Bits, Bytes, and Integers – Part 2

CS2011: Introduction to Computer Systems
Lecture 4

Bits, Bytes, and Integers

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

    previous lecture

  - **Addition, negation, multiplication, shifting**

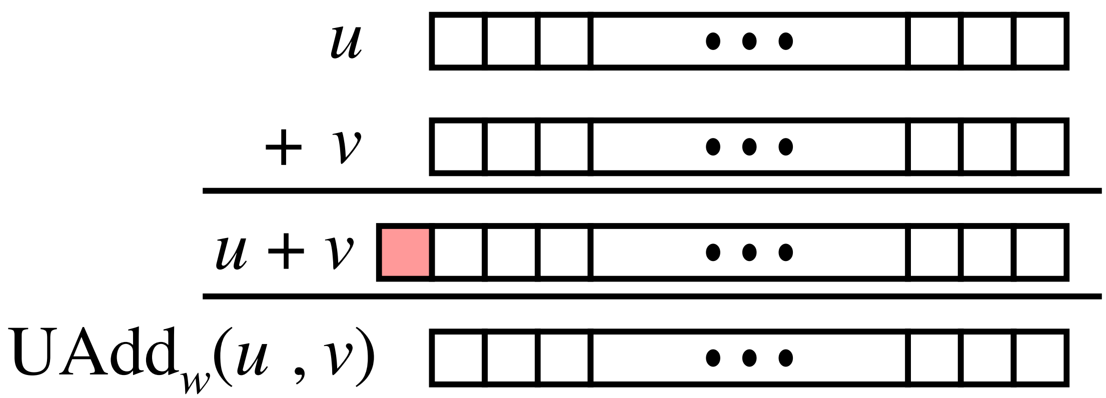
    this lecture
- **Representations in memory, pointers, strings**
- **Summary**

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



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

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

```
unsigned char      1110 1001      E9        233
                 + 1101 0101    + D5      + 213
                 -----------    ----      -----

                 -----------    ----      -----
```

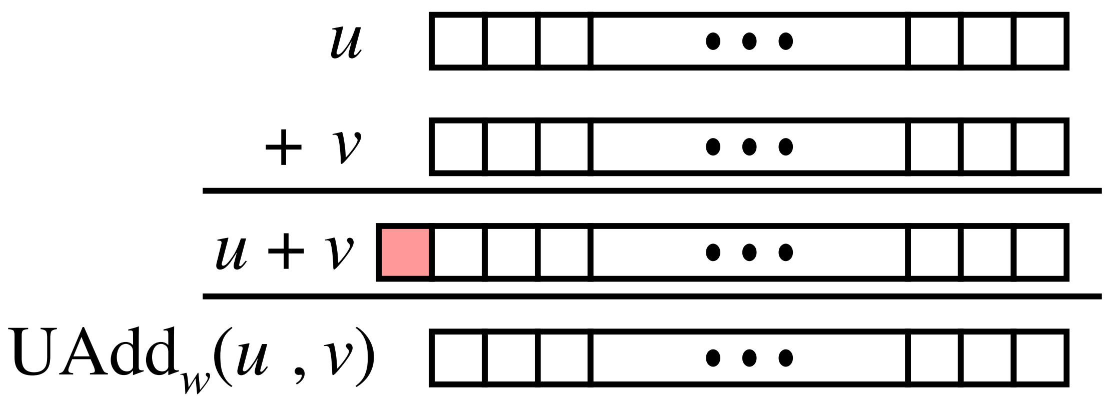
Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



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

$s \quad = \quad \mathrm{UAdd}_w(u\ ,\ v) \quad = \quad u + v \ \bmod 2^w$

= 446 - 256

unsigned char	1110 1001	E9	233
	+ 1101 0101	+ D5	+ 213
	1 1011 1110	1BE	446
	1011 1110	BE	190

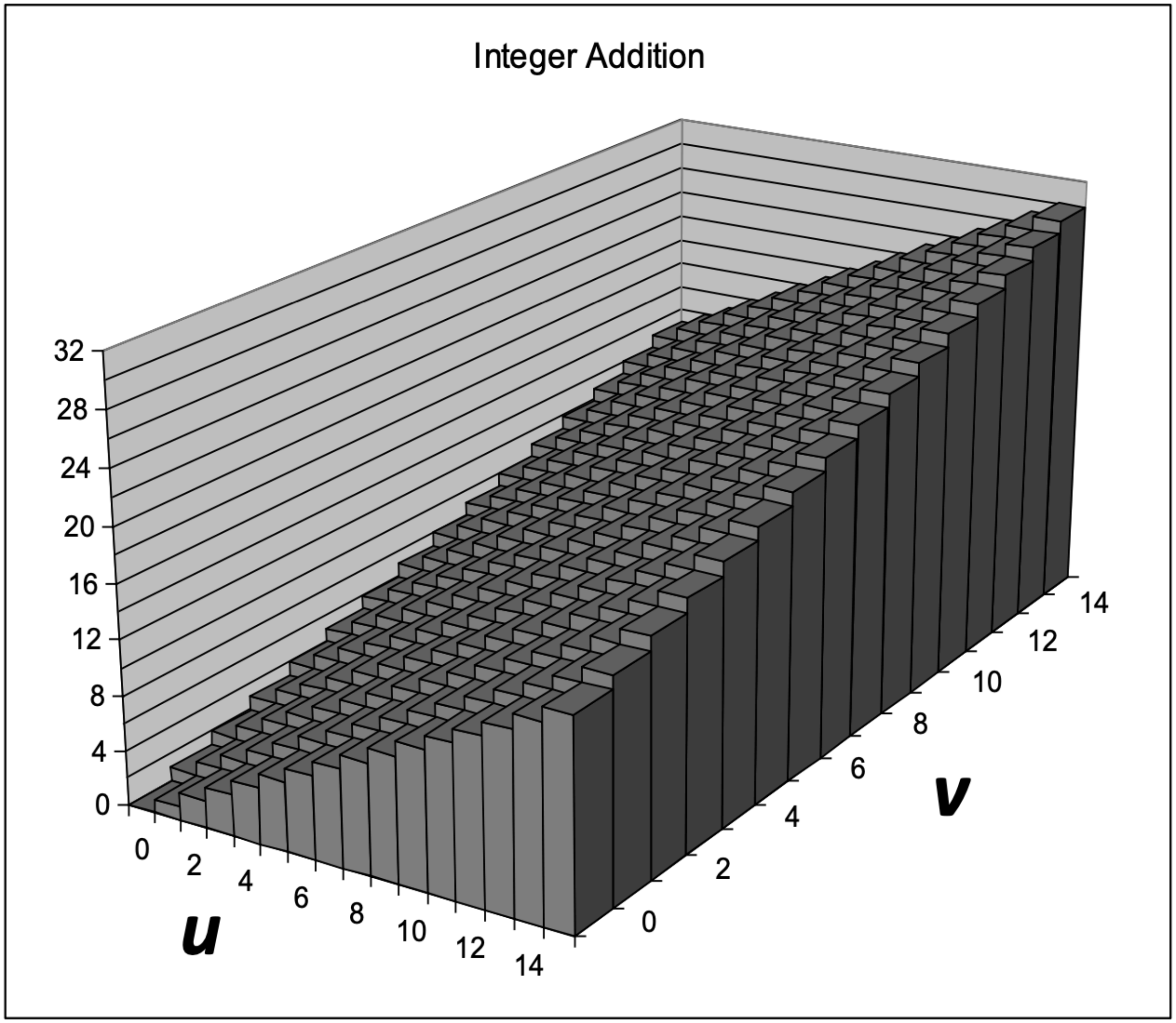
Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Visualizing (Mathematical) Integer Addition

Integer Addition

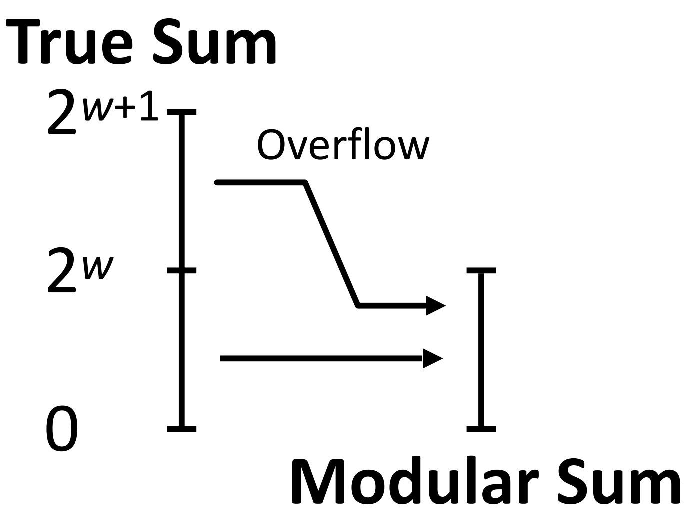
- **4-bit** integers *u*, *v*
- Compute true sum $\text{Add}_4(u, v)$
- Values increase **linearly** with *u* and *v*
- Forms planar surface

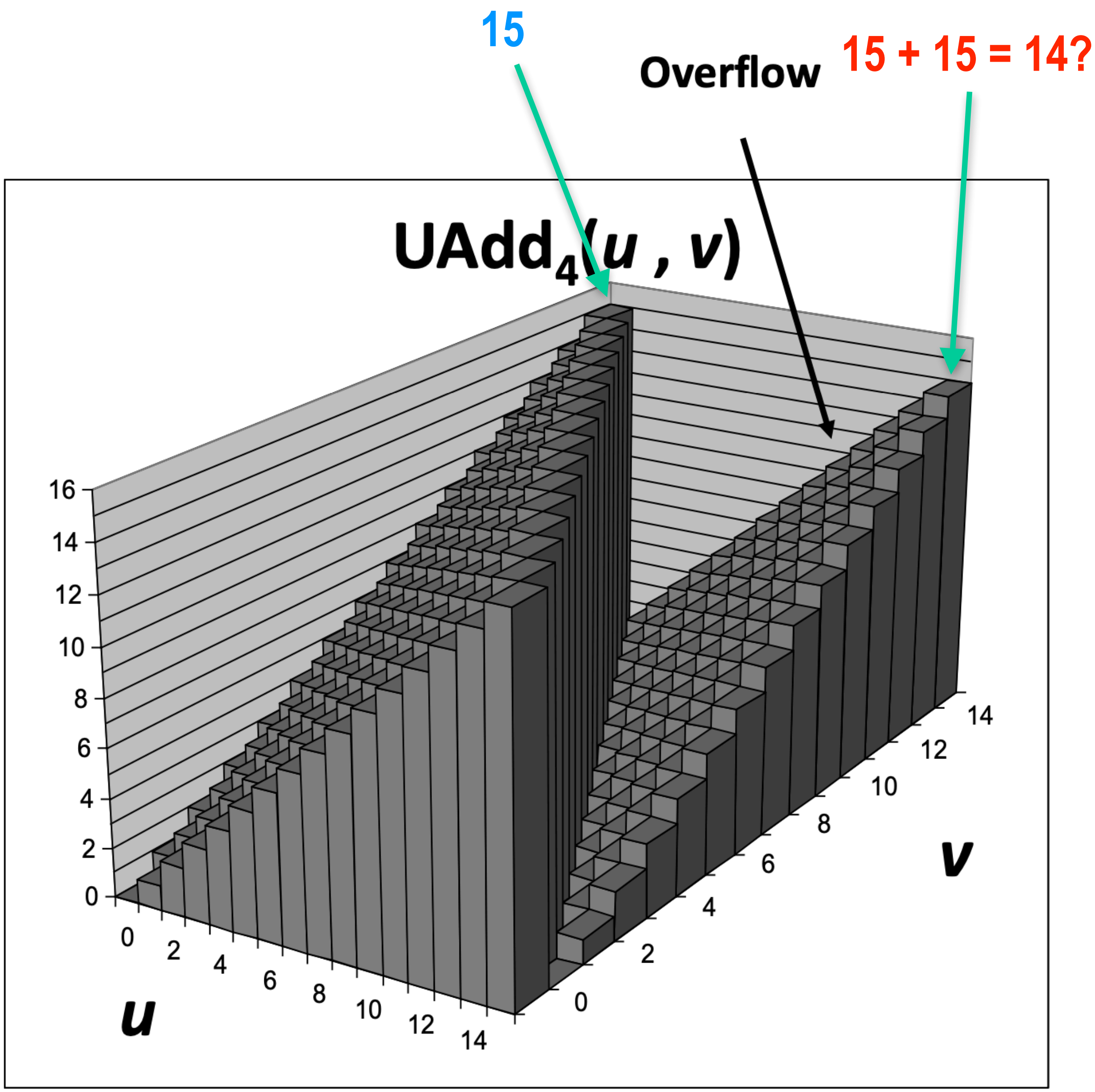
$\text{Add}_4(u, v)$



Visualizing Unsigned Addition

- **Wraps Around**
  - If true sum ≥ 2^w
  - At most once
  - E.g., w = 4, after wrap around —> max sum = 14



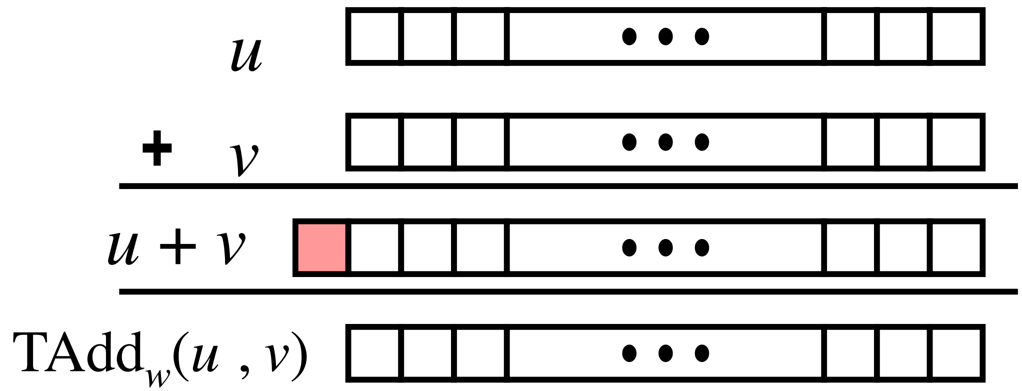


Two's Complement Addition

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:

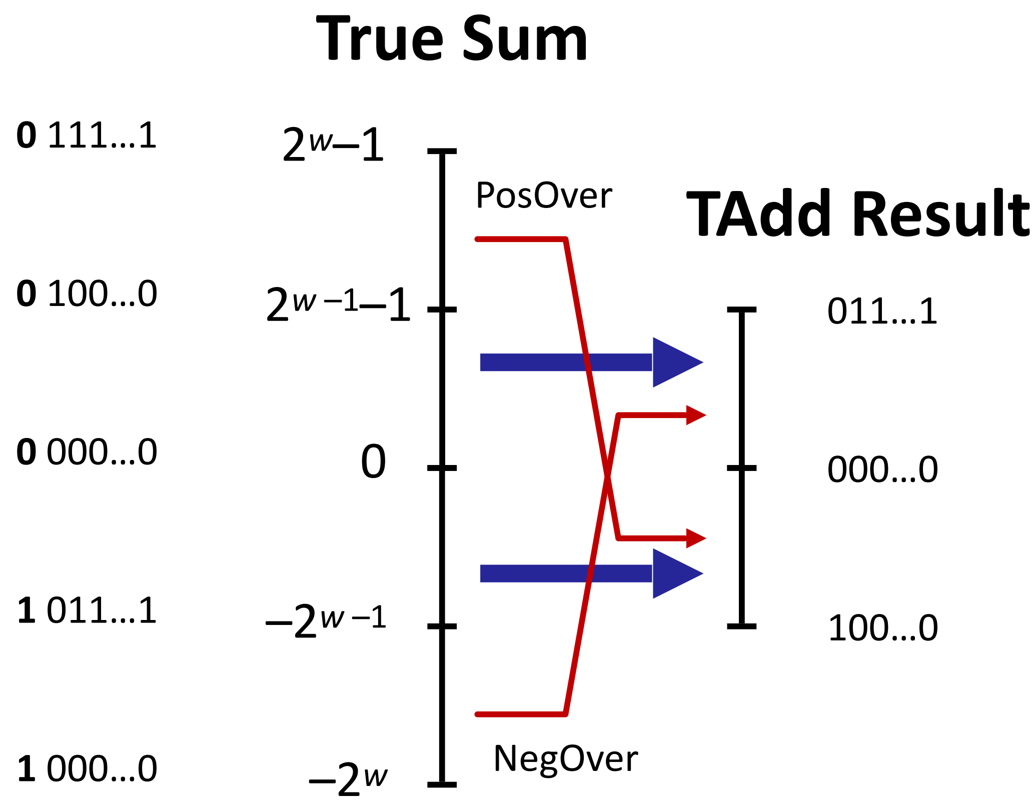
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v;
```

  - Will give `s == t`

```
    1110 1001       E9      -23
+   1101 0101     + D5    + -43
  -----------     ----    -----
  1 1011 1110      1BE      -66
  -----------     ----    -----
    1011 1110       BE      -66
```

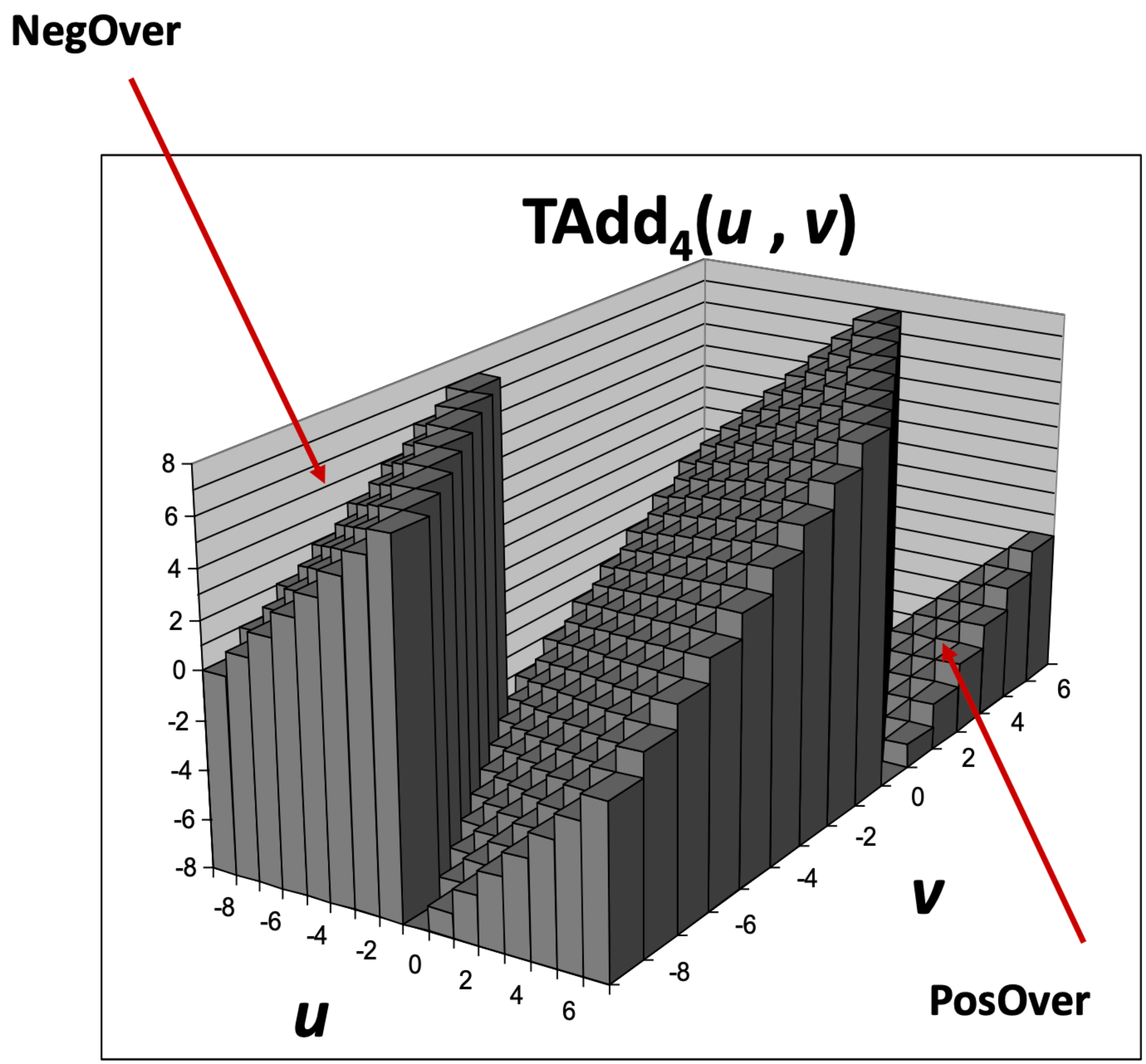
TAdd Overflow

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



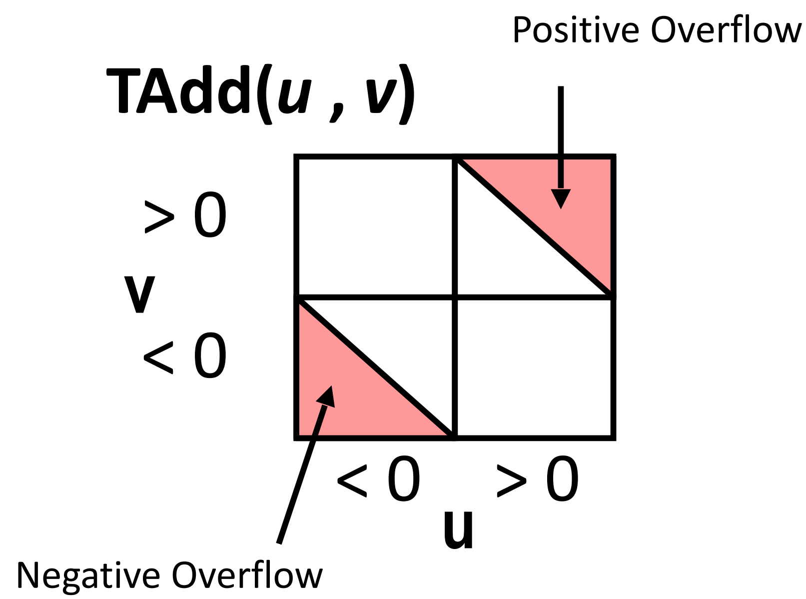
Visualizing 2's Complement Addition

- **Values**
  - 4-bit two's comp.
  - Range from -8 to +7
- **Wraps Around**
  - If sum $\geq 2^{w-1}$
    - **Becomes negative**
    - At most once
  - If sum $< -2^{w-1}$
    - **Becomes positive**
    - At most once



Characterizing TAdd

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



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

e.g., -5 + -5

e.g., 3 + -8

e.g., 4 + 4

Multiplication

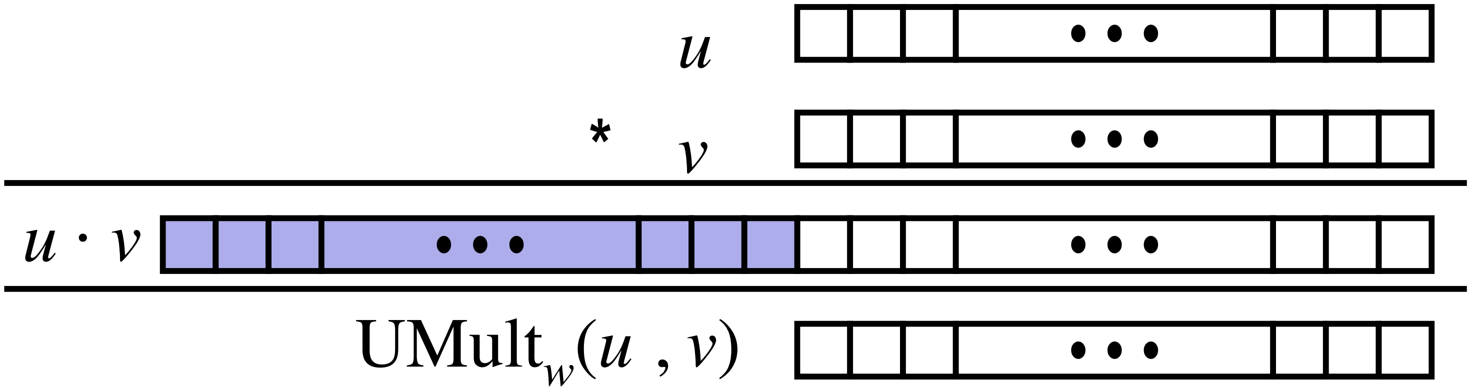
- **Goal: Computing Product of *w*-bit numbers *x*, *y***
  - Either signed or unsigned
- **But, exact results can be bigger than *w* bits**
  - **Unsigned**: up to 2*w* bits
    - Result range: $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
    - E.g., 4-bits: $0 \le x * y \le (2^4 - 1)^2 = 2^8 - 2^5 + 1 = 256 - 32 + 1 = 225$
  - **Two's complement** min (negative): Up to 2*w*-**1** bits
    - Result range: $x * y \ge (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
    - E.g., 4-bits: $x * y \ge (-2^3)*(2^3-1) = -8 * 7 = -56$
  - **Two's complement** max (positive): Up to 2*w* bits, but only for $(TMin_w)^2$
    - Result range: $x * y \le (-2^{w-1})^2 = 2^{2w-2}$
    - E.g., 4-bits: $x * y \le (-2^3)^2 = (-8)^2 = 64$
- **So, maintaining exact results…**
  - would need to keep expanding word size with each product computed
  - is done in software, if needed
    - e.g., 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**

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

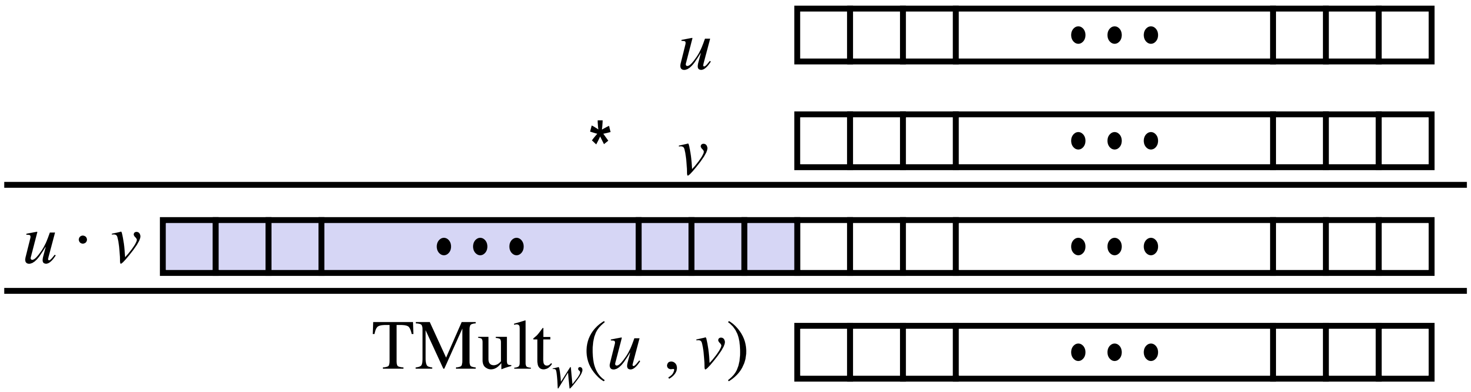
```
              1110 1001      E9       233
*             1101 0101    * D5     * 213
  1100 0001 1101 1101      C1DD     49629
              1101 1101      DD       221
```

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

```
             1110 1001        E9        -23
*            1101 0101    *   D5    *   -43
 0000 0011   1101 1101      03DD        989
             1101 1101        DD        -35
```

Power-of-2 Multiply with Shift

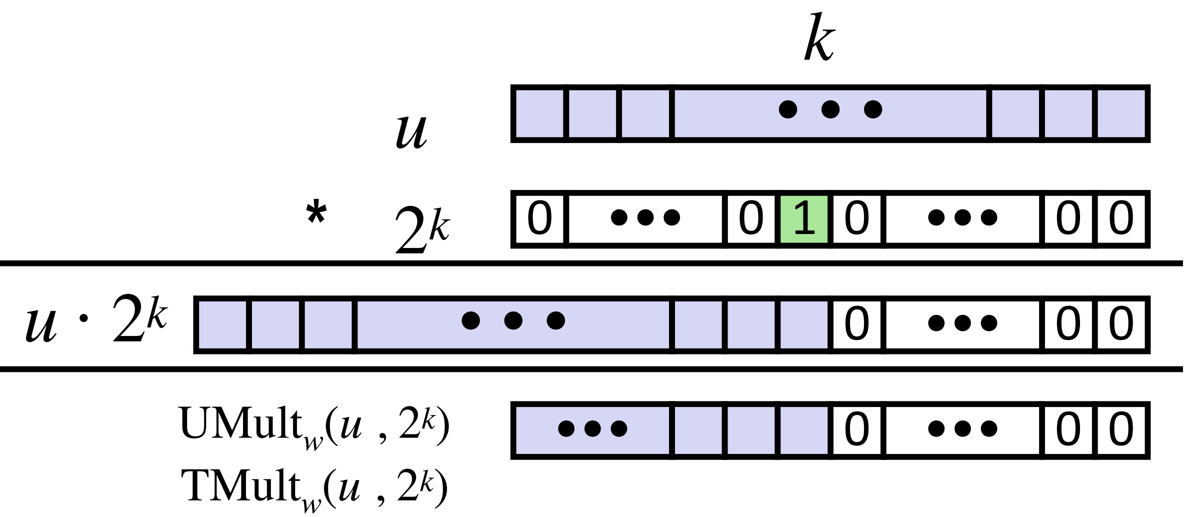
Operation

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

Operands: *w* bits

True Product: *w+k* bits

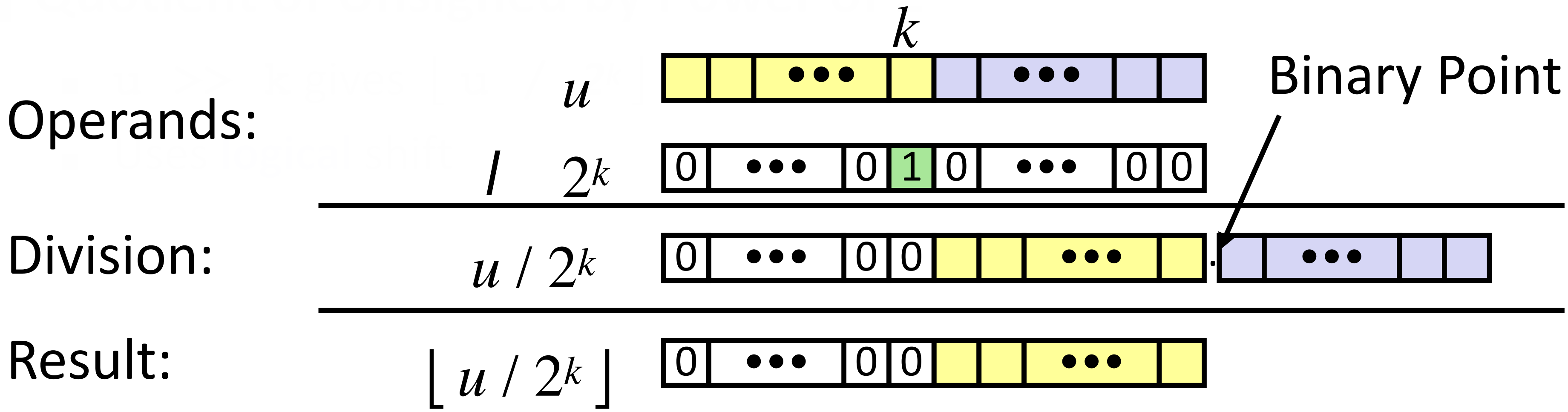
Discard *k* bits: *w* bits



Examples

- `u << 3 == u * 8`
- `(u << 5) - (u << 3) == u * 24`
- `E.g., 2 * 24 = 2 * (32 - 8) = (2 * 32) - (2 * 8)`
  `(2 << 5) - (2 << 3)`
- Most machines shift and add faster than multiply
  - Compiler generates this code automatically

Unsigned Power-of-2 Divide with Shift

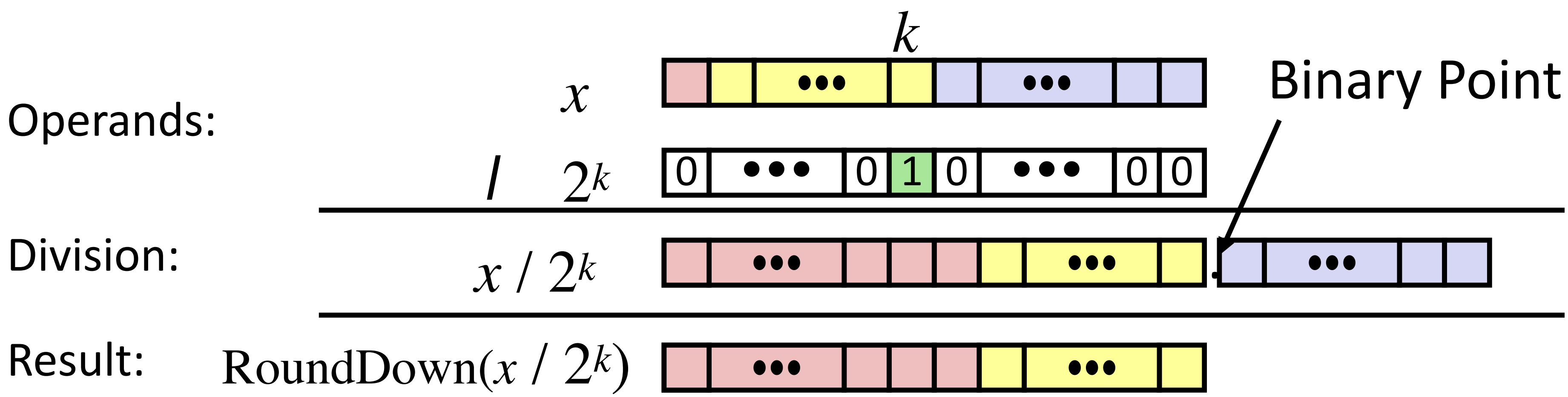


example with no binary point:
32 / 8 = 4 …. 32 >> 3 = 4

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



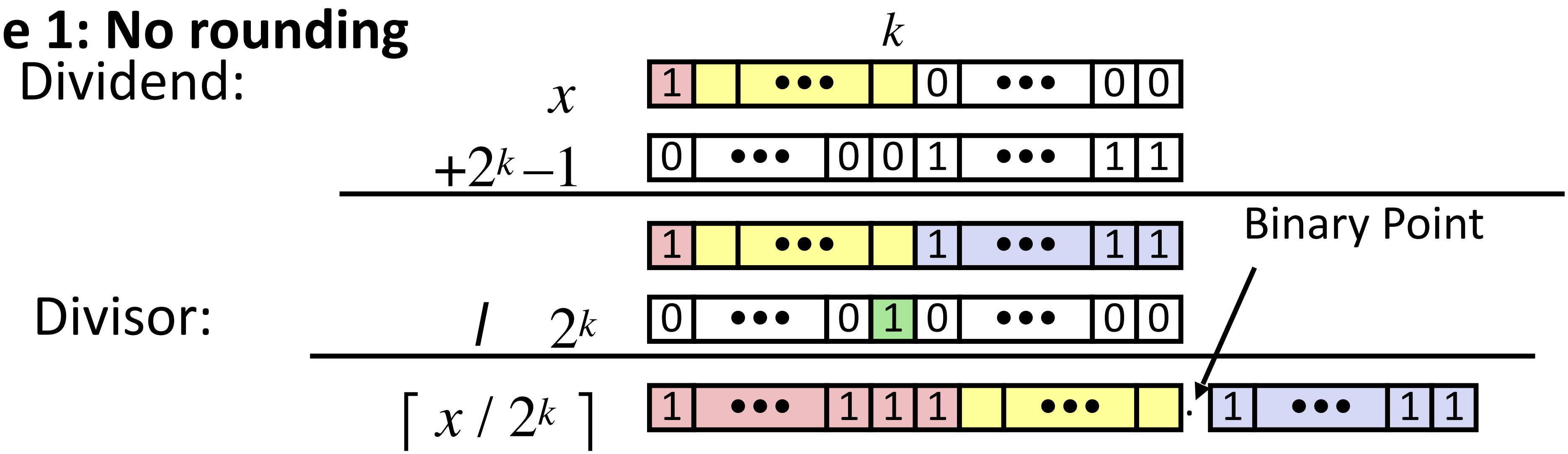
	Division	Computed	Hex	Binary
`y`	-15213	-15213	`C4 93`	`11000100 10010011`
`y >> 1`	-7606.5	-7607	`E2 49`	`11100010 01001001`
`y >> 4`	-950.8125	-951	`FC 49`	`11111100 01001001`
`y >> 8`	-59.4257813	-60	`FF C4`	`11111111 11000100`

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

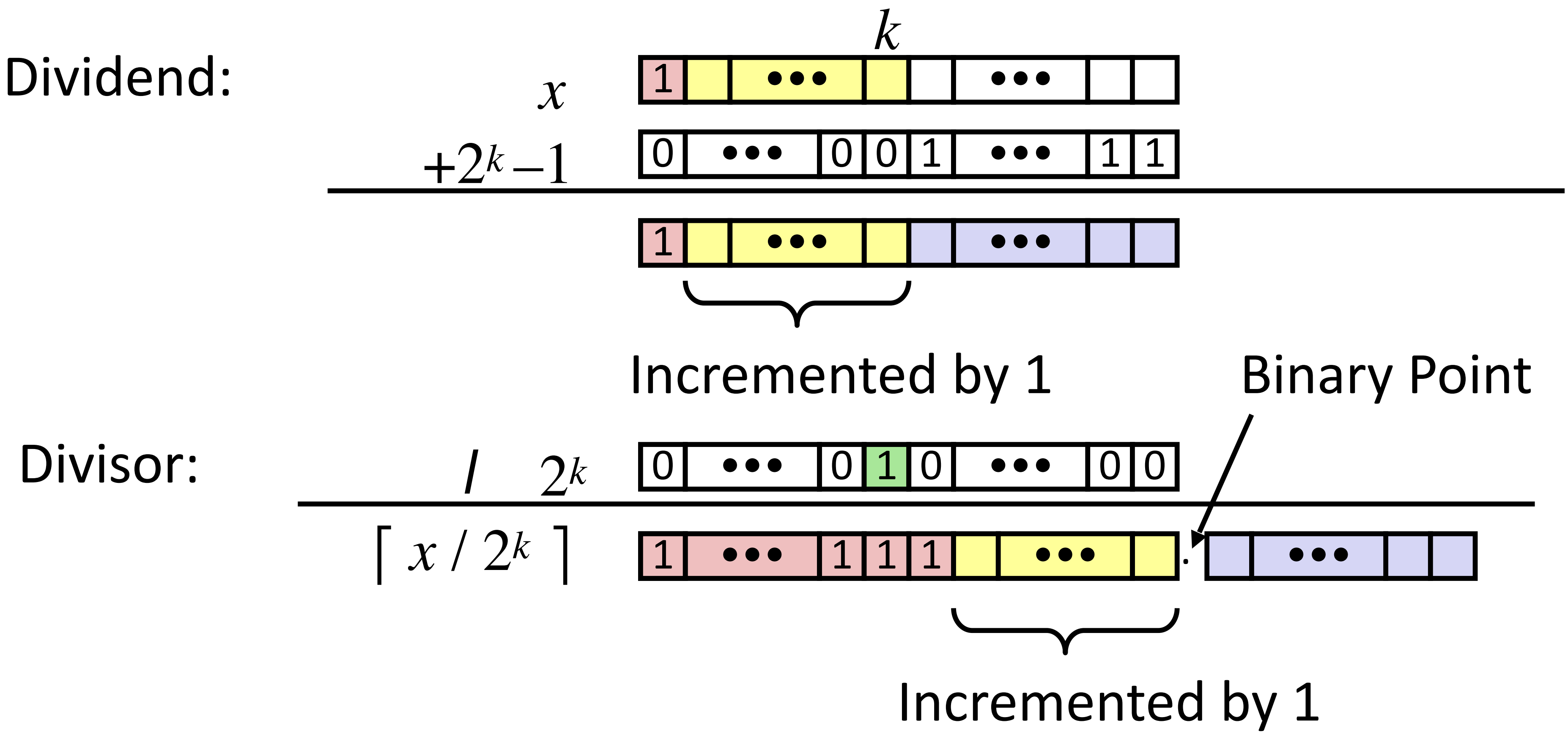
Case 1: No rounding



Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding



Biasing adds 1 to final result

Correct Power-of-2 Divide (Example)

-17 / 16

Dividend:

x	101111	-17
$+2^k - 1$	1111	Bias (1 << 4) - 1
	111110	

Incremented by 1

Arithmetic Shift right by 4 111110 >> 4

$\lceil x / 2^k \rceil = \lceil -17 / 2^4 \rceil = -1$ 11 . 1111

Binary Point

Incremented by 1

Biasing adds 1 to final result

Negation: Complement & Increment

- **Negate through complement and increase**

  `~x + 1 == -x (works for both signed and unsigned)`

- **Example**
  - 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

x = 15213

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

Complement & Increment Examples (Special Cases)

x = 0

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

Same for unsigned

x = TMin (Signed Two's Complement)

	Decimal	Hex	Binary
x	-32768	80 00	10000000 00000000
~x	32767	7F FF	01111111 11111111
~x+1	-32768	80 00	10000000 00000000

Bits, Bytes, and Integers

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

    previous lecture

    ---

    this lecture
  - Addition, negation, multiplication, shifting
  - **Summary**
- **Representations in memory, pointers, strings**

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 2^w
  - Signed: modified addition mod 2^w (result in proper range)
    - Mathematical addition + possible addition or subtraction of 2^w

- **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)

Why Should I Use Unsigned?

- ***Don't*** **use without understanding implications**
  - Easy to make mistakes

```
unsigned i;
for (i = cnt-2; i >= 0; i--)
  a[i] += a[i+1];
```

  - Can be very subtle

```
#define DELTA sizeof(int)
int i;
for (i = CNT; i-DELTA >= 0; i-= DELTA)
  . . .
```

Counting Down with Unsigned

- **Proper way to use unsigned as loop index**

```
unsigned i;
for (i = cnt-2; i < cnt; i--)
  a[i] += a[i+1];
```

- **See Robert Seacord, *Secure Coding in C and C++***
  - C Standard guarantees that unsigned addition will behave like modular arithmetic
    - 0 – 1 → *UMax*

- **Even better**

```
size_t i;
for (i = cnt-2; i < cnt; i--)
  a[i] += a[i+1];
```

  - Data type `size_t` defined as unsigned value with length = word size
  - **Code will work even if cnt == UMax**

Why Should I Use Unsigned? (cont.)

- ***Do*** **Use When Performing Modular Arithmetic**
  - Multiprecision arithmetic
- ***Do*** **Use When Using Bits to Represent Sets**
  - Logical right shift, no sign extension
- ***Do*** **Use In System Programming**
  - Bit masks, device commands,…

—> unsigned numbers can be very **useful** and C language supports it (**Java** does not support unsigned numbers)

Integer C Puzzles

`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`			✗

Initialization

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

Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating

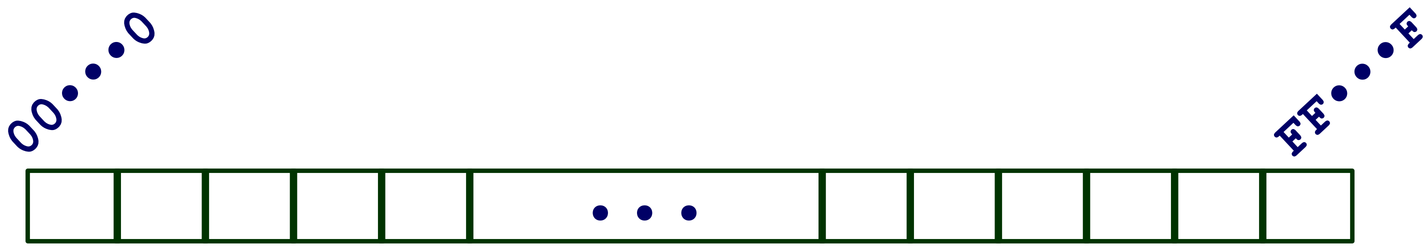
previous lecture

---

this lecture

  - Addition, negation, multiplication, shifting
  - Summary
- **Representations in memory, pointers, strings**

Byte-Oriented Memory Organization



- **Programs refer to data by address**
  - Conceptually, envision it as a very large array of bytes
    - In reality, it's not, but can think of it that way
  - An address is like an index into that array
    - and, a pointer variable stores an address

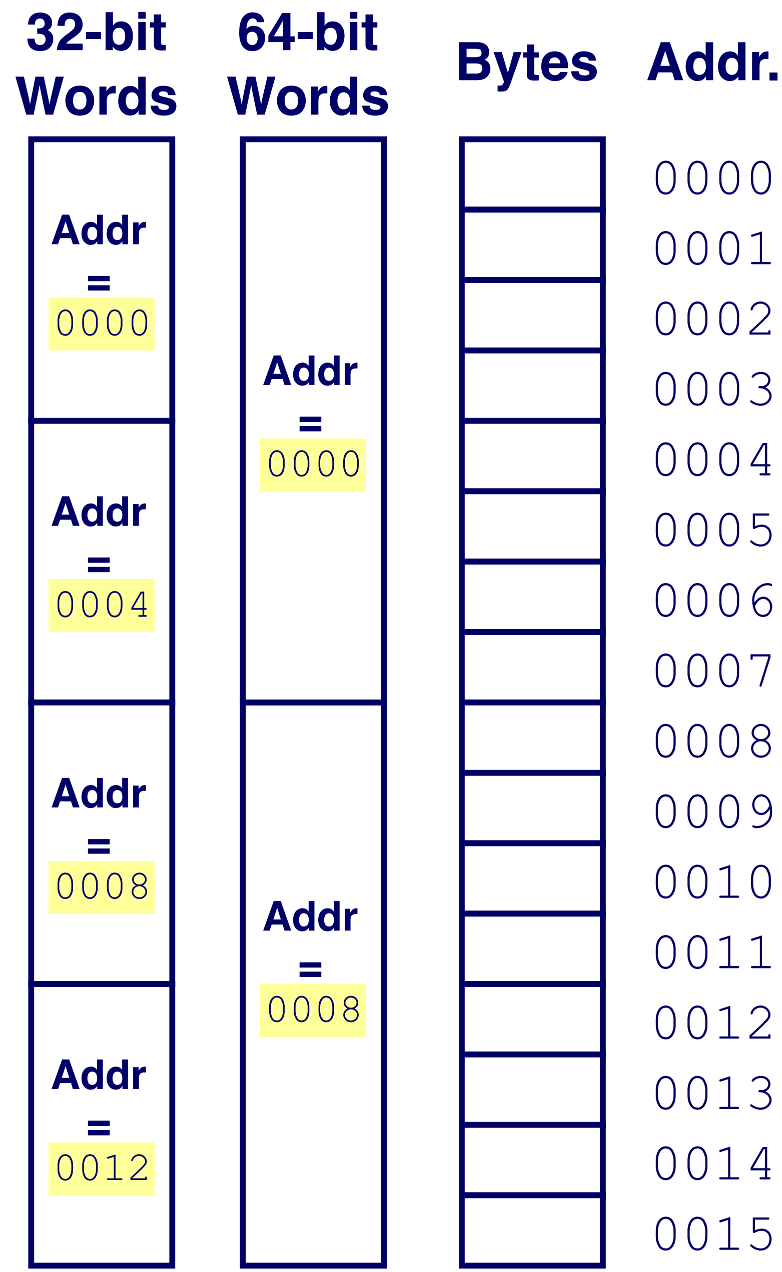
- **Note: system provides private address spaces to each "process"**
  - Think of a process as a program being executed
  - So, a program can clobber its own data, but not that of others

Machine Words

- **Any given computer has a "Word Size"**
  - Nominal size of integer-valued data
    - and of addresses
  - Until recently, most machines used 32 bits (4 bytes) as word size
    - Limits addresses to 4GB (2^{32} bytes)
  - Increasingly, machines have 64-bit word size
    - Potentially, could have 18 EB (exabytes) of addressable memory
    - That's 18.4 X 10^{18}
  - Machines still support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes

Word-Oriented Memory Organization

- **Addresses Specify Byte Locations**
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



Example Data Representations

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
`char`	1	1	1
`short`	2	2	2
`int`	4	4	4
`long`	4	8	8
`float`	4	4	4
`double`	8	8	8
pointer	4	8	8

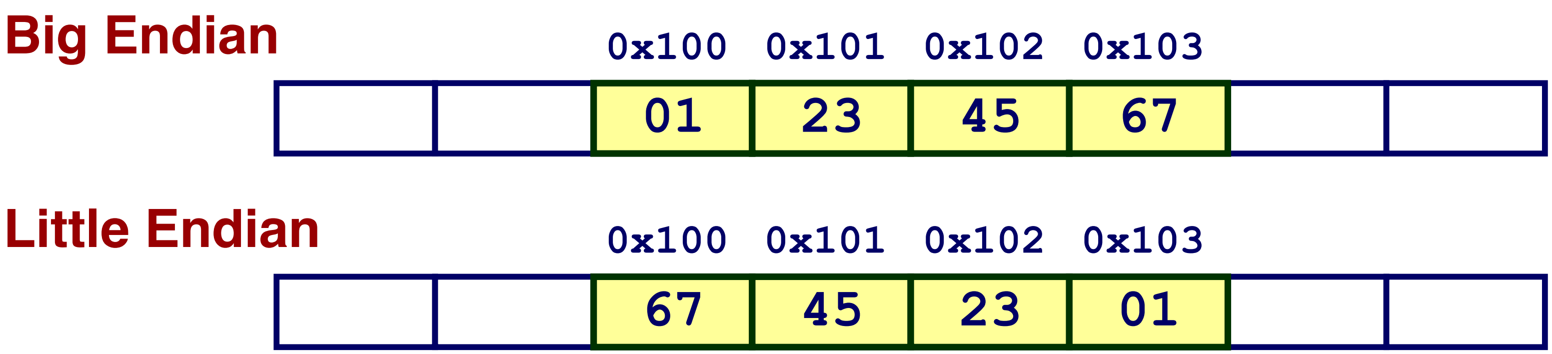
Byte Ordering

- **So, how are the bytes within a multi-byte word ordered in memory?**
- **Conventions**
  - Big Endian: Sun (Oracle SPARC), PPC Mac, *Internet*
    - Least significant byte has highest address
  - Little Endian: *x86*, ARM processors running Android, iOS, and Linux
    - Least significant byte has lowest address

Byte Ordering Example

Example

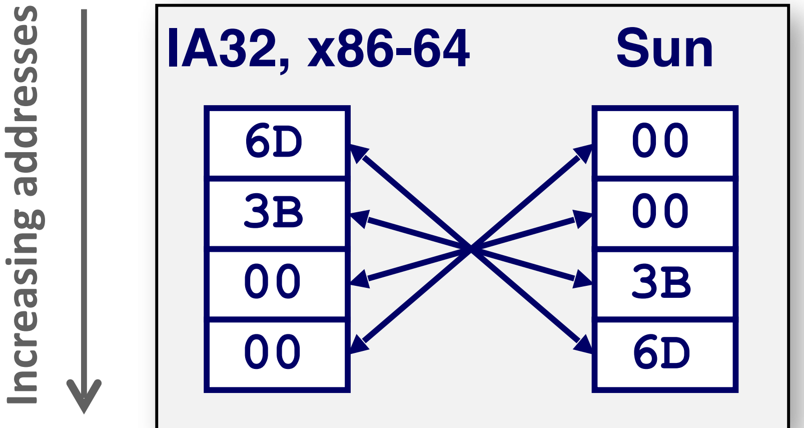
- Variable x has 4-byte value of 0x01234567
- Address given by &x is 0x100



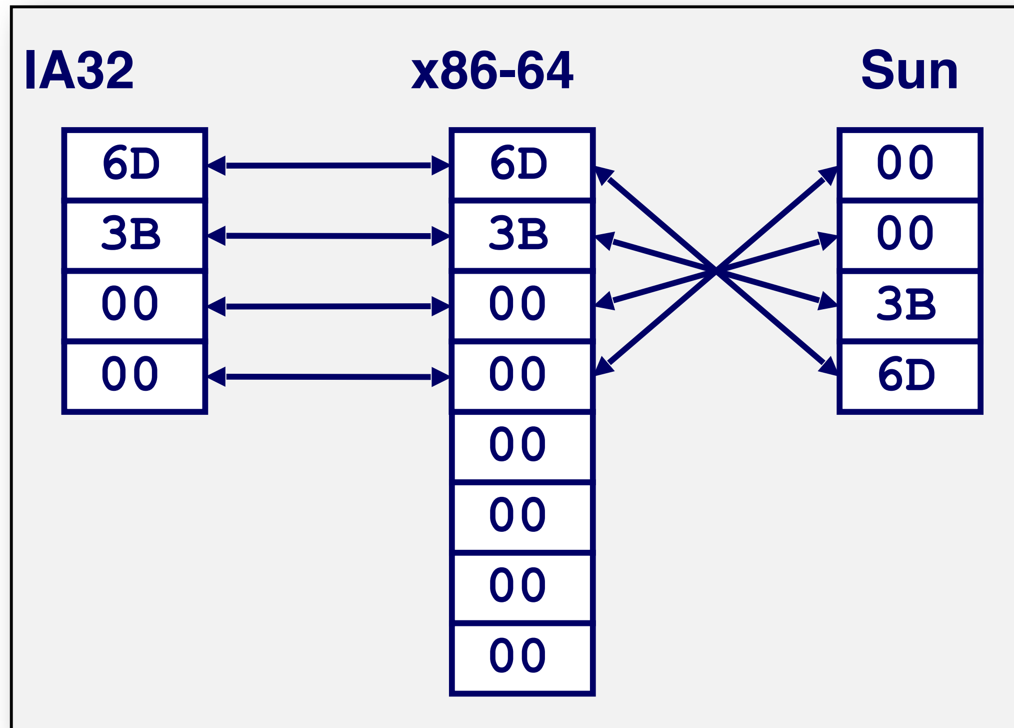
Representing Integers

Decimal:	`15213`			
Binary:	`0011`	`1011`	`0110`	`1101`
Hex:	`3`	`B`	`6`	`D`

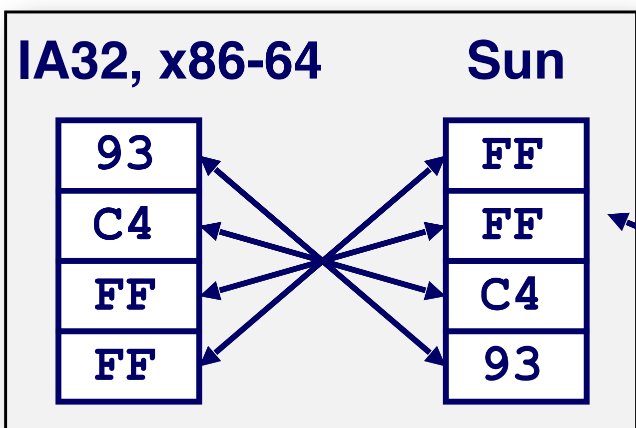
```
int A = 15213;
```




```
long int C = 15213;
```




```
int B = -15213;
```



Two's complement representation

Examining Data Representations

- **Code to Print Byte Representation of Data**
  - Casting pointer to unsigned char * allows treatment as a byte array

```
typedef unsigned char *pointer;

void show_bytes(pointer start, size_t len){
  size_t i;
  for (i = 0; i < len; i++)
    printf("%p\t0x%.2x\n",start+i, start[i]);
  printf("\n");
}
```

Printf directives:

%p: Print pointer

%x: Print Hexadecimal

`show_bytes` Execution Example

```
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux x86-64):

```
int a = 15213;
0x7fffb7f71dbc    6d
0x7fffb7f71dbd    3b
0x7fffb7f71dbe    00
0x7fffb7f71dbf    00
```

Representing Pointers

```
int B = -15213;
int *P = &B;
```

Sun	IA32	x86-64
EF	AC	3C
FF	28	1B
FB	F5	FE
2C	FF	82
		FD
		7F
		00
		00

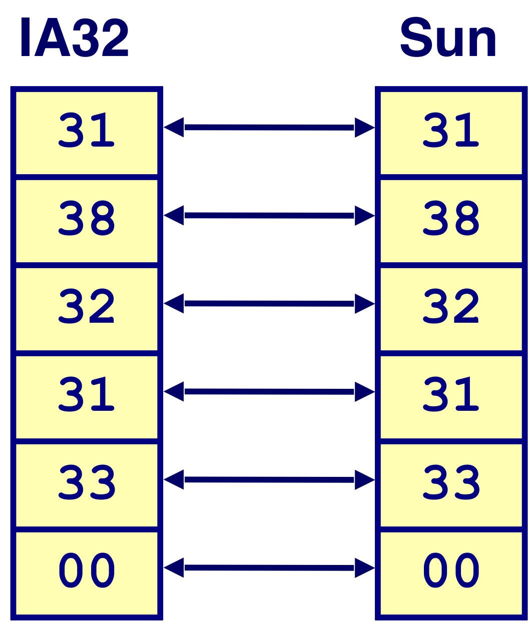
Different compilers & machines assign different locations to objects

Even get different results each time run program

Representing Strings

```
char S[6] = "18213";
```

- **Strings in C**
  - Represented by array of characters
  - Each character encoded in ASCII format
    - Standard 7-bit encoding of character set
    - Character “0” has code 0x30
      - Digit *i* has code 0x30+*I*
    - ***man ascii*** *for code table*
  - String should be null-terminated
    - Final character = 0
- **Compatibility**
  - Byte ordering not an issue



Reading Byte-Reversed Listings

Disassembly

- Text representation of binary machine code
- Generated by program that reads the machine code

Example Fragment

Address	Instruction Code	Assembly Rendition
`8048365:`	`5b`	`pop    %ebx`
`8048366:`	`81 c3 ab 12 00 00`	`add    $0x12ab,%ebx`
`804836c:`	`83 bb 28 00 00 00 00`	`cmpl   $0x0,0x28(%ebx)`

Deciphering Numbers

- Value: `0x12ab`
- Pad to 32 bits: `0x000012ab`
- Split into bytes: `00 00 12 ab`
- Reverse: `ab 12 00 00`

Summary

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