Data Representation and Manipulation

Module 03

Course Notes

The MIPS Word

- 32-bit architecture
- 1 byte = 8 bits; 4 bytes = 1 word
- Bits numbered 31, 30, . . ., 0
- Most significant bit (MSB) is bit 31
- Least significant bit (LSB) is bit 0
- In many examples, we will use only 4 bits to illustrate
- Sometimes, numbers written in *hexadecimal*

Basic Error detection when transmitting Data

- Parity Bit: Top most bit, very basic error detection.
- When bits, bytes(8bits), words(32 bits) are transmitted from computer to computer, parity check in case any error in transmission
- 10011011 : Highest bit states if there is an even or odd number of 1's in the 7 bit number: 1 indicates there is an even number of 1s
- Could also be the 9th bit in an 8bit number: 000110111
- Even parity: highest order bit is true when there are even number of 1s
- Odd parity: highest order bit is true when there are an odd number of 1s

USASCII code chart

Bits				b_7 b_6 b_5 →	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4 ↓	b_3 ↓	b_2 ↓	b_1 ↓	Column → Row ↓	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

128 ASCII characters: 1972

Characters

- ASCII (American Standard Code for Information Interchange)
- Uses 7 bits to represent 128 different characters
- 8th bit (topmost) used as parity check (error detection)
- 4 characters fit into MIPS 32-bit word

  128 possibilities include upper and lower case Roman letters, punctuation marks, some computer control characters
- Partial table on page 106 of text
- Unicode: 16 bits per character

  (English isn't the only language!)

Mips: Single word 32 bits allows 4 chars

- **Need many 32 bit words to represent "hello world"**

Mips: Single word 32 bits allows 4 chars

- **Need many 32 bit words to represent “hello world”**
- **How many bytes are needed to represent this**
- **A) 7**
- **B) 8**
- **C) 9**
- **D)10**
- **E)11**

Two's Complement Representation

In use today

- Idea: Let MSB represent the negative of a power of 2
- With 4 bits, bit 3 (MSB) represents -2^3
- $1110 = -2^3 + 2^2 + 2^1 = -2$
- With 4 bits, can represent -8 (1000) to $+7$ (0111)
- With 32 bits, can represent $-2,147,483,648$ to 2,147,483,647
- Usefulness becomes apparent when we try arithmetic

Useful: all negative numbers will have MSB to 1

Negating a Two's Complement Number

- For a bit pattern x, let $\bar{x}$ be the result of inverting each bit
- Example: $x = 0110$, $\bar{x} = 1001$
- Since $x + \bar{x} = -1$, $-x = \bar{x} + 1$
- To negate a number in two's complement representation, invert every bit and add 1 to the result

How is -4 represented in Two's complement

- A)1011
- B)1001
- C)0111
- D) 1111
- E) 1100

Addition of Two's complement

Addition

- To add two two's complement numbers, simply use the "elementary school algorithm", throwing away any carry out of the MSB position
- To subtract, simply negate and add
- Problem: what if answer cannot be represented? (called overflow)
- Overflow in addition cannot occur if one number is positive and the other negative
- If both addends have same sign but answer has different sign, overflow has occurred

Examples on the Board

Sign Extension

- With 4 bits, 0110 is +6. With 8 bits, what is +6?
- With 4 bits, 1010 is -6. With 8 bits, what is -6?
- To expand number of bits used, copy old MSB into new bit positions.
- This works because

$$-2^i + 2^{i-1} + 2^{i-2} + \cdots + 2^{j+1} + 2 \cdot 2^j = 0$$

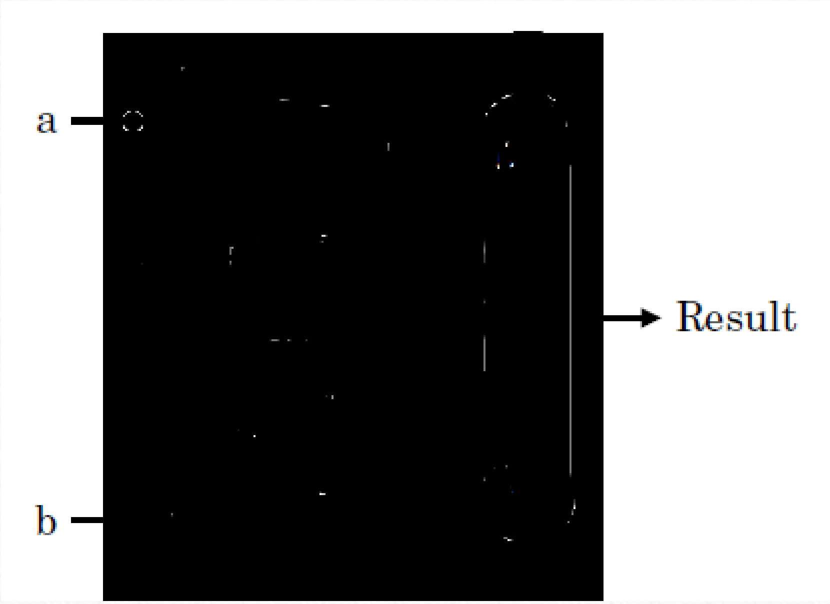
- **For example : -4 1100 in 8 bits. Simply extend MSB:**
- **➔ 1111 1100 = -4 in two's complement form**

Building An Addition Circuit

	(0)	*(0)*	*(1)*	*(1)*	*(0)*	*(Carries)*
. . .	0	0	0	1	1	1
. . .	0	0	0	1	1	0
. . .	(0) 0	(0) 0	(0) 1	(1) 1	(1) 0	(0) 1

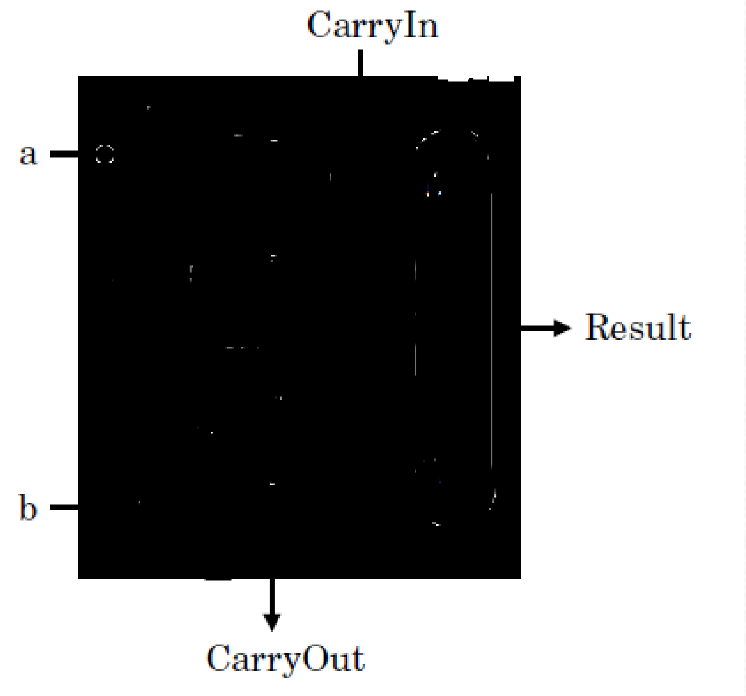
Designing an Adder

Two bit input
1 bit result

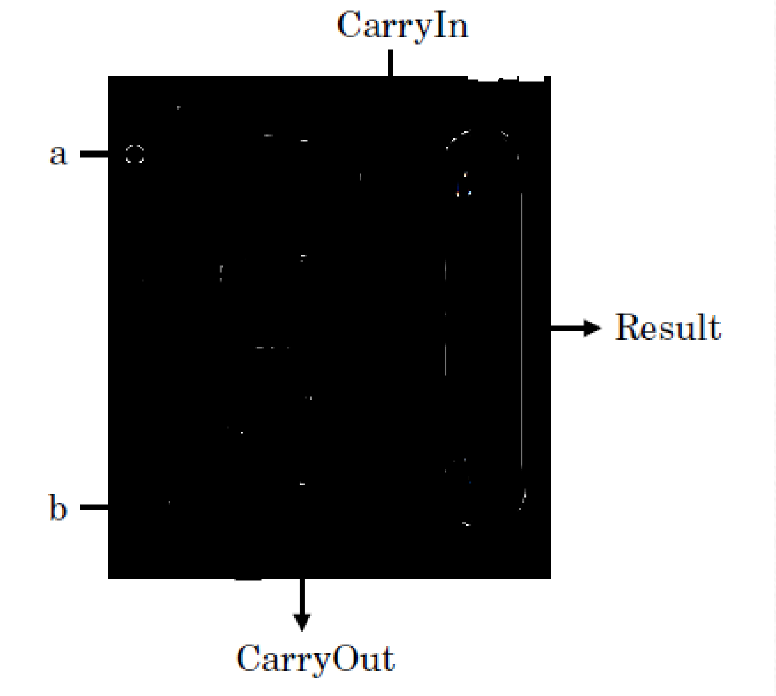


Is this all we need?

Need to have carryIn and carryout bits



How might I design this?

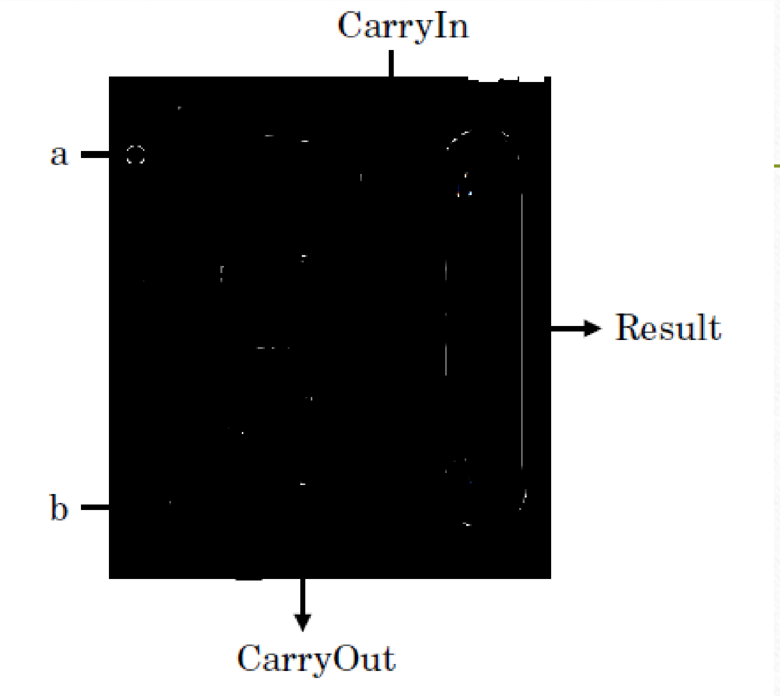


AND GATE

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

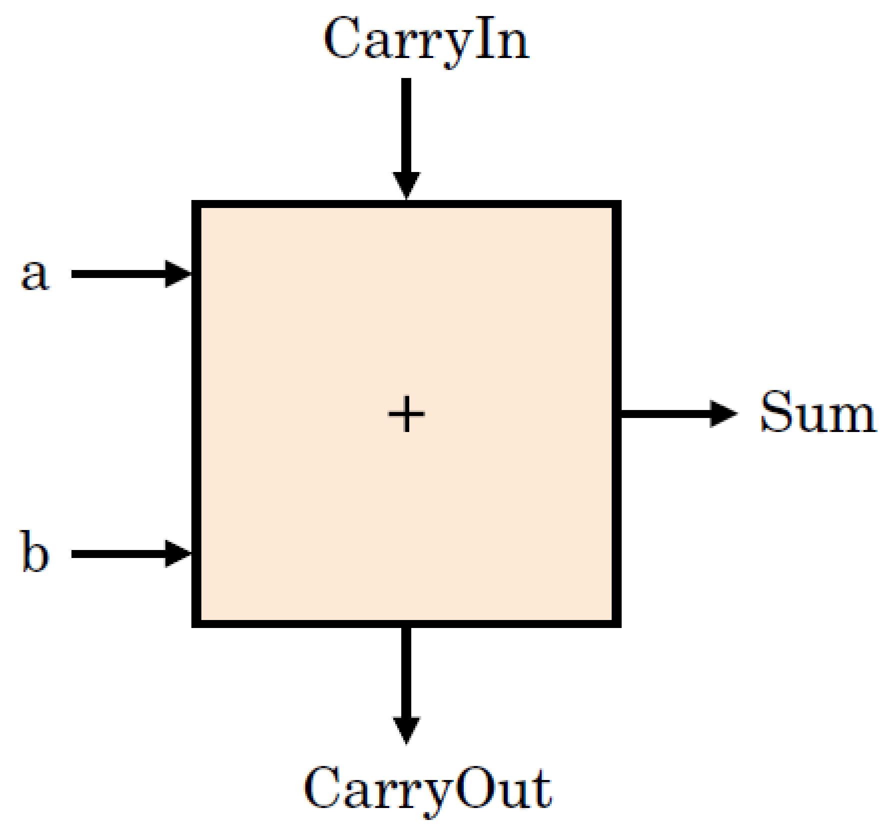
OR GATE

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1



A	B	C_{in}	C_{out}	Result
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
…			…	

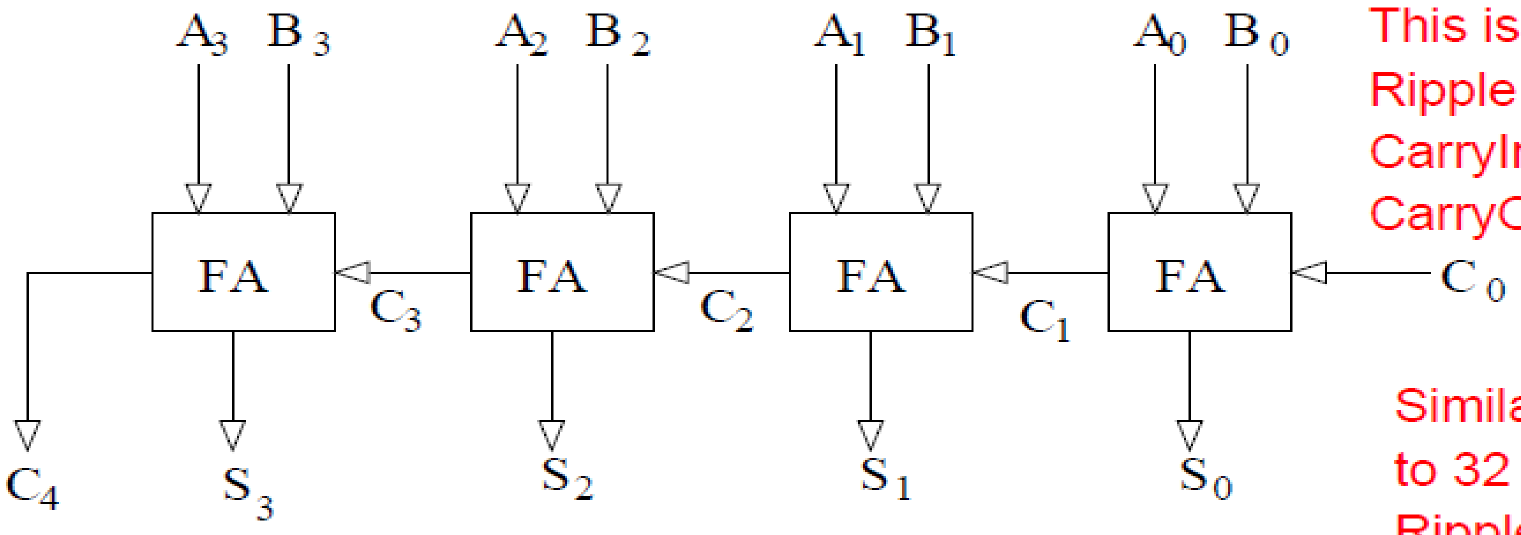
Complete the Truth Table, Implement the circuit



Course Notes Full Adder part of Arithmetic Logic Unit

Ripple-Carry Adder

- 4-bit example



This is called Ripple Carry Adder CarryIns and CarryOuts

Similar to 32 bit Ripple Carry Adder

- Easy to extend to 32 bits
- Can be slow; "carry-lookahead" idea improves speed

Arithmetic Logic Unit

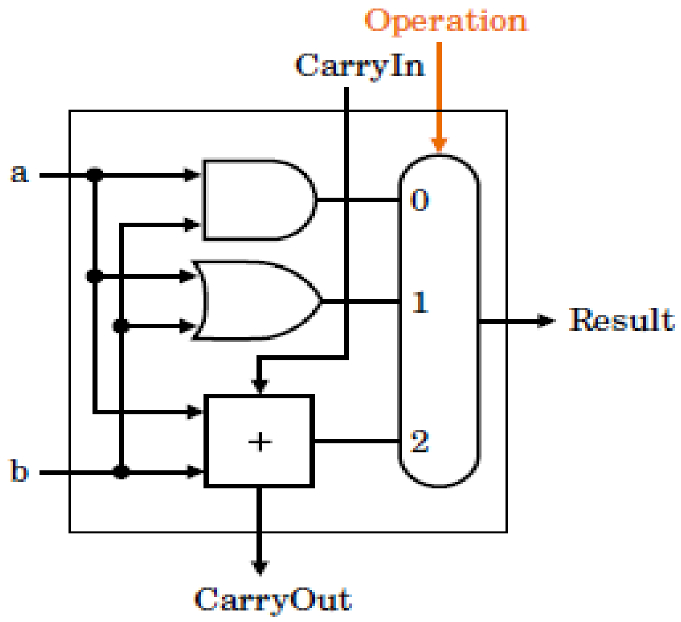
*Basic ALU performs 3 operations.

*Need to incorporate subtraction:

*Subtraction can be thought of
Addition- negating one of the inputs

*Negate second input
- Invert all the bits and add 1

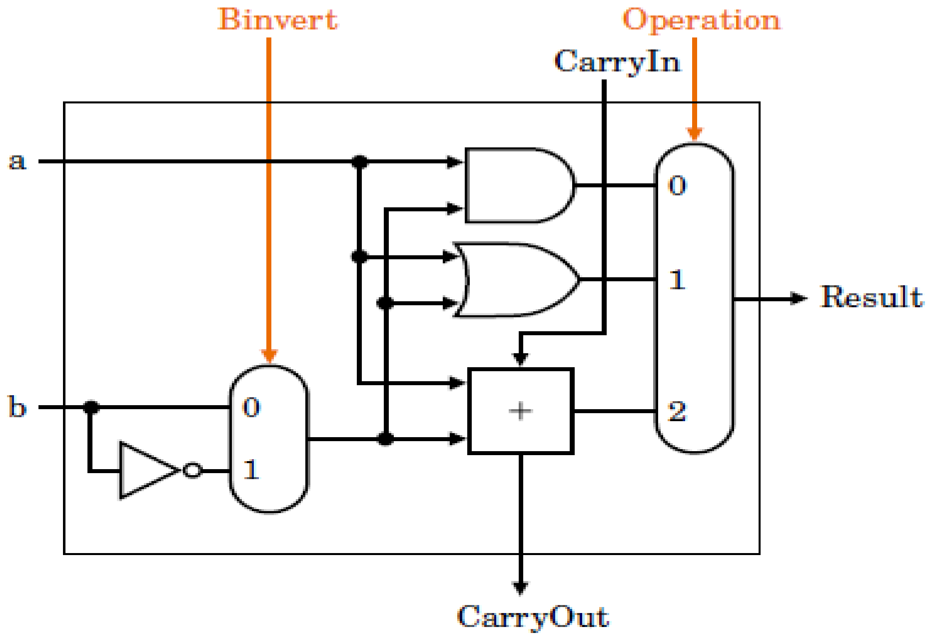
A 1-Bit ALU



- Extends functionality of full adder
- Performs AND, OR, addition
- Connect 32 of these as with ripple-carry adder to perform 32-bit operations

Improving the 1-Bit ALU

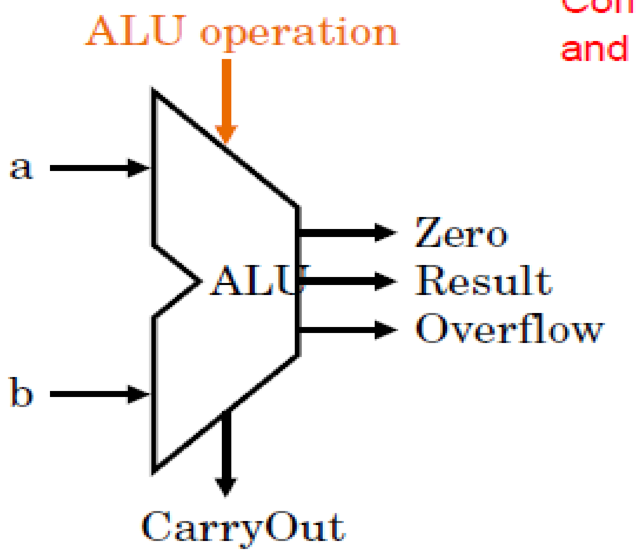
- How to implement subtraction?
- To subtract b from a, invert bits of b, add to a, add 1
- Box below will do this, if added 1 is put into CarryIn at top of chain when subtraction is desired



Abstracting Away ALU Details

- Book makes further improvements to support other operations that assist in branching (Appendix B.5-
  figures B.5.9 and B.5.10 in particular)
- From now on, we use symbol below
- Same shape used for ripple-carry adder, so remember to label them

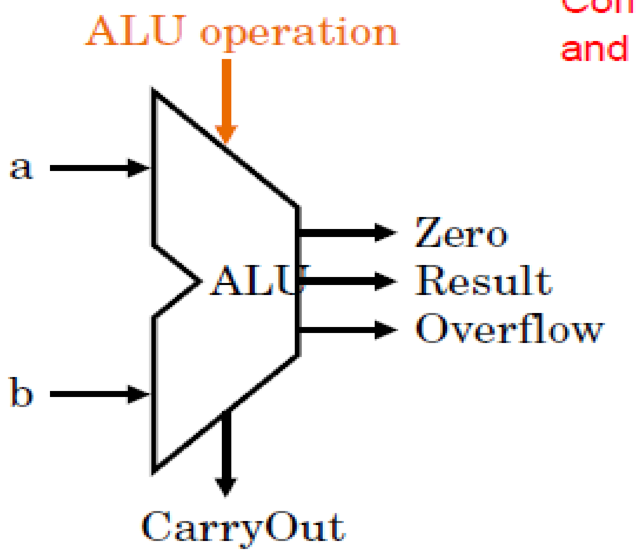
zero bit: supports branching.
Comparing two registers by subtracting them and checking of the result is zero.



Abstracting Away ALU Details

- Book makes further improvements to support other operations that assist in branching (Appendix B.5-
  figures B.5.9 and B.5.10 in particular)
- From now on, we use symbol below
- Same shape used for ripple-carry adder, so remember to label them

zero bit: supports branching.
Comparing two registers by subtracting them and checking of the result is zero.



Representing Numbers That Aren't Integers

- Uses idea of scientific notation: -3.45×10^3
- Sign, significand (fraction, mantissa, exponent)
- Normalized: single digit to left of decimal point
- For computers, natural to use 2 as base
- Example: $1.01_2 \times 2^4$
- In normalized binary, leading digit of significand is always 1 (can omit it from internal representation)
- How to represent 0?

Floating-Point Representation

- MIPS uses the IEEE 754 floating-point standard format

31	30 Range 23	22 Fraction: increasing precision 0
s	exponent	significand
1 bit	8 bits	23 bits

- allows numbers from 2.0×10^{-38} to 2.0×10^{38}, roughly
- Double precision: uses two 32-bit words, 11 bits for exponent, 52 bits for significand
- Exponent is stored in "biased" notation: most negative exponent is all 0's, most positive is all 1's
  This allows for quick comparisons, speeds up sorting
- Thus value represented is
  $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-\text{Bias})}$,
  where Bias $= 127$ for single precision
- Special case: 0000 0000 exponent reserved for 0

Overflow and Underflow can occur

Bias notation : IEEE Floating Point standard

- Let 1111 1111 be the most positive exponent
- 0000 0000 be the most negative exponent.
  - Makes sorting easier
- Normalized exponent: takes positive binary number represented by exponent bits and subtracts 127 from this.
- Therefore all exponents have positive value: take the value minus the bias
- Exponent of +1: 1000 000 **(128 -127)**
- Exponent of -1: 0111 1110 **(126-127)**

- IEEE 754 Standard: floating point representation
- Also makes sorting and comparing numbers easier
  - This is why sign is most significant bit
  - Also why exponent bits are before significand bits

  - **0000 0000 and 1111 1111 are used as special cases**
  - Therefore Range of Exponents: 8 bit exponent :
  - ____

Algorithm for conversion of fractions

- Multiply fraction by 2 repeatedly.
- 0.625 x 2 = **1**.25 KEEP 1 as first binary digit **0.1**
- Next .25 x 2 = **0**.5 : 0 as next binary digit : **0.10**
- Next .5 x 2 = **1**.0 : 1 as next binary digit : **0.101**
- **Done**
- **.625 is .101 as binary**
- **NOT ALL fractions can be represented in binary exactly**

Algorithm for conversion of fractions

- 0.1 decimal 1/10
- 0.1 x 2 = **0.2** KEEP 0 as first binary digit **0.0**
- 0.2 x 2 = **0.4** KEEP 0 as next binary digit **0.00**
- 0.4 x 2 = **0.8** KEEP 0 as next binary digit **0.000**
- 0.8 x 2 = **1.6** KEEP 1 as next binary digit **0.0001**
- 0.6 x 2 = **1.2** KEEP 1 as next binary digit **0.00011**
- **Repeat with 0.2 will lead to**
- 0.1 x 2 = **0.2** KEEP 0 as first binary digit **0.0 …. repeating**
- **KEEP Going .00011000110001100011 . . .**
- **Repeating pattern**
- **Therefore some numbers produce infinite binary expansion.**

Convert 0.375 to binary

- A) 0.11
- B)0.0101
- C) 0.01101
- D) 1.11
- E) 0.011

Convert this Fractional number to IEEE Floating Point Representation

Start: 42.3125

42 = 101010

.3125x2 = 0.625 Apply the same algorithm from previous slides

.625x2 = 1.25

.25x2 = 0.5

.5x2 = 1

.3125 = .0101

→ 42.3125 = 101010.0101 = 1.010100101x2^5 **Need to Normalize : Only one leading 1**

Sign bit: 0 (pos)

Exponent – 127 = 5 → Exponent = 132 = 10000100

IEEE754: 0 10000100 01010010100000000000000 **Final 32 Bits Representation**

Floating-Point Addition

- Decimal example: $9.54 \times 10^2 + 6.83 \times 10^1$ (assume we can only store two digits to right of decimal point)

  Only 1 leading digit before decimal

  1. Match exponents: $9.54 \times 10^2 + .683 \times 10^2$
  2. Add significands, with sign: 10.223×10^2
  3. Normalize: 1.0223×10^3
  4. Check for exponent overflow/underflow
  5. Round: 1.02×10^3
  6. May have to normalize again
- Same idea works for binary

A: 0 10000100 0101001010...

B: 1 10000011 0001001010...

A's exponent: 5

B's exponent: 4

A's mantissa: 1.0101001010...

B's mantissa: 1.0001001010...

Must shift B's mantissa, exponent by 1 so they become 5

0.1000100101

Because we are adding two numbers of different signs, we use signed magnitude addition: subtract the smaller mantissa from the larger mantissa, and keep the sign of the larger

1.0101001010... x2^5

- 0.10001001010... x2^5 Performing Subtration

(+) 0.11001001010... x2^5

Normalize: 1.1001001010... x2^4

Sign bit = 0

Exponent -127 = 4 -> exponent = 131 = 10000011

Answer: 0 10000011 10010010100000000000000 = 25.15625