Integer Computer Arithmetic

Computer Systems Sections 2.2, 2.3

Abstraction

Computer Integers behave like mathematical integers

Infinite & Ordered ($x < x+1$)

Associative $(x + y) + z = x + (y + z)$

Distributive $x*(y + z) = x*y + x*z$

Commutative $x+y = y+x$, $x*y=y*x$

Leaky Abstraction

Computer Integers behave like mathematical integers but…

Not infinite & ordered ($x < x+1$) e.g. (char) 127+1 = -128

Associative $(x + y) + z = x + (y + z)$

Distributive $x*(y + z) = x*y + x*z$

Commutative $x+y = y+x$, $x*y=y*x$

Decimal, Binary, and Hexadecimal Bases

Dec	Bin	Hex
0	0b0000	0x0
1	0b0001	0x1
2	0b0010	0x2
3	0b0011	0x3
4	0b0100	0x4
5	0b0101	0x5
6	0b0110	0x6
7	0b0111	0x7

Dec	Bin	Hex
8	0b1000	0x8
9	0b1001	0x9
10	0b1010	0xA
11	0b1011	0xB
12	0b1100	0xC
13	0b1101	0xD
14	0b1110	0xE
15	0b1111	0xF

Non-Decimal Bases

d_n	d_{n-1}	...	d_1	d_0

$$value = \sum_{i=0}^{\#digits} d_i \times b^i, 0 \leq d_i < b$$

$123 = 1 \text{ x } 10^2 + 2 \text{ x } 10^1 + 3 \text{ x } 10^0$

$0b0111\ 1011 = 2^6+2^5+2^4+2^3+2^1+2^0 = 64+32+16+8+2+1 = 123$

$0x7B = 7x16 + 11 = 112 + 11 = 123$

Notation

- A decimal number will be represented by decimal digits
  - $327 = 3 \text{ x } 100 + 2 \text{ x } 10 + 7 = 3 \text{ x } 10^2 + 2 \text{ x } 10^1 + 7 \text{ x } 10^0$
- A binary number will be prefixed by 0b, followed by 0/1
  - $0b0110 = 1 \text{ x } 4 + 1 \text{ x } 2 = 1 \text{ x } 2^2 + 1 \text{ x } 2^1 = 6$
- A hexadecimal number will be prefix by 0x, followed by hex digits
  - Hex digits are 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
  - $0x01AC = 1 \text{ x } 256 + 10 \text{ x } 16 + 12 = 1 \text{ x } 16^2 + 10 \text{ x } 16^1 + 12 \text{ x } 16^0 = 428$

Computer Unsigned Integer Arithmetic

- Represent a number as a vector of bits (binary data)

$$value = \sum_{i=0}^{n-1} B_i \times 2^i$$

B_{n-1}	B_{n-2}					B_1	B_0
0	0	0	1	0	1	0	1

$$0b0001\ 0101 = 0x15 = 1 \text{ x } 2^4 + 1 \text{ x } 2^2 + 1 \text{ x } 2^0 = 16 + 4 + 1 = 21$$

How Many Bits in an Integer in C?

Data Type	*n*	Max Value [2]
char	8[1]	UCHAR_MAX
short int	≥16	USHRT_MAX
int	≥16 [3]	UINT_MAX
long int	≥32	ULONG_MAX
long long int	≥64	ULLONG_MAX

[1] Typical value assuming addressable unit is 8 bits. Use "CHAR_BIT" in <limits.h> to get size of character.
[2] Constant defined when you "#include <limits.h>"
[3] Usually 32

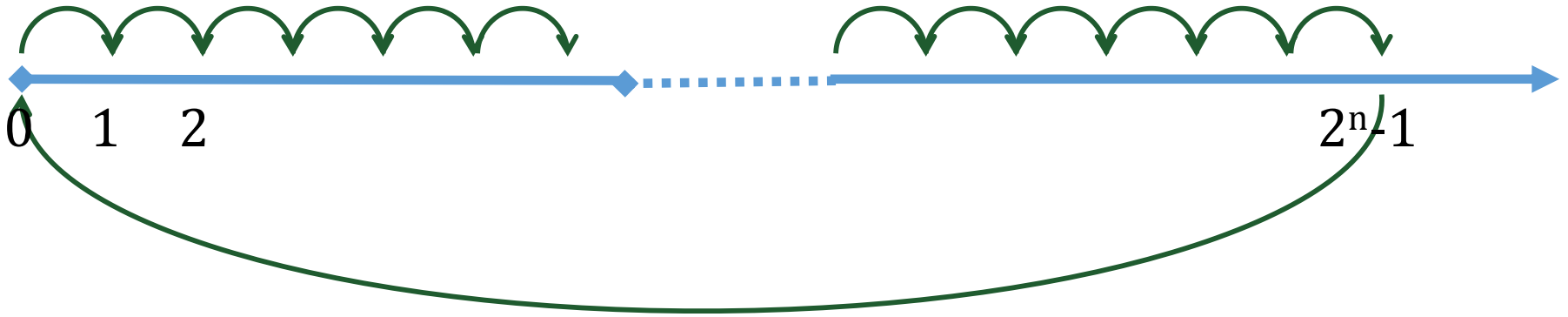
See http://en.wikipedia.org/wiki/C_data_types

What happens on unsigned overflow?

- Overflowing bits ignored

(uchar) 0xff + (uchar) 0x01 = 0x100 = (uchar) 0x00
(uchar) 255 + (uchar) 1 = 256 = (uchar) 0

- See xmp_counting/count.c



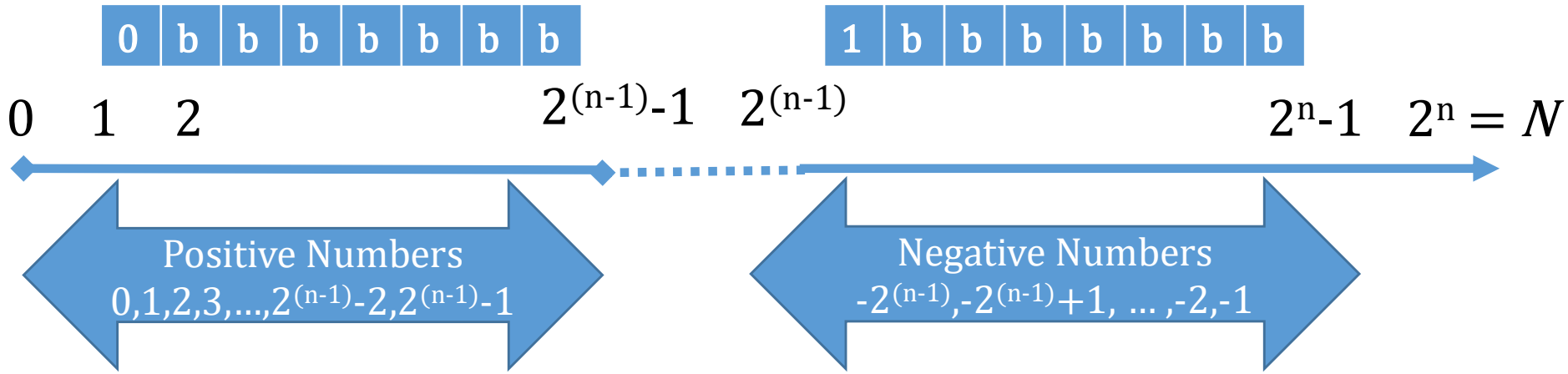
Unsigned: 8 bits

Hex	0x00	0x01	0x02	…	0x7E	0x7F	0x80	0x81	…	0xFE	0xFF
Binary	0000 0000	0000 0001	0000 0010		0111 1110	0111 1111	1000 0000	1000 0001		1111 1110	1111 1111
Unsigned Value	0	1	2	…	126	127	128	129	…	254	255

Computer Representation of Negatives

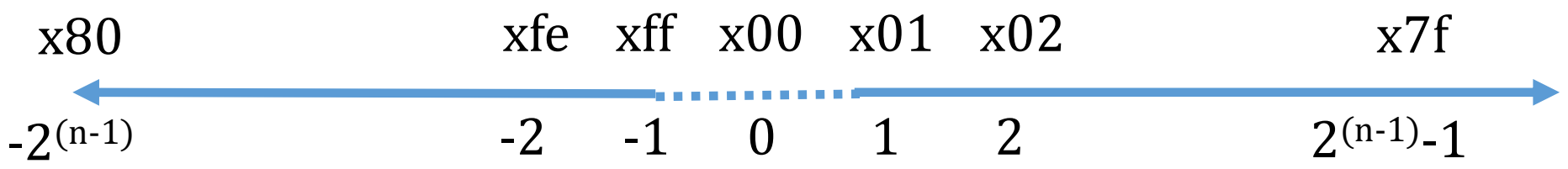
- Vector of 1's and 0's can't have "-"
- Trick : Subtract from N
  - N is bigger than any number you can represent
  - If you have n bits for a number, $N=2^n$
  - Given $x < 0$, represent x using $N - |x|$ or $N - (-x)$

BITS



Signed Integers

- Represented as "Two's Complement" Numbers in n bits
- Positive numbers from 1 to $2^{(n-1)} - 1$: $\sout{value}bits = \sum_{i=0}^{n-2} B_i \times 2^i$
- Negative numbers from -1 to $-2^{(n-1)}$: $\sout{value}bits = 2^n - \left(\sum_{i=0}^{n-1} B_i \times 2^i\right)$
- Negative numbers have a "1" in the high order bit
- -1 is 0xffff...ff
- Shortcut… $-x$ = "Flip the bits and add 1"



Relationship between *bits* and *value*

- For $value >= 0$
  - $bits = \sum_{i=0}^{n-1} B_i \times 2^i = value$
- For $value < 0$
  - $bits = 2^n - |value|$
  - $bits = 2^n - (-value) \quad -value = \sum_{i=0}^{n-1} B_i \times 2^i$
  - $bits = 2^n - \sum_{i=0}^{n-1} B_i \times 2^i$
  - $bits = (2^n - 1) - \left(\sum_{i=0}^{n-1} B_i \times 2^i\right) + 1$
  - $value = 2^n - bits$

Two's Complement : 8 bits $N=2^8 = 256$

BITS

Value

Hex	0x80	0x81		0xFE	Value 0xFF	0x00	0x01	0x02	…	0x7E	0x7F
Binary	1000 0000	1000 0001		1111 1110	1111 1111	0000 0000	0000 0001	0000 0010		0111 1110	0111 1111
Unsigned Value	128	129	…	254	255	0	1	2	…	126	127
256-UVal	-128	-127	…	-2	-1						
Signed Value	-128	-127	…	-2	-1	0	1	2	…	126	127

Why does the shortcut work?

- ~~*value*~~$bits = 2^n - \left(\sum_{i=0}^{n-1} B_i \times 2^i\right)$ (Assuming -)
- $2^n - (-bits) = \left(\sum_{i=0}^{n-1} B_i \times 2^i\right)$
- $(2^n - 1) - (-value) + 1 = \left(\sum_{i=0}^{n-1} B_i \times 2^i\right)$
- So, for example, for signed char (8 bits) value of -3…

Flip the Bits

2^n-1

-*value*

+1

-*value, flipped*

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

What does "Flip the bits" do?

- Same as binary subtraction from a vector of all 1's

	1	1	1	1	1	1	1	1
-	0	0	1	0	0	0	1	1
	1	1	0	1	1	1	0	0

- Vector of all 1's $= \left(\sum_{i=0}^{n-1} 2^i\right) = 2^n - 1$

- So, "flip the bits" is really $2^n - 1 - value = 2^n - value - 1$

Why does the shortcut work?

- For $value >= 0, bits = \sum_{i=0}^{n-1} B_i \times 2^i = value$
  - $-value = 2^n - \sum_{i=0}^{n-1} B_i \times 2^i = (2^n - 1) - bits + 1$

Flip the Bits

+1

- For $value < 0, bits = 2^n - \sum_{i=0}^{n-1} B_i \times 2^i$ where $-value = \sum_{i=0}^{n-1} B_i \times 2^i$
  - $-value = 2^n - 2^n + \sum_{i=0}^{n-1} B_i \times 2^i$
  - $-value = 2^n - \left(2^n - \sum_{i=0}^{n-1} B_i \times 2^i\right) = (2^n - 1) - bits + 1$

+1

Flip the Bits

How Many Bits in an Integer?

Data Type	n	Min Signed [2]	Max Signed [2]	Max Unsigned [2]
char	8[1]	SCHAR_MIN	SCHAR_MAX	UCHAR_MAX
short int	≥16	SHRT_MIN	SHRT_MAX	USHRT_MAX
int	≥16[3]	INT_MIN	INT_MAX	UINT_MAX
long int	≥32	LONG_MIN	LONG_MAX	ULONG_MAX
long long int	≥64	LLONG_MIN	LLONG_MAX	ULLONG_MAX

[1] Typical value assuming addressable unit is 8 bits. Use "CHAR_BIT" in <limits.h> to get size of character.
[2] Constant defined when you "#include <limits.h>"
[3] Usually 16, often 32

See xmp_limits/limits.c
See http://en.wikipedia.org/wiki/C_data_types

What happens on signed overflow?

- Overflows into sign bit

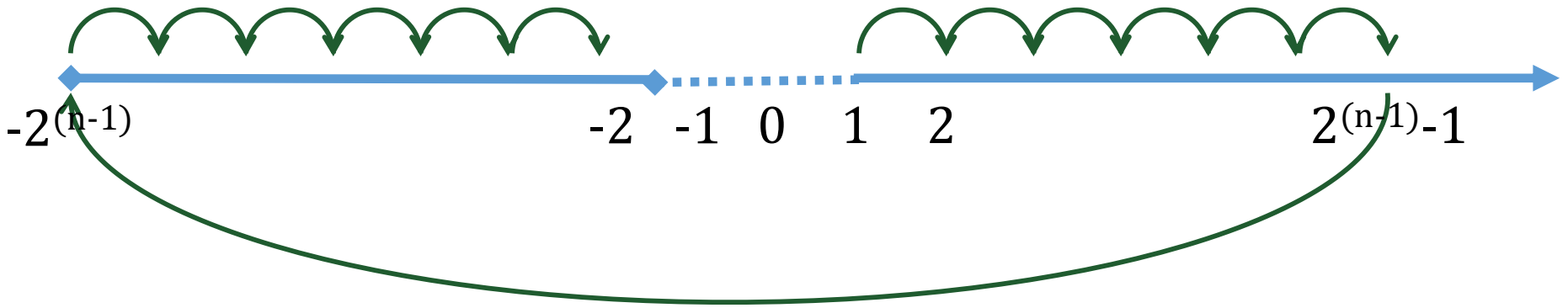
(char) 0x7f + (char) 0x01 = 0x80 = (char) 0x80

(char) 127 + (char) 1 = 128 = (char) -128

BITS

Value

- See [xmp_counting/count.c]



Extending Positive Binary Numbers

- How do we make a number wider (more bits)?
- How do we add digits to a positive decimal number?
- 0023 (4 digits) -> ???????? (8 digits)
- 0b0101 (4 bits) -> 0b???? ???? (8 bits)

Extending Negative Binary Numbers

- ~~*value*~~$bits = 2^n - \left(\sum_{i=0}^{n-1} B_i \times 2^i\right)$

- $2^n = 1\overbrace{0\,0\,0\,0\,\ldots\,0\,0\,0}^{\text{n zeroes}}$

$$\begin{array}{r} 1\overbrace{0\,0\,0\,0\,\ldots\,0\,0\,0}^{\text{n zeroes}} \\ -\quad 0\,0\,1\,0\,\ldots\,0\,1\,0 \\ \hline 1\,1\,0\,1\,\ldots\,1\,0\,1 \end{array}$$

$$\begin{array}{r} 1\overbrace{0\,0\,0\,0\,\ldots\,0\,0\,0\,\ldots\,0\,0\,0}^{\text{m zeroes}} \\ -\quad 0\,0\,0\,0\,\ldots\,0\,1\,0\,\ldots\,0\,1\,0 \\ \hline 1\,1\,1\,1\,\ldots\,1\,0\,1\,\ldots\,1\,0\,1 \end{array}$$

$2^m = 1\overbrace{0\,0\,0\,0\,\ldots\,0\,0\,0\,\ldots\,0\,0\,0}^{\text{m zeroes}}$

Extending Binary Numbers

- Pad on the left with the sign bit

Truncating Binary Numbers

- Remove left most bits
- If any of the left most bits (including the resulting sign bit) are different from the sign bit, result is incorrect (OVERFLOW)

- Examples 8 bit to 4 bit…

0b 0000 0011 -> 0b 0011 (3->3)

0b 0010 0011 -> 0b 0011 (35->3) OVFL

0b 1111 1001 -> 0b 1001 (-7 -> -7)

0b 1101 1001 -> 0b 1001 (-39 -> -7) OVFL

0b 1111 0111 -> 0b 0111 (-9 -> +7) OVFL

Mixing Types ("Casting")

- How do you mix signed and unsigned numbers?
- How do you mix long / short numbers?
- Strong Typing – Compiler prevents mixing of types

```
int x; unsigned int y; x=y;
```

Error: Cannot assign unsigned int to int

- Explicit Casting – Programmer tells compiler "treat this unsigned integer like a signed integer"

```
int x; unsigned int y; x=(int) y;
```

- Implicit Casting

```
int x; unsigned int y; x=y;
```

General Conversion Strategy

- If “from” size is smaller than “to” size (to precision is greater)
  - Pad on left with sign bit for signed data
  - Pad on left with 0’s for unsigned data
- If “from” size is greater than “to” size (to precision is less)
  - Truncate
- For signed->unsigned or unsigned->signed (same precision)
  - Bits stay the same… just interpreted differently.

Casting Examples

Legend
= precision
unsigned >precision
signed >precision
<precision

From			To			To			To		
Type	Bits	Value	Type	Bits	Value	Type	Bits	Value	Type	Bits	Value
uchar	0x01	1	char	0x01	1	uint	0x0001	1	int	0x0001	1
uchar	0xFF	255	char	0xFF	-1	uint	0x00FF	255	int	0xFFFF	-1
char	0x01	1	uchar	0x01	1	uint	0x0001	1	int	0x0001	1
char	0xFF	-1	uchar	0xFF	255	uint	0x00FF	255	int	0xFFFF	-1
uint	0xF0F3	61683	uchar	0xF3	243	char	0xF3	-13	int	0xF0F3	-3853
int	0x01D2	466	uchar	0xD2	210	char	0xD2	-46	uint	0x01D2	466

Integer Constant Types

- If not specified, integer constants are considered to be signed integers.
  - Make a constant "unsigned" by adding a "U" suffix

```
unsigned char x = 12U;
```

Implicit Casting for Integers in C

- In an expression,
  - if anything is unsigned, everything is unsigned!
    ```
    int x=-1; unsigned int y=321;
    if (x>y) printf("I didn't expect this\n");
    ```
  - everything smaller than int is cast to int
    ```
    char x=100; char y=50; x = (x*y)/500;
    ```
  - everything smaller than the largest type is cast to the largest type
- In an assignment or function call,
  - the result is cast to the type of the receiver
- When in doubt, use explicit casts!

Abstraction

I don't need to worry about mixing types.

C takes care of implicit conversion

and does the right thing.

Leaky Abstraction

One of the most common causes of bugs in code occurs when naïve programmers assume C will do the right thing.

- Single unsigned variable causes all data to get treated as unsigned, especially in comparisons
- Assumption of floating point division when integer division (with truncation) is performed

Examples of implicit casting bugs

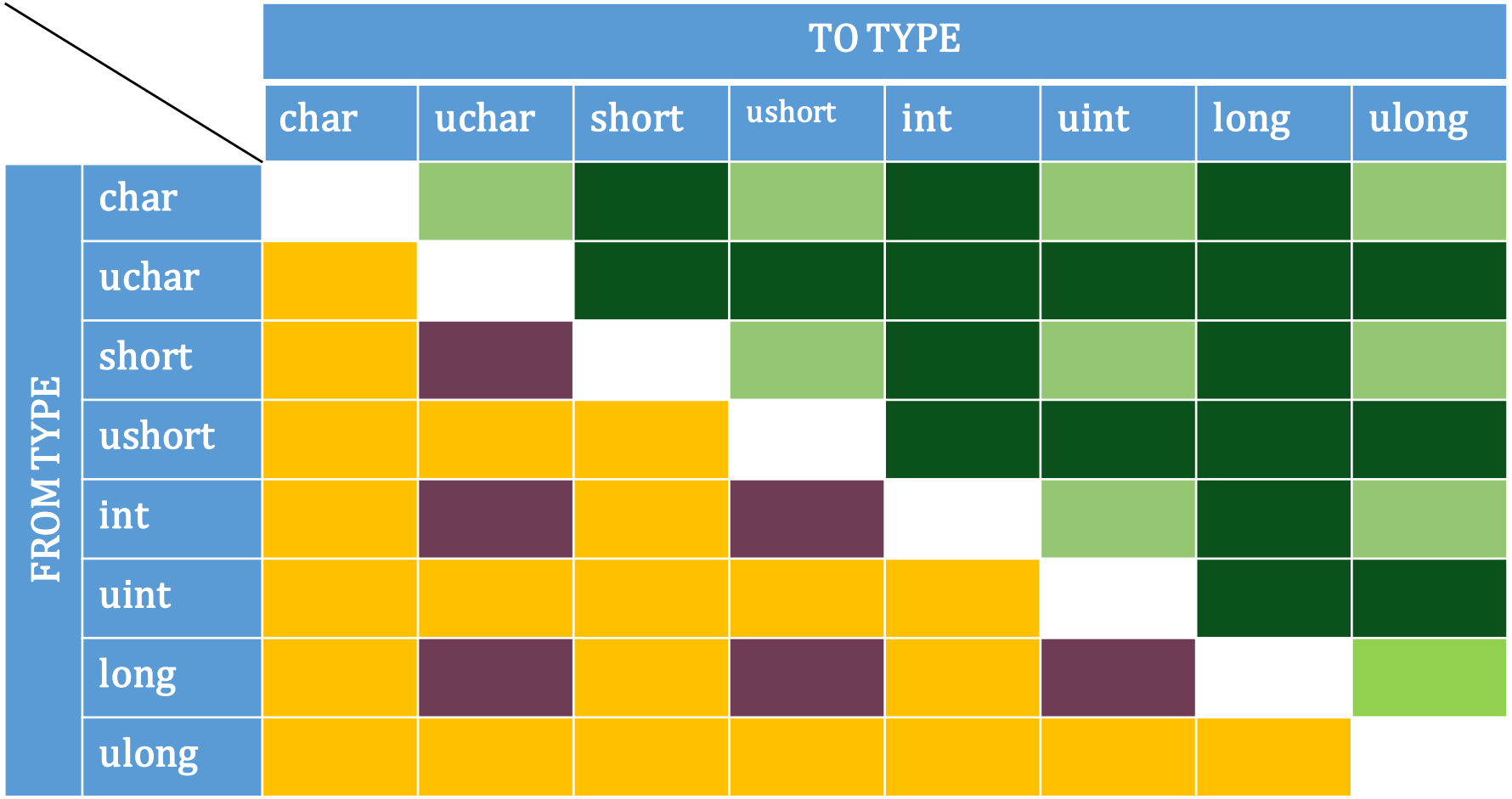
```
unsigned int length;
int x;

if (x < length) {
    /* what if x=-1, length=3 */
   …
}
```

```
int grade;
int test1,test2; // Scores <= 100

grade=((test1+test2)/200) * 100;
//Everybody gets zero!
```

Conversion Error Summary



LEGEND	
	Extend >=prec. No errors
	Extend >=prec. Wrong if <0
	Truncate <prec Wrong if too big
	Truncate <prec Wrong if <0 or too big