Lecture 10

Integer Arithmetic

CPSC 275
Introduction to Computer Systems

Negation: Complement & Increment

- Claim: Following Holds for 2's Complement

  `~x + 1 == -x`

  But why?
- Complement
  - Observation: `~x + x == 1111…111 == -1`

x	1 0 0 1 1 1 0 1
+ ~x	0 1 1 0 0 0 1 0
-1	1 1 1 1 1 1 1 1

Complement & Increment Examples

x = 15213

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
~x	-15214	C4 92	11000100 10010010
~x+1	-15213	C4 93	11000100 10010011

x = 0

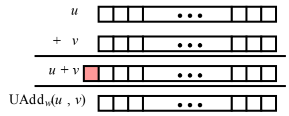
	Decimal	Hex	Binary
0	0	00 00	00000000 00000000
~0	-1	FF FF	11111111 11111111
~0+1	0	00 00	00000000 00000000

Unsigned Addition

Operands: w bits

True Sum: w+1 bits

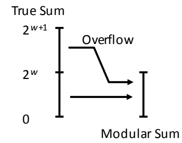
Discard Carry: w bits



- Standard Addition Function
  - Ignores carry output
- Implements Modular Arithmetic

  UAdd$_w$(u , v) = (u + v) mod 2^w

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

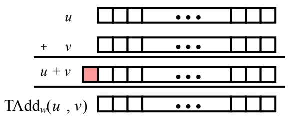


Two's Complement Addition

Operands: w bits

True Sum: w+1 bits

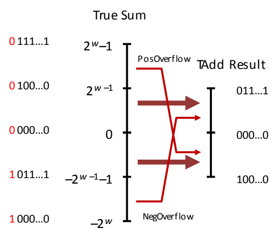
Discard Carry: w bits



- TAdd and UAdd have identical bit-level behavior

TAdd Overflow

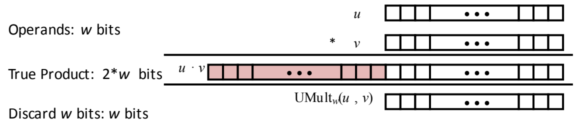
- True sum requires w+1 bits
- Drop off MSB
- Treat remaining bits as 2's comp. integer



Multiplication

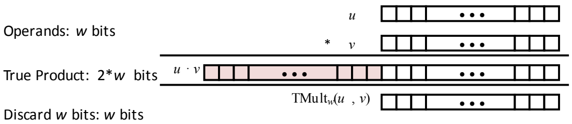
- Computing exact product of w-bit numbers x, y (either signed or unsigned) gives the following ranges:
  - Unsigned:
    $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - 2's comp:
    min: $x * y \ge (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
    max: $x * y \le (-2^{w-1})^2 = 2^{2w-2}$
  - require up to $2w$ bits

Unsigned Multiplication in C



- Standard multiplication function
  - Ignores high order w bits
- Implements modular arithmetic
  $\text{UMult}_w(u, v) = (u \cdot v) \bmod 2^w$

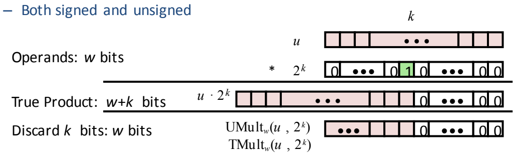
Signed Multiplication in C



- Standard Multiplication Function
  - Ignores high order w bits
  - Different interpretation for signed vs. unsigned multiplication
  - Lower bits are the same
- Example: $x = 100_2$, $y = 111_2$

Power-of-2 Multiply with Shift

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



- Most machines shift and add faster than multiply
  - Compiler generates this code automatically
- Examples

```
u << 3 == u * 8
u * 12 == ?
```

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`	`00011101 10110110`
`x >> 4`	950.8125	950	`03 B6`	`00000011 10110110`
`x >> 8`	59.4257813	59	`00 3B`	`00000000 00111011`

Signed Power-of-2 Divide with Shift

- Quotient of signed by power of 2
  - `u >> k` gives $\lfloor$ `u / 2^k` $\rfloor$
  - Uses arithmetic shift
  - Rounds wrong direction when `u < 0` (round down!)

	Division	Computed	Hex	Binary
`y`	-15213	-15213	`C4 93`	`11000100 10010011`
`y >> 1`	-7606.5	-7607	`E2 49`	`11100010 01001001`
`y >> 4`	-950.8125	-951	`FC 49`	`11111100 01001001`
`y >> 8`	-59.4257813	-60	`FF C4`	`11111111 11000100`

Correct Power-of-2 Divide

- Quotient of Negative Number by Power of 2
  - Want $\lceil$ **`x / 2`**^{k} $\rceil$ (round toward 0)
  - Compute as $\lfloor$ **`(x + 2`**^{k}**`- 1)/ 2`**^{k} $\rfloor$ (adding a *bias*)
  - In C: **`(x + (1 << k)- 1) >> k`**

Practice Problems

- Read CSaPP Sec. 2.3 and try the following problems:

  2.28, 2.29, 2.33, 2.34, 2.40