ASSEMBLY Programming/ISA

MIPS

By

Dr Jeff Drobman

website drjeffsoftware.com/classroom.html

email jeffrey.drobman@csun.edu

ISA Index

MIPS

MIPS Technologies

From Wikipedia, the free encyclopedia
(Redirected from MIPS Computer)

MIPS Technologies, Inc., formerly **MIPS Computer Systems, Inc.**, was an American fabless semiconductor design company that is most widely known for developing the MIPS architecture and a series of RISC CPU chips based on it.[1][2] MIPS provides processor architectures and cores for digital home, networking, embedded, Internet of things and mobile applications.[3][4]

MIPS Technologies, Inc. is owned[5] by Wave Computing, who acquired it from Tallwood MIPS Inc., a company indirectly owned by Tallwood Venture Capital. Tallwood bought it on 2017-10-25 from Imagination Technologies, a UK-based company best known for their PowerVR graphics processor family.[6] Imagination Technologies had previously bought MIPS after CEVA, Inc. pulled out of a bidding on 2013-02-08.

MIPS Technologies, Inc.

MIPS TECHNOLOGIES
Logo

The former MIPS Technologies building in Santa Clara

Type	Subsidiary
Industry	RISC microprocessors
Fate	Acquired in 2018 by Wave Computing
Founded	1984; 36 years ago
Founder	John L. Hennessy
Defunct	2013
Headquarters	Sunnyvale, California, U.S.
Key people	Sandeep Vij
Products	Semiconductor intellectual property
Number of employees	up to 50 (according to LinkedIn in May 2018), previously 146 (September 2010)
Parent	Wave Computing

MIPS

History [edit]

Defunct	2013
Headquarters	Sunnyvale, California, U.S.
Key people	Sandeep Vij
Products	Semiconductor intellectual property
Number of employees	up to 50 (according to LinkedIn in May 2018), previously 146 (September 2010)
Parent	Wave Computing
Website	www.mips.com

MIPS Computer Systems Inc. was founded in 1984[7][8] by a group of researchers from Stanford University that included John L. Hennessy and Chris Rowen. These researchers had worked on a project called MIPS (for *Microprocessor without Interlocked Pipeline Stages*), one of the projects that pioneered the RISC concept. Other principal founders were Skip Stritter, formerly a Motorola technologist, and John Moussouris, formerly of IBM.[9]

The initial CEO was Vaemond Crane, formerly President and CEO of Computer Consoles Inc., who arrived in February 1985 and departed in June 1989. He was replaced by Bob Miller, a former senior IBM and Data General executive. Miller ran the company through its IPO and subsequent sale to Silicon Graphics.

In 1988, MIPS Computer Systems designs were noticed by Silicon Graphics (SGI) and the company adopted the MIPS architecture for its computers.[10] A year later, in December 1989, MIPS held its first IPO. That year, Digital Equipment Corporation (DEC) released a Unix workstation based on the MIPS design.

After developing the R2000 and R3000 microprocessors, a management change brought along the larger dreams of being a computer vendor. The company found itself unable to compete in the computer market against much larger companies and was struggling to support the costs of developing both the chips and the systems (MIPS Magnum). To secure the supply of future generations of MIPS microprocessors (the 64-bit R4000), SGI acquired the company in 1992[11] for $333 million[12][13] and renamed it as MIPS Technologies Inc., a wholly owned subsidiary of SGI.[14]

During SGI's ownership of MIPS, the company introduced the R8000 in 1994 and the R10000[15] in 1996 and a follow up the R12000 in 1997.[16] During this time, two future microprocessors code-named *The Beast* and *Capitan* were in development; these were cancelled after SGI decided to migrate to the Itanium architecture[17] in 1998.[12][18] As a result, MIPS was spun out as an intellectual property licensing company, offering licences to the MIPS architecture as well as microprocessor core designs.

MIPS

Company timeline [edit]

Year	
1981	Dr. John Hennessy at Stanford University founds and leads Stanford MIPS, a research program aimed at building a microprocessor using RISC principles.
1984	MIPS Computer Systems, Inc. co-founded by Dr. John Hennessy, Skip Stritter, and Dr. John Moussouris[43]
1986	First product ships: R2000 microprocessor, Unix workstation, and optimizing compilers
1988	R3000 microprocessor
1989	First IPO in November as MIPS Computer Systems with Bob Miller as CEO
1991	R4000 microprocessor
1992	SGI acquires MIPS Computer Systems. Transforms it into internal MIPS Group, and then incorporates and renames it to MIPS Technologies, Inc. (a wholly owned subsidiary of SGI)
1994	R8000 microprocessor
1994	Sony PlayStation released, using an R3000 CPU with custom GTE coprocessor
1996	R10000 microprocessor; Nintendo 64 released, incorporating a cut down R4300 processor.
1998	Re-IPO as MIPS Technologies, Inc
1999	Sony PlayStation 2 released, using an R5900 cpu with custom vector coprocessors
2002	Acquires Algorithmics Ltd, a UK-based MIPS development hardware/software and consultancy company.
September 6, 2005	Acquires First Silicon Solutions (FS2), a Lake Oswego, Oregon company as a wholly owned subsidiary. FS2 specializes in silicon IP, design services and OCI (On-Chip Instrumentation) development tools for programming, testing, debug and trace of embedded systems in SoC, SOPC, FPGA, ASSP and ASIC devices.
2007	MIPS Technologies acquires Portugal-based mixed-signal intellectual property company Chipidea
February 2009	MIPS Joins Linux Foundation[44]
May 8, 2009	Chipidea is sold to Synopsys.
June 2009	Android is ported to MIPS[45]

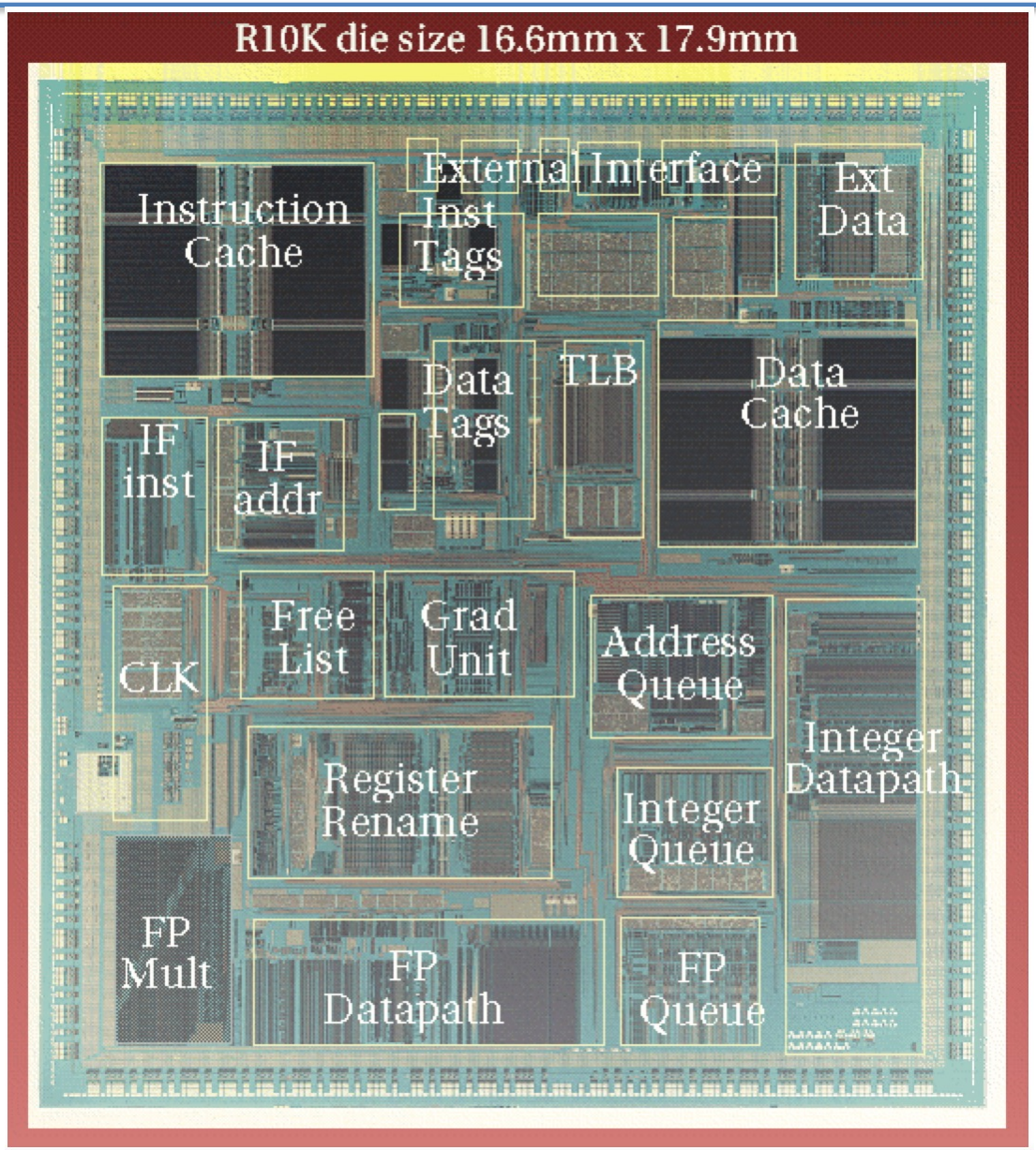
IDT's MIPS R3000 Die

First MIPS RISC CPUs

32-bit

64-bit

MIPS R10000



MIPS

Wikipedia MIPS

MIPS architecture

From Wikipedia, the free encyclopedia

Not to be confused with Millions of instructions per second, MIPS-X, or Stanford MIPS.

MIPS (Microprocessor without Interlocked Pipelined Stages)[2] is a reduced instruction set computer (RISC) instruction set architecture (ISA)[3]:A-1[4]:19 developed by MIPS Computer Systems, now MIPS Technologies, based in the United States.

There are multiple versions of MIPS: including MIPS I, II, III, IV, and V; as well as five releases of MIPS32/64 (for 32- and 64-bit implementations, respectively). The early MIPS architectures were 32-bit only; 64-bit versions were developed later. As of April 2017, the current version of MIPS is MIPS32/64 Release 6.[5][6] MIPS32/64 primarily differs from MIPS I–V by defining the privileged kernel mode System Control Coprocessor in addition to the user mode architecture.

Computer architecture courses in universities and technical schools often study the MIPS architecture.[7] The architecture greatly influenced later RISC architectures such as Alpha.

As of April 2017, MIPS processors are used in embedded systems such as residential gateways and routers. Originally, MIPS was designed for general-purpose computing. During the 1980s and 1990s, MIPS processors for personal, workstation, and server computers were used by many companies such as Digital Equipment Corporation, MIPS Computer Systems, NEC, Pyramid Technology, SiCortex, Siemens Nixdorf, Silicon Graphics, and Tandem Computers. Historically, video game consoles such as the Nintendo 64, Sony PlayStation, PlayStation 2, and PlayStation Portable used MIPS processors. MIPS processors also used to be popular in supercomputers during the 1990s, but all such systems have dropped off the TOP500 list. These uses were complemented by embedded applications at first, but during the 1990s, MIPS became a major presence in the embedded processor market, and by the 2000s, most MIPS

ISA

MIPS

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

MIPS ISA's

Wikipedia MIPS

Base ISA

2 MIPS II — R6000 — 1989
3 MIPS III — R4000 — 1991 — 64-bit
4 MIPS IV
5 MIPS V
6 MIPS32/MIPS64 — 1999
6.1 MIPS32/MIPS64 Release 1
6.2 MIPS32/MIPS64 Release 3
6.3 MIPS32/MIPS64 Release 5
6.4 MIPS32/MIPS64 Release 6
7 microMIPS

Extensions

8 Application-specific extensions
8.1 MIPS MCU
8.2 MIPS16
8.2.1 MIPS16e
8.2.2 MIPS16e2
8.3 MIPS DSP
8.4 MIPS SIMD architecture
8.5 MIPS virtualization
8.6 MIPS multi-threading
8.7 SmartMIPS

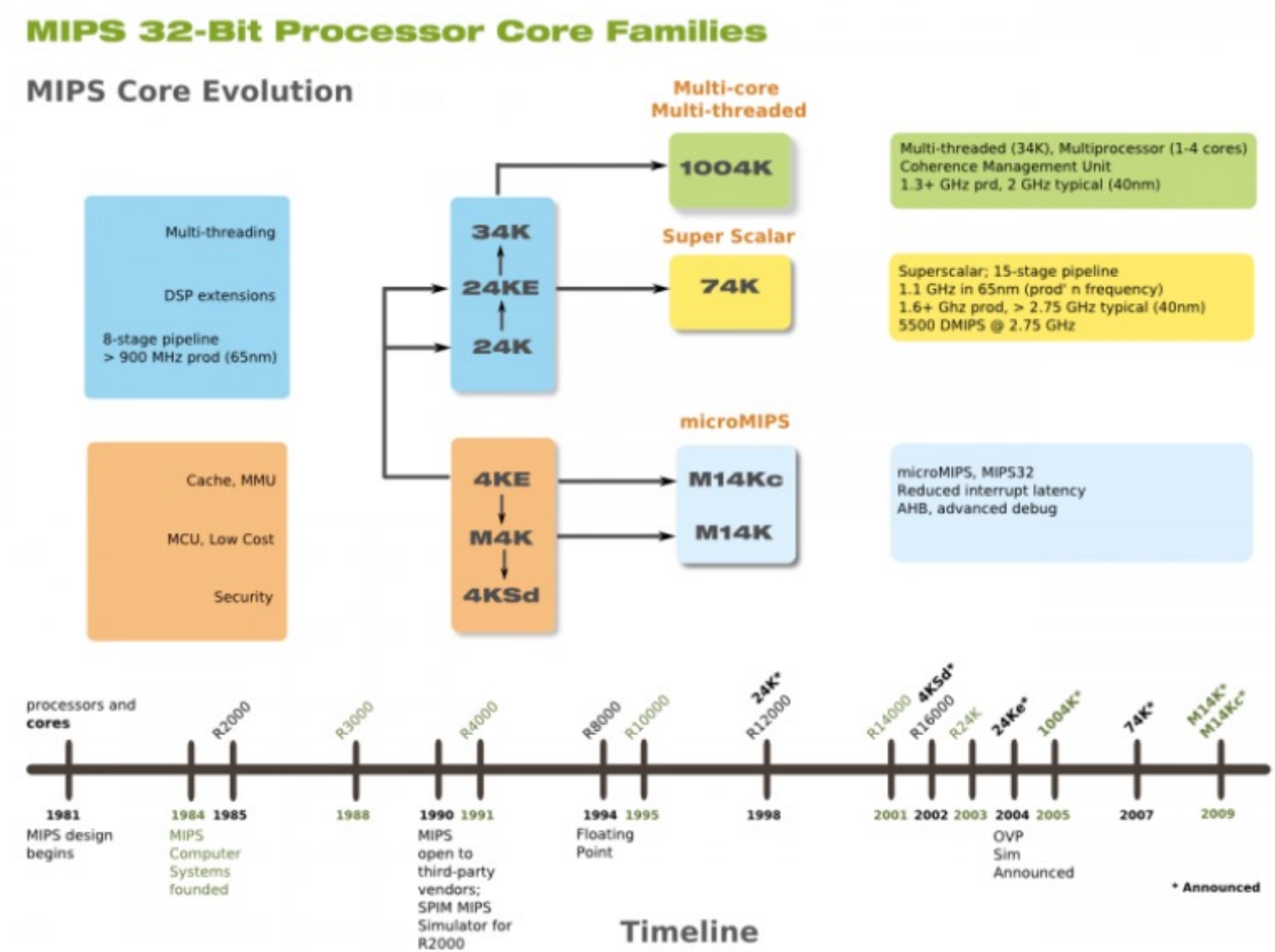
- Privilege states
  - **Kernel**
  - Supervisor
  - User

1985

R2000→R3000 — 32-bit

Classic Processors	MIPS I	R2000 · R3000
	MIPS II	R6000
	MIPS III	R4000 (R4400) · R4200 (R4300i) · R4600 (R4700)
	MIPS IV	R5000 · R8000 · R10000 (R12000 · R12000A · R14000 · R14000A · R16000 · R16000A · R18000)

MIPS Cores



MIPS ISA's & Modules

Architectures

Based on a heritage built over more than three decades of constant innovation, the MIPS architecture is the industry's most efficient RISC architecture, delivering the best performance and lowest power consumption in a given silicon area.

- nanoMIPS Architecture
- MIPS32 Instruction Set Architecture (ISA)
- MIPS64 Architecture ISA
- microMIPS ISA
- MIPS Multi-Threading architecture module
- MIPS Virtualization architecture module
- MIPS SIMD architecture module
- MIPS DSP architecture module
- MIPS MCU architecture module
- MIPS16e architecture module

MIPS Core Details

Microcontrollers (Embedded Device)	4Kc/4KEc	ATI/AMD/Broadcom Xilleon
	MIPS32 compatible	Loongson 1 Series (LS1A0300 · LS1B · LS1C300 · LS1C101 · LS1D · LS1G · LS1H)
Networking	4Kc/4KEc	Qualcomm Atheros (AR2313 · AR2318) · MediaTek (RT2880) · Texas Instruments/Infineon/Lantiq (AR7) · Lantiq (AMAZON)
	5Kc	Marvell (88E6318 "Link Street")
	24Kc/24KEc	Qualcomm Atheros (AR7240 · AR7161 · AR9132 · AR9331) · MediaTek (RT3050 · RT3052 · RT3350 · RT5350 · RT6856 · MT7620) · Lantiq (DANUBE · VINAX)
	34Kc	Lantiq (AR188 · VRX288 · GRX388) · Ikanos (Fusiv Vx175/173 · Fusiv Vx180 · Fusiv Vx185/183)
	74Kc	Qualcomm Atheros (AR9344 · QCA9558) · MediaTek (RT3662 · RT3883) · Broadcom (BCM4706)
	1004Kc	MediaTek (MT7621)
	1074Kc	Realtek (RTL8198C)
	MIPS32 compatible	Broadcom (various) · Cavium (various) · Alchemy Semiconductor (Alchemy) · RMI Corporation (XLR)
	MIPS64 compatible	Broadcom (various) · Cavium (Octeon)
Gaming	various	PlayStation 1 MIPS R3000A-compatible · Nintendo 64 NEC VR4300 · PlayStation Portable R4000-based · PlayStation 2 Emotion Engine
Supercomputer	MIPS64 compatible	Loongson-based systems (LS2F/LS2F1000 · LS3A1000 · LS3B1000) · SiCortex
Aerospace	MIPS64 compatible	Loongson 1 Series (LS1E0300/LS1E1000)
	MIPS32 compatible	Loongson 1 Series (LS1E04 · LS1F04/LS1F0300 · LS1J)

MIPS32 ISA

MIPS32 Instruction Set - MIPS

Page

The **MIPS32 instruction set** is an instruction set standard published in 1999 that was promulgated by MIPS Technologies after its demerger from Silicon Graphics in 1998. The MIPS32 instruction set was developed along side the MIPS64 Instruction Set which includes 64-bit instructions. The MIP32 standard included coprocessor 0 control instructions for the first time. Today, the MIP32 instruction set is the most common MIPS instruction set, compatible with most CPUs. Due to its relative simply, the MIP32 instruction set is also the most common instruction set thought in computer architecture university courses.

The latest MIPS32 revision is revision 5, which added a set of new memory-efficient operations for large memory footprint applications.

MIPS32	
Designer:	MIPS Technologies, Inc.
Bits:	32-bits
Introduced:	1999
Version:	Revision 5.3
Design:	RISC
Type:	Register-Register
Encoding:	Fixed-length
Branching:	Condition Register
Endianness:	Bi-endian
Extensions:	SPECIAL2, COP2, LWC2, SWC2, LDC2, SDC2
Application-specific extension:	MIPS16e, MCU, SmartMIPS
Multimedia extension:	MIPS-3D
Registers	
General purpose:	32
Floating point:	32
Special purpose:	PRId

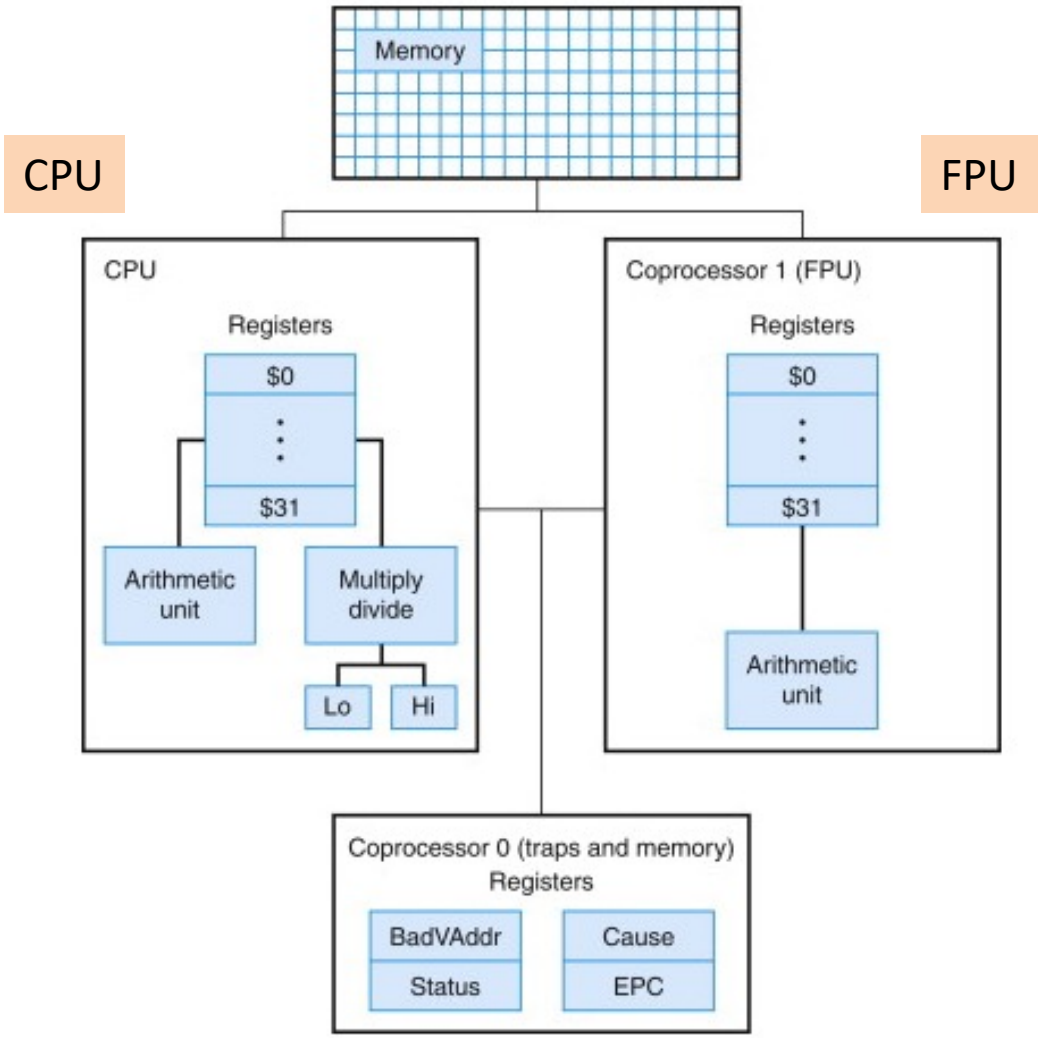
MIPS	
Designer	MIPS Technologies, Imagination Technologies
Bits	64-bit (32 → 64)
Introduced	1985; 34 years ago
Version	MIPS32/64 Release 6 (2014)
Design	RISC
Type	Register-Register
Encoding	Fixed
Branching	Compare and branch
Endianness	Bi
Page size	4 KB
Extensions	MDMX, MIPS-3D
Open	Yes, and royalty free[1]
Registers	
General purpose	32

MIPS I– Base (R2000) Org

Hennessy & Patterson

MIPS

Figure 7.10.1: MIPS R2000 CPU and FPU (COD Figure A.10.1).



Instruction Formats

Wikipedia MIPS

The following are the three formats used for the core instruction set:

Type	-31-			format (bits)		-0-
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	address (26)				

R_d R_{S1} R_{S2}

- ❑ "shamt" ::= shift amount (5 bits)
- ❑ "funct" ::= function (opcode extension – 6 bits)

Directives

Hennessy & Patterson | MIPS

Labels:

Memory segments

```
        .data
item:   .word 1
        .text
        .globl main        # Must be global
main:   lw          $t0, item
```

- Special chars in "Strings"

  - newline \n
  - tab \n t
  - quote \"

Directives

Table 7.10.1: MIPS assembler directives supported by SPIM.

Directive	Definition
`.align n`	Align the next datum on a 2^n byte boundary. For example, `.align 2` aligns the next value on a word boundary. `.align` 0 turns off automatic alignment of `.half`, `.word`, `.float`, and `.double` directives until the next `.data` or `.kdata` directive.
`.ascii str`	Store the string *str* in memory, but do not null-terminate it.
`.asciiz str`	Store the string *str* in memory and null-terminate it.
`.byte b1,..., bn`	Store the *n* values in successive bytes of memory.
`.data <addr>`	Subsequent items are stored in the data segment. If the optional argument *addr* is present, subsequent items are stored starting at address *addr*.
`.double d1,..., dn`	Store the *n* floating-point double precision numbers in successive memory locations.
`.extern sym size`	Declare that the datum stored at *sym* is *size* bytes large and is a global label. This directive enables the assembler to store the datum in a portion of the data segment that is efficiently accessed via register `$gp`.

Directives

`.extern sym size`	Declare that the datum stored at *sym* is *size* bytes large and is a global label. This directive enables the assembler to store the datum in a portion of the data segment that is efficiently accessed via register `$gp`.
`.float f1,..., fn`	Store the *n* floating-point single precision numbers in successive memory locations.
`.globl sym`	Declare that label *sym* is global and can be referenced from other files.
`.half h1,..., hn`	Store the *n* 16-bit quantities in successive memory halfwords.
`.kdata <addr>`	Subsequent data items are stored in the kernel data segment. If the optional argument *addr* is present, subsequent items are stored starting at address *addr*.
`.ktext <addr>`	Subsequent items are put in the kernel text segment. In SPIM, these items may only be instructions or words (see the `.word` directive below). If the optional argument *addr* is present, subsequent items are stored starting at address *addr*.
`.set noat` and `.set at`	The first directive prevents SPIM from complaining about subsequent instructions that use register `$at`. The second directive re-enables the warning. Since pseudoinstructions expand into code that uses register `$at`, programmers must be very careful about leaving values in this register.
`.space n`	Allocates *n* bytes of space in the current segment (which must be the data segment in SPIM).
`.text <addr>`	Subsequent items are put in the user text segment. In SPIM, these items may only be instructions or words (see the `.word` directive below). If the optional argument *addr* is present, subsequent items are stored starting at address *addr*.
`.word w1,..., wn`	Store the *n* 32-bit quantities in successive memory words.

GP Registers

Register use convention: Hennessy & Patterson

The calling convention described in this section is the one used by the gcc compiler. The native MIPS compiler uses a more complex convention that is slightly faster.

The MIPS CPU contains 32 general-purpose registers that are numbered 0—31. Register `$0` always contains the hardwired value 0.

- Registers `$at` (1), `$k0` (26), and `$k1` (27) are reserved for the assembler and operating system and should not be used by user programs or compilers.
- Registers `$a0-$a3` (4—7) are used to pass the first four arguments to routines (remaining arguments are passed on the stack). Registers `$v0` and `$v1` (2, 3) are used to return values from functions.
- Registers `$t0-$t9` (8—15, 24, 25) are *caller-saved registers* that are used to hold temporary quantities that need not be preserved across calls (see COD Section 2.8 (Supporting Procedures in Computer Hardware)).
- Registers `$s0-$s7` (16—23) are *callee-saved registers* that hold long-lived values that should be preserved across calls.
- Register `$gp` (28) is a global pointer that points to the middle of a 64K block of memory in the static data segment.
- Register `$sp` (29) is the stack pointer, which points to the last location on the stack. Register `$fp` (30) is the frame pointer. The `jal` instruction writes register `$ra` (31), the return address from a procedure call. These two registers are explain in COD Section A.7 (Exceptions and interrupts)

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

- $a(0:3) *args*
- $at, $k(0:1) *reserve*
- $v(0:1) *values*
- $t(0-9) *temp*
- $s(0:7) *saved*
- $gp *global ptr*
- $sp *stack ptr*
- $fp *frame ptr*
- $ra *return addr*

GP Registers

Register use convention: Hennessy & Patterson

Figure 2.8.1: What is and what is not preserved across a procedure call (COD Figure 2.11).

If the software relies on the frame pointer register or on the global pointer register, discussed in the following subs
preserved.

Preserved	Not preserved
Saved registers: $s0–$s7	Temporary registers: $t0–$t9
Stack pointer register: $sp	Argument registers: $a0–$a3
Return address register: $ra	Return value registers: $v0–$v1
Stack above the stack pointer	Stack below the stack pointer

- $a(0:3) *args*
- $at, $k(0:1) *reserved*
- $v(0:1) *values*
- $t(0-9) *temp*
- $s(0:7) *saved*
- $gp *global ptr*
- $sp *stack ptr*
- $fp *frame ptr*
- $ra *return addr*

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP122

MARS

DR JEFF SOFTWARE
INDIE APP DEVELOPER

MARS (MIPS Assembler and Runtime Simulator)

Registers

Mars

MARS 4.5

File Edit Run Settings Tools Help

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$8 (vaddr)	8	0x00000000
$12 (status)	12	0x0000ff11
$13 (cause)	13	0x00000000
$14 (epc)	14	0x00000000

Registers | Coproc

Name	Float
$f0	0x00000000
$f1	0x00000000
$f2	0x00000000
$f3	0x00000000
$f4	0x00000000
$f5	0x00000000
$f6	0x00000000
$f7	0x00000000
$f8	0x00000000
$f9	0x00000000
$f10	0x00000000

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

MIPS Assembly: Branches

P&H Ch2

Example 2.10.2: Branching far away.

Given a branch on register **$s0** being equal to register **$s1**,

```
beq $s0, $s1, L1
```

Cond'l JUMP

replace it by a pair of instructions that offers a much greater branching distance.

Answer

These instructions replace the short-address conditional branch:

```
      bne $s0, $s1, L2
      j   L1
L2:
```

SKIP (cond'l)

MIPS Assembly: Address Modes

P&H Ch2

I

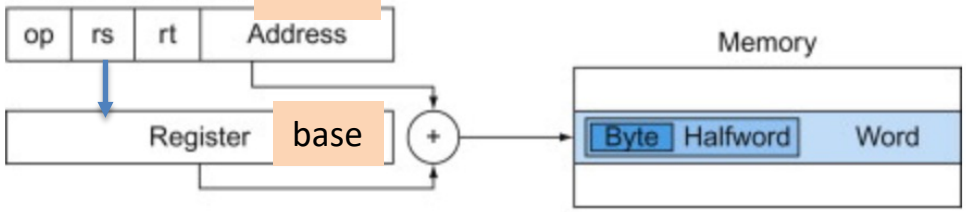
1. Immediate addressing

R

2. Register addressing

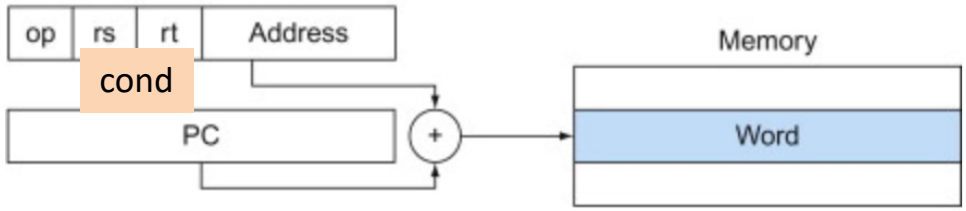
I

3. Base addressing



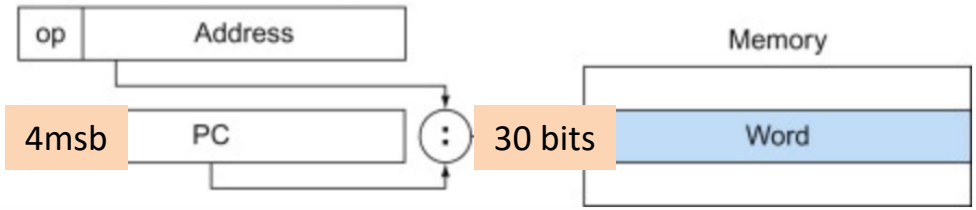
I

4. PC-relative addressing



J

5. Pseudodirect addressing



Address Formats

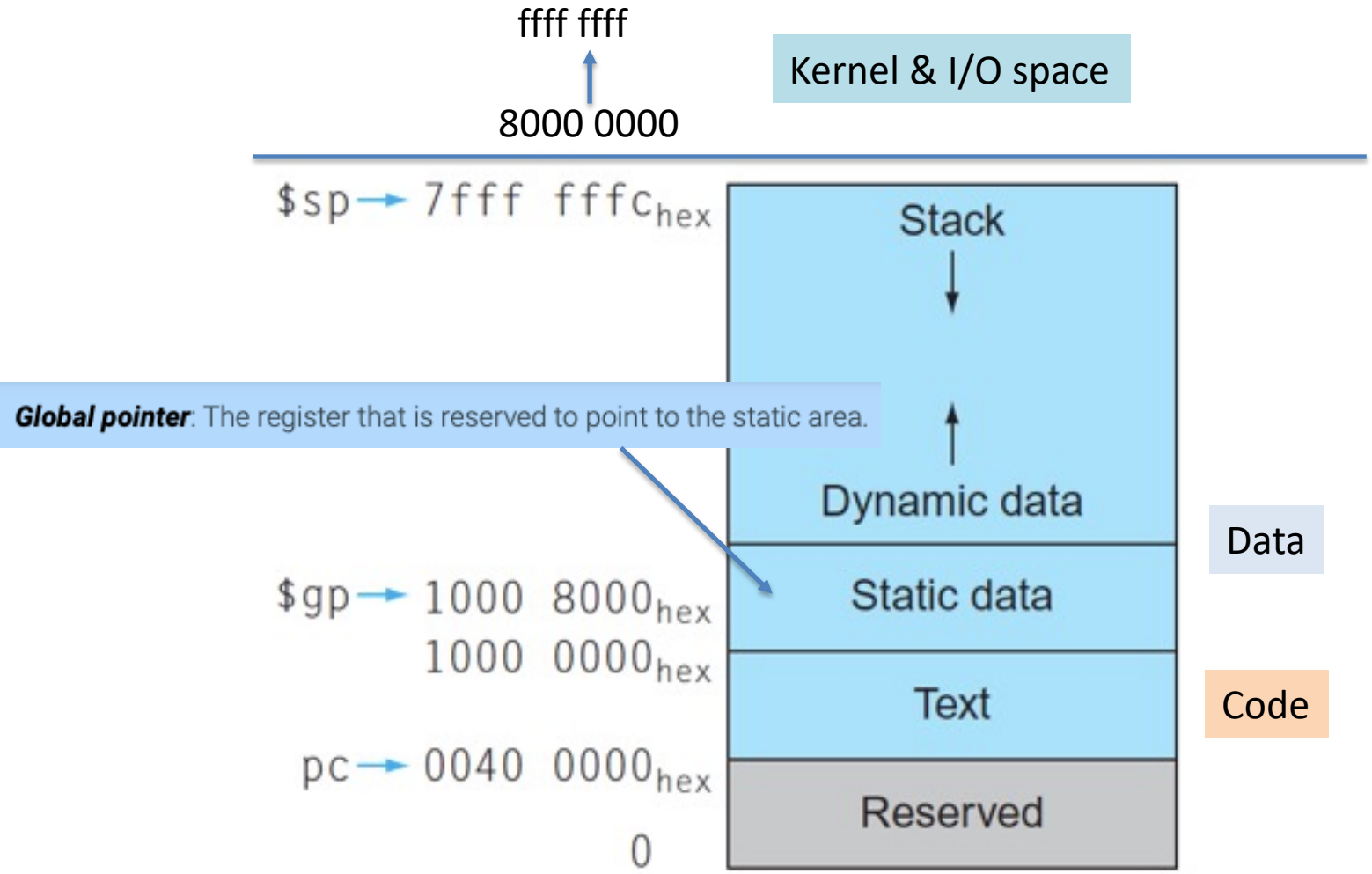
Patterson & Hennessy

Format		Address computation
(register)	($at)	contents of register
imm	+4	immediate
imm (register)	+4 ($at)	immediate + contents of register
label		address of label
label ± imm	Label +4	address of label + or – immediate
label ± imm (register)	Label +4 ($at)	address of label + or – (immediate + contents of register)

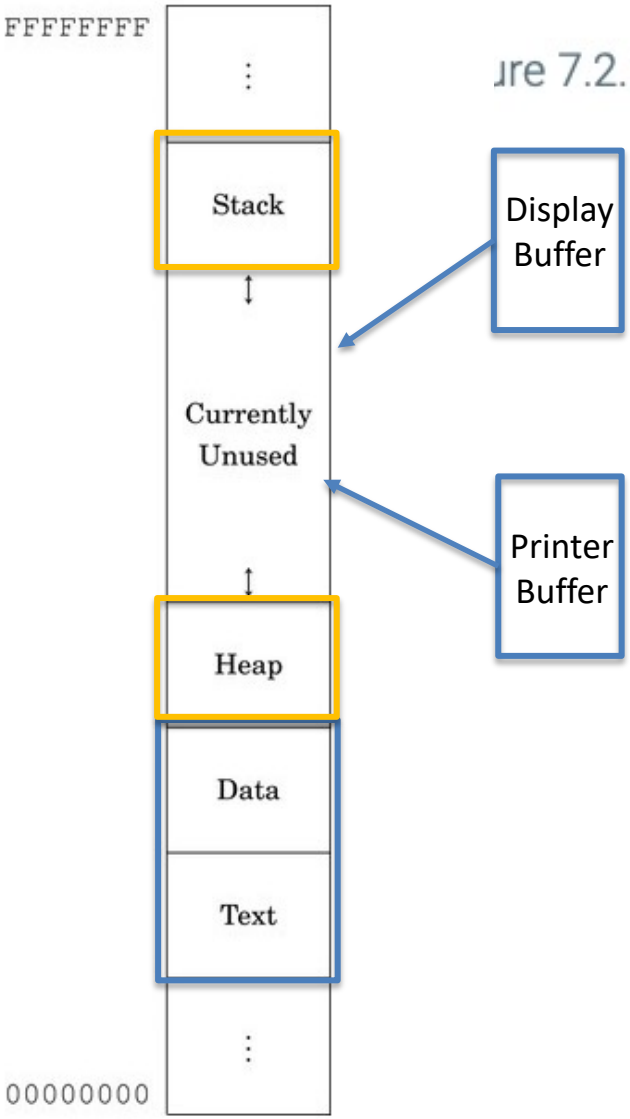
Memory Segment Model

Patterson & Hennessy

Figure 2.8.3: The MIPS memory allocation for program and data (COD Figure 2.13).



Memory Segments



Typical memory layout for a program with a 32-bit address space.

Display Buffer

Printer Buffer

ure 7.2.1: Object file (COD Figure A.2.1).

A UNIX assembler produces an object file with six distinct sections.

Object file header	Text segment	Data segment	Relocation information	Symbol table	Debugging information

MARS

Memory Map

MARS *(MIPS Assembler and Runtime Simulator)*

MIPS Memory Configuration

0xffffffff	memory map limit address
0xffffffff	kernel space high address
0xffff0000	MMIO base address
0xfffeffff	kernel data segment limit address
0x90000000	.kdata base address
0x8ffffffc	kernel text limit address
0x80000180	exception handler address
0x80000000	kernel space base address
0x80000000	.ktext base address
0x7fffffff	user space high address
0x7fffffff	data segment limit address
0x7ffffffc	stack base address
0x7fffeffc	stack pointer $sp
0x10040000	stack limit address
0x10040000	heap base address
0x10010000	.data base address
0x10008000	global pointer $gp
0x10000000	data segment base address
0x10000000	.extern base address
0x0ffffffc	text limit address
0x00400000	.text base address

Configuration
- (●) Default
- ( ) Compact, Data at Address 0
- ( ) Compact, Text at Address 0

MIPS Assembly

MIPS Lab 1

MARS 4.5 Help

MIPS | MARS | License | Bugs/Comments | Acknowledgements | Instruction Set Song

Operand Key for Example Instructions

`label, target`	any textual label
`$t1, $t2, $t3`	any integer register
`$f2, $f4, $f6`	*even-numbered* floating point register
`$f0, $f1, $f3`	*any* floating point register

Basic Instructions | Extended (pseudo) Instructions | Directives | Syscalls | Exceptions | Macros

`sltiu $t1,$t2,-100`	Set less than immediate unsigned : If $t2 is less than sign-extended 16-bit imme
`sltu $t1,$t2,$t3`	Set less than unsigned : If $t2 is less than $t3 using unsigned comparision, then
`sqrt.d $f2,$f4`	Square root double precision : Set $f2 to double-precision floating point square
`sqrt.s $f0,$f1`	Square root single precision : Set $f0 to single-precision floating point square
`sra $t1,$t2,10`	Shift right arithmetic : Set $t1 to result of sign-extended shifting $t2 right by
`srav $t1,$t2,$t3`	Shift right arithmetic variable : Set $t1 to result of sign-extended shifting $t2
`srl $t1,$t2,10`	Shift right logical : Set $t1 to result of shifting $t2 right by number of bits s
`srlv $t1,$t2,$t3`	Shift right logical variable : Set $t1 to result of shifting $t2 right by number
`sub $t1,$t2,$t3`	Subtraction with overflow : set $t1 to ($t2 minus $t3)
`sub.d $f2,$f4,$f6`	Floating point subtraction double precision : Set $f2 to double-precision floatin
`sub.s $f0,$f1,$f3`	Floating point subtraction single precision : Set $f0 to single-precision floatin
`subu $t1,$t2,$t3`	Subtraction unsigned without overflow : set $t1 to ($t2 minus $t3), no overflow
`sw $t1,-100($t2)`	Store word : Store contents of $t1 into effective memory word address
`swc1 $f1,-100($t2)`	Store word from Coprocesor 1 (FPU) : Store 32 bit value in $f1 to effective memor
`swl $t1,-100($t2)`	Store word left : Store high-order 1 to 4 bytes of $t1 into memory, starting with
`swr $t1,-100($t2)`	Store word right : Store low-order 1 to 4 bytes of $t1 into memory, starting with
`syscall`	Issue a system call : Execute the system call specified by value in $v0
`teq $t1,$t2`	Trap if equal : Trap if $t1 is equal to $t2
`teqi $t1,-100`	Trap if equal to immediate : Trap if $t1 is equal to sign-extended 16 bit immedia
`tge $t1,$t2`	Trap if greater or equal : Trap if $t1 is greater than or equal to $t2

MIPS Assembly

Hennessy & Patterson

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
addi	I	8	18	17		100		addi $s1,$s2,100
lw	I	35	18	17		100		lw $s1,100($s2)
sw	I	43	18	17		100		sw $s1,100($s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

offset

add $t0 $s1 $s2

add	$s1	$s2	$t0	unused	add
0	17	18	8	0	32

000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP122

MIPS I ISA

CPU

Wikipedia

ALU

Instruction name	Mnemonic	Format	Encoding					
Add	ADD	R	0_{10}	rs	rt	rd	0_{10}	32_{10}
Add Unsigned	ADDU	R	0_{10}	rs	rt	rd	0_{10}	33_{10}
Subtract	SUB	R	0_{10}	rs	rt	rd	0_{10}	34_{10}
Subtract Unsigned	SUBU	R	0_{10}	rs	rt	rd	0_{10}	35_{10}
And	AND	R	0_{10}	rs	rt	rd	0_{10}	36_{10}
Or	OR	R	0_{10}	rs	rt	rd	0_{10}	37_{10}
Exclusive Or	XOR	R	0_{10}	rs	rt	rd	0_{10}	38_{10}
Nor	NOR	R	0_{10}	rs	rt	rd	0_{10}	39_{10}
Set on Less Than	SLT	R	0_{10}	rs	rt	rd	0_{10}	42_{10}
Set on Less Than Unsigned	SLTU	R	0_{10}	rs	rt	rd	0_{10}	43_{10}
Add Immediate	ADDI	I	8_{10}	rs	rd	immediate		
Add Immediate Unsigned	ADDIU	I	9_{10}	$s	$d	immediate		
Set on Less Than Immediate	SLTI	I	10_{10}	$s	$d	immediate		
Set on Less Than Immediate Unsigned	SLTIU	I	11_{10}	$s	$d	immediate		
And Immediate	ANDI	I	12_{10}	$s	$d	immediate		
Or Immediate	ORI	I	13_{10}	$s	$d	immediate		
Exclusive Or Immediate	XORI	I	14_{10}	$s	$d	immediate		
Load Upper Immediate	LUI	I	15_{10}	0_{10}	$d	immediate		

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP122

MIPS32 ISA

CPU

Hennessy & Patterson

Arithmetic instructions [edit]

Mnemonic	Description
ADD	Add Word
ADDI	Add Immediate Word
ADDIU	Add Immediate Unsigned Word
ADDU	Add Unsigned Word
CLO	Count Leading Ones in Word
CLZ	Count Leading Zeros in Word
DIV	Divide Word
DIVU	Divide Unsigned Word
MADD	Multiply and Add Word to Hi, Lo
MADDU	Multiply and Add Unsigned Word to Hi, Lo
MSUB	Multiply and Subtract Word to Hi, Lo
MSUBU	Multiply and Subtract Unsigned Word to Hi, Lo
MUL	Multiply Word to GPR
MULT	Multiply Word
MULTU	Multiply Unsigned Word

X*Y +A

Logical instruction [edit]

Mnemonic	Description
AND	And
ANDI	And Immediate
LUI	Load Upper Immediate
NOR	Not Or
OR	Or
ORI	Or Immediate
XOR	Exclusive Or
XORI	Exclusive Or Immediate

SEB	Sign-Extend Byte
SEH	Sign-Extend Halftword
SLT	Set on Less Than
SLTI	Set on Less Than Immediate
SLTIU	Set on Less Than Immediate Unsigned
SLTU	Set on Less Than Unsigned
SUB	Subtract Word
SUBU	Subtract Unsigned Word

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP122

MIPS I ISA

CPU

Wikipedia

Shift

Instruction name	Mnemonic	Format	Encoding					
Shift Left Logical	SLL	R	0_{10}	0_{10}	rt	rd	sa	0_{10}
Shift Right Logical	SRL	R	0_{10}	0_{10}	rt	rd	sa	2_{10}
Shift Right Arithmetic	SRA	R	0_{10}	0_{10}	rt	rd	sa	3_{10}
Shift Left Logical Variable	SLLV	R	0_{10}	rs	rt	rd	0_{10}	4_{10}
Shift Right Logical Variable	SRLV	R	0_{10}	rs	rt	rd	0_{10}	6_{10}
Shift Right Arithmetic Variable	SRAV	R	0_{10}	rs	rt	rd	0_{10}	7_{10}

Mult
Div

Instruction name	Mnemonic	Format	Encoding					
Move from HI	MFHI	R	0_{10}	0_{10}	0_{10}	rd	0_{10}	16_{10}
Move to HI	MTHI	R	0_{10}	rs	0_{10}	0_{10}	0_{10}	17_{10}
Move from LO	MFLO	R	0_{10}	0_{10}	0_{10}	rd	0_{10}	18_{10}
Move to LO	MTLO	R	0_{10}	rs	0_{10}	0_{10}	0_{10}	19_{10}
Multiply	MULT	R	0_{10}	rs	rt	0_{10}	0_{10}	24_{10}
Multiply Unsigned	MULTU	R	0_{10}	rs	rt	0_{10}	0_{10}	25_{10}
Divide	DIV	R	0_{10}	rs	rt	0_{10}	0_{10}	26_{10}
Divide Unsigned	DIVU	R	0_{10}	rs	rt	0_{10}	0_{10}	27_{10}

MIPS32 ISA

Move instructions [edit]

Mnemonic	Description
MFHI	Move From HI Register
MFLO	Move From LO Register
MOVF	Move Conditional on Floating Point False
MOVN	Move Conditional on Not Zero
MOVT	Move Conditional on Floating Point True
MOVZ	Move Conditional on Zero
MTHI	Move To HI Register
MTLO	Move To LO Register
RDHWR	Read Hardware Register

Shift instructions [edit]

Mnemonic	Description
ROTR	Rotate Word Right
ROTRV	Rotate Word Right Variable
SLL	Shift Word Left Logical
SLLV	Shift Word Left Logical Variable
SRA	Shift Word Right Arithmetic
SRAV	Shift Word Right Arithmetic Variable
SRL	Shift Word Right Logical
SRLV	Shift Word Right Logical Variable

Shift & Bit Logic

Hennessy & Patterson

Figure 2.6.1: C and Java logical operators and their corresponding MIPS instructions (COD Figure 2.8).

MIPS implements NOT using a NOR with one operand being zero.

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

MIPS I ISA

CPU

Wikipedia

LOAD

Instruction name	Mnemonic	Format	Encoding			
Load Byte	LB	I	32_{10}	rs	rt	offset
Load Halfword	LH	I	33_{10}	rs	rt	offset
Load Word Left	LWL	I	34_{10}	rs	rt	offset
Load Word	LW	I	35_{10}	rs	rt	offset
Load Byte Unsigned	LBU	I	36_{10}	rs	rt	offset
Load Halfword Unsigned	LHU	I	37_{10}	rs	rt	offset
Load Word Right	LWR	I	38_{10}	rs	rt	offset
Store Byte STORE	SB	I	40_{10}	rs	rt	offset
Store Halfword	SH	I	41_{10}	rs	rt	offset
Store Word Left	SWL	I	42_{10}	rs	rt	offset
Store Word	SW	I	43_{10}	rs	rt	offset
Store Word Right	SWR	I	46_{10}	rs	rt	offset

MIPS32 ISA

CPU

Hennessy & Patterson

LOAD

LHU	Load Halfword Unsigned
LHUE	Load Halfword Unsigned EVA
LL	Load Linked Word
LLE	Load Linked Word-EVA
LW	Load Word
LWE	Load Word EVA
LWL	Load Word Left
LWLE	Load Word Left EVA
LWR	Load Word Right
LWRE	Load Word Right EVA
PREF	Prefetch
PREFE	Prefetch-EVA

STORE

SB	Store Byte
SBE	Store Byte EVA
SC	Store Conditional Word
SCE	Store Conditional Word EVA
SH	Store Halfword
SHE	Store Halfword EVA
SW	Store Word
SWE	Store Word EVA
SWL	Store Word Left
SWLE	Store Word Left EVA
SWR	Store Word Right
SWRE	Store Word Right EVA
SYNC	Synchronize Shared Memory
SYNCI	Synchronize Caches to Make Instruction Writes Effective

- Multiprocessing extensions

MIPS I ISA

Jump + Branch

Instruction name	Mnemonic	Format	Encoding					
Jump Register	JR	R	0_{10}	rs	0_{10}	0_{10}	0_{10}	8_{10}
Jump and Link Register	JALR	R	0_{10}	rs	0_{10}	rd	0_{10}	9_{10}
Branch on Less Than Zero	BLTZ	I	1_{10}	rs	0_{10}	offset		
Branch on Greater Than or Equal to Zero	BGEZ	I	1_{10}	rs	1_{10}	offset		
Branch on Less Than Zero and Link	BLTZAL	I	1_{10}	rs	16	offset		
Branch on Greater Than or Equal to Zero and Link	BGEZAL	I	1_{10}	rs	17	offset		
Jump	J	J	2_{10}	instr_index				
Jump and Link	JAL	J	3_{10}	instr_index				
Branch on Equal	BEQ	I	4_{10}	rs	rt	offset		
Branch on Not Equal	BNE	I	5_{10}	rs	rt	offset		
Branch on Less Than or Equal to Zero	BLEZ	I	6_{10}	rs	0_{10}	offset		
Branch on Greater Than Zero	BGTZ	I	7_{10}	rs	0_{10}	offset		

MIPS32 ISA

CPU

Hennessy & Patterson

Branch instructions [edit]

Note that all the likely branches have been obsoleted; they will be removed in future revisions of the MIPS32 architecture.
instructions.

Mnemonic	Description
B	Unconditional Branch
BAL	Branch and Link
BEQ	Branch on Equal
BGEZ	Branch on Greater Than or Equal to Zero
BGEZAL	Branch on Greater Than or Equal to Zero and Link
BGTZ	Branch on Greater Than Zero
BLEZ	Branch on Less Than or Equal to Zero
BLTZ	Branch on Less Than Zero
BLTZAL	Branch on Less Than Zero and Link
BNE	Branch on Not Equal
~~BEQL~~	Branch on Equal Likely
~~BGEZALL~~	Branch on Greater Than or Equal to Zero and Link Likely
~~BGEZL~~	Branch on Greater Than or Equal to Zero Likely
~~BGTZL~~	Branch on Greater Than Zero Likely
~~BLEZL~~	Branch on Less Than or Equal to Zero Likely
~~BLTZALL~~	Branch on Less Than Zero and Link Likely
~~BLTZL~~	Branch on Less Than Zero Likely
~~BNEL~~	Branch on Not Equal Likely

Jump instructions [edit]

Mnemonic	Description
J	Jump
JAL	Jump and Link
JALR	Jump and Link Register
JALR.HB	Jump and Link Register with Hazard Barrier
JALX	Jump and Link Exchange
JR	Jump Register
JR.HB	Jump Register with Hazard Barrier

Control instructions [edit]

Mnemonic	Description
EHB	Execution Hazard Barrier
NOP	No Operation
PAUSE	Wait for LLBit to Clear
SSNOP	Superscalar No Operation

MIPS I ISA

CP1

Wikipedia

FP

Name	Instruction syntax	Meaning	opcode	rs	rt	rd	sham	funct
Floating-Point Add	`add.s $x,$y,$z`	`$x = $y + $z`	17_{10}	0_{10}	$z	$y	$x	0_{10}
Floating-Point Subtract	`sub.s $x,$y,$z`	`$x = $y - $z`	17_{10}	0_{10}	$z	$y	$x	1_{10}
Floating-Point Multiply	`mul.s $x,$y,$z`	`$x = $y * $z`	17_{10}	0_{10}	$z	$y	$x	2_{10}
Floating-Point Divide	`div.s $x,$y,$z`	`$x = $y / $z`	17_{10}	0_{10}	$z	$y	$x	3_{10}
Floating-Point Add	`add.d $x,$y,$z`	`$x = $y + $z`	17_{10}	1_{10}	$z	$y	$x	0_{10}
Floating-Point Subtract	`sub.d $x,$y,$z`	`$x = $y - $z`	17_{10}	1_{10}	$z	$y	$x	1_{10}
Floating-Point Multiply	`mul.d $x,$y,$z`	`$x = $y * $z`	17_{10}	1_{10}	$z	$y	$x	2_{10}
Floating-Point Divide	`div.d $x,$y,$z`	`$x = $y / $z`	17_{10}	1_{10}	$z	$y	$x	3_{10}
Floating-Point Compare (eq,ne,lt,le,gt,ge)	`c.lt.s $f2,$f4`	`cond = ($f2 < $f4)`						
Floating-Point Compare (eq,ne,lt,le,gt,ge)	`c.lt.d $f2,$f4`	`cond = ($f2 < $f4)`						

System

Instruction name	Mnemonic	Format	Encoding		
System Call	SYSCALL	?	0_{10}	Code	12_{10}
Breakpoint	BREAK	?	0_{10}	Code	13_{10}

MIPS I ISA

Data transfer [edit]

Name	Instruction syntax	Meaning	Format	opcode	funct	Notes/Encoding
Load word coprocessor	`lwcZ $x,CONST ($y)`	`Coprocessor[Z].DataRegister[$x] = Memory[$y + CONST]`	I			Loads the 4 byte word stored from: MEM[$y+CONST] into a Coprocessor data register. Sign extension.
Store word coprocessor	`swcZ $x,CONST ($y)`	`Memory[$y + CONST] = Coprocessor[Z].DataRegister[$x]`	I			Stores the 4 byte word held by a Coprocessor data register into: MEM[$y+CONST]. Sign extension.

Branch [edit]

Name	Instruction syntax	Meaning	Format	opcode	funct	Notes/Encoding
Branch on FP True	`bclt 100`	`if (cond) goto PC+4+100;`				PC relative branch if FP condition
Branch on FP False	`bclf 100`	`if (!cond) goto PC+4+100;`				PC relative branch if not condition

MIPS32 ISA

Coprocessor 1

Hennessy & Patterson

FPU instructions [edit]

FPU

Arithmetic instructions [edit]

Mnemonic	Description
ABS.fmt	Floating Point Absolute Value
ADD.fmt	Floating Point Add
DIV.fmt	Floating Point Divide
MADD.fmt	Floating Point Multiply Add
MSUB.fmt	Floating Point Multiply Subtract
MUL.fmt	Floating Point Multiply
NEG.fmt	Floating Point Negate
NMADD.fmt	Floating Point Negative Multiply Add
NMSUB.fmt	Floating Point Negative Multiply Subtract
RECIP.fmt	Reciprocal Approximation
RSQRT.fmt	Reciprocal Square Root Approximation
SQRT.fmt	Floating Point Square Root
SUB.fmt	Floating Point Subtract

Branch instructions [edit]

Mnemonic	Description
BC1F	Branch on FP False
BC1T	Branch on FP True
~~BC1FL~~	Branch on FP False Likely
~~BC1TL~~	Branch on FP True Likely

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP122

MIPS32 ISA

Coprocessor 1

Hennessy & Patterson

FPA

Memory control instructions [edit]

Mnemonic	Description
LDC1	Load Doubleword to Floating Point
LDXC1	Load Doubleword Indexed to Floating Point
LUXC1	Load Doubleword Indexed Unaligned to Floating Point
LWC1	Load Word to Floating Point
LWXC1	Load Word Indexed to Floating Point
PREFX	Prefetch Indexed
SDC1	Store Doubleword from Floating Point
SDXC1	Store Doubleword Indexed from Floating Point
SUXC1	Store Doubleword Indexed Unaligned from Floating Point
SWC1	Store Word from Floating Point
SWXC1	Store Word Indexed from Floating Point

Move instructions [edit]

Mnemonic	Description
CFC1	Move Control Word from Floating Point
CTC1	Move Control Word to Floating Point
MFC1	Move Word from Floating Point
MFHC1	Move Word from High Half of Floating Point Register
MOV.fmt	Floating Point Move
MOVF.fmt	Floating Point Move Conditional on Floating Point False
MOVN.fmt	Floating Point Move Conditional on Not Zero
MOVT.fmt	Floating Point Move Conditional on Floating Point True
MOVZ.fmt	Floating Point Move Conditional on Zero

MIPS32 ISA

Coprocessor 1

FPU

Hennessy & Patterson

Convert instructions [edit]

Mnemonic	Description
ALNV.PS	Floating Point Align Variable
CEIL.L.fmt	Floating Point Ceiling Convert to Long Fixed Point
CEIL.W.fmt	Floating Point Ceiling Convert to Word Fixed Point
CVT.D.fmt	Floating Point Convert to Double Floating Point
CVT.L.fmt	Floating Point Convert to Long Fixed Point
CVT.PS.S	Floating Point Convert Pair to Paired Single
CVT.S.PL	Floating Point Convert Pair Lower to Single Floating Point
CVT.S.PU	Floating Point Convert Pair Upper to Single Floating Point
CVT.S.fmt	Floating Point Convert to Single Floating Point
CVT.W.fmt	Floating Point Convert to Word Fixed Point
FLOOR.L.fmt	Floating Point Floor Convert to Long Fixed Point
FLOOR.W.fmt	Floating Point Floor Convert to Word Fixed Point
PLL.PS	Pair Lower Lower
PLU.PS	Pair Lower Upper
PUL.PS	Pair Upper Lower
PUU.PS	Pair Upper Upper
ROUND.L.fmt	Floating Point Round to Long Fixed Point
ROUND.W.fmt	Floating Point Round to Word Fixed Point
TRUNC.L.fmt	Floating Point Truncate to Long Fixed Point
TRUNC.W.fmt	Floating Point Truncate to Word Fixed Point

Compare instructions [edit]

Mnemonic	Description
C.cond.fmt	Floating Point Compare

MIPS I ISA

Formats | Instruction Details

MIPS Instructions

Instruction Formats:

Instruction formats: all 32 bits wide (one word):

```
                    6        5        5        5        5        6
                +--------+--------+-------+-------+------+--------+
R-type format   | Op-code|   Rs   |  Rt   |  Rd   |  SA  |Funct-code|
                +--------+--------+-------+-------+------+--------+

                    6        5        5                16
                +--------+--------+-------+------------------------+
I-type format   |Op-code |   Rs   |  Rt   | 2's complement constant|
                +--------+--------+-------+------------------------+

                    6                          26
                +--------+-------------------------------------------+
J-type format   | Op-code|                jump_target                |
                +--------+-------------------------------------------+
                    ^                                                ^
                    |                                                |
            bit 31                                              bit 0
```

Instructions and their formats

General notes:

a. **R_s**, **R_t**, and **R_d** specify general purpose registers
b. Square brackets (**[]**) indicate "the contents of"
c. **[PC]** specifies the address of the instruction in execution
d. **I** specifies part of instruction and its subscripts indicate bit positions of sub-fields
e. || indicates concatenation of bit fields
f. Superscripts indicate repetition of a binary value
g. **M{i}** is a value (contents) of the word beginning at the memory address i
h. **m{i}** is a value (contents) of the byte at the memory address i
i. all integers are in 2's complement representation if not indicated as unsigned

1. *addition with overflow:* **add** instruction

```
                +--------+-------+-------+-------+-------+--------+
R-type format   | 000000 |  Rs   |  Rt   |  Rd   | 00000 | 100000 |
                +--------+-------+-------+-------+-------+--------+
```

Effects of the instruction: R_d <-- [R_s] + [R_t]; PC <-- [PC] + 4

(If overflow then exception processing)

Assembly format: **add R_d,R_s,R_t**

MIPS I ISA

Add/Sub | Instruction Details

1. *addition with overflow:* **add** instruction

R-type format	000000	R_s	R_t	R_d	00000	100000

Effects of the instruction: R_d <-- [R_s] + [R_t]; PC <-- [PC] + 4
(If overflow then exception processing)

Assembly format: **add R_d,R_s,R_t**

2. *add without overflow:* **addu** instruction

Identical as **add** instruction, except:
- funct=33_{dec}
- overflow ignored

3. *subtract with overflow:* **sub** instruction

R-type format	000000	R_s	R_t	R_d	00000	100010

Effects of the instruction: R_d <-- [R_s] - [R_t]; PC <-- [PC] + 4
(If overflow then exception processing)

Assembly format: **sub R_d,R_s,R_t**

4. *subtract without overflow:* **subu** instruction

Identical as **sub** instruction, except:
- funct=35_{dec}
- overflow ignored

MIPS I ISA

5. *multiply:* **mul** instruction

R-type format	000000	R_a	R_c	00000	00000	011000

Effects of the instruction: Hi||Lo <-- $[R_a]$ * $[R_c]$; PC <-- [PC] + 4

Assembly format: **mult R_a,R_c**

6. *unsigned multiply:* **mulu** instruction

Identical as **mut** instruction, except:

- funct = 25_{dec}
- contents of R_a and R_c are considered as unsigned integers

7. *divide:* **div** instruction

R-type format	000000	R_a	R_c	00000	00000	011010

Effects of the instruction: Lo <-- $[R_a]$ / $[R_c]$; Hi <-- $[R_a]$mod$[R_c]$
PC <-- [PC] + 4

Assembly format: **div R_a,R_c**

8. *unsigned divide:* **divu** instruction

Identical as **div** instruction, except:

- funct = 27_{dec}
- contents of R_a and R_c are considered as unsigned integers

MIPS I ISA

22. load word: **lw** instruction

```
                +--------+-------+-------+-----------------------+
I-type format:  | 100011 | R_s   | R_t   |        offset         |
                +--------+-------+-------+-----------------------+
Effects of the instruction: R_t <-- M{ [R_s] + [I_15]^16 || [I_15..0] }
                            PC <-- [PC] + 4
      (If an illegal memory address then exception processing)
Assembly format: lw R_t,offset(R_s)
```

23. store word: **sw** instruction

```
                +--------+-------+-------+-----------------------+
I-type format:  | 101011 | R_s   | R_t   |        offset         |
                +--------+-------+-------+-----------------------+

Effects of the instruction: M{ [R_s] + [I_15]^16 || [I_15..0] } <-- [R_t]
                            PC <-- [PC] + 4
      (If an illegal memory address then exception processing)
Assembly format: sw R_t,offset(R_s)
```

27. *load upper immediate:* **lui** instruction

```
                +--------+-------+-------+-----------------------+
I-type format:  | 001111 | 00000 | R_t   |       immediate       |
                +--------+-------+-------+-----------------------+
Effects of the instruction: R_t <-- [I_15-0] || 0^16; PC <-- [PC] + 4
Assembly format: lui R_t,immediate
```

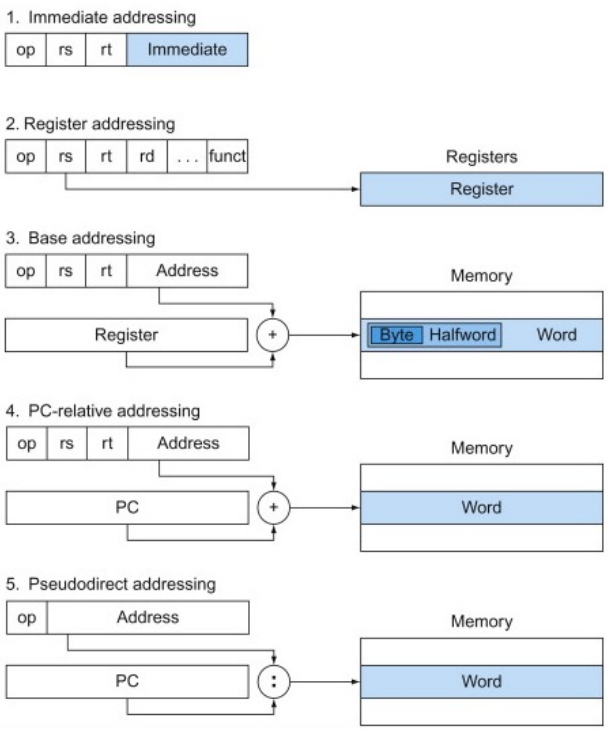
Addressing Modes

Patterson & Hennessy

MIPS addressing mode summary

Multiple forms of addressing are generically called *addressing modes*. The figure below shows how operands are identified for each addressing mode. The MIPS addressing modes are the following:

1. ***Immediate addressing***: The operand is a constant within the instruction itself
2. ***Register addressing***: The operand is a register
3. ***Base addressing / displacement addressing***: The operand is at the memory location whose address is the sum of a register and a constant in the instruction
4. ***PC-relative addressing***: The branch address is the sum of the PC and a constant in the instruction
5. ***Pseudodirect addressing***: The jump address is the 26 bits of the instruction concatenated with the upper bits of the PC



MIPS Assembly: Address Modes

P&H Ch2

I

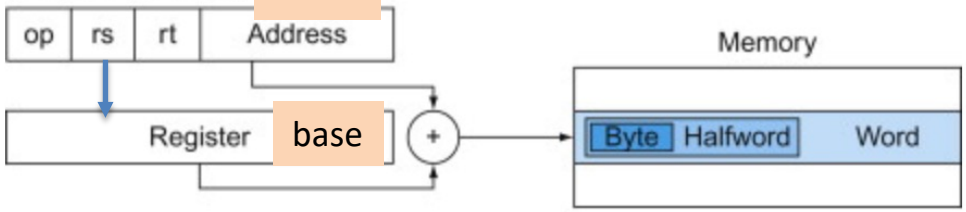
1. Immediate addressing

R

2. Register addressing

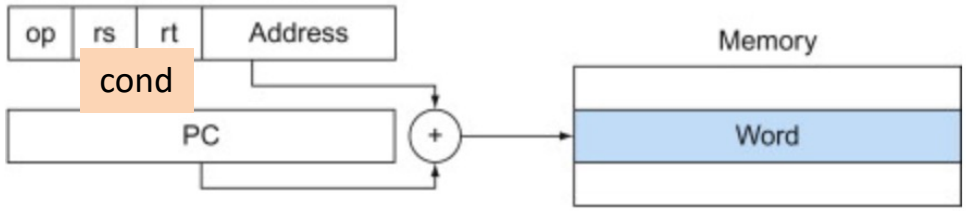
I

3. Base addressing



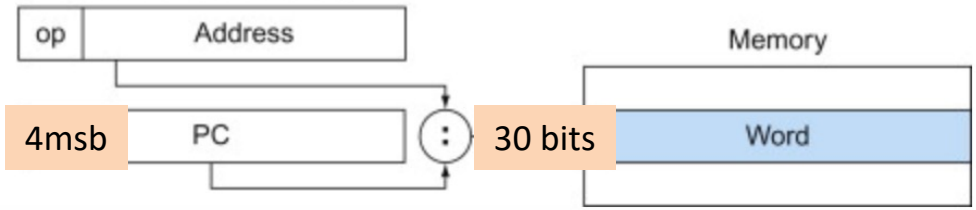
I

4. PC-relative addressing



J

5. Pseudodirect addressing



MIPS Assembly: Address Modes

lw *offset*(reg) *primitive*

MARS 4.5 Help

MIPS | MARS | License | Bugs/Comments | Acknowledgements | Instruction Set Song

Load & Store addressing mode, basic instructions

-100($t2)	sign-extended 16-bit integer added to contents of $t2

Load & Store addressing modes, pseudo instructions

($t2)	contents of $t2
-100	signed 16-bit integer
100	unsigned 16-bit integer
100000	signed 32-bit integer
100($t2)	zero-extended unsigned 16-bit integer added to contents of $t2
100000($t2)	signed 32-bit integer added to contents of $t2
label	32-bit address of label
label($t2)	32-bit address of label added to contents of $t2
label+100000	32-bit integer added to label's address
label+100000($t2)	sum of 32-bit integer, label's address, and contents of $t2

Load *Pseudo* Ops

The MIPS assembler (and SPIM) synthesizes the more complex addressing modes by producing one or more instructions before the load or store to compute a complex address. For example, suppose that the label **`table`** *referred to memory location 0x10000004 and a program contained the instruction*

```
ld $a0, table + 4($a1)
```

The assembler would translate this instruction into the instructions

```
lui $at, 4096
addu $at, $at, $a1
lw $a0, 8($at)
```

The first instruction loads the upper bits of the label's address into register **`$at`**, *which is the register that the assembler reserves for its own use. The second instruction adds the contents of register* **`$a1`** *to the label's partial address. Finally, the load instruction uses the hardware address mode to add the sum of the lower bits of the label's address and the offset from the original instruction to the value in register* **`$at`**.

la = load address (32-bit)

Immediates (16/32-bit)

1. Immediate addressing

op	rs	rt	Immediate

32-bit immediate operands

PARTICIPATION ACTIVITY 2.10.1: The lui instruction, and how to load a 32-bit constant (COD Figure 2.17 (The effect of the lui instruction)).

Start 2x speed

lui

```
lui $t0, 255   # $t0 is register 8
```

lui instruction:

001111	00000	01000	0000 0000 1111 1111

$t0 | 0000 0000 1111 1111 | 0000 0000 0000 0000 |

How load following 32-bit constant into $s0?

0000 0000 0011 1101 (61 in decimal) 0000 1001 0000 0000 (2304 in decimal)

High HW

```
lui $s0, 61
ori $s0, $s0, 2304
```

Low HW

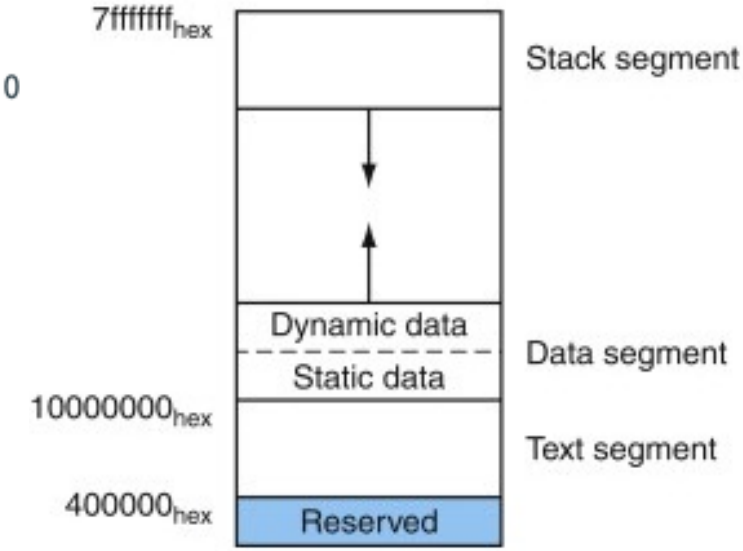
$s0 | 0000 0000 0011 1101 | 0000 0000 0000 0000 |

$s0 | 0000 0000 0011 1101 | 0000 1001 0000 0000 |

Load: lui vs. $gp

Because the data segment begins far above the program at address 10000000_{hex}, load and store instructions cannot directly reference data objects with their 16-bit offset fields (see COD Section 2.5 (Representing Instructions in the Computer)). For example, to load the word in the data segment at address 10010020_{hex} into register **$v0** requires two instructions:

```
lui $s0, 0x1001      # 0x1001 means 1001 base 16
lw $v0, 0x0020($s0) # 0x10010000 + 0x0020 = 0x10010020
```



To avoid repeating the **lui** instruction at every load and store, MIPS systems typically dedicate a register (**$gp**) as a *global pointer* to the static data segment. This register contains address 10008000_{hex}, so load and store instructions can use their signed 16-bit offset fields to access the first 64 KB of the static data segment. With this global pointer, we can rewrite the example as a single instruction:

```
lw $v0, 0x8020($gp)
```

Of course, a global pointer register makes addressing locations 10000000_{hex} – 10010000_{hex} faster than other heap locations. The MIPS compiler usually stores *global variables* in this area, because these variables have fixed locations and fit better than other global data, such as arrays.

MIPS I ISA

28. *branch on equal:* **beq** instruction

```
                +--------+-------+-------+-----------------------+
I-type format:  | 000100 | R_s   | R_t   |        offset         |
                +--------+-------+-------+-----------------------+
```

Effects of the instruction:

if $[R_s] = [R_t]$ then $PC \leftarrow [PC] + 4 + ([I_{15}]^{14} \,||\, [I_{15..0}] \,||\, 0^2)$

(i.e. $PC \leftarrow [PC] + 4 + 4*offset$)

else $PC \leftarrow [PC] + 4$

Assembly format: **beq R_s,R_t,offset**

32. *branch on less than zero:* **bltz** instruction

```
                +--------+-------+-------+-----------------------+
I-type format:  |000001  |  R_s  | 00000 |        offset         |
                +--------+-------+-------+-----------------------+
```

Effects of the instruction:

if $[R_s] < 0$ then $PC \leftarrow [PC] + 4 + ([I_{15}]^{14} \,||\, [I_{15..0}] \,||\, 0^2)$

else $PC \leftarrow [PC] + 4$

Assembly format: **bltz R_s,offset**

33. *jump:* **j** instruction

```
                +--------+-------------------------------------+
J-type format   |000010  |              jump_target            |
                +--------+-------------------------------------+
```

Effects of the instruction: $PC \leftarrow [PC_{31..28}] \,||\, [I_{25..0}] \,||\, 0^2$

Assembly format: **j jump_target**

MIPS I ISA

Exception Handling

When a condition for any exception (overflow, illegal op-code, division by zero, etc.) occurs the following hardware exception processing is performed:

```
EPC <-- [PC]
                 /
                 |  0^28 || 1010    if illegal op-code (10)
Cause_Reg <--    |  0^28 || 1100    if overflow (12)
                 |  0^29 || 100     if illegal memory address (4)
                 \  ............   etc.
PC <-- 80000180hex
```

40. *move from EPC:* **mfepc** instruction

R-type format	010000	00000	R_t	01110	00000	000000

Effects of the instruction: R_d <-- [EPC]; PC <-- [PC] + 4
Assembly format: **mfepc** $\mathbf{R_t}$ (This is mfc0 Rt,CP0reg14)

41. *move from Cause_Reg:* **mfco** instruction

R-type format	010000	00000	R_d	01101	00000	000000

Effects of the instruction: R_d <-- [Cause_Reg]; PC <-- [PC] + 4

MIPS I ISA

Floating Point Instructions

42. load word *into co-processor 1:* **lwc1** instruction

```
                +--------+-------+-------+-----------------------+
I-type format:  | 110001 | R_s   |  f_t  |        offset         |
                +--------+-------+-------+-----------------------+
```

Effects of the instruction: f_t <-- $M\{[R_s] + [I_{15}]^{16} \,||\, [I_{15..0}]\}$
PC <-- [PC] + 4

Assembly format: **lwc1 f_t,offset(R_s)**

43. store word from co-processor 1: **swc1** instruction

```
                +--------+-------+-------+-----------------------+
I-type format:  | 111001 | R_s   |  f_t  |        offset         |
                +--------+-------+-------+-----------------------+
```

Effects of the instruction: $M\{[R_s] + [I_{15}]^{16} \,||\, [I_{15..0}]\}$ <-- $[f_t]$
PC <-- [PC] + 4

Assembly format: **swc1 f_t,offset(R_s)**

MIPS I ISA

FP Instruction Details

44. *addition single precision:* **add.s** instruction

R-type format	010001	00000	f_t	f_s	f_d	000000

Effects of the instruction: f_d <-- [f_s] + [f_t]; PC <-- [PC] + 4
(If overflow then exception processing)

Assembly format: **add.s** $\mathbf{R_d, R_s, R_t}$

45. *addition double precision:* **add.d** instruction

R-type format	010001	00001	f_t	f_s	f_d	0000000

Effects of the instruction: $f_d||f_{d+1}$ <-- [f_s]||[f_{s+1}] + [f_t]||[f_{t+1}];
PC <-- [PC] + 4
(If overflow then exception processing)

Assembly format: add.d $\mathbf{f_d, f_s, f_t}$

45. *subtract single precision:* **sub.s** instruction
Similar as add.s but with funct=1

46. *subtract double precision:* **sub.d** instruction
Similar as add.d but with funct=1

Control CP0

7.7 Exceptions and interrupts

Hennessy & Patterson

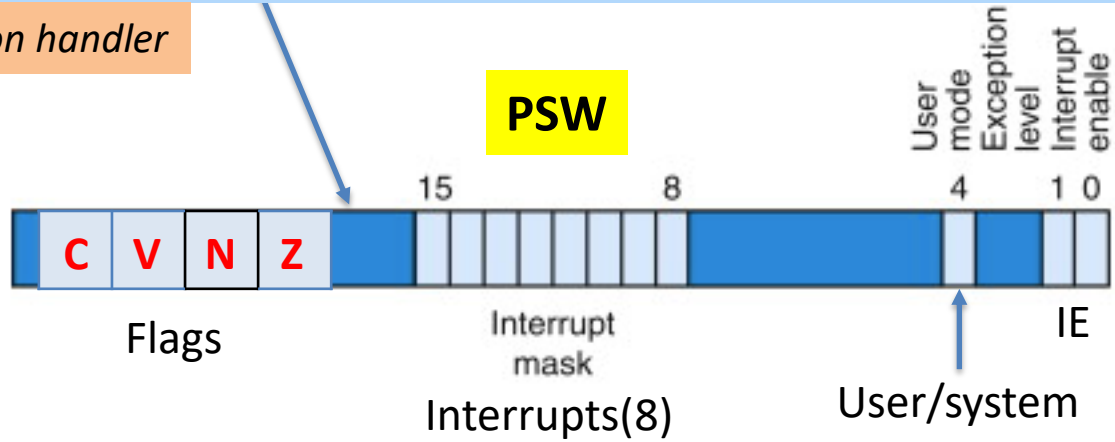
Figure 7.7.1: Coprocessor 0 registers.

Register name	Register number	Usage
BadVAddr	8	memory address at which an offending memory reference occurred
Count	9	timer
Compare	11	value compared against timer that causes interrupt when they match
Status	12	interrupt mask and enable bits
Cause	13	exception type and pending interrupt bits
EPC	14	address of instruction that caused exception
Config	16	configuration of machine

Figure 7.7.2: The status register (COD Figure

Interrupt handler: A piece of code that is run as a result of an exception or an interrupt.

Exception handler



Exceptions (EPC)

Hennessy & Patterson

Figure 7.7.3: The cause register (COD Figure A.7.2).

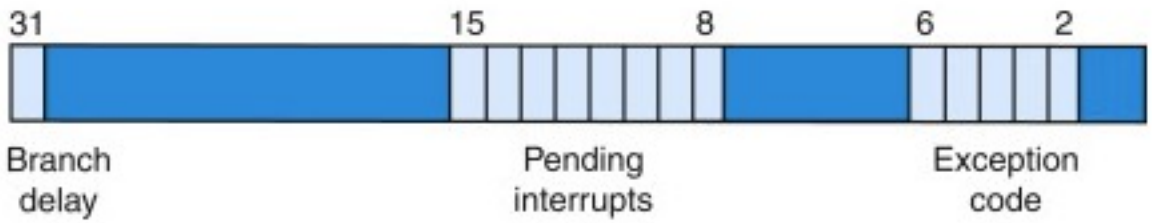


Figure 7.7.4: Causes of exceptions.

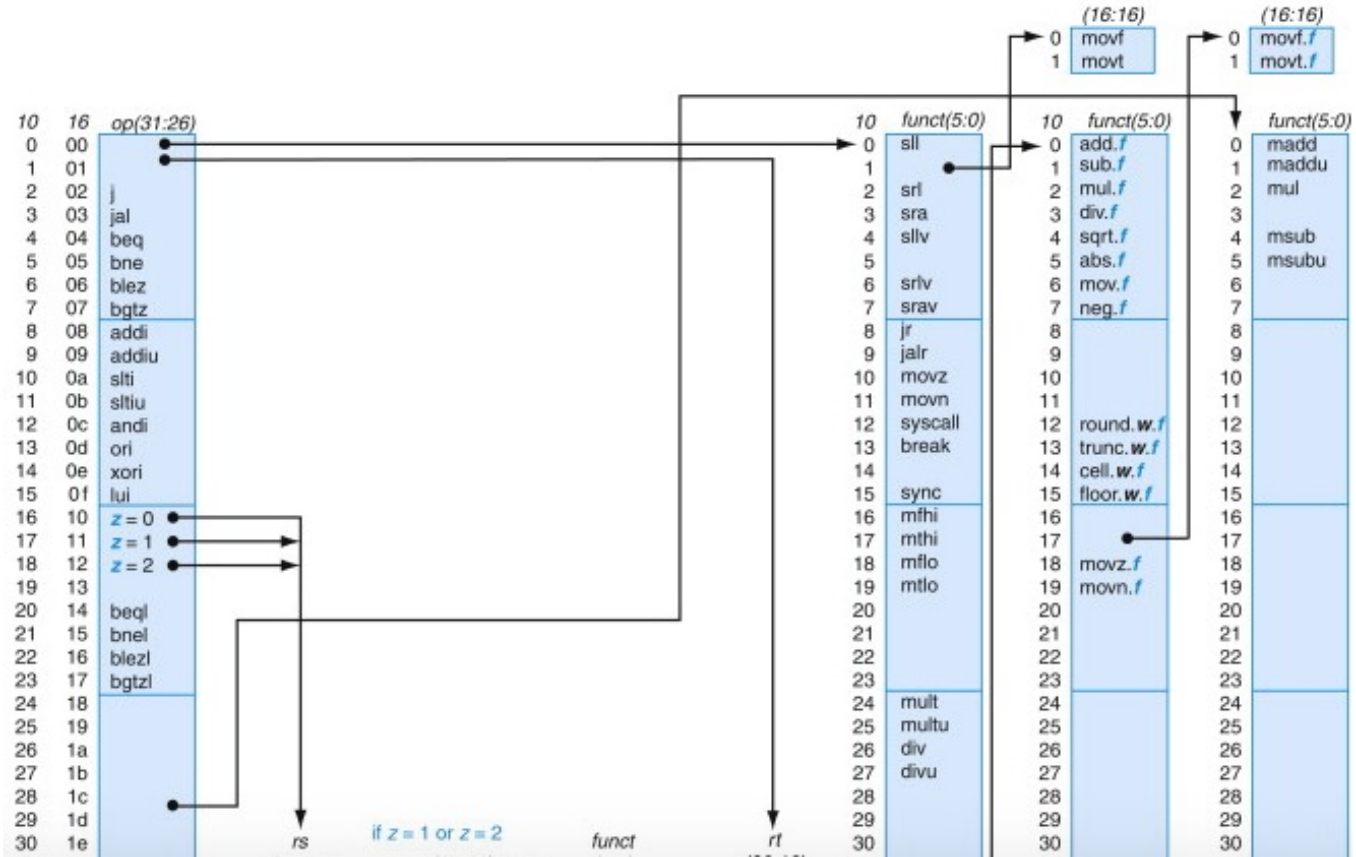
Number	Name	Cause of exception
0	Int	interrupt (hardware)
4	AdEL	address error exception (load or instruction fetch)
5	AdES	address error exception (store)
6	IBE	bus error on instruction fetch
7	DBE	bus error on data load or store
8	Sys	syscall exception
9	Bp	breakpoint exception
10	RI	reserved instruction exception
11	CpU	coprocessor unimplemented
12	Ov	arithmetic overflow exception
13	Tr	trap
15	FPE	floating point

Opcode Map

Hennessy & Patterson

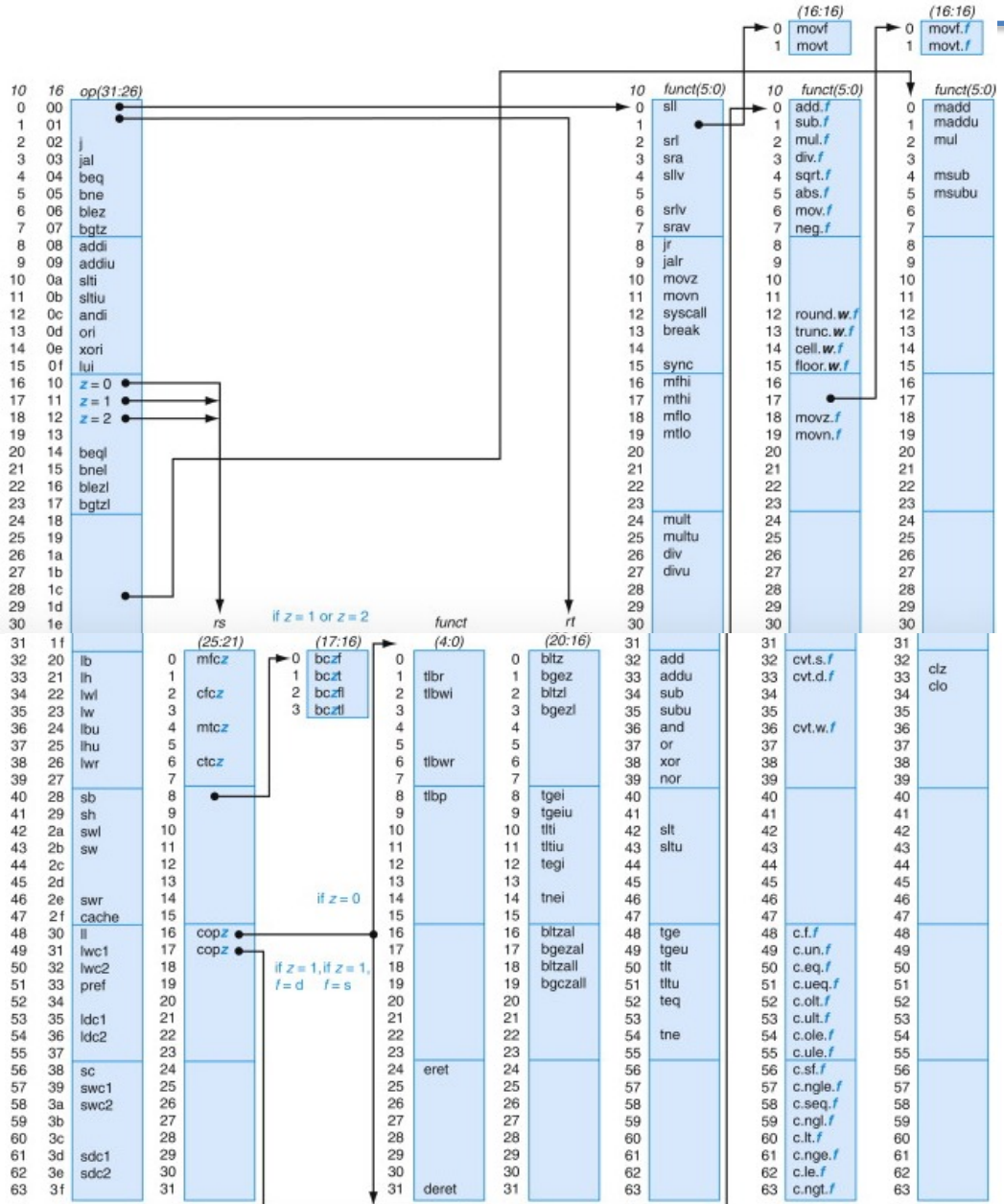
Figure 7.10.2: MIPs opcode map (COD Figure A.10.2).

The values of each field are shown to its left . The first column shows the values in base 10, and the second shows base 16 for the op field (bits 31 to 26) in the third column. This op field completely specifies the MIPS operation except for six op values: 0, 1, 16, 17, 18, and 19. These operations are determined by other fields, identified by pointers. The last field (funct) uses "*f*" to mean "s" if rs = 16 and op = 17 or "d" if rs = 17 and op = 17. The second field (rs) uses "z" to mean "0", "1", "2", or "3" if op = 16, 17, 18, or 19, respectively. If rs = 16, the operation is specified elsewhere: if *z* = 0, the operations are specified in the fourth field (bits 4 to 0); if *z* = 1, then the operations are in the last field with *f* = s. If rs = 17 and *z* = 1, then the operations are in the last field with *f* = d.



Opcode Map

Hennessy & Patterson



MIPS II-III ISA

CPU

Wikipedia

MIPS II [edit]

MIPS II removed the load delay slot[4]:41 and added several sets of instructions. For shared-memory multiprocessing, the *Synchronize Shared Memory*, *Load Linked Word*, and *Store Conditional Word* instructions were added. A set of Trap-on-Condition instructions were added. These instructions caused an exception if the evaluated condition is true. All existing branch instructions were given *branch-likely* versions that executed the instruction in the branch delay slot only if the branch is taken.[4]:40 These instructions improve performance in certain cases by allowing useful instructions to fill the branch delay slot.[4]:212 Doubleword load and store instructions for COP1–3 were added. Consistent with other memory access instructions, these loads and stores required the doubleword to be naturally aligned.

The instruction set for the floating point coprocessor also had several instructions added to it. An IEEE 754-compliant floating-point square root instruction was added. It supported both single- and double-precision operands. A set of instructions that converted single- and double-precision floating-point numbers to 32-bit words were added. These complemented the existing conversion instructions by allowing the IEEE rounding mode to be specified by the instruction instead of the Floating Point Control and Status Register.

MIPS Computer Systems' R6000 microprocessor (1989) was the first MIPS II implementation.[4]:8 Designed for servers, the R6000 was fabricated and sold by Bipolar Integrated Technology, but was a commercial failure. During the mid-1990s, many new 32-bit MIPS processors for embedded systems were MIPS II implementations because the introduction of the 64-bit MIPS III architecture in 1991 left MIPS II as the newest 32-bit MIPS architecture until MIPS32 was introduced in 1999.A[4]:19

MIPS III [edit]

MIPS III is a backwards-compatible extension of MIPS II that added support for 64-bit memory addressing and integer operations. The 64-bit data type is called a doubleword, and MIPS III extended the general-purpose registers, HI/LO registers, and program counter to 64 bits to support it. New instructions were added to load and store doublewords, to perform integer addition, subtraction, multiplication, division, and shift operations on them, and to move doubleword between the GPRs and HI/LO registers. Existing instructions originally defined to operate on 32-bit words were redefined, where necessary, to sign-extend the 32-bit results to permit words and doublewords to be treated identically by most instructions. Among those instructions redefined was *Load Word*. In MIPS III it sign-extends words to 64 bits. To complement *Load Word*, a version that zero-extends was added.

The R instruction format's inability to specify the full shift distance for 64-bit shifts (its 5-bit shift amount field is too narrow to specify the shift distance for doublewords) required MIPS III to provide three 64-bit versions of each MIPS I shift instruction. The first version is a 64-bit version of the original shift instructions, used to specify constant shift distances of 0–31 bits. The second version is similar to the first, but adds 32_{10} the shift amount field's value so that constant shift distances of 32–64 bits can be specified. The third version obtains the shift distance from the six low-order bits of a GPR.

MIPS III added a *supervisor* privilege level in between the existing kernel and user privilege levels. This feature only affected the implementation-defined System Control Processor (Coprocessor 0).

MIPS III ISA

CPU

Wikipedia

CPU instructions added by MIPS III

Instruction name	Mnemonic	Format	Encoding					
Doubleword Shift Left Logical Variable	DSLLV	R	0_{10}	rs	rt	rd	0_{10}	20_{10}
Doubleword Shift Right Logical Variable	DSRLV	R	0_{10}	rs	rt	rd	0_{10}	22_{10}
Doubleword Shift Right Arithmetic Variable	DSRAV	R	0_{10}	rs	rt	rd	0_{10}	23_{10}
Doubleword Multiply	DMULT	R	0_{10}	rs	rt	0_{10}	0_{10}	28_{10}
Doubleword Multiply Unsigned	DMULTU	R	0_{10}	rs	rt	0_{10}	0_{10}	29_{10}
Doubleword Divide	DDIV	R	0_{10}	rs	rt	0_{10}	0_{10}	30_{10}
Doubleword Divide Unsigned	DDIVU	R	0_{10}	rs	rt	0_{10}	0_{10}	31_{10}
Doubleword Add	DADD	R	0_{10}	rs	rt	rd	0_{10}	44_{10}
Doubleword Add Unsigned	DADDU	R	0_{10}	rs	rt	rd	0_{10}	45_{10}
Doubleword Subtract	DSUB	R	0_{10}	rs	rt	rd	0_{10}	46_{10}
Doubleword Subtract Unsigned	DSUBU	R	0_{10}	rs	rt	rd	0_{10}	47_{10}
Doubleword Shift Left Logical	DSLL	R	0_{10}	0_{10}	rt	rd	sa	56_{10}
Doubleword Shift Right Logical	DSRL	R	0_{10}	0_{10}	rt	rd	sa	58_{10}
Doubleword Shift Right Arithmetic	DSRA	R	0_{10}	0_{10}	rt	rd	sa	59_{10}
Doubleword Shift Left Logical + 32	DSLL32	R	0_{10}	0_{10}	rt	rd	sa	60_{10}
Doubleword Shift Right Logical + 32	DSRL32	R	0_{10}	0_{10}	rt	rd	sa	62_{10}
Doubleword Shift Right Logical + 32	DSRL32	R	0_{10}	0_{10}	rt	rd	sa	63_{10}
Doubleword Add Immediate	DADDI	I	24_{10}	rs	rd	immediate		
Doubleword Add Immediate Unsigned	DADDIU	I	25_{10}	rs	rd	immediate		
Load Doubleword Left	LDL	I	26_{10}	rs	rt	offset		
Load Doubleword Right	LDR	I	27_{10}	rs	rt	offset		
Load Word Unsigned	LWU	I	39_{10}	rs	rt	offset		
Store Doubleword Left	SDL	I	44_{10}	rs	rt	offset		

MIPS32 ISA

Trap instructions [edit]

Mnemonic	Description
BREAK	Breakpoint
SYSCALL	System Call
TEQ	Trap if Equal
TEQI	Trap if Equal Immediate
TGE	Trap if Greater or Equal
TGEI	Trap if Greater of Equal Immediate
TGEIU	Trap if Greater or Equal Immediate Unsigned
TGEU	Trap if Greater or Equal Unsigned
TLT	Trap if Less Than
TLTI	Trap if Less Than Immediate
TLTIU	Trap if Less Than Immediate Unsigned
TLTU	Trap if Less Than Unsigned
TNE	Trap if Not Equal
TNEI	Trap if Not Equal Immediate

❖ Traps

MIPS32 ISA

MIPS32/MIPS64 [edit]

When MIPS Technologies was spun-out of Silicon Graphics in 1998, it refocused on the embedded market. Up to MIPS V, each successive version was a strict superset of the previous version, but this property was found to be a problem,[citation needed] and the architecture definition was changed to define a 32-bit and a 64-bit architecture: MIPS32 and MIPS64. Both were introduced in 1999.[19] MIPS32 is based on MIPS II with some additional features from MIPS III, MIPS IV, and MIPS V; MIPS64 is based on MIPS V.[19] NEC, Toshiba and SiByte (later acquired by Broadcom) each obtained licenses for MIPS64 as soon as it was announced. Philips, LSI Logic, IDT, Raza Microelectronics, Inc., Cavium, Loongson Technology and Ingenic Semiconductor have since joined them.

MIPS32/MIPS64 Release 1 [edit]

The first release of MIPS32, based on MIPS II, added conditional moves, prefetch instructions, and other features from the R4000 and R5000 families of 64-bit processors.[19] The first release of MIPS64 adds a MIPS32 mode to run 32-bit code.[19] The MUL and MADD (multiply-add) instructions, previously available in some implementations, were added to the MIPS32 and MIPS64 specifications, as were cache control instructions.[19]

MIPS32/MIPS64 Release 3 [edit]

MIPS32/MIPS64 Release 5 [edit]

Announced on December 6, 2012.[20] Release 4 was skipped because the number four is perceived as unlucky in many Asian cultures.[21]

MIPS32/MIPS64 Release 6 [edit]

MIPS32/MIPS64 Release 6 in 2014 added[22] the following:

- a new family of branches with no delay slot:

MIPS32 ISA

Privileged instructions [edit]

Mnemonic	Description
CACHE	Perform Cache Operation
CACHEE	Perform Cache Operation EVA
DI	Disable Interrupts
EI	Enable Interrupts
ERET	Exception Return
MFC0	Move from Coprocessor 0
MTC0	Move to Coprocessor 0
RDPGPR	Read GPR from Previous Shadow Set
TLBP	Probe TLB for Matching Entry
TLBR	Read Indexed TLB Entry
TLBWI	Write Indexed TLB Entry
TLBWR	Write Random TLB Entry
WAIT	Enter Standby Mode
WRPGPR	Write GPR to Previous Shadow Set

EJTAG instructions [edit]

Mnemonic	Description
DERET	Debug Exception Return
SDBBP	Software Debug Breakpoint

MIPS32 ISA

Coprocessor 2

Hennessy & Patterson

Execute instructions [edit]

Mnemonic	Description
COP2	Coprocessor Operation to Coprocessor 2

Memory control instructions [edit]

Mnemonic	Description
DC2	Load Doubleword to Coprocessor 2
LWC2	Load Word to Coprocessor 2
SDC2	Store Doubleword from Coprocessor 2
SWC2	Store Word from Coprocessor 2

Move instructions [edit]

Mnemonic	Description
CFC2	Move Control Word from Coprocessor 2
CTC2	Move Control Word to Coprocessor 2
MFC2	Move Word from Coprocessor 2
MFHC2	Move Word from High Half of Coprocessor 2 Register
MTC2	Move Word to Coprocessor 2
MTHC2	Move Word to High Half of Coprocessor 2 Register

Branch instructions [edit]

Mnemonic	Description
BC2F	Branch on COP2 False
BC2T	Branch on COP2 True
~~BC2FL~~	Branch on COP2 False Likely
~~BC2TL~~	Branch on COP2 True Likely

ISA

Comparative

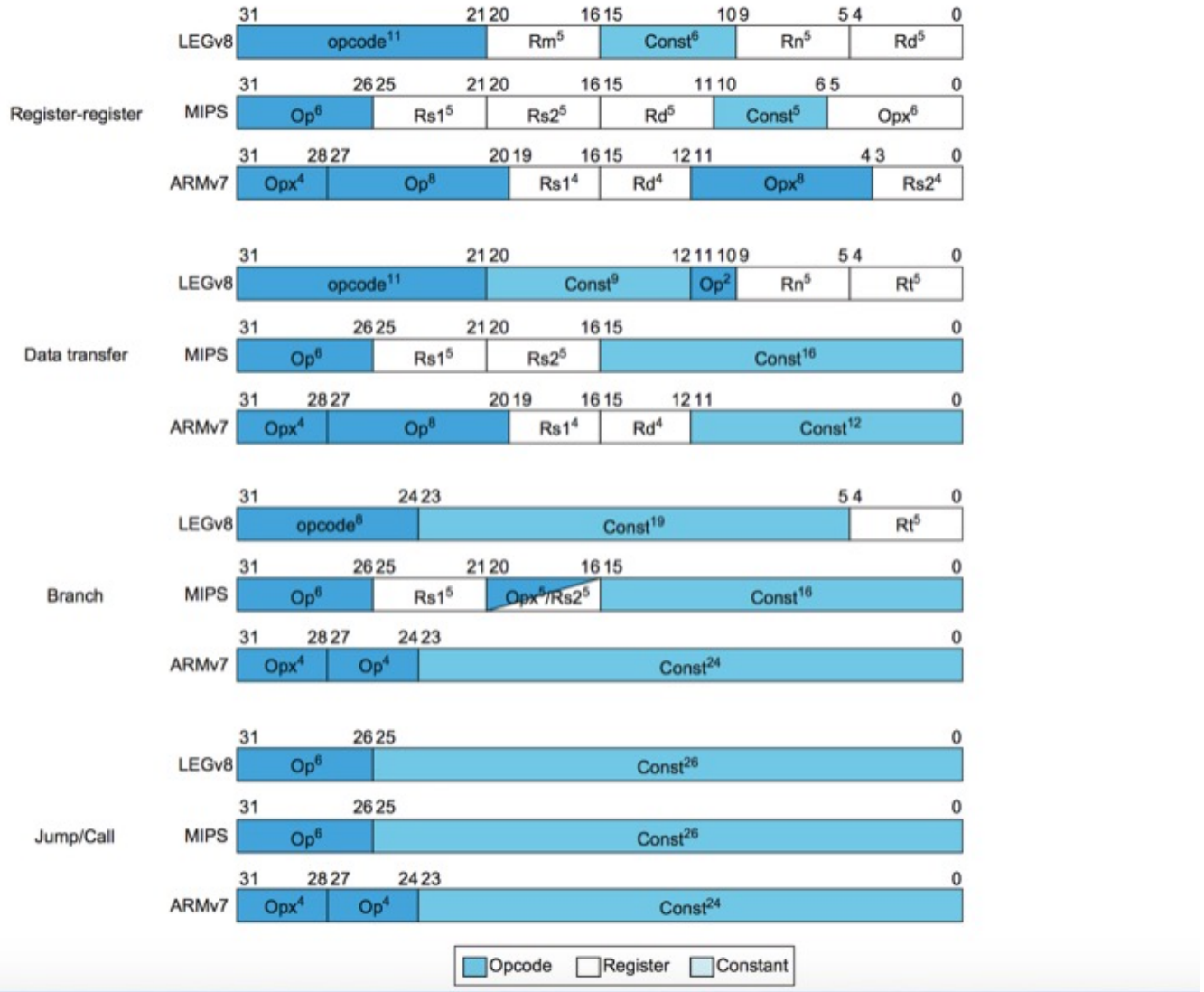
Compare ISA's

- Desktop CPU's (Large)
  - Alpha
  - MIPS
  - PowerPC
  - PA-RISC
  - SPARC
- Embedded CPU's (small)
  - ARM
  - Thumb
  - SuperH
  - M32R
  - MIPS-16

MIPS vs. ARM Instructions

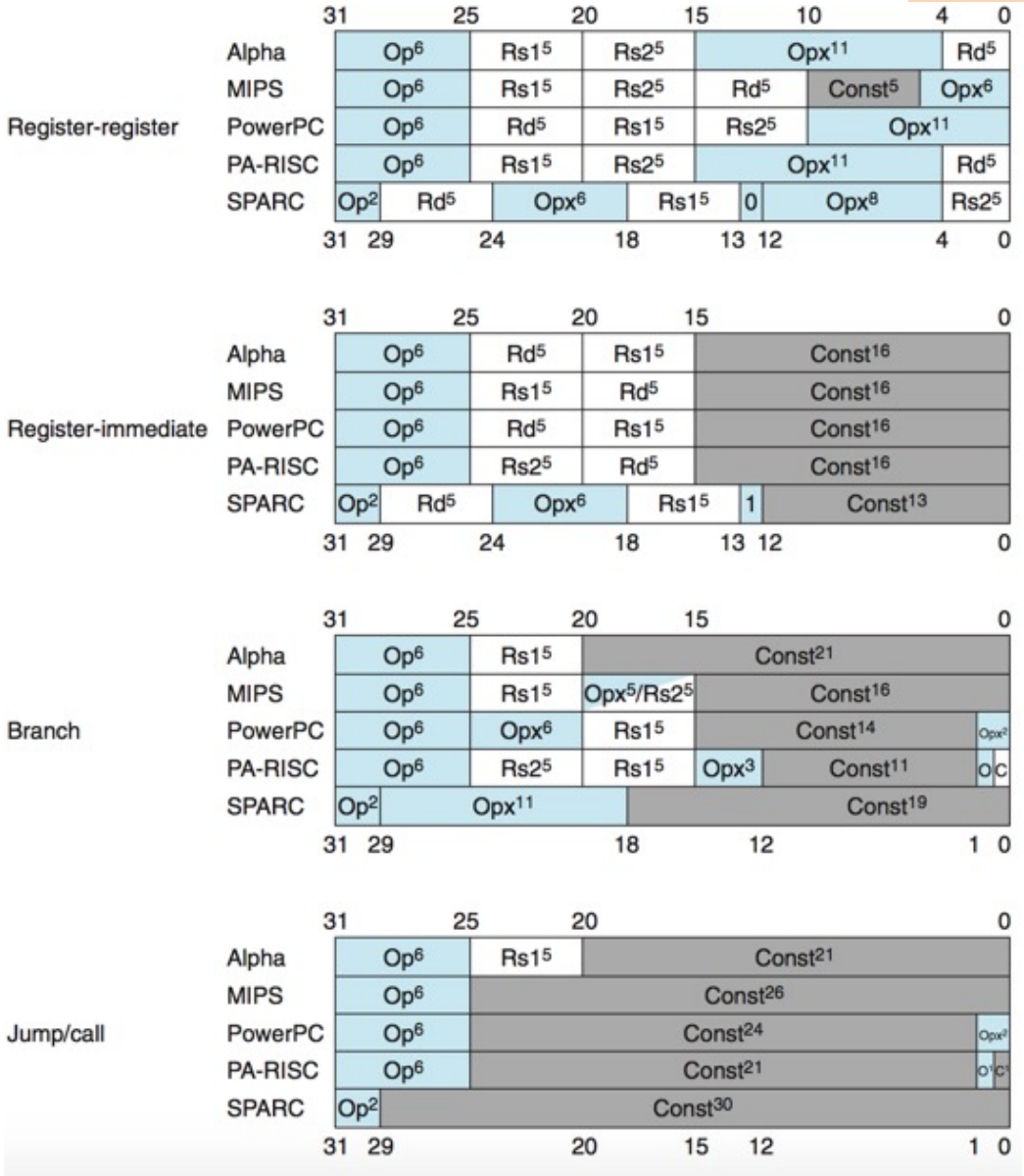
Hennessy & Patterson

esult in part from whether the architecture has 16 registers (ARMv7) or 32 registers (ARMv8 and MIPS).



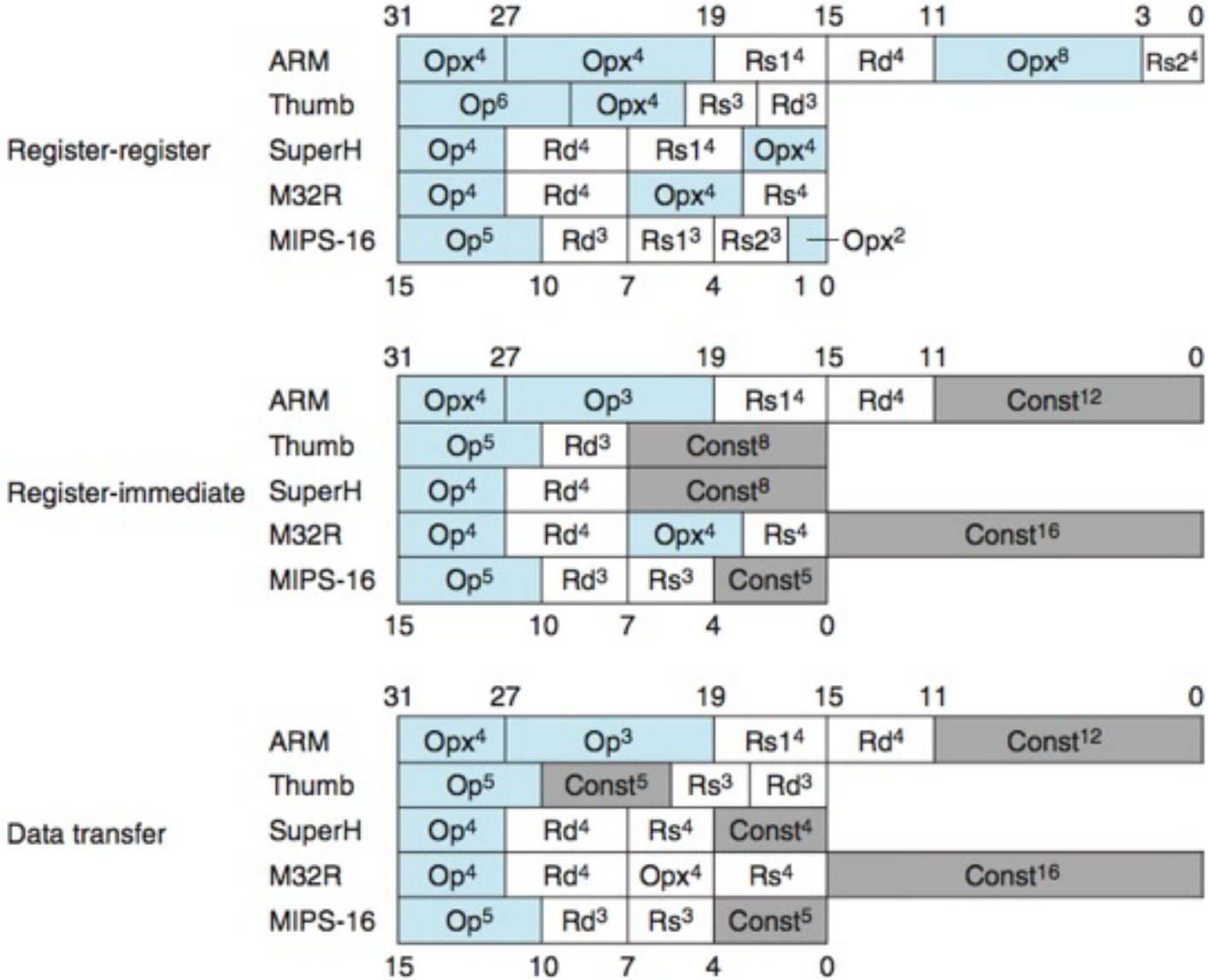
Desktop Instructions

Hennessy & Patterson



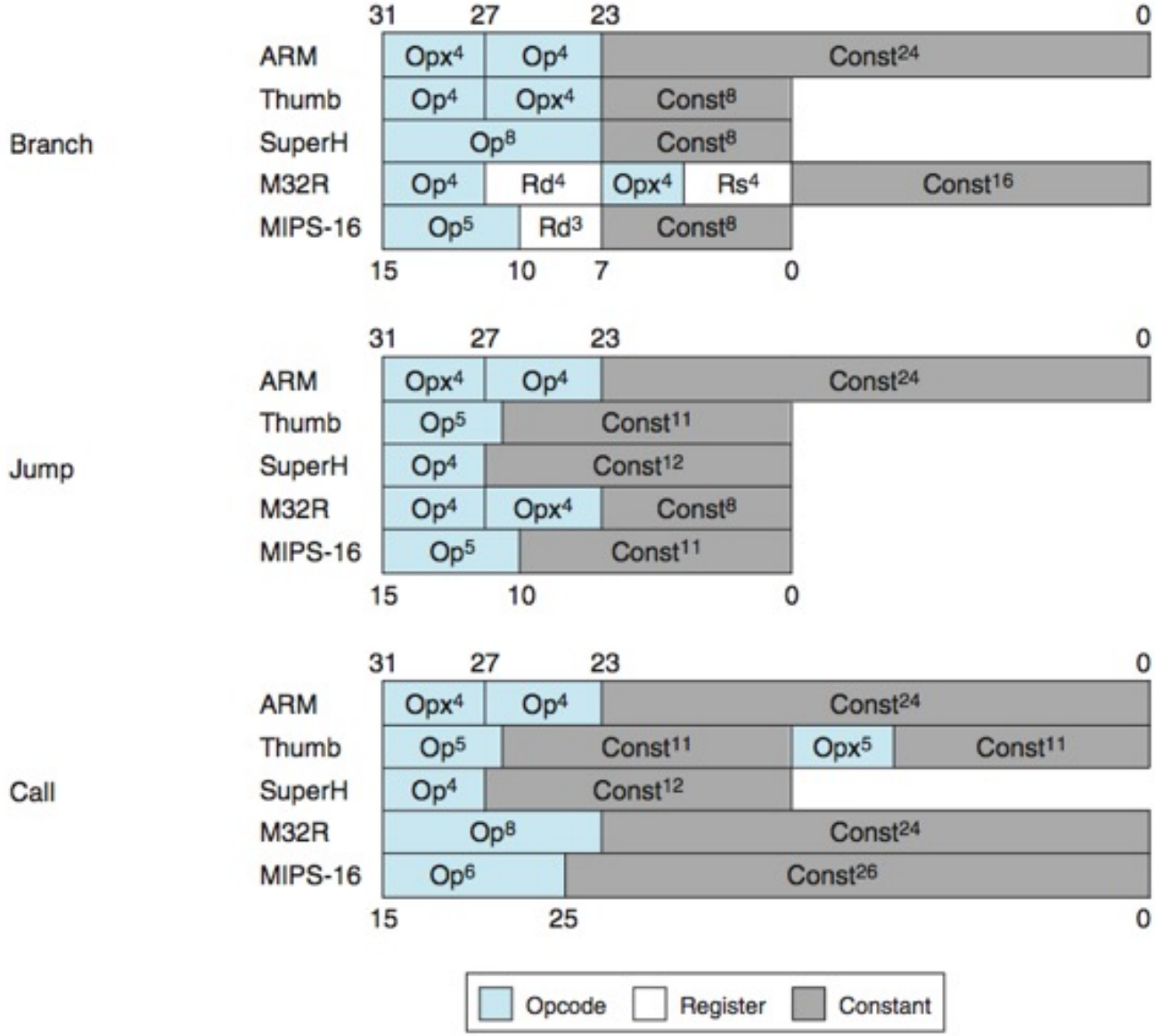
Embedded Instructions

Hennessy & Patterson



Embedded Instructions

Hennessy & Patterson



Addressing Modes

Hennessy & Patterson

Desktop

Addressing mode	Alpha	MIPS-64	PA-RISC 2.0	PowerPC	SPARCv9
Register + offset (displacement or based)	X	X	X	X	X
Register + register (indexed)		X (FP)	X (Loads)	X	X
Register + scaled register (scaled)			X		
Register + offset and update register			X	X	
Register + register and update register			X	X	

Embedded

Addressing mode	ARMv4	Thumb	SuperH	M32R	MIPS-16
Register + offset (displacement or based)	X	X	X	X	X
Register + register (indexed)	X	X	X		
Register + scaled register (scaled)	X				
Register + offset and update register	X				
Register + register and update register	X				
Register indirect			X	X	
Autoincrement, autodecrement	X	X	X	X	
PC-relative data	X	X (loads)	X		X (loads)

Address Sign Extension

Hennessy & Patterson

Desktop

Format: instruction category	Alpha	MIPS-64	PA-RISC 2.0	PowerPC	SPARCv9
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	Zero	Sign	Sign	Sign	Sign
Register-immediate: logical	Zero	Zero	—	Zero	Sign

Embedded

Format: instruction category	Armv4	Thumb	SuperH	M32R	MIPS-16
Branch: all	Sign	Sign	Sign	Sign	Sign
Jump/call: all	Sign	Sign/Zero	Sign	Sign	—
Register-immediate: data transfer	Zero	Zero	Zero	Sign	Zero
Register-immediate: arithmetic	Zero	Zero	Sign	Sign	Zero/Sign
Register-immediate: logical	Zero	—	Zero	Zero	—

MIPS vs ARM

Hennessy & Patterson

10.12 Instructions unique to ARM

Present Note

(Original section[1])

It's hard to pick the most unusual feature of ARM, but perhaps it is the conditional execution of instructions. Every instruction starts with a 4-bit field that determines whether it will act as a nop or as a real instruction, depending on the condition codes. Hence, conditional branches are properly considered as conditionally executing the unconditional branch instruction. Conditional execution allows avoiding a branch to jump over a single instruction. It takes less code space and time to simply conditionally execute one instruction.

The 12-bit immediate field has a novel interpretation. The 8 least significant bits are zero-extended to a 32-bit value, then rotated right the number of bits specified in the first 4 bits of the field multiplied by two. Whether this split actually catches more immediates than a simple 12-bit field would be an interesting study. One advantage is that this scheme can represent all powers of two in a 32-bit word.

Operand shifting is not limited to immediates. The second register of all arithmetic and logical processing operations has the option of being shifted before being operated on. The shift options are shift left logical, shift right logical, shift right arithmetic, and rotate right. Once again, it would be interesting to see how often operations like rotate-and-add, shift -right-and-test, and so on occur in ARM programs.

MIPS vs ARM

Hennessy & Patterson

Remaining instructions

Below is a list of the remaining unique instructions of the ARM architecture:

- *Block loads and stores*—Under control of a 16-bit mask within the instructions, any of the 16 registers can be loaded or stored into memory in a single instruction. These instructions can save and restore registers on procedure entry and return. These instructions can also be used for block memory copy—offering up to four times the bandwidth of a single register load-store—and today, block copies are the most important use.
- *Reverse subtract*—RSB allows the first register to be subtracted from the immediate or shifted register. RSC does the same thing, but includes the carry when calculating the difference.
- *Long multiplies*—Similarly to MIPS, Hi and Lo registers get the 64-bit signed product (SMULL) or the 64-bit unsigned product (UMULL).
- *No divide*—Like the Alpha, integer divide is not supported in hardware.
- *Conditional trap*—A common extension to the MIPS core found in desktop RISCs (COD Figure D.6.1 (Data transfer instructions not found in MIPS core ...), COD Figure D.6.2 (Arithmetic/logical instructions not found in MIPS core ...), COD Figure D.6.3 (Control instructions not found in MIPS core ...), COD Figure D.6.4 (Floating-point instructions not found in MIPS core ...)), it comes for free in the conditional execution of all ARM instructions, including SWI.
- *Coprocessor interface*—Like many of the desktop RISCs, ARM defines a full set of coprocessor instructions: data transfer, moves between general-purpose and coprocessor registers, and coprocessor operations.
- *Floating-point architecture*—Using the coprocessor interface, a floating-point architecture has been defined for ARM. It was implemented as the FPA10 coprocessor.
- *Branch and exchange instruction sets*—The BX instruction is the transition between ARM and Thumb, using the lower 31 bits of the register to set the PC and the most significant bit to determine if the mode is ARM (1) or Thumb (0).

MIPS vs ARM

Hennessy & Patterson

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

MIPS vs ARM

Hennessy & Patterson

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]

9: Conventions of embedded RISC instructions equivalent to MIPS core (COD Figure

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