C Survey of RISC Architectures

RISC: any computer announced after 1985.

Steven Przybylski
A Designer of the Stanford MIPS

C.1 Introduction

We cover four examples of reduced instruction set computer (RISC) architectures in this appendix:

- Hewlett Packard PA-RISC
- IBM and Motorola PowerPC
- SGI MIPS
- SPARC, developed originally by Sun Microsystems

We also include a discussion of DLX, the instruction set architecture invented for this book. (A review of DLX can be found on the back inside cover or in Figure 2.25 of Chapter 2.) There has never been another class of computers so similar. This similarity allows the presentation of four architectures in 25 pages, with DLX thrown in for good measure! Characteristics of these architectures are found in Figure C.1.

Readers of the first edition will note that the Intel i860 and Motorola M88000 now sleep with the fishes; HP PA-RISC and IBM PowerPC took their place in this appendix. Had we space for another architecture in our figures, the Digital Alpha AXP would join this group. Its similarities to MIPS made it the obvious candidate for omission.

	DLX	MIPS I	PA-RISC	PowerPC	SPARC V8
Date announced	1990	1986	1986	1993	1987
Instruction size (bits)	32	32	32	32	32
Address space (size, model)	32 bits, flat	32 bits, flat	48 bits, segmented	32 bits, flat	32 bits, flat
Data alignment	Aligned	Aligned	Aligned	Unaligned	Aligned
Data addressing modes	2	2	5	4	2
Protection	Page	Page	Page	Page	Page
Minimum page size	4 KB	4 KB	4 KB	4 KB	8 KB
I/O	Memory mapped	Memory mapped	Memory mapped	Memory mapped	Memory mapped
Integer registers (number, model, size)	31 GPR $\times$ 32 bits	31 GPR $\times$ 32 bits	31 GPR $\times$ 32 bits	32 GPR $\times$ 32 bits	31 GPR $\times$ 32 bits
Separate floating-point registers	32×32 or 16×64 bits	16×32 or 16×64 bits	56×32 or 28×64 bits	32×32 or 32×64 bits	32×32 or 16×64 bits
Floating-point format	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single, double

FIGURE C.1 Summary of the first version of five recent architectures. Except for number of data address modes and some instruction set details, the integer instruction sets of these architectures are very similar. Contrast this to Figure C.12 on page C-23. Later versions of these architectures all support a flat, 64-bit address space.

After discussing the addressing modes and instruction formats of our four RISC architectures, we present the survey of the instructions in three steps:

- Instructions found in DLX
- Instructions not found in DLX but found in two or more architectures
- The unique instructions and characteristics of each architecture

We conclude with a speculation about the future directions for RISCs.

The one complication in this second edition appendix is that some of the older RISCs have been extended over the years. As this book will be in print for several years, we decided to describe the latest version of the architectures, as computers with these instruction sets will be common soon even if they are not at the time of this writing: MIPS IV, PA-RISC 1.1; and SPARC version 9. We will also allude to version 2.0 of the PA-RISC instruction set occasionally, which will be published and shipped in systems shortly after the second edition of this book is complete. We give the evolution of the instruction sets in the final section.

C.2 Addressing Modes and Instruction Formats

Figure C.2 shows the data addressing modes supported by each architecture. Since all have one register that always has the value 0 when used in address modes—in fact, it is `r0` in every architecture—the absolute address mode with limited range can be synthesized using `r0` as the base in displacement addressing. (Register 0 can be changed by ALU operations in PowerPC; register 0 is always zero in the other machines.) Similarly, register-indirect addressing is synthesized by using displacement addressing with an offset of 0. Simplified addressing modes is one distinguishing feature of RISC architectures.

Addressing mode	DLX	MIPS IV	PA-RISC 1.1	PowerPC	SPARC V9
Register + offset (displacement or based)	√	√	√	√	√
Register + register (indexed)	—	√ (FP)	√	√	√
Register + scaled register (scaled)	—	—	√	—	—
Register + offset & update register	—	—	√	√	—
Register + register & update register	—	—	√	√	—

FIGURE C.2 Summary of data addressing modes. PA-RISC also has short address versions of the offset addressing modes. MIPS IV has indexed addressing for floating-point loads and stores. (These addressing modes are described in Figure 2.5, page 75.)

References to code are normally PC-relative, although register indirect is supported for returning from procedures, for case statements, and for pointer function calls. One variation is that PC-relative branch addresses in everything but DLX are shifted left 2 bits before being added to the PC, thereby increasing the branch distance. This works because the length of all instructions is 32 bits and instructions must be aligned on 32-bit words in memory.

Figure C.3 shows the format of instructions, which includes the size of the address in the instructions. Each instruction set architecture uses these four primary instruction formats. The primary differences are subtle, concerning how to extend constant fields to 32 bits. Figure C.4 shows the variations.

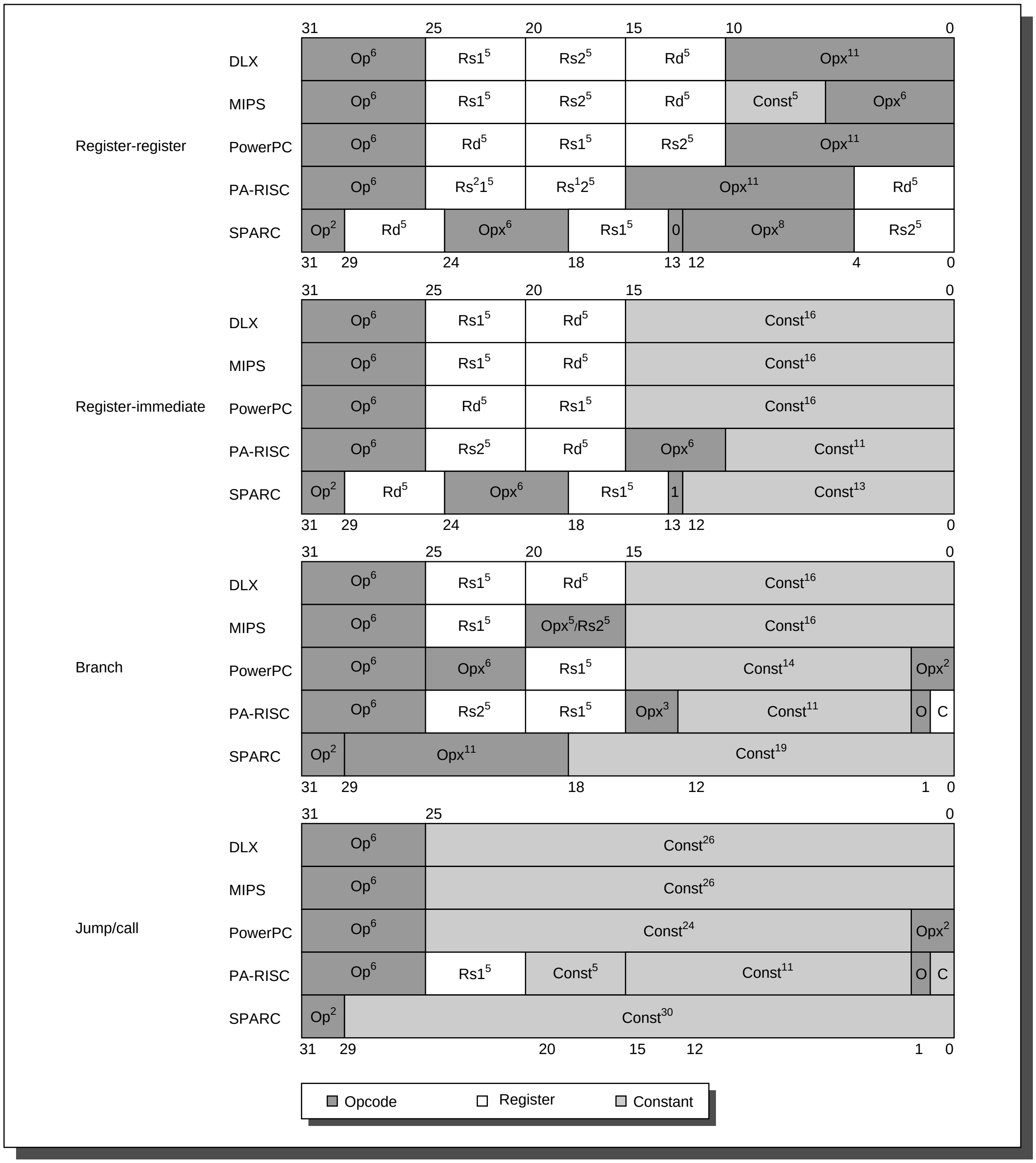
Register-register
DLX: Op6 | Rs15 | Rs25 | Rd5 | Opx11 (31, 25, 20, 15, 10, 0)
MIPS: Op6 | Rs15 | Rs25 | Rd5 | Const5 | Opx6
PowerPC: Op6 | Rd5 | Rs15 | Rs25 | Opx11
PA-RISC: Op6 | Rs215 | Rs125 | Opx11 | Rd5
SPARC: Op2 | Rd5 | Opx6 | Rs15 | 0 | Opx8 | Rs25 (31, 29, 24, 18, 13, 12, 4, 0)
Register-immediate
DLX: Op6 | Rs15 | Rd5 | Const16 (31, 25, 20, 15, 0)
MIPS: Op6 | Rs15 | Rd5 | Const16
PowerPC: Op6 | Rd5 | Rs15 | Const16
PA-RISC: Op6 | Rs25 | Rd5 | Opx6 | Const11
SPARC: Op2 | Rd5 | Opx6 | Rs15 | 1 | Const13 (31, 29, 24, 18, 13, 12, 0)
Branch
DLX: Op6 | Rs15 | Rd5 | Const16 (31, 25, 20, 15, 0)
MIPS: Op6 | Rs15 | Opx5/Rs25 | Const16
PowerPC: Op6 | Opx6 | Rs15 | Const14 | Opx2
PA-RISC: Op6 | Rs25 | Rs15 | Opx3 | Const11 | O | C
SPARC: Op2 | Opx11 | Const19 (31, 29, 18, 12, 1, 0)
Jump/call
DLX: Op6 | Const26 (31, 25, 0)
MIPS: Op6 | Const26
PowerPC: Op6 | Const24 | Opx2
PA-RISC: Op6 | Rs15 | Const5 | Const11 | O | C
SPARC: Op2 | Const30 (31, 29, 20, 15, 12, 1, 0)
Opcode
Register
Constant

FIGURE C.3 Instruction formats for five architectures. These four formats are found in all five architectures. (The superscript notation in this figure means something different from our standard notation; it shows the width of a field in bits.) Although the register fields are located in similar pieces of the instruction, be aware that the destination and two source fields are scrambled. Here are the meanings of the abbreviations: Op = the main opcode, Opx = an opcode extension, Rd = the destination register, Rs1 = source register 1, Rs2 = source register 2, and Const = a constant (used as an immediate or as an address). Version 2.0 of PA-RISC will include a 16-bit add immediate format and a 17-bit address for calls. Note that our discussion of DLX in Chapters 2 and 3 numbers bits from left to right, while this figure uses right-to-left numbering.

Format: instruction category	DLX	MIPS IV	PA-RISC 1.1	PowerPC	SPARC V9
Branch: all	Sign	Sign	Sign	Sign	Sign
Jump/call: all	Sign	—	Sign	Sign	Sign
Register-immediate: data transfer	Sign	Sign	Sign	Sign	Sign
Register-immediate: arithmetic	Sign	Sign	Sign	Sign	Sign
Register-immediate: logical	Sign	Zero	—	Zero	Sign

FIGURE C.4 Summary of constant extension. The constants in the jump and call instructions of MIPS are not sign extended since they only replace the lower 28 bits of the PC, leaving the upper 4 bits unchanged (PA-RISC has no logical immediate instructions).

C.3 Instructions: The DLX Subset

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

DLX Instructions

Almost every instruction found in DLX is found in the other architectures, as Figure C.5 shows. (For reference, definitions of the DLX instructions are found in Figure 2.25 of Chapter 2 and on the back inside cover.) Instructions are listed under four categories: data transfer; arithmetic, logical; control; and floating point. A fifth category in the figure shows conventions for register usage and pseudo-instructions on each architecture. If a DLX instruction requires a short sequence of instructions in other architectures, these instructions are separated by semicolons in Figure C.5. (To avoid confusion, the destination register will *always* be the leftmost operand in this appendix, independent of the notation normally used with each architecture.)

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.

Instruction name	DLX	MIPS IV	PA-RISC 1.1	PowerPC	SPARC V9
Data transfer (instruction formats)	**R–I**	**R–I**	**R–I, R–R**	**R–I, R–R**	**R–I, R–R**
Load byte signed	`LB`	`LB`	`LDB; EXTRS,8,31`	`LBZ; EXTSB`	`LDSB`
Load byte unsigned	`LBU`	`LBU`	`LDB,LDBX,LDBS`	`LBZ`	`LDUB`
Load half word signed	`LH`	`LH`	`LDH; EXTRS16,31`	`LHA`	`LDSH`
Load half word unsigned	`LHU`	`LHU`	`LDH,LDHX,LDHS`	`LHZ`	`LDUH`
Load word	`LW`	`LW`	`LDW,LDWX, LDWS`	`LW`	`LD`
Load SP float	`LF`	`LWC1`	`FLDWX,FLDWS`	`LFS`	`LDF`
Load DP float	`LD`	`LDC1`	`FLDDX,FLDDS`	`LFD`	`LDDF`
Store byte	`SB`	`SB`	`STB,STBX,STBS`	`STB`	`STB`
Store half word	`SH`	`SH`	`STH,STHX,STHS`	`STH`	`STH`
Store word	`SW`	`SW`	`STW,STWX,STWS`	`STW`	`ST`
Store SP float	`SF`	`SWC1`	`FSTWX,FSTWS`	`STFS`	`STF`
Store DP float	`SD`	`SWD1`	`FSTDX,FSTDS`	`STFD`	`STDF`
Read, write special registers	`MOVS2I, MOVI2S`	`MF, MT_`	`MFCTL, MTCTL`	`MFSPR, MF_, MTSPR, MT_`	`RD,WR, RDPR,WRPR, LDXFSR, STXFSR`
Move int. to FP reg.	`MOVI2FP`	`MFC1`	`STW; FLDWX`	`STW; LDFS`	`ST; LDF`
Move FP to int. reg.	`MOVFP2I`	`MTC1`	`FSTWX; LDW`	`STFS; LW`	`STF; LD`
Arithmetic, logical (instruction formats)	**R–R, R–I**	**R–R, R–I**	**R–R, R–I**	**R–R, R–I**	**R–R, R–I**
Add	`ADDU,ADDUI`	`ADDU, ADDIU`	`ADDL, LD0, ADDI, UADDCM`	`ADD,ADDI`	`ADD`
Add (trap if overflow)	`ADD,ADDI`	`ADD, ADDI`	`ADDO, ADDIO`	`ADDO; MCRXR; BC`	`ADDcc; TVS`
Sub	`SUBU,SUBUI`	`SUBU`	`SUB,SUBI`	`SUBF`	`SUB`
Sub (trap if overflow)	`SUB,SUBI`	`SUB`	`SUBTO,SUBIO`	`SUBF/oe`	`SUBcc; TVS`
Multiply	`MULTU, MULTUI`	`MULT, MULTU`	`SHiADD; ...; (i=1,2,3)`	`MULLW, MULLI`	`MULX`
Multiply (trap if ovf)	`MULT,MULTI`	—	`SHiADDO; ...;`	—	—
Divide	`DIVU,DIVUI`	`DIV,DIVU`	`DS; ...; DS`	`DIVW`	`DIVX`
Divide (trap if ovf)	`DIV,DIVI`	—	—	—	—
And	`AND,ANDI`	`AND,ANDI`	`AND`	`AND,ANDI`	`AND`
Or	`OR,ORI`	`OR,ORI`	`OR`	`OR,ORI`	`OR`
Xor	`XOR,XORI`	`XOR,XORI`	`XOR`	`XOR,XORI`	`XOR`

Figure continued on next page

Instruction Name	DLX	MIPS IV	PA-RISC 1.1	PowerPC	SPARC V9
Arithmetic *(continued)* (instruction formats)	**R–I**	**R–I**	**R–I, R–R**	**R–I, R–R**	**R–I, R–R**
Load high part register	`LHI`	`LUI`	`LDIL`	`ADDIS`	`SETHI (B fmt.)`
Shift left logical	`SLL,SLLI`	`SLLV,SLL`	`ZDEP 31-i, 32-i`	`RLWINM`	`SLL`
Shift right logical	`SRL,SRLI`	`SRLV,SRL`	`EXTRU 31, 32-i`	`RLWINM 32-i`	`SRL`
Shift right arithmetic	`SRA,SRAI`	`SRAV,SRA`	`EXTRS 31, 32-i`	`SRAW`	`SRA`
Compare	`S_(<,>,≤,≥,=,≠)`	`SLT/I, SL/ITU`	`COMB`	`CMP(I)CLR`	`SUBcc r0,...`
Control (instruction formats)	**B, J/C**	**B, J/C**	**B, J/C**	**B, J/C**	**B, J/C**
Branch on integer compare	`BEQ,BNE`	`BEQ,BNE, B_Z (<,>,≤,≥)`	`COMB, COMIB`	`BC`	`BR_Z, BPcc (<,>,≤,≥,=,≠)`
Branch on floating-point compare	`BFPT,BFPF`	`BC1T,BC1F`	`FSTWX f0; LDW t; BB t`	`BC`	`FBPfcc (<,>,≤,≥,=,...)`
Jump, jump register	`J,JR`	`J,JR`	`BL r0, BLR r0`	`B, BCLR, BCCTR`	`BA, JMPL r0,...`
Call, call register	`JAL,JALR`	`JAL,JALR`	`BL, BLE`	`BL,BLA, BCLRL, BCCTRL`	`CALL, JMPL`
Trap	`TRAP`	`BREAK`	`BREAK`	`TW, TWI`	`Ticc, SIR`
Return from interrupt	`RFE`	`JR; RFE`	`RFI,RFIR`	`RFI`	`DONE, RETRY, RETURN`
Floating point (instruction formats)	**R–R**	**R–R**	**R–R**	**R–R**	**R–R**
Add single, double	`ADDF, ADDD`	`ADD.S, ADD.D`	`FADD FADD/dbl`	`FADDS, FADD`	`FADDS, FADDD`
Sub single, double	`SUBF, SUBD`	`SUB.S, SUB.D`	`FSUB FSUB/dbl`	`FSUBS, FSUB`	`FSUBS, FSUBD`
Mult single, double	`MULF, MULD`	`MUL.S, MUL.D`	`FMPY FMPY/dbl`	`FMULS, FMUL`	`FMULS, FMULD`
Div single, double	`DIVF, DIVD`	`DIV.S, DIV.D`	`FDIV, FDIV/dbl`	`FDIVS, FDIV`	`FDIVS, FDIVD`
Compare	`_F, _D (<,>,≤,≥,=, ...)`	`C_.S, C_.D (<,>,≤,≥,=, ...)`	`FCMP, FCMP/dbl (<,=,>)`	`FCMP`	`FCMPS, FCMPD`
Move R–R	`MOVF`	`MOV.S`	`FCPY`	`FMV`	`FMOVS/D/Q`
Convert (single,double,integer) to (single,double,integer)	`CVTF2D, CVTD2F, CVTF2I, CVTD2I, CVTI2F, CVTI2D`	`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 continued on next page

Instruction Name	DLX	MIPS IV	PA-RISC 1.1	PowerPC	SPARC V9
Conventions					
Register with value 0	`r0`	`r0`	`r0`	`r0 (addressing)`	`r0`
Return address reg.	`r31`	`r31`	`r2, r31`	`link (special)`	`r31`
No-op	`ADD r0,r0,r0`	`SLL r0,r0,r0`	`OR r0,r0,r0`	`ORI r0,r0,#0`	`SETHI r0,0`
Move R–R integer	`ADD ...,r0,...`	`ADD ...,r0,...`	`OR ...,r0,...`	`OR rx, ry, ry`	`OR ...,r0,...`
Operand order	`OP Rd,Rs1,Rs2`	`OP Rd,Rs1,Rs2`	`OP Rs1,Rs2,Rd`	`OP Rd,Rs1,Rs2`	`OP Rs1,Rs2,Rd`

FIGURE C.5 Instructions equivalent to DLX. 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 DLX, they are separated by commas. Note that in the “Arithmetic, logical” category all machines but SPARC use separate instruction mnemonics to indicate an immediate operand; SPARC offers immediate versions of these instructions but uses a single mnemonic. (Of course these are separate opcodes!)

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 =,≠,<,≤,≥, 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 was 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 expands this to eight floating-point condition codes, with the floating-point comparisons and branch instructions specifying the condition to set or test.

PA-RISC has many branch options, which we'll see in section C.8. The most straightforward is a compare and branch instruction (`COMB`), which compares two registers, then branches depending on the standard relations, and tests the least-significant bit of the result of the comparison.

Figure C.6 summarizes the four schemes used for conditional branches.

	DLX	**MIPS IV**	**PA-RISC 1.1**	**PowerPC**	**SPARC V9**
Number of condition code bits (integer and FP)	1 FP	8 FP	1 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, 1 FP	4 integer, 2 FP	1 FP
Basic branch instructions (integer and FP)	1 integer, 1 FP	2 integer, 1 FP	7 integer	1 both	3 integer, 1 FP
Compare register with register/ const and branch	=,≠	=,≠	=,≠,<,≤,>,≥, even, odd	—	—
Compare register to zero and branch	=,≠	=,≠,<,≤,>,≥	=,≠,<,≤,>,≥, even, odd	—	=,≠,<,≤,>,≥

FIGURE C.6 Summary of five approaches to conditional branches. 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. PA-RISC 2.0 will have eight floating-point condition code bits.

C.4 Instructions: Common Extensions to DLX

Figure C.7 lists instructions not found in Figure C.5 in the same four categories. Instructions are put in this list if they appear in more than one of the four architectures. The instructions are defined using the hardware description language, which is described on the page facing the inside back cover.

Name	Definition	MIPS IV	PA-RISC 1.1	PowerPC	SPARC V9
Data transfer					
Atomic swap R/M (for semaphores)	Temp←Rd; Rd← Mem[x]; Mem[x]←Temp	`LL;SC`	— (see C.8)	`LWARX; STWCX`	`CASA, CASX`
Load 64-bit integer	Rd$\leftarrow_{64}$ Mem[x]	`LD`	(in 2.0)	`LD`	`LDX`
Store 64-bit int.	Mem[x]$\leftarrow_{64}$ Rd	`SD`	(in 2.0)	`STD`	`STX`
Load 32-bit int. unsigned	$Rd_{32..63}\leftarrow_{32}$ Mem[x]; $Rd_{0..31}\leftarrow_{32} 0$	`LWU`	(in 2.0)	`LWZ`	`LDUW`
Load 32-bit int. signed	$Rd_{32..63}\leftarrow_{32}$ Mem[x]; $Rd_{0..31}\leftarrow_{32} \text{Mem}[x]_0^{32}$	`LW`	(in 2.0)	`LWA`	`LDSW`
Prefetch	Cache[x]←*hint*	`PREF, PREFX`	`LDWX, LDWS, STWX,STWS`	`DCBT, DCBTST`	`PREFETCH`
Load coprocessor	Coprocessor←Mem[x]	`LWCi`	`CLDWX,CLDWS`	—	—
Store coprocessor	Mem[x]←Coprocessor	`SWCi`	`CSTWX,CSTWS`	—	—
Endian	(Big/Little Endian?)	Either	Either	Either	Either
Cache flush	(Flush cache block at this address)	`CP0op`	`FDC, FIC`	`DCBF`	`FLUSH`
Shared memory synchronization	(All prior data transfers complete before next data transfers may start)	`SYNC`	`SYNC`	`SYNC`	`MEMBAR`
Arithmetic, logical					
64-bit integer arithmetic ops	Rd$\leftarrow_{64}$Rs1 op_{64} Rs2	`DADD,DSUB DMULT, DDIV`	(in 2.0)	`ADD,SUBF, MULLD, DIVD`	`ADD, SUB, MULX, S/UDIVX`
64-bit integer logical ops	Rd$\leftarrow_{64}$Rs1 op_{64} Rs2	`AND,OR,XOR`	(in 2.0)	`AND,OR,XOR`	`AND,OR,XOR`
64-bit shifts	Rd$\leftarrow_{64}$Rs1 op_{64} Rs2	`DSLL,DSRA, DSRL`	(in 2.0)	`SLD,SRAD, SRLD`	`SLLX, SRAX, SRLX`
Conditional move	if (cond) Rd← Rs	`MOVN/Z`	`SUBc,n; ADD`	—	`MOVcc, MOVr`
Support for multi-word integer add	CarryOut,Rd ← Rs1 + Rs2 + OldCarryOut	`ADU;SLTU; ADDU`	`ADDC`	`ADDC, ADDE.`	`ADDcc`
Support for multi-word integer sub	CarryOut,Rd ← Rs1 Rs2 + OldCarryOut	`SUBU;SLTU; SUBU`	`SUBB`	`SUBFC, SUBFE.`	`SUBcc`
And not	Rd ← Rs1 & ~(Rs2)	—	`ANDCM`	`ANDC`	`ANDN`
Or not	Rd ← Rs1 | ~(Rs2)	—	—	`ORC`	`ORN`
Add high immediate	$Rd_{0..15}\leftarrow Rs1_{0..15}$ + (Const<<16);	—	`ADDIL (R-I)`	`ADDIS (R-I)`	—
Coprocessor operations	(Defined by coprocessor)	`COPi`	`COPR,i`	—	`IMPDEPi`

Figure continued on next page

Name	Definition	MIPS IV	PA-RISC 1.1	PowerPC	SPARC V9
Control					
Optimized delayed branches	(Branch not always delayed)	`BEQL,BNEL, B_ZL (<,>,≤,≥)`	`COMBT,n, COMBF,n`	–	`BPcc,A FPBcc,A`
Conditional trap	if (COND) {R31←PC; PC ←0..0#i}	`T_,T_I (=,≠,<,>,≤,≥)`	`SUBc,n; BREAK`	`TW, TD, TWI, TDI`	`Tcc`
No. control regs.	Misc. regs (virtual memory, interrupts,...)	≈12	32	33	29
Floating point					
Multiply & Add	Fd ← (Fs1 × Fs2) + Fs3	`MADD.S/D`	— (see C.8)	`FMADD/S`	
Multiply & Sub	Fd ← (Fs1 × Fs2) – Fs3	`MSUB.S/D`	— (see C.8)	`FMSUB/S`	
Neg Mult & Add	Fd ← –((Fs1 × Fs2)+Fs3)	`NMADD.S/D`		`FNMADD/S`	
Neg Mult & Sub	Fd ←–((Fs1 × Fs2)–Fs3)	`NMSUB.S/D`		`FNMSUB/S`	
Square Root	Fd ← SQRT(Fs)	`SQRT.S/D`	`FSQRTsgl/dbl`	`FSQRT/S`	`FSQRTS/D`
Conditional Move	if (cond) Fd←Fs	`MOVF/T, MOVF/T.S/D,`	`FTEST;FCPY`	–	`FMOVcc`
Negate	Fd ← Fs ^ x80000000	`NEG.S/D`	(in 2.0)	`FNEG`	`FNEGS/D/Q`
Absolute value	Fd ← Fs & x7FFFFFFF	`ABS.S/D`	`FABS/dbl`	`FABS`	`FABSS/D/Q`

FIGURE C.7 Instructions not found in DLX but found in two or more of the four architectures.

Although most of the categories are self-explanatory, a few bear comment:

- The “atomic swap” row means a primitive that can exchange a register with memory without interruption. This is useful for operating system semaphores in uniprocessor as well as for multiprocessor synchronization (see section 8.5 of Chapter 8).
- The 64-bit data transfer and operation rows show how MIPS, PowerPC, and SPARC define 64-bit addressing and integer operations. SPARC simply defines all register and addressing operations to be 64 bits, adding only special instructions for 64-bit shifts, data transfers, and branches. MIPS includes the same extensions, plus it adds separate 64-bit signed arithmetic instructions. PowerPC added 64-bit right shift, load, store, divide, and compare and has a separate mode determining whether instructions are interpreted as 32- or 64-bit operations; 64-bit operations will not work in a machine that only supports 32-bit mode. PA-RISC is expanded to 64-bit addressing and operations in version 2.0.
- The “prefetch” instruction supplies an address and hint to the implementation about the data. Hints include that the data is likely to be read or written soon,

likely to be read or written only once, or likely to be read or written many times. Prefetch does not cause exceptions. MIPS has a version that adds two registers to get the address for floating-point programs, unlike non-floating-point MIPS programs. (See pages 412–414 in Chapter 5 to learn more about prefetching.)

- In the “Endian” row, “Big or Little” means there is a bit in the program status register that allows the processor to act either as Big Endian or Little Endian (see page 73 in Chapter 2). This can be accomplished by simply complementing some of the least-significant bits of the address in data transfer instructions.
- The “shared memory synchronization” helps with cache-coherent multiprocessors: All loads and stores executed before the instruction must complete before loads and stores after it can start. (See section 8.5 of Chapter 8.)
- The “coprocessor operations” row lists several categories that allow for the processor to be extended with special-purpose hardware.

One difference that needs a longer explanation is the optimized branches. Figure C.8 shows the options. The PowerPC offers branches that take effect immediately, like branches on earlier architectures. This avoids executing NOPs when there is no instruction to fill the delay slot; all the rest offer delayed branches. The other three provide a version of delayed branch that makes it easier to fill the delay slot. The SPARC “annulling” branch executes the instruction in the delay slot only if the branch is taken; otherwise the instruction is annulled. This means the instruction at the target of the branch can safely be copied into the delay slot since it will only be executed if the branch is taken. The restrictions are that the target is not another branch and that the target is known at compile time. (SPARC also offers a non-delayed jump because an unconditional branch with the annul bit set does *not* execute the following instruction.) Recent versions of the MIPS architecture have added a branch likely instruction that also annuls the following instruction if the branch is not taken. PA-RISC allows almost any instruction to annul the next instruction, including branches. Its “nullifying” branch option will execute the next instruction depending on the direction of the branch and whether it is taken (i.e., if a forward branch is *not* taken or a backward branch is taken). Presumably this choice was made to optimize loops, allowing the instructions following the exit branch and the looping branch to execute in the common case.

Now that we have covered the similarities, we will focus on the unique features of each architecture, ordering them by length of description of the unique features from shortest to longest.

	(Plain) Branch	**Delayed branch**	**Annulling delayed branch**	
Found in architectures	PowerPC	DLX, MIPS, PA-RISC, SPARC	MIPS, SPARC	PA-RISC
Execute following instruction	Only if branch *not* taken	Always	Only if branch taken	If forward branch *not* taken or backward branch taken

FIGURE C.8 When the instruction following the branch is executed for three types of branches.

C.5 Instructions Unique to MIPS

MIPS has gone through four generations of instruction set evolution, and this evolution has generally added features found in other architectures. Here are the salient unique features of MIPS, the first several of which were found in the original instruction set.

Nonaligned Data Transfers

MIPS has special instructions to handle misaligned words in memory. A rare event in most programs, it is included for COBOL programs where the programmer can force misalignment by declarations. Although most RISCs trap if you try to load a word or store a word to a misaligned address, on all architectures misaligned words can be accessed without traps by using four load byte instructions and then assembling the result using shifts and logical ORs. The MIPS load and store word left and right instructions (`LWL`, `LWR`, `SWL`, `SWR`) allow this to be done in just two instructions: `LWL` loads the left portion of the register and `LWR` loads the right portion of the register. `SWL` and `SWR` do the corresponding stores. Figure C.9 shows how they work. There are also 64-bit versions of these instructions.

TLB Instructions

TLB misses are handled in software in MIPS, so the instruction set also has instructions for manipulating the registers of the TLB (see pages 455–456 in Chapter 5 for more on TLBs). These registers are considered part of the "system coprocessor" and thus can be accessed by the instructions that move between coprocessor registers and integer registers. The contents of a TLB entry are read by loading via read indexed TLB entry (`TLBR`) and written using either write indexed TLB entry (`TLBWI`) or write random TLB entry (`TLBWR`). The TLB contents are searched using probe TLB for matching entry (`TLBP`).

Remaining Instructions

Below is a list of the remaining unique details of the MIPS architecture:

- *`NOR`*—This logical instruction calculates ~(Rs1 | Rs2).
- *Constant shift amount*—Non-variable shifts use the 5-bit constant field shown in the register-register format in Figure C.3.
- *`SYSCALL`*—This special trap instruction is used to invoke the operating system.
- *Move to/from control registers*—`CTCi` and `CFCi` move between the integer registers and control registers.

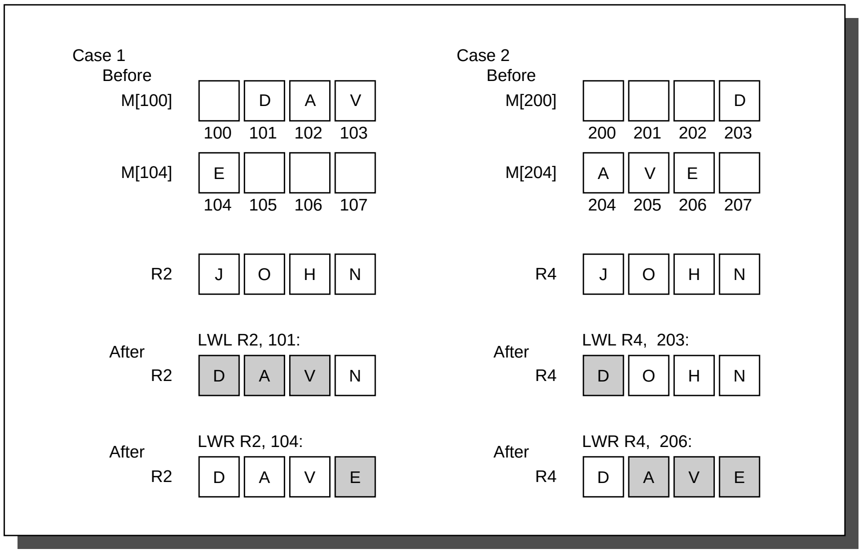

FIGURE C.9 MIPS instructions for unaligned word reads. This figure assumes operation in Big Endian mode. Case 1 first loads the 3 bytes 101,102, and 103 into the left of R2, leaving the least-significant byte undisturbed. The following LWR simply loads byte 104 into the least-significant byte of R2, leaving the other bytes of the register unchanged using LWL. Case 2 first loads byte 203 into the most-significant byte of R4, and the following LWR loads the other 3 bytes of R4 from memory bytes 204, 205, and 206. LWL reads the word with the first byte from memory, shifts to the left to discard the unneeded byte(s), and changes only those bytes in Rd. The byte(s) transferred are from the first byte until the lowest-order byte of the word. The following LWR addresses the last byte, right shifts to discard the unneeded byte(s), and finally changes only those bytes of Rd. The byte(s) transferred are from the last byte up to the highest-order byte of the word. Store word left (SWL) is simply the inverse of LWL, and store word right (SWR) is the inverse of LWR. Changing to Little Endian mode flips which bytes are selected and discarded. (If big-little, left-right, load-store seem confusing, don't worry, it works!)

- *Jump/call not PC-relative*—The 26-bit address of jumps and calls is not added to the PC. It is shifted left 2 bits and replaces the lower 28 bits of the PC. This would only make a difference if the program were located near a 256-MB boundary.
- *Load linked/store conditional*—This pair of instructions gives MIPS atomic operations for semaphores, allowing data to be read from memory, modified, and stored without fear of interrupts or other machines accessing the data in a multiprocessor (see section 8.5 of Chapter 8). There are both 32- and 64-bit versions of these instructions.
- *Reciprocal and reciprocal square root*—These instructions, which do *not* follow IEEE 754 guidelines of proper rounding, are included apparently for applications that value speed of divide and square root more than they value accuracy.

- *Conditional procedure call instructions*—BGEZAL saves the return address and branches if the content of Rs1 is greater than or equal to zero, and BLTZAL does the same for less than zero. The purpose of these instructions is to get a PC-relative call. (There are “likely” versions of these instructions as well.)

There is no specific provision in the MIPS architecture for floating-point execution to proceed in parallel with integer execution, but the MIPS implementations of floating point allow this to happen by checking to see if arithmetic interrupts are possible early in the cycle (see Appendix A). Normally interrupts are not possible when integer and floating point operate in parallel.

C.6 Instructions Unique to SPARC

Several features are unique to SPARC.

Register Windows

The primary unique feature of SPARC is register windows, an optimization for reducing register traffic on procedure calls. Several banks of registers are used, with a new one allocated on each procedure call. Although this could limit the depth of procedure calls, the limitation is avoided by operating the banks as a circular buffer, providing unlimited depth. The knee of the cost-performance curve seems to be six to eight banks.

SPARC can have between two and 32 windows, typically using eight registers each for the globals, locals, incoming parameters, and outgoing parameters. (Given each window has 16 unique registers, an implementation of SPARC can have as few as 40 physical registers and as many as 520, although most have 128 to 136, so far.) Rather than tie window changes with call and return instructions, SPARC has the separate instructions SAVE and RESTORE. SAVE is used to “save” the caller’s window by pointing to the next window of registers in addition to performing an add instruction. The trick is that the source registers are from the caller’s window of the addition operation, while the destination register is in the callee’s window. SPARC compilers typically use this instruction for changing the stack pointer to allocate local variables in a new stack frame. RESTORE is the inverse of SAVE, bringing back the caller’s window while acting as an add instruction, with the source registers from the callee’s window and the destination register in the caller’s window. This automatically deallocates the stack frame. Compilers can also make use of it for generating the callee’s final return value.

The danger of register windows is that the larger number of registers could slow down the clock rate. This was not the case for early implementations. The SPARC architecture (with register windows) and the MIPS R2000 architecture (without) have been built in several technologies since 1987. For several generations the SPARC clock rate has not been slower than the MIPS clock rate for implementations in similar technologies, probably because cache-access times dominate register-access times in these implementations. The current generation machines took different implementation strategies—superscalar vs. superpipelining—and it's unlikely that the number of registers by themselves determined the clock rate in either machine.

Another data transfer feature is alternate space option for loads and stores. This simply allows the memory system to identify memory accesses to input/output devices, or to control registers for devices such as the cache and memory-management unit.

Fast Traps

Version 9 SPARC includes support to make traps fast. It expands the single level of traps to at least four levels, allowing the window overflow and underflow trap handlers to be interrupted. The extra levels mean the handler does not need to check for page faults or misaligned stack pointers explicitly in the code, thereby making the handler faster. Two new instructions were added to return from this multilevel handler: `RETRY` (which retries the interrupted instruction) and `DONE` (which does not). To support user-level traps, the instruction `RETURN` will return from the trap in nonprivileged mode.

Support for LISP and Smalltalk

The primary remaining arithmetic feature is tagged addition and subtraction. The designers of SPARC spent some time thinking about languages like LISP and Smalltalk, and this influenced some of the features of SPARC already discussed: register windows, conditional trap instructions, calls with 32-bit instruction addresses, and multiword arithmetic (see Taylor et al. [1986] and Ungar et al. [1984]). A small amount of support is offered for tagged data types with operations for addition, subtraction, and hence comparison. The two least-significant bits indicate whether the operand is an integer (coded as 00), so `TADDcc` and `TSUBcc` set the overflow bit if either operand is not tagged as an integer or if the result is too large. A subsequent conditional branch or trap instruction can decide what to do. (If the operands are not integers, software recovers the operands, checks the types of the operands, and invokes the correct operation based on those types.) It turns out that the misaligned memory access trap can also be put to use for tagged data, since loading from a pointer with the wrong tag can be an invalid access. Figure C.10 shows both types of tag support.

Overlapped Integer and Floating-Point Operations

SPARC allows floating-point instructions to overlap execution with integer instructions. To recover from an interrupt during such a situation, SPARC has a queue of pending floating-point instructions and their addresses. `RDPR` allows the

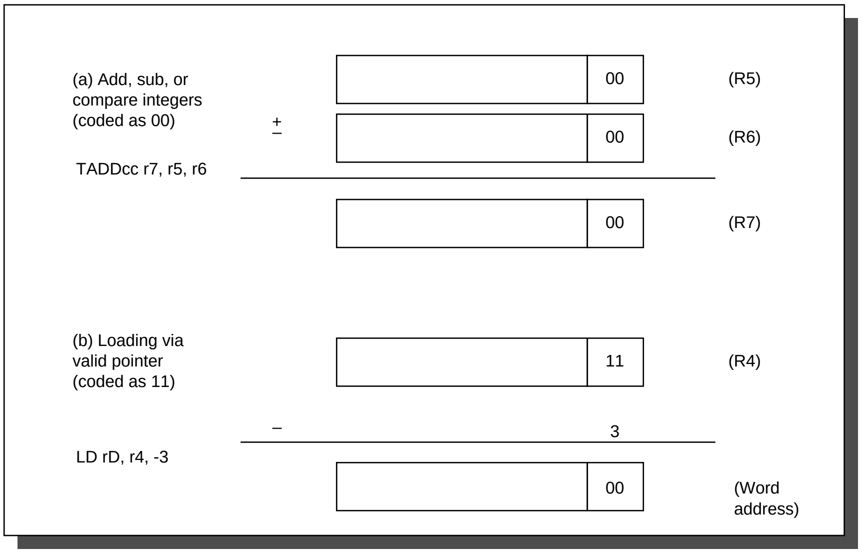
(a) Add, sub, or compare integers (coded as 00)
TADDcc r7, r5, r6
+
−
00
(R5)
00
(R6)
00
(R7)
(b) Loading via valid pointer (coded as 11)
LD rD, r4, -3
11
(R4)
–
3
00
(Word address)

FIGURE C.10 SPARC uses the two least-significant bits to encode different data types for the tagged arithmetic instructions. (a) Integer arithmetic, which takes a single cycle as long as the operands and the result are integers. (b) The misaligned trap can be used to catch invalid memory accesses, such as trying to use an integer as a pointer. For languages with paired data like LISP, an offset of –3 can be used to access the even word of a pair (CAR) and +1 can be used for the odd word of a pair (CDR).

processor to empty the queue. The second floating-point feature is the inclusion of floating-point square root instructions `FSQRTS`, `FSQRTD`, and `FSQRTQ`.

Remaining Instructions

The remaining unique features of SPARC are

- *`JMPL`* uses `Rd` to specify the return address register, so specifying `r31` makes it similar to `JALR` in DLX and specifying `r0` makes it like `JR`.
- *`LDSTUB`* loads the value of the byte into `Rd` and then stores $\texttt{FF}_{16}$ into the addressed byte. This version 8 instruction can be used to implement a semaphore.
- *CASA* (*CASXA*) atomically compares a value in a processor register to 32-bit (64-bit) value in memory; if and only if they are equal, it swaps the value in memory with the value in a second processor register. This version 9 instruction can be used to construct wait-free synchronization algorithms that do not require the use of locks.
- *`XNOR`* calculates the exclusive or with the complement of the second operand.

- `BPcc`, `BPr`, and `FBPcc` include a branch prediction bit so that the compiler can give hints to the machine about whether a branch is likely to be taken or not.
- `ILLTRAP` causes an illegal instruction trap. Muchnick [1988] explains how this is used for proper execution of aggregate returning procedures in C.
- `POPC` counts the number of bits set to one in an operand.
- *Non-faulting loads* allow compilers to move load instructions ahead of conditional control structures that control their use. Hence, non-faulting loads will be executed speculatively.
- *Quadruple precision floating-point arithmetic and data transfer* allow the floating-point registers to act as eight 128-bit registers for floating-point operations and data transfers.
- *Multiple-precision floating-point results for multiply* mean that two single-precision operands can result in a double-precision product and two double-precision operands can result in a quadruple-precision product. These instructions can be useful in complex arithmetic and some models of floating-point calculations.

C.7 Instructions Unique to PowerPC

PowerPC is the result of several generations of IBM commercial RISC machines: IBM RT/PC, IBM Power-1, and IBM Power-2.

Branch Registers: Link and Counter

Rather than dedicate one of the 32 general-purpose registers to save the return address on procedure call, PowerPC puts the address into a special register called the *link register.* Since many procedures will return without calling another procedure, link doesn't always have to be saved away. Making the return address a special register makes the return jump faster since the hardware need not go through the register read pipeline stage for return jumps.

In a similar vein, PowerPC has a *count register* to be used in for loops where the program iterates for a fixed number of times. By using a special register the branch hardware can determine quickly whether a branch based on the count register is likely to branch, since the value of the register is known early in the execution cycle. Tests of the value of the count register in a branch instruction will automatically decrement the count register.

Given that the count register and link register are already located with the hardware that controls branches, and that one of the problems in branch prediction is getting the target address early in the pipeline (see Chapter 3, section 3.5), the PowerPC architects decided to make a second use of these registers. Either register can hold a target address of a conditional branch. Thus PowerPC supplements its basic conditional branch with two instructions that get the target address from these registers (`BCLR`, `BCCTR`).

Remaining Instructions

Unlike other RISC machines, register 0 is not hardwired to the value 0. It cannot be used as a base register, but in base+index addressing it can be used as the index. The other unique features of the PowerPC are

- *Load multiple* and *store multiple* save or restore up to 32 registers in a single instruction.
- `LSW` and `STSW` permit fetching and storing of fixed and variable-length strings that have arbitrary alignment.
- *Rotate with mask* instructions support bit field extraction and insertion. One version rotates the data and then performs logical AND with a mask of ones, thereby extracting a field. The other version rotates the data but only places the bits into the destination register where there is a corresponding 1 bit in the mask, thereby inserting a field.
- *Algebraic right shift* sets the carry bit (`CA`) if the operand is negative and any one bits are shifted out. Thus a signed divide by any constant power of two that rounds toward zero can be accomplished with a `SRAWI` followed by `ADDZE`, which adds `CA` to the register.
- *CBTLZ* will count leading zeros.
- *SUBFIC* computes (immediate – RA), which can be used to develop a one’s or two’s complement.
- *Logical shifted immediate* instructions shift the 16-bit immediate to the left 16 bits before performing AND, OR, or XOR.

C.8 Instructions Unique to PA-RISC

PA-RISC was expanded slightly in 1990 with version 1.1 and changed significantly in 2.0 with 64-bit extensions that will be in systems shipped in 1996. PA-RISC perhaps has the most unusual features of any commercial RISC machine. For example, it has the most addressing modes, instruction formats, and, as we shall see, several instructions that are really the combination of two simpler instructions.

Nullification

As shown in Figure C.8 on page C-12, several RISC machines can choose to not execute the instruction following a delayed branch, in order to improve utilization of the branch slot. This is called *nullification* in PA-RISC, and it has been generalized to apply to any arithmetic-logical instruction as well as to all branches. Thus an add instruction can add two operands, store the sum, and cause the following instruction to be skipped if the sum is zero. Like conditional move instructions, nullification allows PA-RISC to avoid branches in cases where there is just one instruction in the `then` part of an `if` statement.

A Cornucopia of Conditional Branches

Given nullification, PA-RISC did not need to have separate conditional branch instructions. The inventors could have recommended that nullifying instructions precede unconditional branches, thereby simplifying the instruction set. Instead, PA-RISC has the largest number of conditional branches of any RISC machine. Figure C.11 shows the conditional branches of PA-RISC. As you can see, several are really combinations of two instructions.

Name	**Instruction**	**Notation**	
`COMB`	Compare and branch	`if (cond(Rs1,Rs2))`	`{PC ← PC + offset12}`
`COMIB`	Compare imm. and branch	`if (cond(imm5,Rs2))`	`{PC ← PC + offset12}`
`MOVB`	Move and branch	`Rs2 ← Rs1,` `if (cond(Rs1,0))`	`{PC ← PC + offset12}`
`MOVIB`	Move immediate and branch	`Rs2 ← imm5,` `if (cond(imm5,0))`	`{PC ← PC + offset12}`
`ADDB`	Add and branch	`Rs2 ← Rs1 + Rs2,` `if (cond(Rs1 + Rs2,0))`	`{PC ← PC + offset12}`
`ADDIB`	Add imm. and branch	`Rs2 ← imm5 + Rs2,` `if (cond(imm5 + Rs2,0))`	`{PC ← PC + offset12}`
`BB`	Branch on bit	if (cond(Rs_p,0)	`{PC ← PC + offset12}`
`BVB`	Branch on variable bit	if (cond(Rs_{sar},0)	`{PC ← PC + offset12}`

FIGURE C.11 The PA-RISC conditional branch instructions. The 12-bit offset is called `offset12` in this table, and the 5-bit immediate is called `imm5`. The 16 conditions are =, <, ≤, odd, signed overflow, unsigned no overflow, zero or no overflow unsigned, never, and their respective complements. The BB instruction selects one of the 32 bits of the register and branches depending if its value is 0 or 1. The BVB selects the bit to branch using the shift amount register, a special-purpose register. The subscript notation specifies a bit field.

Synthesized Multiply and Divide

PA-RISC provides several primitives so that multiply and divide can be synthesized in software. Instructions that shift one operand 1, 2, or 3 bits and then add, trapping or not on overflow, are useful in multiplies. Divide step performs the critical step of nonrestoring divide, adding or subtracting depending on the sign of the prior result. Magenheimer et al. [1988] measured the size of operands in multiplies and divides to show how well the multiply step would work. Using these data for C programs, Muchnick [1988] found that by making special cases the average multiply by a constant takes 6 clock cycles and multiply of variables takes 24 clock cycles. PA-RISC has 10 instructions for these operations.

The original SPARC architecture used similar optimizations, but with increasing number of transistors the instruction set was expanded to include full multiply and divide operations. PA-RISC gives some support along these lines by putting a full 32-bit integer multiply in the floating-point unit; however, the integer data must first be moved to floating-point registers.

Decimal Operations

COBOL programs will compute on decimal values, stored as 4 bits per digit, rather than converting back and forth between binary and decimal. PA-RISC has instructions that will convert the sum from a normal 32-bit add into proper decimal digits. It also provides logical and arithmetic operations that set the condition codes to test for carries of digit, bytes, or half words. These operations also test whether bytes or half words are zero. These operations would be useful in arithmetic on 8-bit ASCII characters. Five PA-RISC instructions provide decimal support.

Remaining Instructions

Here are some remaining PA-RISC instructions:

- *Branch vectored* shifts an index register left 3 bits, adds it to a base register and then branches to the calculated address. It is used for case statements.
- *Extract* and *deposit* instructions allow arbitrary bit fields to be selected from or inserted into registers. Variations include whether the extracted field is sign-extended, whether the bit field is specified directly in the instruction or indirectly in another register, and whether the rest of the register is set to zero or left unchanged. PA-RISC has 12 such instructions.
- To simplify use of 32-bit address constants, PA-RISC includes `ADDIL`, which adds a left-adjusted 21-bit constant to a register and places the result in register 1. The following data transfer instruction uses offset addressing to add the lower 11 bits of the address to register 1. This pair of instructions allows PA-RISC to add a 32-bit constant to a base register, at the cost of changing register 1.
- PA-RISC has nine debug instructions that can set breakpoints on instruction or data addresses and return the trapped addresses.

- *Load* and *clear* instructions provide a semaphore that reads a value from memory and then writes zero.
- *Store bytes short* optimizes unaligned data moves, moving either the leftmost or the rightmost bytes in a word to the effective address depending on the instruction options and condition code bits.
- Loads and stores work well with caches by having options that give hints about whether to load data into the cache if it's not already in the cache. For example, load with a destination of register 0 is defined to be a cache hint.
- *Multiply/add* and *multiply/subtract* are floating-point operations that can launch two independent floating-point operations in a single instruction. Version 2.0 of PA-RISC will have fused multiply-add like the PowerPC.

In addition to instructions, here are a few features that distinguish PA-RISC:

- The segmented address space above the 2^{32} boundary means that there must be instructions to manipulate the segment registers and branch instructions that can leave the current segment.
- The data addressing modes use either a 14-bit offset or a 5-bit offset, and the sum of the base register and the immediate can be used to update the base register. The decision of whether to use only the base register or the sum as the effective address is optional. For 5-bit offsets there is a bit in the instruction that makes the decision, but in the 14-bit offsets it depends on the sign bit offset: Negative means use the sum, positive means use the register. These options turn the standard 6-integer data transfers into 20 instructions. PA-RISC 2.0 makes the set of addressing options more orthogonal.

C.9 Concluding Remarks

This appendix covers the addressing modes, instruction formats, and all instructions found in four recent RISC architectures. Although the later sections concentrate on the differences, it would not be possible to cover four architectures in these few pages if there were not so many similarities. In fact, we would guess that more than 90% of the instructions executed for any of these architectures would be found in Figure C.5 on pages C-6–C-8. To contrast this homogeneity, Figure C.12 gives a summary for four architectures from the 1970s in a format similar to that shown in Figure C.1 on page C-2. (Imagine trying to write a single appendix in this style for those architectures.) In the history of computing, there has never been such widespread agreement on computer architecture.

	IBM 360/370	**Intel 8086**	**Motorola 68000**	**DEC VAX**
Date announced	1964/1970	1978	1980	1977
Instruction size(s) (bits)	16,32,48	8,16,24,32, 40,48	16,32,48,64,80	8,16,24,32,..., 432
Addressing (size, model)	24 bits, flat/ 31 bits, flat	4+16 bits, segmented	24 bits, flat	32 bits, flat
Data aligned?	Yes 360/ No 370	No	16-bit aligned	No
Data addressing modes	2/3	5	9	≥ 14
Protection	Page	None	Optional	Page
Page size	2 KB & 4 KB	—	0.25 to 32 KB	0.5 KB
I/O	Opcode	Opcode	Memory mapped	Memory mapped
Integer registers (size, model, number)	16 GPR × 32 bits	8 dedicated data × 16 bits	8 data & 8 address × 32 bits	15 GPR × 32 bits
Separate floating-point registers	4 × 64 bits	Optional: 8 × 80 bits	Optional: 8 × 80 bits	0
Floating-point format	IBM (floating hexadecimal)	IEEE 754 single, double, extended	IEEE 754 single, double, extended	DEC

FIGURE C.12 Summary of four 1970s architectures. Unlike the architectures in Figure C.1 on page C-2, there is little agreement between these architectures in any category. (See Appendix D for more details on the 8086; in fact, the description of just this one machine is as long as this whole appendix!)

This style of architectures cannot remain static, however. Like people, instruction sets tend to get bigger as they get older. Figure C.13 shows the genealogy of these instruction sets, and Figure C.14 shows which features were added to or deleted from generations of machines over time.

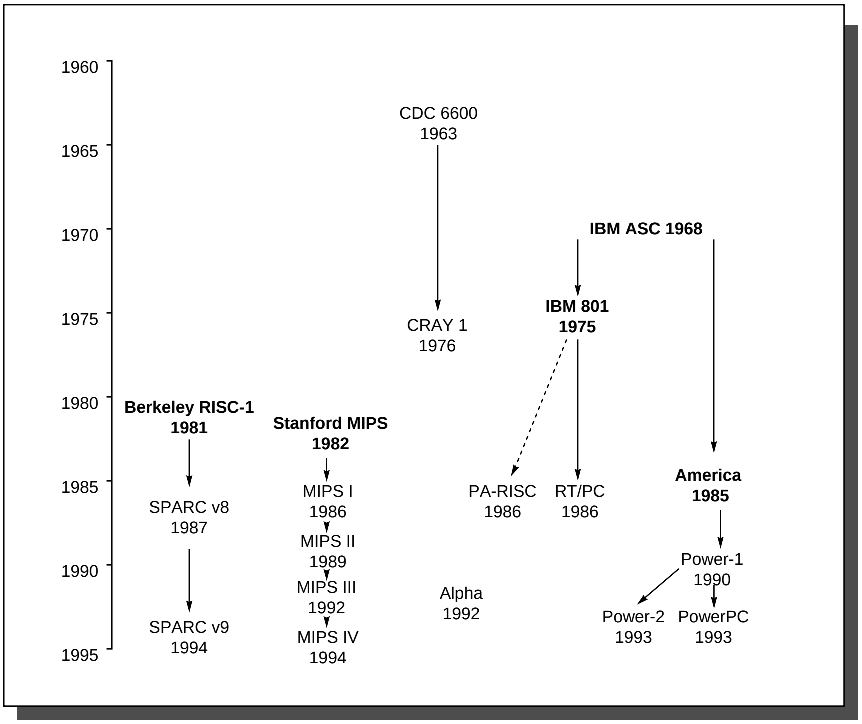

FIGURE C.13 The lineage of RISC instruction sets. Commercial machines are shown in plain text and research machines in **bold**. The CDC-6600 and Cray-1 were load-store machines with register 0 fixed at 0, and separate integer and floating-point registers. Instructions could not cross word boundaries. An early IBM research machine led to the 801 and America research projects, with the 801 leading to the unsuccessful RT/PC and America leading to the successful Power architecture. Some people who worked on the 801 later joined Hewlett Packard to work on the PA-RISC. The two university projects were the basis of MIPS and SPARC machines. DEC shipped workstations using MIPS microprocessors for three years before they brought out their own RISC instruction set, Alpha, which is very similar to MIPS III.

	PA-RISC			SPARC		MIPS				Power		
Feature	**1.0**	**1.1**	**2.0**	**v. 8**	**v. 9**	**I**	**II**	**III**	**IV**	**1**	**2**	**PC**
Interlocked loads	√	"	"	√	"		+	"	"	√	"	"
Load/store FP double	√	"	"	√	"		+	"	"	√	"	"
Semaphore	√	"	"	√	"		+	"	"	√	"	"
Square root	√	"	"	√	"		+	"	"		+	"
Single-precision FP ops	√	"	"	√	"	√	"	"	"			+
Memory synchronization	√	"	"	√	"		+	"	"	√	"	"
Coprocessor	√	"	"	√	–	√	"	"	"			
Base + index addressing	√	"	"	√	"				+	√	"	"
≈ 32 64-bit FP registers		"	"		+			+	"	√	"	"
Annulling delayed branch	√	"	"	√	"		+	"	"			
Branch register contents	√	"	"		+	√	"	"	"			
Big or Little Endian		+	"		+	√	"	"	"			+
Branch prediction bit					+		+	"	"	√	"	"
Conditional move					+				+	√	"	–
Prefetch data into cache			+		+				+	√	"	"
64-bit addressing/ int. ops			+		+			+	"			+
32-bit multiply, divide		+	"		+	√	"	"	"	√	"	"
Load/store FP quad					+						+	–
Fused FP mul/add			+						+	√	"	"
String instructions	√	"	"							√	"	–

FIGURE C.14 Features added to RISC machines. √ means in the original machine, + means added later, " means continued from prior machine, and – means removed from architecture.

C.10 References

BHANDARKAR, D. P. [1995]. *Alpha Architecture and Implementations*, Digital Press, Newton, Mass.

HEWLETT PACKARD [1994]. *PA-RISC 1.1 Architecture Reference Manual*, 3rd ed.

IBM [1994]. *The PowerPC Architecture*, Morgan Kaufmann, San Francisco.

KANE, G. [1988]. *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, N. J.

MAGENHEIMER, D. J., L. PETERS, K. W. PETTIS, AND D. ZURAS [1988]. "Integer multiplication and division on the HP Precision Architecture," *IEEE Trans. on Computers*, 37:8, 980–990.

MUCHNICK, S. S. [1988]. "Optimizing compilers for SPARC," *Sun Technology* (Summer) 1:3, 64–77.

SILICON GRAPHICS [1994]. *MIPS IV Instruction Set*, Revision 2.2.

SITES, R. L. (ED.) [1992]. *Alpha Architecture Reference Manual*, Digital Press, Newton, Mass.

SUN MICROSYSTEMS [1989]. *The SPARC Architectural Manual*, Version 8, Part No. 800-1399-09, August 25, 1989.

Taylor, G., P. Hilfinger, J. Larus, D. Patterson, and B. Zorn [1986]. “Evaluation of the SPUR LISP architecture,” *Proc. 13th Symposium on Computer Architecture (*June), Tokyo.

Ungar, D., R. Blau, P. Foley, D. Samples, and D. Patterson [1984]. “Architecture of SOAR: Smalltalk on a RISC,” *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 188–197.

Weaver, D. L. and T. Germond [1994]. *The SPARC Architectural Manual*, Version 9, Prentice Hall, Englewood Cliffs, N. J.

Weiss, S. and J. E. Smith [1994]. *Power and PowerPC*, Morgan Kaufmann, San Francisco.