Interpretation of Bit Vectors
Mapping signed ↔ unsigned

- The computer itself has no idea if a given bit pattern at a particular location in memory “signed” or “unsigned”.

- The program interprets some given bit pattern according to the *type* that value has been assigned.

- Moreover, mappings between unsigned and two’s complement numbers keep the same bit representations but are interpreted differently depending on type, *which may yield a different value in your program.*

Two’s Complement

![Diagram](image)

Maintain Same Bit Pattern
Mapping signed ↔ unsigned con’t

<table>
<thead>
<tr>
<th>Bits</th>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>-8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>-7</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
<td>15</td>
</tr>
</tbody>
</table>

T2U
U2T

\[ \text{T2U} = \text{U2T} = \pm 2^w \]
Insights into overflow

- Lets say you have a signed char with the bit pattern…

  01111111

- What is its value in two’s complement in decimal? How about unsigned?
Insights into overflow

- Lets say you have a signed char with the bit pattern…

01111111

- What is its value in two’s complement in decimal? How about unsigned?

\( t: 127 \) \( u: 127 \)

- Lets say 1 is added to 127. What is the bit pattern for 128?
Insights into overflow

- Lets say you have a signed char with the bit pattern…
  
  \[ \text{01111111} \]

- What is its value in two’s complement in decimal? How about unsigned?
  
  \[ t: 127 \quad u: 127 \]

- Lets say 1 is added to 127. What is the bit pattern for 128?
  
  \[ \text{10000000} \]

- What is this bit pattern’s value in two’s complement in decimal? How about unsigned?
Insights into overflow

- Lets say you have a signed char with the bit pattern...

  01111111

- What is its value in two’s complement in decimal? How about unsigned?

  t: 127          u: 127

- Lets say 1 is added to 127. What is the bit pattern for 128?

  10000000

- What is this bit pattern’s value in two’s complement in decimal? How about unsigned?

  t: -128          u: 128

- See overflow.c
It’s all a matter of interpretation

- The key idea so far here is that a bit pattern is just a bit pattern!!
  - It has no intrinsic value or semantics.
- How that bit pattern is ‘interpreted’ determines its value in your program.
- Ok, so how are bit patterns interpreted in programs?
It’s all a matter of interpretation

- The key idea so far here is that a bit pattern is just a bit pattern!!
  - It has no intrinsic value or semantics.
- How that bit pattern is ‘interpreted’ determines its value in your program.
- Ok, so how are bit patterns interpreted in programs?

Datatypes!
Conversion & Casting with Integers
Signed vs. unsigned in C

- **Constants**
  - By default are considered to be signed integers
  - If you want unsigned you must add a “U” suffix
    
    ```
    unsigned int x = 0U;
    unsigned int y = 4294967259U;
    ```

- **Casting**
  - *Explicit* casting between signed & unsigned
    
    ```
    int tx, ty;
    unsigned int ux, uy;
    tx = (int) ux;
    uy = (unsigned) ty;
    ```
  
  - *Implicit* casting also occurs during assignments and function calls
    
    ```
    tx = ux;
    uy = ty;
    ```
Casting surprises

- If there is a mix of unsigned and signed in single expression, signed values are *implicitly cast to unsigned*
  - Includes expressions with comparison operators: `<`, `>`, `==`, `<=`, `>=`
  - See `casting_surprise.c`
- There can also be unexpected results when working with array indices
  - See `array_surprise.c` and `array_surprise2.c`
Casting signed ↔ unsigned: summary

- When the coercion takes place the bit pattern is *maintained*
  - However the program will *reinterpret* its value!
  - Can have unexpected effects if not careful, as we just observed.
- Again, expressions containing signed and unsigned int…
  - signed integral is coerced to an unsigned integral!!
Signed ‘extension’

- When we do a ‘widening conversion’ of a value via casting, what happens?

- In other words, given \( w \)-bit signed typed integer value \( x \), convert it to \( w+k \)-bit typed integer with same value.
  - \( w \) is the number of bits in the type of \( x \)
    - ex. \text{short} = 16
  - \( k \) is the number of bits difference between the two types
    - ex. \( k \) of \text{short} vs \text{int} = 16

- Moreover, what happens in cases like this?

```
short x = 15213;
int ix = (int) x;
short y = -15213;
int iy = (int) y;
```
Signed ‘extension’ con’t

Solution: make $k$ copies of the sign bit

$X = x_{w-1}x_{w-2} \ldots x_1x_0$

$X' = x_{w-1}x_{w-1}x_{w-1}x_{w-2} \ldots x_1x_0$

- Unsigned: zeros added
- Signed: sign bit extension
- Both yield intuitive and expected result
Signed ‘extension’ con’t

- Therefore, converting from smaller to larger integer data type C automatically performs sign extension.

- Therefore, this code...

  ```
  short x  = 15213;
  int ix  = (int) x;
  short y  = -15213;
  int iy  = (int) y;
  ```

- …has the values….

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>ix</td>
<td>15213</td>
<td>00 00 3B 6D</td>
<td>00000000 00000000 00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>iy</td>
<td>-15213</td>
<td>FF FF C4 93</td>
<td>11111111 11111111 11000100 10010011</td>
</tr>
</tbody>
</table>
Truncation

- When we do a ‘narrowing conversion’ of a value via coercion or casting, what happens? (i.e. from 32-bit int to 16-bit short)

- Higher-order bits are truncated. Value is altered, will be reinterpreted.

- Might yield reasonable result if value is ‘small enough’ to fit in smaller type…

```
int i = 1;
short s = (short) i;
```

- But what about something like this?

```
short s = 256;
char c = (char) s;
```

- This non-intuitive behavior can lead to buggy code!

- See coercion.c
Summary

- **Extension** (e.g. short to int)
  - Unsigned: zeroes added
  - Signed: sign extension
  - Both yield expected results

- **Truncation** (e.g. unsigned short to unsigned int)
  - Unsigned/signed: Higher weighted bits are lopped off
  - Result must be reinterpreted
  - For ‘small numbers’ (e.g. int w/ value 16 into short), ok
  - For ‘large numbers’ (e.g. int w/ value $2^{20}$ into short), problematic.
Negation & Addition
Negation

- **Task:** given a bit-vector \( X \) compute \(-X\)

- **Solution:** \(-X = \neg X + 1\)

  - Negating a value is done by computing its complement and adding 1

- **Example:**

  \[
  x = 011001_2 = 25_{10}
  \]

  \[
  \neg x = 100110_2 = -26_{10}
  \]

  \[
  \neg x + 1 = 100111_2 = -25_{10}
  \]

  - Notice, therefore, that for any signed integral type \( x \), \( \neg x + x = -1 \)

- See *negation.c*
Addition in base 2

- Very simple, works same as base 10, just remember..
  - 0 + 0 = 0
  - 0 + 1 = 1
  - 1 + 0 = 1
  - 1 + 1 = 10

- Examples:
  
  \[
  \begin{array}{c}
  101 \\
  +101 \\
  \hline
  1010
  \end{array}
  \quad
  \begin{array}{c}
  1011 \\
  +1011 \\
  \hline
  10110
  \end{array}
  \]

- Note in the second example that in the 2^1 column, we have 1 + (1 + 1), where the first 1 is "carried" from the 2^0 column.
Unsigned addition

Operands: \( w \) bits

True Sum: \( w+1 \) bits

Discard Carry: \( w \) bits

However since types have a limited number of bits, any carry bits after the MSB simply get truncated.

\[
\begin{array}{c}
\begin{array}{c}
10010_2 \\
+ 11011_2
\end{array} \\
\hline
101101_2 = 45_{10}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
10010_2 \\
+ 11011_2
\end{array} \\
\hline
101101_2 = 13_{10}
\end{array}
\]

See *unsigned_addition_overflow.c*
Signed addition

Operands: \( w \) bits

True Sum: \( w+1 \) bits

Discard Carry: \( w \) bits

- \( \text{TAdd} \) and \( \text{UAdd} \) have identical bit-level behavior. If true sum requires \( w+1 \) bits, any carry bits after the MSB simply get truncated.

\[
\begin{align*}
10010_2 + 11011_2 &= \underbrace{101101}_2 = -19_{10} \\
01101_2 &= 13_{10}
\end{align*}
\]
Signed addition *con’t*

- One important notable difference!
  - If $\text{sum} \geq 2^{w-1}$, value becomes negative (overflow)
  - If $\text{sum} < -2^{w-1}$, value becomes positive (underflow)
- An now you can explain integer overflow to all your friends!!
- See *signed_addition_overflow.c*
Summary

- **Addition:**
  - Unsigned/signed: Normal addition followed by truncate, same operation on bit level
  - Unsigned: addition mod $2^w$
    - Mathematical addition + possible subtraction of $2^w$
  - Signed: modified addition mod $2^w$ (result in proper range)
    - Mathematical addition + possible addition or subtraction of $2^w$