Module II
Data Representation

Dr. Arijit Roy
Computer Science and Engineering Group
Indian Institute of Information Technology Sri City

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, **multiplication, shifting**
- Summary

Multiplication

- Computing Exact Product of w-bit numbers x, y
  - Either signed or unsigned
- Ranges
  - Unsigned: $0 \leq x * y \leq (2^w - 1)^2 = 0$ and $(2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Two's complement min: $x * y \geq (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - Two's complement max: $x * y \leq (-2^{w-1})^2 = 2^{2w-2}$
  - **Up to 2*w* bits**

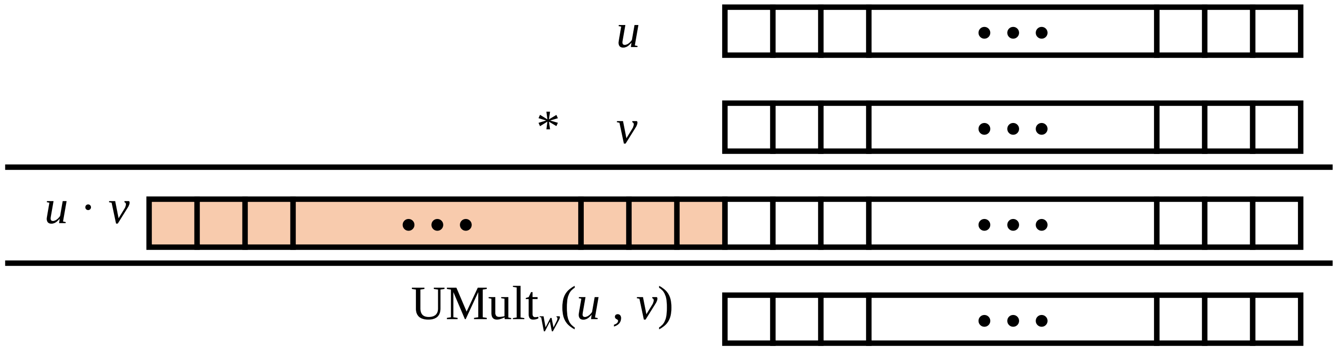
- Maintaining Exact Results
  - Would need to keep expanding word size with each product computed
  - Done in software by "arbitrary precision" arithmetic packages

Unsigned Multiplication in C

Operands: w bits

True Product: $2*w$ bits

Discard w bits: w bits



- Standard Multiplication Function
  - Ignores high order w bits
- Implements Modular Arithmetic

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

e.g.,
w=2, Let us consider the highest number i.e., 3 = 11

3*3 = 9
11 * 11 = 1001
From **10**01, it ignores higher order w bit. The result is 01.

It can be obtained by $u \cdot v \ \bmod 2^w$

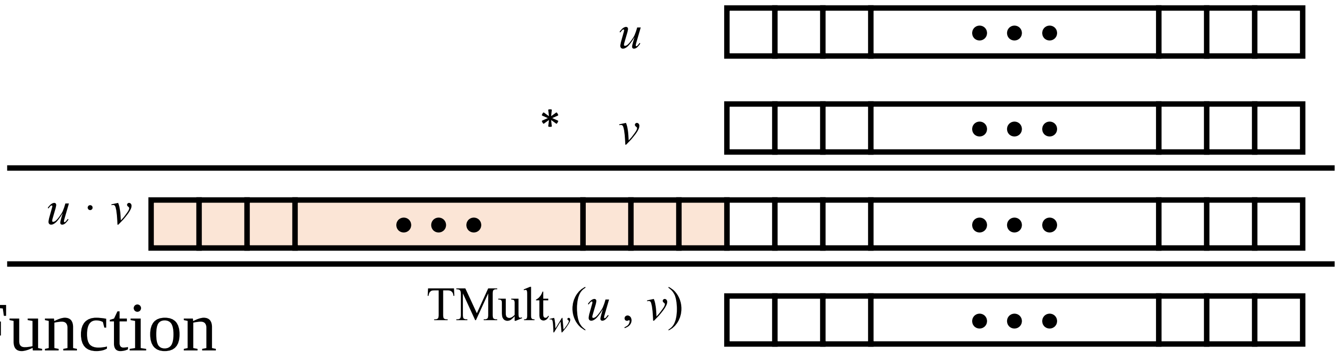
Therefore, $3*3 \bmod 2^2 = 1 = 01$, which is the answer

Signed Multiplication in C

Operands: w bits

True Product: $2*w$ bits

Discard w bits: w bits



- Standard Multiplication Function
  - Ignores high order w bits
  - Some of which are different for signed vs. unsigned multiplication
  - Lower bits are the same

Truncating a two's-complement number to w bits is equivalent to first computing its value modulo 2^w and then converting from unsigned to two's complement.

e.g.,
w=3, Let us consider the highest number i.e., -3 = 101
And 3 = 011
-3*3 = -9
101 * 011 = 110111
From **110**111, it ignores higher order w bit. The result is 111.
It can be obtained by $u \cdot v$ mod 2

Therefore, -3*3 mod 2 = 1 = 001, the 2's complement. We get the answer

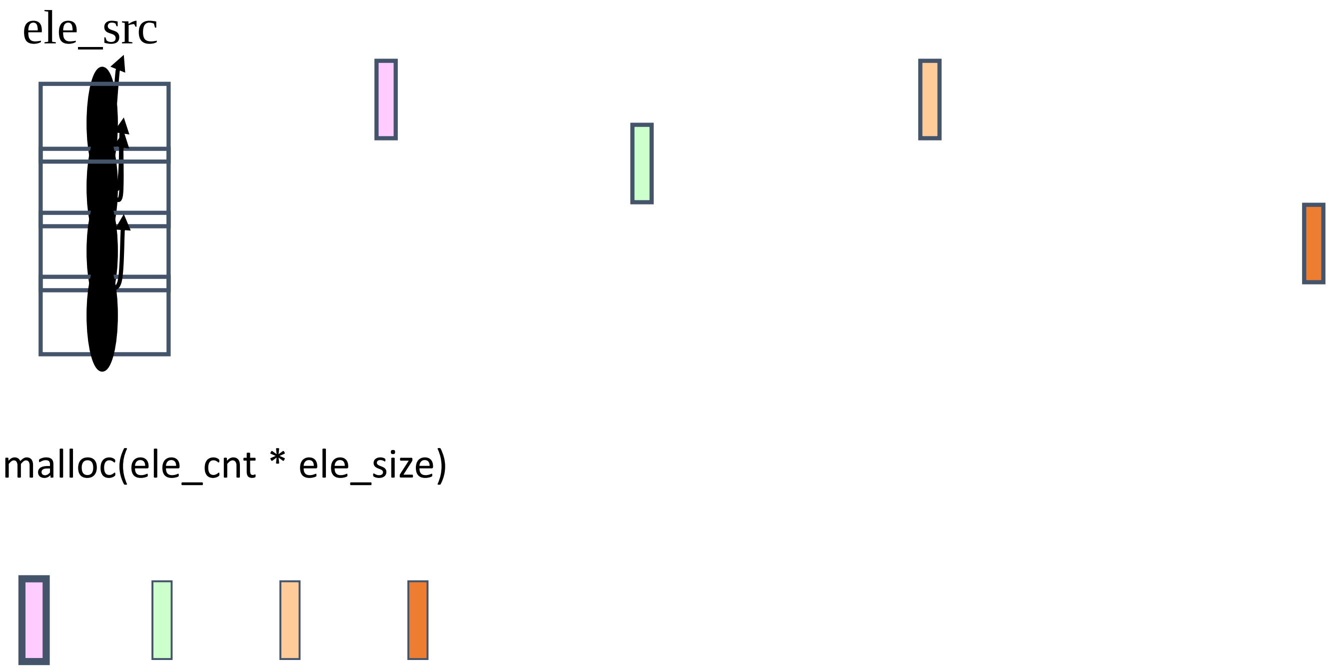
Mode	x		y		$x \cdot y$		Truncated $x \cdot y$	
Unsigned	5	[101]	3	[011]	15	[001111]	7	[111]
Two's comp.	–3	[101]	3	[011]	–9	[110111]	–1	[111]
Unsigned	4	[100]	7	[111]	28	[011100]	4	[100]
Two's comp.	–4	[100]	–1	[111]	4	[000100]	–4	[100]
Unsigned	3	[011]	3	[011]	9	[001001]	1	[001]
Two's comp.	3	[011]	3	[011]	9	[001001]	1	[001]

Figure 2.26 **Three-bit unsigned and two's-complement multiplication examples.** Although the bit-level representations of the full products may differ, those of the truncated products are identical.

Code Security Example #2

- SUN XDR library
  - Widely used library for transferring data between machines

```
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size);
```



XDR Code

Designed to copy **ele_cnt** data structure

Each consisting of **ele_size** bytes into a buffer

```
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size) {
    /*
     * Allocate buffer for ele_cnt objects, each of ele_size bytes
     * and copy from locations designated by ele_src
     */
    void *result = malloc(ele_cnt * ele_size);
    if (result == NULL)
       /* malloc failed */
       return NULL;
    void *next = result;
    int i;
    for (i = 0; i < ele_cnt; i++) {
        /* Copy object i to destination */
        memcpy(next, ele_src[i], ele_size);
       /* Move pointer to next memory region */
       next += ele_size;
    }
    return result;
}
```

XDR Vulnerability

malloc(ele_cnt * ele_size)

Multiplication can overflow, without giving notice

- What if:
  - **ele_cnt** = $2^{20} + 1$
  - **ele_size** = 4096 = 2^{12}
  - Allocation = ??
  - Allocation needed: 4,294,971,392
  - Actual allocation: 4096

A malicious programmer:
Call this function with **`ele_cnt`** being 1048577 = $2^{20} + 1$ and **`ele_size`** being 4096 = 2^{12}

Now the problem is overflow

- How can I make this function secure?

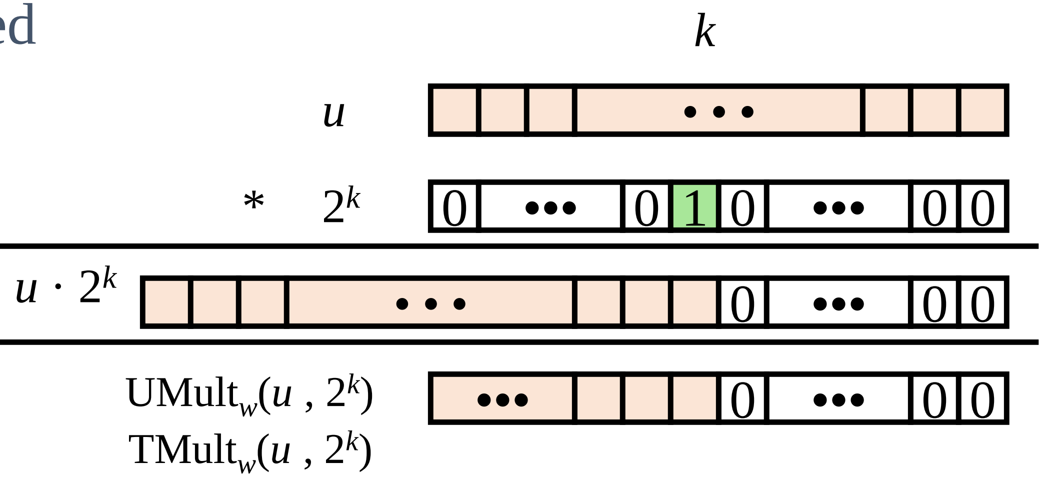
Power-of-2 Multiply with Shift

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

Operands: w bits

True Product: $w+k$ bits

Discard k bits: w bits



- Examples
  - **u << 3 == u * 8**
  - **u << 5 - u << 3== u * 24 (24 = 2^5 – 2^3)**
  - Most machines shift and add faster than multiply
    - Compiler generates this code automatically

Compiled Multiplication Code

C Function

```
int mul12(int x)
{
  return x*12;
}
```

Compiled Arithmetic Operations

```
leal    (%eax,%eax,2), %eax
sall    $2, %eax
```

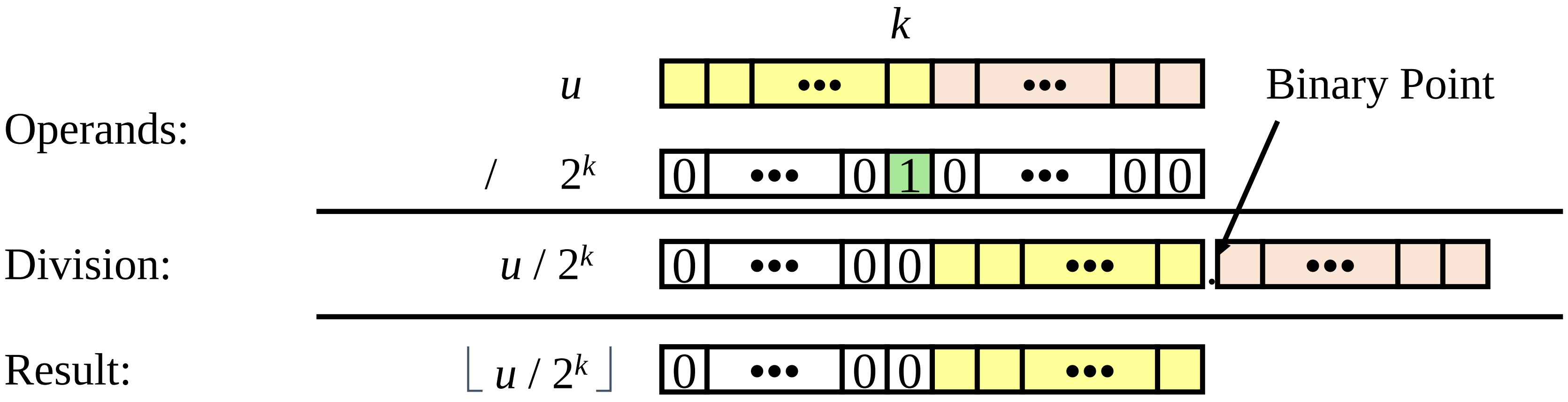
Explanation

```
t <- x+x*2
return t << 2;
```

- C compiler automatically generates shift/add code when multiplying by constant

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



k	>> k (Binary)	Decimal	$12340/2^k$
0	0011000000110100	12340	12340.0
1	0001100000011010	6170	6170.0
4	0000001100000011	771	771.25
8	0000000000110000	48	48.203125

Figure 2.27 **Dividing unsigned numbers by powers of 2.** The examples illustrate how performing a logical right shift by k has the same effect as dividing by 2^k and then rounding toward zero.

Compiled Unsigned Division Code

C Function

```
unsigned udiv8(unsigned x)
{
  return x/8;
}
```

Compiled Arithmetic Operations

```
    shrl    $3, %eax
```

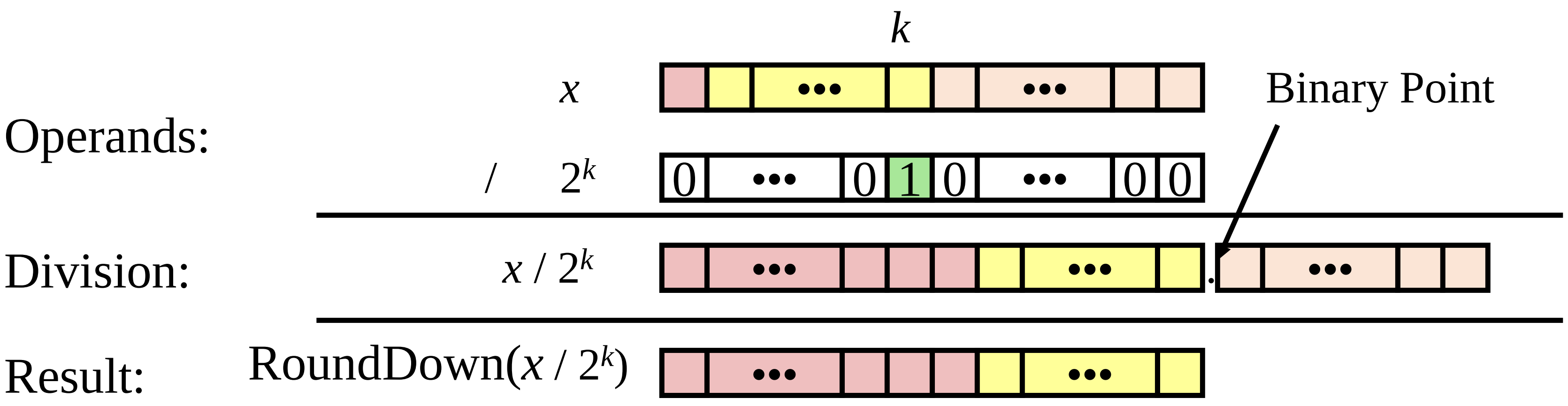
Explanation

```
    # Logical shift
    return x >> 3;
```

- Uses logical shift for unsigned
- For Java Users
  - Logical shift written as >>>

Signed Power-of-2 Divide with Shift

- Quotient of Signed by Power of 2
  - **x >> k** gives $\lfloor \mathbf{x / 2^k} \rfloor$
  - Uses arithmetic shift
  - Rounds wrong direction when **u < 0**



k	>> k (Binary)	Decimal	$-12340/2^k$
0	1100111111001100	−12340	−12340.0
1	*1*110011111100110	−6170	−6170.0
4	*1111*110011111100	−772	−771.25
8	*11111111*11001111	−49	−48.203125

Figure 2.28 **Applying arithmetic right shift.** The examples illustrate that arithmetic right shift is similar to division by a power of 2, except that it rounds down rather than toward zero.

Correct division

- 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*** $\rfloor$
    - In C: **(x + (1<<k)-1) >> k**
    - Biases dividend toward 0

This technique exploits the property that $\lceil x/y \rceil = \lfloor (x+y-1)/y \rfloor$ for integers x and y such that $y > 0$. As examples, when $x = -30$ and $y = 4$, we have $x + y - 1 = -27$, and $\lceil -30/4 \rceil = -7 = \lfloor -27/4 \rfloor$.

k	Bias	−12,340 + Bias (Binary)	>> k (Binary)	Decimal	$-12340/2^k$
0	0	1100111111001100	1100111111001100	−12340	−12340.0
1	1	1100111111001101	1110011111100110	−6170	−6170.0
4	15	1100111111011011	1111110011111101	−771	−771.25
8	255	1101000011001011	1111111111010000	−48	−48.203125

Figure 2.29 **Dividing two's-complement numbers by powers of 2.** By adding a bias before the right shift, the result is rounded toward zero.

Compiled Signed Division Code

C Function

```
int idiv8(int x)
{
  return x/8;
}
```

Compiled Arithmetic Operations

```
    testl %eax, %eax
    js      L4
L3:
    sarl    $3, %eax
    ret
L4:
    addl    $7, %eax
    jmp     L3
```

Explanation

```
if x < 0
x += 7;
# Arithmetic shift
return x >> 3;
```

- Uses arithmetic shift for int
- For Java Users
  - Arith. shift written as >>

Arithmetic: Basic Rules

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

- Multiplication:
  - Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
  - Unsigned: multiplication mod 2^w
  - Signed: modified multiplication mod 2^w (result in proper range)

Arithmetic: Basic Rules

- Left shift
  - Unsigned/signed: multiplication by 2^k
  - Always logical shift
- Right shift
  - Unsigned: logical shift, div (division + round to zero) by 2^k
  - Signed: arithmetic shift
    - Positive numbers: div (division + round to zero) by 2^k
    - Negative numbers: div (division + round away from zero) by 2^k
      Use biasing to fix

Today: Integers

- Representation: unsigned and signed
- Conversion, casting
- Expanding, truncating
- Addition, negation, multiplication, shifting
- Summary

Integer C Puzzles

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
- (x|-x)>>31 == -1
- ux >> 3 == ux/8
- x >> 3 == x/8
- x & (x-1) != 0