

STEPS IN CREATING AN EXECUTABLE PROGRAM

- Creation of source “code”
 - ▷ Accomplished with a text editor or a programming environment
- Compilation (if source program is in a higher-level language)
 - ▷ Includes one of various levels of optimization
 - ▷ Result of compilation may be an assembly-language program
- Assembly produces an **object module**
 - ▷ Human-readable instructions and macros \mapsto machine language
- Linking (a.k.a. link editing) produces a **load module**
 - ▷ Resolution of calls to user and library functions
 - ▷ Unresolved function calls produce run-time exceptions
- Loading produces a memory-resident executable module
 - ▷ Memory references to code and data are updated to reflect actual locations
 - ▷ Executable program is copied into memory

A C PROGRAM AND ITS TRANSLATION INTO MIPS ASSEMBLER

- C program:

```
swap( int v[] , int k )  
{ int temp;  
    temp = v[k];  
    v[k] = v[k+1];  
    v[k+1] = temp;  
}
```

- ▷ Purpose of this program: swap two array elements
 - Note the use of a temporary variable (**temp**)
- ▷ Compilation into MIPS assembly language was performed with the **lcc** compiler (see following page)
- ▷ Using a temporary variable in assembler requires unnecessary memory traffic (an extra store and an extra load)
- ▷ All that is needed is to load both array elements and then swap them when storing (see the page after next)

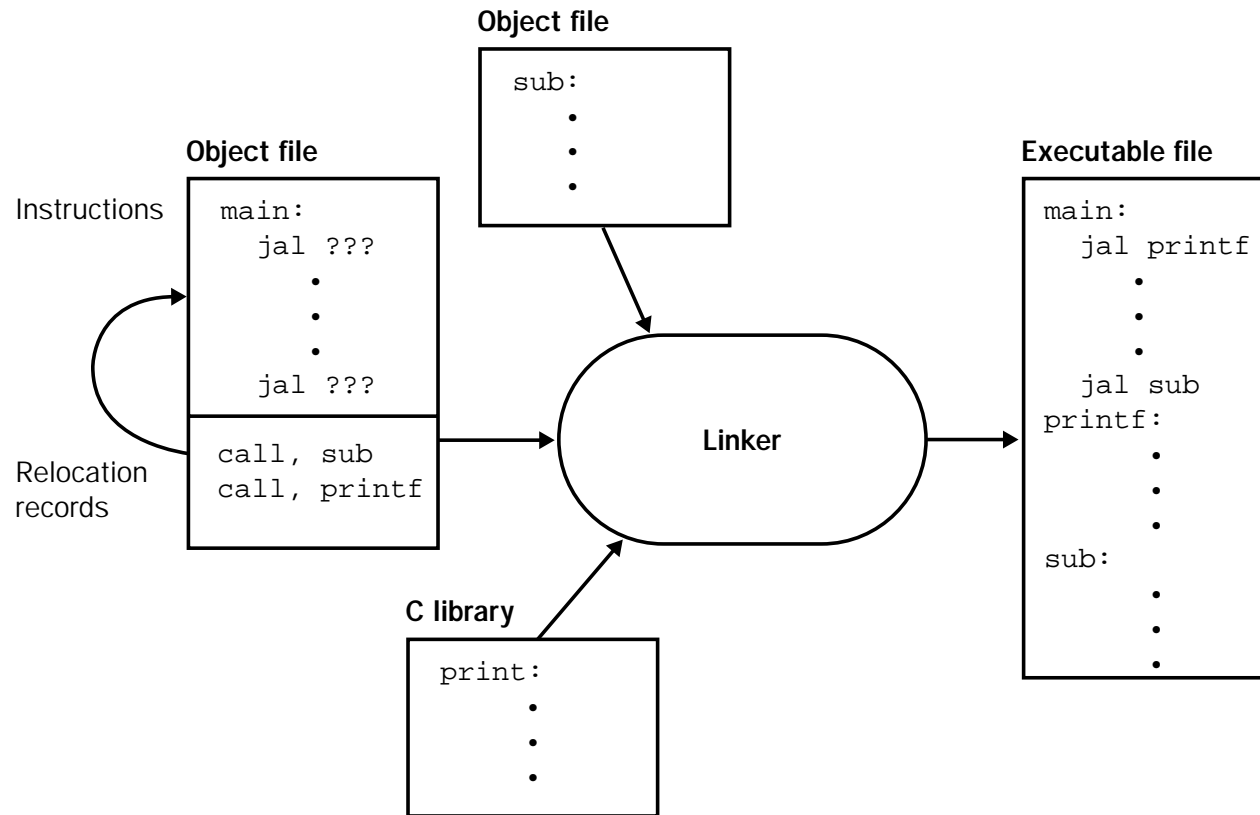
UNIX OBJECT FILE FORMAT

- Object file header
 - ▷ Describes the sizes and positions of the remaining parts
- Text segment
 - ▷ Contains the machine language code
- Data segment
 - ▷ Includes both static and dynamic data
- Relocation information
 - ▷ Identifies instructions and data that depend on absolute addresses
- Symbol table
 - ▷ Contains undefined labels (e.g., external references)
- Debugging information

FUNCTIONS OF THE LINK EDITOR (LINKER)

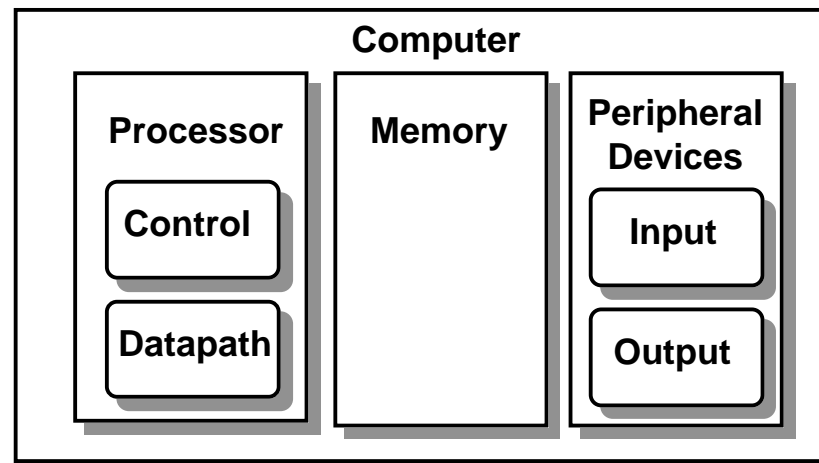
- Goal: To confine recompilation to those modules that have changed
 - ▷ Each procedure can be compiled independently
- The linker:
 - ▷ Places text and data modules symbolically in memory
 - ▷ Determines the addresses that correspond to the labels that the programmer has given to instructions and data
 - ▷ Patches the external and internal memory references
 - ▷ Produces an executable file
- Executable file format
 - ▷ Same as the format of an object file, except that there are no unresolved references, relocation information, symbol table, or debugging information

FUNCTIONS OF THE LINK EDITOR/LOADER



COMPONENTS OF AN INSTRUCTION SET ARCHITECTURE

- Data types supported in hardware
- Computational operations supported in hardware
- Memory access and addressing modes
- Branch implementation
- Support for procedure calls
- Instruction encoding
- Interrupt and exception handling



INSTRUCTION SET DESIGN

- A key step in both hardware and software design
 - ▷ Software:
 - The instruction set architecture (ISA), and its implementation, determine what kinds of programs perform well
 - ▷ Hardware:
 - Up to now, we have worked at the gate level
 - In order to design or understand an ISA, we have to work at a higher level of organization, using modules or functional units
 - ◇ Main memory
 - ◇ Cache and cache controller
 - ◇ Register file
 - ◇ Integer unit (ALU)
 - ◇ Floating-point unit (FPU)
 - ◇ Branch unit
 - ◇ Exception/interrupt processing unit
 - ◇ Vector unit

INSTRUCTION SET DESIGN ISSUES

- What operations should be provided in hardware?
 - ▷ An optimization problem
 - ▷ Any computation can be done using only load/store/increment/branch
 - Resulting code: large, with many memory references \Rightarrow slow
- How many / what kinds of operands?
 - ▷ Must provide for one operand; should provide for two
 - ▷ Examples: $a \mapsto \bar{a}$, $\langle a, b \rangle \mapsto c$
- What addressing modes should be implemented?
 - ▷ Choices made here affect complexity of both datapath and control
 - ▷ Fixed-length vs. variable-length instructions
- How to encode operations in a few bits?
 - ▷ Hardware design issue: how to translate operation encoding and addressing into control signals?
- How should interrupts and exceptions be handled?

KINDS OF INSTRUCTION SET ARCHITECTURES

- Accumulator
 - ▷ Requires only one address
 - ▷ Example: `add x` means `acc` \mapsto `acc + M[x]`
- Stack
 - ▷ Requires no explicit memory addresses
 - ▷ Example: `add` means `tos` \mapsto `tos + next`
- General-purpose register set
 - ▷ Some operands may be in memory
 - ▷ Example of adding two numbers in memory locations `a` and `b`:
`load r1,a; add r1,b; store r1,c`
 - ▷ Load/store architectures
 - All arithmetic/logical operations use register operands
 - Only `load` and `store` instructions reference memory

ADVANTAGES AND DISADVANTAGES OF VARIOUS ISAs

- Accumulator
 - + Minimizes internal state of CPU
 - Maximizes memory-to-CPU traffic
- Stack
 - + Reverse-polish expression evaluation (like HP calculator)
 - Stack can't be randomly accessed \Rightarrow bottleneck
- General-purpose register set
 - + Most general model for code generation
 - Each operand must be named \Rightarrow longer instructions
- Load/store
 - + Fixed-length instruction encoding, with one addressing mode
 - ▷ Simplifies hardware datapath and control \Rightarrow maximum speed
 - Increases total instruction count

WHY REGISTER ARCHITECTURES ARE DOMINANT

- Since 1980, logic circuits have become much faster than dynamic RAM
 - ▷ Registers are faster than main memory
 - ▷ Registers are faster than a stack
 - A stack is built in main memory
 - Operands that are in registers can be used in any order
 - ▷ Registers can hold variables as they are updated in a multi-step computation
 - Reduction of main memory traffic makes computations faster
 - Register addresses (typically 5 bits) are shorter than main memory addresses (typically 32 to 64 bits)
 - ◇ Code density (information per bit in an instruction) is higher in load-store architectures than in architectures that allow memory-resident operands
 - ◇ Short, fixed-length addressing \Rightarrow fixed-length instructions
 - For a register machine, one can optimize the most frequently used instructions (which are also simple, by experiment)

The top 10 instructions for the 80x86

Rank	80x86 instruction	Integer average (% total executed)
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
Total		96%

ASSEMBLERS (1)

- Internal components of a computer system:
 - ▷ Integer processor (CPU)
 - integer registers
 - integer arithmetic
 - logical operations
 - ▷ Floating-point coprocessor (floating-point unit, FPU)
 - floating-point registers
 - floating-point arithmetic
 - ▷ Random-access memory (RAM)
 - ▷ I/O
- Assembly-language instructions control each of these components
 - ▷ One line of assembly code \Rightarrow one hardware instruction (usually)

ASSEMBLERS (2)

- How to design a programming language using only hardware instructions?
 - ▷ Must take care of dataflow and control
 - ▷ Data segment(s) of program
 - Begins with a special directive (.data)
 - Static (assembly-time) memory allocations for inputs & outputs of program
 - Area for dynamic (run-time) memory allocation
 - ▷ Control (“text”) segment(s) of program
 - Uses hardware instructions to tell the processor what to do with the inputs & how to obtain the outputs
 - Implicitly allocates memory for instructions

ASSEMBLERS (3)

- General syntax of an assembly-language program:
 - ▷ Only one statement per line
 - ▷ Fixed format
 - ▷ Statements may have names (labels) to make programming easier
 - ▷ Kinds of statements (explained in detail on later slides):
 - Comments
 - Declarations
 - Assignment statements
 - Branch/jump instructions
 - Communication with user or OS

ASSEMBLERS (4)

- Parts of a MIPS assembly-language program:
 - ▷ Comments
 - Everything on a line after a # symbol is a comment
 - Comments may not span more than one line
 - ▷ Directives and declarations
 - Directives (`.data`, `.text`, `.stack`)
 - ◇ Tell the assembler about the program's major memory regions (data, instructions, stack)
 - Declarations of variables
 - ◇ Tell the assembler how much memory is needed for variables
 - ◇ Assign names to memory blocks
 - ◇ Form:
`label: .keyword value:number_of_elements`

SAL — SIMPLE ABSTRACT LANGUAGE

- Motivation for SAL:
 - ▷ SAL purposely hides the details of MIPS assembly language
 - ▷ SAL code looks a lot like C or Pascal code
 - ▷ Introduces a useful level of abstraction between higher-level languages and true assembly language
- Each C or Pascal statement translates to 1 or more SAL instructions
- Each SAL instruction translates to 1 or more MIPS assembly language instructions
- The SPIM/SAL program (Windows and Macintosh) accepts both SAL and MIPS assembly language instructions

SPIM

- Motivation for SPIM:
 - ▷ The MIPS simulator known as SPIM accepts human-readable MIPS assembly language instructions
 - SPIM code uses labels and pseudoinstructions to hide some of the details of the actual hardware instructions
 - Introduces a small amount of abstraction between the programmer and the hardware
 - ▷ Each SAL instruction translates to 1 or more MIPS assembly language instructions
- There's also True Assembly Language (TAL)
 - ▷ TAL instructions correspond exactly to the hardware instructions produced by the assembler
 - ▷ Each SPIM instruction translates to 1 or more TAL instructions
 - ▷ TAL instructions can be inspected in SPIM's "Text" window

MIPS DATA TYPES

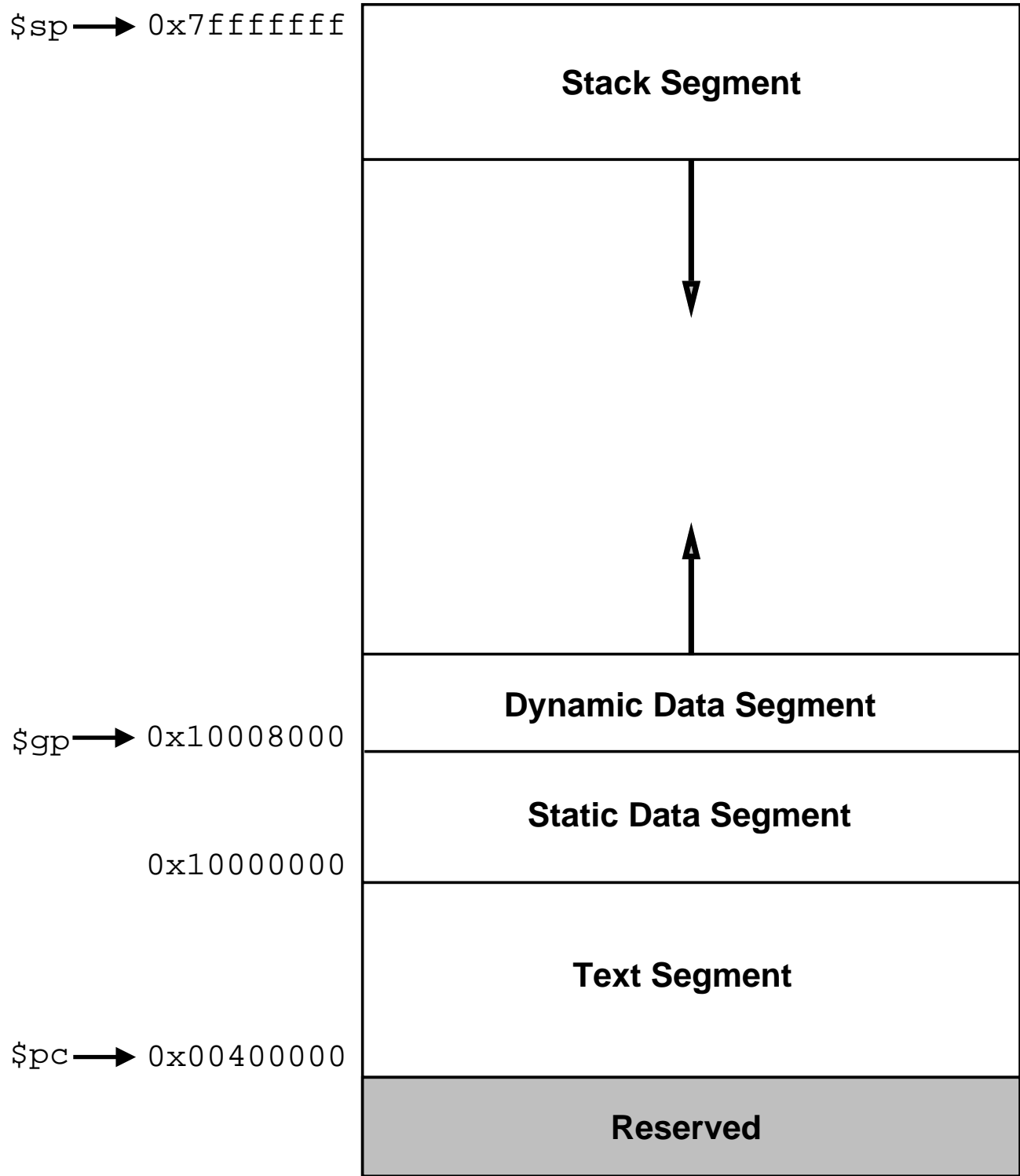
- Integer (several different sizes)
- Floating point (2 sizes)
- Character (really 1-byte unsigned integer)

VARIABLE DECLARATIONS (1)

- Static variables and arrays belong in a section that begins with `.data`
- Character and numerical variables:

C	SPIM/SAL
<code>char c;</code>	<code>c: .byte value:num_elements</code>
<code>int n;</code>	<code>n: .word value:num_elements</code>
<code>float x;</code>	<code>x: .float value:num_elements</code>

- ▷ Initial values are optional
- ▷ **value** must be appropriate for the type of data being stored:
 - **byte**: **value** is an ASCII character enclosed in single quotes(' ')
 - For **word**, **value** must be an integer
 - Example for **float**: **value** is `3.2767e4` (= 32,767)



ALIGNMENT

- In MIPS assembler (and in most RISC assemblers), the address of the low-address byte of a block of memory that holds a data type must be a multiple of the size of the data type
 - ▷ The address of a byte can be any unsigned integer within the address space
 - ▷ The address of a word (4 bytes) must be a multiple of 4
 - A word address ends with 2 zero bits (00)
 - Possible last hexadecimal digits in a word address: 0, 4, 8, C
 - ▷ The address of a doubleword (8 bytes) must be a multiple of 8
 - A doubleword address ends with 3 zero bits (000)
 - Possible last hexadecimal digits in a doubleword address: 0, 8
- The directive `.align n` aligns block addresses on multiples of `n`

VARIABLE DECLARATIONS (2)

- Strings in SPIM and SAL:
 - ▷ `label: .asciiz string` — Puts a null-terminated string into memory starting at location `label`. `string` is an ASCII string enclosed in double quotes (" ")
 - ▷ `label: .ascii string` — Puts a string into memory starting at location `label`. `string` is an ASCII string enclosed in double quotes (" ")
- Why put a null byte (`0x00`) at the end of a string?
 - ▷ C convention: A string is scanned up to a null byte (`\0`)
 - ▷ String length is determined only when data is read

ASSEMBLERS (5)

- Parts of a MIPS assembly-language program (continued):
 - ▷ Assignment statements
 - In a higher-level language, an assignment statement changes the value of a variable
 - When translated into assembly language, assignment statements become arithmetic, or logical, or data-transfer instructions
 - Form of an assembly-language arithmetic instruction:
`operation destination, source1, source2`
 - ▷ Loops
 - Must provide equivalents for C loops that use `while` and `for`

ASSIGNMENT STATEMENTS (1)

- An **assignment statement** changes the value of a variable
 - ▷ Belongs in a segment that begins with `.text`

C	SAL	action
<code>z = y;</code>	<code>move z,y</code>	$z \leftarrow (y)$
<code>z = y + x;</code>	<code>add z,y,x</code>	$z \leftarrow (y) + (x)$
<code>z = y - x;</code>	<code>sub z,y,x</code>	$z \leftarrow (y) - (x)$
<code>z = y*x;</code>	<code>mul z,y,x</code>	$z \leftarrow (y) \cdot (x)$
<code>z = y/x;</code>	<code>div z,y,x</code>	$z \leftarrow (y)/(x)$
<code>z = y%x;</code>	<code>rem z,y,x</code>	$z \leftarrow (y)(\text{mod}(x))$
<code>z = (type)y;</code>	<code>cvt z,y</code>	$z \leftarrow (y)$ (with type conversion)

- ▷ SAL allows memory-resident operands; SPIM does not
- ▷ (y) means the contents of the memory location with label y
- ▷ $z \leftarrow (y)$ means, “Store the contents of y into location z ”

ASSIGNMENT STATEMENTS (2)

- All assignment statements in SPIM must use operands that are in registers

C	SPIM	action
<code>z = y;</code>	<pre>lw \$rs, y sw \$rs, z</pre>	$z \leftarrow (y)$
<code>z = y + x;</code>	<pre>lw \$rs, x lw \$rt, y add \$rd, \$rs, \$rt sw \$rd, z</pre>	$z \leftarrow (y) + (x)$

- ▷ (y) means the contents of the memory location with label y
- ▷ x , y and z must be labels defined in a `.data` directive
- ▷ The same instruction sequence shown for `add` applies to other arithmetic instructions such as `sub`, `mul`, `div`, and `rem`

REGISTERS (1)

- A **register** is an array of flip-flops with special properties:
 - ▷ Registers are the fastest kind of addressable memory
 - ▷ Registers are located on the same die as the processor
 - Registers are **NOT** located in main memory
 - ▷ The set of registers in a processor is called the **register file**
- Registers are addressed separately & differently from main memory
 - ▷ MIPS R2000 architecture:
 - Main memory addresses are 32 bits
 - ⇒ 4,294,967,296 addressable 8-bit bytes
 - Register addresses are 5 bits
 - ⇒ 32 addressable 32-bit general-purpose registers
 - Register addresses run from $00000_2 = 0_{10}$ to $11111_2 = 31_{10}$
 - MIPS assembler: Register addresses are written \$0, ..., \$31

REGISTERS (2)

- Why are registers necessary?
 - ▷ The processor needs a “scratch pad” for operands and intermediate results
 - ▷ Some data items must be accessed in 1 clock period to avoid wait states
- Why isn't the register file a part of main memory?
 - ▷ Really fast memory is really expensive
 - ⇒ number of words in registers \ll number of words in main memory
 - ▷ Really fast memory is really hot
 - ⇒ major heat-dissipation problem, even if you're rich!
 - ▷ Propagation velocity of a signal \approx 6 inches per nanosecond
 - ⇒ fast memory must be physically close to the ALU & control unit

Immediate addressing

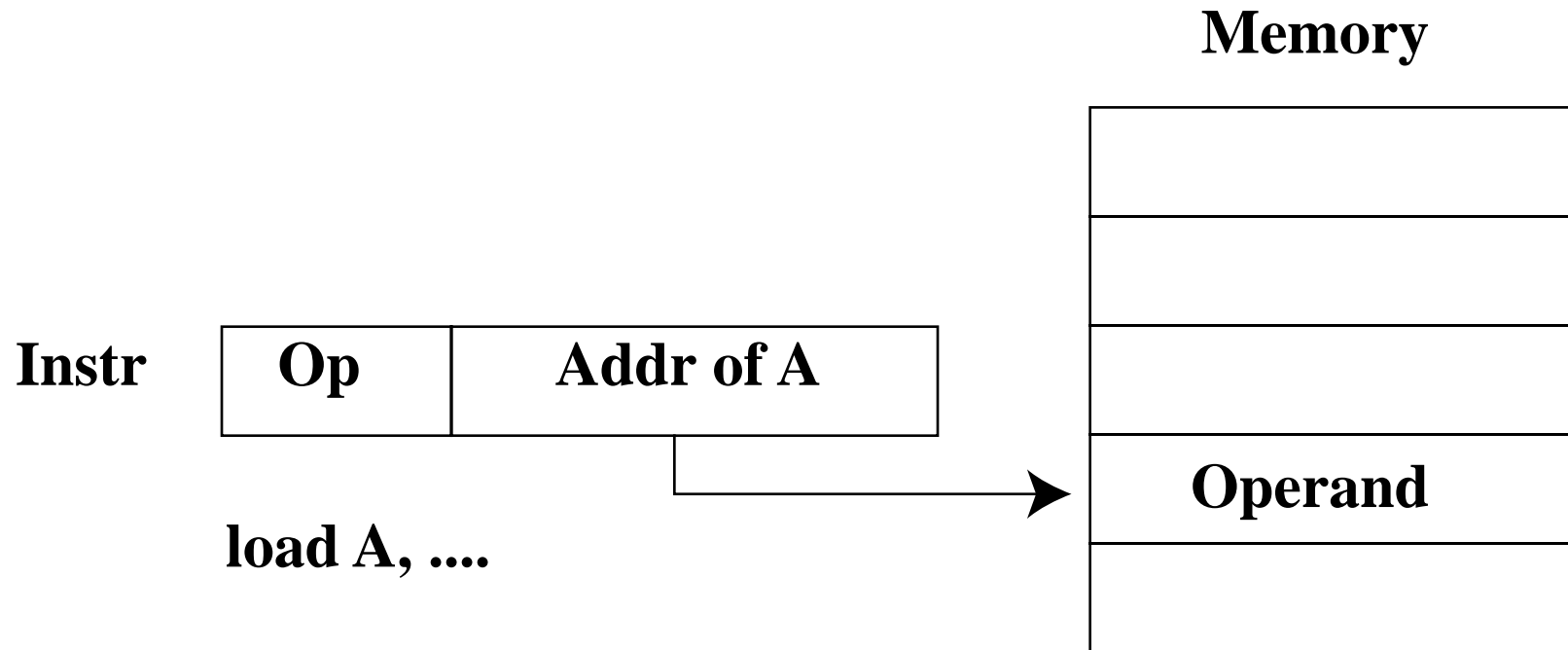
instruction contains
the operand



load #3,

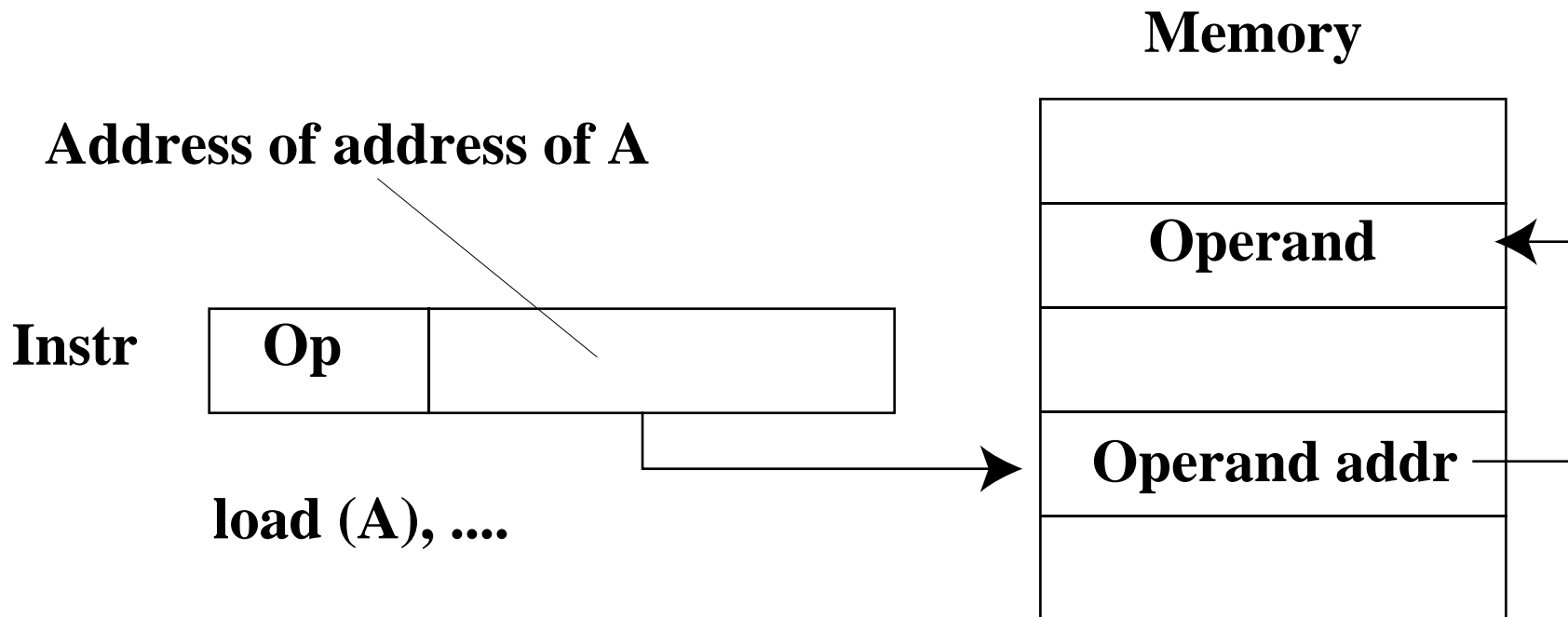
Direct addressing

instruction contains
address of operand



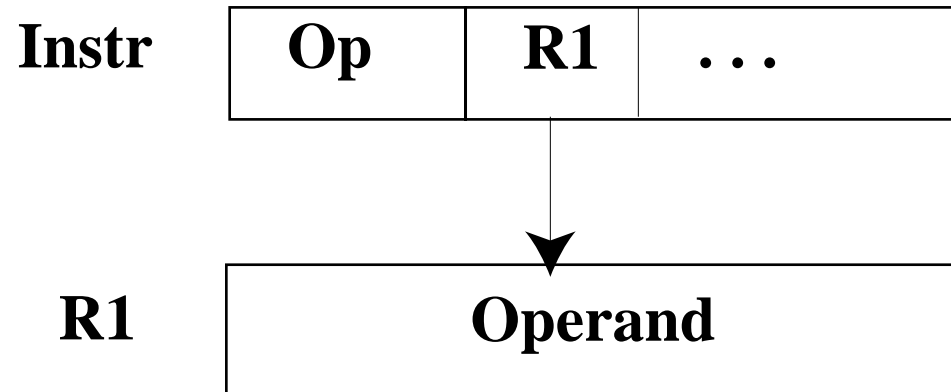
Indirect addressing

instruction contains
address of address
of operand



Register direct addressing

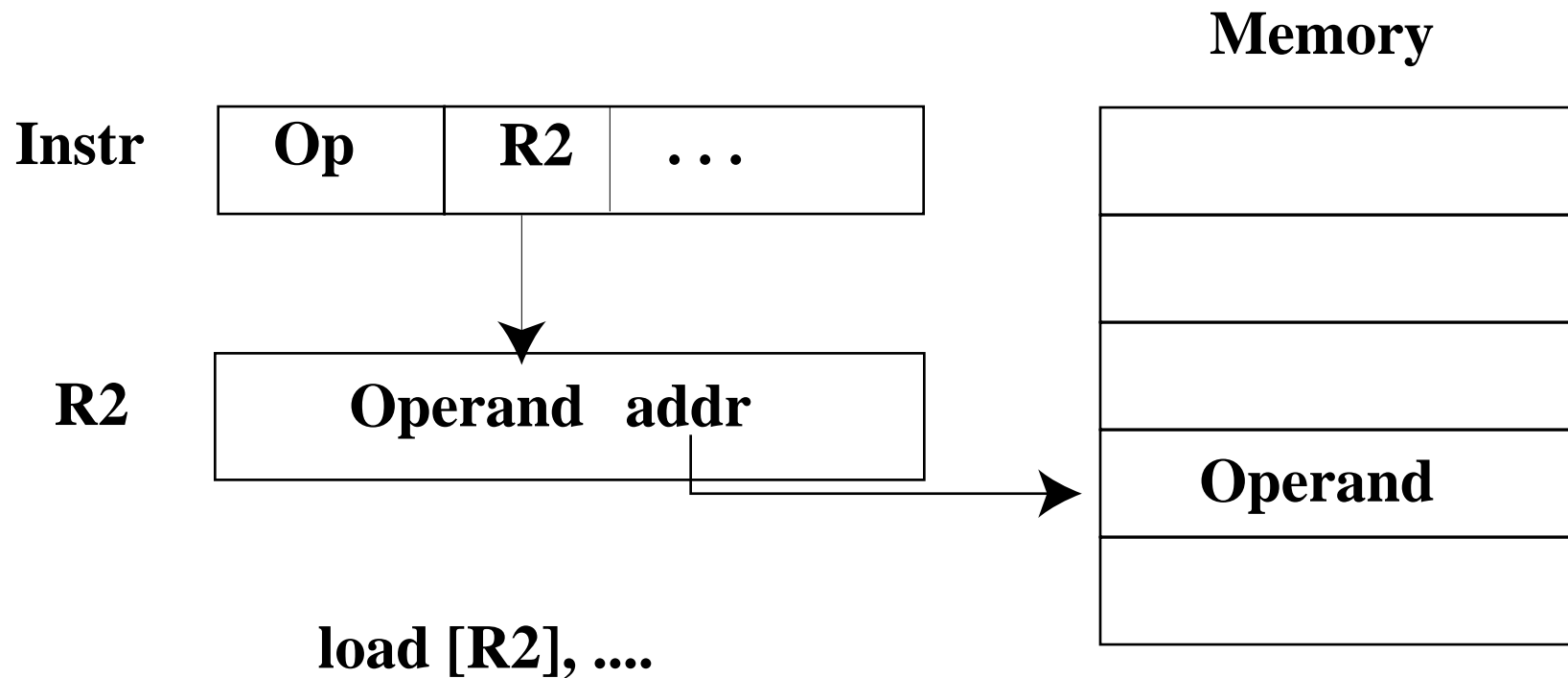
register contains
operand



load R1,

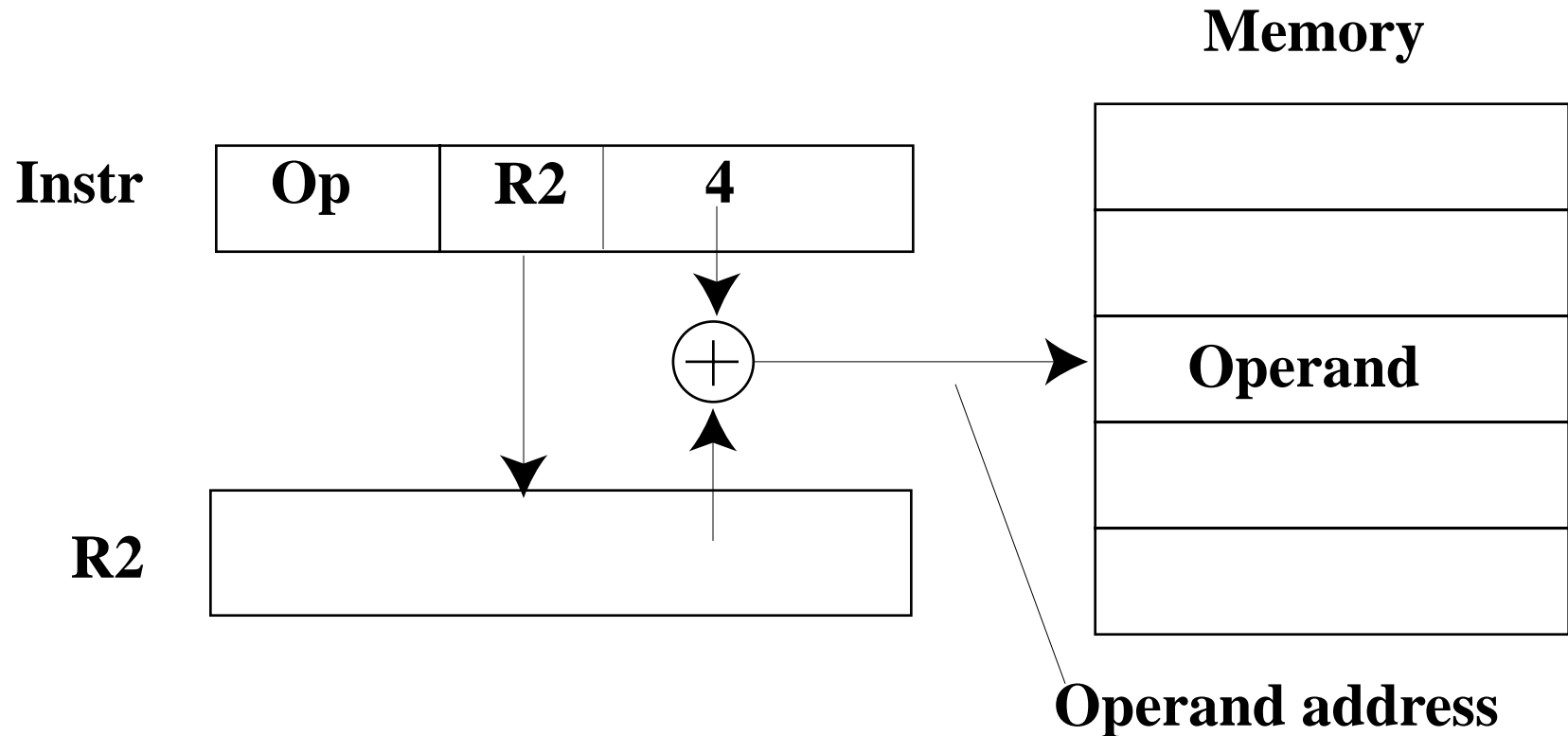
Register indirect addressing

register contains
address of
operand



Displacement (based or indexed) addressing

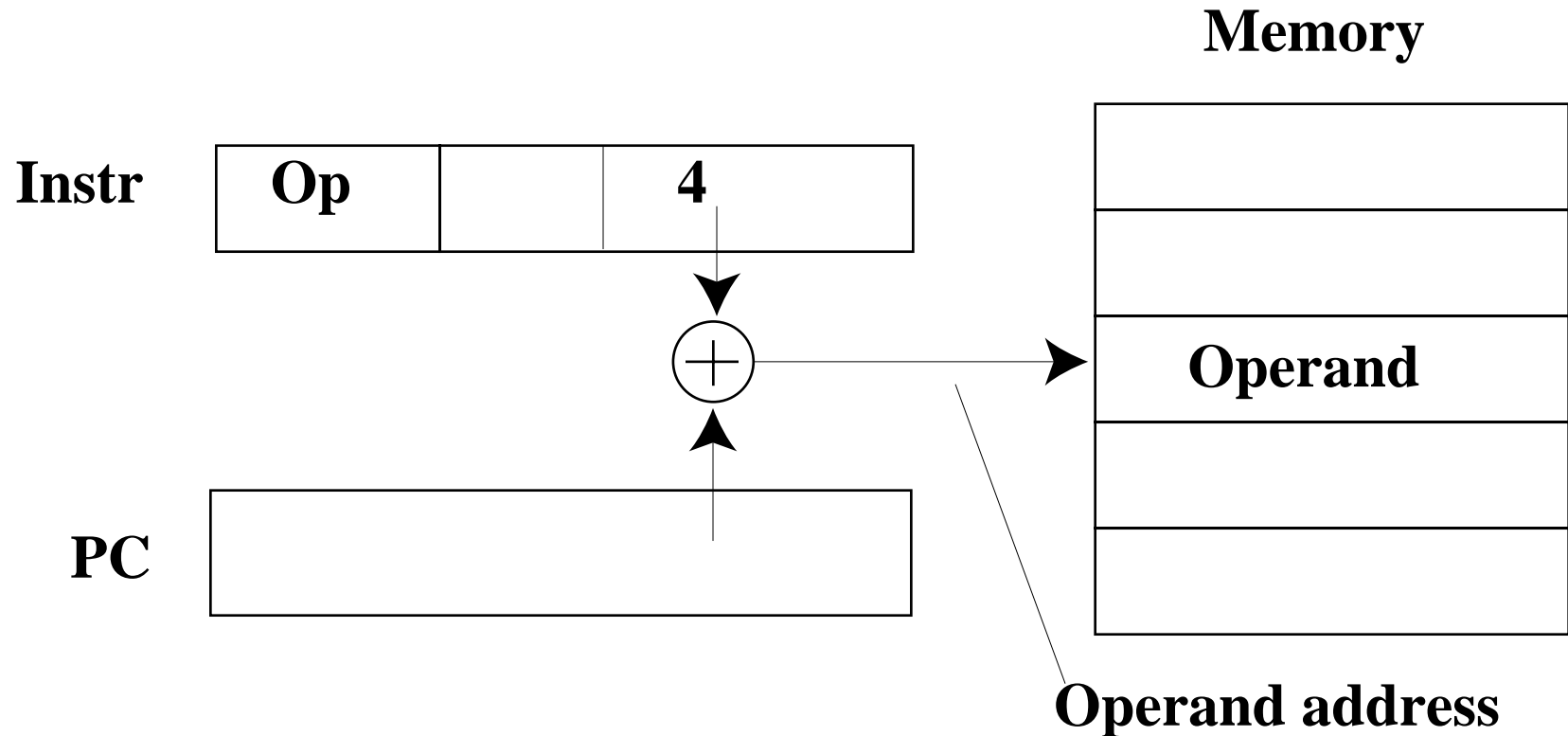
address of operand =
register + constant



load 4[R2],

Relative addressing

$$\text{address of operand} = \text{PC} + \text{constant}$$



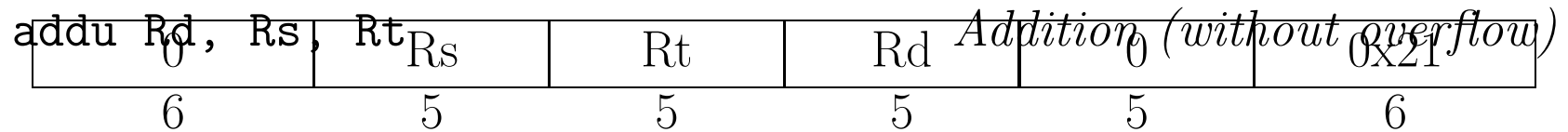
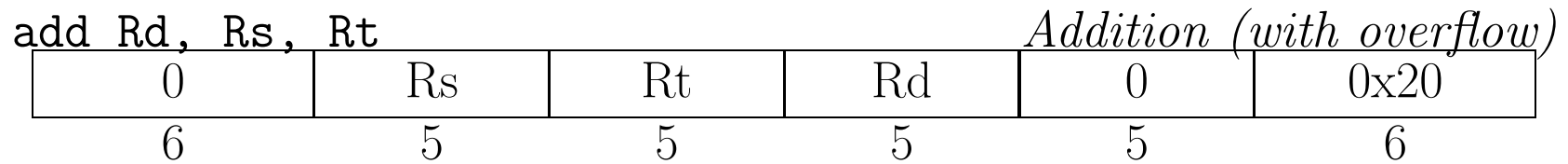
`loadrel 4[PC],`

LOAD-STORE ARCHITECTURES

- All instructions other than **load** and **store** operate only on:
 - ▷ Data in registers
 - ▷ Immediate data (data encoded in the assembled instruction)
- Advantages of RISC load-store architectures:
 - ▷ Speed
 - ALU operations execute in 1 clock period
 - ▷ Small number of addressing modes
 - MIPS processors (RISC): 1 form of add instruction
 - VAX processors (CISC): $\approx 20,000$ forms of add instruction
 - ▷ Simple control unit
 - Short clock period
 - Easily & quickly scalable IC design

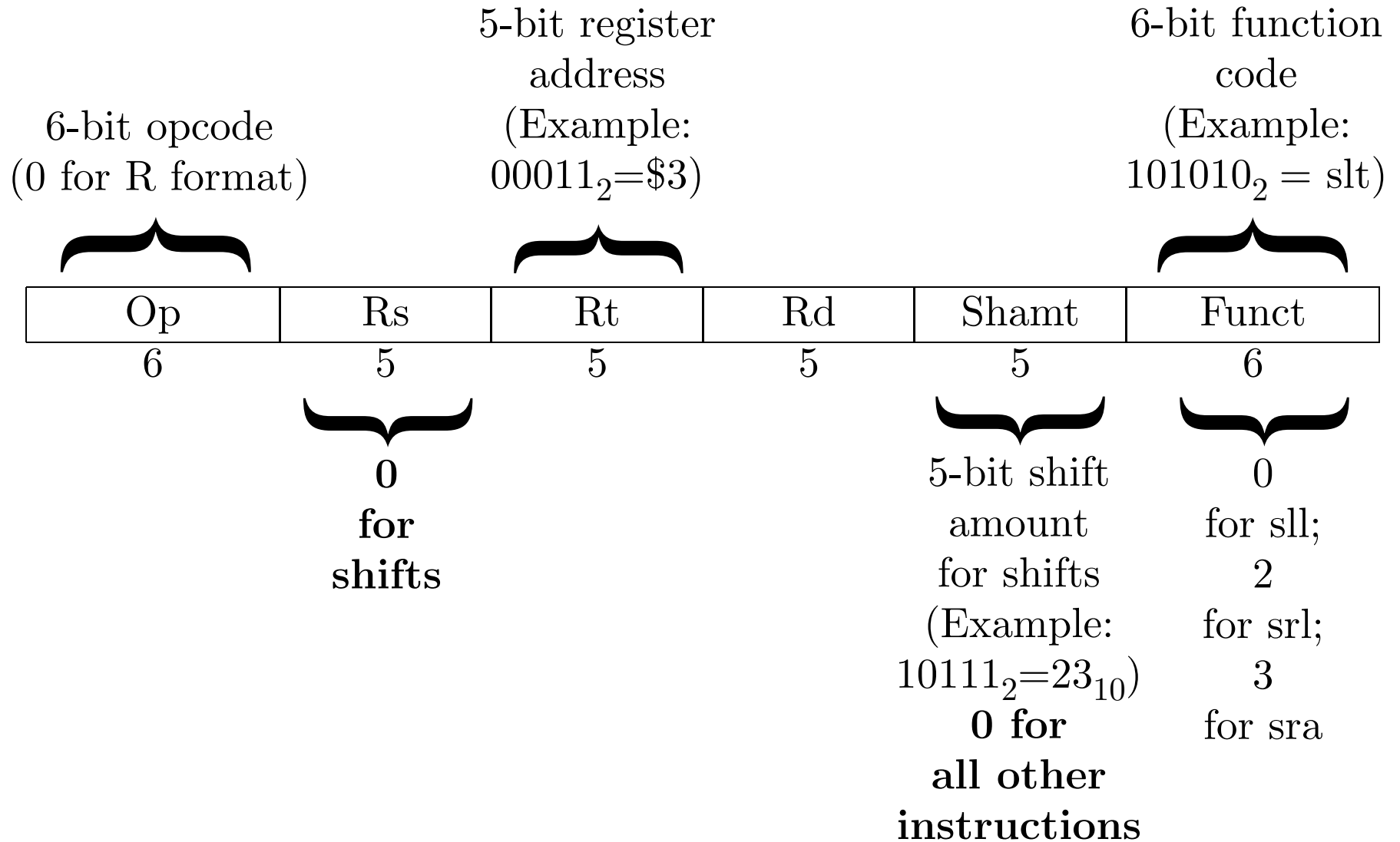
MIPS ADDRESSING MODES (1)

- Register direct addressing: **R-format instructions**



▷ `Rs`, `Rt`, `Rd` are addresses within the register file

R-FORMAT INSTRUCTIONS



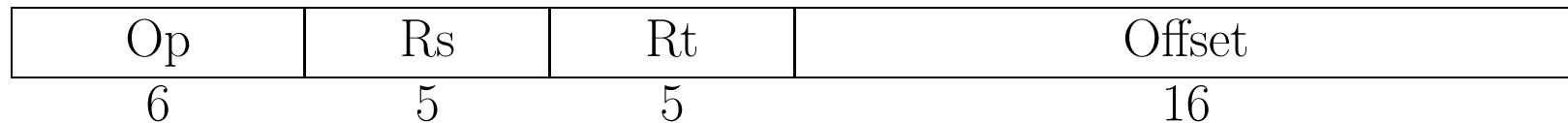
MIPS ADDRESSING MODES (2)

- Load and store instructions move data between main memory and the register file
 - ▷ `lw $Rt, address` copies a 32-bit word from `address` in main memory to register `$Rt`
 - ▷ `sw $Rs, address` copies a 32-bit word from register `$Rs` to `address` in main memory

Syntax for <code>address</code>	Meaning	Used when:
<code>label</code>	Address of <code>label</code>	<code>label</code> is the name of a location in memory
<code>label ± Imm</code>	Address of <code>label</code> ± <code>Imm</code>	Value of <code>Imm</code> is known
<code>(\$Rn)</code>	Contents of <code>\$Rn</code>	Full address is in <code>\$Rn</code>
<code>Imm(\$Rn)</code>	<code>Imm</code> + contents of <code>\$Rn</code>	Value of <code>Imm</code> is known
<code>label ± Imm(\$Rn)</code>	Address of <code>label</code> ± [<code>Imm</code> + contents of <code>\$Rn</code>]	<code>\$Rn</code> contains offset from address of <code>label</code>

MIPS ADDRESSING MODES (3)

- Displacement (indexed) addressing: **I-format instructions**



- ▷ **Rs**, **Rt**, **Rd** are 5-bit addresses within the register file
- ▷ **Offset** is a signed 16-bit integer (two's complement representation)
 - Range of **Offset** is $0x8001 = -32,767_{10}$ to $0x7fff = 32,767_{10}$
 - If **Offset** were interpreted as an unsigned integer address in main memory, there would be only $64k = 65,536$ addressable locations (as in the Intel 8080 and Motorola 6502)

MIPS ADDRESSING MODES (4)

- How to use 32-bit or 64-bit addresses when the instruction length is only 32 bits???



- **Base-offset (indexed) addressing** in the MIPS R2000 processor:

$$\text{Address in memory} = [\text{32-bit address in register } \mathbf{Rs}] + \mathbf{Offset}$$

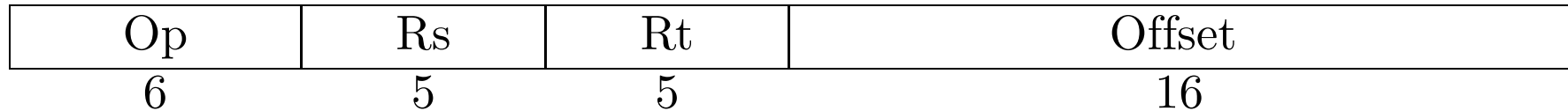
- ▷ Register **Rs** holds a 32-bit **base address**
- ▷ **Offset** is a signed 16-bit integer (two's complement representation)
- ▷ The effective address in main memory can be anywhere in a 64 kB segment centered on the base address

I-FORMAT INSTRUCTIONS

6-bit opcode

Example:

$100011_2 = lw$



5-bit register
address

Example:

$00100_2 = \$4$



16-bit, two's
complement offset

Example:

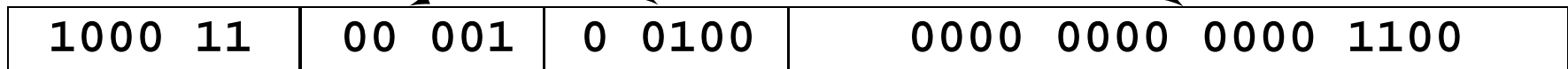
$1111111111110100_2 = -12$

lw \$4 , 12 (\$1)

will hold contents of memory location at address 12 + (\$1)

offset (coded in instruction) (must fit in 16 bits)

holds base address (32 bits)



lw

\$1

\$4

12

Assembled instruction: 0x8c24000c

MIPS REGISTERS (1)

Register	Name or Value	Purpose
PC	Program Counter	Address of next instruction
\$0	Value always 0	Comparisons; address construction
\$1	\$at	Assembler temporary register
\$2	\$v0	Value to be passed to function
\$3	\$v1	Value to be passed to function
\$4	\$a0	First argument
\$5	\$a1	Second argument
\$6	\$a2	Third argument
\$7	\$a3	Fourth argument

MIPS TEMPORARY REGISTERS

- Hold temporary variables in a procedure that calls other procedures
 - ▷ Programming convention: Not preserved across a procedure call

Register	Name	Purpose
\$8	\$t0	Holds a local variable
\$9	\$t1	Holds a local variable
\$10	\$t2	Holds a local variable
\$11	\$t3	Holds a local variable
\$12	\$t4	Holds a local variable
\$13	\$t5	Holds a local variable
\$14	\$t6	Holds a local variable
\$15	\$t7	Holds a local variable
\$24	\$t8	Holds a local variable
\$25	\$t9	Holds a local variable

MIPS SAVED TEMPORARY REGISTERS

- Hold temporary variables used by more than one procedure
 - ▷ Programming convention: Preserved across a procedure call

Register	Name	Purpose
\$16	\$s0	Holds a saved temporary variable
\$17	\$s1	Holds a saved temporary variable
\$18	\$s2	Holds a saved temporary variable
\$19	\$s3	Holds a saved temporary variable
\$20	\$s4	Holds a saved temporary variable
\$21	\$s5	Holds a saved temporary variable
\$22	\$s6	Holds a saved temporary variable
\$23	\$s7	Holds a saved temporary variable

OTHER MIPS GENERAL-PURPOSE REGISTERS

Register	Name	Purpose
\$26	\$k0	Reserved for use by OS kernel
\$27	\$k1	Reserved for use by OS kernel
\$28	\$gp	Pointer to global area
\$29	\$sp	Pointer to first free location on stack
\$30	\$fp	Frame pointer
\$31	\$ra	Return address

MIPS SPECIAL-PURPOSE REGISTERS (1)

- Program counter (PC)
 - ▷ Holds the address of the next instruction to be executed
 - ▷ Modified by branch and jump instructions

MIPS SPECIAL-PURPOSE REGISTERS (2)

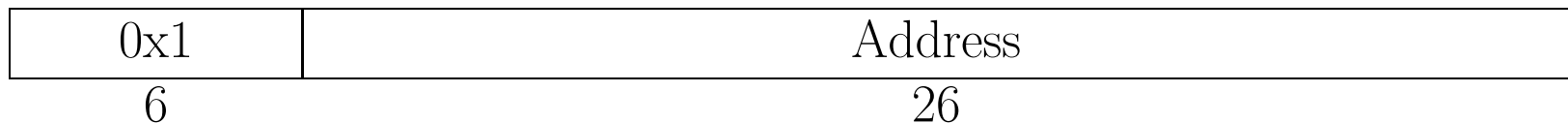
- Floating-point registers (located in coprocessor 1, the FPU)
 - ▷ Used as a local “scratch pad” by the FPU
 - ▷ In the MIPS R2000 ISA, there are 32 32-bit floating-point registers (\$f0–\$f31)
- BadVaddr register (coprocessor 0, register 8)
 - ▷ Memory address at which an addressing exception occurred
- Status register (coprocessor 0, register 12)
 - ▷ Interrupt mask and interrupt enable bits
 - ▷ Kernel/user bits for old, previous and current processes
- Cause register (coprocessor 0, register 13)
 - ▷ Holds a code for the cause of an exception
- Exception program counter (EPC) (coprocessor 0, register 14)
 - ▷ Holds address of instruction that caused an exception

MIPS JUMP and BRANCH INSTRUCTIONS (1)

- **Control transfer instructions** determine program flow by altering the contents of the program counter (PC) register
 - ▷ **Jump register** (`jr Rs`) loads the PC with the contents of **Rs**



- Used to return from function calls
- ▷ **Jump** (`j Address`) loads the PC with the 26-bit contents of **Address**, *multiplied by 4* to give a 28-bit word address in the low 268,435,456 bytes (= 256 MB) of main memory



MIPS JUMP and BRANCH INSTRUCTIONS (2)

- Relative addressing is used to implement conditional transfers
 - ▷ **Branch instructions** are conditional (transfer control only if certain conditions are met, as in “look before you leap”)
 - Used for loop control
 - Example: **branch when equal** (`beq Rs, Rt, Target`)
 - ◇ Branch to instruction at label **Target** is taken when

$\text{contents of } \mathbf{Rs} = \text{contents of } \mathbf{Rt}$
 - ◇ In the assembled instruction,

$$\text{Offset} = \frac{1}{4}(\text{address of Target} - \text{value in PC when beq executes})$$

0x4	Rs	Rt	Offset
6	5	5	16

MIPS JUMP and BRANCH INSTRUCTIONS (3)

- Example of a conditional branch in program `init1d.s`

```
[0x00400028] 0x01890018  mult $12, $9 # top of loop
[0x0040002c] 0x00007012  mflo $14 # offset=index*size
[0x00400030] 0x01c86820  add $13, $14, $8 # addr=base+offset
[0x00400034] 0xadb80000  sw $24, 0($13) # value -> address
[0x00400038] 0x218c0001  addi $12, $12, 1 # index is in $12
[0x0040003c] 0x016c7822  sub $15, $11, $12 # max. array index
                                     # is in $11
[0x00400040] 0x05e1fffa  bgez $15 -24 # loop again if
                                     # index <= maximum
```

▷ Arithmetic of PC-relative addressing:

PC when `bgez` starts to execute contains `0x00400040`; offset is shown as $-24_{10} = -18_{16}$ (**bytes**, not instructions)

In assembled `bgez`, offset is `0xffffa` = -6_{two} **instructions**

Branch target is at address `0x00400028`; let's check the arithmetic:

`0x00400040` (branch) - `0x18` (offset) = `0x00400028` (target) (checks)

SHIFTING (1)

- Let u represent an unsigned, n -bit binary integer,

$$u = b_{n-1} \cdots b_0$$

$$b_i = \text{coefficient of } 2^i$$

- Multiply u by a positive power of 2:

$$2^k u = \underbrace{b_{n-1} \cdots b_{n-k}}_{\text{overflow bits}} \underbrace{b_{n-k-1} \cdots b_0}_{\text{shifted by } k} \underbrace{00 \cdots 0}_{k \text{ zeros}}$$

- ▷ Accomplished by the SPIM instruction

```
sll $rd,$rs,sa
```

where **sa** is the number of places to shift left

\$rs is the source

\$rd is the destination

- ▷ Example: In a jump instruction, a 26-bit offset (in instructions) is shifted left 2 bits to make a byte offset

SHIFTING (2)

- Let u represent an **unsigned**, n -bit binary integer,

$$u = b_{n-1} \cdots b_0$$

$$b_i = \text{coefficient of } 2^i$$

- Multiply u by a negative power of 2 & take the integer part:

$$\lfloor 2^{-k} u \rfloor = \underbrace{00 \cdots 0}_{k \text{ zeros}} \underbrace{b_{n-1} \cdots b_{n-k}}_{\text{shifted by } k}$$

- ▷ Bits $b_{n-k-1} \cdots b_0$ are shifted out
- ▷ Accomplished by the SPIM instruction “shift right logical”:

```
srl $rd,$rs,$sa
```

where **sa** is the number of places to shift right

\$rs is the source

\$rd is the destination

- ▷ The k places vacated on the left are filled with zeros

SHIFTING (3)

- Let the **two's complement**, n -bit binary integer

$$s = b_{n-1}b'_{n-2} \cdots b'_0 = b_{n-1}2^n + i$$

represent the integer

$$i = (-1)^{b_{n-1}} b_{n-2} \cdots b_0$$

- Multiply i by a negative power of 2 & take integer part:

$$\text{result} = i' = \lfloor (-1)^{b_{n-1}} 2^{-k} b_{n-2} \cdots b_0 \rfloor$$

- The two's complement integer that represents i' is

$$s' = b_{n-1}2^n + i'$$

which has k leading 0's if $i > 0$, and k leading 1's if $i < 0$

SHIFTING (4)

- Example of shifting a two's complement representation of an integer when $n = 8$:

$$i = -125_{10} = -7D_{16} = -0111\ 1101$$

The two's complement representative of i is

$$1000\ 0010 + 1 = 1000\ 0011$$

Multiply by 2^{-k} , where $k = 3$:

$$\frac{125}{8} = 15\frac{5}{8} \Rightarrow \left\lfloor -\frac{125}{8} \right\rfloor = -16 = -10_{16} = -0001\ 0000$$

The two's complement representative of -16 is

$$1110\ 1111 + 1 = 1111\ 0000$$

= result of shifting 1000 0011 right by 3 places
and filling the 3 empty places on the left with 1's

SHIFTING (5)

- Let u represent an **two's complement**, n -bit binary integer,

$$u = b_{n-1} \cdots b_0$$

$$b_i = \text{coefficient of } 2^i \text{ for } i \in (0 : n - 2), \quad b_{n-1} = \text{coefficient of } -2^{n-1}$$

- Multiply u by a negative power of 2 & take the integer part:

$$\lfloor 2^{-k} u \rfloor = \underbrace{b_{n-1} b_{n-1} \cdots b_{n-1}}_{k \text{ bits equal to } b_{n-1}} \underbrace{b_{n-1} \cdots b_{n-k}}_{\text{shifted by } k}$$

- ▷ Bits $b_{n-k-1} \cdots b_0$ are shifted out
- ▷ Accomplished by the SPIM instruction “shift right arithmetic”:

```
sra $rd,$rs,sa
```

where **sa** is the number of places to shift right

\$rs is the source

\$rd is the destination

- ▷ The k places vacated on the left are filled with bits equal to b_{n-1}
(**sign extension**)

SHIFTING (6)

- Shift operators in C:

`destination = number shiftop n`

where `number` is an integer, `n` is the number of places to shift (in base 2), and

$$\text{shiftop} = \begin{cases} >>, & \text{if a right shift;} \\ <<, & \text{if a left shift.} \end{cases}$$

- ▷ A C left shift is equivalent to MIPS `sll`
- ▷ A C right shift is equivalent to MIPS `srl`, if `number` is `unsigned` or is positive
- ▷ For other integer or logical data, the result of a C right shift is implementation-dependent
 - May be equivalent to MIPS `sra`

LOGICAL OPERATIONS ON WORDS (1)

- Form of SPIM logical instructions:
 op destination, mask, source
 where op is and, or, nor or xor

- Effects of logical operations on each bit:

mask bit →	0	1
AND	clear	copy
OR	copy	set
NOR	toggle	clear
XOR	copy	toggle

- ▷ **Clear** means “set destination bit equal to 0”
- ▷ **Set** means “set destination bit equal to 1”
- ▷ **Toggle** means “set destination bit equal to 1 if source bit is 0, and set destination bit equal to 0 if source bit is 1” (logical negation)

LOGICAL OPERATIONS ON WORDS (2)

- Example of AND:

$$\begin{array}{r} 1001\ 0011 \text{ (source)} \\ \text{AND } 1111\ 0000 \text{ (mask)} \\ \hline 1001\ 0000 \text{ (destination)} \end{array}$$

The effect is to copy the high nibble and clear the low nibble

- Example of OR:

$$\begin{array}{r} 1001\ 0000 \text{ (source)} \\ \text{OR } 0000\ 0011 \text{ (mask)} \\ \hline 1001\ 0011 \text{ (destination)} \end{array}$$

The effect is to set bits 0 and 1 and copy the remaining bits

LOGICAL OPERATIONS ON WORDS (3)

- Example of NOR:

$$\begin{array}{r} 1001\ 0011 \text{ (source)} \\ \text{NOR } 1111\ 0000 \text{ (mask)} \\ \hline 0000\ 1100 \text{ (destination)} \end{array}$$

The effect is to clear the high nibble and toggle the low nibble

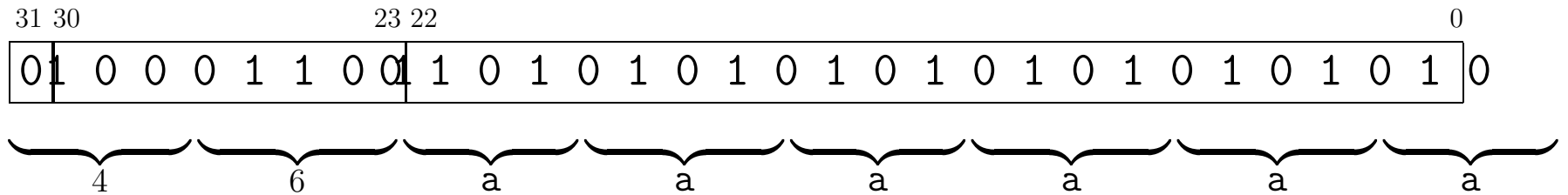
- Example of XOR:

$$\begin{array}{r} 1001\ 0011 \text{ (source)} \\ \text{XOR } 1111\ 0000 \text{ (mask)} \\ \hline 0110\ 0011 \text{ (destination)} \end{array}$$

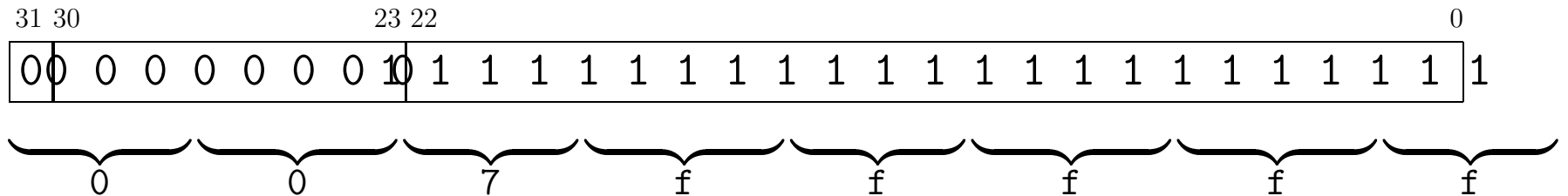
The effect is to toggle the high nibble and copy the low nibble

LOGICAL OPERATIONS ON WORDS (4)

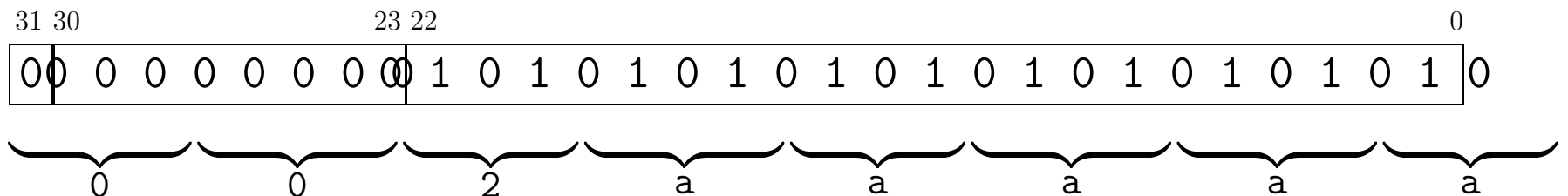
- Use AND to pick out the fraction of a single-precision number:
source:



mask:



destination:



LOGICAL OPERATIONS ON WORDS (5)

- Previous slide:

$0x46aaaaaa \text{ AND } 0x007fffff = 0x002aaaaa$

▷ The effect is to extract the fraction as an unsigned integer (without the implicit 1 bit)

- AND can also be used to extract the sign bit:

$0xc6aaaaaa \text{ AND } 0x80000000 = 0x80000000$

- (By the way — $0x46aaaaaa$ represents 21,845.33, and $0xc6aaaaaa$ represents $-21,845.33$)

LOGICAL OPERATIONS ON WORDS (6)

- C syntax:

`destination = mask op source`

where `op` is:

NOT	\sim
AND	$\&$
OR	$ $
XOR	\wedge

- ▷ Example:

`number = 0x8000 0000 | number`

sets the sign bit of `number`

- `destination = \sim source`
puts the bitwise logical negation of `source` into `destination`

- ▷ Example: `\sim 1` has all bits set except for bit 0