Practice Questions from Deck 8 - Ai made csc 230

1. What was the original purpose of the C programming language when it was developed at Bell Labs in the early 1970s?

2. When dealing with bitwise and logical operations in C, what is the key difference in how zero and non-zero values are interpreted?

3. In the context of AVR C programming, what is the purpose of declaring a variable as "volatile" and when should it be used?

4. In the AVR architecture, how are parameters passed to C functions according to the ABI (Application Binary Interface)?

5. What is the significance of the #include directive in C compared to Java's import statement?

6. When mixing C and assembly code, what is the purpose of the .global directive in GNU assembler format?

7. What are the two categories of registers that assembly code can use according to the AVR ABI?

8. What is the purpose of using a function prototype in C when calling assembly routines?

9. When implementing interrupt handlers in C, what are the two main "gotchas" to be aware of?

10. What advantage does implementing a switch statement have over using multiple if-else statements when the possible values are known at compile time?

Answers

Question 1: What was the original purpose of the C programming language when it was developed at Bell Labs in the early 1970s?

Answer: According to Slide 3, C was invented at Bell Labs in the early 1970s with the original purpose of coding the kernel of an operating system. Writing the kernel in C made it easier to port the OS to different hardware architectures.

Question 2: When dealing with bitwise and logical operations in C, what is the key difference in how zero and non-zero values are interpreted?
Answer: As shown in Slides 32-34, in C when a conditional expression is evaluated, zero means false, and all other non-zero values mean true. This is particularly important when using bitwise operations in control flow statements, where confusion between bitwise and logical operators can lead to subtle bugs.

Question 3: In the context of AVR C programming, what is the purpose of declaring a variable as "volatile" and when should it be used?
Answer: According to Slide 52, the volatile keyword is a declaration that gives the compiler advice that the variable may change at any time (such as through an interrupt handler). It prevents certain optimizations like keeping values in registers, ensuring the most current value is always read from memory. Slide 53 shows a practical example with the LED counter program.

Question 4: In the AVR architecture, how are parameters passed to C functions according to the ABI (Application Binary Interface)?
Answer: As detailed in Slides 59-60, the ABI specifies that:
- For fixed parameter lists, parameters appear from register r25 down through to r8
- All arguments use an even number of registers
- For variable-length parameter lists, parameters are pushed onto the stack in left-to-right order
- Return values use registers r25 down through to r18

Question 5: What is the significance of the #include directive in C compared to Java's import statement?
Answer: According to Slide 11, #include is not the same as Java's import. The slide specifically notes that #include does not always include the source code for functions. Instead, it makes available constants and function signatures to the program. When discussing linking, we learn how compiled code really accesses library code.

Question 6: When mixing C and assembly code, what is the purpose of the .global directive in GNU assembler format?
Answer: As shown in Slide 61, when mixing C and assembler, the .global directive introduces a name to the program namespace. The slide demonstrates this with multiple assembly functions that can be called from C.

Question 7: What are the two categories of registers that assembly code can use according to the AVR ABI?
Answer: According to Slide 59, there are:
- Registers that can be freely used by assembly code: (r18–r27, r0, r31)
- Registers that may be used by assembly code but must be saved and restored: (r2–r17, r28, r29)

Question 8: What is the purpose of using a function prototype in C when calling assembly routines?
Answer: As covered in Slide 55-56, function prototypes are necessary when C code will call assembly routines. The prototype provides information to help the compiler understand the interface of the assembled routine.

Question 9: When implementing interrupt handlers in C, what are the two main "gotchas" to be aware of?
Answer: According to Slide 48, the two main concerns are:
- Finding the name of the interrupt
- Dealing with program-scope variables

Question 10: What advantage does implementing a switch statement have over using multiple if-else statements when the possible values are known at compile time?
Answer: As shown in Slides 43-45, switch statements highlight the importance of individual cases, making it easier to add new cases and quickly find and modify existing ones. The slides demonstrate this with the LED counter example, showing how switch statements can provide clearer organization of code compared to multiple if-else statements.