Table 4.3 Conditional Jump Instructions

Mnemonic	Condition
	Signed Operations
JG/JNLE	Greater/not less or equal $((SF \oplus OF) + ZF) = 0$
JGE/JNL	Greater or equal/not less $(SF \oplus OF) = 0$
JL/JNGE	Less/not greater or equal $(SF \oplus OF) = 1$
JLE/JNG	Less or equal/not greater $((SF \oplus OF) + ZF) = 1$
JO	Overflow (OF = 1)
JS	Sign (SF = 1)
JNO	Not overflow (OF = 0)
JNS	Not sign (SF = 0)
	Unsigned Operations
JA/JNBE	Above/not below or equal $(CF \oplus ZF) = 0$
JAE/JNB	Above or equal/not below (CF = 0)
JB/JNAE	Below/not above or equal (CF = 1)
JBE/JNA	Below or equal/not above $(CF \oplus ZF) = 1$
	Either
JC	Carry (CF = 1)
JE/JZ	Equal/zero (ZF = 1)
JP/JPE	Parity/parity even (PF = 1)
JNC	Not carry (CF = 0)
JNE/JNZ	Not equal/not zero (ZF = 0)
JNP/JPO	Not parity/parity odd (PF = 0)

Note that in this program we have used the name A1 to represent the memory location in which the IN instruction is stored.

Large programs are often made up of many smaller *subprograms* (subroutines) or *procedures*. When it is desired to execute one of these procedures, the CALL instruction is used. The CALL transfers control to the subprogram but also saves the return address on the stack.[10] When the procedure finishes, it executes an RET instruction. This recovers the return address from the stack, allowing the calling program to resume from where it left off. (Chapters 4, 5, and 6 explore this programming concept in more detail.)

Finally, the *interrupt* instructions transfer control to an *interrupt service routine (ISR)* whose address is set up in a special interrupt vector table in low memory. (These instructions are covered in Section 4.3.)

High-Level Language Instructions [Table 4.2(g)]. These instructions assist in the development of high-level languages. BOUND, for example, checks that the specified register is pointing within a predefined range of memory addresses. If not, an interrupt is generated. ENTER and LEAVE are used to allow variables to be passed to a subroutine via the stack.

[10]The stack is a special area of memory used for storing temporary values and memory addresses. The stack operates in a *last-in first-out* manner; that is, the last item written to the stack becomes the first item read from the stack. For more details, see *Using the Stack* on page 142.