Data representation

College of Saint Benedict & Saint John's University

decimal refresher

58036

decimal refresher

5		8		0		3		6
50000	+	8000	+	0	+	30	+	6

decimal refresher

5		8		0		3		6
50000	$+$	8000	$+$	0	$+$	30	$+$	6
5×10000	$+$	8×1000	$+$	0×100	$+$	3×10	$+$	6×1

decimal refresher

5		8		0		3		6
50000	$+$	8000	$+$	0	$+$	30	$+$	6
5×10000	$+$	8×1000	$+$	0×100	$+$	3×10	$+$	6×1
5×10^4	$+$	8×10^3	$+$	0×10^2	$+$	3×10^1	$+$	6×10^0

binary refresher

10110

binary refresher

1		0		1		1		0
1×2^4	$+$	0×2^3	$+$	1×2^2	$+$	1×2^1	$+$	0×2^0

binary refresher

1		0		1		1		0
1×2^4	$+$	0×2^3	$+$	1×2^2	$+$	1×2^1	$+$	0×2^0
1×16	$+$	0×8	$+$	1×4	$+$	1×2	$+$	0×1

binary refresher

1		0		1		1		0
1×2^4	$+$	0×2^3	$+$	1×2^2	$+$	1×2^1	$+$	0×2^0
1×16	$+$	0×8	$+$	1×4	$+$	1×2	$+$	0×1
16	$+$	0	$+$	4	$+$	2	$+$	0

- this is the representation for unsigned binary integers
- so how to represent signed integers?
  - why not use the leftmost bit to store the sign?
    - what is the range of values if we choose this? — 0 is represented twice, so our range has one less value — not the end of the world
    - what happens if we add to -5 to +5? — the result is -10?

unsigned addition

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 10$$

	0	0	0	1	0	1	$= 5$
ADD	0	0	0	1	0	1	$= 5$

unsigned addition

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 10$$

	0	0	0	1	0	1	= 5
ADD	0	0	0	1	0	1	= 5
C = 0	0	0	1	0	1	0	= 10

- the hardware has a special bit known as the *carry* bit, denoted by C, which stores a 1 if the result of the addition was a carry, and 0 otherwise.

signed addition

	0	0	0	1	0	1	$= +5$
ADD	1	0	0	1	0	1	$= -5$

signed addition

	0	0	0	1	0	1	$= +5$
ADD	1	0	0	1	0	1	$= -5$
$C = 0$	1	0	1	0	1	0	$= -10$

- what is the problem
  - in this case, we had two additional symbols, $+$ and $-$, and we were making some assumptions about their behavior.
  - For example, we know that $5\text{ADD}5 = 10$, but what does $+\text{ADD}-$ equal? — we have no rule for that in our definition of decimal.
  - we are trying to apply the addition algorithm, when we should be applying a different algorithm, called *subtraction*
  - can we choose a different representation that we can directly use the addition algorithm with?

one’s complement

NOT	0	0		0	1	0	1

one's complement

NOT	0	0	0	1	0	1
	1	1	1	0	1	0

- one's complement is known as logical not

one's complement

NOT	0	0	0	1	0	1
	1	1	1	0	1	0

	0	0	0	1	0	1
ADD	1	1	1	0	1	0

- one's complement is known as logical not

one's complement

NOT	0	0	0	1	0	1
	1	1	1	0	1	0

	0	0	0	1	0	1
ADD	1	1	1	0	1	0
$C = 0$	1	1	1	1	1	1

- one's complement is known as logical not
- adding the one's complement will always result in all 1s

one's complement

NOT	0	0	0	1	0	1
	1	1	1	0	1	0

	0	0	0	1	0	1
ADD	1	1	1	0	1	0
$C = 0$	1	1	1	1	1	1
ADD	0	0	0	0	0	1

- one's complement is known as logical not
- adding the one's complement will always result in all 1s

one's complement

NOT	0	0	0	1	0	1
	1	1	1	0	1	0

	0	0	0	1	0	1
ADD	1	1	1	0	1	0
$C = 0$	1	1	1	1	1	1
ADD	0	0	0	0	0	1
$C = 1$	0	0	0	0	0	0

- one's complement is known as logical not
- adding the one's complement will always result in all 1s
- so two's complement is NOT $+ 1$

cpu bits

	0	1
N	otherwise	result is negative
Z	otherwise	result is all zeros
V	otherwise	signed integer overflow occurred
C	otherwise	unsigned integer overflow occurred

- C is set to carry out of leftmost bit
- V is set to detects an overflow by comparing the carry into the leftmost bit with the C bit. If they are different, an overflow has occurred, and V gets 1. If they are the same, V gets 0.

register transfer language

operation	RTL symbol
AND	$\wedge$
OR	$\vee$
XOR	$\oplus$
NOT	$\neg$
Implies	$\rightarrow$
Transfer	$\leftarrow$
Bit index	$\langle\ \rangle$
Informal description	$\{\ \}$
Sequential separator	;
Concurrent separator	,

register transfer language

operation	RTL symbol
AND	$\wedge$
OR	$\vee$
XOR	$\oplus$
NOT	$\neg$
Implies	$\rightarrow$
Transfer	$\leftarrow$
Bit index	$\langle\ \rangle$
Informal description	$\{\ \}$
Sequential separator	;
Concurrent separator	,

$$c \leftarrow a \oplus b; N \leftarrow c < 0, Z \leftarrow c = 0$$

- XOR

another example

	0	0	0	1	0	1
ADD	1	1	1	0	1	1
N ← 0	0	0	0	0	0	0
Z ← 1						
V ← ?						
C ← 1						

another example

	0	0	0	1	0	1
ADD	1	1	1	0	1	1
$N \leftarrow 0$	0	0	0	0	0	0

$Z \leftarrow 1$

$V \leftarrow \neg(a\langle 0\rangle \oplus b\langle 0\rangle) \wedge (a\langle 0\rangle \oplus N)$

$C \leftarrow 1$

arithmetic shift

arithmetic shift left (asl)

$C \leftarrow r\langle 0\rangle,\ r\langle 0..4\rangle \leftarrow \langle 1..5\rangle,\ r\langle 5\rangle \leftarrow 0;$

$N \leftarrow r < 0,\ Z \leftarrow r = 0,\ V \leftarrow \{\text{overflow}\}$

arithmetic shift right (asr)

?

- how will you assign V bit? — what is the RTL?
- $V \leftarrow C = r\langle 0\rangle$
- what is RTL for ASR?
- where are the N and V bits for ASR?

arithmetic shift

arithmetic shift left (asl)

$C \leftarrow r\langle 0\rangle,\ r\langle 0..4\rangle \leftarrow \langle 1..5\rangle,\ r\langle 5\rangle \leftarrow 0;$

$N \leftarrow r < 0,\ Z \leftarrow r = 0,\ V \leftarrow \{\text{overflow}\}$

arithmetic shift right (asr)

$C \leftarrow r\langle 5\rangle,\ r\langle 1..5\rangle \leftarrow \langle 0..4\rangle;$

$Z \leftarrow r = 0$

- how will you assign V bit? — what is the RTL?
- $V \leftarrow C = r\langle 0\rangle$
- what is RTL for ASR?
- where are the N and V bits for ASR?

hexadecimal

unicode

Hello world.
¡Hola!, Grüß Gott, Hyvää päivää, Tere õhtust, Bonġu Cześć!, Dobrý den
你好, 早晨, こんにちは

- used to access >127 different characters
- backwards compatible with ASCII, i.e., code points have same value in UTF that they have in ASCII
- comes in different flavors
  - UTF-32 — easier to understand, requires 32bits to represent each of the code points (glyphs), but wasteful if you are mostly storing ascii characters
  - UTF-8 — variable width code, i.e., some bits in the pattern are reserved for storing information about the structure of the pattern instead of information about the code point
- also define some emojis

unicode

Hello world.

¡Hola!, Grüß Gott, Hyvää päivää, Tere õhtust, Bonġu Cześć!, Dobrý den

你好, 早晨, こんにちは

https://www.paypal.com

- used to access >127 different characters
- backwards compatible with ASCII, i.e., code points have same value in UTF that they have in ASCII
- comes in different flavors
  - UTF-32 — easier to understand, requires 32bits to represent each of the code points (glyphs), but wasteful if you are mostly storing ascii characters
  - UTF-8 — variable width code, i.e., some bits in the pattern are reserved for storing information about the structure of the pattern instead of information about the code point
- also define some emojis
- can also be used for nefarious things

unicode

Hello world.
¡Hola!, Grüß Gott, Hyvää päivää, Tere õhtust, Bonġu Cześć!, Dobrý den
你好, 早晨, こんにちは

https://www.paypal.com

- used to access >127 different characters
- backwards compatible with ASCII, i.e., code points have same value in UTF that they have in ASCII
- comes in different flavors
  - UTF-32 — easier to understand, requires 32bits to represent each of the code points (glyphs), but wasteful if you are mostly storing ascii characters
  - UTF-8 — variable width code, i.e., some bits in the pattern are reserved for storing information about the structure of the pattern instead of information about the code point
- also define some emojis
- can also be used for nefarious things

floating-point

IEEE 754

single precision 1.8.23 — excess 127 / 126

double precision 1.11.52 — excess 1023 / 1022

special values

	exponent	significand
zero	all zeros	all zeros
denormalized	all zeros	non-zero
inifinity	all ones	all zeros
not a number (NaN)	all ones	non-zero

operations that result in NaN

- The divisions 0/0 and $\pm\infty / \pm\infty$
- The multiplications $0 \times \pm\infty$ and $\pm\infty \times 0$
- The additions $\infty + (-\infty)$, $(-\infty) + \infty$ and equivalent subtractions

- mantissa → significand
- more bits in exponent → more range
- more bits in significand → more precision
- how to find the range for floating-point numbers

floating-point

IEEE 754

single precision 1.8.23 — excess 127 / 126
double precision 1.11.52 — excess 1023 / 1022
special values

	exponent	significand
zero	all zeros	all zeros
denormalized	all zeros	non-zero
inifinity	all ones	all zeros
not a number (NaN)	all ones	non-zero

operations that result in NaN

- The divisions $0/0$ and $\pm\infty / \pm\infty$
- The multiplications $0 \times \pm\infty$ and $\pm\infty \times 0$
- The additions $\infty + (-\infty)$, $(-\infty) + \infty$ and equivalent subtractions

- mantissa → significand
- more bits in exponent → more range
- more bits in significand → more precision
- how to find the range for floating-point numbers
- draw attention to Figure 3.38 — make connection with radix sort — floats can be sorted using radix sort by NOT'ing the values then treating them like unsigned integers