CPU Instruction Set Details

A

This appendix provides a detailed description of the operation of each R4000 instruction in both 32- and 64-bit modes. The instructions are listed in alphabetical order.

Exceptions that may occur due to the execution of each instruction are listed after the description of each instruction. Descriptions of the immediate cause and manner of handling exceptions are omitted from the instruction descriptions in this appendix.

Figures at the end of this appendix list the bit encoding for the constant fields of each instruction, and the bit encoding for each individual instruction is included with that instruction.

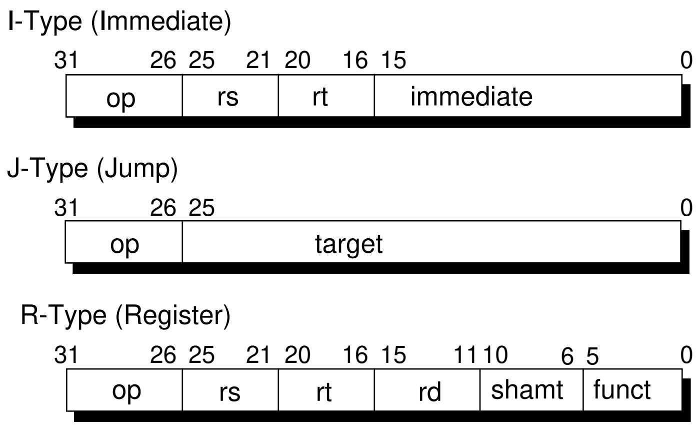
A.1 Instruction Classes

CPU instructions are divided into the following classes:

- **Load** and **Store** instructions move data between memory and general registers. They are all I-type instructions, since the only addressing mode supported is *base register + 16-bit immediate offse*t.
- **Computational** instructions perform arithmetic, logical and shift operations on values in registers. They occur in both R-type (both operands are registers) and I-type (one operand is a 16-bit immediate) formats.
- **Jump** and **Branch** instructions change the control flow of a program. Jumps are always made to absolute 26-bit word addresses (J-type format), or register addresses (R-type), for returns and dispatches. Branches have 16-bit offsets relative to the program counter (I-type). **Jump and Link** instructions save their return address in register *31*.
- **Coprocessor** instructions perform operations in the coprocessors. Coprocessor loads and stores are I-type. Coprocessor computational instructions have coprocessor-dependent formats (see the FPU instructions in Appendix B). Coprocessor zero (CP0) instructions manipulate the memory management and exception handling facilities of the processor.
- **Special** instructions perform a variety of tasks, including movement of data between special and general registers, trap, and breakpoint. They are always R-type.

A.2 Instruction Formats

Every CPU instruction consists of a single word (32 bits) aligned on a word boundary and the major instruction formats are shown in Figure A-1.



op	6-bit operation code
rs	5-bit source register specifier
rt	5-bit target (source/destination) or branch condition
immediate	16-bit immediate, branch displacement or address displacement
target	26-bit jump target address
rd	5-bit destination register specifier
shamt	5-bit shift amount
funct	6-bit function field

Figure A-1 CPU Instruction Formats

A.3 Instruction Notation Conventions

In this appendix, all variable subfields in an instruction format (such as *rs*, *rt*, *immediate*, etc.) are shown in lowercase names.

For the sake of clarity, we sometimes use an alias for a variable subfield in the formats of specific instructions. For example, we use *rs* = *base* in the format for load and store instructions. Such an alias is always lower case, since it refers to a variable subfield.

Figures with the actual bit encoding for all the mnemonics are located at the end of this Appendix, and the bit encoding also accompanies each instruction.

In the instruction descriptions that follow, the *Operation* section describes the operation performed by each instruction using a high-level language notation. The R4000 can operate as either a 32- or 64-bit microprocessor and the operation for both modes is included with the instruction description.

Special symbols used in the notation are described in Table A-1.

Table A-1 CPU Instruction Operation Notations

Symbol	Meaning
$\leftarrow$	Assignment.
||	Bit string concatenation.
x^y	Replication of bit value *x* into a *y*-bit string. Note: *x* is always a single-bit value.
$x_{y:z}$	Selection of bits *y* through *z* of bit string *x*. Little-endian bit notation is always used. If *y* is less than *z*, this expression is an empty (zero length) bit string.
+	2's complement or floating-point addition.
-	2's complement or floating-point subtraction.
*	2's complement or floating-point multiplication.
div	2's complement integer division.
mod	2's complement modulo.
/	Floating-point division.
<	2's complement less than comparison.
and	Bit-wise logical AND.
or	Bit-wise logical OR.
xor	Bit-wise logical XOR.
nor	Bit-wise logical NOR.
GPR[*x*]	General-Register x. The content of GPR[0] is always zero. Attempts to alter the content of GPR[0] have no effect.
CPR[*z,x*]	Coprocessor unit *z*, general register *x.*
CCR[*z,x*]	Coprocessor unit *z*, control register *x.*
COC[*z*]	Coprocessor unit *z* condition signal.
BigEndianMem	Big-endian mode as configured at reset (0 $\rightarrow$ Little, 1 $\rightarrow$ Big). Specifies the endianness of the memory interface (see LoadMemory and StoreMemory), and the endianness of Kernel and Supervisor mode execution.
ReverseEndian	Signal to reverse the endianness of load and store instructions. This feature is available in User mode only, and is effected by setting the *RE* bit of the *Status* register. Thus, ReverseEndian may be computed as (SR_{25} and User mode).
BigEndianCPU	The endianness for load and store instructions (0 $\rightarrow$ Little, 1 $\rightarrow$ Big). In User mode, this endianness may be reversed by setting SR_{25}. Thus, BigEndianCPU may be computed as BigEndianMem XOR ReverseEndian.
LLbit	Bit of state to specify synchronization instructions. Set by *LL*, cleared by *ERET* and *Invalidate* and read by *SC.*
T+*i*:	Indicates the time steps between operations. Each of the statements within a time step are defined to be executed in sequential order (as modified by conditional and loop constructs). Operations which are marked *T+i:* are executed at instruction cycle *i* relative to the start of execution of the instruction. Thus, an instruction which starts at time *j* executes operations marked T+*i:* at time *i + j*. The interpretation of the order of execution between two instructions or two operations which execute at the same time should be pessimistic; the order is not defined.

Instruction Notation Examples

The following examples illustrate the application of some of the instruction notation conventions:

Example #1:

$$GPR[rt] \leftarrow immediate \,||\, 0^{16}$$

Sixteen zero bits are concatenated with an immediate value (typically 16 bits), and the 32-bit string (with the lower 16 bits set to zero) is assigned to General-Purpose Register *rt*.

Example #2:

$$(immediate_{15})^{16} \,||\, immediate_{15...0}$$

Bit 15 (the sign bit) of an immediate value is extended for 16 bit positions, and the result is concatenated with bits 15 through 0 of the immediate value to form a 32-bit sign extended value.

A.4 Load and Store Instructions

In the R4000 implementation, the instruction immediately following a load may use the loaded contents of the register. In such cases, the hardware *interlocks*, requiring additional real cycles, so scheduling load delay slots is still desirable, although not required for functional code.

Two special instructions are provided in the R4000 implementation of the MIPS ISA, Load Linked and Store Conditional. These instructions are used in carefully coded sequences to provide one of several synchronization primitives, including test-and-set, bit-level locks, semaphores, and sequencers/event counts.

In the load and store descriptions, the functions listed in Table A-2 are used to summarize the handling of virtual addresses and physical memory.

Table A-2 Load and Store Common Functions

Function	Meaning
AddressTranslation	Uses the TLB to find the physical address given the virtual address. The function fails and an exception is taken if the required translation is not present in the TLB.
LoadMemory	Uses the cache and main memory to find the contents of the word containing the specified physical address. The low-order two bits of the address and the *Access Type* field indicates which of each of the four bytes within the data word need to be returned. If the cache is enabled for this access, the entire word is returned and loaded into the cache.
StoreMemory	Uses the cache, write buffer, and main memory to store the word or part of word specified as data in the word containing the specified physical address. The low-order two bits of the address and the *Access Type* field indicates which of each of the four bytes within the data word should be stored.

As shown in Table A-3, the *Access Type* field indicates the size of the data item to be loaded or stored. Regardless of access type or byte-numbering order (endianness), the address specifies the byte which has the smallest byte address in the addressed field. For a big-endian machine, this is the leftmost byte and contains the sign for a 2's complement number; for a little-endian machine, this is the rightmost byte.

Table A-3 Access Type Specifications for Loads/Stores

Access Type Mnemonic	Value	Meaning
DOUBLEWORD	7	8 bytes (64 bits)
SEPTIBYTE	6	7 bytes (56 bits)
SEXTIBYTE	5	6 bytes (48 bits)
QUINTIBYTE	4	5 bytes (40 bits)
WORD	3	4 bytes (32 bits)
TRIPLEBYTE	2	3 bytes (24 bits)
HALFWORD	1	2 bytes (16 bits)
BYTE	0	1 byte (8 bits)

The bytes within the addressed doubleword which are used can be determined directly from the access type and the three low-order bits of the address.

A.5 Jump and Branch Instructions

All jump and branch instructions have an architectural delay of exactly one instruction. That is, the instruction immediately following a jump or branch (that is, occupying the delay slot) is always executed while the target instruction is being fetched from storage. A delay slot may not itself be occupied by a jump or branch instruction; however, this error is not detected and the results of such an operation are undefined.

If an exception or interrupt prevents the completion of a legal instruction during a delay slot, the hardware sets the *EPC* register to point at the jump or branch instruction that precedes it. When the code is restarted, both the jump or branch instructions and the instruction in the delay slot are reexecuted.

Because jump and branch instructions may be restarted after exceptions or interrupts, they must be restartable. Therefore, when a jump or branch instruction stores a return link value, register *31* (the register in which the link is stored) may not be used as a source register.

Since instructions must be word-aligned, a **Jump Register** or **Jump and Link Register** instruction must use a register whose two low-order bits are zero. If these low-order bits are not zero, an address exception will occur when the jump target instruction is subsequently fetched.

A.6 Coprocessor Instructions

Coprocessors are alternate execution units, which have register files separate from the CPU. The MIPS architecture provides four coprocessor units, or classes, and these coprocessors have two register spaces, each space containing thirty-two 32-bit registers.

- The first space, *coprocessor general* registers, may be directly loaded from memory and stored into memory, and their contents may be transferred between the coprocessor and processor.
- The second space, *coprocessor control* registers, may only have their contents transferred directly between the coprocessor and the processor. Coprocessor instructions may alter registers in either space.

A.7 System Control Coprocessor (CP0) Instructions

There are some special limitations imposed on operations involving CP0 that is incorporated within the CPU. Although load and store instructions to transfer data to/from coprocessors and to move control to/from coprocessor instructions are generally permitted by the MIPS architecture, CP0 is given a somewhat protected status since it has responsibility for exception handling and memory management. Therefore, the move to/from coprocessor instructions are the only valid mechanism for writing to and reading from the CP0 registers.

Several CP0 instructions are defined to directly read, write, and probe TLB entries and to modify the operating modes in preparation for returning to User mode or interrupt-enabled states.

ADD — Add — ADD

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	ADD 1 0 0 0 0 0
6	5	5	5	5	6

Format:

ADD rd, rs, rt

Description:

The contents of general register *rs* and the contents of general register *rt* are added to form the result. The result is placed into general register *rd*. In 64-bit mode, the operands must be valid sign-extended, 32-bit values.

An overflow exception occurs if the carries out of bits 30 and 31 differ (2's complement overflow). The destination register *rd* is not modified when an integer overflow exception occurs.

Operation:

32	T:	$GPR[rd] \leftarrow GPR[rs] + GPR[rt]$
64	T:	$temp \leftarrow GPR[rs] + GPR[rt]$
		$GPR[rd] \leftarrow (temp_{31})^{32} \parallel temp_{31...0}$

Exceptions:

Integer overflow exception

ADDI — Add Immediate — ADDI

31 … 26	25 … 21	20 … 16	15 … 0
ADDI 0 0 1 0 0 0	rs	rt	immediate
6	5	5	16

Format:

ADDI rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and added to the contents of general register *rs* to form the result. The result is placed into general register *rt*. In 64-bit mode, the operand must be valid sign-extended, 32-bit values.

An overflow exception occurs if carries out of bits 30 and 31 differ (2's complement overflow). The destination register *rt* is not modified when an integer overflow exception occurs.

Operation:

32 T: $\text{GPR [rt]} \leftarrow \text{GPR[rs]} + (\text{immediate}_{15})^{16} \mid\mid \text{immediate}_{15...0}$

64 T: $\text{temp} \leftarrow \text{GPR[rs]} + (\text{immediate}_{15})^{48} \mid\mid \text{immediate}_{15...0}$
$\text{GPR[rt]} \leftarrow (\text{temp}_{31})^{32} \mid\mid \text{temp}_{31...0}$

Exceptions:

Integer overflow exception

ADDIU Add Immediate Unsigned ADDIU

31 — 26	25 — 21	20 — 16	15 — 0
ADDIU 0 0 1 0 0 1	rs	rt	immediate
6	5	5	16

Format:

ADDIU rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and added to the contents of general register *rs* to form the result. The result is placed into general register *rt.* No integer overflow exception occurs under any circumstances. In 64-bit mode, the operand must be valid sign-extended, 32-bit values.

The only difference between this instruction and the ADDI instruction is that ADDIU never causes an overflow exception.

Operation:

32	T:	$GPR[rt] \leftarrow GPR[rs] + (immediate_{15})^{16} \,\|\|\, immediate_{15...0}$
64	T:	$temp \leftarrow GPR[rs] + (immediate_{15})^{48} \,\|\|\, immediate_{15...0}$ $GPR[rt] \leftarrow (temp_{31})^{32} \,\|\|\, temp_{31...0}$

Exceptions:

None

ADDU Add Unsigned ADDU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	ADDU 1 0 0 0 0 1
6	5	5	5	5	6

Format:

ADDU rd, rs, rt

Description:

The contents of general register *rs* and the contents of general register *rt* are added to form the result. The result is placed into general register *rd*. No overflow exception occurs under any circumstances. In 64-bit mode, the operands must be valid sign-extended, 32-bit values.

The only difference between this instruction and the ADD instruction is that ADDU never causes an overflow exception.

Operation:

32	T:	$GPR[rd] \leftarrow GPR[rs] + GPR[rt]$
64	T:	$temp \leftarrow GPR[rs] + GPR[rt]$
		$GPR[rd] \leftarrow (temp_{31})^{32} \parallel temp_{31...0}$

Exceptions:

None

AND And AND

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	AND 1 0 0 1 0 0
6	5	5	5	5	6

Format:

AND rd, rs, rt

Description:

The contents of general register *rs* are combined with the contents of general register *rt* in a bit-wise logical AND operation. The result is placed into general register *rd*.

Operation:

32	T:	GPR[rd] $\leftarrow$ GPR[rs] and GPR[rt]
64	T:	GPR[rd] $\leftarrow$ GPR[rs] and GPR[rt]

Exceptions:

None

ANDI And Immediate ANDI

31 26	25 21	20 16	15 0
ANDI 0 0 1 1 0 0	rs	rt	immediate
6	5	5	16

Format:

ANDI rt, rs, immediate

Description:

The 16-bit *immediate* is zero-extended and combined with the contents of general register *rs* in a bit-wise logical AND operation. The result is placed into general register *rt*.

Operation:

32	T:	$GPR[rt] \leftarrow 0^{16} \mid\mid (immediate\ and\ GPR[rs]_{15...0})$
64	T:	$GPR[rt] \leftarrow 0^{48} \mid\mid (immediate\ and\ GPR[rs]_{15...0})$

Exceptions:

None

BCzF Branch On Coprocessor z False BCzF

31 26	25 21	20 16	15 0
COPz 0 1 0 0 x x*	BC 0 1 0 0 0	BCF 0 0 0 0 0	offset
6	5	5	16

Format:

BCzF offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If coprocessor *z*'s condition signal (CpCond), as sampled during the previous instruction, is false, then the program branches to the target address with a delay of one instruction.

Because the condition line is sampled during the previous instruction, there must be at least one instruction between this instruction and a coprocessor instruction that changes the condition line.

Operation:

```
32   T–1:  condition ← not COC[z]
     T:    target ← (offset_15)^14 || offset || 0^2
     T+1:  if condition then
                      PC ← PC + target
           endif
64   T–1:  condition ← not COC[z]
     T:    target ← (offset_15)^46 || offset || 0^2
     T+1:  if condition then
                      PC ← PC + target
           endif
```

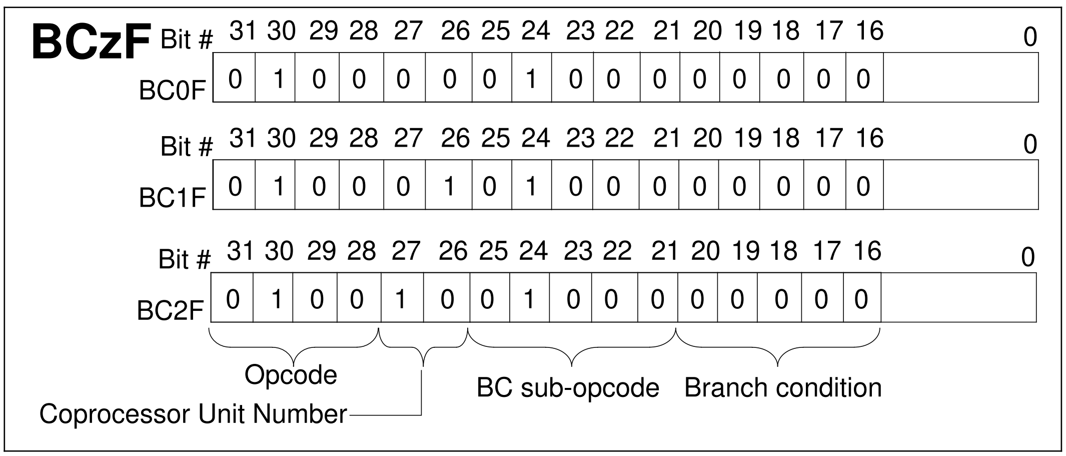
*See the table "Opcode Bit Encoding" on next page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

BCzF Branch On Coprocessor z False (continued) BCzF

Exceptions:

Coprocessor unusable exception

Opcode Bit Encoding:



BCzFL Branch On Coprocessor z False Likely BCzFL

31 26	25 21	20 16	15 0
COPz 0 1 0 0 x x*	BC 0 1 0 0 0	BCFL 0 0 0 1 0	offset
6	5	5	16

Format:

BCzFL offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of coprocessor *z*'s condition line, as sampled during the previous instruction, is false, the target address is branched to with a delay of one instruction.

If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

Because the condition line is sampled during the previous instruction, there must be at least one instruction between this instruction and a coprocessor instruction that changes the condition line.

*See the table "Opcode Bit Encoding" on next page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

BCzFL Branch On Coprocessor z False Likely (continued) BCzFL

Operation:

32	T–1:	condition ← not COC[z]		
	T:	target ← $(\text{offset}_{15})^{14}$ || offset || 0^2		
	T+1:	if condition then		
			PC ← PC + target	
		else		
			NullifyCurrentInstruction	
		endif		
64	T–1:	condition ← not COC[z]		
	T:	target ← $(\text{offset}_{15})^{46}$ || offset || 0^2		
	T+1:	if condition then		
			PC ← PC + target	
		else		
			NullifyCurrentInstruction	
		endif		

Exceptions:

Coprocessor unusable exception

Opcode Bit Encoding:

BCzFL

Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC0FL	0	1	0	0	0	0	0	1	0	0	0	0	0	0	1	0	

Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC1FL	0	1	0	0	0	1	0	1	0	0	0	0	0	0	1	0	

Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC2FL	0	1	0	0	1	0	0	1	0	0	0	0	0	0	1	0	

Opcode (bits 31–28) · Coprocessor Unit Number (bits 27–26) · BC sub-opcode (bits 25–21) · Branch condition (bits 20–16)

BCzT Branch On Coprocessor z True BCzT

31 — 26	25 — 21	20 — 16	15 — 0
COPz 0 1 0 0 x x*	BC 0 1 0 0 0	BCT 0 0 0 0 1	offset
6	5	5	16

Format:

BCzT offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the coprocessor *z*'s condition signal (CpCond) is true, then the program branches to the target address, with a delay of one instruction.

Because the condition line is sampled during the previous instruction, there must be at least one instruction between this instruction and a coprocessor instruction that changes the condition line.

Operation:

32 T–1: condition ← COC[z]
T: target ← $(\text{offset}_{15})^{14}$ || offset || 0^2
T+1: if condition then
PC ← PC + target
endif

64 T–1: condition ← COC[z]
T: target ← $(\text{offset}_{15})^{46}$ || offset || 0^2
T+1: if condition then
PC ← PC + target
endif

*See the table "Opcode Bit Encoding" on next page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

BCzT — Branch On Coprocessor z True (continued) — BCzT

Exceptions:

Coprocessor unusable exception

Opcode Bit Encoding:

BCzT

Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC0T	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1	

Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC1T	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	1	

Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC2T	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	1	

Opcode (bits 31–28) · Coprocessor Unit Number (bits 27–26) · BC sub-opcode (bits 25–21) · Branch condition (bits 20–16)

BCzTL — Branch On Coprocessor z True Likely — BCzTL

31–26	25–21	20–16	15–0
COPz 0 1 0 0 x x*	BC 0 1 0 0 0	BCTL 0 0 0 1 1	offset
6	5	5	16

Format:

BCzTL offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of coprocessor *z*'s condition line, as sampled during the previous instruction, is true, the target address is branched to with a delay of one instruction.

If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

Because the condition line is sampled during the previous instruction, there must be at least one instruction between this instruction and a coprocessor instruction that changes the condition line.

Operation:

```
32   T–1:  condition ← COC[z]
     T:    target ← (offset15)^14 || offset || 0^2
     T+1:  if condition then
                  PC ← PC + target
           else
                  NullifyCurrentInstruction
           endif
64   T–1:  condition ← COC[z]
     T:    target ← (offset15)^46 || offset || 0^2
     T+1:  if condition then
                  PC ← PC + target
           else
                  NullifyCurrentInstruction
           endif
```

*See the table "Opcode Bit Encoding" on next page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

BCzTL — Branch On Coprocessor z True Likely (continued) — BCzTL

Exceptions:

Coprocessor unusable exception

Opcode Bit Encoding:

BCzTL

Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC0TL	0	1	0	0	0	0	0	1	0	0	0	0	0	0	1	1	
BC1TL	0	1	0	0	0	1	0	1	0	0	0	0	0	0	1	1	
BC2TL	0	1	0	0	1	0	0	1	0	0	0	0	0	0	1	1	

Opcode (bits 31–28) · Coprocessor Unit Number (bits 27–26) · BC sub-opcode (bits 25–21) · Branch condition (bits 20–16)

BEQ — Branch On Equal — BEQ

31–26	25–21	20–16	15–0
BEQ 0 0 0 1 0 0	rs	rt	offset
6	5	5	16

Format:

BEQ rs, rt, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. The contents of general register *rs* and the contents of general register *rt* are compared. If the two registers are equal, then the program branches to the target address, with a delay of one instruction.

Operation:

32 T: $\text{target} \leftarrow (\text{offset}_{15})^{14} \,||\, \text{offset} \,||\, 0^2$

condition $\leftarrow$ (GPR[rs] = GPR[rt])

T+1: if condition then

PC $\leftarrow$ PC + target

endif

64 T: $\text{target} \leftarrow (\text{offset}_{15})^{46} \,||\, \text{offset} \,||\, 0^2$

condition $\leftarrow$ (GPR[rs] = GPR[rt])

T+1: if condition then

PC $\leftarrow$ PC + target

endif

Exceptions:

None

BEQL — Branch On Equal Likely — BEQL

31 26	25 21	20 16	15 0
BEQL 0 1 0 1 0 0	rs	rt	offset
6	5	5	16

Format:

BEQL rs, rt, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. The contents of general register *rs* and the contents of general register *rt* are compared. If the two registers are equal, the target address is branched to, with a delay of one instruction. If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

Operation:

32 T: $\text{target} \leftarrow (\text{offset}_{15})^{14} \,||\, \text{offset} \,||\, 0^2$
condition $\leftarrow$ (GPR[rs] = GPR[rt])
T+1: if condition then
PC $\leftarrow$ PC + target
else
NullifyCurrentInstruction
endif

64 T: $\text{target} \leftarrow (\text{offset}_{15})^{46} \,||\, \text{offset} \,||\, 0^2$
condition $\leftarrow$ (GPR[rs] = GPR[rt])
T+1: if condition then
PC $\leftarrow$ PC + target
else
NullifyCurrentInstruction
endif

Exceptions:

None

BGEZ — Branch On Greater Than Or Equal To Zero — BGEZ

31–26	25–21	20–16	15–0
REGIMM 0 0 0 0 0 1	rs	BGEZ 0 0 0 0 1	offset
6	5	5	16

Format:

BGEZ rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of general register *rs* have the sign bit cleared, then the program branches to the target address, with a delay of one instruction.

Operation:

32 T: $\text{target} \leftarrow (\text{offset}_{15})^{14} \parallel \text{offset} \parallel 0^2$
condition $\leftarrow (\text{GPR}[rs]_{31} = 0)$
T+1: if condition then
PC $\leftarrow$ PC + target
endif

64 T: $\text{target} \leftarrow (\text{offset}_{15})^{46} \parallel \text{offset} \parallel 0^2$
condition $\leftarrow (\text{GPR}[rs]_{63} = 0)$
T+1: if condition then
PC $\leftarrow$ PC + target
endif

Exceptions:

None