CAS CS 210 : COMPUTER SYSTEMS FUNDAMENTALS OF REPRESENTING AND MANIPULATING INFORMATION I

Professor: Appavoo

BITS AND NUMBER SYSTEMS

INTERPRETATION

A specific method or rule for using the vectors to represent information and operations

Eg. as an index into a table of symbols
eg. english characters

binary vector	character
[000]	'A'
[001]	'B'
[010]	'C'
[011]	'D'
[100]	'E'
[101]	'F'
[110]	'G'
[111]	'H'

INTERPRETATION

A specific method or rule for using the vectors to represent information and operations

Eg. as an n bit integer binary numbers

$$\sum_{i=0}^{n} b_i 2^i$$

imposes an order and operations (+,-,/,*) on the vectors

binary vector	binary number	decimal
[000]	000.	0
[001]	001.	1
[010]	010.	2
[011]	011.	3
[100]	100.	4
[101]	101.	5
[110]	110.	6
[111]	111.	7

0 … 2^{n-1} : 2^n

REMINDER OF NUMBER SYSTEMS

GENERALIZED BINARY POSITIONAL NUMBER SYSTEM

most significant ← $\ldots b_2 b_1 b_0 . b_{-1} b_{-2} b_{-3} \ldots$ → least significant

binary point

where b is a base 2 digit — 0 or 1
integers — positive power to left of the binary point
fractions — negative powers to right of binary point

most significant ← $\ldots b_2 b_1 b_0 . b_{-1} b_{-2} b_{-3} \ldots$ → least significant

binary point

convert to a decimal number as a sum of powers of 2

$$\ldots + b_1 * 2^1 + b_0 * 2^0 + b_{-1} * 2^{-1} + b_{-2} * 2^{-2} + \ldots$$

$$\sum_i b_i 2^i$$

1101.011

$1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 \; . \; 0*2^{-1} + 1*2^{-2} + 1*2^{-3}$

1*8 + 1*4 + 0*2 + 1*1 . 0*1/2 + 1*1/4 + 1*1/8

8 + 4 + 0 + 1 . 0 + 1/4 + 1/8

13 . 0.25 + 0.125

13.375

---

most significant ← $\ldots d_2 d_1 d_0 . d_{-1} d_{-2} d_{-3} \ldots$ → least significant

decimal point

convert to a binary number is kind of gross ;-)

- Use "remainder method" for the integer portion
- Use "multiplication method" for the fraction
- Example: convert 11.25 from base 10 to 2

- convert **11.25** from base 10 to 2

remainder method

```
11 / 2 = 5 * 2 + 1
 5 / 2 = 2 * 2 + 1
 2 / 2 = 1 * 2 + 0
 1 / 2 = 0 * 2 + 1
```

1011

multiplication method

```
.25 * 2 = 0.5
 .5 * 2 = 1.0
```

.01

1011.01

BASE 2, 10, 16 NUMBER SYSTEMS

- Binary (base 2):
  - 0000, 0001, 0010, ..., 1001, 1010, ..., 1111
- Decimal (base 10):
  - 0, 1, 2, ..., 9, 10, ..., 15
- Hexadecimal (base 16):
  - 0, 1, 2, ..., 9, A, ..., F
  - In C, 0xFA1D5, printf("%x", i)
- Conversion among power-of-2 bases is simple
  - Example: convert 01101101 from base 2 to 16

INFORMATION STORAGE : MEMORY (2.1)

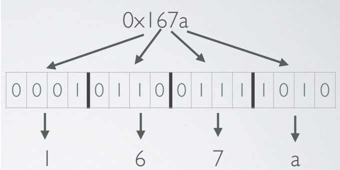
- Byte: basic unit of bits 8 bits: 2^8 possible patterns
- Machine level program view : virtual array of bytes: M[a]
- a: addresses
- pointers: address and type : provides interpretation for a set of bytes at a given address

HEXADECIMAL NOTATION (2.1.1)

- binary values in base 2 are tedious: 10001010
- Base 10 not convenient : 0 - 255
  - Conversion among power-of-2 bases is simple
- Base 16 concise and easy to translate
  - Hex Digit has 16 possible values form 0 to F: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
  - 4 bits represent 16 unique numbers from 0 - 15 — one hex digit
  - To convert simply work in groups of 4 bits. Padded left with zero's as necessary. 1 Byte value is represented as 2 Hex Digits.

HEX

Binary	Hex	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15



- Worth remembering that positive power's of two convert simply: if $x=2^n$ then binary 1 followed by **n** zeros and thus convert to hex easily too: eg.

  2^{16} =1 0000 0000 0000 0000b = 0x10000

- but the general case of conversion from decimal to hex requires remainder method with division by 16 to find quotients and remainders

x = q * 16 + r eg.

```
1227 = 76 * 16 + 11 -> (B)
  76 =  4 * 16 + 12 -> (C)
   4 =  0 * 16 +  4 -> (4)
     = 0x4CB
```

- In 'C' constants that are prefixed with 0x are hex values eg.

```
unsigned int x = 0x10,  y = 16;
printf("x=%d y=%d\n", x ,y);
```

WORDS (2.1.2)

- Computers have a word size, ***w*** bits. Were w bits is the natural type that the system can natively operate on/ manipulate.
  - ***w*** bits : 2^w values ranging from : 0 - (2^w-1)
- pointer/addresses are word size -> what does this mean:
  - virtual address size is limited to 2^w
  - machine can efficiently represent and operate on values that range from 0 - (2^w-1)

What are the common values of w today?
Is 4 GB (gigabytes) = 2^{32} enough?

'C' : DATA SIZES & POINTERS (2.1.3)

Sizes (in bytes) of C numeric data types

C Declaration	32-bit	64-bit
char	1	1
short int	2	2
int	4	4
long int	4	8
long long int	8	8
char *	4	8
float	4	4
double	8	8
void *	4	8

Pointers combine address and type to provide an **exact interpretation** for the values of bytes at a particular address.

T *p;
p is a pointer to an object of type T
eg. :

```
int *iptr;
char *cptr;
```

ADDRESSING AND BYTE ORDERING (2.1.4)

- Multibyte object stored in contiguous sequence of bytes with address of object the smallest address of the bytes used
- **ENDIANESS:** Two common choices for ordering bytes of a multibyte object big endian (IBM 360) vs little endian (Intel x86). Bi-endian (ARM, PowerPC)

int x = 0x01234567; // assume &x = 0x100

	0x100	0x101	0x102	0x103
little	0x67	0x45	0x23	0x01
big	0x01	0x23	0x45	0x67

Network code, Memory dumps, and Advanced/Systems Programming

FIGURE 2.4 AND 2.5

- Playing with this code and understanding it in detail will pay dividends

```c
#include <stdio.h>

typedef unsigned char *byte_pointer;

void show_bytes(byte_pointer start, int len) {
  int i;
  for(i=0; i<len; i++)
    printf(" %.2x", start[i]);
  printf("\n");
}

int main(void) {
  short x = 12345;
  short mx = -x;
  unsigned short ux = (unsigned short)x;
  unsigned short umx = (unsigned short)mx;

  show_bytes((byte_pointer) &x, (sizeof(short)));
  show_bytes((byte_pointer) &mx, (sizeof(short)));
  show_bytes((byte_pointer) &ux, (sizeof(unsigned short)));
  show_bytes((byte_pointer) &umx, (sizeof(unsigned short)));
  return 1;
}
```

```
bash-3.2$ gcc code1.c -o c1
bash-3.2$ ./c1
 39 30
 c7 cf
 39 30
 c7 cf
bash-3.2$
```

0x3039	0011 0000 0011 1001
0xCFC7	1100 1111 1100 0111

Why are the bytes reordered?

PRINTF IS YOUR FRIEND GET TO KNOW IT

REPRESENTING STRINGS (2.1.5)

ASCII: Standard encoding of English characters, punctuation, and some special characters into byte values.

String a sequence of ASCII Byte Values with a final Byte that has a 0 value to indicate the end of the string.

int i=15; // 0x0000000F -> 0x0F 0x00 0x00 0x00
char str[] = "bugs"; // ???

ASCII

Lower Nibble

High Nibble

ASCII Code Chart

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

http://en.wikipedia.org/wiki/File:ASCII_Code_Chart.svg

REPRESENTING CODE (2.1.6)

Hardware dependent encoding of machine's operations into byte and multibyte values.

Stored Program seems obvious but was a big deal!

It also means that programs can be treated as data and programs can generate programs on the fly.

We can have pointers to instructions sequences: C function pointers!

BOOLEAN ALGEBRA (2.1.7)

ALGEBRA OF TRUTH=1 AND FALSE=0

~ NOT: ~X=Y

X	Y
0	1
1	0

& AND: X & Y = Z

X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

| OR: X | Y = Z

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

Primitives for working with raw bit patterns

These are your building blocks!

^ XOR: X^Y = Z

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	0

C TWO KINDS OF BOOLEAN OPERATIONS (2.1.8, 2.1.9)

- BITWISE
- ~, |, &, ^
- operate on vector of bits
- result is a vector of bits

- LOGICAL
- !, &&, ||
- 1=TRUE = X != 0 and 0=FALSE = X == 0
- operates on integral types first map then apply boolean operation to produce 0|1
- Conditional evaluation

BIT OPERATIONS (2.1.8)

'C' Expression	Binary Expression	Binary Result	Hex Result
~0x41	~[0100 0001]	[1011 1110]	0xBE
~0x00	~[0000 0000]	[1111 1111]	0xFF
0x69 & 0x55	[0110 1001] & [0101 0101]	[0100 0001]	0x41
0x69 | 0x55	[0110 1001] | [0101 0101]	[0111 1101]	0x7D

BIT VECTORS AND SETS

- Bit level operations to maintain and manipulate sets: | is union (A|B is union of A and B) and & is intersections (A&B is intersection of A and B)
- Each bit position is represents the presence of an element
- Low level programming power and interfacing to machine hardware is all about bit level manipulation.
- MASKING : a mask identifies a particular signals by having ones in the right position:

X	MASK	X & MASK
0x8BADF00D	0xFF	0x0D
0x8BADF00D	0x000F000F	0x000D000D

SHIFT OPERATORS

- left shift : x << k : where 0 <= k <= n-1: x is shifted k bits to the left, dropping off the k most significant bits and filling the right end with k zeros.
- right shift : x >> k : 2 types logical and arithmetic:
  - logical right shift: left end filled with k zeros
  - arithmetic right shift: left end filled with k repetitions of most significant bit.

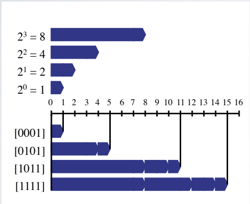
	x=[01100011]	x=[10010101]
x<<4	[0011***0000***]	[0101***0000***]
x>>4 (logical)	[***0000***0110]	[***0000***1001]
x>>4 (arithmetic)	[***0000***0110]	[***1111***1001]

LOGICAL OPERATORS

'C' Expression	Binary Expression	Binary Result	Hex Result
!0x41	![0100 0001]	[0000 0000]	0x00
!0x00	![0000 0000]	[0000 0001]	0x01
0x69 && 0x55	[0110 1001] && [0101 0101]	[0000 0001]	0x01
0x69 || 0x55	[0110 1001] || [0101 0101]	[0000 0001]	0x01
0x69 && (!0x55)	[0110 1001] && (![0101 0101])	[0000 0000]	0x00

UNSIGNED INTEGERS (2.2.2)

Encode a bit vector of length w "efficiently" into positive integers



$\vec{x}$ $[x_{w-1}, x_{w-2}, ..., x_0]$

$$B2U_w(\vec{x}) \doteq \sum_{i=0}^{w-1} x_i 2^i$$

$$B2U_w : \{0,1\}^w \rightarrow \{\texttt{Umin}, ..., \texttt{Umax}\}$$
$$\rightarrow \{0, ..., 2^w - 1\}$$

$\texttt{UMin}_w(\vec{x}) \doteq 0$ $\texttt{UMax}_w(\vec{x}) \doteq \sum_{i=0}^{w-1} 2^i = 2^w - 1$

UNSIGNED ADDITION

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

UNSIGNED ADDITION

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

UNSIGNED ADDITION

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

UNSIGNED ADDITION

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

UNSIGNED ADDITION

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

WHAT ABOUT NEGATIVE NUMBERS?

- How do we work with signed integers?
- What do we have at our disposal?
- What kind of properties would we like?

ALTERNATIVES

```
Unsigned   Sign Magnitude
   000        000 = +0
   001        001 = +1
   010        010 = +2
   011        011 = +3
   100        100 = -0
   101        101 = -1
   110        110 = -2
   111        111 = -3
```

ALTERNATIVES

```
Unsigned   Sign Magnitude   One's Comp.
   000        000 = +0       000 = +0
   001        001 = +1       001 = +1
   010        010 = +2       010 = +2
   011        011 = +3       011 = +3
   100        100 = -0       100 = -3
   101        101 = -1       101 = -2
   110        110 = -2       110 = -1
   111        111 = -3       111 = -0
```

ALTERNATIVES

Unsigned	Sign Magnitude	One's Comp.	Two's Comp.
000	000 = +0	000 = +0	000 = +0
001	001 = +1	001 = +1	001 = +1
010	010 = +2	010 = +2	010 = +2
011	011 = +3	011 = +3	011 = +3
100	100 = -0	100 = -3	100 = -4
101	101 = -1	101 = -2	101 = -3
110	110 = -2	110 = -1	110 = -2
111	111 = -3	111 = -0	111 = -1

WHICH ONE IS BEST? WHY?

- Issues: order, number of zeros, ease of operations
- Problems with SM and 1's complement:
  - two representations for zero
  - addition does not just work:

SM: 1 + -1 1's complement: 1 + -1

WHICH ONE IS BEST? WHY?

- Issues: order, number of zeros, ease of operations
- Problems with SM and 1's complement:
  - two representations for zero
  - addition does not just work:

SM: 1 + -1 1's complement: 1 + -1


```
SM:            1's complement:
  001            001
  101            110
  ---            ---
```

WHICH ONE IS BEST? WHY?

- Issues: order, number of zeros, ease of operations
- Problems with SM and 1's complement:
  - two representations for zero
  - addition does not just work:

SM: 1 + -1

```
 001
 101
----
 110
```

1's complement: 1 + -1

```
 001
 110
----
 111
```

WHICH ONE IS BEST? WHY?

- Issues: order, number of zeros, ease of operations
- Problems with SM and 1's complement:
  - two representations for zero
  - addition does not just work:

SM: 1 + -1

```
 001
 101
----
 110
```


1 + -1=-2???

1's complement: 1 + -1

```
 001
 110
----
 111
```


1 + -1=-0 close but still weird

2's Complement:

To obtain negative of a number flip the bits and add 1

$$-x = \sim x + 1$$

MATHEMATICAL DEFINITION

$$B2T_w(\vec{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

So that half the patterns represent

1. 0
2. 1<=x<2^{w-1}

and the other half map to

3. -2^{w-1}
4. (~x + 1)for1<=x<2^{w-1}

X	B2U(X)	B2T(X)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

~x + 1 mapping

SIGNED INTEGERS (2.2.3)

$$B2T_w(\vec{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

sign bit

negative weight $\{0, -2^{w-1}\}$

a signed magnitude of w-1 bits $(0, 2^{w-1} - 1)$

$x_{w-1} = 0$	$x_{w-1} = 1$
$B2T([0,0,...,0]) = 0$	$B2T([1,0,...,0]) = -2^{w-1}$ **Tmin$_w$**
$B2T([0,1,...,1]) = 2^{w-1} - 1$ **Tmax$_w$**	$B2T([1,1,...,1]) = -2^{w-1} + 2^{w-1} - 1 = -1$

IMPORTANT NUMBERS

C data type	8	16	32	64
UMax	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFFFF
	255	65,535	4,294,967,295	18,446,744,073,709,551,615
Tmin	0x80	0x8000	0x80000000	0x8000000000000000
	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808
TMax	0x7F	0x7FFF	0x7FFFFFFF	0x7FFFFFFFFFFFFFFF
	127	32,767	2,147,483,647	9,223,372,036,854,775,807
-1	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFFFF
0	0	0x00	0x0000	0x0000000000000000

'C' standard does not specify two's comp

C data type	min	max
char	-127	127
unsigned char	0	255
short	-32,767	32,767
unsigned short	0	65,535
int	-32,767	32,767
unsigned	0	65,535
long	-2,147,483,647	2,147,483,647
unsigned long	0	4,294,967,295
long long	-((2^64) -1)/2	((2^64)-1)/2
unsigned long long	0	(2^64)-1

however, 'typical' 8, 16, 32 and 64 bit two complement numbers can be expected <limits.h>

"CONVERSIONS" BETWEEN SIGNED AND UNSIGNED

```c
#include <stdio.h>

int main(void) {
  short int v = -12345;
  unsigned short uv = (unsigned short) v;
  printf("v = %d, uv = %u\n", v, uv);
  return 1;
}
```

```
bash-3.2$ gcc -m32 code2.c -o c2
bash-3.2$ ./c2
v = -12345, uv = 53191
bash-3.2$
```

```c
#include <stdio.h>

int main(void) {
  unsigned  u = 4294967295u;
  int       tu = (int) u;
  printf("u = %u, tu = %d\n", u, tu);
  return 1;
}
```

```
bash-3.2$ gcc -m32 code3.c -o c3
bash-3.2$ ./c3
u = 4294967295, tu = -1
bash-3.2$
```

"CONVERSIONS" BETWEEN SIGNED AND UNSIGNED

```c
#include <stdio.h>

int main(void) {
  short int v = -12345;
  unsigned short uv = (unsigned short) v;
  printf("v = %d, uv = %u\n", v, uv);
  return 1;
}
```

```
bash-3.2$ gcc -m32 code2.c -o c2
bash-3.2$ ./c2
v = -12345, uv = 53191
bash-3.2$
```



```
v = -12345 0xcfc7, uv = 53191 0xcfc7
```



```c
#include <stdio.h>

int main(void) {
  unsigned  u = 4294967295u;
  int       tu = (int) u;
  printf("u = %u, tu = %d\n", u, tu);
  return 1;
}
```

```
bash-3.2$ gcc -m32 code3.c -o c3
bash-3.2$ ./c3
u = 4294967295, tu = -1
bash-3.2$
```

"CONVERSIONS" BETWEEN SIGNED AND UNSIGNED

```c
#include <stdio.h>

int main(void) {
  short int v = -12345;
  unsigned short uv = (unsigned short) v;
  printf("v = %d, uv = %u\n", v, uv);
  return 1;
}
```

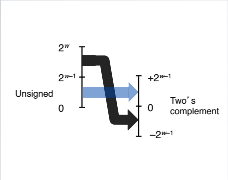
```
bash-3.2$ gcc -m32 code2.c -o c2
bash-3.2$ ./c2
v = -12345, uv = 53191
bash-3.2$
```

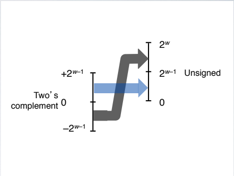
```
v = -12345 0xcfc7, uv = 53191 0xcfc7
```

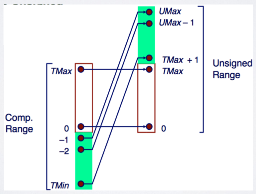
```c
#include <stdio.h>

int main(void) {
  unsigned  u = 4294967295u;
  int       tu = (int) u;
  printf("u = %u, tu = %d\n", u, tu);
  return 1;
}
```

```
bash-3.2$ gcc -m32 code3.c -o c3
bash-3.2$ ./c3
u = 4294967295, tu = -1
bash-3.2$
```

```
u = 4294967295 0xffffffff, tu = -1 0xffffffff
```







SIGNED VS UNSIGNED IN 'C'

When either operand of a comparison is unsigned, the other operand is implicitly cast to unsigned.

Expression	Type	Evaluation
0 == 0U	unsigned	1
-1 < 0	signed	1
-1< 0U	unsigned	0*
2147483647>-2147483647-1	signed	1
2147483647U>-2147483647-1	unsigned	0*
2147483647>(int)2147483648U	signed	1*
-1 > -2	signed	1
(unsigned) -1 > -2	unsigned	1

EXPANSION (2.2.6)

zero extension

expanding unsigned representation by simply adding leading zero's as needed

signed extension

expanding signed representation by replicating sign bit as needed

```
char c = x;
int i = c;
printf("%d %d\n", c, i);
show_bytes((byte_pointer)&c, sizeof(c));
show_bytes((byte_pointer)&i, sizeof(i));
```

These rules result in what you expect

2.2.6 presents the simple proof based using induction on the definitions

THERE ARE STILL THINGS TO BE CAREFUL OF

default is to first do size expansion and then do assignment

```
short sx = -12345;
unsigned uy = sx;   // (unsigned) (int) sx;
// unsigned uy = (unsigned) (unsigned short) sx;

printf("uy = %u:\t", uy);
show_bytes((byte_pointer)&uy, sizeof(unsigned));
```

This is not the same as doing the unsigned interpretation and then the expansion

The second preserves the bit representation vs the first will sign extend and then use this as the pattern for the assignment

TRUNCATION (2.2.7)

truncating **w** bit number is means we drop the high order **w-k** bits

$$[x_{w-1}, x_{w-2}, ..., x_0] \rightarrow [x_{k-1}, x_{k-2}, ..., x_0]$$

```
int x = 53191;
short sx = (short) x;
int y = sx;
printf("%d %d %d\n", x, sx, y);
```

```
bash-3.2$ gcc trunc.c -o t
bash-3.2$ ./t
53191 -12345 -12345
```

Important relationship!

Truncating x to k bits is equivalent to x mod 2^k

$$[x_{w-1}, x_{w-2}, ..., x_k, x_{k-1}, x_{k-2}, ..., x_0]$$

remainder of a division by 2^k

2 COMPLEMENT SUMMARY

Unsigned & Signed Numeric Values

X	B2U(X)	B2T(X)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

Equivalence

- Same encodings for nonnegative values

Uniqueness

- Every bit pattern represents unique integer value
- Each representable integer has unique bit encoding

binary w length bit vector

$$\overrightarrow{x} \quad [x_{w-1}, x_{w-2}, ..., x_0]$$

binary to unsigned int

$$B2U_w(\overrightarrow{x}) \doteq \sum_{i=0}^{w-1} x_i 2^i$$

binary to signed int

$$B2T_w(\overrightarrow{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

'conversion' in C is reinterpret binary vector

$$T2U_w(x) \doteq B2U_w(T2B_w(x))$$

$$U2T_w(x) \doteq B2T_w(U2B_w(x))$$

relationships

$$T2U_w(x) = \begin{cases} x + 2^w & x < 0 \\ x & x \geq 0 \end{cases}$$

$$U2T_w(u) = \begin{cases} u & u < 2^{w-1} \\ u - 2^w & u \geq 2^{w-1} \end{cases}$$

Know critical numbers and Rules for Expanding and Truncating bit vectors for both unsigned and signed