Table 4-19. Floating-Point Store Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Store Floating-Point Double with Update Indexed	**stfdux**	**fr**S,**r**A,**r**B	The EA is the sum (**r**A) + (**r**B). The contents of **fr**S are stored into the double word in memory addressed by EA. The EA is placed into register **r**A.
Store Floating-Point as Integer Word Indexed	**stfiwx**	**fr**S,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The contents of the low-order 32 bits of **fr**S are stored, without conversion, into the word in memory addressed by the EA. **Note**: The **stfiwx** instruction is defined as optional by the PowerPC architecture to ensure backwards compatibility with earlier processors; however, it will likely be required for subsequent PowerPC processors.

4.2.4 Branch and Flow Control Instructions

Some branch instructions can redirect instruction execution conditionally based on the value of bits in the CR. When the processor encounters one of these instructions, it scans the execution pipelines to determine whether an instruction in progress may affect the particular CR bit. If no interlock is found, the branch can be resolved immediately by checking the bit in the CR and taking the action defined for the branch instruction.

If an interlock is detected, the branch is considered unresolved and the direction of the branch may either be predicted using the *y* bit (as described in Table 4-20) or by using dynamic prediction. The interlock is monitored while instructions are fetched for the predicted branch. When the interlock is cleared, the processor determines whether the prediction was correct based on the value of the CR bit. If the prediction is correct, the branch is considered completed and instruction fetching continues. If the prediction is incorrect, the fetched instructions are purged, and instruction fetching continues along the alternate path.

4.2.4.1 Branch Instruction Address Calculation

Branch instructions can alter the sequence of instruction execution. Instruction addresses are always assumed to be word aligned; the PowerPC processors ignore the two low-order bits of the generated branch target address.

Branch instructions compute the effective address (EA) of the next instruction address using the following addressing modes:

- Branch relative
- Branch conditional to relative address
- Branch to absolute address
- Branch conditional to absolute address
- Branch conditional to link register
- Branch conditional to count register

In the 32-bit mode of a 64-bit implementation, the final step in the address computation is clearing the high-order 32 bits of the target address.

4.2.4.1.1 Branch Relative Addressing Mode

Instructions that use branch relative addressing generate the next instruction address by sign extending and appending 0b00 to the immediate displacement operand LI, and adding the resultant value to the current instruction address. Branches using this addressing mode have the absolute addressing option disabled (AA field, bit 30, in the instruction encoding = 0). The link register (LR) update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This option causes the effective address of the instruction following the branch instruction to be placed in the LR.

Figure 4-6 shows how the branch target address is generated when using the branch relative addressing mode.

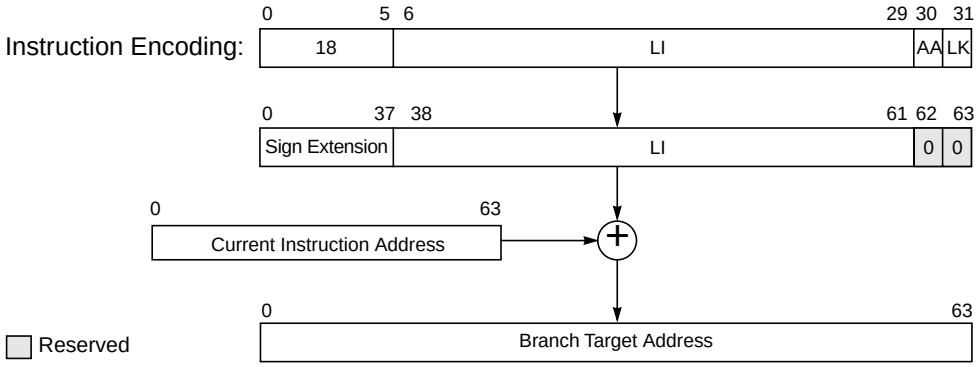

Figure 4-6. Branch Relative Addressing

4.2.4.1.2 Branch Conditional to Relative Addressing Mode

If the branch conditions are met, instructions that use the branch conditional to relative addressing mode generate the next instruction address by sign extending and appending 0b00 to the immediate displacement operand (BD) and adding the resultant value to the current instruction address. Branches using this addressing mode have the absolute addressing option disabled (AA field, bit 30, in the instruction encoding = 0). The link register update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This option causes the effective address of the instruction following the branch instruction to be placed in the LR.

Figure 4-7 shows how the branch target address is generated when using the branch conditional relative addressing mode.

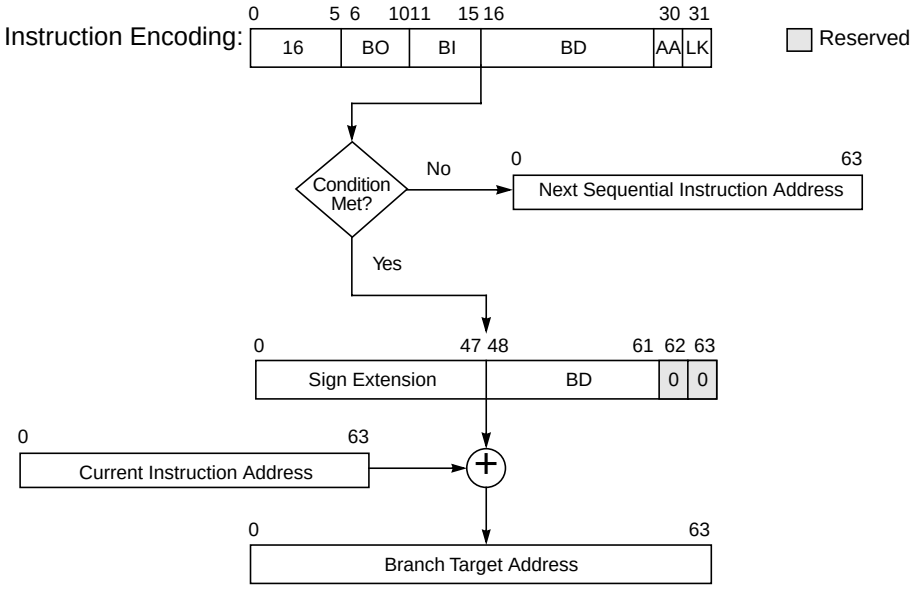

Figure 4-7. Branch Conditional Relative Addressing

4.2.4.1.3 Branch to Absolute Addressing Mode

Instructions that use branch to absolute addressing mode generate the next instruction address by sign extending and appending 0b00 to the LI operand. Branches using this addressing mode have the absolute addressing option enabled (AA field, bit 30, in the instruction encoding = 1). The link register update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This option causes the effective address of the instruction following the branch instruction to be placed in the LR.

Figure 4-8 shows how the branch target address is generated when using the branch to absolute addressing mode.

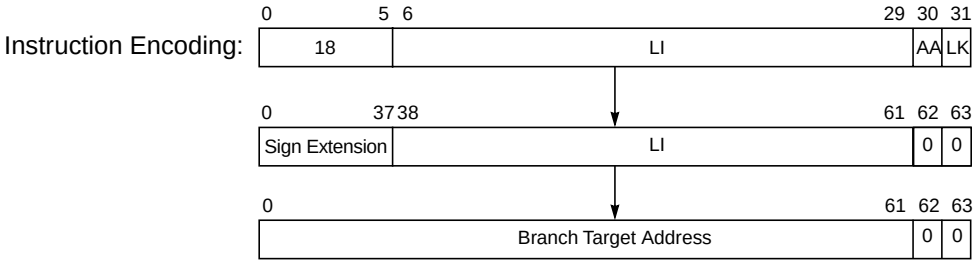

Figure 4-8. Branch to Absolute Addressing

4.2.4.1.4 Branch Conditional to Absolute Addressing Mode

If the branch conditions are met, instructions that use the branch conditional to absolute addressing mode generate the next instruction address by sign extending and appending 0b00 to the BD operand. Branches using this addressing mode have the absolute addressing option enabled (AA field, bit 30, in the instruction encoding = 1). The link register update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This option causes the effective address of the instruction following the branch instruction to be placed in the LR.

Figure 4-9 shows how the branch target address is generated when using the branch conditional to absolute addressing mode.

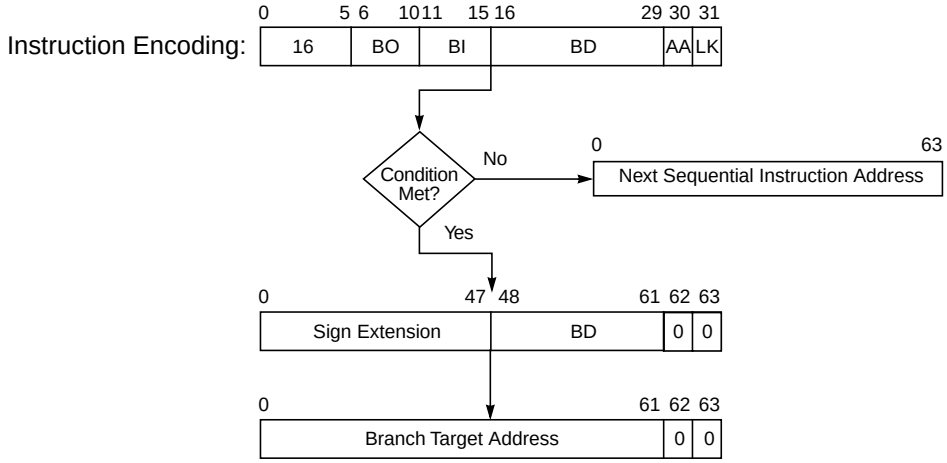

Figure 4-9. Branch Conditional to Absolute Addressing

4.2.4.1.5 Branch Conditional to Link Register Addressing Mode

If the branch conditions are met, the branch conditional to link register instruction generates the next instruction address by fetching the contents of the LR and clearing the two low-order bits to zero. The link register update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This option causes the effective address of the instruction following the branch instruction to be placed in the LR.

Figure 4-10 shows how the branch target address is generated when using the branch conditional to link register addressing mode.

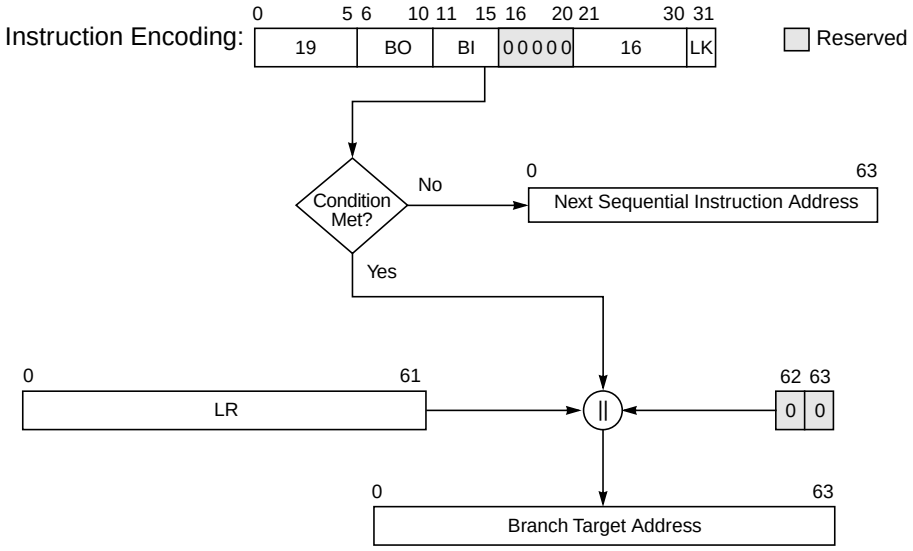

Figure 4-10. Branch Conditional to Link Register Addressing

4.2.4.1.6 Branch Conditional to Count Register Addressing Mode

If the branch conditions are met, the branch conditional to count register instruction generates the next instruction address by fetching the contents of the count register (CTR) and clearing the two low-order bits to zero. The link register update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This option causes the effective address of the instruction following the branch instruction to be placed in the LR.

Figure 4-11 shows how the branch target address is generated when using the branch conditional to count register addressing mode.

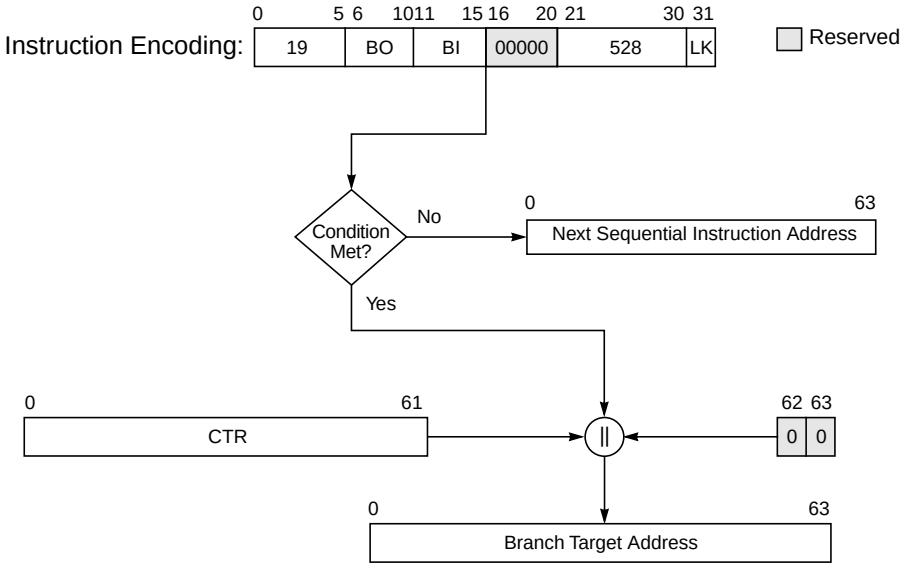

Figure 4-11. Branch Conditional to Count Register Addressing

4.2.4.2 Conditional Branch Control

For branch conditional instructions, the BO operand specifies the conditions under which the branch is taken. The first four bits of the BO operand specify how the branch is affected by or affects the condition and count registers. The fifth bit, shown in Table 4-20 as having the value *y*, is used by some PowerPC implementations for branch prediction as described below.

The encodings for the BO operands are shown in Table 4-20.

Table 4-20. BO Operand Encodings

BO	Description
0000*y*	Decrement the CTR, then branch if the decremented CTR ≠ 0 and the condition is FALSE.
0001*y*	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is FALSE.
001*zy*	Branch if the condition is FALSE.
0100*y*	Decrement the CTR, then branch if the decremented CTR ≠ 0 and the condition is TRUE.
0101*y*	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is TRUE.
011*zy*	Branch if the condition is TRUE.
1*z*00*y*	Decrement the CTR, then branch if the decremented CTR ≠ 0.
1*z*01*y*	Decrement the CTR, then branch if the decremented CTR = 0.
1*z*1*zz*	Branch always.

In this table, *z* indicates a bit that is ignored.
Note that the *z* bits should be cleared, as they may be assigned a meaning in some future version of the PowerPC architecture.

The *y* bit provides a hint about whether a conditional branch is likely to be taken, and may be used by some PowerPC implementations to improve performance.

The branch always encoding of the BO operand does not have a *y* bit.

Clearing the *y* bit indicates a predicted behavior for the branch instruction as follows:

- For **bc***x* with a negative value in the displacement operand, the branch is taken.
- In all other cases (**bc***x* with a non-negative value in the displacement operand, **bclr***x*, or **bcctr***x*), the branch is not taken.

Setting the *y* bit reverses the preceding indications.

The sign of the displacement operand is used as described above even if the target is an absolute address. The default value for the *y* bit should be 0, and should only be set to 1 if software has determined that the prediction corresponding to $y = 1$ is more likely to be correct than the prediction corresponding to $y = 0$. Software that does not compute branch predictions should clear the *y* bit.

In most cases, the branch should be predicted to be taken if the value of the following expression is 1, and predicted to fall through if the value is 0.

((BO[0] & BO[2]) | S) ≈ BO[4]

In the expression above, S (bit 16 of the branch conditional instruction coding) is the sign bit of the displacement operand if the instruction has a displacement operand and is 0 if the operand is reserved. BO[4] is the *y* bit, or 0 for the branch always encoding of the BO operand. (Advantage is taken of the fact that, for **bclr***x* and **bcctr***x*, bit 16 of the instruction is part of a reserved operand and therefore must be 0.)

The 5-bit BI operand in branch conditional instructions specifies which of the 32 bits in the CR represents the condition to test.

When the branch instructions contain immediate addressing operands, the target addresses can be computed sufficiently ahead of the branch instruction that instructions can be fetched along the target path. If the branch instructions use the link and count registers, instructions along the target path can be fetched if the link or count register is loaded sufficiently ahead of the branch instruction.

Branching can be conditional or unconditional, and optionally a branch return address is created by the access of the effective address of the instruction following the branch instruction in the LR after the branch target address has been computed. This is done regardless of whether the branch is taken. Some processors may keep a stack of the link register values most recently set by branch and link instructions, with the possible exception of the form shown below for obtaining the address of the next instruction. To benefit from this stack, the following programming conventions should be used.

In the following examples, let A, B, and Glue represent subroutine labels:

- Obtaining the address of the next instruction– use the following form of branch and link:

  bcl 20,31,$+4

- Loop counts:

  Keep them in the count register, and use one of the branch conditional instructions to decrement the count and to control branching (for example, branching back to the start of a loop if the decremented counter value is nonzero).

- Computed GOTOs, case statements, etc.:

  Use the count register to hold the address to branch to, and use the **bcctr** instruction with the link register option disabled (LK = 0) to branch to the selected address.

- Direct subroutine linkage—where A calls B and B returns to A. The two branches should be as follows:
  - — A calls B: use a branch instruction that enables the link register (LK = 1).
  - — B returns to A: use the **bclr** instruction with the link register option disabled (LK = 0) (the return address is in, or can be restored to, the link register).
- Indirect subroutine linkage:

  Where A calls Glue, Glue calls B, and B returns to A rather than to Glue. (Such a calling sequence is common in linkage code used when the subroutine that the programmer wants to call, here B, is in a different module from the caller: the binder inserts “glue” code to mediate the branch.) The three branches should be as follows:
  - — A calls Glue: use a branch instruction that sets the link register with the link register option enabled (LK = 1).
  - — Glue calls B: place the address of B in the count register, and use the **bcctr** instruction with the link register option disabled (LK = 0).
  - — B returns to A: use the **bclr** instruction with the link register option disabled (LK = 0) (the return address is in, or can be restored to, the link register).

4.2.4.3 Branch Instructions

Table 4-21 describes the branch instructions provided by the PowerPC processors.

Table 4-21. Branch Instructions

Name	Mnemonic	Operand Syntax	Operation
Branch	**b** **ba** **bl** **bla**	target_addr	**b** Branch. Branch to the address computed as the sum of the immediate address and the address of the current instruction. **ba** Branch Absolute. Branch to the absolute address specified. **bl** Branch then Link. Branch to the address computed as the sum of the immediate address and the address of the current instruction. The instruction address following this instruction is placed into the link register (LR). **bla** Branch Absolute then Link. Branch to the absolute address specified. The instruction address following this instruction is placed into the LR.
Branch Conditional	**bc** **bca** **bcl** **bcla**	BO,BI,target_addr	The BI operand specifies the bit in the CR to be used as the condition of the branch. The BO operand is used as described in Table 4-20. **bc** Branch Conditional. Branch conditionally to the address computed as the sum of the immediate address and the address of the current instruction. **bca** Branch Conditional Absolute. Branch conditionally to the absolute address specified. **bcl** Branch Conditional then Link. Branch conditionally to the address computed as the sum of the immediate address and the address of the current instruction. The instruction address following this instruction is placed into the LR. **bcla** Branch Conditional Absolute then Link. Branch conditionally to the absolute address specified. The instruction address following this instruction is placed into the LR.

Table 4-21. Branch Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Branch Conditional to Link Register	**bclr** **bclrl**	BO,BI	The BI operand specifies the bit in the CR to be used as the condition of the branch. The BO operand is used as described in Table 4-20. **bclr** Branch Conditional to Link Register. Branch conditionally to the address in the LR. **bclrl** Branch Conditional to Link Register then Link. Branch conditionally to the address specified in the LR. The instruction address following this instruction is then placed into the LR.
Branch Conditional to Count Register	**bcctr** **bcctrl**	BO,BI	The BI operand specifies the bit in the CR to be used as the condition of the branch. The BO operand is used as described in Table 4-20. **bcctr** Branch Conditional to Count Register. Branch conditionally to the address specified in the count register. **bcctrl** Branch Conditional to Count Register then Link. Branch conditionally to the address specified in the count register. The instruction address following this instruction is placed into the LR. **Note:** If the "decrement and test CTR" option is specified (BO[2] = 0), the instruction form is invalid.

4.2.4.4 Simplified Mnemonics for Branch Processor Instructions

To simplify assembly language programming, a set of simplified mnemonics and symbols is provided for the most frequently used forms of branch conditional, compare, trap, rotate and shift, and certain other instructions. See Appendix F, "Simplified Mnemonics," for a list of simplified mnemonic examples.

4.2.4.5 Condition Register Logical Instructions

Condition register logical instructions, shown in Table 4-22, and the Move Condition Register Field (**mcrf**) instruction are also defined as flow control instructions.

Note that if the LR update option is enabled for any of these instructions, the PowerPC architecture defines these forms of the instructions as invalid.

Table 4-22. Condition Register Logical Instructions

Name	Mnemonic	Operand Syntax	Operation
Condition Register AND	**crand**	**crb**D,**crb**A,**crb**B	The CR bit specified by **crb**A is ANDed with the CR bit specified by **crb**B. The result is placed into the CR bit specified by **crb**D.
Condition Register OR	**cror**	**crb**D,**crb**A,**crb**B	The CR bit specified by **crb**A is ORed with the CR bit specified by **crb**B. The result is placed into the CR bit specified by **crb**D.
Condition Register XOR	**crxor**	**crb**D,**crb**A,**crb**B	The CR bit specified by **crb**A is XORed with the CR bit specified by **crb**B. The result is placed into the CR bit specified by **crb**D.
Condition Register NAND	**crnand**	**crb**D,**crb**A,**crb**B	The CR bit specified by **crb**A is ANDed with the CR bit specified by **crb**B. The complemented result is placed into the CR bit specified by **crb**D.

Table 4-22. Condition Register Logical Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Condition Register NOR	**crnor**	**crb**D,**crb**A,**crb**B	The CR bit specified by **crb**A is ORed with the CR bit specified by **crb**B. The complemented result is placed into the CR bit specified by **crb**D.
Condition Register Equivalent	**creqv**	**crb**D,**crb**A, **crb**B	The CR bit specified by **crb**A is XORed with the CR bit specified by **crb**B. The complemented result is placed into the CR bit specified by **crb**D.
Condition Register AND with Complement	**crandc**	**crb**D,**crb**A, **crb**B	The CR bit specified by **crb**A is ANDed with the complement of the CR bit specified by **crb**B and the result is placed into the CR bit specified by **crb**D.
Condition Register OR with Complement	**crorc**	**crb**D,**crb**A, **crb**B	The CR bit specified by **crb**A is ORed with the complement of the CR bit specified by **crb**B and the result is placed into the CR bit specified by **crb**D.
Move Condition Register Field	**mcrf**	**crf**D,**crf**S	The contents of **crf**S are copied into **crf**D. No other condition register fields are changed.

4.2.4.6 Trap Instructions

The trap instructions shown in Table 4-23 are provided to test for a specified set of conditions. If any of the conditions tested by a trap instruction are met, the system trap handler is invoked. If the tested conditions are not met, instruction execution continues normally. See Appendix F, “Simplified Mnemonics,” for a complete set of simplified mnemonics.

Table 4-23. Trap Instructions

Name	Mnemonic	Operand Syntax	Operand Syntax
Trap Word Immediate	**twi**	TO,**r**A,SIMM	The contents of **r**A are compared with the sign-extended SIMM operand. If any bit in the TO operand is set and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.
Trap Word	**tw**	TO,**r**A,**r**B	The contents of **r**A are compared with the contents of **r**B. If any bit in the TO operand is set and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.