#3 : Data Representation

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 2.1 – 2.4, 2.9, 3.1 – 3.2 and 3.5

Stored Program Concept

Modern computers are built on **two key principles**:

- Instructions are represented as numbers
- Programs are stored in memory to be read or written, just like data

These principles lead to the **stored program concept**:

- No distinction between data and program in memory – memory can contain the source code for a program, the corresponding compiled machine code, the data that the compiled program is using, and even the compiler that generated the machine code
- Programs are shipped as files of binary numbers
- Computers can inherit ready-made software provided they are compatible with an existing instruction set – such compatibility often leads industry to align around a small number of instruction set architectures

Stored Program Concept

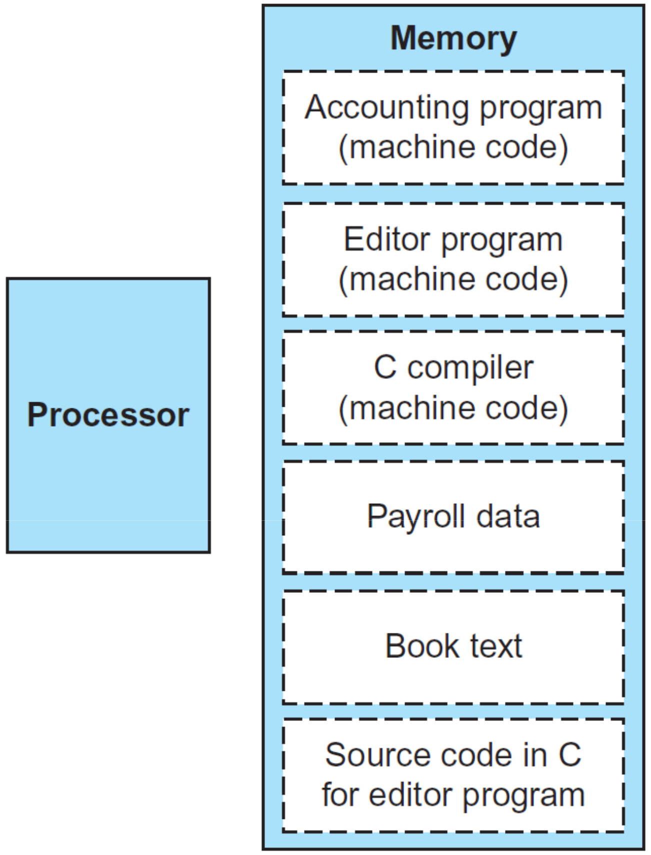

Processor
Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
Source code in C for editor program

FIGURE 2.7 The stored-program concept. Stored programs allow a computer that performs accounting to become, in the blink of an eye, a computer that helps an author write a book. The switch happens simply by loading memory with programs and data and then telling the computer to begin executing at a given location in memory. Treating instructions in the same way as data greatly simplifies both the memory hardware and the software of computer systems. Specifically, the memory technology needed for data can also be used for programs, and programs like compilers, for instance, can translate code written in a notation far more convenient for humans into code that the computer can understand.

Character Data

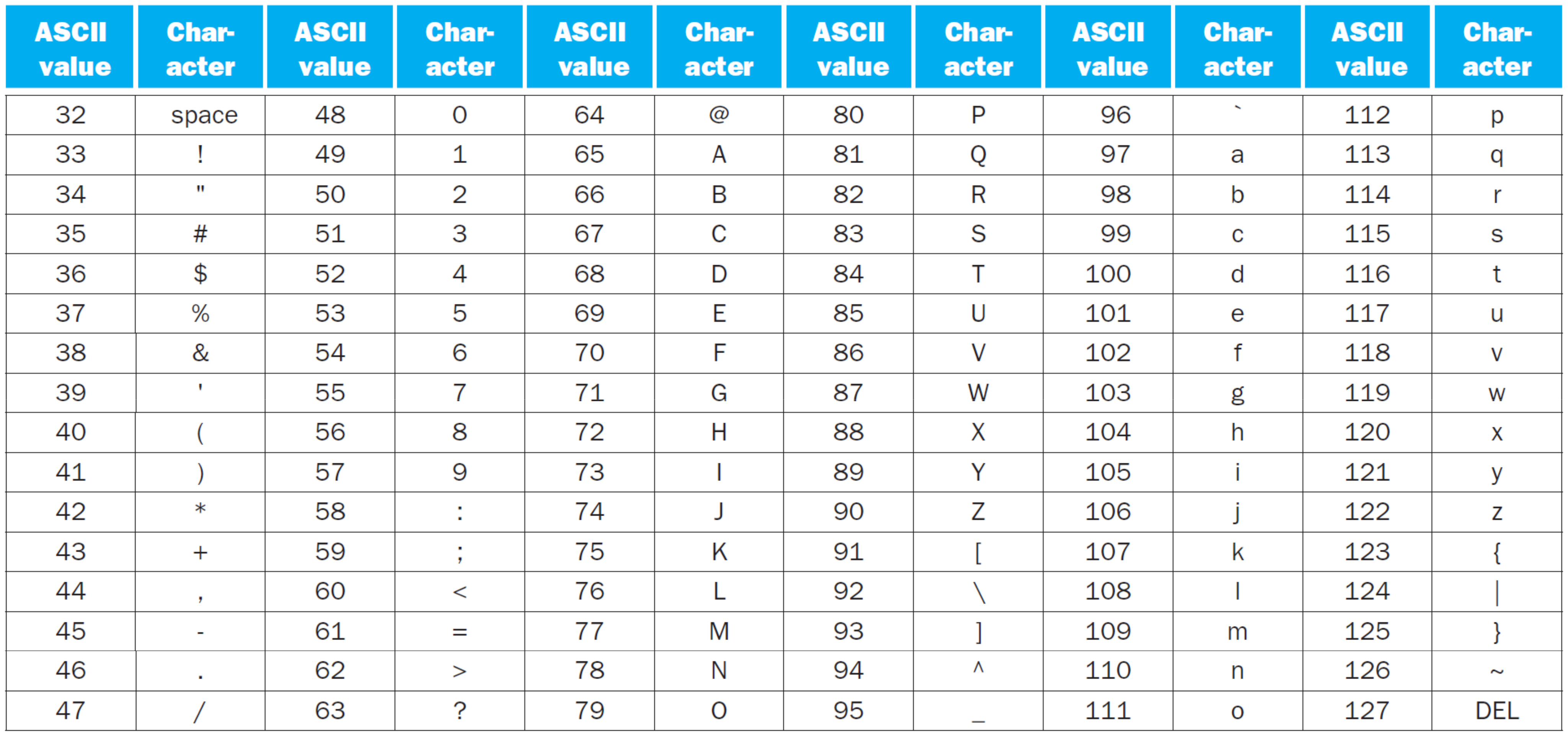
Byte-encoded character sets

- ASCII: 128 characters (95 graphic, 33 control)
- Latin-1: 256 characters (ASCII, +96 more graphic characters)

Unicode 32-bit character set

- Most of the world's alphabets plus symbols
- Variable-length encodings: UTF-8 (one to four 8-bit code units), UTF-16 (one or two 16-bit code units)
- Fixed-length encoding: UTF-32 (32-bit code units)

ASCII



ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

FIGURE 2.15 ASCII representation of characters. Note that upper- and lowercase letters differ by exactly 32; this observation can lead to shortcuts in checking or changing upper- and lowercase. Values not shown include formatting characters. For example, 8 represents a backspace, 9 represents a tab character, and 13 a carriage return. Another useful value is 0 for null, the value the programming language C uses to mark the end of a string. This information is also found in Column 3 of the MIPS Reference Data Card at the front of this book.

UTF-8

UTF-8 was designed for backward compatibility with ASCII. The first 128 characters correspond one-to-one with ASCII, so that valid ASCII text is valid UTF-8 encoded text as well. The next 1920 characters need two bytes and use 11 significant bits (number of Xs in the table below) to cover alphabets like Latin, Greek, Arabic, etc. Three and four bytes are used for more specific alphabets like Chinese, Korean, Japanese, etc.

Number of bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+0000	U+007F	0xxxxxxx			
2	11	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Strings

Characters are normally combined into strings, which have a variable number of characters. There are three choices for representing a string:

- The first position of the string is reserved to give the length of a string
- An accompanying variable has the length of the string (as in a structure)
- The last position of a string is indicated by a specific character used to mark the end of a string

C uses the third choice, terminating a string with a byte whose value is 0 (named null in ASCII).

- For example, the string "AC" is represented by 3 bytes with the decimal numbers <81, 83, 0>

Unsigned Integers N-Bits

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

Range from 0 to $(2^n - 1)$

- Using 32 bits, range from 0 to +4,294,967,295

32 bits example:

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$

$= 1 \times 2^3 + 1 \times 2^1 + 1 \times 2^0$

$= 8 + 2 + 1$

$= 11$

Two's Complement Signed Integers N-Bits

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

Range from (-2^{n-1}) to $(2^{n-1} - 1)$

- Using 32 bits, range from −2,147,483,648 to +2,147,483,647
- Note that $-(-2^{n-1}) = 2^{n-1}$ cannot be represented

32 bits example:

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$

$= -1 \times 2^{31} + 1 \times 2^{30} + \ldots + 1 \times 2^2$

$= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644$

$= -4$

Two’s Complement Signed Integers N-Bits

Most significant (leftmost) bit is the **sign bit**:

- 1 for negative numbers
- 0 for non-negative numbers

Non-negative numbers have the same unsigned and two’s complement representation.

Some specific numbers in 32 bits:

0	$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2$
Most-positive	$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2$
–1	$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2$
Most-negative	$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2$

Two’s Complement Shortcuts

A quick way to negate a two’s complement binary number is to simply invert every 0 to 1 and every 1 to 0, then add one to the result.

$$x + \overline{x} = 1111 \ldots 1111_2 = -1$$

$$\overline{x} + 1 = -x$$

8 bits example:

$+2 = 0000\ 0010_2$

$-2 = (1111\ 1101 + 1)_2 = 1111\ 1110_2$

$+2 = (0000\ 0001 + 1)_2 = 0000\ 0010_2$

Two’s Complement Shortcuts

To convert a two’s complement binary number represented in N bits to a number represented with more than N bits, the shortcut is to take the sign bit from the smaller quantity and replicate it to fill the new bits of the larger quantity. This shortcut is commonly called **sign extension**.

From 8 bits to 16 bits:

$+2 = \mathbf{0}000\ 0010_2 \rightarrow \mathbf{0000\ 0000}\ 0000\ 0010_2$

$-2 = \mathbf{1}111\ 1110_2 \rightarrow \mathbf{1111\ 1111}\ 1111\ 1110_2$

Integer Addition

Addition works as expected, digits are added bit by bit, from right to left, with carries passed to the next left digit, just as is done by hand. Subtraction simply uses addition – the appropriate operands are first negated before being added.

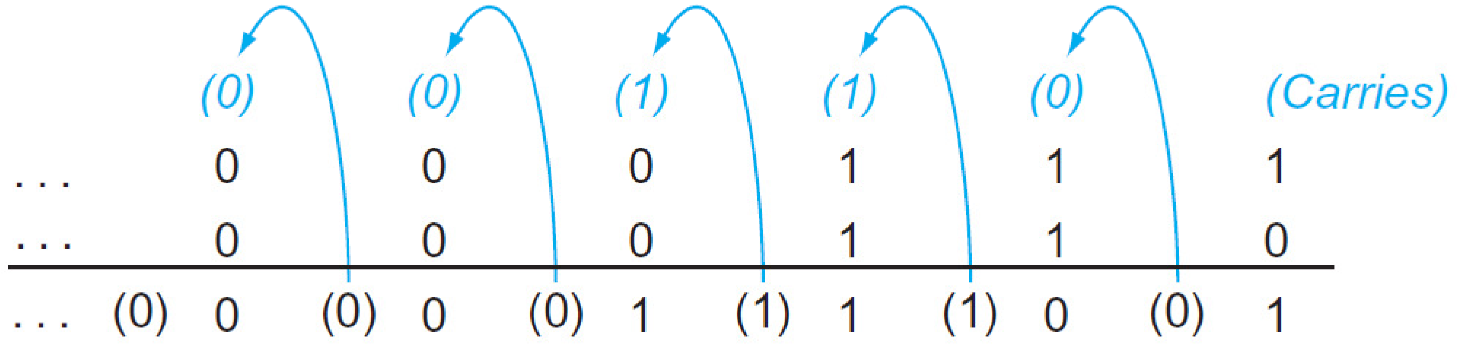


FIGURE 3.1 Binary addition, showing carries from right to left. The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0. Hence, the operation for the second digit to the right is 0 + 1 + 1. This generates a 0 for this sum bit and a carry out of 1. The third digit is the sum of 1 + 1 + 1, resulting in a carry out of 1 and a sum bit of 1. The fourth bit is 1 + 0 + 0, yielding a 1 sum and no carry.

Integer Addition

Overflow occurs when the result cannot be represented with the available hardware.

- Adding operands with different signs, overflow cannot occur
- Adding two positive operands and the sum is negative (sign bit is 1), overflow
- Adding two negative operands and the sum is positive (sign bit is 0), overflow

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

Floating Point Numbers

Scientific notation renders numbers with a single digit to the left of the decimal point:

- $0.315576 \times 10^{10} = 3.15576 \times 10^{9}$ (seconds in a typical century)
- $0.1 \times 10^{-10} = 1.0 \times 10^{-9}$ (seconds in a nanosecond)

Normalized number is a scientific notation number without leading 0s:

- 3.15576×10^{9}
- 1.0×10^{-9}

Computer arithmetic that supports **binary numbers** represented in normalized scientific notation in which the binary point is not fixed (in this case, the leading number is always 1), is called **floating point**.

Floating Point Numbers

$$(-1)^s \times 1.m_2 \times 2^e$$

Floating-point numbers are a **multiple of the size of a word** (32 bits):

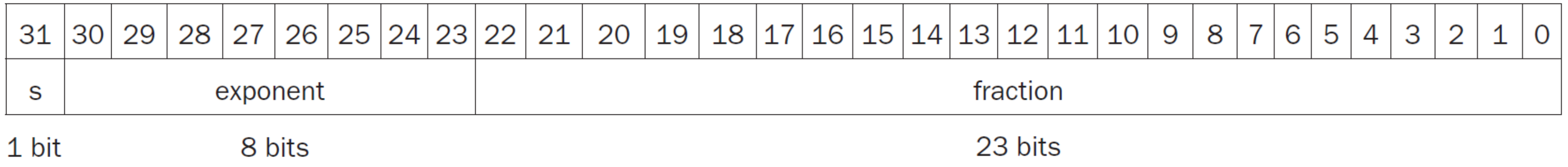
- **Single precision** format requires one word (32 bits)
- **Double precision** format requires two words (64 bits)

A designer of a floating-point representation must find a **compromise between the size of the mantissa (m) and the size of the exponent (e)** since adding a bit to one side requires taking one bit from the other side.

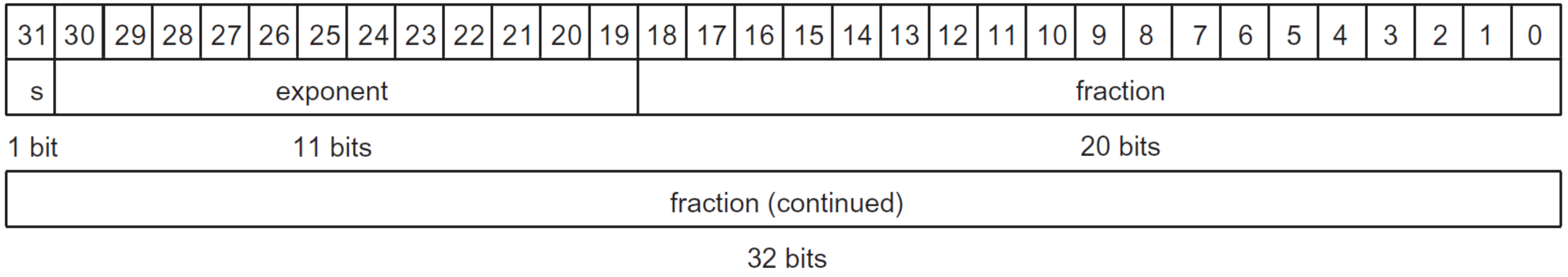
- Increasing the size of the mantissa enhances the **precision of the fraction**
- Increasing the size of the exponent increases the **range of numbers represented**

IEEE 754 Precision Formats

IEEE 754 single precision format



IEEE 754 double precision format



To pack even more bits, IEEE 754 makes the **leading 1-bit implicit**, i.e., a number is actually 24 bits long in single precision (implied 1 plus 23-bit fraction) and 53 bits long in double precision (1+52).

- The term **significand** represents the 24- or 53-bit number (1 plus mantissa)

IEEE 754 Precision Formats

IEEE 754 designers also wanted a representation that could be easily processed by **integer comparison instructions**, especially for sorting.

- This is why **the sign is in the most significant bit**, allowing a quick test of less than, greater than, or equal to 0
- Placing the **exponent before the significand** also simplifies sorting, since numbers with bigger exponents look larger than with smaller exponents
- However, exponents with different signs pose a challenge since the desirable notation must represent the most negative exponent as 00…00 and the most positive as 11…11 – this is called **biased notation**, with the bias being the number subtracted from the normal unsigned representation to determine the real value

IEEE 754 Precision Formats

For example, without biased notation, the number 1.0×2^{-1} would be greater than 1.0×2^{1} if using integer comparison.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

The exponent bias are:

- **127 for single precision**
- **1023 for double precision**

IEEE 754 Precision Formats

$$(-1)^s \times (1+m) \times 2^{e_{10}-bias}$$

For single precision numbers, the range is then from:

- as small as $\pm 1.00000000000000000000000_2 \times 2^{-126}$
- to as large as $\pm\ 1.11111111111111111111111_2 \times 2^{127}$

Single precision example:

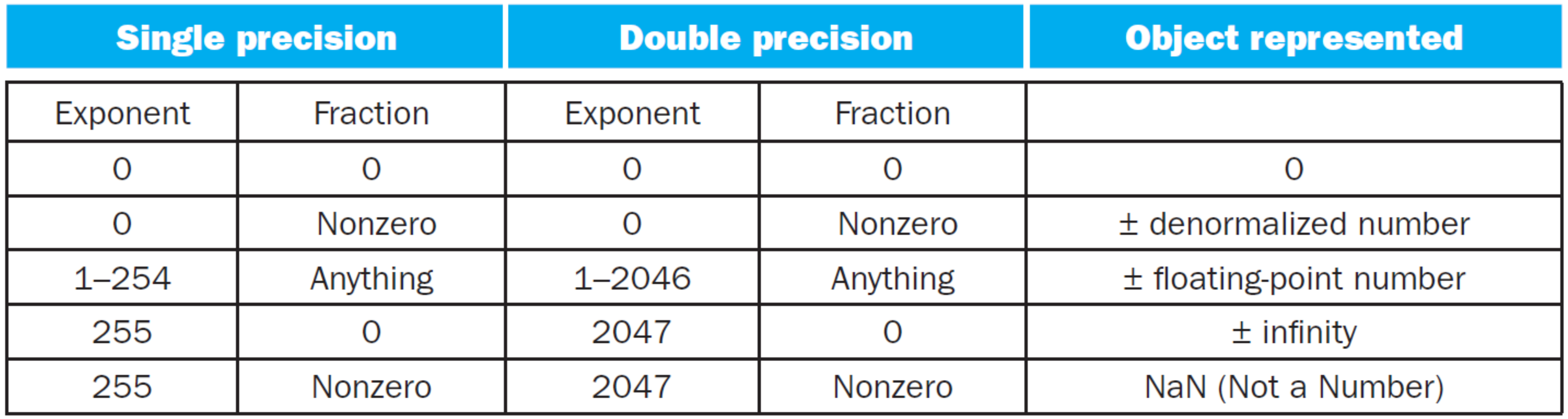
$-0.75 = -(0.50 + 0.25) = -(2^{-1} + 2^{-2}) = -0.11_2 \times 2^0 = -1.1_2 \times\ 2^{-1}$ (normalized)

$= (-1)^1 \times (1_2 + 0.1000\ 0000\ 0000\ 0000\ 0000\ 000_2) \times\ 2^{126-127}$ (single precision)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 8 bits 23 bits

IEEE 754 Precision Formats



Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

FIGURE 3.13 EEE 754 encoding of floating-point numbers. A separate sign bit determines the sign. Denormalized numbers are described in the *Elaboration* on page 222. This information is also found in Column 4 of the MIPS Reference Data Card at the front of this book.

Special cases

- Divide by 0 is represented as **infinity**
- Invalid operations, such as 0/0 or subtracting infinity from infinity, is represented as **NaN (Not a Number)**

Denormalized Numbers

$$(-1)^s \times m \times 2^{1-bias}$$

Rather than having a gap between 0 and the smallest normalized number, IEEE allows **denormalized numbers**. They allow a number to degrade in significance until it becomes 0. They have the **same exponent as zero but a nonzero mantissa**.

For example, the smallest positive single precision normalized number is:

- $1.00000000000000000000000_2 \times 2^{-126} = 2^{-126}$

But the smallest single precision denormalized number is:

- $0.00000000000000000000001_2 \times 2^{-126} = 2^{-149}$

For double precision, the denorm gap goes from 2^{-1022} to 2^{-1074}.

Floating Point Addition

Assume a decimal notation with **4 decimal digits of the significand** and **2 decimal digits of the exponent** and the following two numbers:

- 99.99 and 0.1610 ($9.999 \times 10^1 + 1.610 \times 10^{-1} =$ **100.151**)

The steps to add the two numbers using floating point addition are:

- $= 9.999 \times 10^1 +$ **0.01610×10^1** (align decimal point of smaller exponent)
- $= 9.999 \times 10^1 +$ **0.016** $\times 10^1$ (round number to four decimal digits)
- $=$ **10.015** $\times 10^1$ (sum significands)
- $=$ **1.0015×10^2** (normalize and adjust exponent)
- $= 1.0015 \times 10^2$ (check for overflow or underflow)
- $=$ **1.002** $\times 10^2$ (round number to four decimal digits)
- $= 1.002 \times 10^2$ (check for normalization and repeat steps if necessary)
- $=$ **100.2**

Floating Point Addition

Assume now a binary notation with **4 bits of the significand** and **8 bits of the exponent** and the following two numbers:

- 0.5 and -0.4375 ($0.5 - 0.4375 =$ **0.0625**)
- $0.5 = 0.1_2 \times 2^0 = \mathbf{1.000_2 \times 2^{-1}}$
- $-0.4375 = -(0.25 + 0.125 + 0.0625) = -0.0111_2 \times 2^0 = \mathbf{-1.110_2 \times 2^{-2}}$

The steps to add the two numbers using floating point addition are:

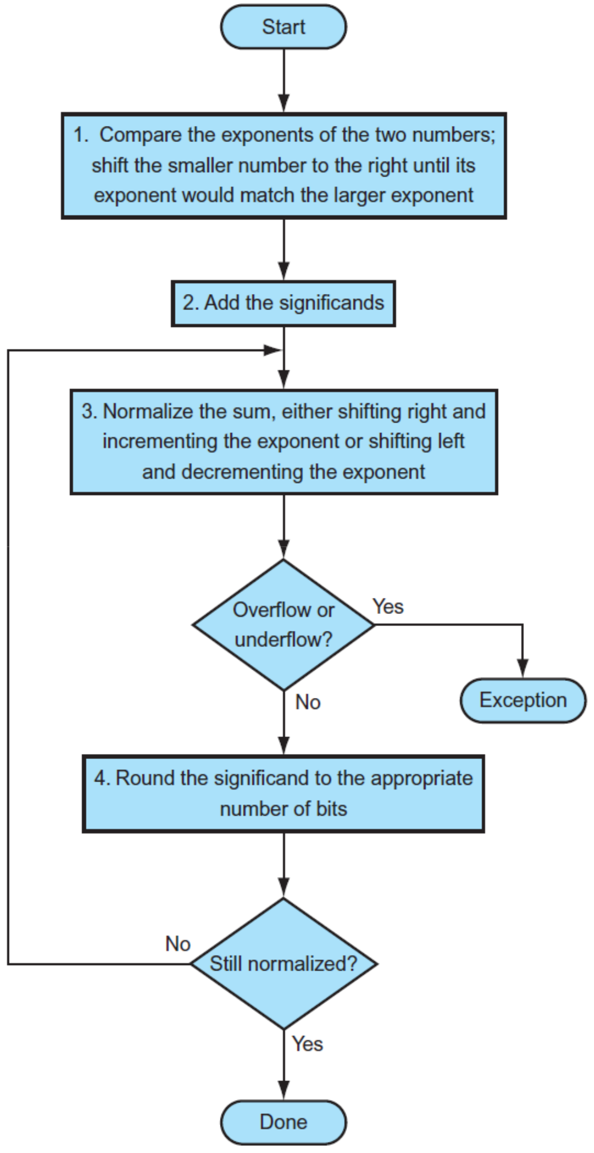
- $= 1.000_2 \times 2^{-1} - \mathbf{0.111_2 \times 2^{-1}}$ (align decimal point of smaller exponent)
- $= \mathbf{0.001_2} \times 2^{-1}$ (sum significands)
- $= \mathbf{1.000_2 \times 2^{-4}}$ (normalize and adjust exponent)
- $= 1.000_2 \times 2^{-4}$ (check for overflow or underflow)
- $= \mathbf{1.000_2} \times 2^{-4}$ (round number to four bits)
- $= 1.000_2 \times 2^{-4}$ (check for normalization and repeat steps if necessary)
- $=$ **0.0625**

Floating Point Addition

Overflow/Underflow occurs when a **positive/negative exponent** becomes too large to fit in the exponent field.

Rounding sounds simple enough, but to round accurately requires the hardware to include **extra bits** in the calculation.

- If every intermediate result had to be truncated to the exact number of digits, there would be no opportunity to round
- To improve rounding accuracy, IEEE 754 keeps extra bits on the right during intermediate additions



Start
1. Compare the exponents of the two numbers; shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
Still normalized?
No
Yes
Done

FIGURE 3.14 Floating-point addition. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

Rounding with IEEE 754

One possible algorithm is **round to the nearest even**, which always creates a 0 in the least significant bit in the tie-breaking case.

- The two first extra bits kept on the right during intermediate calculations of floating-point numbers are called **guard and round bits**
- A third bit, named **sticky bit**, is set whenever there are nonzero bits to the right of the round bits

Rounding rules:

- $m0x \rightarrow m$ (round down)
- $m11 \rightarrow m + 0\ldots01$ (round up)
- $m10 \wedge sticky{=}1 \rightarrow m + 0\ldots01$ (round up)
- $m10 \wedge sticky{=}0 \rightarrow m + 0\ldots01$ (round up if least sigficant bit of m is 1)
- $m10 \wedge sticky{=}0 \rightarrow m$ (round down if least sigficant bit of m is 0)

MIPS Registers

Unlike programs in high-level languages, in MIPS the operands of arithmetic instructions are **restricted** – they must be from a limited number of special locations built directly in hardware called **registers**.

One major difference between the variables of a programming language and registers is the **limited number** of registers, typically 32 on current computers like MIPS.

Memory

Memory is just a large, single-dimensional array, with the **address acting as the index to that array**, starting at 0.

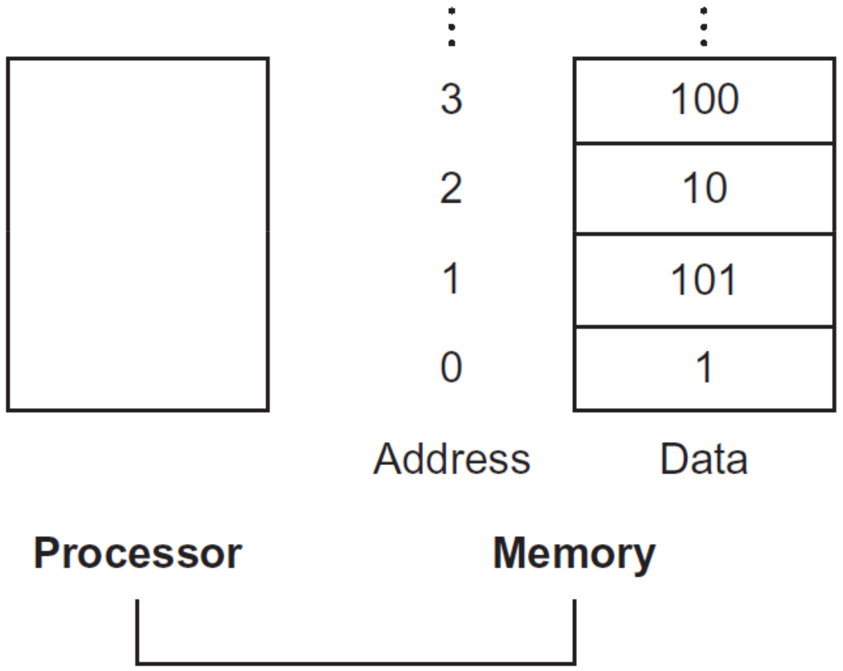


FIGURE 2.2 Memory addresses and contents of memory at those locations. If these elements were words, these addresses would be incorrect, since MIPS actually uses byte addressing, with each word representing four bytes. Figure 2.3 shows the memory addressing for sequential word addresses.

Memory

To access a word in memory, **instructions must supply the memory address** in multiples of 4 bytes (32 bits).

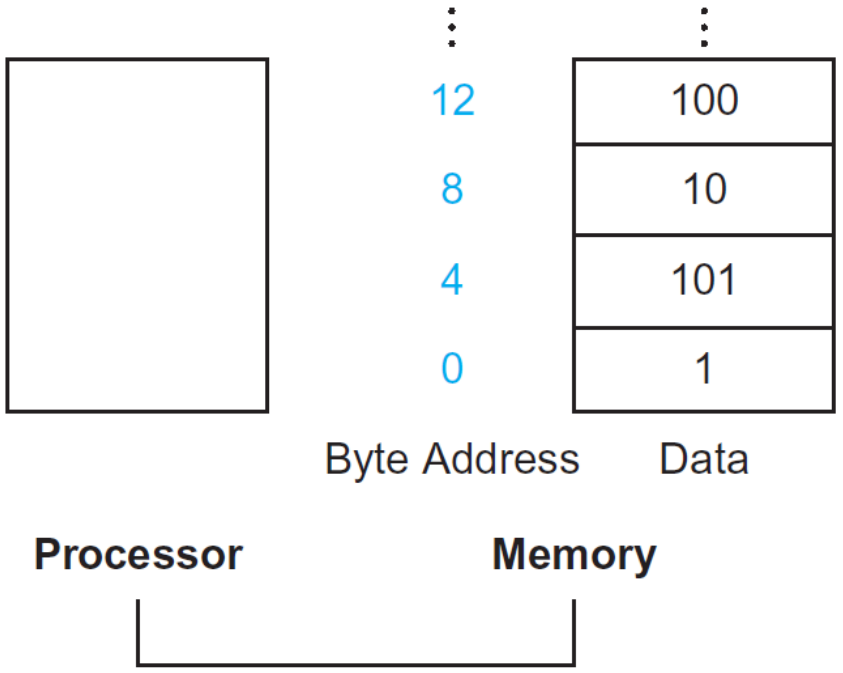


FIGURE 2.3 Actual MIPS memory addresses and contents of memory for those words. The changed addresses are highlighted to contrast with Figure 2.2. Since MIPS addresses each byte, word addresses are multiples of 4: there are 4 bytes in a word.

Registers and Memory in MIPS

Billions of data elements are kept in memory, but the processor can keep only a **small amount of data in registers**.

In MIPS, arithmetic operations occur only on registers, thus MIPS must include instructions that transfer data between memory and registers. Such instructions are called **data transfer instructions**.

The data transfer instruction that copies data from memory to a register is traditionally called a **load instruction**. The instruction complementary to load is traditionally called a **store instruction** and it copies data from a register to memory.

Registers and Memory in MIPS

Registers take **less time to access** and have **higher throughput** than memory, making data in registers faster to access and simpler to use:

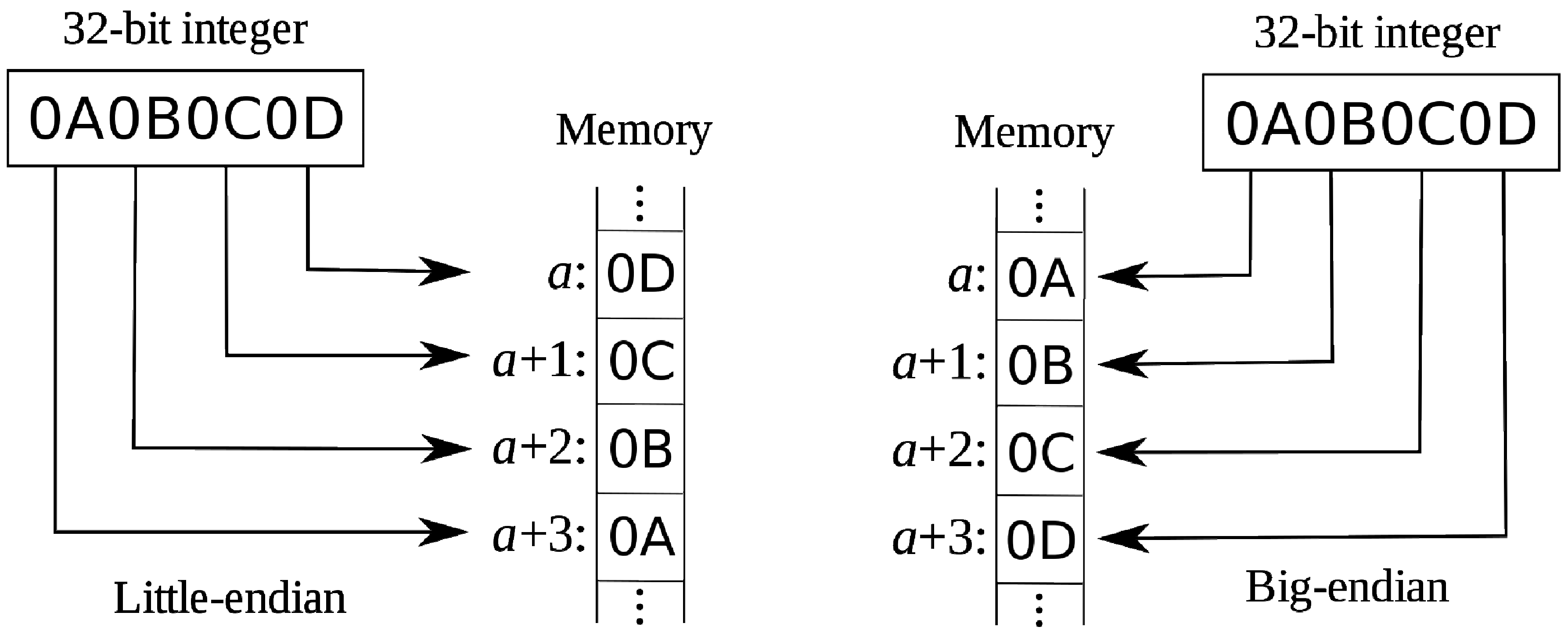
- MIPS arithmetic instructions can read two registers, operate on them, and write the result to another register
- A MIPS data transfer instruction only reads one operand or writes one operand, without operating on it
- Operating on memory data requires loads and stores to/from registers, i.e., more instructions to be executed

Compilers must **use registers efficiently**:

- Use registers for variables as much as possible
- Only spill to memory for less frequently used variables
- Register optimization is important

Endianness

Endianness is the order of bytes in the hardware. Processors can number bytes within a word so the byte with the lowest number is either the rightmost (**little-endian**) or leftmost (**big-endian**). MIPS is **little-endian**.



32-bit integer
0A0B0C0D
Memory
a: 0D
a+1: 0C
a+2: 0B
a+3: 0A
Little-endian
32-bit integer
0A0B0C0D
Memory
a: 0A
a+1: 0B
a+2: 0C
a+3: 0D
Big-endian