#4 : MIPS Programming

Computer Architecture 2021/2022

Ricardo Rocha

Computer Science Department, Faculty of Sciences, University of Porto

Slides based on the book

'Computer Organization and Design, The Hardware/Software Interface, 5th Edition

David Patterson and John Hennessy, Morgan Kaufmann'

Sections 2.5 – 2.8, 2.10 and A.5 – A.6

Instruction Set

The words of a computer's language are called **instructions** and the vocabulary of commands understood by a given architecture is called an **instruction set**. Common groups of instructions are:

- Arithmetic instructions
- Logical instructions
- Data transfer instructions
- Conditional branch instructions
- Unconditional jump instructions

Different computers have different instruction sets but with many aspects in common. Early computers had very simple instruction sets but many modern computers also have simple instruction sets.

Complex Instruction Set Computer (CISC)

A CISC is a computer in which **single instructions can execute several low-level operations** (such as a load from memory, an arithmetic operation, and a memory store) or are capable of multi-step operations or addressing modes within single instructions.

The term was retroactively coined in contrast to RISC and has therefore become an umbrella term for **everything that is not RISC**. The typical differentiating characteristics is that most **RISC designs use uniform instruction length for almost all instructions, and employ strictly separate load/store-instructions**.

Complex Instruction Set Computer (CISC)

Examples of architectures that have been retroactively labeled CISC:

- Intel 8080, iAPX432 and **x86-family** (most personal computers)
- MOS Technology 6502-family
- Motorola 6800, 6809 and 68000-families
- National Semiconductor 32016 and NS320xx-line
- Zilog Z80, Z8 and Z8000-families

Reduced Instruction Set Computer (RISC)

Various suggestions have been made regarding a precise definition of RISC, but the general concept is that such a computer has a **small set of simple and general instructions**, rather than a large set of complex and specialized instructions. Another common RISC characteristic is its **load/store architecture** in which memory is accessed through specific instructions rather than as a part of most instructions.

Reduced Instruction Set Computer (RISC)

The term RISC was coined by **David Patterson** in the Berkeley RISC project, although somewhat similar concepts had appeared before. Another project associated with the popularization of the term RISC is the MIPS project that grew out of a graduate course by **John Hennessy** at Stanford University in 1981, which resulted in a functioning system in 1983, and could run simple programs by 1984.

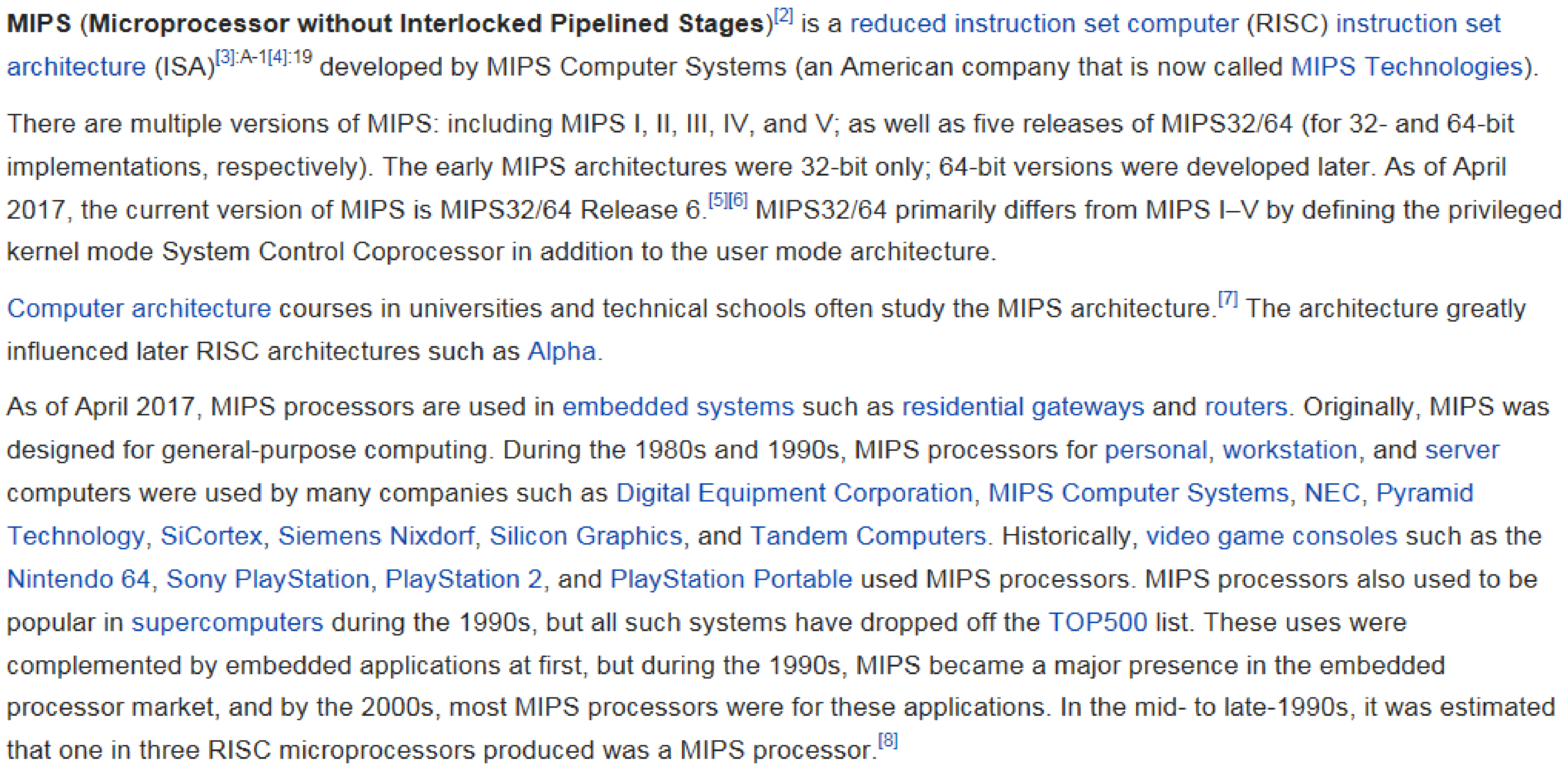
Virtually all new instruction sets since 1982 have followed the RISC philosophy of fixed instruction lengths, load-store instruction sets, limited addressing modes, and limited operations.

Reduced Instruction Set Computer (RISC)

Examples of RISC architectures:

- **ARM** (e.g., smartphones, tablets, laptops and embedded systems)
- Alpha
- Hitachi SH
- IBM PowerPC
- Intel i860, i960
- **MIPS** (e.g., gateways, routers and video game consoles)
- Sun SPARC

MIPS (from Wikipedia)



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 (an American company that is now called MIPS Technologies).

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 processors were for these applications. In the mid- to late-1990s, it was estimated that one in three RISC microprocessors produced was a MIPS processor.[8]

Registers

MIPS has **32 registers**, numbered from 0 to 31, each with 32 bits (64-bit versions also exist). To identify a register in MIPS we need **5 bits (2^5=32)**.

Name	Register number	Usage
`$zero`	0	The constant value 0
`$v0–$v1`	2–3	Values for results and expression evaluation
`$a0–$a3`	4–7	Arguments
`$t0–$t7`	8–15	Temporaries
`$s0–$s7`	16–23	Saved
`$t8–$t9`	24–25	More temporaries
`$gp`	28	Global pointer
`$sp`	29	Stack pointer
`$fp`	30	Frame pointer
`$ra`	31	Return address

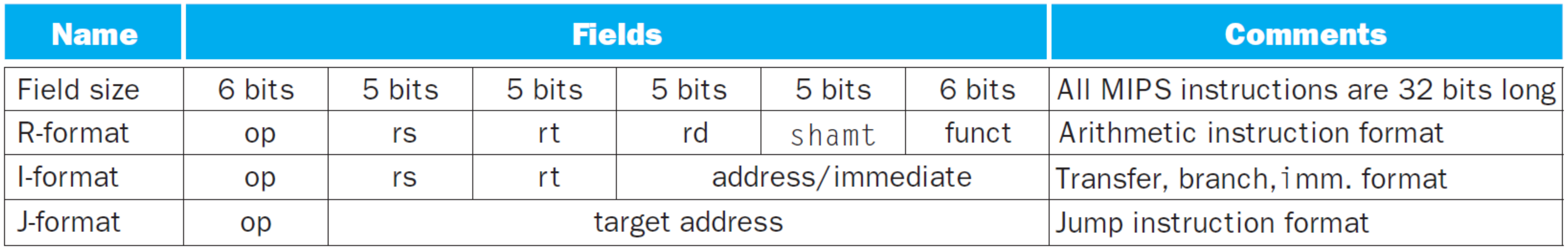
Instruction Format

MIPS instructions are encoded in binary, as **32-bit instruction words**, called **machine code**. The layout of an instruction is called the **instruction format**. Only 3 different formats exist.

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

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)

Instruction Fields



Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

MIPS fields have names to make them easier to discuss:

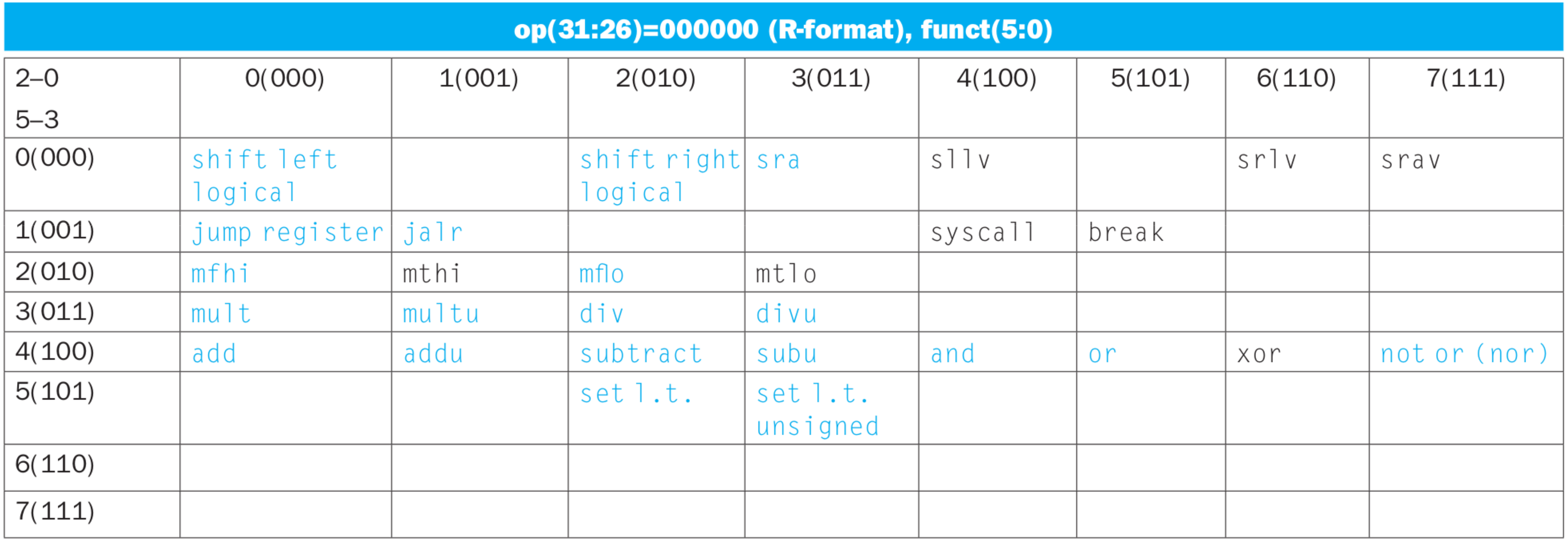
- **op** – basic operation of the instruction, traditionally called the opcode
- **rs** – the first register source operand
- **rt** – the second register source operand
- **rd** – the register destination operand, which gets the result of the operation
- **shamt** – shift amount to be used in shift instructions, zero otherwise
- **funct** – often called the function code, selects the specific variant of the operation in the opcode field

Instruction Opcodes

op(31:26)

28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	load linked word	lwc1						
7(111)	store cond. word	swc1						

R-format Function Codes



op(31:26)=000000 (R-format), funct(5:0)

2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump register	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	set l.t. unsigned				
6(110)								
7(111)								

From Assembly to Machine Code

Let’s see an example of a R-format instruction, first as a combination of decimal numbers and then of binary numbers. Consider the instruction:

add $t0, $s1, $s2

The **op** and **funct** fields in combination (0 and 32 in this case) tell that this instruction performs addition (add).

The **rs** and **rt** fields, registers $s1 (17) and $s2 (18), are the source operands, and the **rd** field, register $t0 (8), is the destination operand.

The **shamt** field is unused in this instruction, so it is set to 0.

From Assembly to Machine Code

Thus, the decimal representation of instruction **add $t0, $s1, $s2** is:

- **op** = 0 (arithmetic)
- **rs** = 17 ($s1)
- **rt** = 18 ($s2)
- **rd** = 8 ($t0)
- **shamt** = 0 (not used)
- **funct** = 32 (add)

0	17	18	8	0	32

And the binary representation is:

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

Arithmetic and Logical Instructions

add $s1, $s2, $s3	$s1 = $s2 + $s3	(add)
addu $s1, $s2, $s3	$s1 = $s2 + $s3	(add unsigned, no overflow)
addi $s1, $s2, 20	$s1 = $s2 + 20	(add immediate)
addiu $s1, $s2, 20	$s1 = $s2 + 20	(add immediate, no overflow)
sub $s1, $s2, $s3	$s1 = $s2 – $s3	(subtract)
and $s1, $s2, $s3	$s1 = $s2 & $s3	(and, bit-by-bit)
andi $s1, $s2, 20	$s1 = $s2 & 20	(and immediate)
or $s1, $s2, $s3	$s1 = $s2 | $s3	(or)
nor $s1, $s2, $s3	$s1 = ~ ($s2 | $s3)	(nor)
sll $s1, $s2, 10	$s1 = $s2 << 10	(shift left logical)
srl $s1, $s2, 10	$s1 = $s2 >> 10	(shift right logical)

Data Transfer Instructions

lw $s1, 20($s2)	$s1 = Mem[$s2 + 20]	(load word, from memory)
lh $s1, 20($s2)	$s1 = Mem[$s2 + 20]	(load half word, sign-extend)
lhu $s1, 20($s2)	$s1 = Mem[$s2 + 20]	(load half word, zero-extend)
lb $s1, 20($s2)	$s1 = Mem[$s2 + 20]	(load byte, sign-extend)
lbu $s1, 20($s2)	$s1 = Mem[$s2 + 20]	(load byte, zero-extend)
li $s1, 20	$s1 = 20	(load immediate)
la $s1, L	$s1 = L	(load address)
move $s1, $s2	$s1 = $s2	(data move)
sw $s1, 20($s2)	Mem[$s2 + 20] = $s1	(store word, to memory)
sh $s1, 20($s2)	Mem[$s2 + 20] = $s1	(store half word)
sb $s1, 20($s2)	Mem[$s2 + 20] = $s1	(store byte)

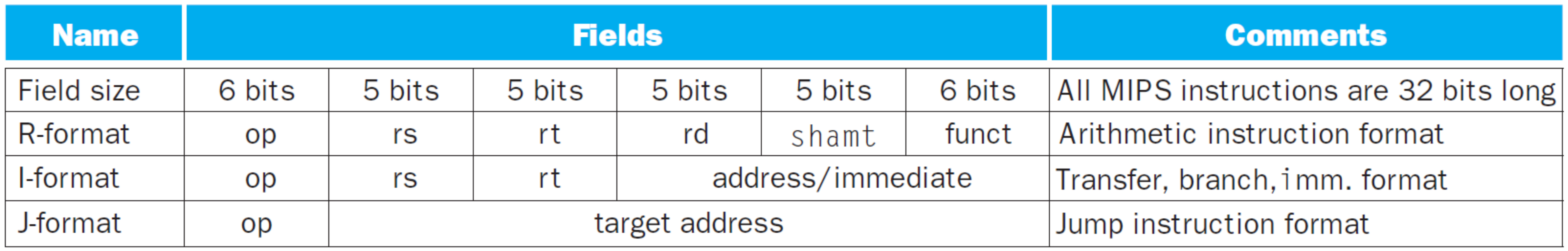
Branch Instructions

beq \$s1, \$s2, 25	if (\$s1 == \$s2) go to (PC+4+100)	(branch on equal)
beq \$s1, \$s2, L	if (\$s1 == \$s2) go to L	(branch on equal)
bne \$s1, \$s2, L	if (\$s1 != \$s2) go to L	(branch on not equal)
blt \$s1, \$s2, L	if (\$s1 < \$s2) go to L	(branch on less than)
bgt \$s1, \$s2, L	if (\$s1 > \$s2) go to L	(branch on greater than)
ble \$s1, \$s2, L	if (\$s1 <= \$s2) go to L	(branch on less than or equal)
slt \$s1, \$s2, \$s3	if (\$s2 < \$s3) \$s1 = 1 else \$s1 = 0	(set on less than, for use with beq/bne)
slti \$s1, \$s2, 20	if (\$s2 < 20) \$s1 = 1 else \$s1 = 0	(set on less than immediate)

Jump Instructions

j 2500	go to 10000	(jump to target address)
j L	go to L	(jump to target address)
jal L	\$ra = PC+4; go to L	(jump and link, for procedure call)
jr \$ra	go to \$ra	(jump register, for procedure return)

From Assembly to Machine Code



Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

Remember that, in branch (I-format) and jump (J-format) instructions, the **address** and **target address** fields need to be shifted left 2 bits to correctly represent a valid instruction address (32-bits aligned).

beq \$s1, \$s2, 25	**(25<<2 = 100)**
j 2500	**(2500<<2 = 10000)**

Pseudo-Instructions

Most assembler instructions represent machine instructions one-to-one. The assembler can also treat common variations of machine instructions as if they were instructions in their own right. Such instructions are called **pseudo-instructions**.

The hardware need not implement the pseudo-instructions, but their appearance in assembly language **simplifies programming**. Register $at (assembler temporary) is reserved for this purpose.

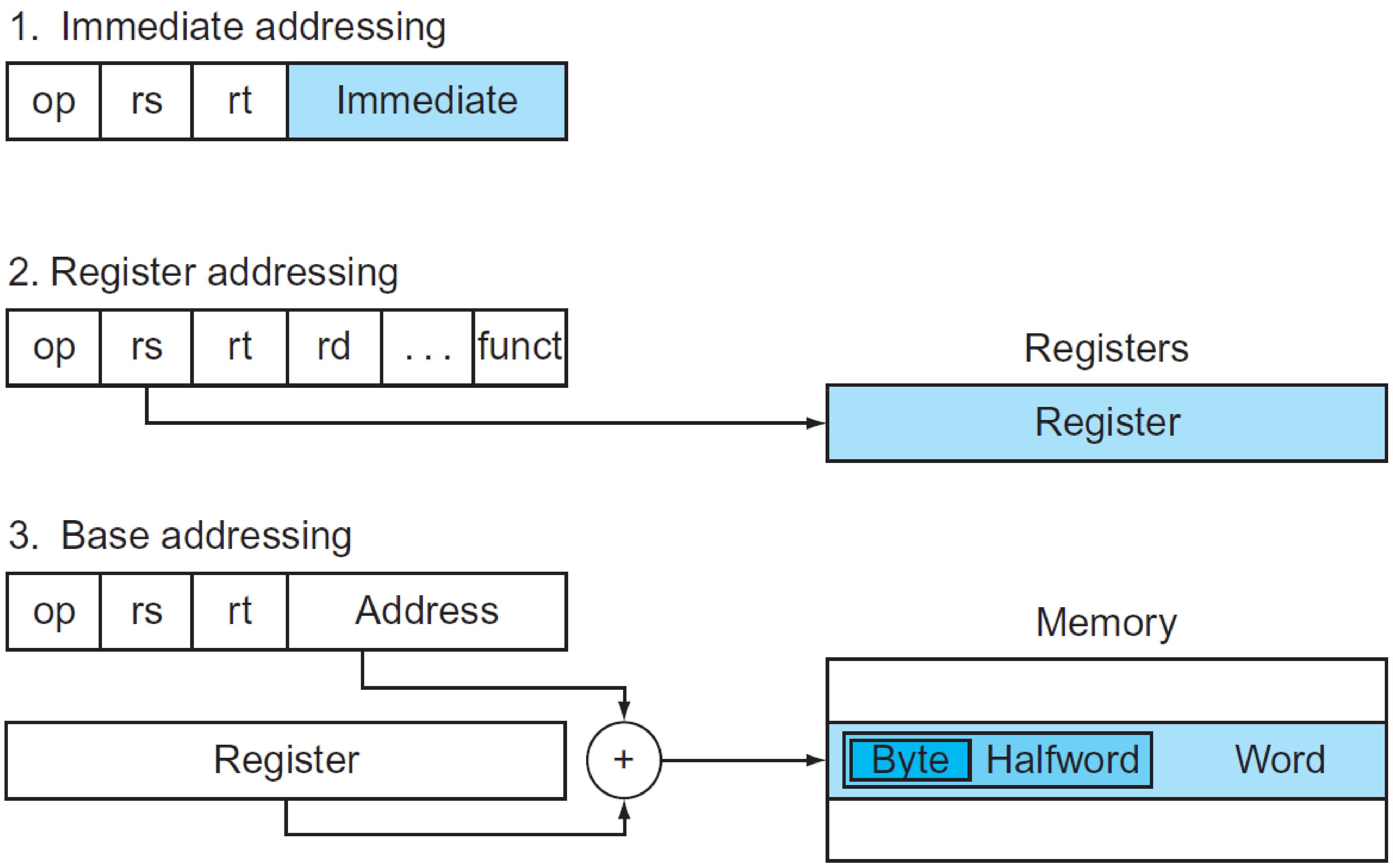
```
li $s1, 20        →   addiu $s1, $zero, 20
move $t0, $t1     →   addu $t0, $zero, $t1
blt $s1, $s2, L   →   slt $at, $s1, $s2
                      bne $at, $zero, L
```

Addressing Modes

MIPS addressing modes are:

- **Immediate addressing** where the operand is a constant in the instruction itself
- **Register addressing** where the operand is a register
- **Base or displacement addressing** where the operand is at the memory location whose address is the sum of a register and a constant in the instruction
- **PC-relative addressing** where the branch address is the sum of the PC with a constant in the instruction
- **Pseudo-direct addressing** where the jump address is a constant in the instruction concatenated with the upper bits of the PC

Addressing Modes



Addressing Modes

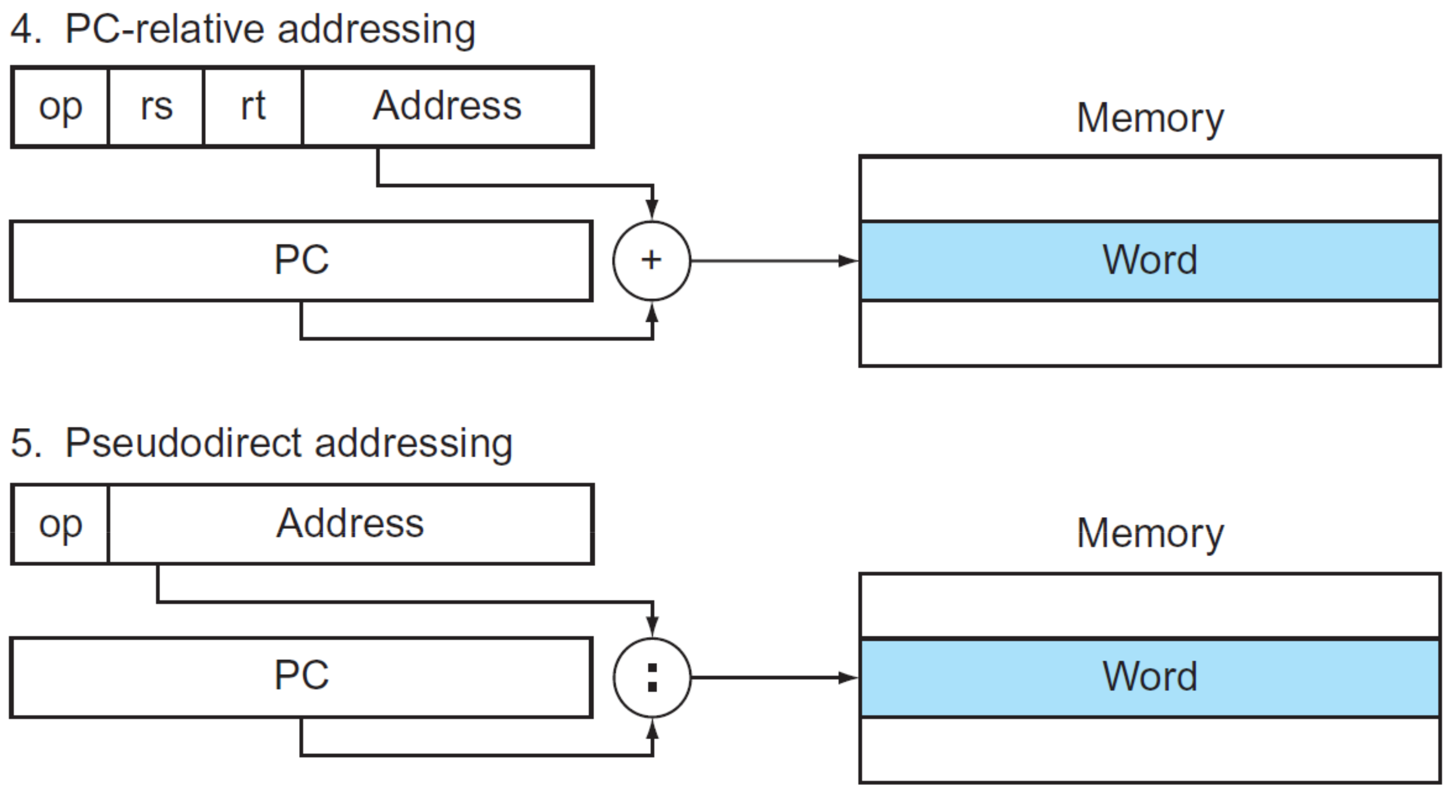


FIGURE 2.18 Illustration of the five MIPS addressing modes. The operands are shaded in color. The operand of mode 3 is in memory, whereas the operand for mode 2 is a register. Note that versions of load and store access bytes, halfwords, or words. For mode 1, the operand is 16 bits of the instruction itself. Modes 4 and 5 address instructions in memory, with mode 4 adding a 16-bit address shifted left 2 bits to the PC and mode 5 concatenating a 26-bit address shifted left 2 bits with the 4 upper bits of the PC. Note that a single operation can use more than one addressing mode. Add, for example, uses both immediate (`addi`) and register (`add`) addressing.

Simple Arithmetic Expression

```
r = (x + y) - (z + w);   // $s0 = ($s1 + $s2) - ($s3 + $s4)

----------------------------------------------------------

add $t0, $s1, $s2        # $t0 = $s1 + $s2
add $t1, $s3, $s4        # $t1 = $s3 + $s4
sub $s0, $t0, $t1        # $s0 = $t0 - $t1
```

Simple Array Expression

```
r[12] = x + r[8];   // $s0[12] = $s1 + $s0[8]   (r[] array of integers)

-------------------------------------------------------------------------

lw  $t0, 32($s0)    # $t0 = $s0[32/4] (Mem[$s0 + 32])
add $t0, $s1, $t0   # $t0 = $s1 + $t0
sw  $t0, 48($s0)    # $s0[48/4] = $t0 (Mem[$s0 + 48])
```

Simple Conditional Statement

```
if (x == y) r = z + w;        // if ($s1 == $s2) $s0 = $s3 + $s4
else        r = z - w;        // else            $s0 = $s3 - $s4

------------------------------------------------------------------------

       bne $s1, $s2, _else   # $s1 != $s2 → _else
       add $s0, $s3, $s4     # $s0 = $s3 + $s4
       j   _end              # → _end
_else:
       sub $s0, $s3, $s4     # $s0 = $s3 - $s4
_end:
```

Simple While Statement

```
i = 0;                        // $t0 = 0
while (i < 16) {              // while ($t0 < 16)
   r = r + x * y;             // $s0 = $s0 + $s1 * $s2
   i = i + 1;                 // $t0 = $t0 + 1
}
-----------------------------------------------------------------------


       li    $t0, 0          # $t0 = 0
       li    $t1, 16         # $t1 = 16
_loop:
       bge   $t0, $t1, _end  # $t0 >= $t1 → _end
       mul   $t2, $s1, $s2   # $t2 = $s1 * $s2
       add   $s0, $s0, $t2   # $s0 = $s0 + $t2
       addiu $t0, $t0, 1     # $t0 = $t0 + 1
       j     _loop           # → _loop
_end:
```

Simple Do-While Statement

```
i = 0;                          // $t0 = 0
do {
   r = r + x * y;               // $s0 = $s0 + $s1 * $s2
   i = i + 1;                   // $t0 = $t0 + 1
} while (i < 16);               // while ($t0 < 16)
-----------------------------------------------------------------------


       li    $t0, 0             # $t0 = 0
       li    $t1, 16            # $t1 = 16
_loop:
       mul   $t2, $s1, $s2      # $t2 = $s1 * $s2
       add   $s0, $s0, $t2      # $s0 = $s0 + $t2
       addiu $t0, $t0, 1        # $t0 = $t0 + 1
       bge   $t0, $t1, _end     # $t0 >= $t1 → _end
       j     _loop              # → _loop
_end:
```

Simple For Statement

```
for (i = 0; i < 16; i++) {      // for ($t0 = 0; $t0 < 16; $t0++)
   r = r + x * y;               // $s0 = $s0 + $s1 * $s2
   x = r;                       // $s1 = $s0
}
------------------------------------------------------------------------

       li    $t0, 0             # $t0 = 0
       li    $t1, 16            # $t1 = 16
_loop:
       bge   $t0, $t1, _end     # $t0 >= $t1 → _end
       mul   $t2, $s1, $s2      # $t2 = $s1 * $s2
       add   $s0, $s0, $t2      # $s0 = $s0 + $t2
       move  $s1, $s0           # $s1 = $s0
       addiu $t0, $t0, 1        # $t0 = $t0 + 1
       j     _loop              # → _loop
_end:
```

Simple Switch-Case Statement

```
switch (x) {                           // switch ($s1)
   case 0: y = z; break;               // _l0: $s2 = $s3
   case 1: y = w; break;               // _l1: $s2 = $s4
   case 2: y = z + w; break;           // _l2: $s2 = $s3 + $s4
   case 3: y = z - w; break;           // _l3: $s2 = $s3 - $s4
}                                      // $s0 = jumpTable[_l0, _l1, _l2, _l3]
--------------------------------------------------------------------


       slti  $t0, $s1, 0               # $t0 = ($s1 < 0)
       bne   $t0, $zero, _end          # $t0 != 0 → _end
       slti  $t0, $s1, 4               # $t0 = ($s1 < 4)
       beq   $t0, $zero, _end          # $t0 == 0 → _end
       sll   $t1, $s1, 2               # $t1 = $s1 * 4
       add   $t1, $s0, $t1             # $t1 = $s0 + $t1
       lw    $t0, 0($t1)               # $t0 = $t1[0] = jumpTable[$s1]
       jr    $t0                       # → [_l0, _l1, _l2, _l3]
       ...
```

Simple Switch-Case Statement

```
switch (x) {                      // switch ($s1)
   case 0: y = z; break;          // _l0: $s2 = $s3
   case 1: y = w; break;          // _l1: $s2 = $s4
   case 2: y = z + w; break;      // _l2: $s2 = $s3 + $s4
   case 3: y = z - w; break;      // _l3: $s2 = $s3 - $s4
}                                 // $s0 = jumpTable[_l0, _l1, _l2, _l3]
--------------------------------------------------------------------------

       ...
_l0:   move $s2, $s3
       j    _end
_l1:   move $s2, $s4
       j    _end
_l2:   add $s2, $s3, $s4
       j    _end
_l3:   sub $s2, $s3, $s4
_end:
```

Address Space

The MIPS address space is divided in four segments:

- **Text**, which contains the program code
- **Data**, which contains constants and global variables
- **Heap**, which contains memory dynamically allocated during runtime
- **Stack**, which contains temporary data for handling procedure calls

The heap and stack segments grow toward each other, thereby allowing the efficient use of memory as the two segments expand and shrink.

Address Space

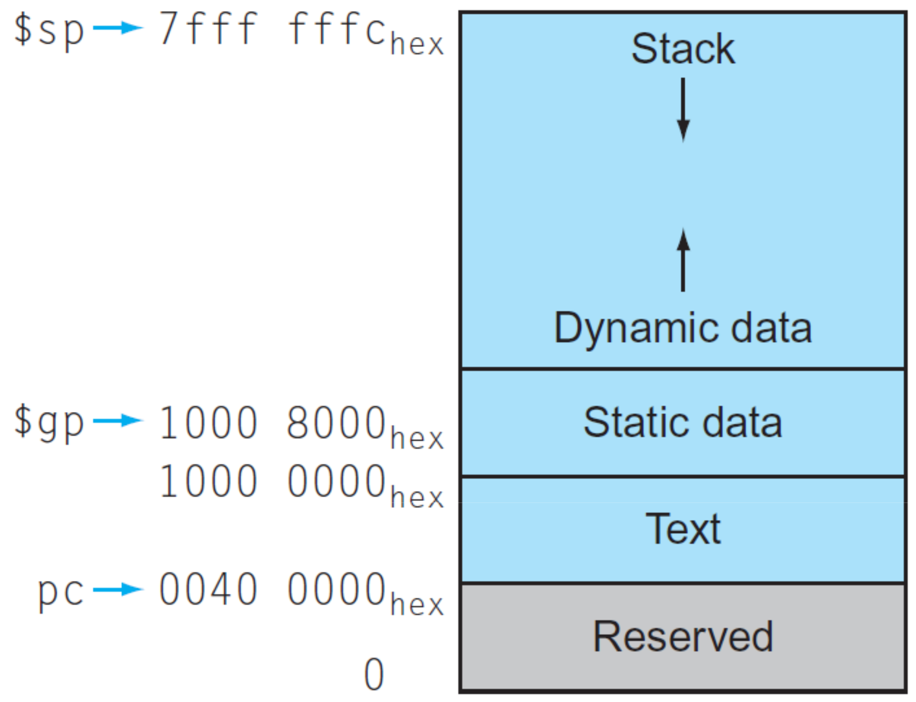


FIGURE 2.13 The MIPS memory allocation for program and data. These addresses are only a software convention, and not part of the MIPS architecture. The stack pointer is initialized to 7fff fffc$_{hex}$ and grows down toward the data segment. At the other end, the program code ("text") starts at 0040 0000$_{hex}$. The static data starts at 1000 0000$_{hex}$. Dynamic data, allocated by malloc in C and by new in Java, is next. It grows up toward the stack in an area called the heap. The global pointer, $gp, is set to an address to make it easy to access data. It is initialized to 1000 8000$_{hex}$ so that it can access from 1000 0000$_{hex}$ to 1000 ffff$_{hex}$ using the positive and negative 16-bit offsets from $gp. This information is also found in Column 4 of the MIPS Reference Data Card at the front of this book.

Procedure Calls

The execution of a procedure call happens when one procedure (the **caller**) invokes another procedure (the **callee**).

In general, the execution of a procedure call follows six steps:

- Put arguments in a place where the callee can access them
- Transfer control to the callee
- Acquire storage resources needed for callee execution
- Perform callee's operations
- Put results in a place where the caller can access them
- Return control to the caller's next instruction

Procedure Calls

How to ensure that a procedure call does not change data that is outside its scope?

Programmers who write code in a high-level language never see the details of how one procedure calls another because **the compiler takes care of the low-level details**.

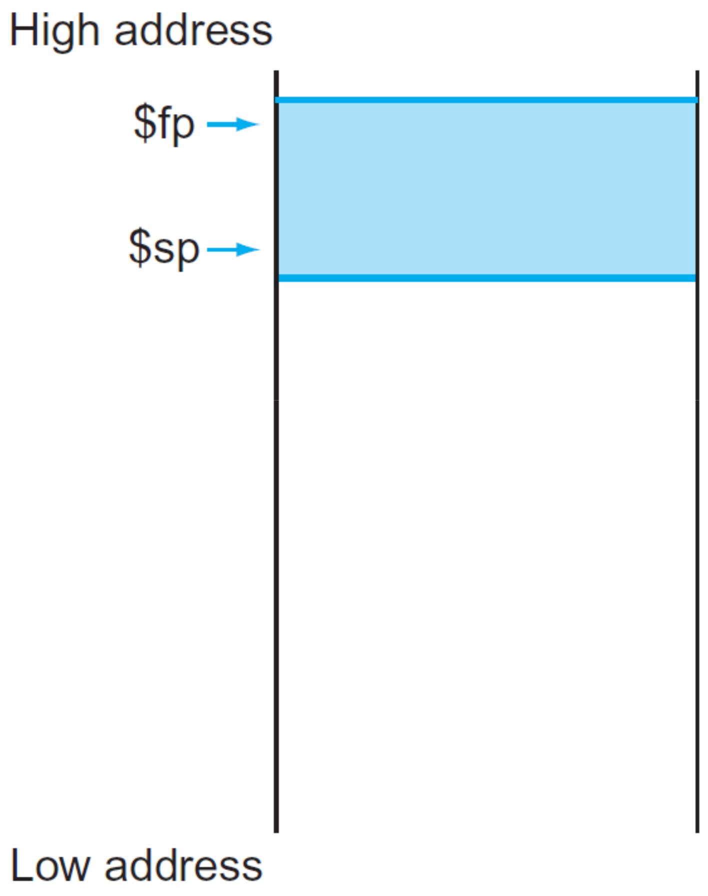
Programmers who write code in assembly **must explicitly implement every procedure call and return**.

- The caller may have to save data before calling the callee
- The callee may have to save data before running its operations

Procedure Calls

The bookkeeping associated with procedure calls is done in the **stack segment** around blocks of memory called **procedure frames**.

- Register **$fp (frame pointer)** points to the **base of the current procedure frame** and offers a stable base register as it does not change in a procedure
- Register **$sp (stack pointer)** points to the **top of the current procedure frame** and since it can change within a procedure, different references to the same (local) variable might have different offsets in the procedure



Stack Pointer

By historical precedent, the **stack grows from higher addresses to lower addresses**. This convention means that you push values onto the stack by subtracting from the stack pointer. Adding to the stack pointer shrinks the stack, thereby popping values off the stack.

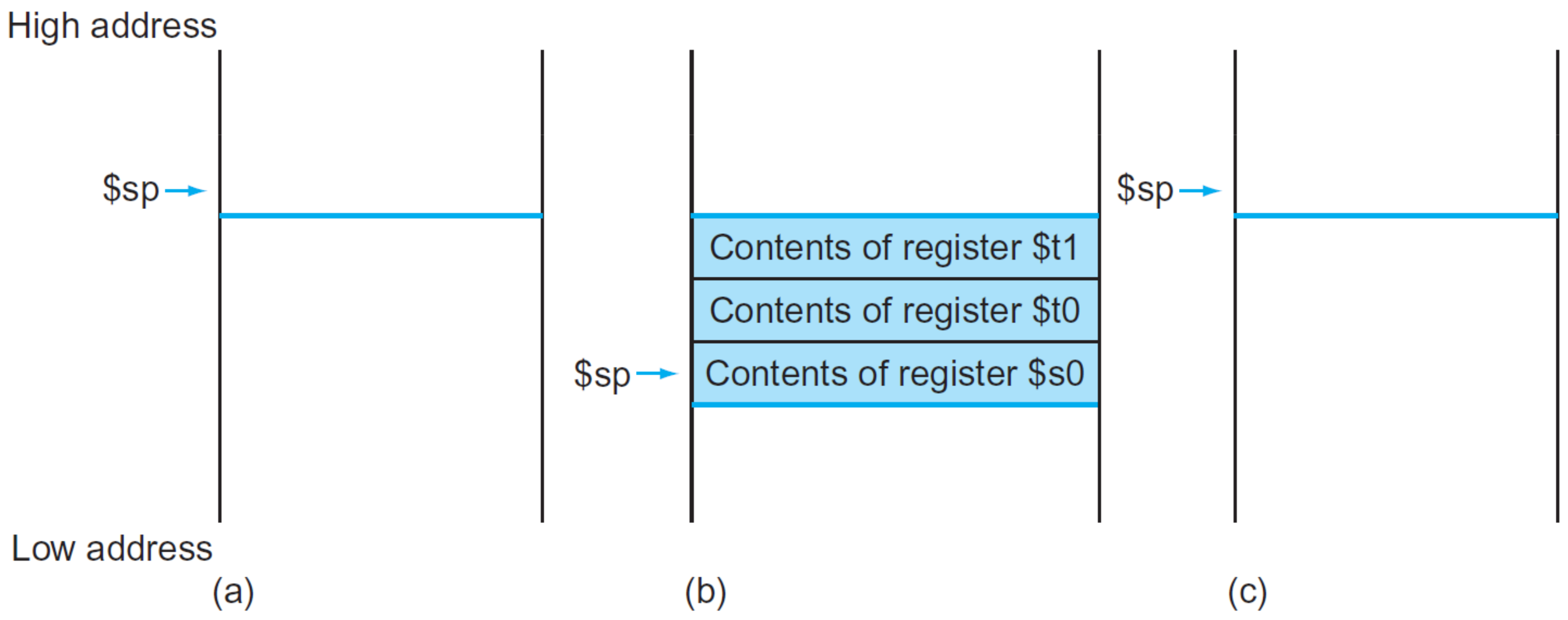


FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

Procedure Call Support

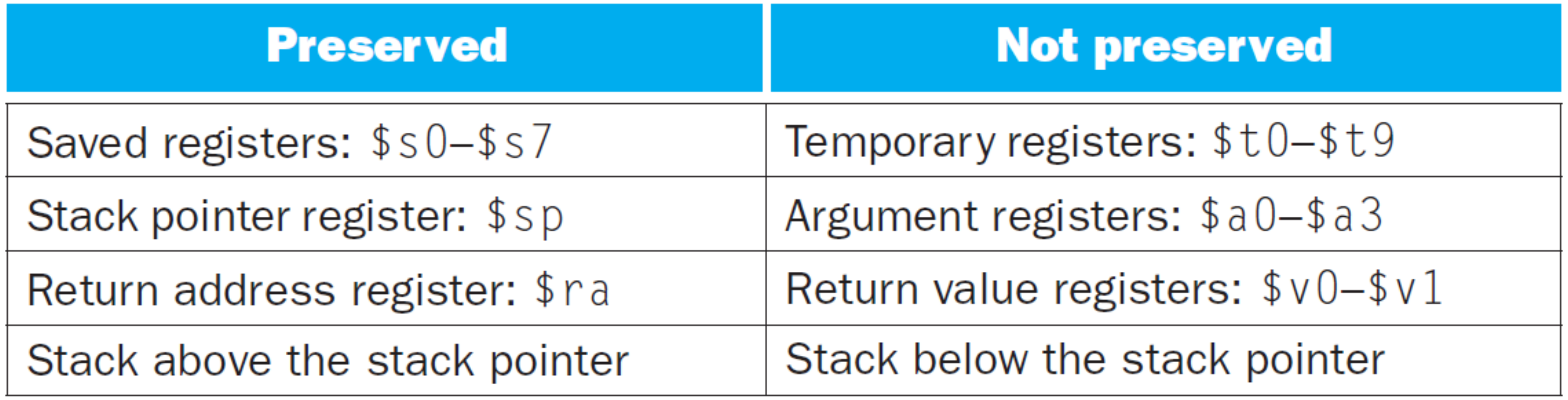
MIPS conventions for procedure calling:

- **\$a0 – \$a3 registers** are used to **pass the first 4 arguments to the callee**
- **\$v0 – \$v1 registers** are used to **return values to the caller**
- **\$t0 – \$t9 registers** are used to **hold temporary values** that can be overwritten by the callee
- **\$s0 – \$s7 registers** are used to **hold long-lived values** that should be preserved across calls
- **\$sp register** is the pointer to the **current top location in the stack**
- **\$ra register** is the return address to the **caller's next instruction**
- **jump-and-link instruction (jal)** jumps to an address and simultaneously saves the address of the following instruction **(PC + 4)** in register **\$ra**
- **jump register instruction (jr)** jumps to the address stored in register **\$ra**

Preserved or Not Preserved

What is preserved across a procedure call?

- **$sp is preserved** by the callee by adding exactly the same amount that was subtracted from it
- **Stack above $sp is preserved** by making sure the callee does not write above $sp, i.e., the caller will get the same data back on a load from the stack as it was stored there
- **Other registers can be preserved** by saving them on the stack (if they are used) and restoring them from there, specially **registers $s0–$s7** and **register $ra**



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

Caller Side

Save not preserved registers

- If the caller expects to use not preserved registers ($t0 – $t9, $a0 – $a3 and $v0 – $v1) after the call, save its values before the call in the current procedure frame

Pass arguments

- The first 4 arguments are put in registers $a0 – $a3
- Additional arguments are pushed on the stack and appear at the beginning of the procedure frame (register $fp points to the base of the procedure frame)

Transfer control to the callee

- Execute a jal instruction to jump to the callee’s first instruction and save the return address in $ra

Callee Side

Allocate memory (and update stack pointer)

- Add a new procedure frame by subtracting the required size from $sp

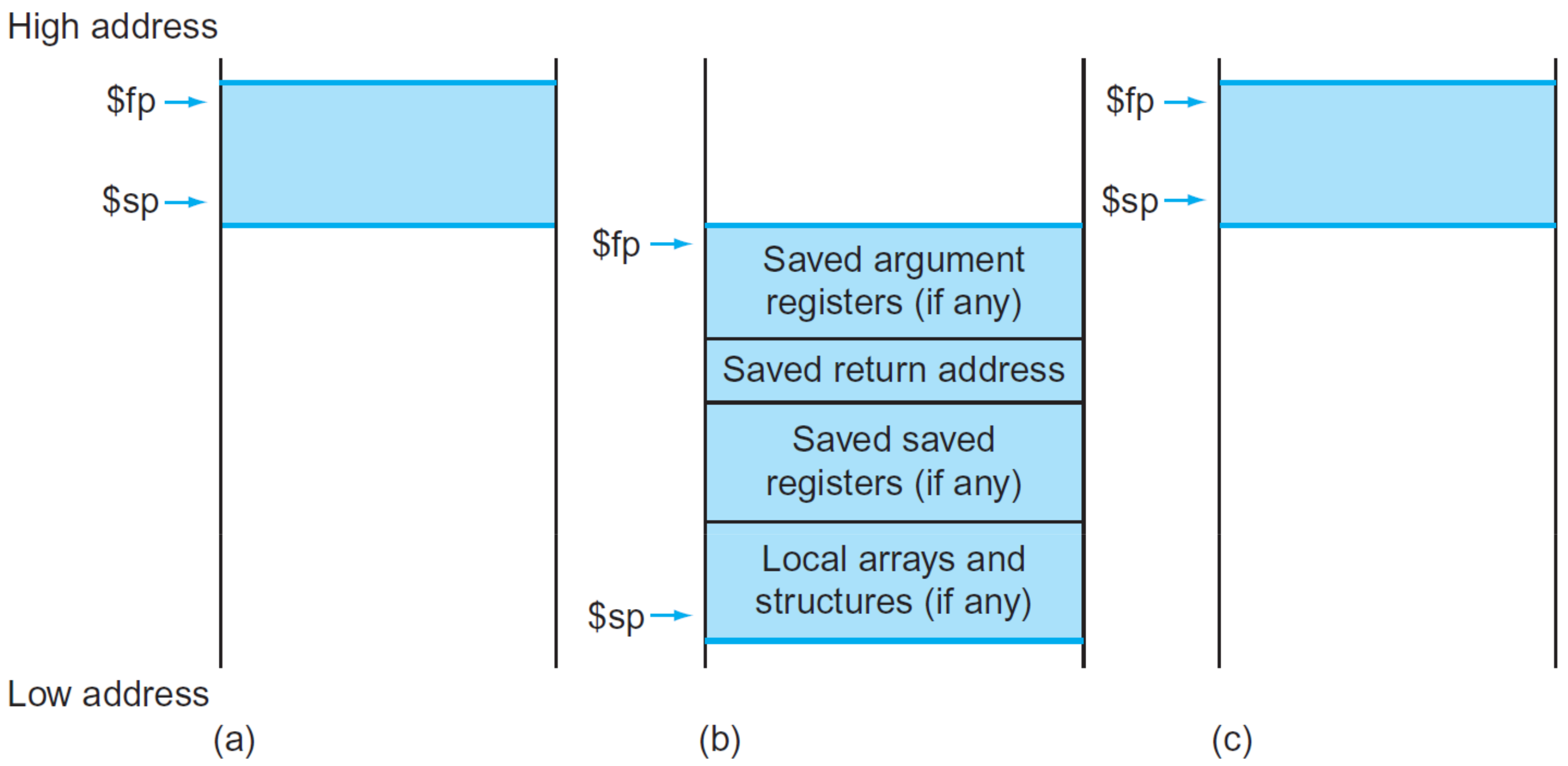
Save preserved registers (and update frame pointer)

- If the callee expects to alter preserved registers ($fp, $ra and $s0 – $s7), save its values in the new procedure frame before altering them ($fp only needs to be saved if the frame's size is not zero; $ra only needs to be saved if the callee itself makes a call)
- Update $fp by adding the new frame's size minus 4 to $sp

Put results and return control to the caller

- If the callee returns something, put the result(s) in $v0 – $v1
- Restore all callee-saved registers ($fp, $ra and $s0 – $s7)
- Pop the procedure frame by adding its size to $sp
- Execute a jr instruction to return by jumping to the address in $ra

Procedure Frame



High address
$fp
$sp
$fp
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
$sp
$fp
$sp
Low address
(a)
(b)
(c)

FIGURE 2.12 Illustration of the stack allocation (a) before, (b) during, and (c) after the procedure call. The frame pointer (`$fp`) points to the first word of the frame, often a saved argument register, and the stack pointer (`$sp`) points to the top of the stack. The stack is adjusted to make room for all the saved registers and any memory-resident local variables. Since the stack pointer may change during program execution, it's easier for programmers to reference variables via the stable frame pointer, although it could be done just with the stack pointer and a little address arithmetic. If there are no local variables on the stack within a procedure, the compiler will save time by *not* setting and restoring the frame pointer. When a frame pointer is used, it is initialized using the address in `$sp` on a call, and `$sp` is restored using `$fp`. This information is also found in Column 4 of the MIPS Reference Data Card at the front of this book.

Simple Procedure Call

```
int mult (int x, int y) {          // arguments in $a0 and $a1
   int r = x * y;                  // r in $s0, need to save $s0 on stack
   return r;
}                                  // result in $v0
------------------------------------------------------------------------


_main:
       ...
       li    $a0, 10               # put argument $a0
       li    $a1, 20               # put argument $a1
       jal   _mult                 # jump and link
       ...
```

Simple Procedure Call

```
int mult (int x, int y) {          // arguments in $a0 and $a1
   int r = x * y;                  // r in $s0, need to save $s0 on stack
   return r;
}                                  // result in $v0
------------------------------------------------------------------------


_mult:
       addiu $sp, $sp, -4          # adjust stack pointer
       sw    $s0, 0($sp)           # save $s0

       mul   $s0, $a0, $a1         # $s0 = $a0 * $a1
       move  $v0, $s0              # return value

       lw    $s0, 0($sp)           # restore $s0
       addiu $sp, $sp, 4           # restore stack pointer
       jr    $ra                   # return
```

Simple Procedure Call (Optimized Version)

```
int mult (int x, int y) {           // arguments in $a0 and $a1
   int r = x * y;                   // r in $t0, avoid saving $s0 on stack
   return r;
}                                   // result in $v0
--------------------------------------------------------------------------


_mult:
       mul   $v0, $a0, $a1          # return value
       jr    $ra                    # return
```

Recursive Procedure Call

```
int mult (int x, int y) {          // arguments in $a0 and $a1
   if (y == 0) return 0;
   return x + mult (x, y - 1);     // need to save $ra and $a0 on stack
}                                  // result in $v0
------------------------------------------------------------------------


_mult:
       addiu $sp, $sp, -8          # adjust stack pointer
       sw    $ra, 4($sp)           # save $ra
       sw    $a0, 0($sp)           # save $a0

       bne   $a1, $zero, _else     # $a1 != 0 → _else
       li    $v0, 0                # return value

       addiu $sp, $sp, 8           # restore stack pointer
       jr    $ra                   # return
       ...
```

Recursive Procedure Call

```
int mult (int x, int y) {         // arguments in $a0 and $a1
   if (y == 0) return 0;
   return x + mult (x, y - 1);    // need to save $ra and $a0 on stack
}                                 // result in $v0
-----------------------------------------------------------------------
       ...
_else:
       addiu $a1, $a1, -1         # $a1 = $a1 - 1
       jal   _mult                # recursive call
       lw    $a0, 0($sp)          # restore $a0
       add   $t0, $a0, $v0        # $t0 = $a0 + $v0
       move  $v0, $t0             # return value

       lw    $ra, 4($sp)          # restore $ra
       addiu $sp, $sp, 8          # restore stack pointer
       jr    $ra                  # return
```

Recursive Procedure Call (Optimized Version)

```
_mult:
        bne   $a1, $zero, _else   # $a1 != 0 → _else
        li    $v0, 0              # return value
        jr    $ra                 # return
_else:
        addiu $sp, $sp, -4        # adjust stack pointer
        sw    $ra, 0($sp)         # save $ra
        addiu $a1, $a1, -1        # $a1 = $a1 - 1
        jal   _mult               # recursive call
        add   $v0, $a0, $v0       # return value
        lw    $ra, 0($sp)         # restore $ra
        addiu $sp, $sp, 4         # restore stack pointer
        jr    $ra                 # return
```

Program Structure

```
        .data                 # data segment (constants and global variables)
_b1:    .byte 1               # byte (8 bits) with value 1
_h1:    .half 10              # half word (16 bits) with value 10
_w1:    .word 100             # word (32 bits) with value 100
_a1:    .byte 1, 2, 3, 4      # array of 4 bytes with values 1, 2, 3 and 4
_a2:    .word 0:100           # array of 100 words with values 0
_s1:    .ascii  "abc\n"       # string not null terminated
_s2:    .asciiz "123"         # string null terminated"
_e1:    .space 100            # leave 100 bytes of space

        .text                 # text segment (program instructions)
_main:                        # main procedure
        ...
        li $v0, 10            # load code 10 for system call exit()
        syscall               # exit()
```

System Calls

To request a service, **load the system call code into register $v0** and **arguments into registers $a0–$a3 or $f12** (floating point values). Return values are put in **register $v0 or $f0** (floating-point results).

Service	System call code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		
print_char	11	$a0 = char	
read_char	12		char (in $v0)
open	13	$a0 = filename (string), $a1 = flags, $a2 = mode	file descriptor (in $a0)
read	14	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars read (in $a0)
write	15	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars written (in $a0)
close	16	$a0 = file descriptor	
exit2	17	$a0 = result	

Endianness (Little-Endian)

```
        .data
mem:    .word 0xABCDE080   # memory contents: 80 E0 CD AB

        .text
_main: lw  $t0, mem        # $t0 = 0xABCDE080
       lh  $t1, mem        # $t1 = 0xFFFFE080
       lb  $t2, mem        # $t2 = 0xFFFFFF80
       lhu $t3, mem        # $t3 = 0x0000E080
       lbu $t4, mem        # $t4 = 0x00000080
```

RISC Design Principles in MIPS

Design Principle 1: Simplicity favors regularity

- Few addressing modes
- Three register operands in arithmetic instructions, keeping the register fields in the same place in each instruction format

Design Principle 2: Smaller is faster

- Use just 32 registers

Design Principle 3: Good design demands good compromises

- Same instruction length

Design Principle 4: Make common case faster

- Specific instructions (e.g., addiu)
- Most procedures are satisfied with 4 arguments, 2 registers for a return value, 8 saved registers, and 10 temporary registers without ever going to memory