10.3 Instructions: The MIPS core subset

(Original section[1])

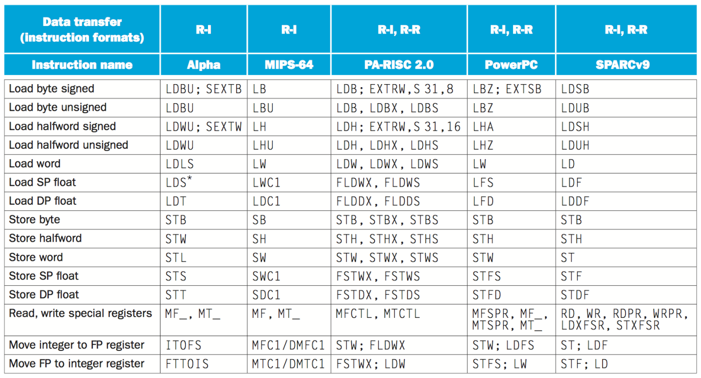
The similarities of each architecture allow simultaneous descriptions, starting with the operations equivalent to the MIPS core.

MIPS core instructions

Almost every instruction found in the MIPS core is found in the other architectures. Instructions are listed under four categories: data transfer (COD Figure D.3.1 (Desktop RISC data transfer instructions ...)); arithmetic/logical (COD Figure D.3.2 (Desktop RISC arithmetic/logical instructions ...)); control (COD Figure D.3.3 (Desktop RISC control instructions ...)); and floating point (COD Figure D.3.4 (Desktop RISC floating-point instructions ...)). A fifth category (COD Figure D.3.5 (Conventions of desktop RISC architectures ...)) shows conventions for register usage and pseudoinstructions on each architecture. If a MIPS core instruction requires a short sequence of instructions in other architectures, these instructions are separated by semicolons. (To avoid confusion, the destination register will always be the leftmost operand in this appendix, independent of the notation normally used with each architecture.)

Figure 10.3.1: Desktop RISC data transfer instructions equivalent to MIPS core (COD Figure D.3.1).

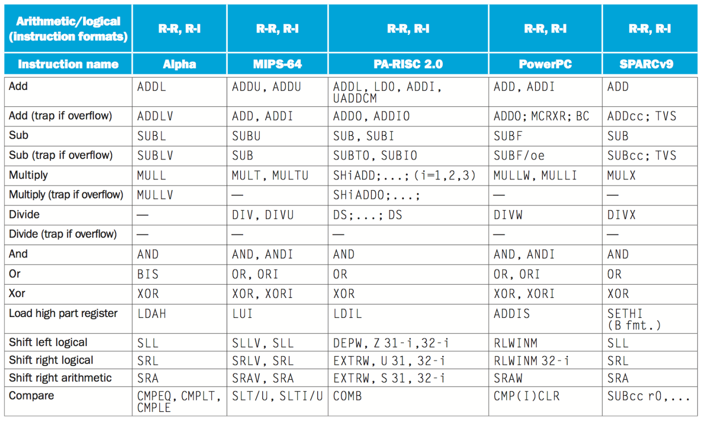
A sequence of instructions to synthesize a MIPS instruction is shown separated by semicolons. If there are several choices of instructions equivalent to MIPS core, they are separated by commas. For this figure, halfword is 16 bits and word is 32 bits. Note that in Alpha, LDS converts single-precision floating point to double precision and loads the entire 64-bit register.



Data transfer (instruction formats)	R-I	R-I	R-I, R-R	R-I, R-R	R-I, R-R
Instruction name	**Alpha**	**MIPS-64**	**PA-RISC 2.0**	**PowerPC**	**SPARCv9**
Load byte signed	LDBU; SEXTB	LB	LDB; EXTRW,S 31,8	LBZ; EXTSB	LDSB
Load byte unsigned	LDBU	LBU	LDB, LDBX, LDBS	LBZ	LDUB
Load halfword signed	LDWU; SEXTW	LH	LDH; EXTRW,S 31,16	LHA	LDSH
Load halfword unsigned	LDWU	LHU	LDH, LDHX, LDHS	LHZ	LDUH
Load word	LDLS	LW	LDW, LDWX, LDWS	LW	LD
Load SP float	LDS*	LWC1	FLDWX, FLDWS	LFS	LDF
Load DP float	LDT	LDC1	FLDDX, FLDDS	LFD	LDDF
Store byte	STB	SB	STB, STBX, STBS	STB	STB
Store halfword	STW	SH	STH, STHX, STHS	STH	STH
Store word	STL	SW	STW, STWX, STWS	STW	ST
Store SP float	STS	SWC1	FSTWX, FSTWS	STFS	STF
Store DP float	STT	SDC1	FSTDX, FSTDS	STFD	STDF
Read, write special registers	MF_, MT_	MF, MT_	MFCTL, MTCTL	MFSPR, MF_, MTSPR, MT_	RD, WR, RDPR, WRPR, LDXFSR, STXFSR
Move integer to FP register	ITOFS	MFC1/DMFC1	STW; FLDWX	STW; LDFS	ST; LDF
Move FP to integer register	FTTOIS	MTC1/DMTC1	FSTWX; LDW	STFS; LW	STF; LD

Figure 10.3.2: Desktop RISC arithmetic/logical instructions equivalent to MIPS core (COD Figure D.3.2).

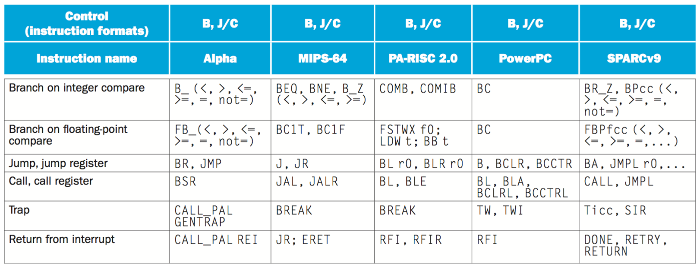
Dashes mean the operation is not available in that architecture, or not synthesized in a few instructions. Such a sequence of instructions is shown separated by semicolons. If there are several choices of instructions equivalent to MIPS core, they are separated by commas. Note that in the "Arithmetic/logical" category, all machines but SPARC use separate instructions mnemonics to indicate an immediate operand; SPARC offers immediate versions of these instructions but uses a single mnemonic. (Of course these are separate opcodes!).



Arithmetic/logical (instruction formats)	R-R, R-I	R-R, R-I	R-R, R-I	R-R, R-I	R-R, R-I
Instruction name	**Alpha**	**MIPS-64**	**PA-RISC 2.0**	**PowerPC**	**SPARCv9**
Add	ADDL	ADDU, ADDU	ADDL, LDO, ADDI, UADDCM	ADD, ADDI	ADD
Add (trap if overflow)	ADDLV	ADD, ADDI	ADDO, ADDIO	ADDO; MCRXR; BC	ADDcc; TVS
Sub	SUBL	SUBU	SUB, SUBI	SUBF	SUB
Sub (trap if overflow)	SUBLV	SUB	SUBTO, SUBIO	SUBF/oe	SUBcc; TVS
Multiply	MULL	MULT, MULTU	SHiADD;...; (i=1,2,3)	MULLW, MULLI	MULX
Multiply (trap if overflow)	MULLV	—	SHiADDO;...;	—	—
Divide	—	DIV, DIVU	DS;...; DS	DIVW	DIVX
Divide (trap if overflow)	—	—	—	—	—
And	AND	AND, ANDI	AND	AND, ANDI	AND
Or	BIS	OR, ORI	OR	OR, ORI	OR
Xor	XOR	XOR, XORI	XOR	XOR, XORI	XOR
Load high part register	LDAH	LUI	LDIL	ADDIS	SETHI (B fmt.)
Shift left logical	SLL	SLLV, SLL	DEPW, Z 31-i,32-i	RLWINM	SLL
Shift right logical	SRL	SRLV, SRL	EXTRW, U 31, 32-i	RLWINM 32-i	SRL
Shift right arithmetic	SRA	SRAV, SRA	EXTRW, S 31, 32-i	SRAW	SRA
Compare	CMPEQ, CMPLT, CMPLE	SLT/U, SLTI/U	COMB	CMP(I)CLR	SUBcc r0,...

Figure 10.3.3: Desktop RISC control instructions equivalent to MIPS core (COD Figure D.3.3).

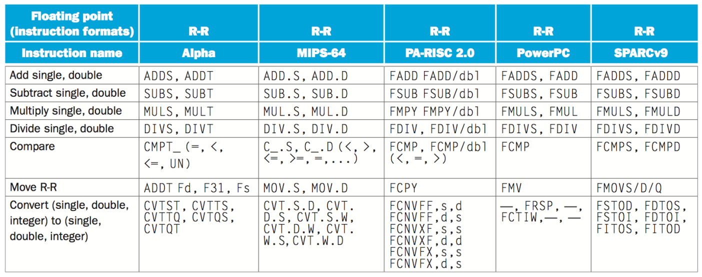
If there are several choices of instructions equivalent to MIPS core, they are separated by commas.



Control (instruction formats)	B, J/C	B, J/C	B, J/C	B, J/C	B, J/C
Instruction name	**Alpha**	**MIPS-64**	**PA-RISC 2.0**	**PowerPC**	**SPARCv9**
Branch on integer compare	`B_ (<, >, <=, >=, =, not=)`	`BEQ, BNE, B_Z (<, >, <=, >=)`	`COMB, COMIB`	`BC`	`BR_Z, BPcc (<, >, <=, >=, =, not=)`
Branch on floating-point compare	`FB_(<, >, <=, >=, =, not=)`	`BC1T, BC1F`	`FSTWX f0; LDW t; BB t`	`BC`	`FBPfcc (<, >, <=, >=, =,...)`
Jump, jump register	`BR, JMP`	`J, JR`	`BL r0, BLR r0`	`B, BCLR, BCCTR`	`BA, JMPL r0,...`
Call, call register	`BSR`	`JAL, JALR`	`BL, BLE`	`BL, BLA, BCLRL, BCCTRL`	`CALL, JMPL`
Trap	`CALL_PAL GENTRAP`	`BREAK`	`BREAK`	`TW, TWI`	`Ticc, SIR`
Return from interrupt	`CALL_PAL REI`	`JR; ERET`	`RFI, RFIR`	`RFI`	`DONE, RETRY, RETURN`

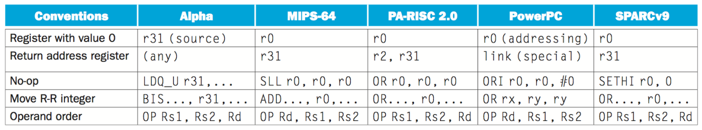
Figure 10.3.4: Desktop RISC floating-point instructions equivalent to MIPS core (COD Figure D.3.4).

Dashes mean the operation is not available in that architecture, or not synthesized in a few instructions. If there are several choices of instructions equivalent to MIPS core, they are separated by commas.



Floating point (instruction formats)	R-R	R-R	R-R	R-R	R-R
Instruction name	**Alpha**	**MIPS-64**	**PA-RISC 2.0**	**PowerPC**	**SPARCv9**
Add single, double	`ADDS, ADDT`	`ADD.S, ADD.D`	`FADD FADD/dbl`	`FADDS, FADD`	`FADDS, FADDD`
Subtract single, double	`SUBS, SUBT`	`SUB.S, SUB.D`	`FSUB FSUB/dbl`	`FSUBS, FSUB`	`FSUBS, FSUBD`
Multiply single, double	`MULS, MULT`	`MUL.S, MUL.D`	`FMPY FMPY/dbl`	`FMULS, FMUL`	`FMULS, FMULD`
Divide single, double	`DIVS, DIVT`	`DIV.S, DIV.D`	`FDIV, FDIV/dbl`	`FDIVS, FDIV`	`FDIVS, FDIVD`
Compare	`CMPT_ (=, <, <=, UN)`	`C_.S, C_.D (<, >, <=, >=, =,...)`	`FCMP, FCMP/dbl (<, =, >)`	`FCMP`	`FCMPS, FCMPD`
Move R-R	`ADDT Fd, F31, Fs`	`MOV.S, MOV.D`	`FCPY`	`FMV`	`FMOVS/D/Q`
Convert (single, double, integer) to (single, double, integer)	`CVTST, CVTTS, CVTTQ, CVTQS, CVTQT`	`CVT.S.D, CVT.D.S, CVT.S.W, CVT.D.W, CVT.W.S, CVT.W.D`	`FCNVFF,s,d FCNVFF,d,s FCNVXF,s,s FCNVXF,d,d FCNVFX,s,s FCNVFX,d,s`	`—, FRSP, —, FCTIW,—, —`	`FSTOD, FDTOS, FSTOI, FDTOI, FITOS, FITOD`

Figure 10.3.5: Conventions of desktop RISC architectures equivalent to MIPS core (COD Figure D.3.5).

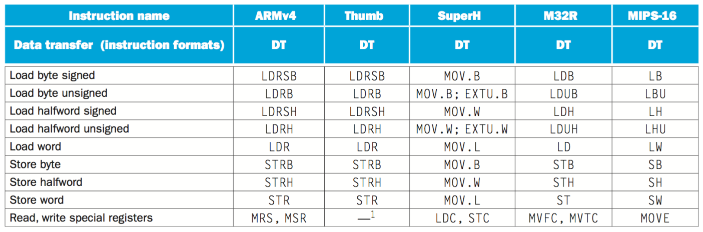


Conventions	Alpha	MIPS-64	PA-RISC 2.0	PowerPC	SPARCv9
Register with value 0	`r31 (source)`	`r0`	`r0`	`r0 (addressing)`	`r0`
Return address register	`(any)`	`r31`	`r2, r31`	`link (special)`	`r31`
No-op	`LDQ_U r31,...`	`SLL r0, r0, r0`	`OR r0, r0, r0`	`ORI r0, r0, #0`	`SETHI r0, 0`
Move R-R integer	`BIS..., r31,...`	`ADD..., r0,...`	`OR..., r0,...`	`OR rx, ry, ry`	`OR..., r0,...`
Operand order	`OP Rs1, Rs2, Rd`	`OP Rd, Rs1, Rs2`	`OP Rs1, Rs2, Rd`	`OP Rd, Rs1, Rs2`	`OP Rs1, Rs2, Rd`

Subsequent figures show the equivalent listing for embedded RISCs. Note that floating point is generally not defined for the embedded RISCs.

Figure 10.3.6: Embedded RISC data transfer instructions equivalent to MIPS core (COD Figure D.3.6).

A sequence of instructions to synthesize a MIPS instruction is shown separated by semicolons. Note that floating point is generally not defined for the embedded RISCs. Thumb and MIPS-16 are just 16-bit instruction subsets of the ARM and MIPS architectures, so machines can switch modes and execute the full instruction set. We use —^{1} to show sequences that are available in 32-bit mode but not 16-bit mode in Thumb or MIPS-16.



Instruction name	ARMv4	Thumb	SuperH	M32R	MIPS-16
Data transfer (instruction formats)	**DT**	**DT**	**DT**	**DT**	**DT**
Load byte signed	`LDRSB`	`LDRSB`	`MOV.B`	`LDB`	`LB`
Load byte unsigned	`LDRB`	`LDRB`	`MOV.B; EXTU.B`	`LDUB`	`LBU`
Load halfword signed	`LDRSH`	`LDRSH`	`MOV.W`	`LDH`	`LH`
Load halfword unsigned	`LDRH`	`LDRH`	`MOV.W; EXTU.W`	`LDUH`	`LHU`
Load word	`LDR`	`LDR`	`MOV.L`	`LD`	`LW`
Store byte	`STRB`	`STRB`	`MOV.B`	`STB`	`SB`
Store halfword	`STRH`	`STRH`	`MOV.W`	`STH`	`SH`
Store word	`STR`	`STR`	`MOV.L`	`ST`	`SW`
Read, write special registers	`MRS, MSR`	—^{1}	`LDC, STC`	`MVFC, MVTC`	`MOVE`

Figure 10.3.7: Embedded RISC arithmetic/logical instructions equivalent to MIPS core (COD Figure D.3.7).

Dashes mean the operation is not available in that architecture, or not synthesized in a few instructions. Such a sequence of instructions is shown separated by semicolons. If there are several choices of instructions equivalent to MIPS core, they are separated by commas. Thumb and MIPS-16 are just 16-bit instruction subsets of the ARM and MIPS architectures, so machines can switch modes and execute the full instruction set. We use —^{1} to show sequences that are available in 32-bit mode but not 16-bit mode in Thumb or MIPS-16. The superscript 2 shows new instructions found only in 16-bit mode of Thumb or MIPS-16, such as CMP/I^2 . ARM includes shifts as part of every data operation instruction, so the shifts with superscript 3 are just a variation of a move instruction, such as LSR^3.

Arithmetic/logical (instruction formats)	R-R, R-I	R-R, R-I	R-R, R-I	R-R, R-I	R-R, R-I
Instruction name	**ARMv4**	**Thumb**	**SuperH**	**M32R**	**MIPS-16**
Add	ADD	ADD	ADD	ADD, ADDI, ADD3	ADDU, ADDIU
Add (trap if overflow)	ADDS; SWIVS	ADD; BVC .+4; SWI	ADDV	ADDV, ADDV3	—^{1}
Subtract	SUB	SUB	SUB	SUB	SUBU
Subtract (trap if overflow)	SUBS; SWIVS	SUB; BVC .+1; SWI	SUBV	SUBV	—^{1}
Multiply	MUL	MUL	MUL	MUL	MULT, MULTU
Multiply (trap if overflow)					—
Divide	—	—	DIV1, DIVoS, DIVoU	DIV, DIVU	DIV, DIVU
Divide (trap if overflow)	—	—			—
And	AND	AND	AND	AND, AND3	AND
Or	ORR	ORR	OR	OR, OR3	OR
Xor	EOR	EOR	XOR	XOR, XOR3	XOR
Load high part register	—	—		SETH	—^{1}
Shift left logical	LSL^3	LSL^2	SHLL, SHLLn	SLL, SLLI, SLL3	SLLV, SLL
Shift right logical	LSR^3	LSR^2	SHRL, SHRLn	SRL, SRLI, SRL3	SRLV, SRL
Shift right arithmetic	ASR^3	ASR^2	SHRA, SHAD	SRA, SRAI, SRA3	SRAV, SRA
Compare	CMP,CMN, TST,TEQ	CMP, CMN, TST	CMP/cond, TST	CMP/I, CMPU/I	CMP/I^2, SLT/I, SLT/IU

Figure 10.3.8: Embedded RISC control instructions equivalent to MIPS core (COD Figure D.3.8).

Thumb and MIPS-16 are just 16-bit instruction subsets of the ARM and MIPS architectures, so machines can switch modes and execute the full instruction set. We use —^{1} to show sequences that are available in 32-bit mode but not 16-bit mode in Thumb or MIPS-16. The superscript 2 shows new instructions found only in 16-bit mode of Thumb or MIPS-16, such as BTEQZ^2.

Control (instruction formats)	B, J, C	B, J, C	B, J, C	B, J, C	B, J, C
Instruction name	**ARMv4**	**Thumb**	**SuperH**	**M32R**	**MIPS-16**
Branch on integer compare	B/cond	B/cond	BF, BT	BEQ, BNE, BC, BNC, B__Z	BEQZ^2, BNEZ^2, BTEQZ^2, BTNEZ^2
Jump, jump register	MOV pc, ri	MOV pc, ri	BRA, JMP	BRA, JMP	B^2, JR
Call, call register	BL	BL	BSR, JSR	BL, JL	JAL, JALR, JALX^2
Trap	SWI	SWI	TRAPA	TRAP	BREAK
Return from interrupt	MOVS pc, r14	—^{1}	RTS	RTE	—^{1}

Figure 10.3.9: Conventions of embedded RISC instructions equivalent to MIPS core (COD Figure D.3.9).

Conventions	ARMv4	Thumb	SuperH	M32R	MIPS-16
Return address reg.	R14	R14	PR (special)	R14	RA (special)
No-op	MOV r0, r0	MOV r0, r0	NOP	NOP	SLL r0, r0
Operands, order	OP Rd, Rs1, Rs2	OP Rd, Rs1	OP Rs1, Rd	OP Rd, Rs1	OP Rd, Rs1, Rs2

Every architecture must have a scheme for compare and conditional branch, but despite all the similarities, each of these architectures has found a different way to perform the operation.

Compare and conditional branch

SPARC uses the traditional four condition code bits stored in the program status word: *negative*, *zero*, *carry*, and *overflow*. They can be set on any arithmetic or logical instruction; unlike earlier architectures, this setting is optional on each instruction. An explicit option leads to fewer problems in pipelined implementation. Although condition codes can be set as a side effect of an operation, explicit compares are synthesized with a subtract using r0 as the destination. SPARC conditional branches test condition codes to determine all possible unsigned and signed relations. Floating point uses separate condition codes to encode the IEEE 754 conditions, requiring a floating-point compare instruction. Version 9 expanded SPARC branches in four ways: a separate set of condition codes for 64-bit operations; a branch that tests the contents of a register and branches if the value is =, not=, <, <=, >=, or <= 0 (see MIPS below); three more sets of floating-point condition codes; and branch instructions that encode static branch prediction.

PowerPC also uses four condition codes—*less than*, *greater than*, *equal*, and *summary overflow*—but it has eight copies of them. This redundancy allows the PowerPC instructions to use different condition codes without conflict, essentially giving PowerPC eight extra 4-bit registers. Any of these eight condition codes can be the target of a compare instruction, and any can be the source of a conditional branch. The integer instructions have an option bit that behaves as if the integer op is followed by a compare to zero that sets the first condition "register." PowerPC also lets the second "register" be optionally set by floating-point instructions. PowerPC provides logical operations among these eight 4-bit condition code registers (`CRAND`, `CROR`, `CRXOR`, `CRNAND`, `CRNOR`, `CREQV`), allowing more complex conditions to be tested by a single branch.

MIPS uses the contents of registers to evaluate conditional branches. Any two registers can be compared for equality (`BEQ`) or inequality (`BNE`), and then the branch is taken if the condition holds. The set on less than instructions (`SLT`, `SLTI`, `SLTU`, `SLTIU`) compare two operands and then set the destination register to 1 if less and to 0 otherwise. These instructions are enough to synthesize the full set of relations. Because of the popularity of comparisons to 0, MIPS includes special compare and branch instructions for all such comparisons: greater than or equal to zero (`BGEZ`), greater than zero (`BGTZ`), less than or equal to zero (`BLEZ`), and less than zero (`BLTZ`). Of course, equal and not equal to zero can be synthesized using `r0` with `BEQ` and `BNE`. Like SPARC, MIPS I uses a condition code for floating point with separate floating-point compare and branch instructions; MIPS IV expanded this to eight floating-point condition codes, with the floating point comparisons and branch instructions specifying the condition to set or test.

Alpha compares (`CMPEQ`, `CMPLT`, `CMPLE`, `CMPULT`, `CMPULE`) test two registers and set a third to 1 if the condition is true and to 0 otherwise. Floating-point compares (`CMTEQ`, `CMTLT`, `CMTLE`, `CMTUN`) set the result to 2.0 if the condition holds and to 0 otherwise. The branch instructions compare one register to 0 (`BEQ`, `BGE`, `BGT`, `BLE`, `BLT`, `BNE`) or its least significant bit to 0 (`BLBC`, `BLBS`) and then branch if the condition holds.

PA-RISC has many branch options, which we'll see in COD Section D.11 (Instructions Unique to PA-RISC 2.0). The most straightforward is a compare and branch instruction (`COMB`), which compares two registers, branches depending on the standard relations, and then tests the least significant bit of the result of the comparison.

ARM is similar to SPARC, in that it provides four traditional condition codes that are optionally set. `CMP` subtracts one operand from the other and the diference sets the condition codes. Compare negative (`CMN`) adds one operand to the other, and the sum sets the condition codes. `TST` performs logical AND on the two operands to set all condition codes but overflow, while `TEQ` uses exclusive OR to set the first three condition codes. Like SPARC, the conditional version of the ARM branch instruction tests condition codes to determine all possible unsigned and signed relations.

As we shall see in COD Section D.12 (Instructions Unique to ARM), one unusual feature of ARM is that every instruction has the option of executing conditionally depending on the condition codes. (This bears similarities to the annulling option of PA-RISC, seen in COD Section D.11 (Instructions Unique to PA-RISC 2.0).)

Not surprisingly, Thumb follows ARM. The differences are that setting condition codes are not optional, the `TEQ` instruction is dropped, and there is no conditional execution of instructions.

The Hitachi SuperH uses a single T-bit condition that is set by compare instructions. Two branch instructions decide to branch if either the T bit is 1 (BT) or the T bit is 0 (BF). The two flavors of branches allow fewer comparison instructions.

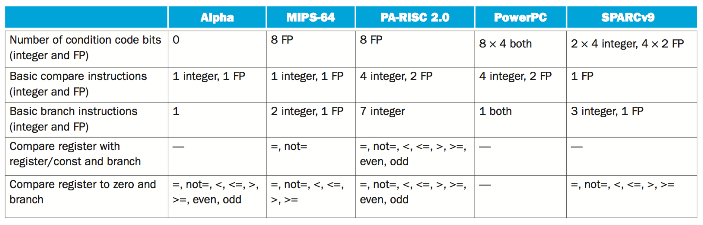
Mitsubishi M32R also offers a single condition code bit (C) used for signed and unsigned comparisons (`CMP`, `CMPI`, `CMPU`, `CMPUI`) to see if one register is less than the other or not, similar to the MIPS set on less than instructions. Two branch instructions test to see if the C bit is 1 or 0: BC and BNC. The M32R also includes instructions to branch on equality or inequality of registers (`BEQ` and `BNE`) and all relations of a register to 0 (`BGEZ`, `BGTZ`, `BLEZ`, `BLTZ`, `BEQZ`, `BNEZ`). Unlike BC and BNC, these last instructions are all 32 bits wide.

MIPS-16 keeps set on less than instructions (`SLT`, `SLTI`, `SLTU`, `SLTIU`), but instead of putting the result in one of the eight registers, it is placed in a special register named T. MIPS-16 is always implemented in machines that also have the full 32-bit MIPS instructions and registers; hence, register T is really register 24 in the full MIPS architecture. The MIPS-16 branch instructions test to see if a register is or is not equal to zero (`BEQZ` and `BNEZ`). There are also instructions that branch if register T is or is not equal to zero (`BTEQZ` and `BTNEZ`). To test if two registers are equal, MIPS added compare instructions (`CMP`, `CMPI`) that compute the exclusive OR of two registers and place the result in register T. Compare was added since MIPS-16 left out instructions to compare and branch if registers are equal or not (`BEQ` and `BNE`).

The following figures summarize the schemes used for conditional branches.

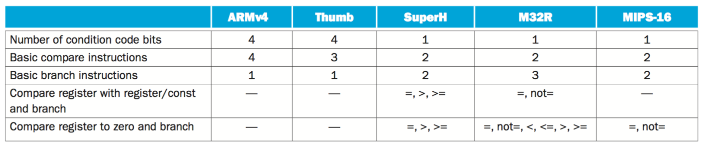
Figure 10.3.10: Summary of five desktop RISC approaches to conditional branches (COD Figure D.3.10).

Floating-point branch on PA-RISC is accomplished by copying the FP status register into an integer register and then using the branch on bit instruction to test the FP comparison bit. Integer compare on SPARC is synthesized with an arithmetic instruction that sets the condition codes using r0 as the destination.



	Alpha	MIPS-64	PA-RISC 2.0	PowerPC	SPARCv9
Number of condition code bits (integer and FP)	0	8 FP	8 FP	8 × 4 both	2 × 4 integer, 4 × 2 FP
Basic compare instructions (integer and FP)	1 integer, 1 FP	1 integer, 1 FP	4 integer, 2 FP	4 integer, 2 FP	1 FP
Basic branch instructions (integer and FP)	1	2 integer, 1 FP	7 integer	1 both	3 integer, 1 FP
Compare register with register/const and branch	—	=, not=	=, not=, <, <=, >, >=, even, odd	—	—
Compare register to zero and branch	=, not=, <, <=, >, >=, even, odd	=, not=, <, <=, >, >=	=, not=, <, <=, >, >=, even, odd	—	=, not=, <, <=, >, >=

Figure 10.3.11: Summary of five embedded RISC approaches to conditional branches (COD Figure D.3.11).



	ARMv4	Thumb	SuperH	M32R	MIPS-16
Number of condition code bits	4	4	1	1	1
Basic compare instructions	4	3	2	2	2
Basic branch instructions	1	1	2	3	2
Compare register with register/const and branch	—	—	=, >, >=	=, not=	—
Compare register to zero and branch	—	—	=, >, >=	=, not=, <, <=, >, >=	=, not=

(*1) This section is in original form.

Provide feedback on this section