Integer Operations

Computer Systems Section 2.1.6-2.1.9,2.3

Abstraction

Computers Deal with bits of information

Ones and Zeroes

On and Off

True and False

Leaky Abstraction

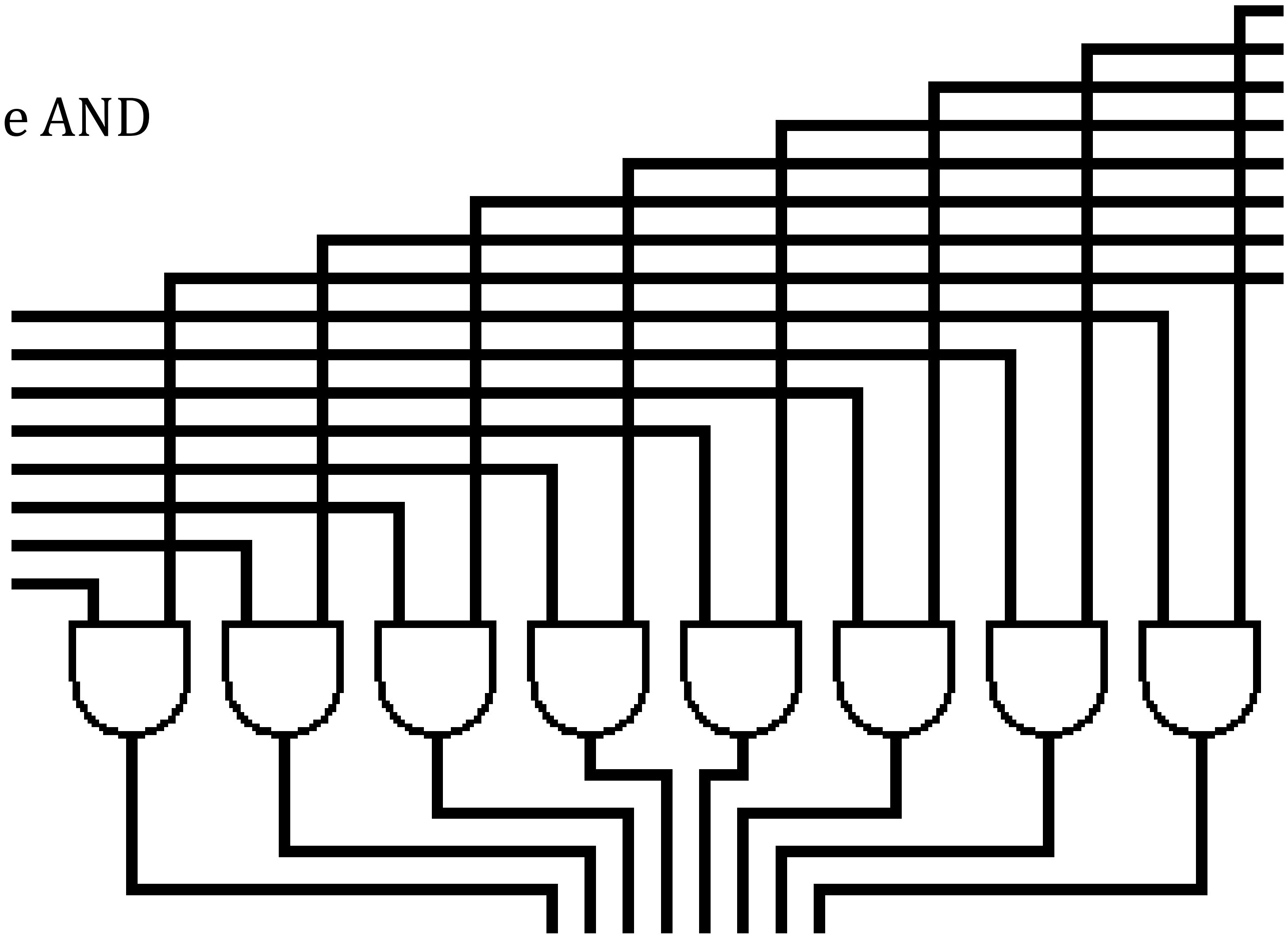
- Smallest addressable “element” in C is 8 bits!
- “bool” data type (using #include <stdbool.h>)
  - Takes 8 bits of storage
- Heavy use of bit-wise “AND” (&) and bit-wise “OR” (|)
  - Character masks expressed in hexadecimal, e.g. “0x02”

A	B	A&B
0	0	0
0	1	0
1	0	0
1	1	1

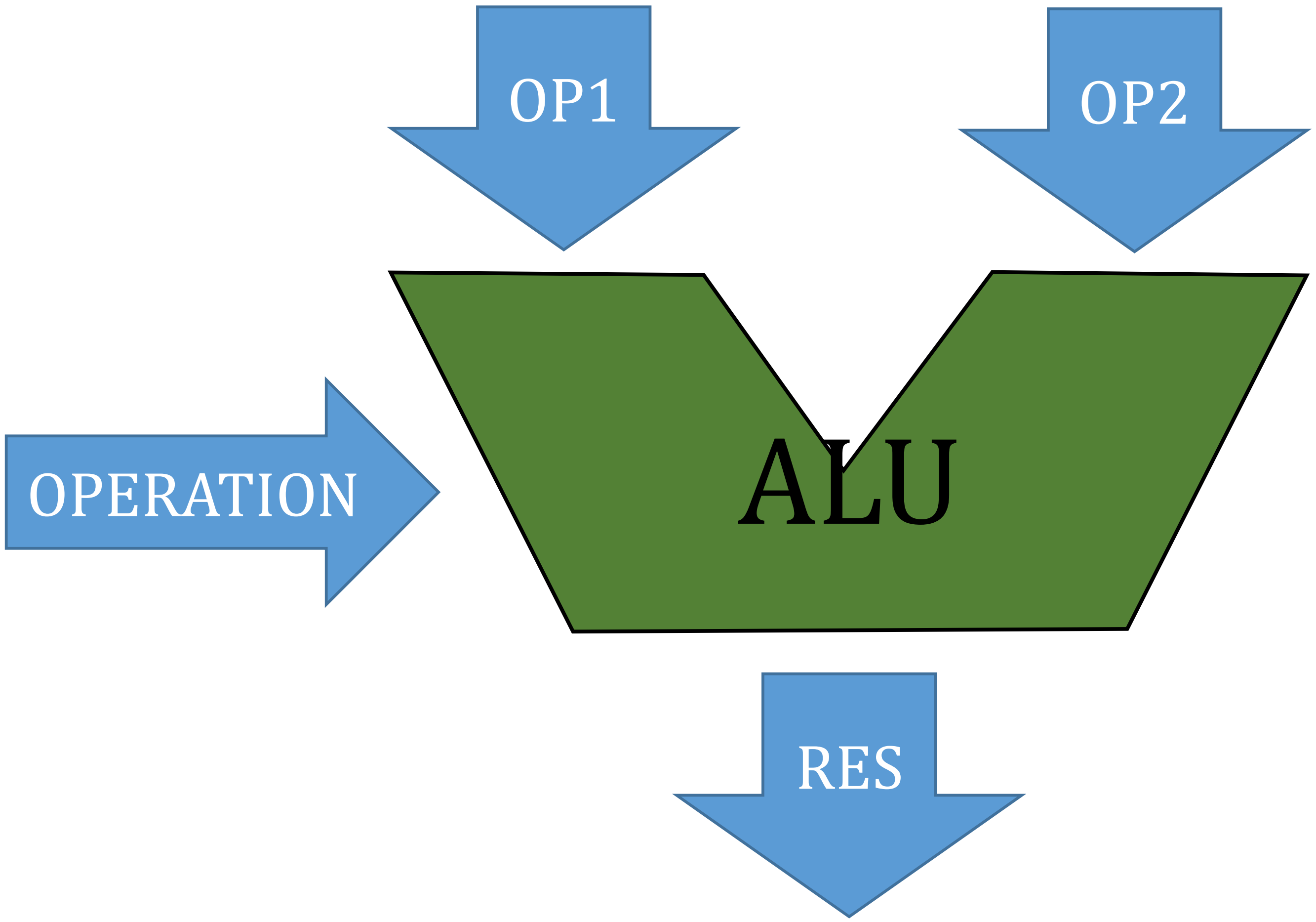
<table>
<tr><th>A</th><th>B</th><th>A|B</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>

Gate Level Implementation

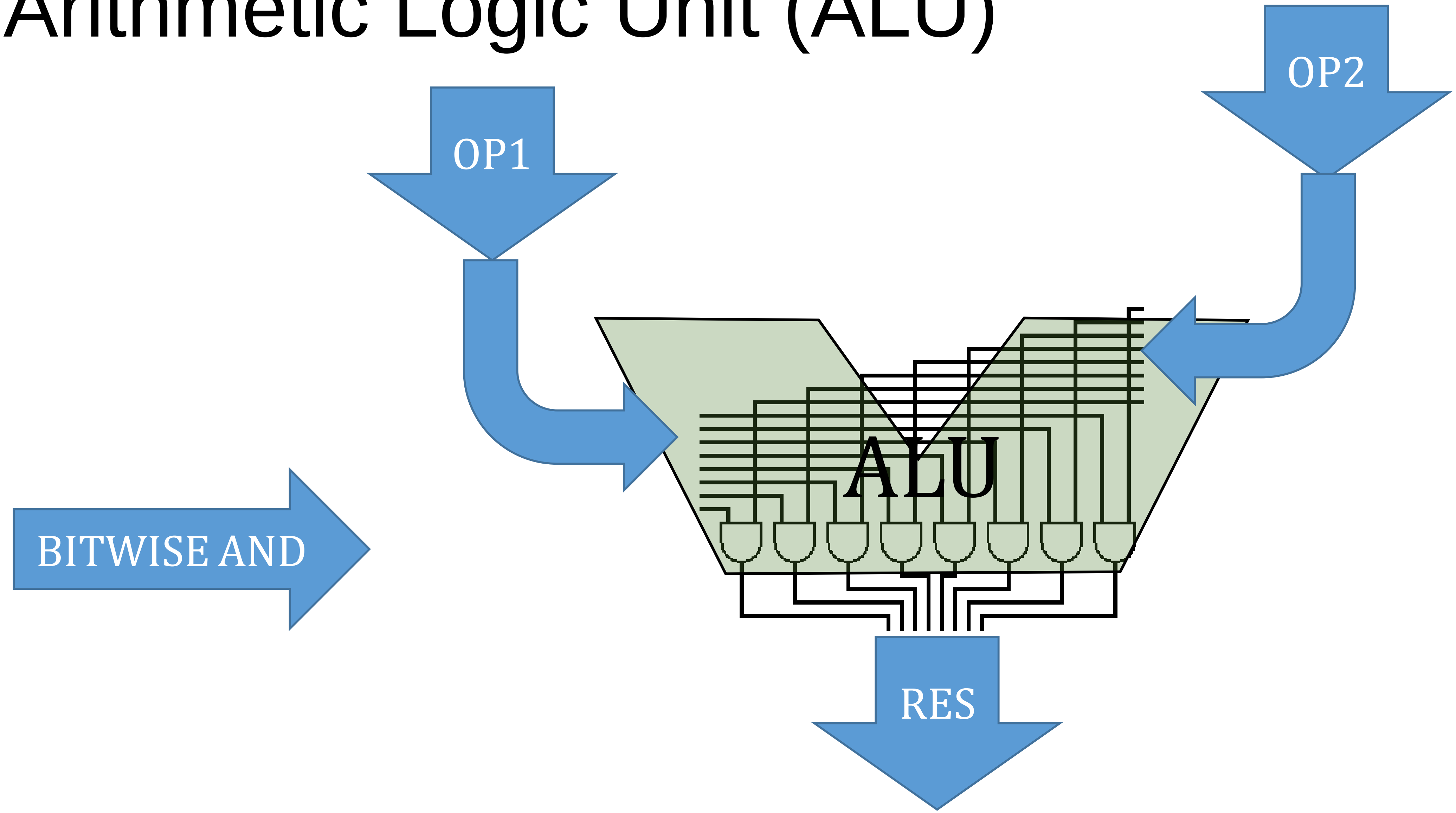
8 bit bitwise AND



Arithmetic Logic Unit (ALU)



Arithmetic Logic Unit (ALU)



Bitwise Operations in C

- AND (&)
- OR (|)
- Exclusive OR (^)
- Not (~)

Bit Twiddling Example

See xmp_bitTwiddling

- if (x & 0xf0) …

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
1	1	1	1	0	0	0	0
x_7	x_6	x_5	x_4	0	0	0	0

- if ((x | 0x0f)==0xff)
  0xf0 = 240

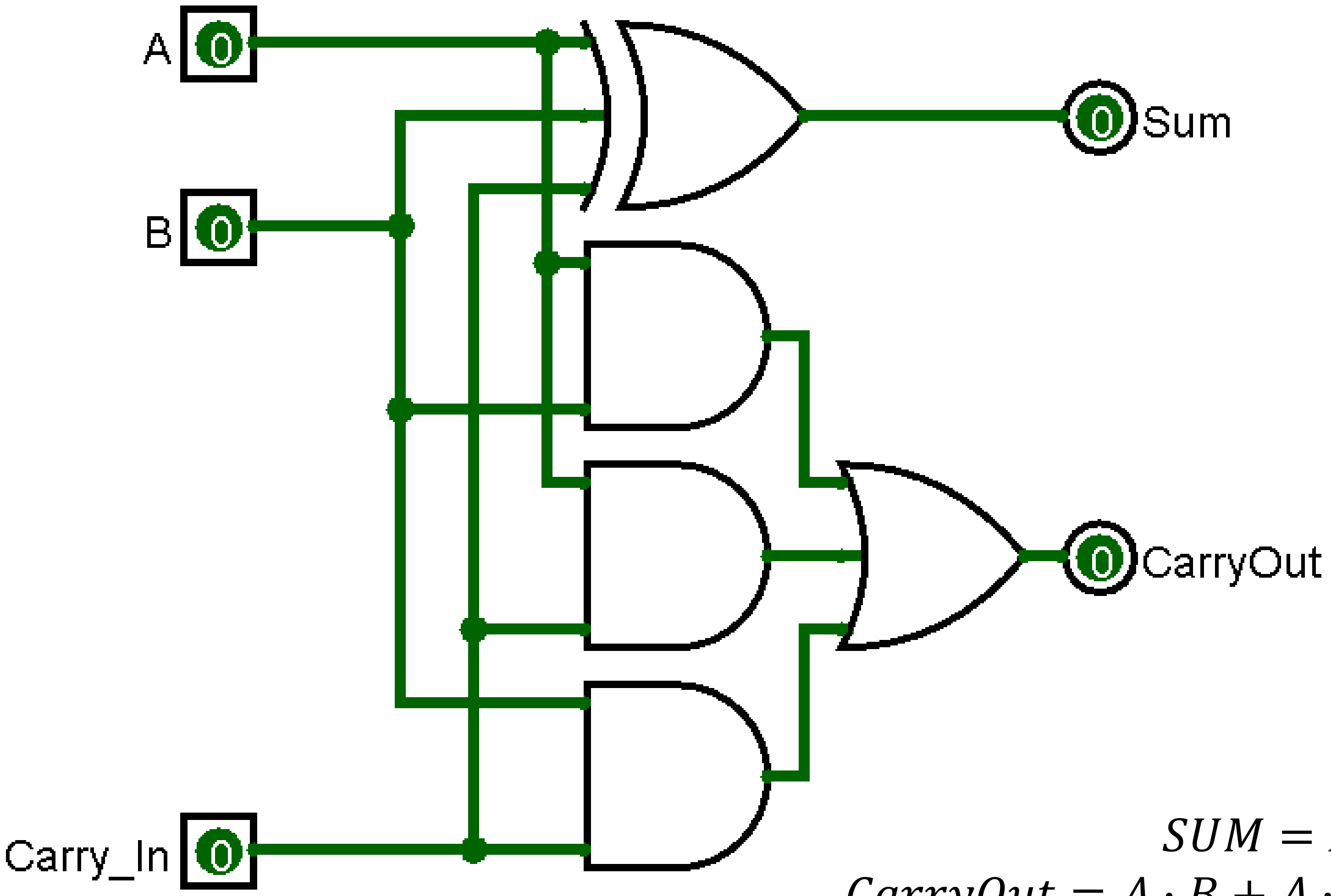
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
0	0	0	0	1	1	1	1
x_7	x_6	x_5	x_4	1	1	1	1
1	1	1	1	x	x	x	x

Binary Addition

- For example:

	1	1	1			1			
	0	1	1	1	0	0	1	1	115
+	0	0	1	1	0	0	1	0	+ 50
	1	0	1	0	0	1	0	1	162

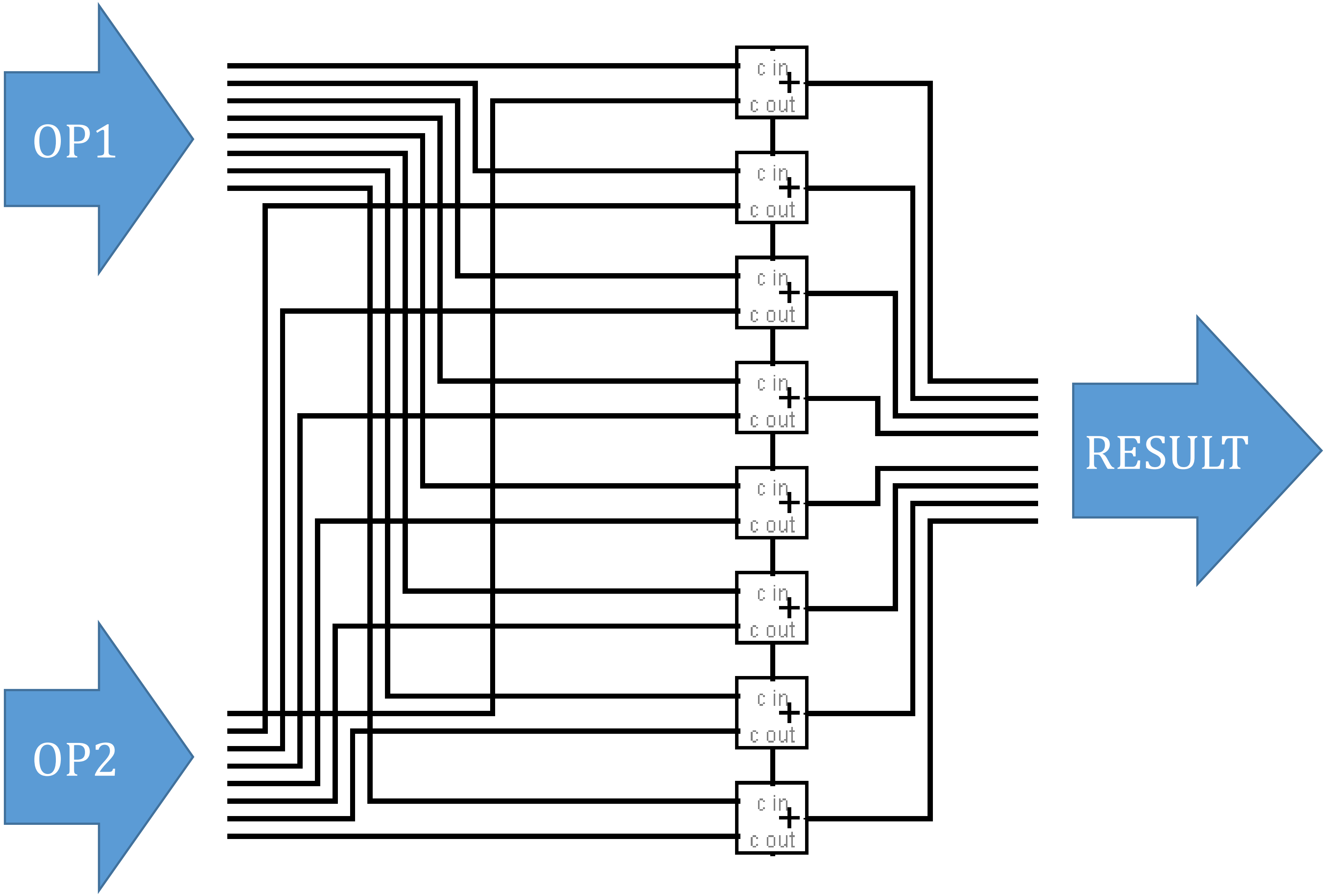
Full Adder



$$SUM = A \oplus B \oplus C$$

$$CarryOut = A \cdot B + A \cdot CarryIn + B \cdot CarryIn$$

Eight Bit Adder



Unsigned vs. Two's Complement Addition

Addition is Addition

1	1	1	1			1			UNS	SGN
	0	1	1	1	0	0	1	1	115	115
+	1	1	1	1	0	0	1	0	+242	+ -14
	0	1	1	0	0	1	0	1	101 OVFL	101

Overflow is Different!

Overflow with Addition

Unsigned

- Carry out of the high order bit

Two's Complement

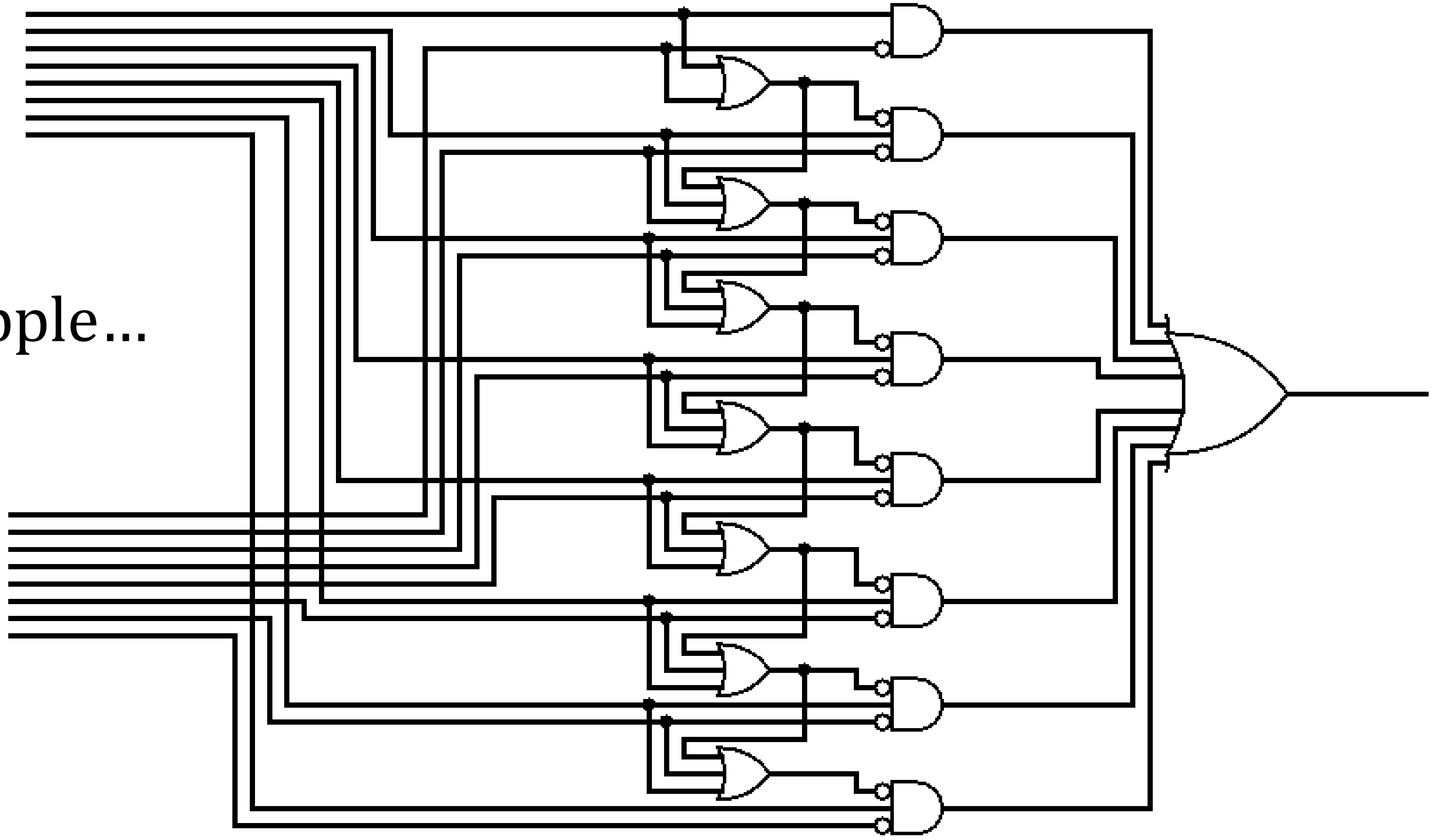
- Sign Bit Incorrect…
  - POS + POS = NEG or
  - NEG + NEG = POS
- Note… Opposite signs never overflow!
  - POS + NEG = No Overflow

Binary Subtraction A-B

- Two's Complement: compute A+(-B)
  - Find –B by flipping bits + 1
  - A + 1 + (~B)
  - Overflow: NEG-POS = POS or POS – NEG = NEG
- Unsigned Subtraction
  - A-B… convert A and B to two's complement, do two's complement subtraction, convert result to Unsigned
  - A+1+(~B)
  - Overflow: A < B

Comparison A vs B

- A>B if A-B>0
- A==B if A-B=0
- A<B if A-B<0
- Much easier than ripple…



What is "True"?

- When dealing with multiple bits, some are "on" and some are "off"
  - e.g. char i=39; /* 0b0010 0111 */
  - Is this "true" or "false"?
- Bitwise operations do multiple (column-wise) evaluations
  - Is the result of the entire operation "true" or "false"?
  - Some columns may evaluate to "true"… some to "false"
- C Logical "Truth"
  - By convention, a group of bits is "True" if ***ANY*** bit is true (1)!
  - Therefore, a group of bits is "False" only if ***ALL*** bits are false (0)!

Logical "Truth Value"

- Zero is "false", non-zero is "true"

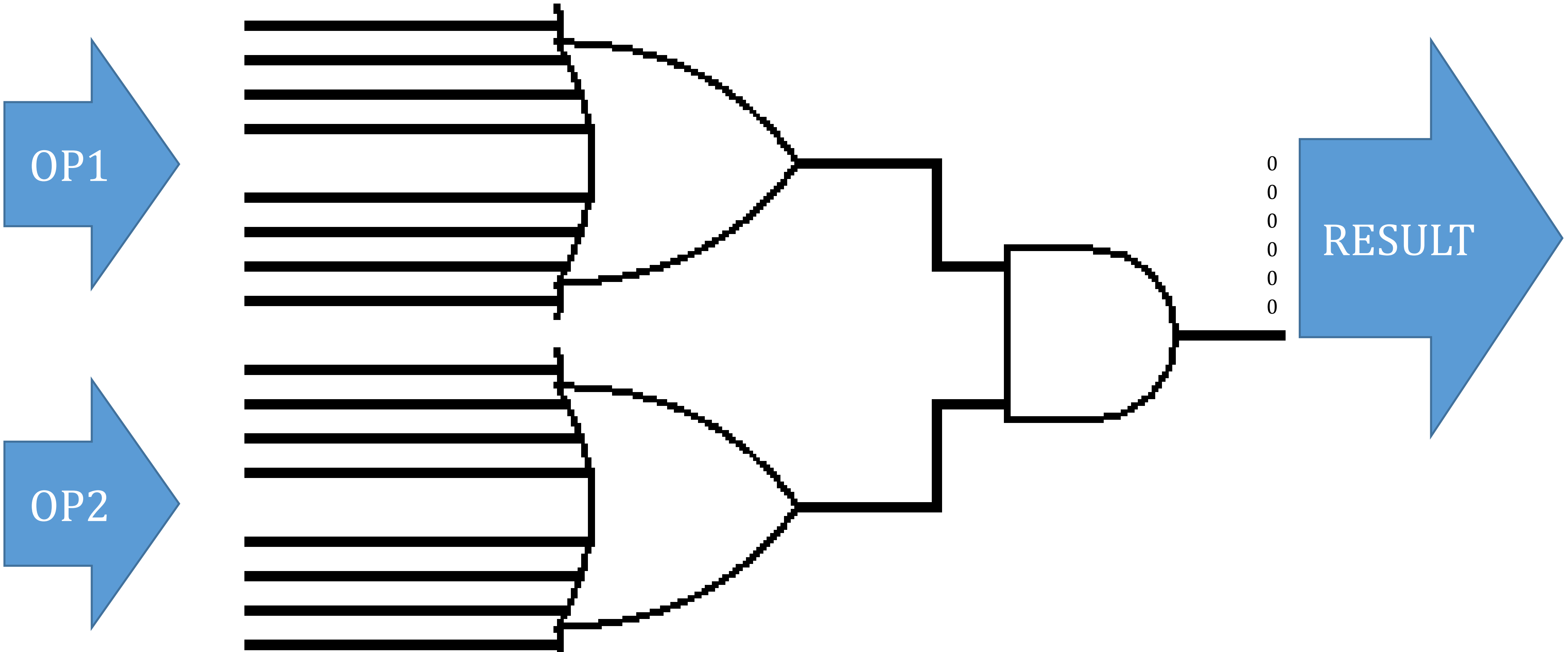
```
int x = 10;
while(x) { ...; x = x - 1; }
x=10; while(x) { ...; x = x - 3; }

if (x & 0x40) { /* If second bit from left is on in X */ ... }

if (x && y) { /* If both x and y are non-zero */ ... }

if (x & 0xf0) { /* ? */ ... }
if ( (x | 0x0f) == 0xff) { /* ? */ ... }
```

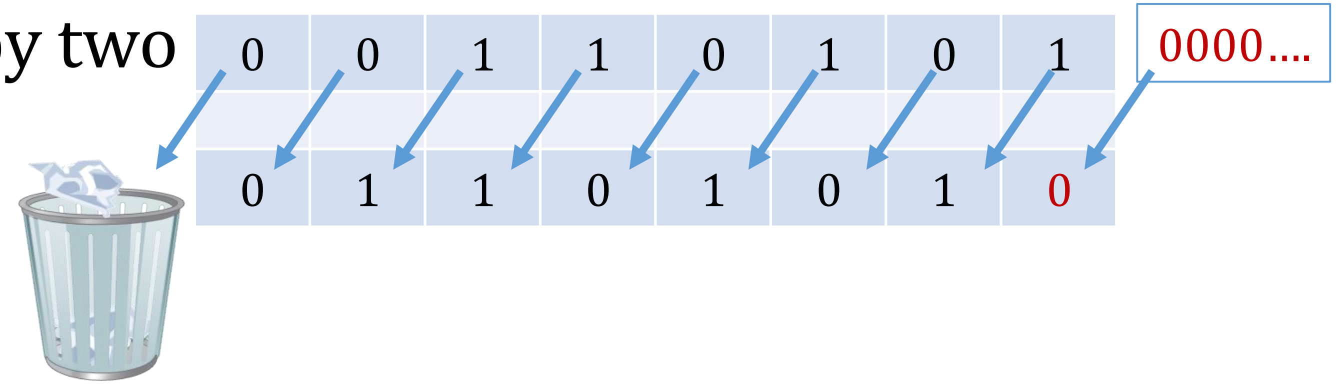
Logical AND (&&)



Bit Shifting

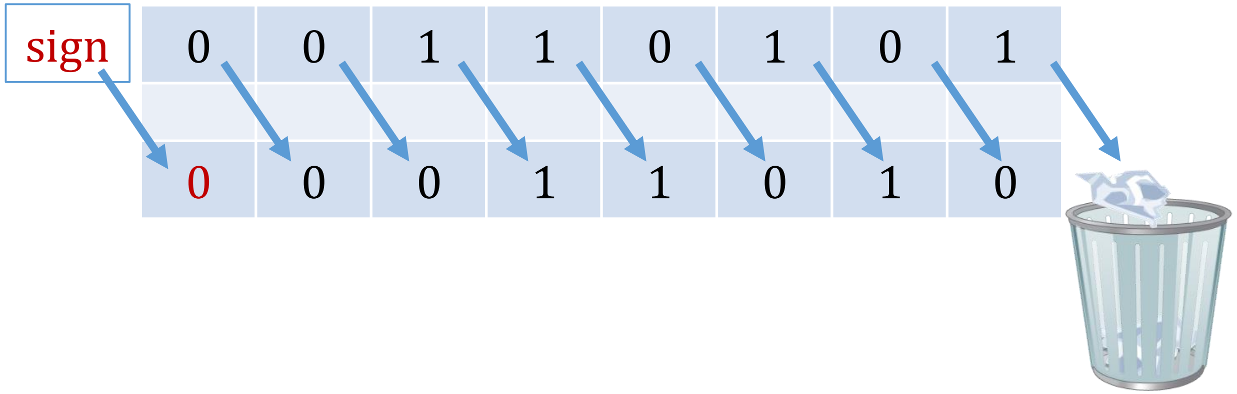
- Shift Left – Same as multiply by two

```
signed char x=53;
signed char y=x<<1;
```



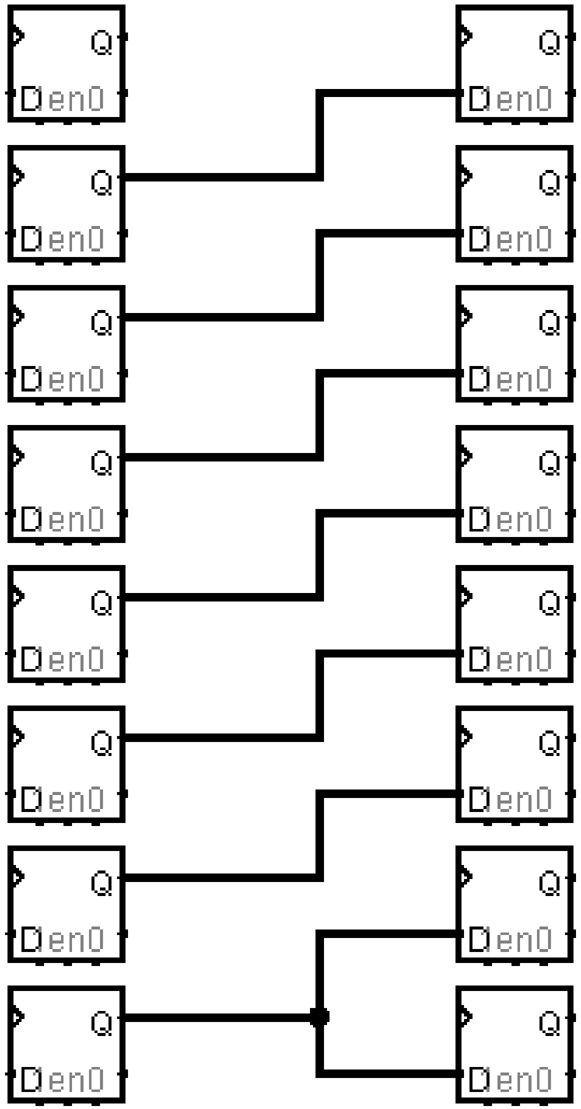
- Shift Right – Same as divide by two (almost)

```
signed char x=53;
signed char y=x>>1;
```



See xmp_shift/shift.c

Shift Left 1 (Arithmetic)



Bit Shifting… Signed vs. Unsigned

- Shift left… no difference – pad on right with 0
- Shift right…
  - Signed… pad on left with sign bit
  - Unsigned… pad on left with "sign" bit… always 0
- In lower level languages…
  - "shift right logical" same as unsigned shift – pad on left with 0
  - "shift right arithmetic" same as signed shift – pad on left with sign bit

Binary Multiplication / Division

			0	0	0	1	0	0	1	0	18
		x	0	0	0	0	0	1	1	1	x7
			0	0	0	1	0	0	1	0	56
+		0	0	0	1	0	0	1	0		+70
+	0	0	0	1	0	0	1	0			
			0	1	1	1	1	1	1	0	126

```
ACCUM=0;
FOR (BIT=0; BIT<32; BIT++) {
        IF (MULTIPLICAND & (1<<BIT)) ACCUM = ACCUM + MULTIPLIER
        MULTIPLIER=MULTIPLIER<<1
}
```

Bit Twiddling

- The fine are of performing neat tricks using bit manipulation, often in ways that are TOTALLY uncomprehendable
- See: https://en.wikipedia.org/wiki/Bit_manipulation
- For example…

```
If (x & (x-1)) { /* x is a power of 2 */
        …
}
```

Abstraction

Bits are stored in memory from most significant at left to least significant at right

(int 100,000 = 0x0001 86A0)

2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}	2^{25}	2^{24}	2^{23}	2^{22}	2^{21}	2^{20}	2^{19}	2^{18}	2^{17}	2^{16}	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	1	0	1	0	1	0	0	0	0	0
0				0				0				1				8				6				A				0			
Byte m								Byte m+1								Byte m+2								Byte m+3							

Leaky Abstraction – “Endian”-ness

Some machines store as expected… (Big–endian)

S	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}	2^{25}	2^{24}	2^{23}	2^{22}	2^{21}	2^{20}	2^{19}	2^{18}	2^{17}	2^{16}	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
b_{31}	b_{30}	b_{29}	b_{28}	b_{27}	b_{26}	b_{25}	b_{24}	b_{23}	b_{22}	b_{21}	b_{20}	b_{19}	b_{18}	b_{17}	b_{16}	b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_{9}	b_{8}	b_{7}	b_{6}	b_{5}	b_{4}	b_{3}	b_{2}	b_{1}	b_{0}
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	1	0	1	0	1	0	0	0	0	0
0				0				0				1				8				6				A				0			
Byte 42								Byte 43								Byte 44								Byte 45							

Some machines store least significant *byte* first! (Little-endian)

2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{23}	2^{22}	2^{21}	2^{20}	2^{19}	2^{18}	2^{17}	2^{16}	S	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}	2^{25}	2^{24}
b_{31}	b_{30}	b_{29}	b_{28}	b_{27}	b_{26}	b_{25}	b_{24}	b_{23}	b_{22}	b_{21}	b_{20}	b_{19}	b_{18}	b_{17}	b_{16}	b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_{9}	b_{8}	b_{7}	b_{6}	b_{5}	b_{4}	b_{3}	b_{2}	b_{1}	b_{0}
1	0	1	0	0	0	0	0	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
A				0				8				6				0				1				0				0			
Byte 42								Byte 43								Byte 44								Byte 45							

Why Little Endian?

- Casting:
  - int x; /* 32 bits starting at byte 42 */
  - y = (short int) x; /* Put the least significant 16 bits from x into y */

00	01	86	A0
42	43	44	45
x			
		(short int) x	

A0	86	01	00
42	43	44	45
x			
(short int) x			

When does Endian-ness Leak?

- Big-endian machine: First byte is the most significant byte
  - Everything works as expected
  - Until: we get binary data from a little-endian machine
- Little-endian machine: First byte is the least significant byte
  - When printing the value of a number, bytes are switched
  - We don’t even know if a machine is big-endian or little-endian!
  - Until: we get binary data from a big-endian machine OR
  - Until we look at the bit representation of the data, not treated as a number

Managing Endian-Ness

- Network standard is big-endian
- stdlib functions
  - machine representation → network (big-endian) representation
    - htons (short) , htonl (long)
  - Network representation (big-endian) → machine representation
    - ntohs (short), ntohl (long)
  - No-ops when hardware is big-endian
- endian.h functions
  - htobe16, htobe32, htobe64, htole16, htole32, htole64
  - be16toh, be32toh, be64toh, le16toh, le32toh, le64toh

See xmp_endian/network.c