3.1 Comparison and Conditions

Introduction to Conditional Jumps

In assembly language programming, conditional jumps are crucial for controlling the flow of execution based on specific conditions. These jumps allow a program to branch into different sections of code depending on the results of previous operations. One fundamental instruction used for comparison is **CMP**, which stands for "compare."

The `CMP` instruction works by subtracting a source operand from a destination operand. Importantly, `CMP` updates the processor's status flags based on the outcome of the subtraction, but it does not change the values of the operands themselves. This makes `CMP` a key instruction for enabling conditional logic in assembly language, as it allows for evaluations of relationships between values.

Comparison Through Subtraction

When performing a comparison with the `CMP` instruction, various conditions can be extracted from the result of the subtraction. Here are some examples of conditions that can be evaluated:

1. **Borrow Needed**:

   - If a larger number is subtracted from a smaller one, this indicates that a borrow is needed. In this case, the **Carry Flag (CF)** will be set. This flag signals that the result of the subtraction cannot be represented within the register's limits.

2. **Equal Numbers**:

   - If two equal numbers are compared (i.e., subtracted), the result will be zero. Consequently, the **Zero Flag (ZF)** will be set, indicating that the operands are equal.

3. **Using Flags for Conditional Logic**:

   - The various flags—**Sign Flag (SF)**, **Carry Flag (CF)**, **Zero Flag (ZF)**, and **Overflow Flag (OF)**—provide a comprehensive set of indicators that reveal the

relationship between the two operands after a comparison. By checking these flags, a programmer can decide which branch of code to execute next.

The Importance of Signed vs. Unsigned Numbers

Understanding the distinction between signed and unsigned numbers is crucial in assembly programming:

- **Unsigned Numbers**: For unsigned numbers, only the magnitude is significant. For example, the number `255` is the maximum representable value in an 8-bit unsigned integer.

- **Signed Numbers**: For signed numbers, both the magnitude and the sign are important. The range for an 8-bit signed integer is from `-128` to `+127`. For instance, while `-2` is greater than `-3`, the number `2` is smaller than `3`. The sign directly influences how comparisons are interpreted.

Two's Complement Representation

In most computer architectures, signed integers are represented using **two's complement notation**. This representation allows for simple arithmetic operations while treating positive and negative numbers uniformly. In this representation:

- A positive number remains unchanged.

- A negative number is obtained by inverting the bits of its absolute value and adding 1 to the least significant bit (LSB).

For instance:

- The number `2` is represented as `0000 0010` in binary (8 bits).

- The number `-2` is represented as `1111 1110` in binary.

Even though `-2` and `65534` can occupy the same memory space in binary, their interpretations differ based on whether they are treated as signed or unsigned.

Intent in Comparisons

When performing comparisons using the `CMP` instruction, the intent of the programmer regarding whether to treat the numbers as signed or unsigned is not immediately clear. This intent is clarified during the **conditional jump operation** that follows the comparison.

For example, consider comparing `-2` (represented as `0xFFFE` in a 16-bit signed integer) with `2` (represented as `0x0002`). If a **Jump Above (JA)** instruction is used after this comparison, it will take the jump because it interprets the values as unsigned. However, if a **Jump Greater (JG)** instruction is used, it will not take the jump since it evaluates the values as signed, determining that `-2` is less than `2`.

Summary of Flags and Comparisons

The table below summarizes how different comparisons translate into flag settings, clarifying the outcomes of operations on signed and unsigned numbers:

1. **DEST = SRC**: **ZF = 1** (Equal numbers set the zero flag.)
2. **UDEST < USRC**: **CF = 1** (When the destination is smaller than the source.)
3. **UDEST ≤ USRC**: **ZF = 1 OR CF = 1** (Either equality or a borrow.)
4. **UDEST ≥ USRC**: **CF = 0** (No borrow means the destination is greater than or equal to the source.)
5. **UDEST > USRC**: **ZF = 0 AND CF = 0** (The destination is greater if both flags are clear.)
6. **SDEST < SSRC**: **SF ≠ OF** (If the result is negative without overflow, the destination is smaller.)
7. **SDEST ≤ SSRC**: **ZF = 1 OR SF ≠ OF** (Equality or negative results indicate the destination is smaller.)
8. **SDEST ≥ SSRC**: **SF = OF** (Positive results without overflow indicate the destination is larger.)
9. **SDEST > SSRC**: **ZF = 0 AND SF = OF** (Non-equal and matching flags indicate the destination is greater.)

Conclusion

Understanding comparisons and conditions in assembly language is vital for writing efficient and functional code. The distinctions between signed and unsigned numbers affect how comparisons are made and interpreted, impacting the program's flow based on conditional jumps. By using the `CMP` instruction effectively, programmers can harness the power of flags to control program execution based on the relationships between values.