MSP430 Assembly Programming vs C: Curriculum Examples

Example 1: Bit Manipulation with Rotate

Problem: Rotate a register's bits left by one position in a loop until a specific bit is set.

Assembly Solution:
```assembly
        MOV.W   #0x01, R4       ; Load R4 with initial value (0000 0001b)
        MOV.W   #8, R5          ; Set counter for 8 rotations
LOOP:
         RLA.W   R4          ; Rotate left through carry
        JNC     LOOP        ; Jump if no carry (bit not set)
         ; Exit loop when a carry is generated
```

C Solution:
```c
unsigned int r4 = 1;
for (int i = 0; i < 8; i++) {
    r4 = r4 << 1;
    if (r4 & 0x100) break; // Check for carry
}
```

- **Why Assembly is Better**:
  - Direct control of hardware instructions (e.g., `RLA`).
  - No function call overhead or additional instructions.
  - Efficient use of the carry flag.

Example 2: Efficient Conditional Jumping

Problem: Write a program to determine whether a number is even or odd using minimal instructions.

Assembly Solution:```assembly
        MOV.B   R4, R5   ; Copy number to check
        AND.B   #0x01, R5   ; Mask the least significant bit
        JNZ   ODD   ; Jump if result is not zero (odd)
EVEN:
         ; Code for even case
        JMP   END
ODD:
        ; Code for odd case
END:
```

**C Solution**:```c
```

```
if (number & 0x01) {
    // Code for odd case
} else {
    // Code for even case
}
```

- **Why Assembly is Better**:
  - Assembly directly uses the processor's conditional branching instructions (`JNZ`), avoiding the overhead of compiler-generated branching code in C.
  - No extra variables or operations like comparisons are generated by the compiler.

Example 3: Handling State Machines

Problem: Implement a simple state machine with states represented as labels and transitions based on conditions.

Assembly Solution:```assembly
START:
    CMP   #1, R4   ; Compare current state
    JEQ   STATE1   ; Jump to STATE1 if R4 == 1
    CMP   #2, R4
    JEQ   STATE2   ; Jump to STATE2 if R4 == 2
    JMP   END   ; Default case

STATE1:
    ; Code for STATE1
    JMP   END

STATE2:
    ; Code for STATE2
    JMP   END

END:
    ; End of program```

C Solution:
```c
switch (state) {
    case 1:
        // Code for state 1
        break;
    case 2:
        // Code for state 2
        break;
    default:
        // Default case
```

```
        break;
}
```

- **Why Assembly is Better**:
  - Direct implementation of jumps without the compiler's intermediate "jump table" or comparison logic.
  - No overhead of a `switch` statement or function calls.

Example 4: High-Speed Counters

Problem: Count the number of set bits in a register.

Assembly Solution:```assembly
            MOV.W   R4, R5   ; Copy register value
            CLR.W   R6   ; Clear counter
LOOP:
            BIT.W   #1, R5   ; Test the least significant bit
            JZ   SKIP   ; Skip increment if bit is zero
            INC.W   R6   ; Increment counter
SKIP:
            RRA.W   R5   ; Rotate right
            JNZ   LOOP   ; Continue until all bits are checked
```

**C Solution**: ```c
unsigned int count = 0;
while (number) {
    if (number & 1) count++;
    number >>= 1;
}
```- **Why Assembly is Better**:
  - Bit testing (`BIT.W`) and rotation (`RRA.W`) are more efficient than shifts and masks in C.
  - The compiler generates more verbose code for such operations in C.

1. **Conditional Jumping**: Showcases direct control of program flow.
2. **Rotate Commands**: Highlights the simplicity of manipulating bits without using logical operations like AND or OR.
3. **Hardware-Level Efficiency**: Demonstrates that assembly eliminates the abstraction layer introduced by C compilers, allowing precise optimizations.

By illustrating these examples, you can effectively demonstrate the advantages of assembly programming for performance-critical tasks on MSP430 processors.