Lecture 6: Integer Representations

Spring 2024
Jason Tang

Topics

- Number representations (somewhat review)
- Integer arithmetic
- Integer overflow

Number Representation

- Computer words consist of groups of bits, which can be used to represent binary numbers

- Whereas natural numbers can be represented in binary, what about:
  - How to represent negative numbers?
  - What is the largest number representable in a computer word?
  - What happens if an operation results in a value larger than that largest number?

Binary-Coded Decimal

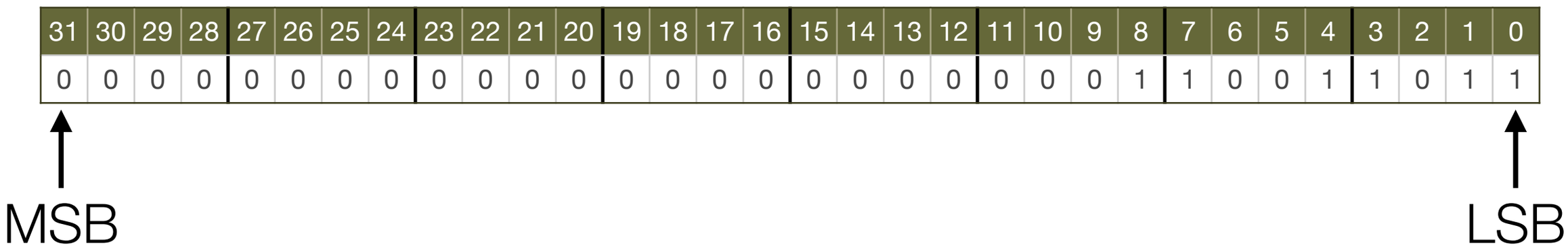
- Early computers used 4 bits to represent a digit from 0 to 9; the other bit patterns were either don’t care or special codes
    - Allowed precise human-readable representation of decimal values
    - Increased complexity in hardware
- Example BCD bit pattern: $(0100\ 0001\ 0001)_{BCD} = 411_{10}$

Unsigned Integer Numbers

- Traditional binary values are written right to left, such that the least significant bit (LSB) is on the far right and the most significant bit (MSB) is on the far left

- In any base **B**, the value of the **i**th digit **d** is **d** × **B**^{i}, where **i** is 0 for the right-most position

- Example: representing 411_{10} as a 32-bit unsigned binary value
  $= (1 \times 2^8) + (1 \times 2^7) + (1 \times 2^4) + (1 \times 2^3) + (1 \times 2^1) + (1 \times 2^0)$
  $= 256 + 128 + 16 + 8 + 2 + 1$



Character Encodings

- Hardware operates at the binary level, but displays values as human-readable representation

- Two historical standard exist to represent characters: American Standard Code for Information Interchange (ASCII) and Extended Binary Coded Decimal Interchange Code (EBCDIC)

- Modern ASCII defines a **7-bit** encoding of 95 printable characters and 33 control characters

    - **Major difference** between a digit (as used within hardware) and an ASCII digit (as seen by a human operator)

ASCII Table

Dec	Hex	Oct	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr
0	0	000	NULL	32	20	040	&#032;	Space	64	40	100	&#064;	@	96	60	140	&#096;	`
1	1	001	Start of Header	33	21	041	&#033;	!	65	41	101	&#065;	A	97	61	141	&#097;	a
2	2	002	Start of Text	34	22	042	&#034;	"	66	42	102	&#066;	B	98	62	142	&#098;	b
3	3	003	End of Text	35	23	043	&#035;	#	67	43	103	&#067;	C	99	63	143	&#099;	c
4	4	004	End of Transmission	36	24	044	&#036;	$	68	44	104	&#068;	D	100	64	144	&#100;	d
5	5	005	Enquiry	37	25	045	&#037;	%	69	45	105	&#069;	E	101	65	145	&#101;	e
6	6	006	Acknowledgment	38	26	046	&#038;	&	70	46	106	&#070;	F	102	66	146	&#102;	f
7	7	007	Bell	39	27	047	&#039;	'	71	47	107	&#071;	G	103	67	147	&#103;	g
8	8	010	Backspace	40	28	050	&#040;	(	72	48	110	&#072;	H	104	68	150	&#104;	h
9	9	011	Horizontal Tab	41	29	051	&#041;	)	73	49	111	&#073;	I	105	69	151	&#105;	i
10	A	012	Line feed	42	2A	052	&#042;	*	74	4A	112	&#074;	J	106	6A	152	&#106;	j
11	B	013	Vertical Tab	43	2B	053	&#043;	+	75	4B	113	&#075;	K	107	6B	153	&#107;	k
12	C	014	Form feed	44	2C	054	&#044;	,	76	4C	114	&#076;	L	108	6C	154	&#108;	l
13	D	015	Carriage return	45	2D	055	&#045;	-	77	4D	115	&#077;	M	109	6D	155	&#109;	m
14	E	016	Shift Out	46	2E	056	&#046;	.	78	4E	116	&#078;	N	110	6E	156	&#110;	n
15	F	017	Shift In	47	2F	057	&#047;	/	79	4F	117	&#079;	O	111	6F	157	&#111;	o
16	10	020	Data Link Escape	48	30	060	&#048;	0	80	50	120	&#080;	P	112	70	160	&#112;	p
17	11	021	Device Control 1	49	31	061	&#049;	1	81	51	121	&#081;	Q	113	71	161	&#113;	q
18	12	022	Device Control 2	50	32	062	&#050;	2	82	52	122	&#082;	R	114	72	162	&#114;	r
19	13	023	Device Control 3	51	33	063	&#051;	3	83	53	123	&#083;	S	115	73	163	&#115;	s
20	14	024	Device Control 4	52	34	064	&#052;	4	84	54	124	&#084;	T	116	74	164	&#116;	t
21	15	025	Negative Ack.	53	35	065	&#053;	5	85	55	125	&#085;	U	117	75	165	&#117;	u
22	16	026	Synchronous idle	54	36	066	&#054;	6	86	56	126	&#086;	V	118	76	166	&#118;	v
23	17	027	End of Trans. Block	55	37	067	&#055;	7	87	57	127	&#087;	W	119	77	167	&#119;	w
24	18	030	Cancel	56	38	070	&#056;	8	88	58	130	&#088;	X	120	78	170	&#120;	x
25	19	031	End of Medium	57	39	071	&#057;	9	89	59	131	&#089;	Y	121	79	171	&#121;	y
26	1A	032	Substitute	58	3A	072	&#058;	:	90	5A	132	&#090;	Z	122	7A	172	&#122;	z
27	1B	033	Escape	59	3B	073	&#059;	;	91	5B	133	&#091;	[	123	7B	173	&#123;	{
28	1C	034	File Separator	60	3C	074	&#060;	<	92	5C	134	&#092;	\	124	7C	174	&#124;	|
29	1D	035	Group Separator	61	3D	075	&#061;	=	93	5D	135	&#093;	]	125	7D	175	&#125;	}
30	1E	036	Record Separator	62	3E	076	&#062;	>	94	5E	136	&#094;	^	126	7E	176	&#126;	~
31	1F	037	Unit Separator	63	3F	077	&#063;	?	95	5F	137	&#095;	_	127	7F	177	&#127;	Del

http://www.asciichars.com/

EBCDIC Table

2nd hex digit

1st hex digit

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	DS		SP	&	-									0
1	SOH	DC1	SOS				/		a	j			A	J		1
2	STX	DC2	FS	SYN					b	k	s		B	K	S	2
3	ETX	TM							c	l	t		C	L	T	3
4	PF	RES	BYP	PN					d	m	u		D	M	U	4
5	HT	NL	LF	RS					e	n	v		E	N	V	5
6	LC	BS	ETB	UC					f	o	w		F	O	W	6
7	DEL	IL	ESC	EOT					g	p	x		G	P	X	7
8		CAN							h	q	y		H	Q	Y	8
9		EM							i	r	z	`	I	R	Z	9
A	SMM	CC	SM		C CENT	!		:								
B	VT	CU1	CU2	CU3		$	,	#								
C	FF	IFS		DC4	<	*	%	@								
D	CR	IGS	ENQ	NAK	(	)	_	'								
E	SO	IRS	ACK		+	;	>	=								
F	SI	IUS	BEL	SUB	|	¬	?	"								

https://www.eetimes.com/author.asp?section_id=216&doc_id=1285465

Number Notations

- As that this class involves lots of binary numbers, values are often written in hexadecimal
  - 411_{10} = 0x19b = 19B**h**
- This class **requires** proper terminology when referring to larger amounts

System of Units (SI)			Binary Numeral				% Difference
Factor	Name	Symbol	Factor	Name	Symbol	# of Bytes	
10^3	kilobyte	KB	2^{10}	kibibyte	KiB	1,024	2.4%
10^6	megabyte	MB	2^{20}	mebibyte	MiB	1,048,576	4.9%
10^9	gigabyte	GB	2^{30}	gibibyte	GiB	1,073,741,824	7.4%
10^{12}	terabyte	TB	2^{40}	tebibyte	TiB	1,099,511,627,776	10.0%
10^{15}	petabyte	PB	2^{50}	pebibyte	PiB	1,125,899,906,842,624	12.6%
10^{18}	exabyte	EB	2^{60}	exbibyte	EiB	1,152,921,504,606,846,976	15.3%
10^{21}	zettabyte	ZB	2^{70}	zebibyte	ZiB	1,180,591,620,717,411,303,424	18.1%
10^{24}	yottabyte	YB	2^{80}	yobibyte	YiB	1,208,925,819,614,629,174,706,176	20.9%

https://www.ramicomer.com/en/blog/conversion-and-difference-kilobyte-to-kibibyte-megabyte-to-mebibyte/

Signed Magnitude Integer

- Hardware needs to represent both positive and negative integers
- In signed magnitude, reserve one bit to indicate the sign (positive or negative)
  - Often, MSB is set to 0 for a positive value, 1 for a negative value
  - Rest of bits are the magnitude
  - Thus there can be a “positive zero” and a “negative zero”
- Examples of 8-bit signed magnitude integers:

$(+41)_{10} = (0010\ 1001)_{2\text{-signed-mag}}$

$(-41)_{10} = (1010\ 1001)_{2\text{-signed-mag}}$

One’s Complement

- With signed magnitude, subtraction requires different hardware than addition
- Complement-based representation allows for easier hardware implementation
- In one’s complement, negative numbers are represented by inverting all bits
  - Still can have positive and negative zero
- Examples of 8-bit one’s complement integers:

$(+41)_{10} = (0010\ 1001)_{2\text{-one's-comp}}$

$(-41)_{10} = (1101\ 0110)_{2\text{-one's-comp}}$

One's Complement Arithmetic

- Digits are added bit-by-bit, from right to left, with carries passed to next digit to the left
  - Subtract by adding the value's complement
  - If need to carry past left-most bit, then carry "wraps" around to right-most bit
- Examples: 42 + 54, 42 – 54, and -42 – 54

Carry Bits 0 1 1 1 1 1 0

	0 0 1 0 1 0 1 0	$= 42_{10}$
+	0 0 1 1 0 1 1 0	$= 54_{10}$
	0 1 1 0 0 0 0 0	$= 96_{10}$

Sign Bit

Carry Bits 0 0 0 1 0 0 0

	0 0 1 0 1 0 1 0	$= 42_{10}$
+	1 1 0 0 1 0 0 1	$=-54_{10}$
	1 1 1 1 0 0 1 1	$=-12_{10}$

Carry Bits 1 1 0 0 0 0 0 1

	1 1 0 1 0 1 0 1	$=-42_{10}$
+	1 1 0 0 1 0 0 1	$=-54_{10}$
	1 0 0 1 1 1 1 0	-97_{10}
+	1	1
	1 0 0 1 1 1 1 1	$=-96_{10}$

Two's Complement

- Improvement over one's complement
  - No wrap-around carry
  - Exactly one representation of zero
- To find a negative, invert all bits (like one's complement) and then add 1
- Examples of 8-bit two's complement integers

$(+41)_{10} = (0010\ 1001)_2$

$(-41)_{10} = (1101\ 0111)_2$

Unless otherwise specified, all binary integer values in this class are either unsigned or two's complement

Two's Complement Arithmetic

- Addition and subtraction same as one's complement
- If need to carry past left-most bit, that carry-out bit is **ignored**
- Examples: 42 + 54, 42 – 54 and -42 – 54

This carry bit is dropped

Carry Bits	0	1	1	1	1	1	0		
	0	0	1	0	1	0	1	0	$= 42_{10}$
+	0	0	1	1	0	1	1	0	$= 54_{10}$
	0	1	1	0	0	0	0	0	$= 96_{10}$

Sign Bit

	0	0	0	1	0	1	0		
	0	0	1	0	1	0	1	0	$= 42_{10}$
+	1	1	0	0	1	0	1	0	$=-54_{10}$
	1	1	1	1	0	1	0	0	$=-12_{10}$

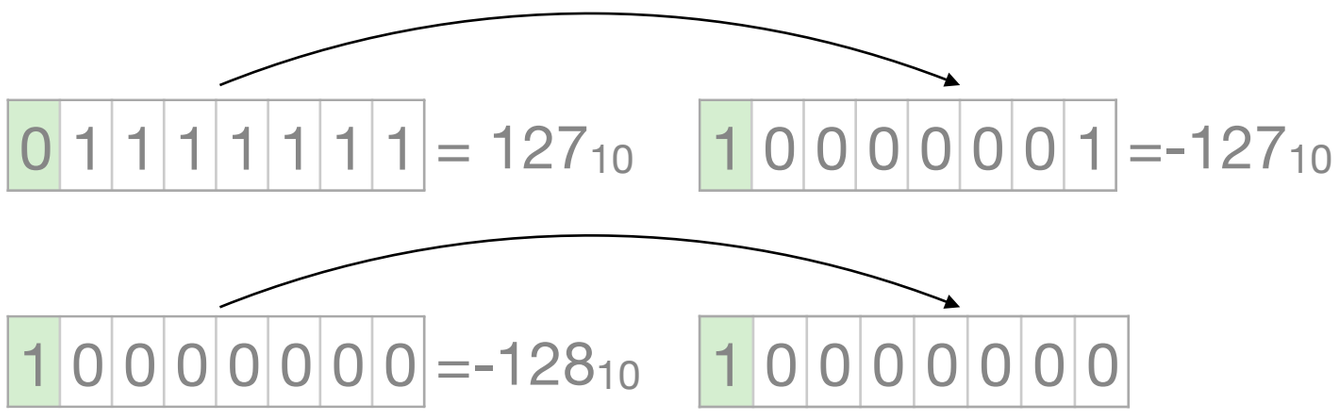
	1	1	0	1	1	1	1	0	
	1	1	0	1	0	1	1	0	$=-42_{10}$
+	1	1	0	0	1	0	1	0	$=-54_{10}$
	1	0	1	0	0	0	0	0	$=-96_{10}$

Consequences of Two's Complement

- "Negative zero" has same representation as positive zero

- Most positive number is $2^{n-1} - 1$, while least negative number is -2^{n-1}

- Thus, the most negative number has no corresponding positive number



Hint: Moving from right-to-left, skip leading 0s until reaching first 1, then flip all bits to left of that 1

Bias

- Two's complement is an example of having a bias
- For a **n**-bit value, the MSB (sign bit) really indicates **-2^n**, not **$+2^n$**
- Example: 42 and -42

	0	0	1	0	1	0	1	0
hex	2				A			

$= 2^5 + 2^3 + 2^1$
$= 32 + 8 + 2$
$= 42_{10}$

	1	1	0	1	0	1	1	0
hex	D				6			

$= \mathbf{-2^7} + 2^6 + 2^4 + 2^2 + 2^1$
$= \mathbf{-128} + 64 + 16 + 4 + 2$
$= \mathbf{-128} + 86$
$= -42_{10}$

Sign Extension

- When storing a **n**-bit integer in something larger than **n** bits, then need to decide what to do with the extra bits

- Example: store the 8-bit binary value 11010110 (“D6h” or “0xD6”) in 16 bits

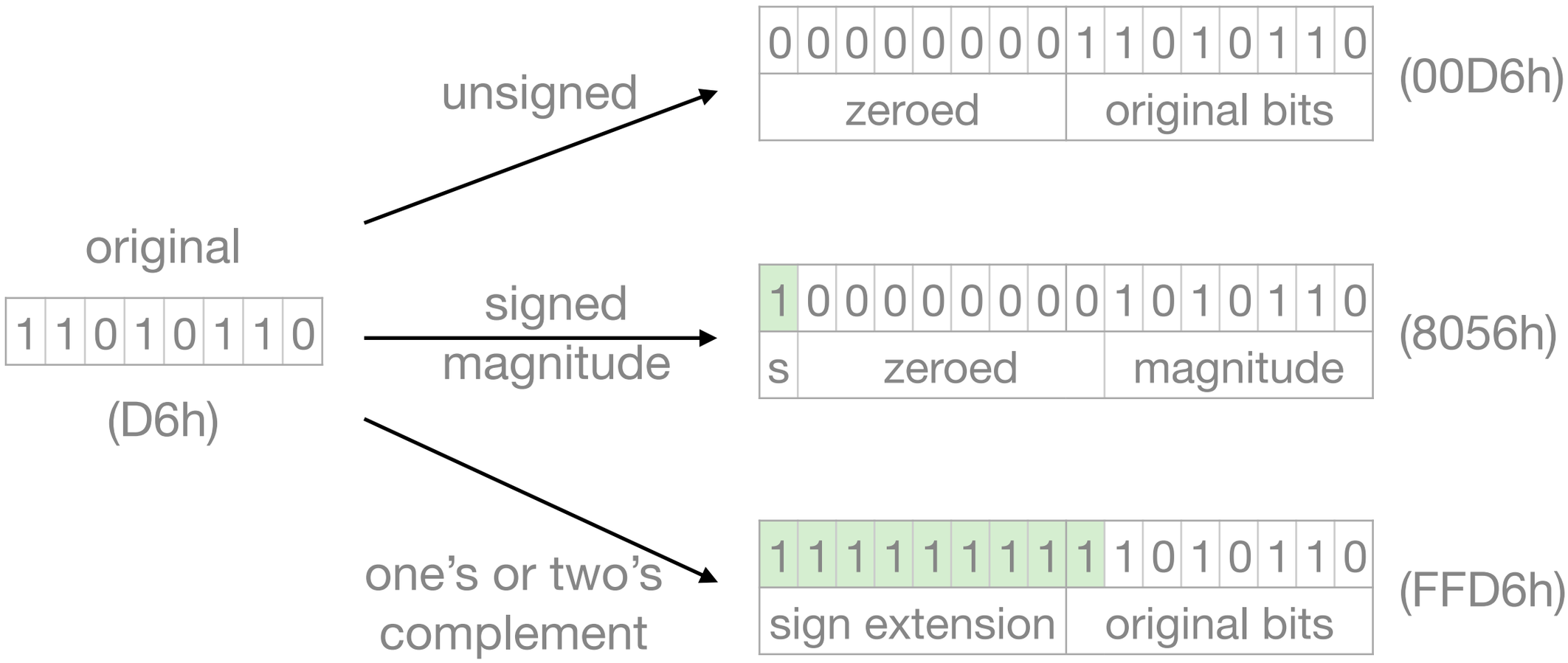
- **Depends upon underlying interpretation of those bits**

  - Unsigned: store original bits in least significant portion; set upper bits to zero

  - Signed magnitude: store magnitude in least significant portion; store sign bit in new sign bit position (probably MSB); set remaining bits to zero

  - Complemented: store original bits in least significant portion; set remaining bits to the sign bit (sign extension)

Sign Extension Examples



Overflow

- Occurs when the result of an operation cannot be represented with available hardware
    - Example: multiply two positive numbers, but resulting sign bit is negative
- For addition, detection depends upon integer representation:
    - For unsigned operands, if carry-out bit is 1
    - For two's complement, if sign bit does not match operands (two positive addends yield a negative, or two negative addends yield a positive)
- Hardware can have overflow detection to set the overflow condition bit

8-bit Overflow Examples

Unsigned Integer

	1	1	1	1	1	1	1	0
	1	1	1	0	1	0	1	0 = 234_{10}
+	0	0	1	1	0	1	1	0 = 54_{10}
	0	0	1	0	0	0	0	0 = 32_{10}

Because carry-out bit is 1, overflow is detected

Two's Complement

	0	1	1	1	1	1	1	0
	0	1	1	0	1	0	1	0 = 106_{10}
+	0	0	1	1	0	1	1	0 = 54_{10}
	1	0	1	0	0	0	0	0 = -96_{10}

Both operands are positive, but resulting sign is negative

	1	1	1	1	1	1	1	0
	1	1	1	0	1	0	1	0 = -22_{10}
+	0	0	1	1	0	1	1	0 = 54_{10}
	0	0	1	0	0	0	0	0 = 32_{10}

Unlike unsigned arithmetic, no overflow here even though carry-out bit is 1. Because operands' signs differ, no overflow is possible

C Basic Data Types

- An **`int`** (synonymous with **`signed int`** or just **`signed`**) is defined as a **signed** integer value of *at least* 16 bits
    - Technically does not require two's complement implementation
- An **`unsigned int`** (or **`unsigned`**) is an **unsigned** integer value of at least 16 bits
- A **`signed char`** is defined as a **signed** integer value of at least 8 bits
    - Again, technically does not require a two's complement implementation
- An **`unsigned char`** is an **unsigned** integer value of at least 8 bits
- A **`char`** is implementation defined, and may be **either** signed or unsigned

C Integer Conversions

- Operands are automatically promoted to an **`int`** type when possible, **`unsigned int`** if not

- A cast means to reinterpret underlying bits differently (such as treat bits as unsigned versus two's complement)

```c
#include <stdio.h>

int main(void) {
    signed char a = -1;
    unsigned int b = a;
    unsigned char c = 0xff;
    signed char d = 0xff;
    printf("a = %d, b = %u\n", a, b);
    printf("c = %08x, d = %08x\n", c, d);
    return 0;
}
```

```
a = -1, b = 4294967295
c = 000000ff, d = ffffffff
```

http://www.idryman.org/blog/2012/11/21/integer-promotion/

Pointer Casts

- In most cases, casting a C pointer leads to very subtle errors
  - Pointer casts should be looked upon with suspicion

```c
#include <stdio.h>

void print_vals(int *val1, int *val2) {
    printf("val1=%08x, val2=%08x\n", *val1, *val2);
}

int main(void) {
    char vals[] = {0x10, 0x20, 0x30, 0x40};
    print_vals(&vals[0], &vals[2]);
    return 0;
}
```

```
val1=40302010, val2=00004030
```