

Integers are represented as n-bit vectors (series of 1's and 0's)

$$B = b_{n-1} \dots b_1 b_0$$

The unsigned value represented is:

$$V(B) = b_{n-1} \times 2^{n-1} + \dots + b_1 \times 2^1 + b_0 \times 2^0$$

The  $b$ 's are the coefficients in a powers of 2 polynomial

The number of bits ( $n$ ) determines the range that can be covered

$$0 \text{ to } 2^n - 1$$

Unsigned overflow exists if an operation yields a value out of range

The total number of patterns =  $2^n$  where  $n$  is the number of bits  
This is called the modulus of the system  
Incrementing the largest pattern (all 1's) rolls over to 0

Signed systems use some of the patterns for negative values

- Sign-and-magnitude
- One's complement
- Two's complement
- Biased (or excess)

Two's complement is by far the most common

Biased is used for integer exponents in floating point numbers

The first three systems are compared below (using 4 bits):

$b_3 b_2 b_1 b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+7	+7	+7
0 1 1 0	+6	+6	+6
0 1 0 1	+5	+5	+5
0 1 0 0	+4	+4	+4
0 0 1 1	+3	+3	+3
0 0 1 0	+2	+2	+2
0 0 0 1	+1	+1	+1
0 0 0 0	+0	+0	+0
1 0 0 0	-0	-7	-8
1 0 0 1	-1	-6	-7
1 0 1 0	-2	-5	-6
1 0 1 1	-3	-4	-5
1 1 0 0	-4	-3	-4
1 1 0 1	-5	-2	-3
1 1 1 0	-6	-1	-2
1 1 1 1	-7	-0	-1

$b_3 b_2 b_1 b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+7	+7	+7
0 1 1 0	+6	+6	+6
0 1 0 1	+5	+5	+5
0 1 0 0	+4	+4	+4
0 0 1 1	+3	+3	+3
0 0 1 0	+2	+2	+2
0 0 0 1	+1	+1	+1
0 0 0 0	+0	+0	+0
1 0 0 0	-0	-7	-8
1 0 0 1	-1	-6	-7
1 0 1 0	-2	-5	-6
1 0 1 1	-3	-4	-5
1 1 0 0	-4	-3	-4
1 1 0 1	-5	-2	-3
1 1 1 0	-6	-1	-2
1 1 1 1	-7	-0	-1

In each system, the MSB is 0 for positive and 1 for negative values  
Positive values have identical representations for each system

- Same as for the unsigned value

They differ in how negative values are represented

- Only negative values require complementing

## Sign-and-magnitude:

MSB only indicates sign (0 for +, 1 for -)

Remaining bits give the magnitude

## One's complement

Invert each bit to get the negative

Same as adding negative value to modulus-1

e.g. -5 is represented as  $-5 + (16-1) = 10$

## Two's complement

Invert each bit and add 1 to get negative

Same as adding negative value to modulus

e.g. -5 represented as  $-5 + 16 = 11$

$b_3 b_2 b_1 b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+7	+7	+7
0 1 1 0	+6	+6	+6
0 1 0 1	+5	+5	+5
0 1 0 0	+4	+4	+4
0 0 1 1	+3	+3	+3
0 0 1 0	+2	+2	+2
0 0 0 1	+1	+1	+1
0 0 0 0	+0	+0	+0
1 0 0 0	-0	-7	-8
1 0 0 1	-1	-6	-7
1 0 1 0	-2	-5	-6
1 0 1 1	-3	-4	-5
1 1 0 0	-4	-3	-4
1 1 0 1	-5	-2	-3
1 1 1 0	-6	-1	-2
1 1 1 1	-7	-0	-1

## Sign-and-magnitude properties:

two distinct representations for 0

+0 → 0 sign bit and all 0's for magnitude

-0 → 1 sign bit and all 0's for magnitude

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

## One's complement:

two distinct representations for 0

+0 → bits are all 0's

-0 → bits all 1's

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

## Two's complement:

Unique representation of 0 (all 0 bits)

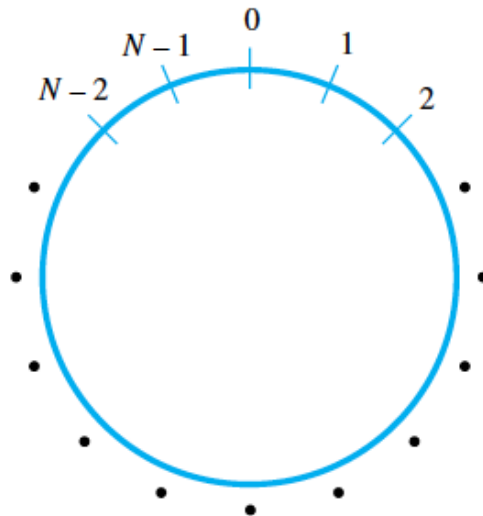
Range =  $-2^{n-1}$  to  $+2^{n-1}-1$

Extra negative pattern ( $-2^{n-1}$ ) has MSB=1, all other bits = 0

We will review the algorithms that the ALU must implement

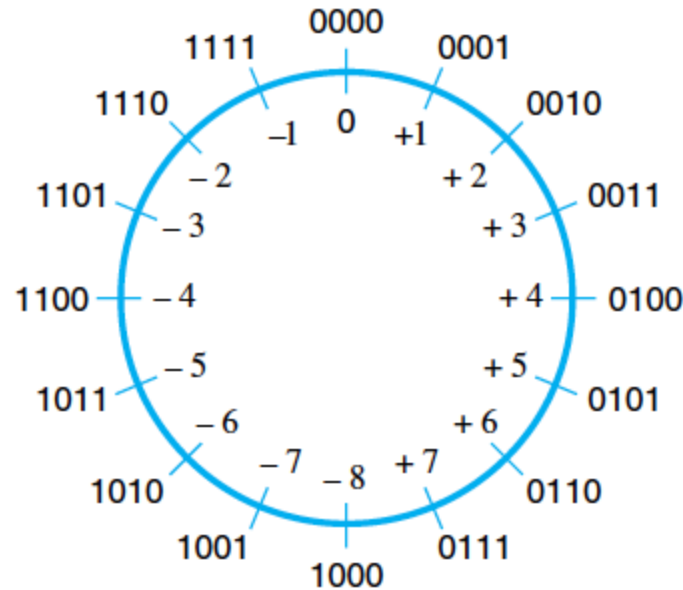
- Addition & subtraction
- Multiplication & division

We will also show why two's complement is preferred



An incremented unsigned integer rolls over from the max value back to 0. (modulo  $N$ )

Stepping counter-clockwise  
from 0 goes to -1



Stepping clockwise from 0 goes  
to +1

In this 4-bit system there are 16  
possible values (modulus=16)

To add  $M$  to a value, perform  $M$  clockwise steps

To subtract  $M$  from a value, perform  $M$  counter-clockwise steps  
this is equivalent to  $16-M$  clockwise steps  
the same as adding the two's complement of  $-M$



Using hex makes dealing with more bits easier

Each hex digit (0,...,9,A,B,C,D,E,F) corresponds to a group of 4 bits

Example: assume 16-bit numbers:

$$\begin{array}{r} -9 \\ +22 \\ \hline 13 \end{array} \quad \longrightarrow \quad \begin{array}{r} 0xFFF7 \\ +0x0016 \\ \hline 0x000D \end{array}$$

$$\text{modulus} = 2^{16} = 65536 = 0x10000$$

$$-9 \text{ in two's complement} = 65536 - 9 = 65527 = 0xFFF7$$

$$\begin{array}{r} 350 \\ +922 \\ \hline 1272 \end{array} \quad \longrightarrow \quad \begin{array}{r} 0x015E \\ +0x039A \\ \hline 0x04F8 \end{array}$$

In two's complement addition:

- add the two bit patterns

- ignore any carry out of the leftmost bit


Subtract by adding the two's complement of the subtrahend

Only negative values need to be complemented

Two's complement is preferred over one's complement

- One's complement has both +0 and -0

- One's complement addition requires an end-around carry

- 9		0xFFF6	modulus-1 = $2^{16} - 1 = 65535 = 0xFFFF$
+22		+0x0016	
<u>13</u>		0x000C	

The carry must be added into the result to obtain  $0x000C + 1 = 0x000D = 13$

Using sign and magnitude representation complicates arithmetic  
extra operations are required (sign checking & absolute value)

For addition or subtraction:

- check the signs of the two numbers

- if they differ, subtract the smaller number from the larger

- use the sign of the larger as the sign for the result

For multiplication or division:

- check the signs of the two numbers

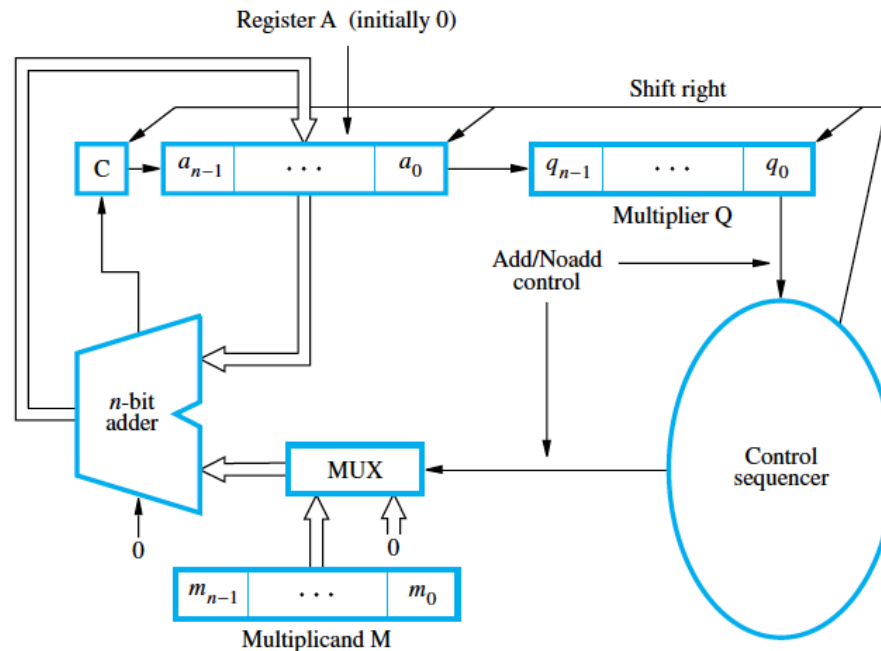
- if they differ, the result is negative

- compute using the absolute values of the two numbers

So two's complement representation and arithmetic is preferred

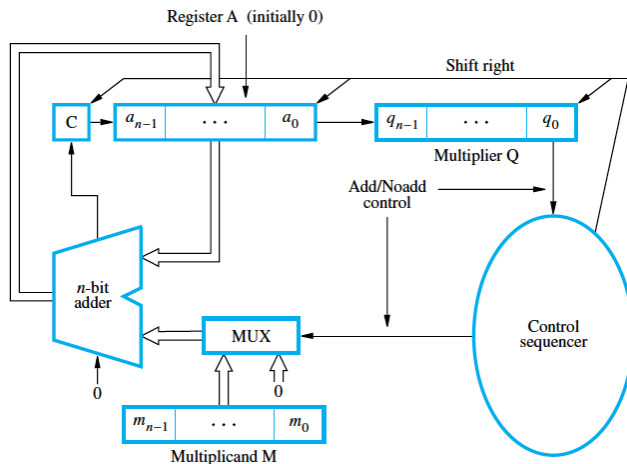
To multiply by  $2^n$ , just shift left  $n$  bits

In general, multiplication involves shifting and adding



The product of two  $n$ -bit numbers requires  $2n$  bits

Configuration for unsigned multiplication



4-bit numbers  $(-13 \times 11)$  unsigned multiply

		M		
		1 1 0 1		
		0 0 0 0	1 0 1 1	Initial configuration
0	C	A	Q	
0		1 1 0 1	1 0 1 1	Add Shift } First cycle
0		0 1 1 0	1 1 0 1	
1		0 0 1 1	1 1 0 1	Add Shift } Second cycle
0		1 0 0 1	1 1 1 0	
0		1 0 0 1	1 1 1 0	No add Shift } Third cycle
0		0 1 0 0	1 1 1 1	
1		0 0 0 1	1 1 1 1	Add Shift } Fourth cycle
0		1 0 0 0	1 1 1 1	
		Product		

Repeat N times (N= 4 here)

Add multiplicand to the A register only if the LSB of Q is 1

Shift for each cycle (C, A and Q together)

Generates an 8-bit product

Compute  $13 \times 11$  and negate the result to get -143



## A more general technique

Works for both positive and negative factors

Reduces the number of operations required

Example: suppose we want to multiply by  $30 = 0011110_2$   
30 can be regarded as

$$\begin{array}{r} 0100000 \quad (32) \\ - 0000010 \quad (2) \\ \hline 0011110 \quad (30) \end{array} \quad \text{i.e., multiply by } (32 - 2)$$

Add 32 x multiplicand plus -2 x multiplicand

Multiplier		Version of multiplicand selected by bit $i$
Bit $i$	Bit $i - 1$	
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$

Scan multiplier bits from right to left

(assume initial 0 bit to right of LSB)

subtract multiplicand when moving from 0 to 1

add multiplicand when moving from 1 to 0

neither add nor subtract when moving from 0 to 0 or 1 to 1

shift multiplicand left one bit for each cycle



The original multiply is “*recoded*”

$30 = 0011110_2$  using 7 bits

0 0 1 1 1 1 1 0



0 +1 0 0 0 0 -1 0

Subtract multiplicand for each -1 in recoded multiplier

Add multiplicand for each +1 in recoded multiplier

Neither add nor subtract for each 0 in recoded multiplier




45 → 0101101

-45 → 1010011

								0	1	0	1	1	0	1	← multiplicand
								0	+1	0	0	0	-1	0	← Recoded multiplier
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	1	1	1	1	1	1	0	1	0	0	1	1			← 2's complement of the multiplicand
0	0	0	0	0	0	0	0	0	0	0	0	0			Sign extended
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	1	1	0	1							
0	0	0	0	0	0	0	0								
0	0	0	1	0	1	0	1	0	0	0	1	1	0	→	0x0546 = 1350 = 45 x 30

One addition and one subtraction is required to generate the product  
 Using 00111110 would have required 5 additions

Assume a 17-bit multiplier:

0	0	1	0	1	1	0	0	1	1	1	0	1	0	1	1	0	0	
																		
0	+1	-1	+1	0	-1	0	+1	0	0	-1	+1	-1	+1	0	-1	0	0	



Longhand division in binary is similar to that in decimal

$$\begin{array}{r} 21 \\ 13 \overline{)274} \\ \underline{26} \phantom{0} \\ 14 \\ \underline{13} \\ 1 \end{array}$$

$$\begin{array}{r} 10101 \\ 1101 \overline{)100010010} \\ \underline{1101} \phantom{0000} \\ 10000 \\ \underline{1101} \\ 1110 \\ \underline{1101} \\ 1 \end{array}$$

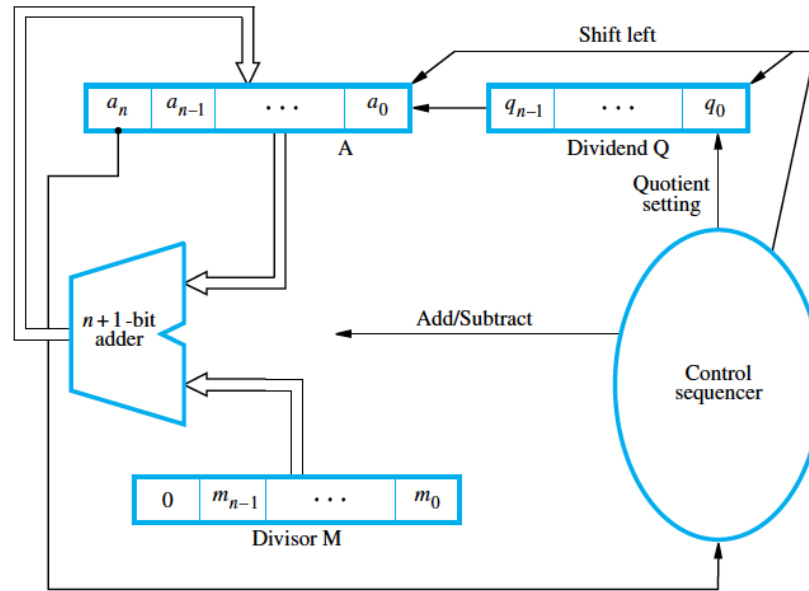
Both a quotient and a remainder are produced

Handle signed numbers by dividing the positive equivalents

Sign of remainder = sign of dividend (dividend rule)

Quotient is negative if the divisor and dividend differ in sign

Set A to 0  
Put divisor in M  
Put dividend in Q

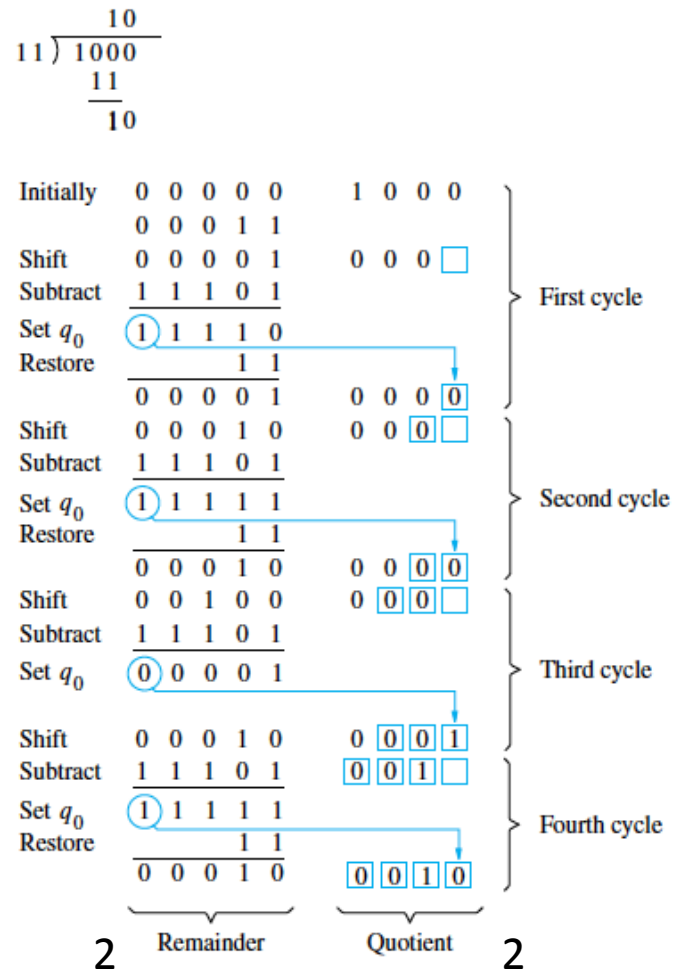


Repeat the following 3 steps  $n$  times:

1. Shift A and Q left one bit position
2. Subtract M from A
3. If A is negative, set  $q_0$  to 0 and add M back to A (i.e. restore A); otherwise set  $q_0$  to 1

When done, Q contains quotient and A contains the remainder

Assume 4-bit values. Division of 8 by 3



Avoids having to add M back when subtracting makes  $A < 0$

Restoring division computes  $A - M$  and if negative adds A back before shifting left 1 bit for the next cycle

Shifting  $A - M$  left first and then adding M gives:

$$2(A - M) + M = 2A - M \text{ (which is needed in the next cycle)}$$

This avoids the restore step

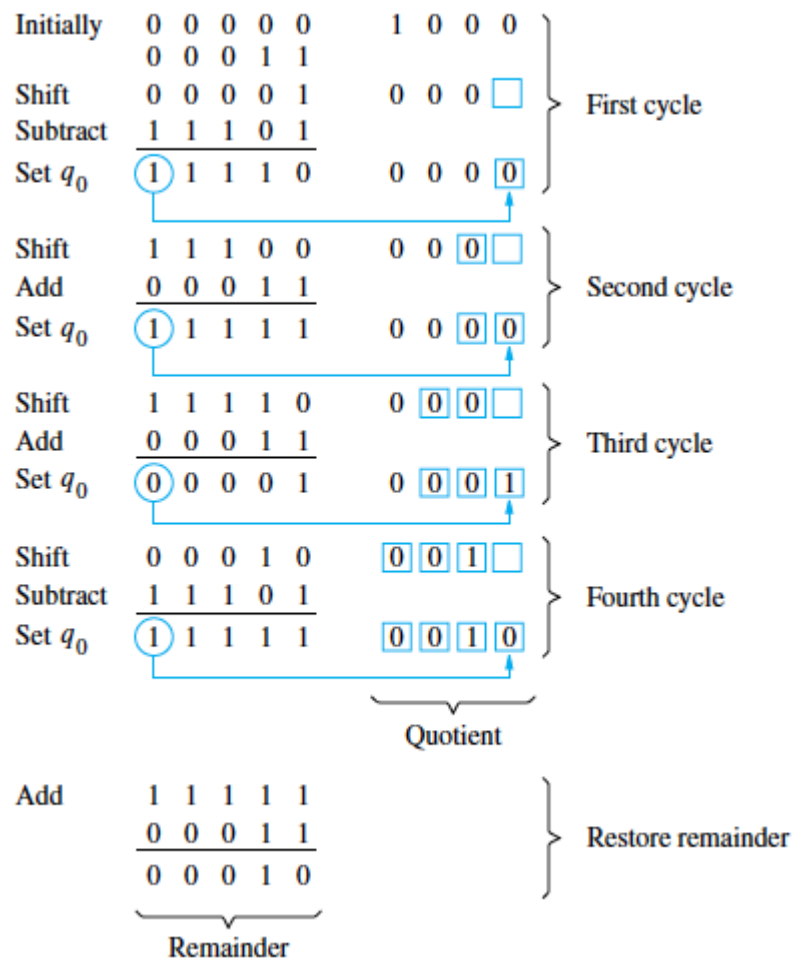
Stage 1: Repeat the following 2 steps  $n$  times:

1. If  $A < 0$ , shift  $A$  and  $Q$  left 1 bit and add  $M$  to  $A$   
Else shift  $A$  and  $Q$  left 1 bit and subtract  $M$  from  $A$
2. If  $A < 0$ , set  $q_0 = 0$  else set  $q_0 = 1$

Stage 2: If  $A < 0$ , add  $M$  to  $A$

Stage 2 is needed to leave the proper positive remainder in  $A$

Assume 4-bit values. Division of 8 by 3





Use absolute values of the dividend and divisor

Quotient is negative if the signs of divisor & dividend differ

Sign of remainder should match the sign of the dividend  
this is called the dividend rule

32-bit integers have an implied number point on the right

Setting the number point in the middle allows fractional values

bits to the left of the point represent non-negative powers of 2

bits to the right of the point represent negative powers of 2

To illustrate assume 8-bit patterns with point in middle:

Implied number point

$$\begin{array}{ccccccc} 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ \uparrow & & \uparrow & & \downarrow & & \uparrow & & \uparrow \\ 2^{+2} & & 2^{+1} & & & & 2^{-1} & & 2^{-3} \end{array} = 4 + 2 + 0.5 + 0.125 = 6.625$$

$$\text{In hex: } 6.A = 6 * 16^0 + 10 * 16^{-1} = 6 + 10 * 0.0625 = 6.625$$

The position of the number point is set, thus the name “fixed-point”

We would like to be able to represent ranges of values

Very large integer parts

Very small fractional parts

This requires that the number point *slide* or *float*

Representation must specify number point location

Floating Point Representation must include:

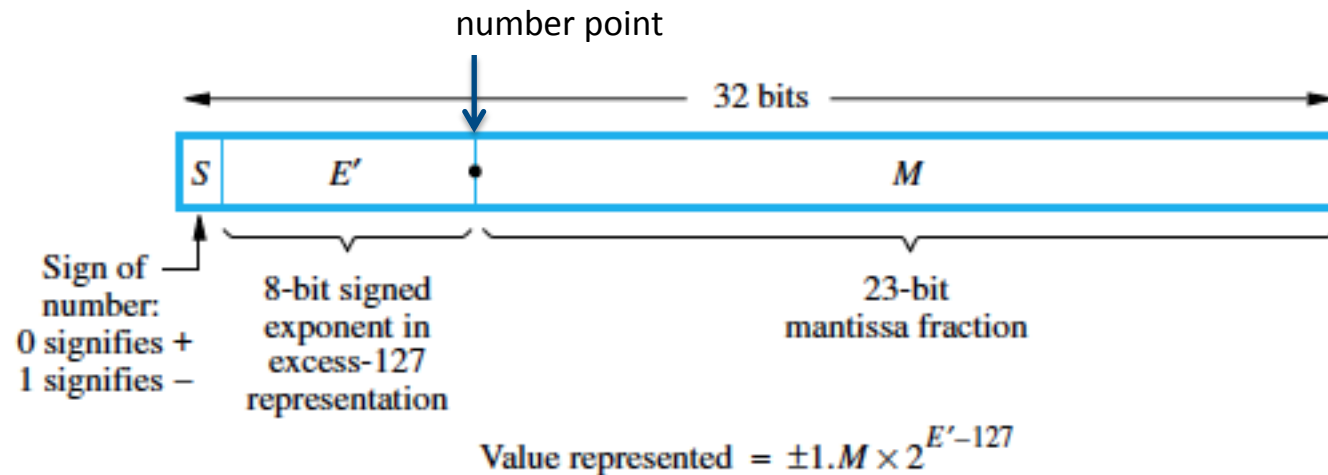
- Sign of number
- Significant bits
- Signed exponent for an implied base of 2

Width of exponent field determines range

Number of significant bits determines precision

IEEE-754 Standard specifies a common floating format

- 32-bit single precision
- 64-bit double precision



$E' = \text{exponent} + 127$  (excess-127 form) also called the “characteristic”

“Normalized” numbers have an implied 1 to the left of the number point

With single precision numbers  $0 \leq E' \leq 255$

But endpoints (0 and 255) are used for special cases

0 denotes exponent of -126 for denormalized

Denormalized numbers have 0 to left of number point

$E'=0$  and  $M \neq 0$  for denormalized numbers

$E'=0$  and  $M=0$  for true zero

Denormalized numbers allow gradual underflow

$E'=255$  denotes  $\pm\text{infinity}$  or NaN (not a number)

$E'=255$  and  $M \neq 0$  for NaN (e.g.  $0/0$  or  $\sqrt{-1}$ )

$E'=255$  and  $M=0$  for  $\pm\infty$

Provides 7 decimal places of precision

With the hidden 1, the significand is essentially 24 bits

X, the number of decimal places would be such that:

$$10^X \approx 2^{24}$$

$$\log(10^X) \cong \log(2^{24})$$

$$X = 24 * \log(2) = 24 * 0.301 = 7.22$$

Approximate range for normalized values is  $\pm(10^{\pm 38})$

$\pm 2^{-126}$  to  $\pm 2^{+128}$

0x14140000



$$\text{Value represented} = 1.001010 \dots 0 \times 2^{-87}$$

$$\text{Exponent} = \text{characteristic } 00101000 - 127 = 40 - 127 = -87$$

How would  $763.5_{10}$  be represented as a single precision IEEE floating point number?

$$763.5_{10} = 2FB.8_{16} = 1011111011.1_2 = 1.0111110111 \times 2^9$$

Hence the sign bit = 0

$$\text{the characteristic} = 9 + 127 = 136_{10} = 10001000_2$$

the 23-bit mantissa is 01111101110000000000000

The corresponding IEEE single precision representation is

s	characteristic	mantissa
0	10001000	01111101110000000000000

This 32-bit pattern can be written in short hand form using 8 hex digits: 443EE000

## Converting decimal numbers to floating point format with the aid of a calculator.

Given a decimal value  $X$ , if we compute  $N = \text{Floor}(\log_2 X)$ , this will produce the exponent needed in expressing  $X$  in the form  $1.M \times 2^N$ .

$\text{Floor}(\text{number})$  is defined as the largest integer less than or equal to number. For example:

$$\text{Floor}(4.8) = 4$$

$$\text{Floor}(-3.2) = -4$$

To find the exponent needed for the decimal value  $9.56 \times 10^4$ , we would compute

$$\begin{aligned}\log_2 (9.56 \times 10^4) &= \log_{10} (9.56 \times 10^4) / \log_{10} 2 = 4.9805 / 0.301 = 16.5447 \\ N &= \text{Floor}(16.5447) = 16\end{aligned}$$

Now if we divide  $9.56 \times 10^4 / 2^{16}$

We get 1.4587, hence  $9.56 \times 10^4 = 1.4587 \times 2^{16}$

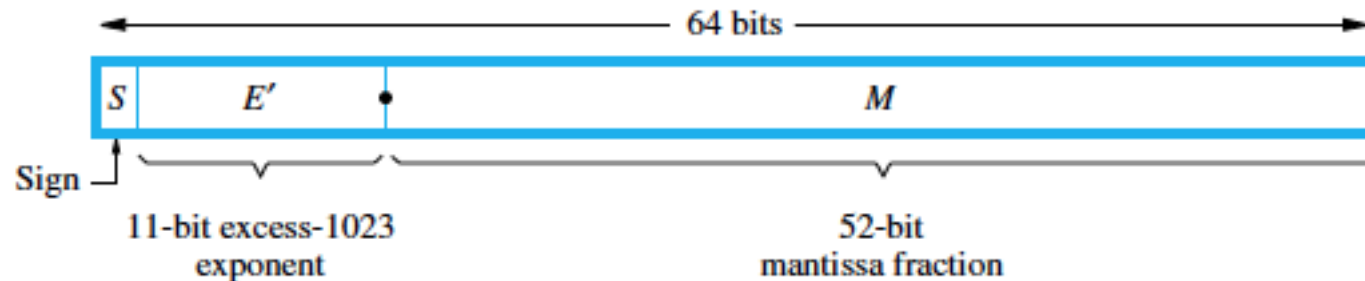
Multiply fraction to convert to 24-bit integer:  $0.4587 \times 2^{24} = 7695708$

So  $0.4587 = 7695708 \times 2^{-24} = 0x756D5C \times 2^{-24} = 0.756D5C$

Mantissa is 011101010110110101011100

Characteristic =  $16 + 127 = 143 = 0x8F$





$$\text{Value represented} = \pm 1.M \times 2^{E'-1023}$$

$$E' \text{ (characteristic)} = \text{exponent} + 1023$$

$E'=0$  and  $M \neq 0$  for denormalized numbers

$E'=0$  and  $M=0$  for true zero

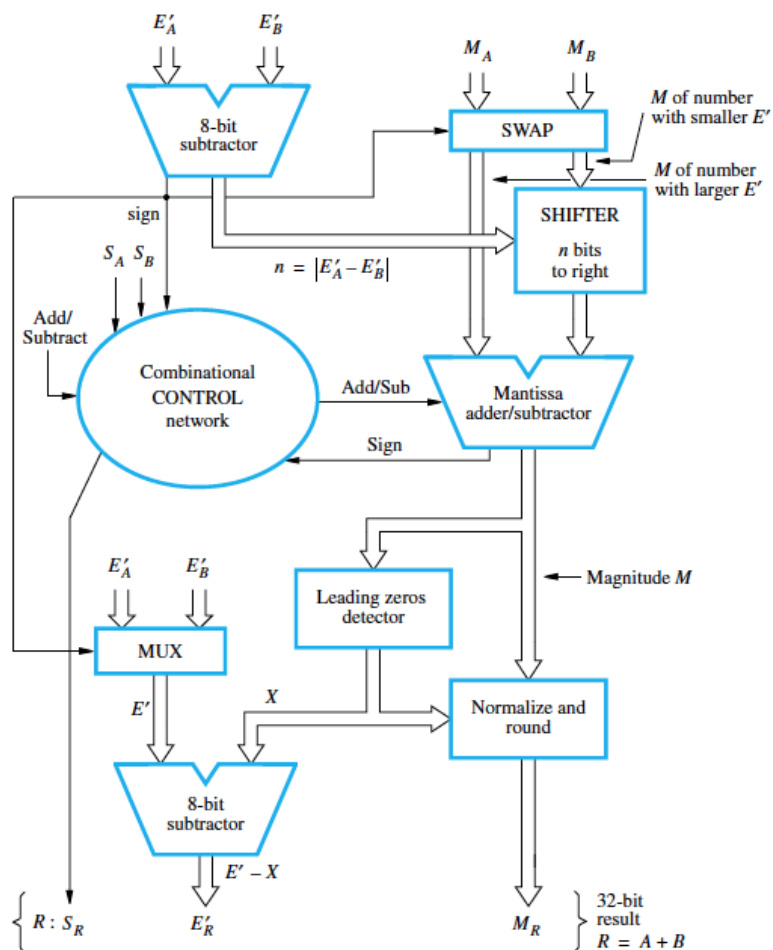
$E'=2047$  and  $M \neq 0$  for NaN (e.g.  $0/0$  or  $\sqrt{-1}$ )

$E'=2047$  and  $M=0$  for  $\pm\infty$

Provides 15 decimal places or precision

Approximate range of  $\pm(10^{\pm 308})$

1. Select number with smaller exponent
2. Shift its mantissa right  $n$  bits  
where  $n$  is the difference between the exponents
3. Add/subtract the two mantissas and set sign of result
4. Normalize and round the result if necessary



mantissa is shifted right to align for addition or subtraction

low bits are lost when this is done

Guard, round and sticky bits increase the accuracy of result

sticky bit remains a 1 once a non-zero bit passes through

guard and round retain the two most recent bits shifted out

Rounding of the result can be based on these 3 bits (GRS)

Most systems allow the programmer to specify the rounding mode



Rounding Mode	Description
Round to nearest	Add 1 to significand if $GRS > 100$ Add 1 to significand if $GRS = 100$ and LSB of significand = 1
Round towards zero	Truncate (drop bits to right of significand)
Round toward +infinity	Add 1 to significand if the result is positive and either guard or sticky bit = 1
Round toward -infinity	Add 1 to significand if result is negative and either guard or sticky bit = 1

## Multiply rule:

1. Exponent of product = sum of exponents of factors  
(check for exponent overflow)
2. mantissa of product = 1<sup>st</sup> mantissa times 2<sup>nd</sup> mantissa
3. Normalize and round result if necessary

## Divide rule:

1. Exponent of quotient = dividend exponent – divisor exponent  
(check for underflow)
2. Quotient mantissa = dividend mantissa / divisor mantissa
3. Normalize and round result if necessary

- Human readable information must also be represented
  - Characters and text
- ASCII and Unicode code are most common
- Some earlier systems used EBCDIC
- ASCII and EBCDIC are 8-bit codes
  - A single character fits into one byte
  - This allows a maximum of 256 characters
- Unicode uses 2 bytes per symbol
  - Allows many more characters
  - Including user defined symbols



#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	'	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	\$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		



- Strings are arrays of characters
  - the location of the leading byte is the string address
- MIPS uses lbu (load byte) and sb (store byte)
- MIPS has no instruction to manipulate entire strings
  - Loops containing lbu or sb are used
  - Unlike some CISC machines

- The Unicode code space allocation is shown at the right.
- The lowest-numbered Unicode characters comprise the ASCII code.
- The highest provide for user-defined codes.

Character Types	Language	Number of Characters	Hexadecimal Values
Alphabets	Latin, Greek, Cyrillic, etc.	8192	0000 to 1FFF
Symbols	Dingbats, Mathematical, etc.	4096	2000 to 2FFF
CJK	Chinese, Japanese, and Korean phonetic symbols and punctuation.	4096	3000 to 3FFF
Han	Unified Chinese, Japanese, and Korean	40,960	4000 to DFFF
	Han Expansion	4096	E000 to EFFF
User Defined		4095	F000 to FFFE



- Bytes are the smallest addressable items on MIPS
  - It is a byte-addressable machine
- Access to bits require:
  - Using the byte or word containing the desired bits
  - Using masking operations or shifting to access bits
- Contents of a register or memory word could be interpreted as a series of bits (bit string)

All MIPS machine instructions are 32 bits

Bits are number 0 to 31 from right to left

Three formats are used:

R-type

Arithmetic and logical instructions

I-Type

Instructions using an immediate operand

Same format is used for load word (lw) and store word (sw)

J-type

Contains part of jump address in bits 0 through 25

Used by jump (j) instruction and jal (to call functions)

Each has a 6-bit opcode in bit positions 26 through 31

## R-type uses:

3 register operands

opcode is always 0

function is specified by rightmost 6 bits

shamt field is used with shift instructions

### R-type instruction (register operands)

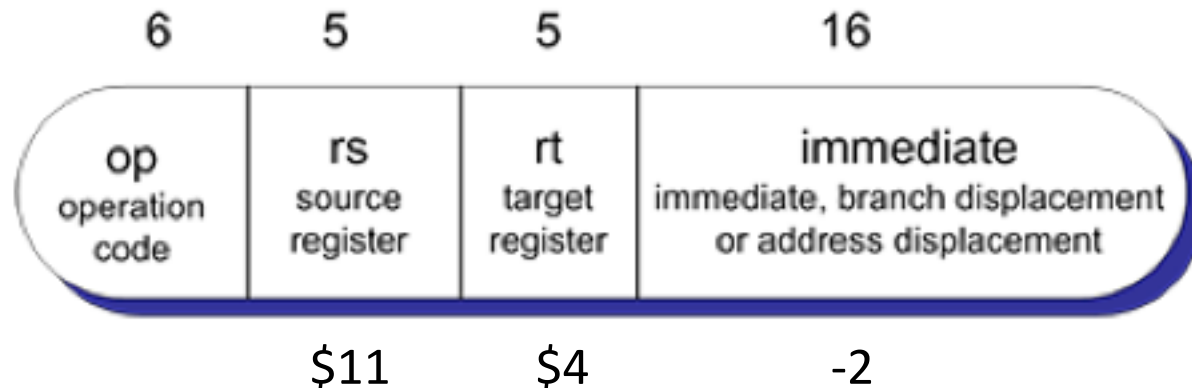


Example: add \$4,\$11,\$5

## I-type uses:

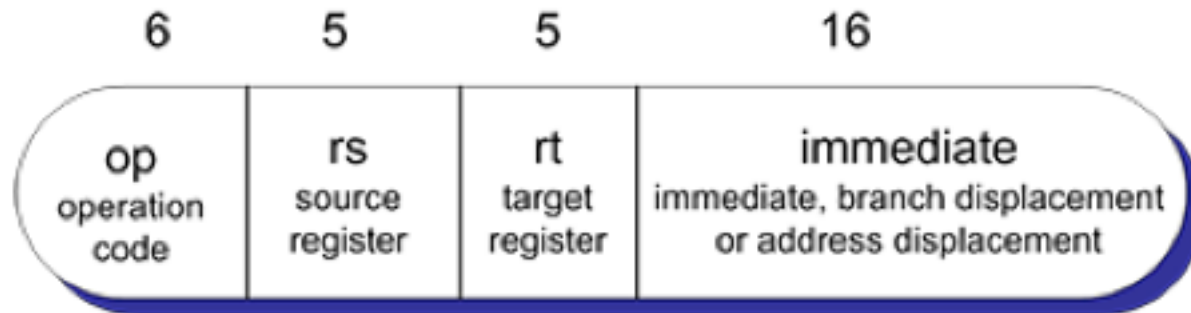
- immediate operand in rightmost 16 bits
- treated as signed operand by arithmetic instructions
- treated as unsigned operand by logical instructions
- each has a unique opcode
- this format is used by lw and sw as well

### I-type instruction (immediate)



Example: `addi $4,$11,-2`

Branch instructions also use the I-type format:  
rightmost 16 bits = # of instructions to branch  
Negative value means branch backwards  
Positive value means branch forward  
Sign-extended to 32 bits & shifted left 2 bits  
then added to the PC to produce branch address  
PC points to location following the branch instruction

**I-type instruction (immediate)**

Example: `beq $4,$8,exit`

## J-type uses:

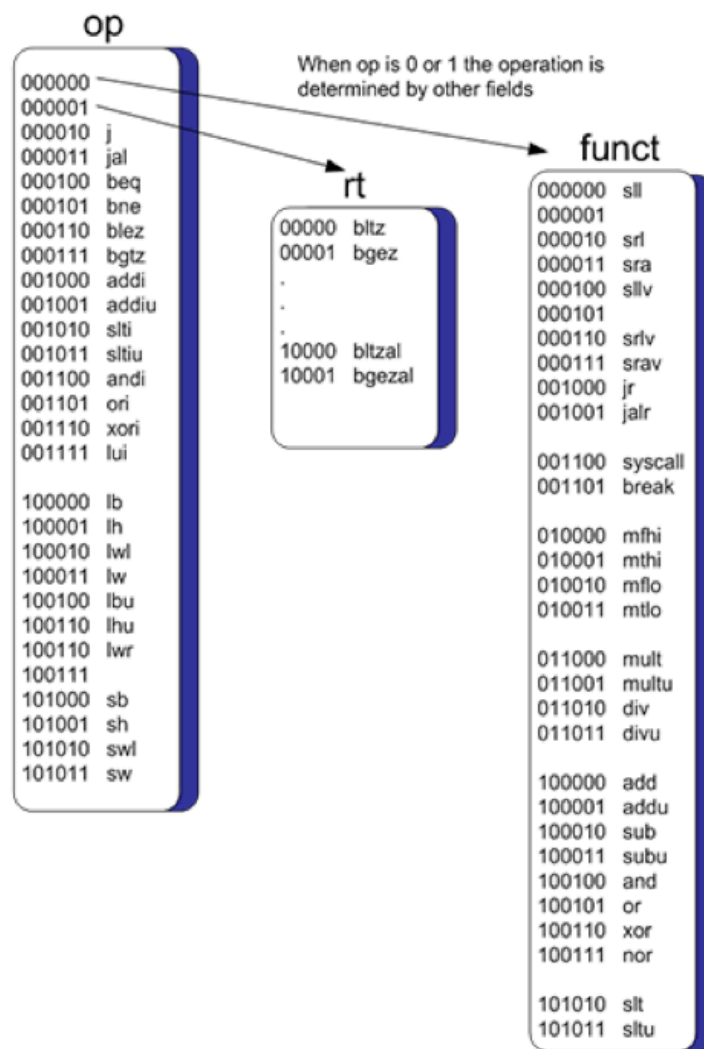
- Bits 0 through 25 in generating jump address
- Shifted left 2 bits to make it a multiple of 4
- Aligned with a 4-byte instruction boundary
- Prepended with 4 upper bits from PC
- This yields the full 32-bit jump address

### J-type Instruction (jump)



Examples: j    exit  
            jal sqrt





Appendix A contains opcodes for all instructions

All information is represented in binary within the computer

Registers and memory hold information as bit patterns

- integers (signed and unsigned)

- real numbers (floating point)

- instructions

- characters

Information is stored using digital devices

- These devices can represent 1 (on) or 0 (off)

- Hence binary (base 2) is a natural representation

The meaning of the patterns depend on their interpretation

- And on the context in which they are used

- The same bit pattern can mean entirely different things

## MIPS registers and memory words hold 32 bits

The 32-bit pattern can represent:

- a single 32-bit integer (signed or unsigned)
- a single precision floating point number
- a single machine instruction
- a group of 4 ASCII characters

## A 32-bit pattern can be written as 8 hex digits:

Example: given the pattern 0x21626364

As an integer it represents **560096100** (decimal)

As floating point it represents **1.7686579**  $\times 2^{-61}$

As a machine instruction it is **addi \$2,\$11,25444**

As ASCII characters it represents **!bcd**

The following videos explain these different encodings

They also review integer and floating point arithmetic

This will serve as a basis for discussing:

- The ALU operation and implementation

- The design and operation of the floating point hardware

If the representation and bit pattern are given, we know the value

## Sign & magnitude

If MSB=0, value = +magnitude (0110 → +110 i.e. +6)

If MSB=1, value = -magnitude (1011 → -011 i.e. -3)

## One's complement

If MSB=1, flip each bit to get the value (1011 → -0100 i.e. -4)

same as subtracting from modulus-1 (for 4 bits, modulus-1 = 16-1=15)

$$(1111 - 1011) = 0100 \text{ or } 15 - 11 = 4$$

If MSB=0, use bit pattern as the value (0110 → +0110 i.e. +6)

## Two's complement

If MSB=1, flip each bit and add 1 to get the value (1011 → -0101 i.e. -5)

Same as subtracting from modulus (16 - 11 = 5)

If MSB=0, use bit pattern as the value (0111 → +0111 i.e. +7)

Biased is also referred to as excess representation

Representation is value + bias

The threshold (bias) must be specified

The number of available bits ( $n$ ) & threshold determine the range

Value is recovered by subtracting bias from representation

midpoint ( $2^{n-1}$ ) as bias gives same range as  $n$ -bit two's complement

In general, range =  $-\text{bias}$  to  $(2^n - 1) - \text{bias}$       $n$  = number of bits

Not using midpoint as bias gives a lopsided range

All zero bits represent the lower limit    $(0 - \text{bias})$

All one bits represent the upper limit    $(2^n - 1) - \text{bias}$

Example: if number of bits  $n=4$ , and bias =  $2^3 = 8$  (excess-8)

+5 is represented as  $5 + 8 = 13$  (1101)

-3 is represented as  $-3 + 8 = 5$  (0101)

Given a bit pattern and the bias, we know the value

Value = representation – bias

1110 →  $14 - 8 = +6$  (the pattern is 6 greater than 8)

0110 →  $6 - 8 = -2$  (the pattern is 2 less than 8)

Range = -bias to  $(2^n - 1)$ -bias (-8 to  $15-8=7$ ) in this case