In a computer, all numbers are represented with a finite number of bits. This determines the maximum value that can be represented.

- int x, short int as;
- What is sizeof(x), sizeof(as)?
  - 2 bytes =16 bits, 4 bytes =32 bits
  - Max integer value given by $2^w$ or $2^w - 1$
- Let's look at unsigned numbers
- Range of values for unsigned short is
  - 0 to 65535
- Range of values for short int is (2 bytes)
  - -32768 to +32767
- Range of values for unsigned int is (4 bytes)
  - 0 to 4,294,967,296
- Range of values for int is
  - -2,147,483,648 to 2,147,483,647

Unsigned Binary addition

<table>
<thead>
<tr>
<th></th>
<th>0001</th>
<th>0010</th>
<th>0000</th>
<th>0010</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0001</td>
<td>0010</td>
<td>0100</td>
<td>0101</td>
</tr>
<tr>
<td>+6</td>
<td>0110</td>
<td>1001</td>
<td>1001</td>
<td>1010</td>
</tr>
<tr>
<td></td>
<td>1011</td>
<td>1001</td>
<td>1010</td>
<td>0001</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>0001</td>
<td>1011</td>
<td>0010</td>
</tr>
<tr>
<td>18</td>
<td>10010</td>
<td>0010</td>
<td>1110</td>
<td>0001</td>
</tr>
</tbody>
</table>

- Sum of 4 bit operands may require 5 bits in the result.
- If we limit the result to only 4 bits, then ignore MSB (value is $2^w$).
- The result is actually $U+V - 2^w$ the same as $(U+V) \mod 2^w$

Implication of finite number of bits

- All numbers have range +/- $2^w$, including results of operations on 2 or more numbers.
- When two (+ve or -ve) numbers are added, the result may end up in the red region.
- The answer will be wrong. Need to detect.
- On some occasions, left to the programmer to detect.
- On other occasions, hardware will provide the necessary information to detect overflow.
Unsigned binary addition

```c
unsigned short int ux;
unsigned short int uy;
unsigned short int ans;
ux=6; uy=5;
ans=uy-ux;
printf("Value of answer is %u\n", ans);
```

- What should happen?
  - Should this operation even be allowed?
  - What will happen?
  - Guess
- Why does this happen?
  - In C, -y is represented as 2s complement

Overflow

- An overflow occurs when the result cannot fit within the word-size limits of the result data type
- When executing C programs, overflows are not signaled as errors!
- It is onus on the programmer to determine if an overflow has occurred or make sure that overflow does not occur by limiting the ranges of the operands
- When two unsigned numbers s, x are added
  - s+x ≥ s and s+x ≥ x
- Overflow has occurred if
  - sum is s+x < s or s+x < x

2's complement addition

- Recall, 2's complement -ve numbers are represented with MSB=1
- MSB has a special meaning in 2s Complement
- Addition of 2s complement is just binary addition with carry ignored
- Subtraction, invert the sign of the subtrahend and add
- Carry out - a bit carried out of MSB
- Carry in – a bit inserted into MSB

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-7</td>
<td>1001</td>
<td>-5</td>
<td>1011</td>
</tr>
<tr>
<td>+5</td>
<td>0101</td>
<td>+2</td>
<td>1110</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(-2) 0010</td>
</tr>
<tr>
<td>-2</td>
<td>1110</td>
<td>-7</td>
<td>1 1001 ignore carry out</td>
</tr>
</tbody>
</table>

Overflow

- When two numbers of w bits are added, the result could need w+1 bits.
- Since, MSB of 1 represents negative numbers, a carry into the MSB creates problems

Example 1: Positive Overflow

```
6  0110
+ 5 0101
------
10 1001
```

Example 2: Negative Overflow

```
-5 1011
+ (-6) 1010
------
10 1000
```

Example 1: need to add 2w to compensate
Example 2: need to add -2w to compensate
Overflow

- In example 1, we added two positive numbers, that resulted a carry into the MSB
  - Result was a –ve number
  - (in 2’s complement), MSB of 1 means negative number
- Positive overflow
- In example 2, we added two negative numbers, that resulted a carry out of MSB (ignore)
  - Result was a –ve number
  - A carry out of MSB leaves MSB bit as 0. A MSB of 0 means a positive number in 2’s complement
- Negative overflow

Normal case

<table>
<thead>
<tr>
<th>Operation</th>
<th>First operand</th>
<th>Second operand</th>
<th>Overflow condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>X+Y</td>
<td>≥ 0</td>
<td>≥ 0</td>
<td>&lt; 0</td>
</tr>
<tr>
<td>X-Y</td>
<td>&lt; 0</td>
<td>&lt; 0</td>
<td>≥ 0</td>
</tr>
<tr>
<td>X*Y</td>
<td>≥ 0</td>
<td>&lt; 0</td>
<td>&lt; 0</td>
</tr>
<tr>
<td>X/Y</td>
<td>&lt; 0</td>
<td>≥ 0</td>
<td>≥ 0</td>
</tr>
</tbody>
</table>

Detecting overflow

- Overflow occurs when
  - Adding two positive numbers and the result is negative and vice versa
  - Subtracting a negative number from a positive number and the result is negative
  - Subtracting a positive number from a negative number and the result is positive

Detecting overflow

- Adding a +ve number and a –ve number will not cause a problem
- Adding two +ve numbers without carry in and carry out is also OK
- Adding two –ve numbers with carry in and carry out is also OK
Actions on overflow

- Detect in hardware
- Raise flags or special bits to indicate overflow
  - Compiler generated code for a Language can make use of these flags to inform programmer
  - Ignore overflow and leave it to programmer to implement checks to detect overflow

Unsigned multiplication

- When two numbers, each w bits long are multiplied, the result can be 2*w bits long
- If the result also needs to be w bits long, then w bits need to be discarded
- The w Most Significant Bits (MSBs) are discarded
- Unsigned Multiplication
  \[
  \text{unsigned int } u, v, up;
  
  up = u \times v
  
  \text{Modular arithmetic: } up = (u \times v) \mod 2^w
  
  \text{Else, the result needs to be } 2^w \text{ bits long}
  \]

Unsigned Multiplication in C

- Standard Multiplication Function
  - Store true product in 2^w bits
  - Or if product is also w bits long, ignore high order w bits
  - Implements Modular Arithmetic
    \[
    \text{UMult}(u, v) = u \cdot v \mod 2^w
    \]

Multiplication terms (recall grade school)

- decimal Multiplication
- Multiplicand
- Multiplier
- Partial products
  - Final sum
  - Each digit in the multiplier has different place values
    - The partial product is shifted left for each consecutive digit in the multiplier
Binary (unsigned) Multiplication

1 0 1
x 1 1

1 0 1
1 0 1
1 1 1 1

Algorithm
- Step 1: Check LSB of multiplier
- Decision Step 2: If 1, then add multiplicand to result
  - If 0, add 0 to result
- Step 3: Shift multiplicand left by 1 bit
  - Multiply by 2
- Step 4: Shift multiplier right by 1 bit
- Stop Condition: Repeat steps for number of bits in the multiplier

Binary multiplication
- Multiplicand, multiplier, and result
- Add to result if LSB of multiplier is 1
- Shift left multiplicand
- Shift right multiplier
- Repeat until multiplier is 0

Binary multiplier
- 16 bits (note: multiplicand is 8 bits but needs 16 bits for shifts)
- ADDER
- Control
- Multiplier
- Shift right
- Result
- 16 bits
Signed Multiplication

- Determine sign of product based on sign of multiplicand and multiplier
  - \( ANS = X \times Y \)
  - \( ANS \) is +ve if \( X \) and \( Y \) have the same sign else \( ANS \) is –ve
- Hence
  - Convert multiplier, multiplicand to positive numbers
  - Multiply unsigned numbers as before
  - Compute sign, convert product accordingly
- Next, 2s complement multiplication

Sign extension

- Converting a 2s complement number to a larger data type (short to long)
- For +ve numbers, you can pad as many 0s to the MSB as you want without changing value
- For 2’s complement, in order to change from 4 bits to 8 bits or from 8 bits to 16 bits, append additional bits on the left side of the number. Fill each extra bit with the value of the number’s most significant bit (the sign bit).

<table>
<thead>
<tr>
<th>Signed integer</th>
<th>4-bit representation</th>
<th>8-bit representation</th>
<th>16-bit representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>0001</td>
<td>00000001</td>
<td>0000000000000001</td>
</tr>
<tr>
<td>-1</td>
<td>1111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
</tbody>
</table>

2s complement multiplication

- With sign extension
- \(-3 \times -3\)
- \(111101 \times 111101\)
- \(Ans = 2^w\) LSBs
- Takes the first 6 Least Significant digits

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Another 2s Complement multiplier

- Perform sign extension on shifts
- Use a combined register for multiplier and result
- \(101 \times 010\)  
  - \(-3 \times 2\)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Note
- \(1 + 1 = 10\)
- \(1 + 1 + 1 = 11\)
- \(1 + 1 + 1 + 1 = 100\)
- \(1 + 1 + 1 + 1 + 1 + 1 = 101\)
Binary multiplier (with sign extension)

8 bits (No Shift)

- Multiplicand
- ADDER
- Result
- Multiplier
- Control

16 bits Shift right

Fast Multipliers

- Number of tricks used to improve multiplication
- Booth multiplication algorithm

Integer division

Repeated subtraction

Shift and subtract
Repeated subtraction

- Q=0; R=Dividend, D=Divisor
- While (D<=R)
  - {Q++; R=R-D;}
- printf(Q(in binary),R)
- 0111 divided by 11
- Q=0; R=0111, D=11
- I1: Q=1; R=0111-11=0100
- I2: Q=2; R=0100-11=0001
- Exit

Integer Division

- How does hardware know if division fits?
  - Condition: if remainder ≥ divisor
  - Use subtraction: (remainder – divisor) ≥ 0
- OK, so if it fits, what do we do?
  - Remainder_{n+1} = Remainder_n – divisor
  - What if it doesn’t fit?
    - Have to restore original remainder
  - Called restoring division
  - Dividend = Q x Divisor + remainder
  - Size(dividend in bits) = Size(Q) + Size(Divisor)

Integer Division (shift and subtract)

- Again, back to 3rd grade (56 ÷ 6 = 9 rem 2)

Binary division (Algorithm 1)

- Remainder reduction
  - Subtract 2^n from remainder
  - Shift right

- Remainder is positive
  - Shift left
  - Dividend is negative
  - Subtract 2^n from dividend
  - Shift right
### Restoring Division Example

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Step</th>
<th>Divisor</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initial values</td>
<td>0010 0000</td>
<td>0000 0111</td>
</tr>
<tr>
<td>1</td>
<td>1: Rem = Rem - Div</td>
<td>0010 0000</td>
<td>1110 1110</td>
</tr>
<tr>
<td></td>
<td>Check: Rem &lt; 0 +D, all Q LSB(Q)=0</td>
<td>0001 0000</td>
<td>0000 0111</td>
</tr>
<tr>
<td></td>
<td>Q=0000</td>
<td></td>
<td>Shift Div right</td>
</tr>
<tr>
<td>2</td>
<td>2: Rem = Rem - Div</td>
<td>0001 0000</td>
<td>1111 0111</td>
</tr>
<tr>
<td></td>
<td>Check Rem &lt; 0 +D, all Q LSB(Q)=0</td>
<td>0000 1000</td>
<td>0000 0111</td>
</tr>
<tr>
<td></td>
<td>Q=0000</td>
<td></td>
<td>Shift Div right</td>
</tr>
<tr>
<td></td>
<td>3: Rem = Rem - Div</td>
<td>00001000</td>
<td>1111 1111</td>
</tr>
<tr>
<td></td>
<td>Check Rem &lt; 0 +D, all Q LSB(Q)=0</td>
<td>00001000</td>
<td>0000 0111</td>
</tr>
<tr>
<td></td>
<td>Q=0000</td>
<td></td>
<td>Shift Div right</td>
</tr>
</tbody>
</table>

### Restoring Division Example

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Step</th>
<th>Divisor</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1: Rem = Rem - Div</td>
<td>0000 0100</td>
<td>0000 0011</td>
</tr>
<tr>
<td></td>
<td>Check: Rem &gt;= 0, all Q LSB(Q)=1</td>
<td>0000 0100</td>
<td>0000 0011</td>
</tr>
<tr>
<td></td>
<td>Q=0001</td>
<td></td>
<td>Shift Div right</td>
</tr>
<tr>
<td></td>
<td>5: Rem = Rem - Div</td>
<td>0000 0010</td>
<td>0000 0001</td>
</tr>
<tr>
<td></td>
<td>Check Rem &gt;= 0, all Q LSB(Q)=1</td>
<td>0000 0010</td>
<td>0000 0001</td>
</tr>
<tr>
<td></td>
<td>Q=0011</td>
<td>R=0001</td>
<td>Shift Div right</td>
</tr>
</tbody>
</table>

### 32 bit Binary Division

![Diagram of 32 bit Binary Division](image)