined as a normal int. So to pass its address as an argument from work() to copycodes() we add the &-sign before the variable name (&count). copycodes() takes this address and creates a pointer.

Question:
What does it mean to increment a pointer? What is the difference between incrementing the pointer that points to the ASCII text string, and incrementing the pointer that points to the integer array? In what way is the assembler code and the C code different?

Answer:
When you increment a pointer, you increment the value of the memory address where the pointer is pointing to. When incrementing the pointer that points to the ASCII text string, you go through each value that the string holds. While when incrementing the array(in our case the allocated memory) you increment the memory slot (if you have an int pointer you increment by 4 bits, while having a char pointer you increment by 1 bit, the compiler knows this). When allocating the memory for the lists, the MIPS code allocates that memory globally, while in the C code you have to allocate that memory in main. In MIPS the registers hold the address for count, while in C when calling copy_codes in work() you have to select the “count” address with the &-sign (like this &count).

Question:
What is the difference between incrementing a pointer and incrementing a variable that a pointer points to? Explain how your code is incrementing the count variable.

Answer:
If you increment a pointer, you increment its address and thus make it point to different data. While if you increment its value, the pointer will continue pointing to the same address, which data has been incremented.

To increment the count variable, we first pass the count-address to copycodes() where it’s turned into a pointer. Then we target its value and save to it its value plus one (increment by one). This is done through (*countAddr)++; or equivalently *countAddr = *countAddr + 1;.

Question:
Explain a statement in your code where you are dereferencing a pointer. What does this mean? Explain by comparing with the corresponding assembler code.

Answer:
In line 16(the for loop(in Pontus code)) we dereference (load value from address) textP, and get the first char in the char “array”. Compared to the MIPS code the “same” thing happens in line 51(lb $t0, 0 ($a0)) where you load the array (store in address $a0) into $t0 and then proceed to loop it to extract the values into the list.

Question:
Is your computer using big-endian or little-endian? How did you come to your conclusion? Is there any benefit of using either of the two alternatives?

Answer:
We can figure out if our computers are using big or small endian by looking at the output from endian_proof(). If count = 35 (we know that 35 belongs to the least significant bytes), the output yields 0x23,0x00,0x00,0x00. These values are printed in increasing address order. Thus 0x23 has the lowest address and our computer uses little endian. There can be benefits of using little-endian and big-endian, but in the bigger picture the choice of endian is redundant.

Assignment 5

Question:
Consider AM18, AM19, and AF1. Explain why gv ends up with the incremented value, but m does not.

Answer:
Gv ends up with incremented value because fun takes m as an argument, increments the value and saves it to gw. And since m wasn't passed as an address, the original remains untouched.

Question:
Pointer cp is a character pointer that points to a sequence of bytes. What is the size of the cp pointer itself?

Answer:
The pointer itself has a size of 8 bytes.

Question:
Explain how a C string is laid out in memory. Why does the character string that cp points to have to be 9 bytes?

Answer:

0x?	**0x?**	**0x?**	**0x?**	**0x?**	**0x?**	**0x?**	**0x?**	**0x00**

The C string is one extra byte long so that we get a null byte which acts as a break/divider. Each letter has its own address and location in memory.

Question:
Which addresses have fun and main? Which sections are they located in? What kind of memory are they stored in? What is the meaning of the data that these symbols points to?

Answer:
Fun has address 0x9D001180 which is located in Virtual Memory, Program Flash 2.
Main has address 0x9D0011D8 which is also located in Virtual Memory, Programs Flash 2.
The meaning of the data that's displayed are addresses referring to reserved memory.

Question:
Which addresses are variables in and gv located at? Which memory sections according to the PIC32 memory map? Why?

Answer:
"in" is located at 0xA0000008 and located at RAM2 in virtual memory.
"gv" is located at 0xA000000C and located at RAM2 in virtual memory.
They are stored in RAM because they are variables that need to be quickly accessible.

Question:
Variables p and m are not global variables. Where are they allocated? Which memory section is used for these variables? Why are the address numbers for p and m much larger than for in and gv?

Answer:
“p” is located at 0xA0003FE8 and “m” at 0xA00eFE4. They are located at reserved memory because they are initialised in main, while “in” and “gv” are initialised globally

Question:
At print statement AM5, what is the address of pointer p, what is the value of pointer p, and what value is pointer p pointing to?

Answer:
At AM5, p’s address is 0xA0003FE8 and the value of the pointer is address 0xA0003FE4, which points to 7.

Question:
At print statement AM7, what is the address of pointer p, what is the value of pointer p, and what value is pointer p pointing to?

Answer:
At AM7, p’s address is 0xA0003FE8 and the value of the pointer is address 0xA0003FE4, which points to 8 (incremented at line 54).

Question:
Consider AM14 to AM17. Is the PIC32 processor using big-endian or little-endian? Why?

Answer:
Because the least significant bits are stored at the lower address, the PIC32 uses little-endian.