CS 47

Integers

Topics

- Basic operations
  - Addition, negation, multiplication
- Programming Implications
  - Consequences of overflow
  - Using shifts to perform power-of-2 multiply/divide

C Puzzles

- Taken from old exams
- Assume machine with 32 bit word size, two's complement integers
- For each of the following C expressions, either:
  - Argue that is true for all argument values
  - Give example where not true

Initialization

```
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```

- `x < 0` ⇒ `((x*2) < 0)`
- `ux >= 0`
- `x & 7 == 7` ⇒ `(x<<30) < 0`
- `ux > -1`
- `x > y` ⇒ `-x < -y`
- `x * x >= 0`
- `x > 0 && y > 0` ⇒ `x + y > 0`
- `x >= 0` ⇒ `-x <= 0`
- `x <= 0` ⇒ `-x >= 0`

C Puzzle Answers

- **Assume machine with 32 bit word size, two's comp. integers**
- ***TMin* makes a good counterexample in many cases**

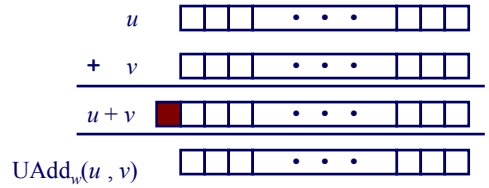
`x < 0`	⇒	`((x*2) < 0)`	**False:**	*TMin*
`ux >= 0`			**True:**	0 = *UMin*
`x & 7 == 7`	⇒	`(x<<30) < 0`	**True:**	$x_1 = 1$
`ux > -1`			**False:**	0
`x > y`	⇒	`-x < -y`	**False:**	`-1`, *TMin*
`x * x >= 0`			**False:**	`46341`
`x > 0 && y > 0`	⇒	`x + y > 0`	**False:**	*TMax*, *TMax*
`x >= 0`	⇒	`-x <= 0`	**True:**	–*TMax* < 0
`x <= 0`	⇒	`-x >= 0`	**False:**	*TMin*

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



Standard Addition Function

- **Ignores carry output**

Implements Modular Arithmetic

$s = \text{UAdd}_w(u , v) = u + v \mod 2^w$

$$UAdd_w(u,v) = \begin{cases} u+v & u+v < 2^w \\ u+v-2^w & u+v \geq 2^w \end{cases}$$

Mathematical Properties

Modular Addition Forms an *Abelian Group*

- **Closed under addition**
  $0 \le \text{UAdd}_w(u, v) \le 2^w - 1$
- **Commutative**
  $\text{UAdd}_w(u, v) = \text{UAdd}_w(v, u)$
- **Associative**
  $\text{UAdd}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UAdd}_w(t, u), v)$
- **0 is additive identity**
  $\text{UAdd}_w(u, 0) = u$
- **Every element has additive inverse**
  - Let $\text{UComp}_w(u) = 2^w - u,\ u > 0.$ $\text{UComp}_w(0) = 0$

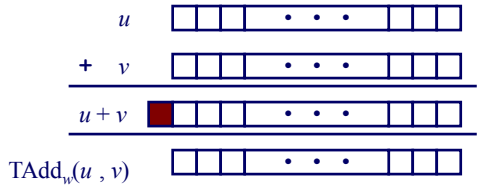
  $\text{UAdd}_w(u, \text{UComp}_w(u)) = 0$

Two's Complement Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



TAdd and UAdd have Identical Bit-Level Behavior

- **Signed vs. unsigned addition in C:**

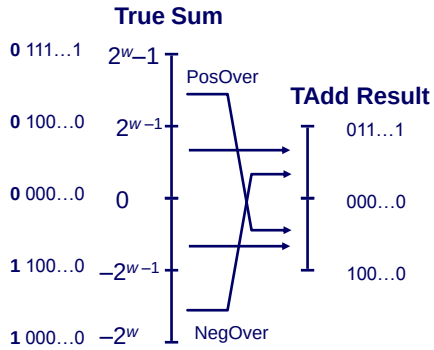
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

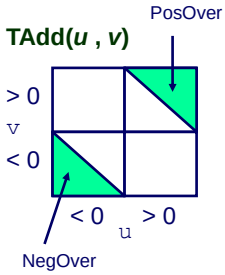
- **Will give** `s == t`

Characterizing TAdd

Functionality

- True sum requires w+1 bits
- Drop off MSB
- Treat remaining bits as 2's comp. integer





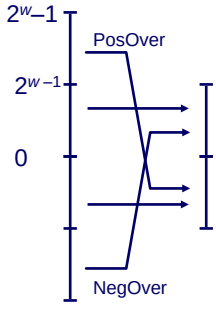
$$TAdd_w(u,v) = \begin{cases} u+v+2^w & u+v < TMin_w \quad \text{(NegOver)} \\ u+v & TMin_w \le u+v \le TMax_w \\ u+v-2^w & TMax_w < u+v \quad \text{(PosOver)} \end{cases}$$

Detecting 2's Comp. Overflow

Task

- **Given** s = $TAdd_w(u , v)$
- **Determine if** s = $Add_w(u , v)$
- **Example**

```
int s, u, v;
s = u + v;
```



Claim

- **Overflow iff either:**
  - $u, v < 0, s \ge 0$ **(NegOver)**
  - $u, v \ge 0, s < 0$ **(PosOver)**

```
ovf = (u<0 == v<0) && (u<0 != s<0);
```

Mathematical Properties of TAdd

Isomorphic Algebra to UAdd

- $TAdd_w(u , v) = U2T(UAdd_w(T2U(u), T2U(v)))$
  - **Since both have identical bit patterns**

Two's Complement Under TAdd Forms a Group

- **Closed, Commutative, Associative, 0 is additive identity**
- **Every element has additive inverse**

  Let $TComp_w(u) = U2T(UComp_w(T2U(u))$

  $TAdd_w(u , TComp_w(u)) = 0$

$$TComp_w(u) = \begin{cases} -u & u \neq TMin_w \\ TMin_w & u = TMin_w \end{cases}$$

Multiplication

Computing Exact Product of *w*-bit numbers *x*, *y*

- **Either signed or unsigned**

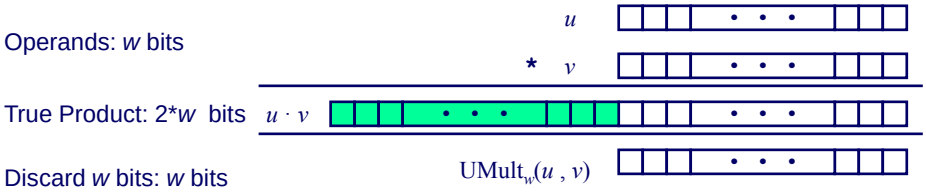
Ranges

- **Unsigned:** $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - **Up to 2*w* bits**
- **Two's complement min:** $x * y \ge (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - **Up to 2*w*–1 bits**
- **Two's complement max:** $x * y \le (-2^{w-1})^2 = 2^{2w-2}$
  - **Up to 2*w* bits, but only for $(TMin_w)^2$**

Maintaining Exact Results

- **Would need to keep expanding word size with each product computed**
- **Done in software by "arbitrary precision" arithmetic packages**

Unsigned Multiplication in C



Standard Multiplication Function

- **Ignores high order *w* bits**

Implements Modular Arithmetic

$\text{UMult}_w(u , v) = u \cdot v \mod 2^w$

Unsigned vs. Signed Multiplication

Unsigned Multiplication

```
unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
unsigned up = ux * uy
```

- **Truncates product to *w*-bit number** $up = \text{UMult}_w(ux, uy)$
- **Modular arithmetic:** $up = ux \cdot uy \mod 2^w$

Two's Complement Multiplication

```
int x, y;
int p = x * y;
```

- **Compute exact product of two *w*-bit numbers *x*, *y***
- **Truncate result to *w*-bit number** $p = \text{TMult}_w(x, y)$

Unsigned vs. Signed Multiplication

Unsigned Multiplication

```
unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
unsigned up = ux * uy
```

Two's Complement Multiplication

```
int x, y;
int p = x * y;
```

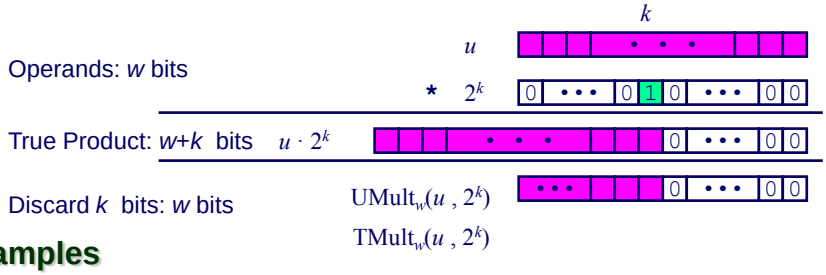
Relation

- Signed multiplication gives same bit-level result as unsigned
- `up == (unsigned) p`

Power-of-2 Multiply with Shift

Operation

- `u << k` gives `u *` 2^k
- Both signed and unsigned



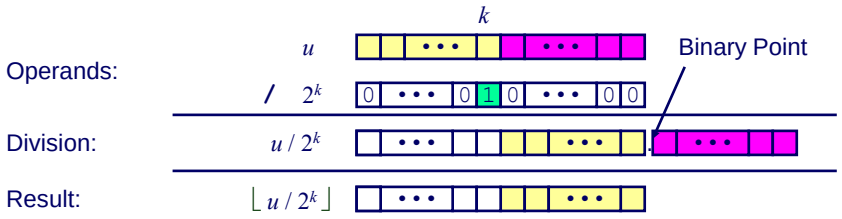
Examples

- `u << 3` == `u * 8`
- `u << 5 - u << 3` == `u * 24`
- Most machines shift and add much faster than multiply
  - Compiler generates this code automatically

Unsigned Power-of-2 Divide with Shift

Quotient of Unsigned by Power of 2

- `u >> k` gives $\lfloor$ `u` / 2^k $\rfloor$
- **Uses logical shift**

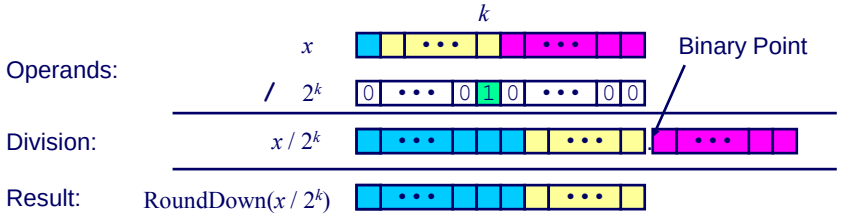


	Division	Computed	Hex	Binary
x	15213	15213	3B 6D	00111011 01101101
x >> 1	7606.5	7606	1D B6	**0**0011101 10110110
x >> 4	950.8125	950	03 B6	**0000**0011 10110110
x >> 8	59.4257813	59	00 3B	**00000000** 00111011

Signed Power-of-2 Divide with Shift

Quotient of Signed by Power of 2

- `x >> k` gives $\lfloor$ `x` / 2^k $\rfloor$
- **Uses arithmetic shift**
- **Rounds "wrong" direction when `x < 0`**



	Division	Computed	Hex	Binary
y	-15213	-15213	C4 93	11000100 10010011
y >> 1	-7606.5	-7607	E2 49	**1**1100010 01001001
y >> 4	-950.8125	-951	FC 49	**1111**1100 01001001
y >> 8	-59.4257813	-60	FF C4	**11111111** 11000100

Repeating Decimals

What rational number does 0.272727 … represent?

- Let x = 0.272727 …
- 100x = 27.272727 … (choose power of 10 to make repeating part match x)
- 100x – x = 99x = 27.0
- x = 27/99 = 3/11

How do we find decimals for 3/37?

- Use long division

```
      .081
  ___________
37 | 3.000000
     2.96
     ----
       40
       37
       --
        300 -> digits repeat: .081081 …
```


Repeating "Decimals" in Binary

What rational number does 0.011011 … represent?

- Let x = 0.011011 …
- 8x = 11.011011 … (choose power of 2 to make repeating part match x. Period = 3, $2^3 = 8$.)
- 8x – x = 7x = 3.0
- x = 3/7

How do we find binary "decimals" for 1/5?

- Use long division (in binary)

```
      .0011
  ___________
101 | 1.000000
      101
      ---
       0110
       0101
       ----
          100 -> digits repeat: .00110011 …
```

Repeating "Decimals" in Hex

What rational number does (hex) 0.666 … represent?

- Let x = 0.666 …
- 16x = 6.666 … (choose power of 16 to make repeating part match x. Period = 1, $16^1 = 16$.)
- 16x – x = 15x = 6.0
- x = 2/5

How do we find hex "decimals" for 2/5?

- Use long division (in hex)

```
    .66
5 | 2.000000
    1 e          1e16 = 3010
      20 -> digits repeat: .666 …
      1e
       2 -> digits repeat: .6666 … (just checking)
```


Properties of Unsigned Arithmetic

Unsigned Multiplication with Addition Forms Commutative Ring

- Addition is commutative group
- Closed under multiplication
  $0 \le \text{UMult}_w(u, v) \le 2^w - 1$
- Multiplication Commutative
  $\text{UMult}_w(u, v) = \text{UMult}_w(v, u)$
- Multiplication is Associative
  $\text{UMult}_w(t, \text{UMult}_w(u, v)) = \text{UMult}_w(\text{UMult}_w(t, u), v)$
- 1 is multiplicative identity
  $\text{UMult}_w(u, 1) = u$
- Multiplication distributes over addtion
  $\text{UMult}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UMult}_w(t, u), \text{UMult}_w(t, v))$

Properties of Two's Comp. Arithmetic

Isomorphic Algebras

- Unsigned multiplication and addition
  - Truncating to *w* bits
- Two's complement multiplication and addition
  - Truncating to *w* bits

Both Form Rings

- Isomorphic to ring of integers mod 2^w

Comparison to Integer Arithmetic

- Both are rings
- Integers obey ordering properties, e.g.,

  $u > 0 \Rightarrow u + v > v$

  $u > 0, v > 0 \Rightarrow u \cdot v > 0$

- These properties are not obeyed by two's comp. arithmetic

  TMax `+ 1 == TMin`

  `15213 * 30426 == -10030` (16-bit words)