Bits, Bytes, and Integers – Part 2

15-213: Introduction to Computer Systems
3rd Lecture, Sept. 5, 2017

Today's Instructor:

Randy Bryant

Summary From Last Lecture

- 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

Encoding Integers

Unsigned

$$D4\,W^*Z+ \;=\; \sum_{K=2}^{Y-3} Z_K \cdot 4^K$$

Two's Complement

$$D4\,V^*Z+ \;=\; -\,Z_{Y-3}\cdot 4^{Y-3} + \sum_{K=2}^{Y-4} Z_K \cdot 4^K$$

Sign Bit

Two's Complement Examples (w = 5)

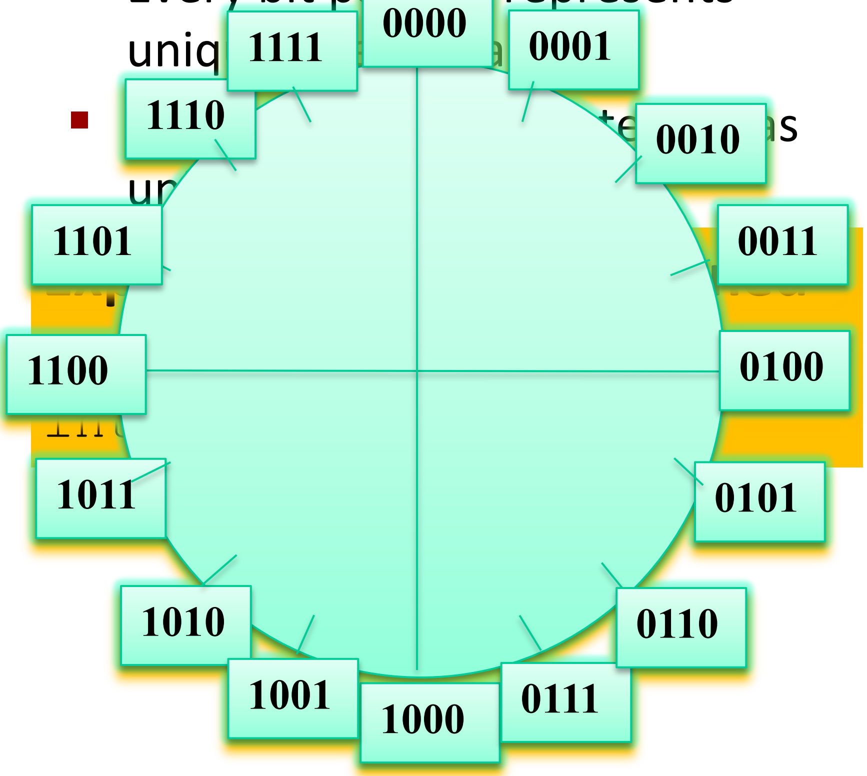
		-16	8	4	2	1	
10	=	0	1	0	1	0	8+2 = 10

		-16	8	4	2	1	
-10	=	1	0	1	1	0	-16+4+2 = -10

Unsigned & Signed Numeric Values

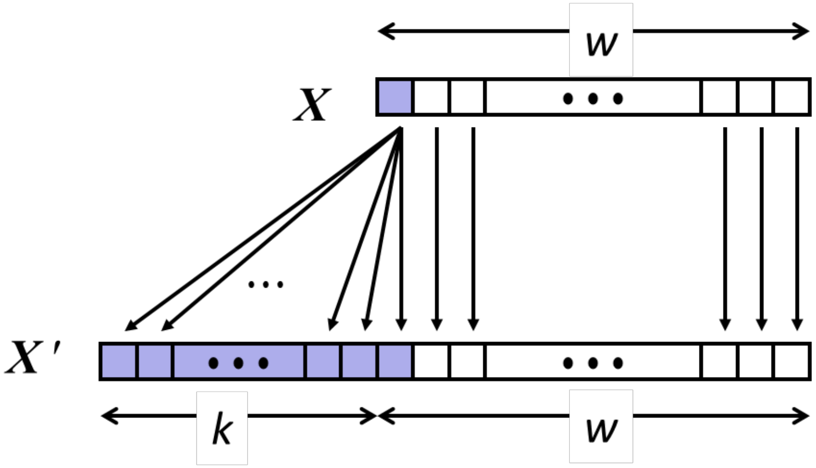
X	B2U(*X*)	B2T(*X*)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

- **Equivalence**
  - Same encodings for nonnegative values
- **Uniqueness**
  - Every bit pattern represents uniq a
  - te as un
- Exp ted
  - Inv

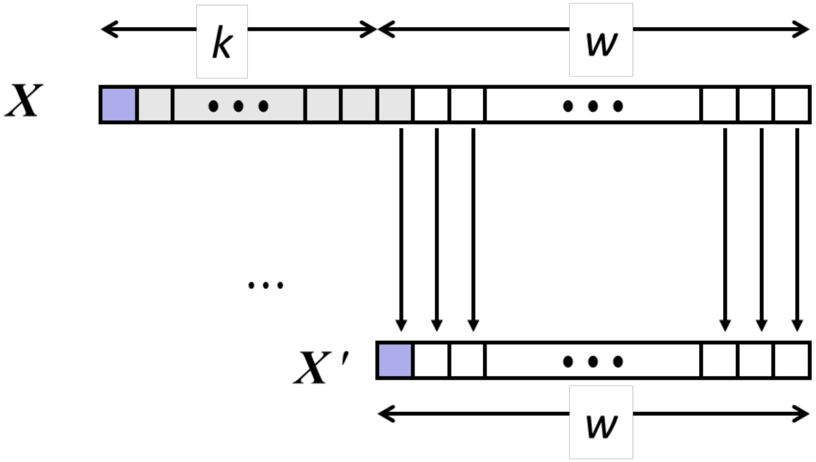


Sign Extension and Truncation

- **Sign Extension**



- **Truncation**



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

数据的运算

- 高级语言程序中涉及的运算（以C语言为例）
  - 整数算术运算、浮点数算术运算
  - 按位、逻辑、移位、位扩展和位截断
- 指令集中涉及到的运算
  - 涉及到的定点数运算
    - 算术运算
      - 带符号整数运算：取负 / 符号扩展 / 算术移位/ 加 / 减 / 乘 / 除
      - 无符号整数运算：0扩展 / 加 / 减 / 乘 / 除
    - 逻辑运算
      - 逻辑操作：与 / 或 / 非 / …
      - 移位操作：逻辑左移 / 逻辑右移
  - 涉及到的浮点数运算：加、减、乘、除
- 基本运算部件ALU的设计

如何实现高级语言源程序中的运算？

- **计算机如何实现高级语言程序中的运算？**
  - 将各类表达式编译（转换）为指令序列
  - 计算机直接执行指令来完成运算

例：C语言赋值语句 “f = (g+h) – (i+j);”中变量i、j、f、g、h由编译器分别分配给MIPS寄存器$t0~$t4。寄存器$t0~$t7的编号对应8~15，上述程序段对应的MIPS机器代码和汇编表示（#后为注释）如下：

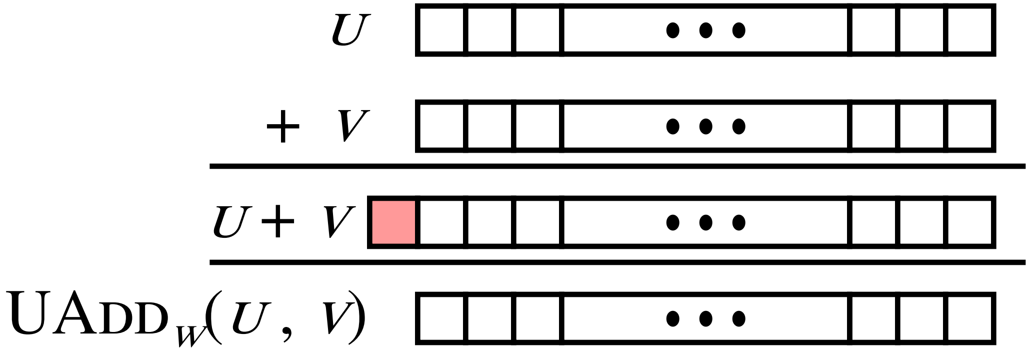
```
000000 01011 01100 01101 00000 100000   add $t5, $t3, $t4   # g+h
000000 01000 01001 01110 00000 100000  add $t6, $t0, $t1   # i+j
000000 01101 01110 01010 00000 100010  sub $t2, $t5, $t6   # f =(g+h)–(i+j)
```

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



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

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

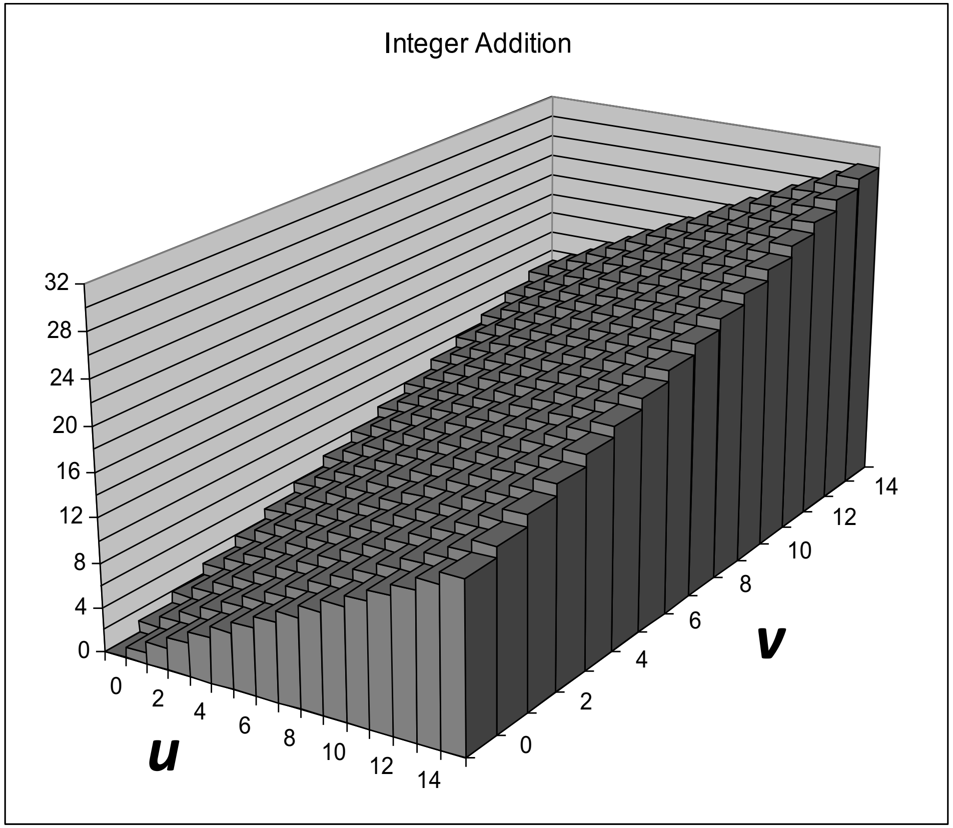
```
unsigned char      1110 1001      E9       223
                 +  1101 0101    + D5     + 213
                 1 1011 1110     1BE       446
                   1011 1110      BE       190
```

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

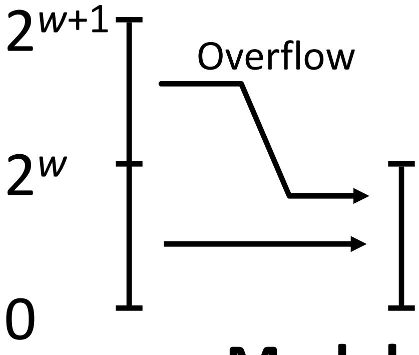
- **Integer Addition**
  - 4-bit integers *u*, *v*
  - Compute true sum $\mathrm{Add}_4(u\ ,\ v)$
  - Values increase linearly with *u* and *v*
  - Forms planar surface

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



Visualizing Unsigned Addition

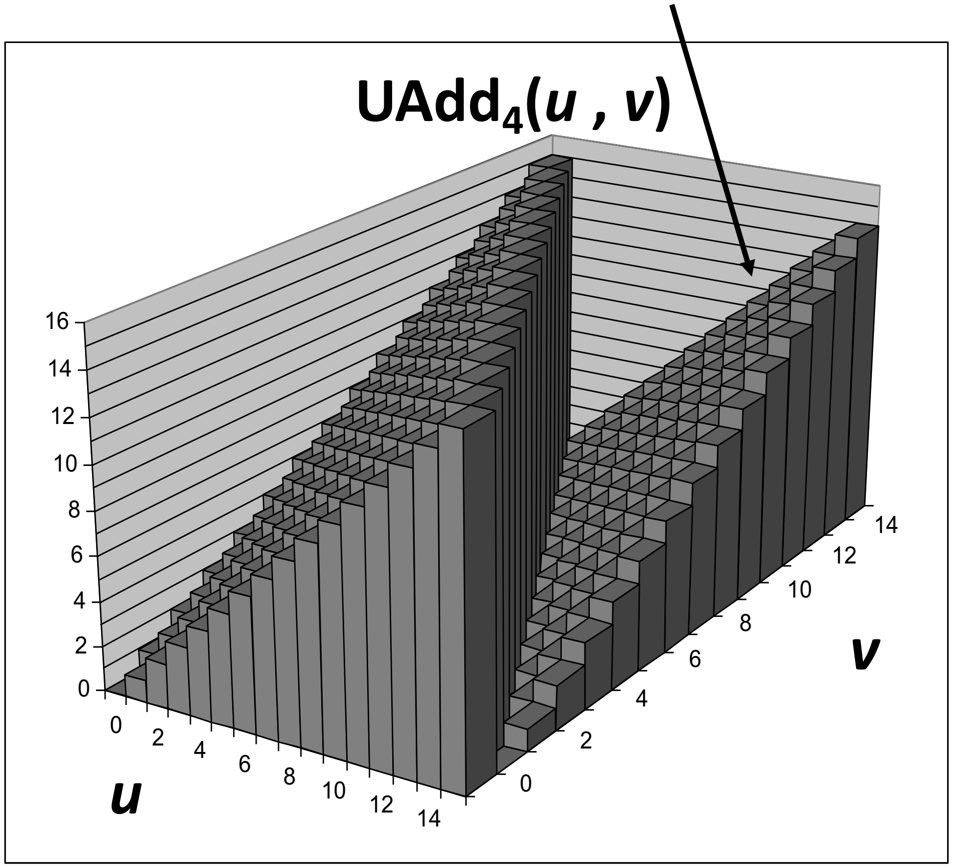
- **Wraps Around**
  - If true sum ≥ 2^w
  - At most once



True Sum

Modular Sum

Overflow

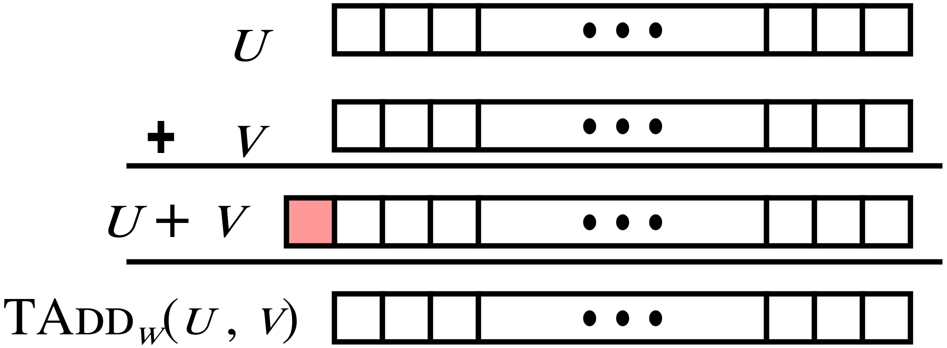


Two’s Complement Addition

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:

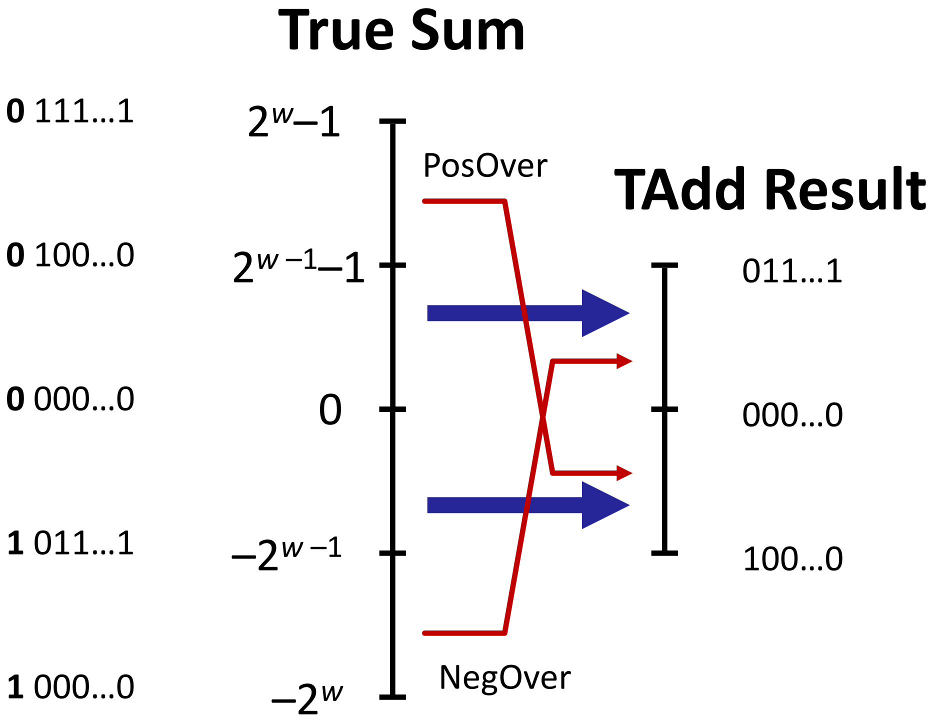
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

  - Will give `s == t`

```
     1110 1001       E9      -23
+    1101 0101     + D5    + -43
   1 1011 1110      1BE      446
     1011 1110       BE      -66
```

TAdd Overflow

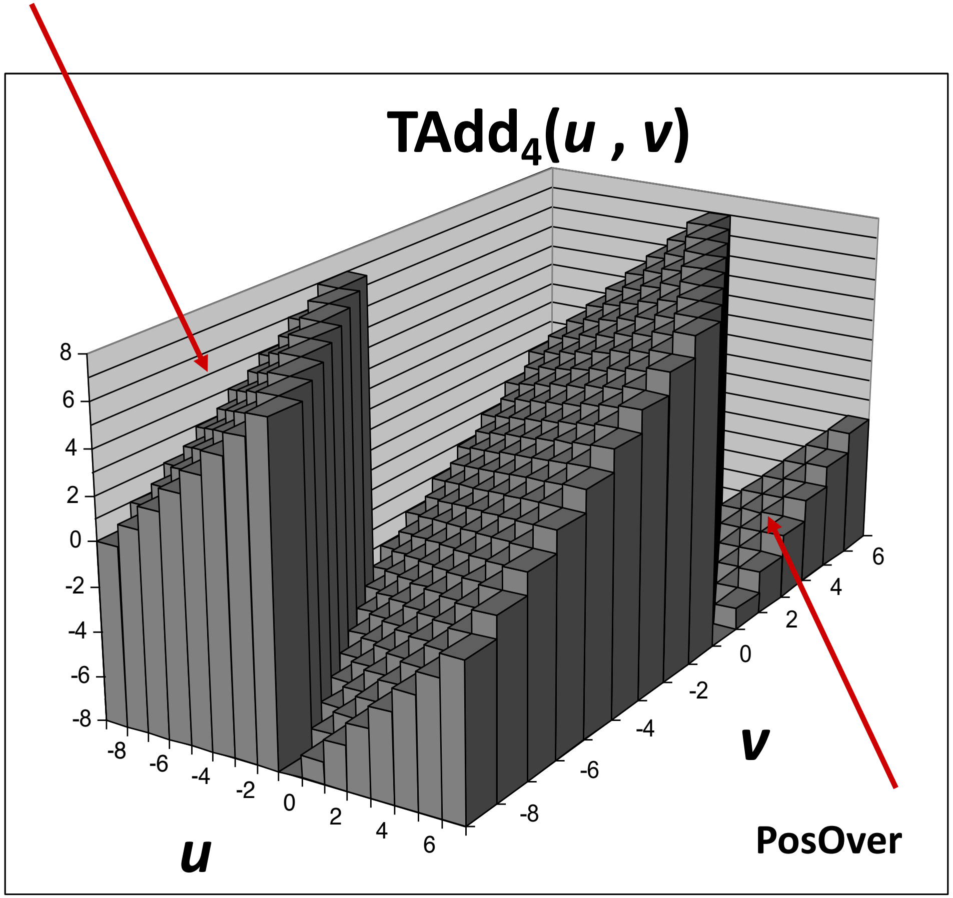
- **Functionality**
  - True sum requires *w*+1 bits
  - Drop off MSB
  - Treat remaining bits as 2’s comp. integer



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

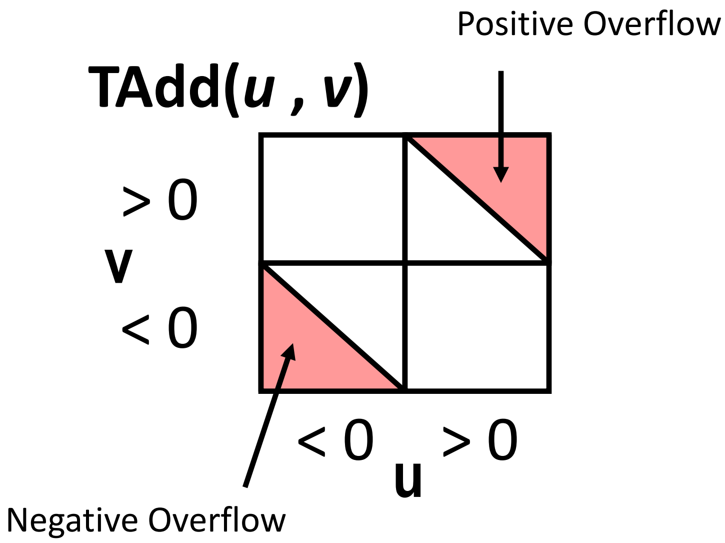
NegOver



PosOver

Characterizing TAdd

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



$$VCFF_Y{}^*W.X+ \;=\; \begin{cases} W\!\!+X+\;2^w & W\!\!+X< VO\,KP_Y \quad \textbf{(NegOver)} \\ W\!\!+X & VO\,KP_Y \le W\!\!+X\le\; VO\,CZ_Y \\ W\!\!+X-\;2^w & VO\,CZ_Y < W\!\!+X \quad \textbf{(PosOver)} \end{cases}$$

整数加减运算及其部件

- 补码加减运算公式
  - $[A+B]_{补} = [A]_{补} + [B]_{补}$ （MOD 2^n)
  - $[A–B]_{补} = [A]_{补} + [–B]_{补}$ （MOD 2^n)

问题：Adder中执行的是什么运算？

相当于无符号数加！

- 补码加减运算要点和运算部件
  - 加、减法运算统一采用加法来处理
  - 符号位(最高有效位MSB)和数值位一起参与运算
  - 直接用Adder实现两个数的加运算（模运算系统）

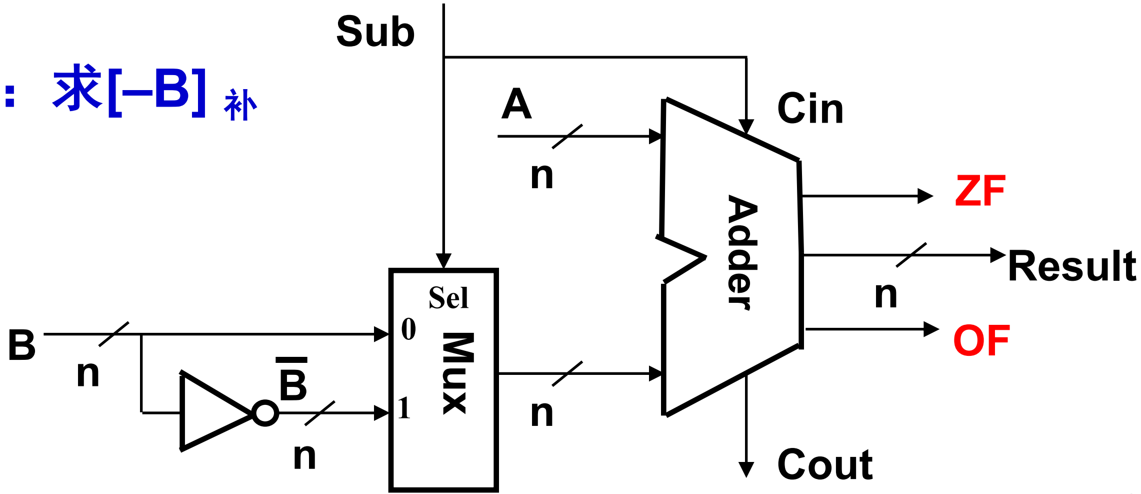
    问题：模是多少？运算结果高位丢弃，保留低n位，相当于取模2^n
  - 实现减法的主要工作在于：求$[–B]_{补}$

问题：如何求$[–B]_{补}$？

$[–B]_{补}=\overline{B}+1$

当Sub为1时，做减法

当Sub为0时，做加法

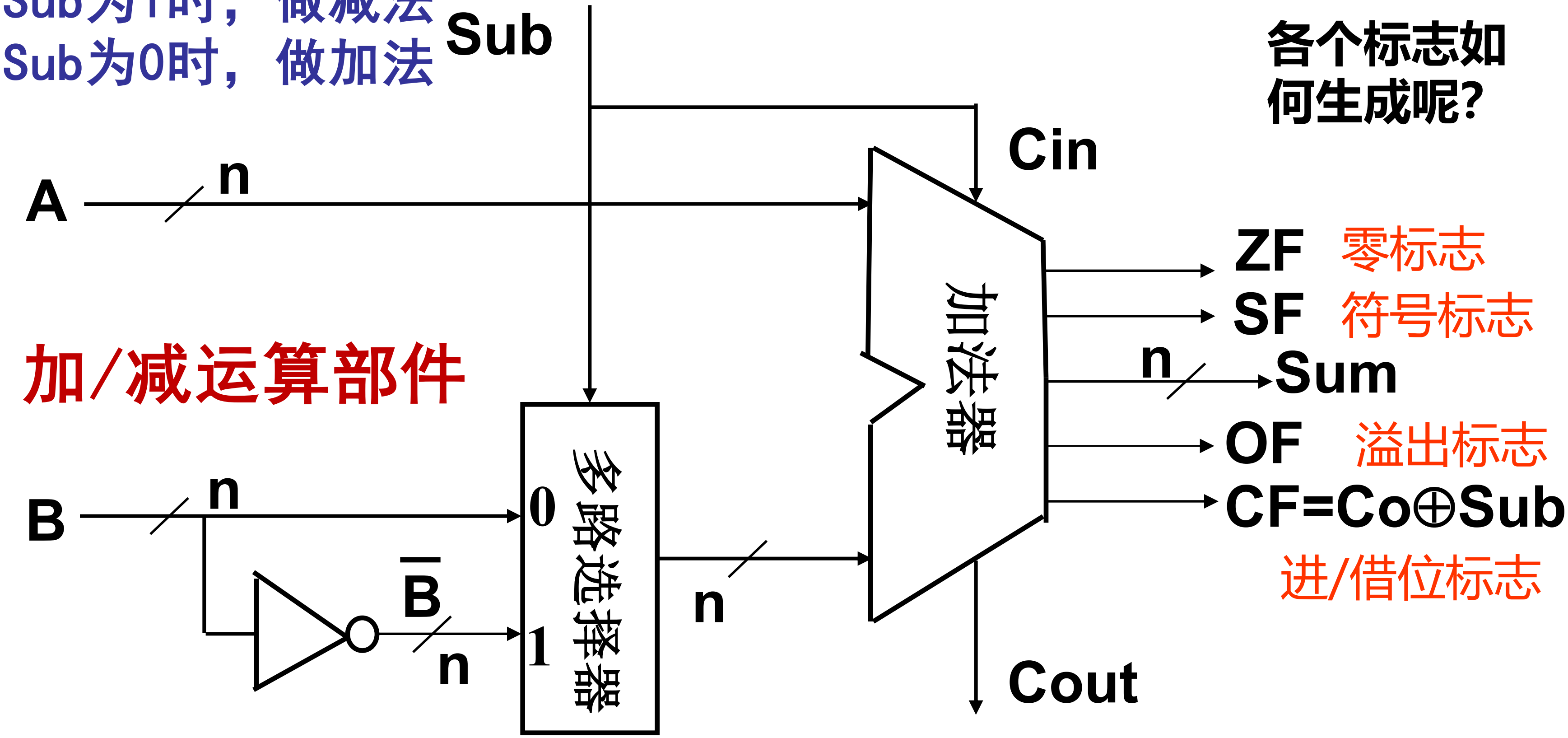


重要认识1：计算机中所有运算都基于加法器实现！

重要认识2：加法器不知道所运算的是带符号数还是无符号数。

重要认识3：加法器不判定对错，总是取低n位作为结果，并生成标志信息。

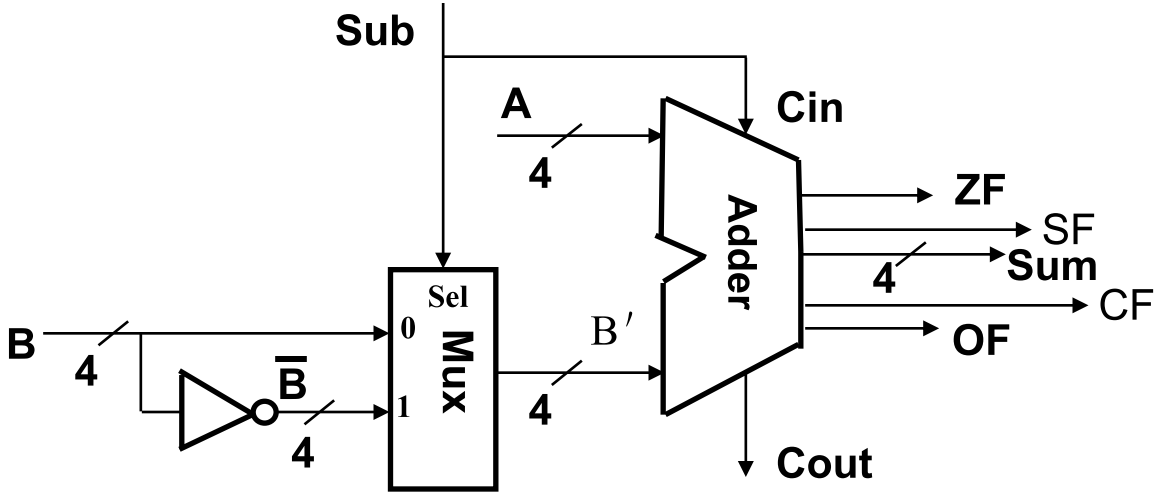
当Sub为1时，做减法
当Sub为0时，做加法



加/减运算部件

各个标志如何生成呢?

条件标志位（条件码CC）



所有其他运算都基于**整数加/减运算器**来实现。

整数加/减运算部件

问题：OF=？ZF=？SF=？CF=？

OF：若A与B’同号但与Sum不同号，则1；否则0。SF：sum符号
ZF：如Sum为0，则1，否则0。CF：Cout ⊕ sub

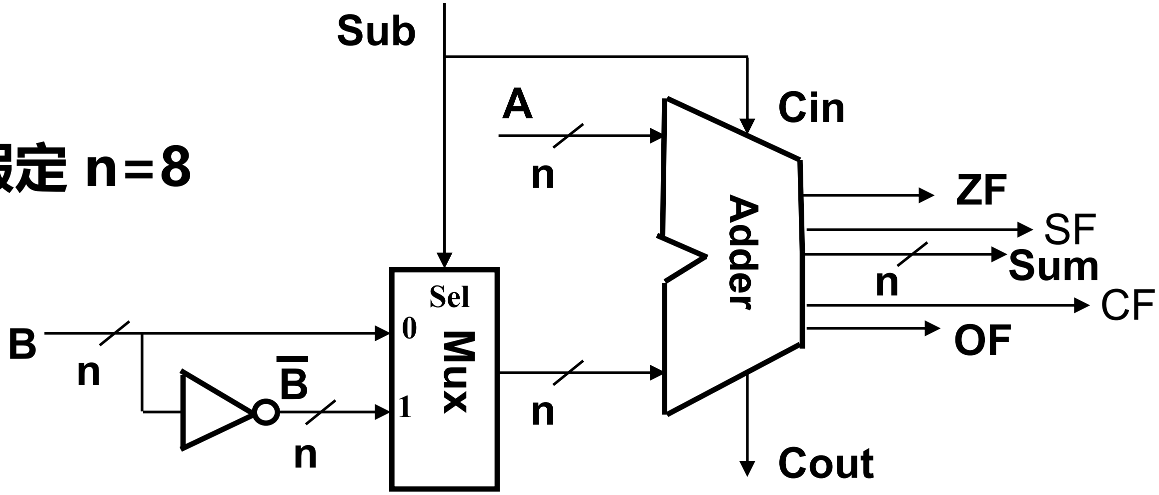
- 零标志**ZF**、溢出标志**OF**、进/借位标志**CF**、符号标志**SF**称为条件标志。
- **条件标志（Flag）**在运算电路中产生，被记录到专门的寄存器中，以便在分支指令中被用来作为条件。
- 存放标志的寄存器通常称为**程序/状态字寄存器**或**标志寄存器**。每个标志对应标志寄存器中的一个标志位。 **如，IA-32中的EFLAGS寄存器**

整数加减运算及其部件

```
unsigned int x=134;
unsigned int y=246;
int m=x;
int n=y;
unsigned int z1=x-y;
unsigned int z2=x+y;
int k1=m-n;
int k2=m+n;
```

无符号数加减运算也用该部件执行

假定 n=8



x和m的机器数一样：1000 0110，y和n的机器数一样：1111 0110

z1和k1的机器数一样：1001 0000，CF=1，OF=0，SF=1

z1的值为144（=134-246+256，x-y<0），k1的值为-112。

z2和k2的机器数一样：0111 1100，CF=1，OF=1，SF=0

z2的值为124（=134+246-256，x+y>256)

k2的值为124（=134+246-256，x+y>128，即正溢出)

结果说明什么?

仅k1的值正确!

Multiplication

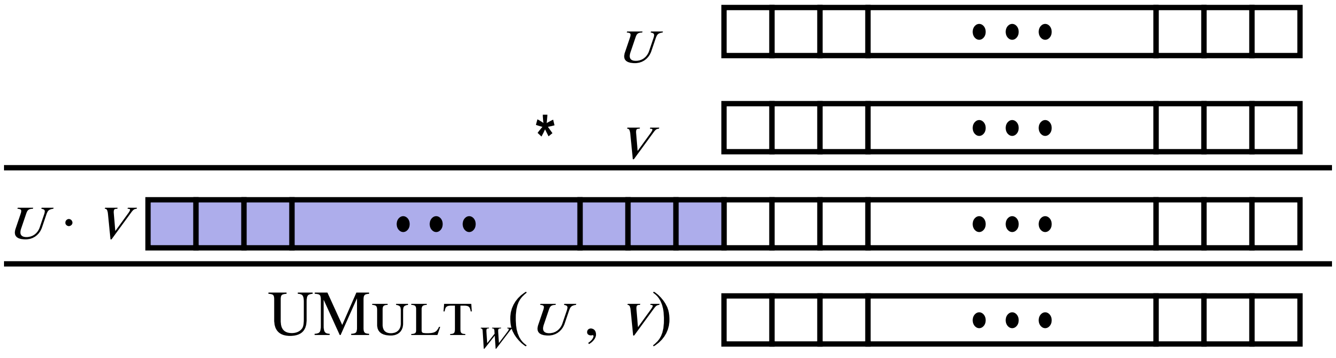
- **Goal: Computing Product of *w*-bit numbers *x*, *y***
  - Either signed or unsigned
- **But, exact results can be bigger than *w* bits**
  - Unsigned: up to 2*w* bits
    - Result range: $0 \leq x * y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Two's complement min (negative): Up to 2*w*-1 bits
    - Result range: $x * y \geq (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - Two's complement max (positive): Up to 2*w* bits, but only for $(TMin_w)^2$
    - Result range: $x * y \leq (-2^{w-1})^2 = 2^{2w-2}$
- **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**

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

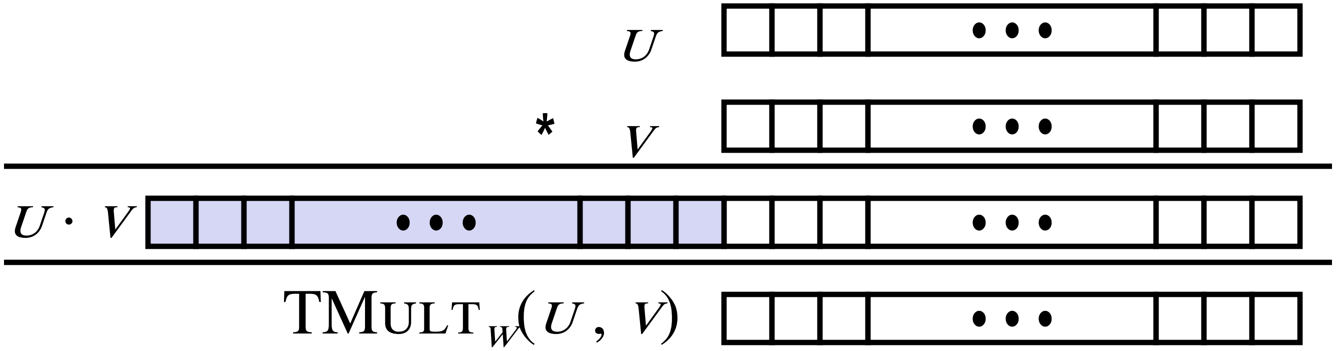
1110 1001	E9	223
* 1101 0101	* D5	* 213
1100 0001 1101 1101	C1DD	47499
1101 1101	DD	221

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
```

整数的乘运算

- 高级语言中两个n位整数相乘得到的结果通常也是一个n位整数，也即结果只取2n位乘积中的低n位。
- 例如，在C语言中，参加运算的两个操作数的类型和结果的类型必须一致，如果不一致则会先转换为一致的数据类型再进行计算。

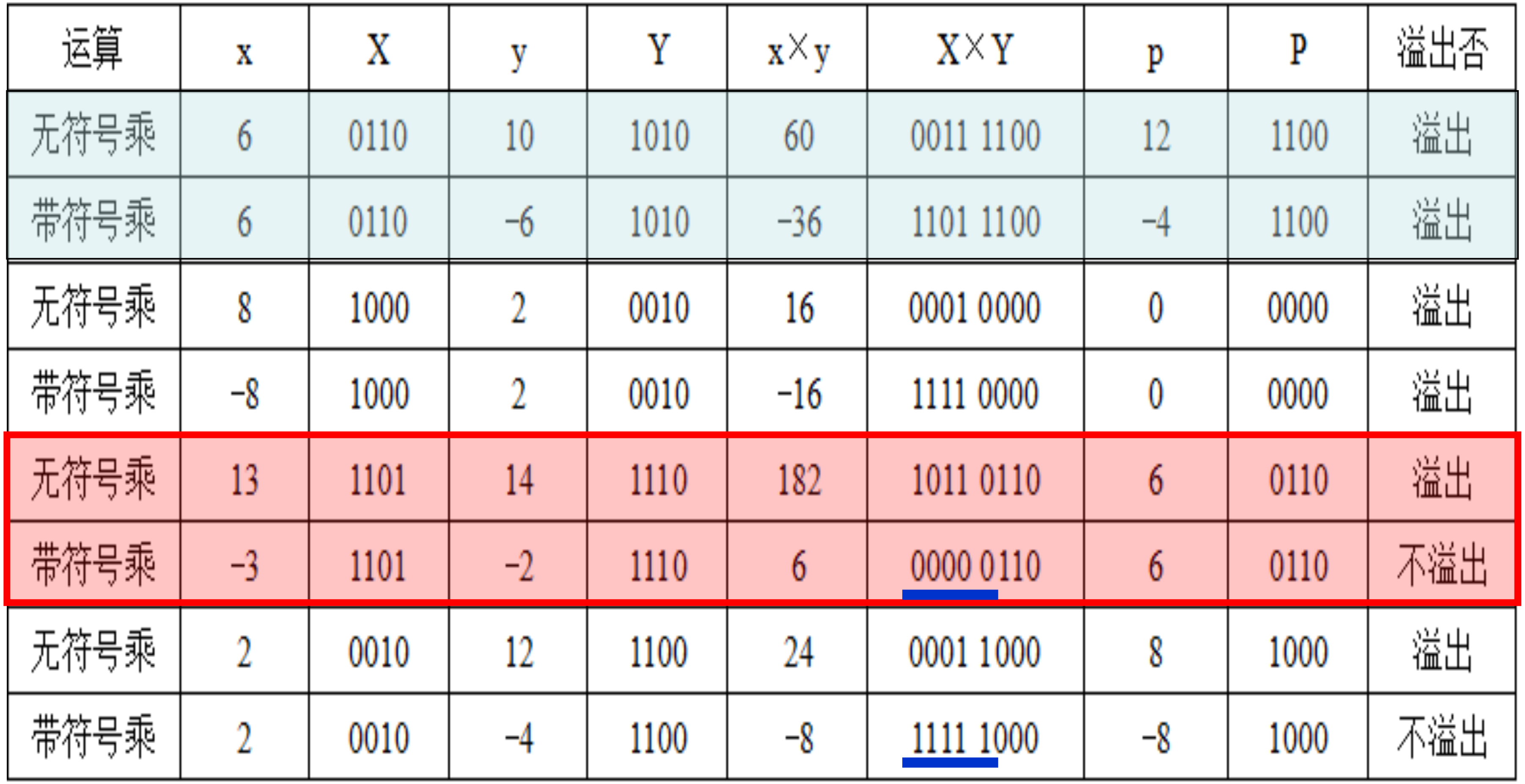
$x^2 \geq 0$?

对于带符号整数，不一定!

例如，当n=4时, $5^2=-7<0$!

整数的乘运算

- X×Y的高n位可以用来判断溢出，规则如下：
  - 无符号：若高n位全0，则不溢出，否则溢出
  - 带符号：若高n位全0或全1且等于低n位的最高位，则不溢出。



运算	x	X	y	Y	x×y	X×Y	p	P	溢出否
无符号乘	6	0110	10	1010	60	0011 1100	12	1100	溢出
带符号乘	6	0110	-6	1010	-36	1101 1100	-4	1100	溢出
无符号乘	8	1000	2	0010	16	0001 0000	0	0000	溢出
带符号乘	-8	1000	2	0010	-16	1111 0000	0	0000	溢出
无符号乘	13	1101	14	1110	182	1011 0110	6	0110	溢出
带符号乘	-3	1101	-2	1110	6	0000 0110	6	0110	不溢出
无符号乘	2	0010	12	1100	24	0001 1000	8	1000	溢出
带符号乘	2	0010	-4	1100	-8	1111 1000	-8	1000	不溢出

整数的乘运算

- 通常硬件不判断乘法是否溢出，而是保留2n位乘积
- 分无符号数乘指令和带符号整数乘指令
- 乘法指令无法得到溢出标志或无法自动判断是否溢出，如果程序本身不采用防止溢出的措施，而且编译器也不生成相应的用于溢出处理的代码的话，就会发生一些由于整数溢出而带来的问题。
- 乘法指令的操作数长度为n,而乘积长度为2n，例如：
  - IA-32中，若指令只给出一个操作数SRC，则另一个源操作数隐含在累加器AL/AX/EAX中，将SRC和累加器内容相乘，结果存放在AX（16位时）或DX-AX（32位时）或EDX-EAX（64位时）中。
  - 在MIPS处理器中，带符号整数乘法指令mult会将两个32位带符号整数相乘得到的64位乘积置于两个32位内部寄存器Hi和Lo中，因此，可以根据Hi寄存器中的每一位是否等于Lo寄存器中的第一位来进行溢出判断。

整数溢出漏洞

- 说明以下程序存在什么漏洞，引起该漏洞的原因是什么。

2002年，Sun Microsystems公司的RPC XDR库带的xdr_array函数发生整数溢出漏洞，攻击者可利用该漏洞从远程或本地获取root权限。

```
/* 复制数组到堆中，count为数组元素个数 */
int copy_array(int *array, int count) {
     int i;
    /* 在堆区申请一块内存 */
     int *myarray = (int *) malloc(count*sizeof(int));
    if (myarray == NULL)
        return -1;
    for (i = 0; i < count; i++)
        myarray[i] = array[i];
    return count;
}
```

攻击者可构造特殊参数来触发整数溢出，以一段预设信息覆盖一个已分配的堆缓冲区，造成远程服务崩溃或者改变内存数据并执行任意代码。

当参数count很大时，则count*sizeof(int)会溢出。如count=2^{30}+1时，count*sizeof(int)=4。

堆（heap）中大量数据被破坏！

变量与常数之间的乘运算

- 整数乘法运算比移位和加法等运算所用时间长得多，通常一次乘法运算需要10个左右时钟周期，而一次移位、加法和减法等运算只要一个或更少的时钟周期，因此，编译器在处理变量与常数相乘时，往往以移位、加法和减法的组合运算来代替乘法运算。

  例如，对于表达式x*20，编译器可以利用 $20=16+4=2^4+2^2$，将x*20转换为(x<<4)+(x<<2)，这样，一次乘法转换成了两次移位和一次加法。

- 不管是无符号数还是带符号整数的乘法，即使乘积溢出时，利用移位和加减运算组合的方式得到的结果都是和采用直接相乘的结果是一样的。

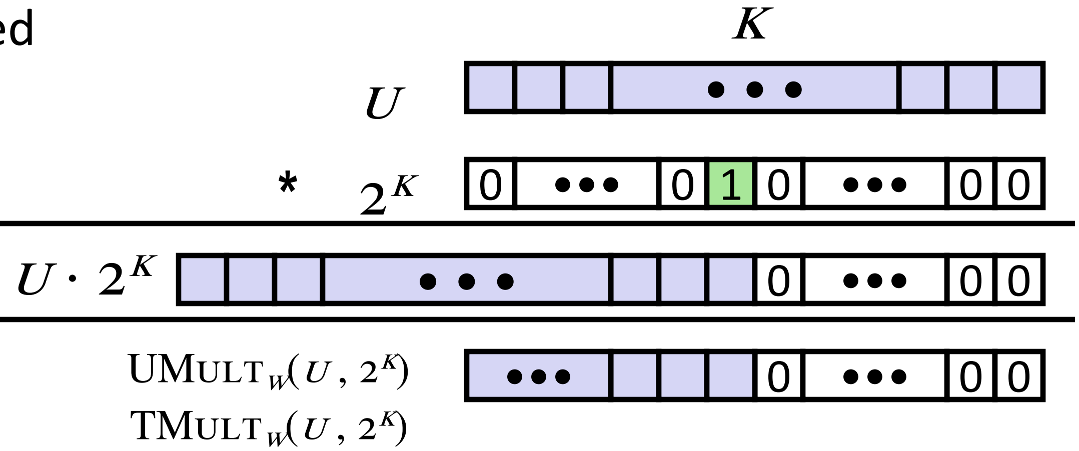
Power-of-2 Multiply with Shift

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

变量与常数之间的除运算

- 对于整数除法运算，由于计算机中除法运算比较复杂，而且不能用流水线方式实现，所以一次除法运算大致需要30个或更多个时钟周期。为了缩短除法运算的时间，编译器在处理一个变量与一个2的幂次形式的整数相除时，常采用右移运算来实现。
- 无符号数除法采用逻辑右移方式，带符号整数采用算术右移方式。
- 结果一定取整数，能整除时，直接右移得到结果。

例如，12/4=3：0000 1100>>2=0000 0011

-12/4=-3：1111 0100 >>2=1111 1101

- 不能整除时，其商采用朝零方向舍入的方式，也就是截断方式，即：移出的低位数直接丢弃。带符号负整数则不对！ （需加偏移量(2^k-1)，然后再右移*k*位 ，低位截断)

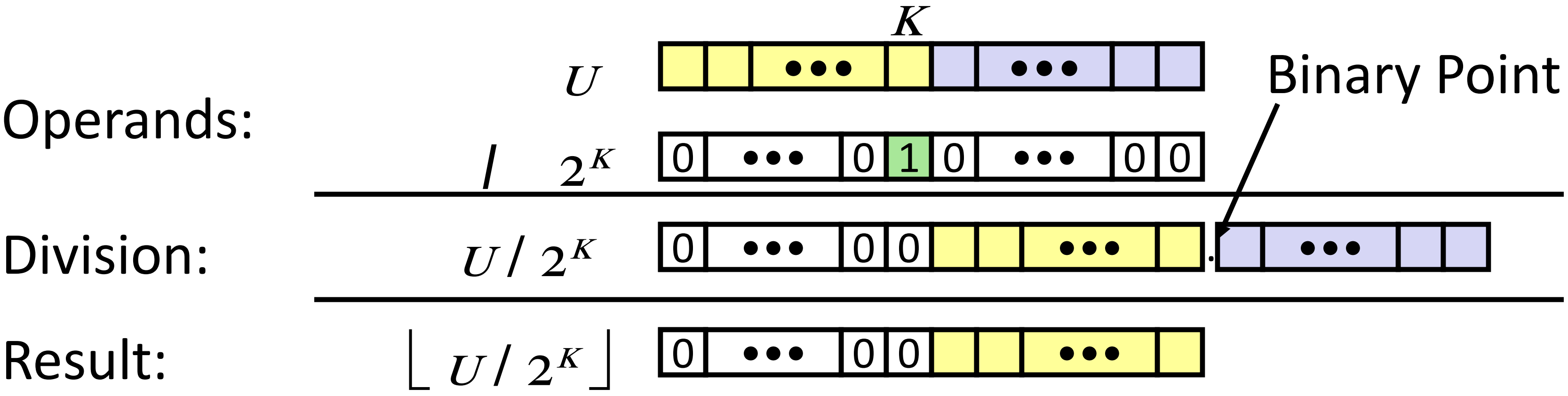
无符号整数：14/4=3：0000 1110>>2=0000 0011

带符号整数：-14/4=-3：1111 0010 >>2=1111 1100=-4≠-3

纠偏: k=2, 故(-14+2^2-1)/4=-3： 1111 0101>>2=1111 1101=-3

Unsigned Power-of-2 Divide with Shift

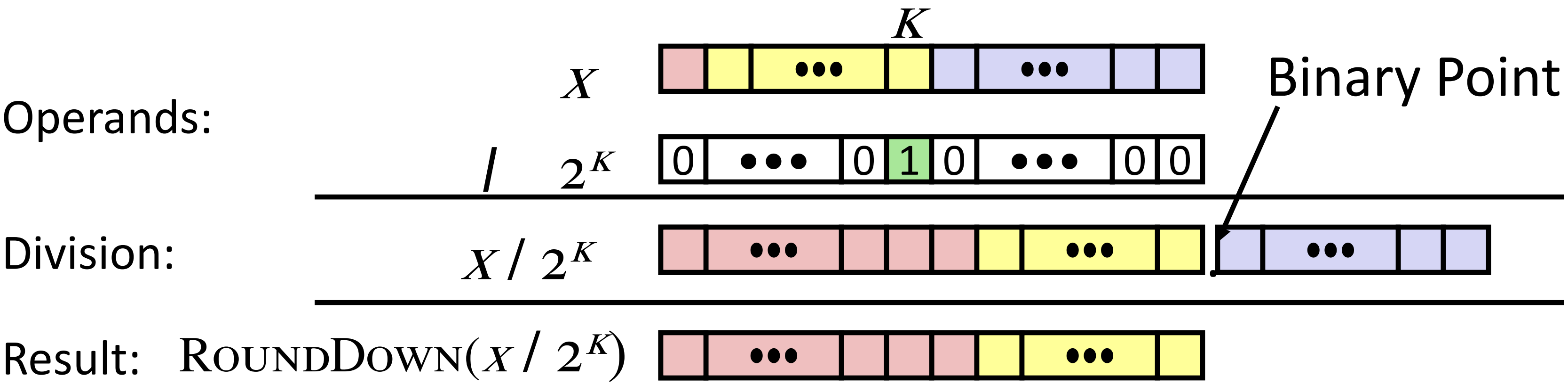
- **Quotient of Unsigned by Power of 2**
  - `u >> k` gives $\lfloor$ `u / ` $2^k \rfloor$
  - Uses logical shift



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



Operands: x ••• ••• with K above

$/\ 2^K$ 0 ••• 0 1 0 ••• 0 0

Binary Point

Division: $x / 2^K$ ••• ••• . •••

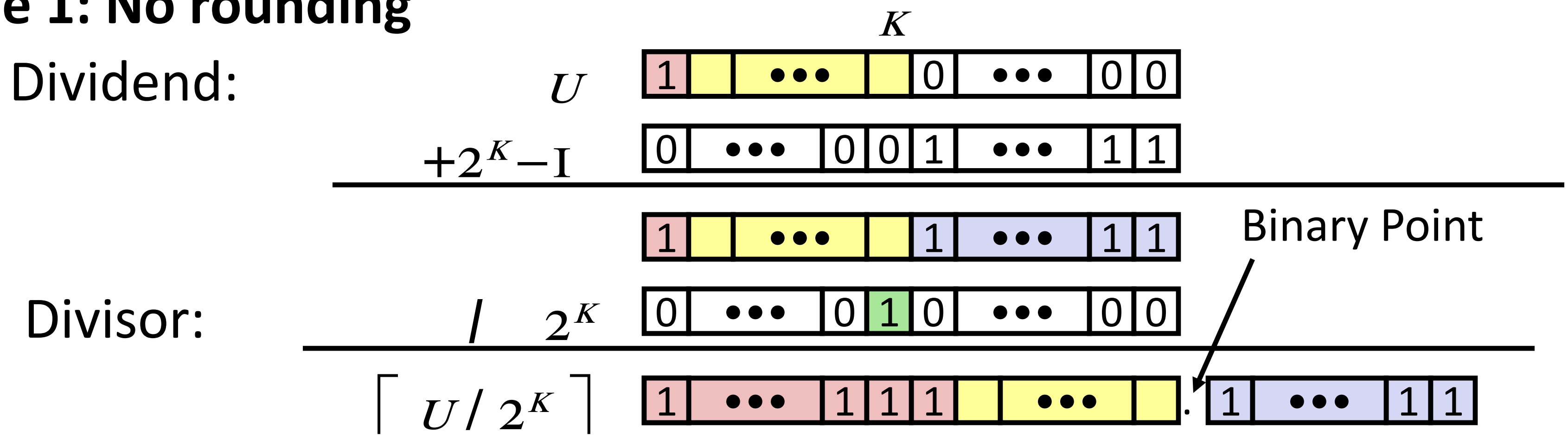
Result: $\text{RoundDown}(x / 2^K)$ ••• •••

	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

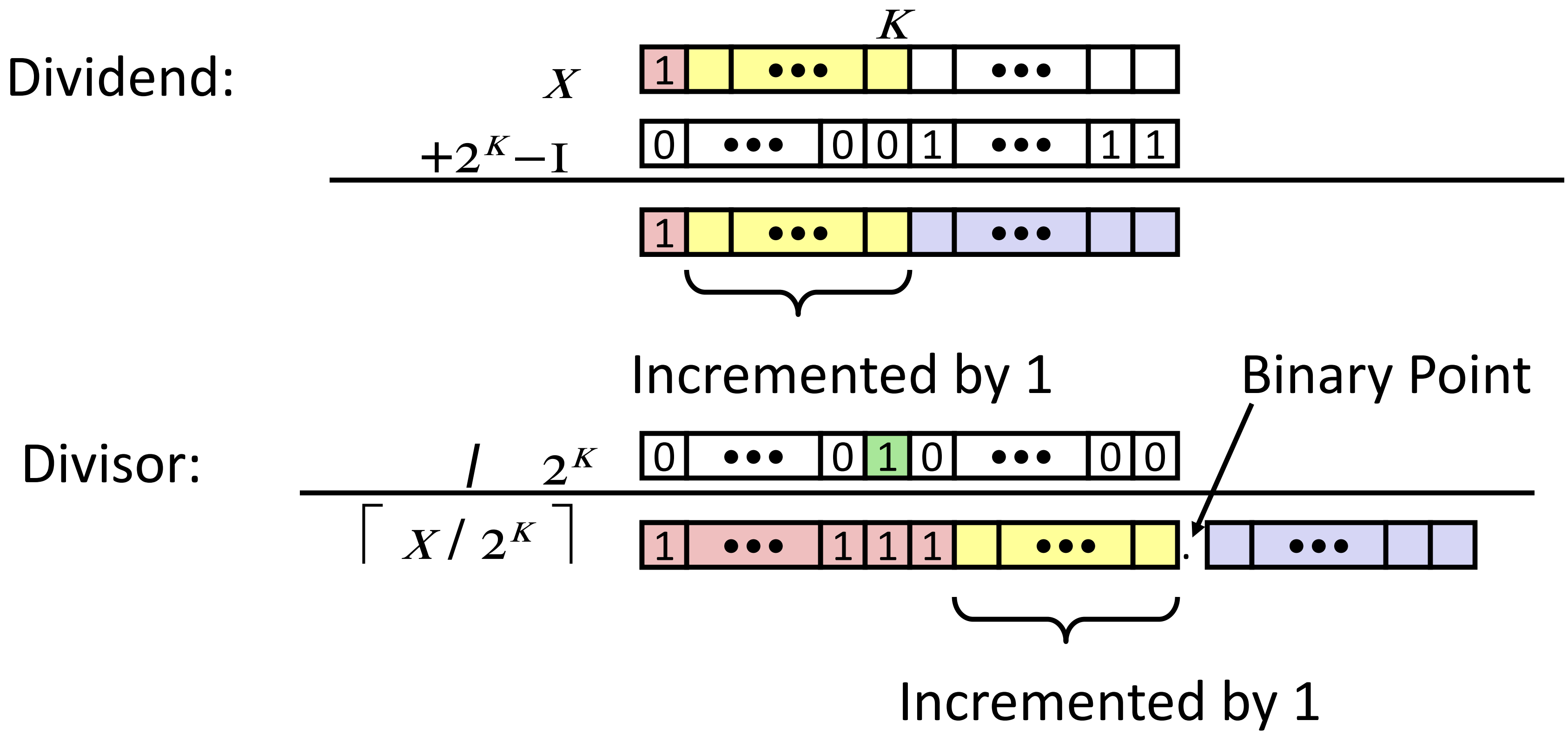
Case 1: No rounding



Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding



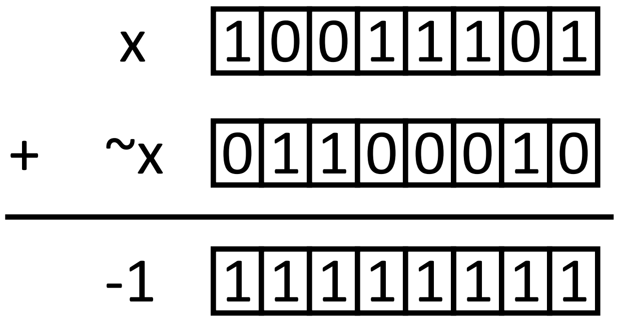
Biasing adds 1 to final result

Negation: Complement & Increment

- **Negate through complement and increase**

  `~x + 1 == -x`

- **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

x = 0

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

x = TMin

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

Canonical counter example

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**
- 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*
  - What if **`cnt`** is signed and < 0?

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,…

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
- **Representations in memory, pointers, strings**

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

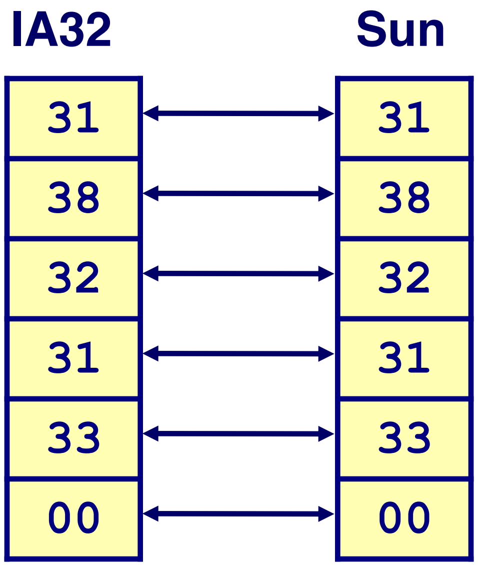
Different compilers & machines assign different locations to objects

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*
  - String should be null-terminated
    - Final character = 0
- **Compatibility**
  - Byte ordering not an issue



IA32	Sun
31	31
38	38
32	32
31	31
33	33
00	00

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`

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;
```

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