CS330 - Computer Organization and Assembly Language Programming

Lecture 15

-Review -

Professor : Mahmut Unan – UAB CS

Agenda

Review

You responsible for all the topics

This is a rough review

Midterm Exam

October 1st, 2021 Friday

Lecture Time

Exam structure

- Multiple choice questions: ~20
- Conversions(binary,decimal,hex,mb,tb..etc), calculations (Amdahl's law)
- Open ended questions

Hello World !

A typical *C program* basically consists of the following parts;

- Preprocessor Commands
- Functions
- Variables
- Statements & Expressions
- Comment

```
#include <stdio.h>
int main()
{
 /* the first program in CS330 */
 printf("hello, world\n");
 return 0;
}
```

Information is Bits + Context

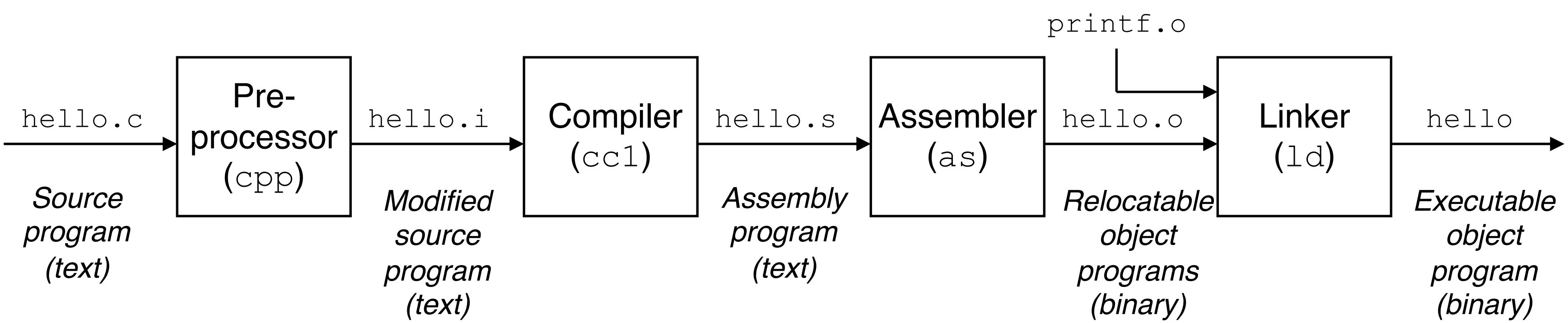
- "hello.c" is a source code
  - Sequence of bits (0 or 1)
  - 8-bit data chunks are called bytes
  - Each byte represents some text character in the program

#	i	n	c	l	u	d	e	<sp>	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	<sp>	m	a	i	n	(	)	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	<sp>	<sp>	<sp>	<sp>	p	r	i	n	t	f	(	"	h	e	l
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
l	o	,	<sp>	w	o	r	l	d	\	n	"	)	;	\n	}
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	125

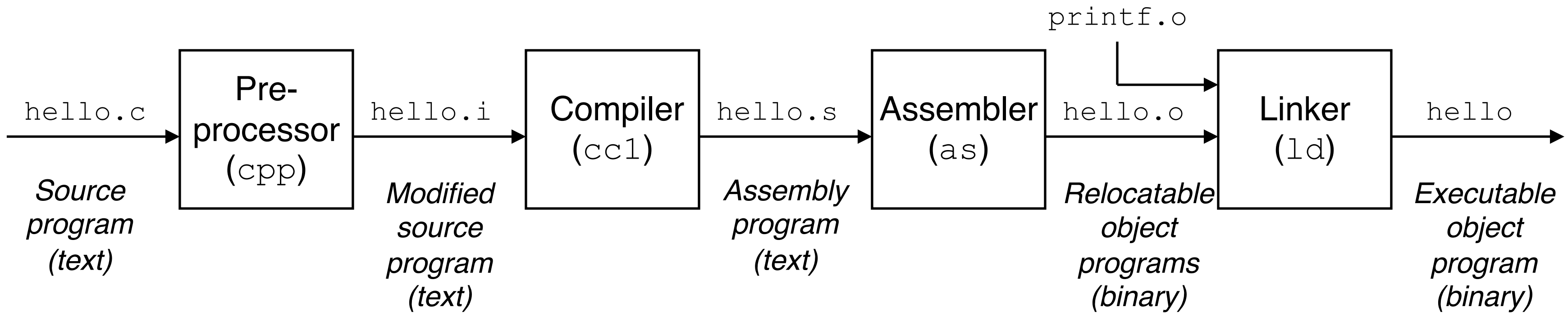
Figure 1.2 **The ASCII text representation of `hello.c`.**

Programs translated by other programs

- `unix> gcc -o hello hello.c`



Compilation System

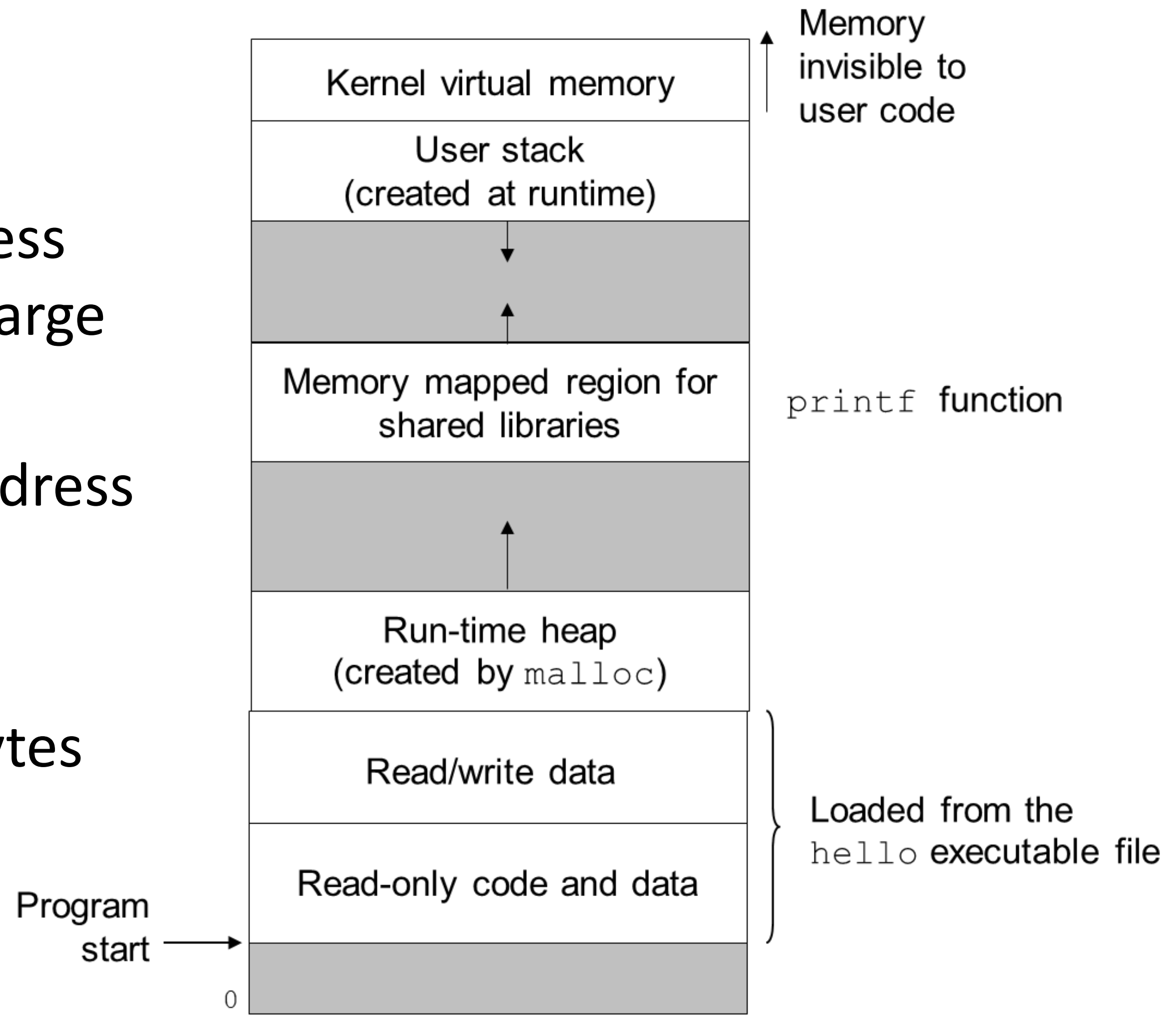


- Pre-processing
  - E.g., `#include<stdio.h>` is inserted into hello.i
- Compilation (.s)
  - Each statement is an assembly language program
- Assembly (.o)
  - A binary file whose bytes encode mach. language instructions
- Linking
  - Get printf() which resides in a separate precompiled object file

Virtual memory

- Illusion that each process has exclusive use of a large main memory
  - Example: Virtual address space for Linux

- **Files:** A sequence of bytes



Amdahl's Law

- Effectiveness of improving the performance of one part of system
- Speed up one part → Effect on the overall system performance?
- $T_{new} = (1 - \alpha)T_{old} + \frac{\alpha T_{old}}{k}$
- $= T_{old}[(1 - \alpha) + \frac{\alpha}{k}]$
- $S = \frac{T_{old}}{T_{new}}$
- $S = \frac{1}{((1-\alpha)+(\alpha/k))}$

Amdahl's Law / Example

- Consider a system;
  - A part of the system initially consumed 60% of the time (α = 0.6)
  - It is sped up by a factor of **3** (k=3)

- Overall improvement ?

Amdahl's Law / Example

- Consider a system;
  - A part of the system initially consumed 60% of the time (α = 0.6)
  - It is sped up by a factor of **3** (k=3)

- Overall improvement ?
- $S = \frac{1}{((1-\alpha)+(\alpha/k))}$

- = 1 / [0.4 + 0.6/3] = **1.67 times**

Amdahl's Law / Example 2

- Calculate the following improvements on a current system and decide which one is better

- **1)** if we make 90% of a program run 10 times faster.

- **2)** if we make 80% of a program run 20% faster

Amdahl's Law / Example 2

- **1)** if we make 90% of a program run 10 times faster.

$$S = \frac{1}{((1-\alpha)+(\alpha/k))} = \frac{1}{((1-0.9)+(0.9/10))} = \mathbf{5.26}$$

- **2)** if we make 80% of a program run 20% faster

$$S = \frac{1}{((1-\alpha)+(\alpha/k))} = \frac{1}{((1-0.8)+(0.8/1.2))} = \mathbf{1.153}$$

Data Measurement Chart

Data Measurement	Size
Bit	Single Binary Digit (1 or 0)
Byte	8 bits
Kilobyte (KB)	1,024 Bytes
Megabyte (MB)	1,024 Kilobytes
Gigabyte (GB)	1,024 Megabytes
Terabyte (TB)	1,024 Gigabytes
Petabyte (PB)	1,024 Terabytes
Exabyte (EB)	1,024 Petabytes

The Decimal System

- System based on decimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) to represent numbers
- For example the number 83 means eight tens plus three:

$$83 = (8 * 10) + 3$$

- The number 4728 means four thousands, seven hundreds, two tens, plus eight:

$$4728 = (4 * 1000) + (7 * 100) + (2 * 10) + 8$$

- The decimal system is said to have a ***base*****,** or ***radix*****,** of 10. This means that each digit in the number is multiplied by 10 raised to a power corresponding to that digit's position:

$$83 = (8 * 10^1) + (3 * 10^0)$$

$$4728 = (4 * 10^3) + (7 * 10^2) + (2 * 10^1) + (8 * 10^0)$$

The Binary System

- Only two digits, 1 and 0
- Represented to the base 2
- The digits 1 and 0 in binary notation have the same meaning as in decimal notation:

$$0_2 = 0_{10}$$
$$1_2 = 1_{10}$$

- To represent larger numbers each digit in a binary number has a value depending on its position:

$$10_2 = (1 * 2^1) + (0 * 2^0) = 2_{10}$$
$$11_2 = (1 * 2^1) + (1 * 2^0) = 3_{10}$$
$$100_2 = (1 * 2^2) + (0 * 2^1) + (0 * 2^0) = 4_{10}$$

For representing numbers in base 2, there are two possible digits (0, 1) in which column values are a power of two:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
128	64	32	16	8	4	2	1	
0	**1**	**1**	**0**	**0**	**0**	**1**	**1**	
0 +	64 +	32 +	0 +	0 +	0 +	2 +	1	= **99**

Although values represented in base 2 are significantly longer than those in base 10, **binary representation is used in digital computing because of the resulting simplicity of hardware design**

Encoding Byte Values

- Byte = 8 bits
  - Binary 00000000_2 to 11111111_2
  - Decimal: 0_{10} to 255_{10}
  - Hexadecimal 00_{16} to FF_{16}
    - Base 16 number representation
    - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - Write $FA1D37B_{16}$ in C as
      - 0xFA1D37B
      - 0xfa1d37b

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

1100 1001 0111 1011 → 0xC97B

Machine Words

- Machine has “word size”
  - Nominal size of integer-valued data
  - More importantly – a virtual address is encoded by such a word
    - Hence, it determines max size of virtual address space
  - Most current machines are 32 bits (4 bytes)
    - Limits addresses to 4GB
    - Becoming too small for memory-intensive applications
  - Newer systems are 64 bits (8 bytes)
    - Potentially address ≈ 1.84 X 10^19 bytes
  - Machines support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes

Data Sizes

- Each computer has a word size
  - For a machine with w-bit word size
  - The virtual address can range from 0 to 2^{w}-1
  - The program access to at most 2^{w} bytes

- 32 bit vs 64 bit

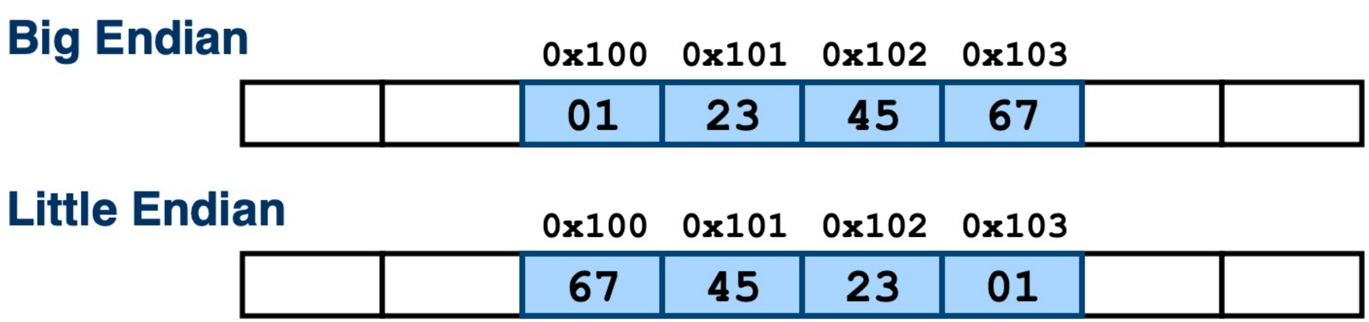
Addressing and Byte Ordering

- For objects that span multiple bytes (e.g. integers), we need to agree on two things
  - what would be the address of the object?
  - how would we order the bytes in memory?

http://www.aqualab.cs.northwestern.edu/component/attachments/download/167

Byte Ordering

- How to order bytes within multi-byte word in memory
- Conventions
  - (most) Sun's, IBMs are "Big Endian" machines
    - Least significant byte has highest address (comes last)
  - (most) Intel's are "Little Endian" machines
    - Least significant byte has lowest address (comes first)
- Example
  - Variable x has 4-byte representation 0x01234567
  - Address given by &x is 0x100 0x100 0x101



Boolean Variables and Operations

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0
  - 〈{0,1}, |, &, ~, 0, 1〉
  - | is “sum” operation, & is “product” operation
  - ~ is “complement” operation (not additive inverse)
  - 0 is identity for sum, 1 is identity for product
- Makes use of variables and operations
  - Are logical
  - A variable may take on the value 1 (TRUE) or 0 (FALSE)
  - Basic logical operations are AND, OR, XOR and NOT

Boolean Variables and Operations / 2

- AND
  - Yields true (binary value 1) if and only if both of its operands are true
  - In the absence of parentheses the AND operation takes precedence over the OR operation
  - When no ambiguity will occur the AND operation is represented by simple concatenation instead of the dot operator
- OR
  - Yields true if either or both of its operands are true
- NOT
  - Inverts the value of its operand

Table: Boolean Operators

(a) Boolean Operators of Two Input Variables

P	Q	NOT P ($\overline{P}$)	P AND Q ($P \bullet Q$)	P OR Q ($P + Q$)	P NAND Q ($\overline{P \bullet Q}$)	P NOR Q ($\overline{P + Q}$)	P XOR Q ($P \oplus Q$)
0	0	1	0	0	1	1	0
0	1	1	0	1	1	0	1
1	0	0	0	1	1	0	1
1	1	0	1	1	0	0	0

(b) Boolean Operators Extended to More than Two Inputs (A, B, . . .)

Operation	Expression	Output = 1 if
AND	$A \bullet B \bullet \ldots$	All of the set {A, B, …} are 1.
OR	$A + B + \ldots$	Any of the set {A, B, …} are 1.
NAND	$\overline{A \bullet B \bullet \ldots}$	Any of the set {A, B, …} are 0.
NOR	$\overline{A + B + \ldots}$	All of the set {A, B, …} are 0.
XOR	$A \oplus B \oplus \ldots$	The set {A, B, …} contains an odd number of ones.

Table: Basic Identities of Boolean Algebra

Basic Postulates		
$A \bullet B = B \bullet A$	$A + B = B + A$	Commutative Laws
$A \bullet (B + C) = (A \bullet B) + (A \bullet C)$	$A + (B \bullet C) = (A + B) \bullet (A + C)$	Distributive Laws
$1 \bullet A = A$	$0 + A = A$	Identity Elements
$A \bullet \overline{A} = 0$	$A + \overline{A} = 1$	Inverse Elements
Other Identities		
$0 \bullet A = 0$	$1 + A = 1$	
$A \bullet A = A$	$A + A = A$	
$A \bullet (B \bullet C) = (A \bullet B) \bullet C$	$A + (B + C) = (A + B) + C$	Associative Laws
$\overline{A \bullet B} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A} \bullet \overline{B}$	DeMorgan's Theorem

Exercise 1

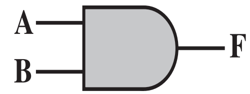
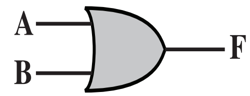
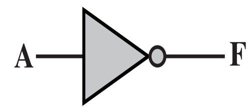
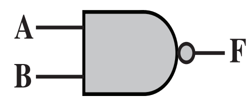
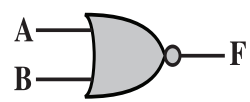
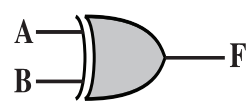
Evaluate the following expression when A =0, B =1, and C= 1

F= B + $\overline{\mathrm{C}}\mathrm{A} + \mathrm{B}\overline{\mathrm{A}}$ + $\mathrm{A}\overline{\mathrm{B}}$

Simplify the following functions;

F= AB + BC+$\overline{\mathrm{B}}\mathrm{C}$

F = A +$\overline{\mathrm{A}}\ \mathrm{B}$

Name	Graphical Symbol	Algebraic Function	Truth Table
AND		$F = A \bullet B$ or $F = AB$	A B | F 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR		$F = A + B$	A B | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
NOT		$F = \overline{A}$ or $F = A'$	A | F 0 | 1 1 | 0
NAND		$F = \overline{AB}$	A B | F 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR		$F = \overline{A + B}$	A B | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0
XOR		$F = A \oplus B$	A B | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0

General Boolean Algebras

- Boolean operations can be extended to work on bit vectors
  - Operations applied bitwise

```
  01101001     01101001     01101001
& 01010101   | 01010101   ^ 01010101   ~ 01010101
  --------     --------     --------     --------
  01000001     01111101     00111100     10101010
```

- All of the properties of Boolean algebra apply
- Now, Boolean |, & and ~ correspond to set union, intersection and complement

http://www.aqualab.cs.northwestern.edu/component/attachments/download/167

Bit-Level Operations in C

- Operations **&, |, ~, ^** Available in C
  - Apply to any "integral" data type
    - `long, int, short, char, unsigned`
  - View arguments as bit vectors
  - Arguments applied bit-wise
- Examples (Char data type)
  - `~0x41` → `0xBE`
    - $\sim 01000001_2 \rightarrow 10111110_2$
  - `~0x00` → `0xFF`
    - $\sim 00000000_2 \rightarrow 11111111_2$
  - `0x69 & 0x55` → `0x41`
    - $01101001_2 \ \& \ 01010101_2 \rightarrow 01000001_2$
  - `0x69 | 0x55` → `0x7D`
    - $01101001_2 \mid 01010101_2 \rightarrow 01111101_2$

Contrast: Logic Operations in C

- Contrast to Logical Operators
  - `&&, ||, !`
    - View 0 as "False"
    - Anything nonzero as "True"
    - Always return 0 or 1
    - Early termination
- Examples (char data type)
  - `!0x41 → 0x00`
  - `!0x00 → 0x01`
  - `!!0x41 → 0x01`
  - `0x69 && 0x55 → 0x01`
  - `0x69 || 0x55 → 0x01`
  - `p && *p` (avoids null pointer access)

Watch out for && vs. & (and || vs. |)…
one of the more common oopsies in C programming

Shift Operations

- Left Shift: `x << y`
  - Shift bit-vector **x** left **y** positions
    - Throw away extra bits on left
    - Fill with 0's on right
- Right Shift: `x >> y`
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left
    - Useful in two's compliment
- Undefined Behavior
  - Shift amount < 0 or ≥ word size

Argument **x**	01100010
<< 3	00010*000*
Log. >> 2	*00*011000
Arith. >> 2	*00*011000

Argument **x**	10100010
<< 3	00010*000*
Log. >> 2	*00*101000
Arith. >> 2	*11*101000

Boolean Algebra ≈ Integer Ring

Commutative	A | B = B | A A & B = B & A	A + B = B + A A * B = B * A
Associativity	(A | B) | C = A | (B | C) (A & B) & C = A & (B & C)	(A + B) + C = A + (B + C) (A * B) * C = A * (B * C)
Product distributes over sum	A & (B | C) = (A & B) | (A & C)	A * (B + C) = A * B + B * C
Sum and product identities	A | 0 = A A & 1 = A	A + 0 = A A * 1 = A
Zero is product annihilator	A & 0 = 0	A * 0 = 0
Cancellation of negation	~ (~ A) = A	– (– A) = A

http://www.aqualab.cs.northwestern.edu/component/attachments/download/167

Boolean Algebra ≠ Integer Ring

Boolean: Sum distributes over product	$A \mid (B \,\&\, C) = (A \mid B) \,\&\, (A \mid C)$	$A + (B * C) \neq (A + B) * (B + C)$
Boolean: Idempotency	$A \mid A = A$ $A \,\&\, A = A$	$A + A \neq A$ $A * A \neq A$
Boolean: Absorption	$A \mid (A \,\&\, B) = A$ $A \,\&\, (A \mid B) = A$	$A + (A * B) \neq A$ $A * (A + B) \neq A$
Boolean: Laws of Complements	$A \mid {\sim}A = 1$	$A + -A \neq 1$
Ring: Every element has additive inverse	$A \mid {\sim}A \neq 0$	$A + -A = 0$

http://www.aqualab.cs.northwestern.edu/component/attachments/download/167

Properties of & and ^

- Boolean ring
  - ⟨{0,1}, ^, &, I, 0, 1⟩
  - Identical to integers mod 2
  - I is identity operation: I (A) = A
    - A ^ A = 0
- Property: Boolean ring

– Commutative sum	A ^ B = B ^ A
– Commutative product	A & B = B & A
– Associative sum	(A ^ B) ^ C = A ^ (B ^ C)
– Associative product	(A & B) & C = A & (B & C)
– Prod. over sum	A & (B ^ C) = (A & B) ^ (B & C)
– 0 is sum identity	A ^ 0 = A
– 1 is prod. identity	A & 1 = A
– 0 is product annihilator	A & 0 = 0
– Additive inverse	A ^ A = 0

Unsigned Encodings

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

(Binary To Unsigned)

e.g. B2U ([1011]) = $1 * 2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 11$

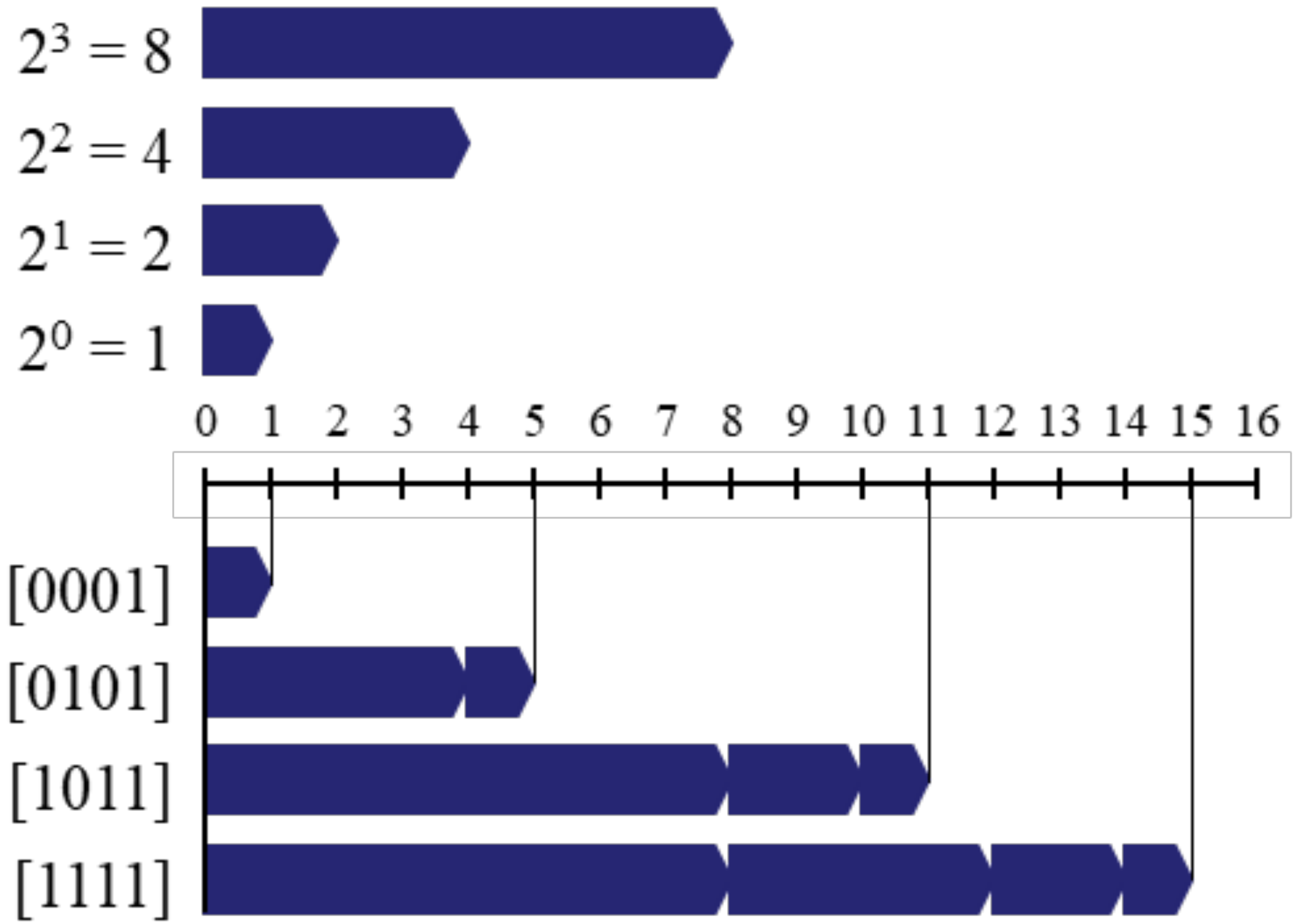
– C `short` 2 bytes long

```
short int x = 15213;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101

http://www.aqualab.cs.northwestern.edu/component/attachments/download/168

Examples


$2^3 = 8$
$2^2 = 4$
$2^1 = 2$
$2^0 = 1$
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
[0001]
[0101]
[1011]
[1111]

Two's Complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Sign Bit

– e.g. B2T ([1011]) = $-1 * 2^3 + 0*2^2 + 1*2^1 + 1*2^0 = -5$

– C `short` 2 bytes long

```
short int y = -15213;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

- Sign bit
  – For 2's complement, most significant bit indicates sign
    - 0 for nonnegative; 1 for negative

http://www.aqualab.cs.northwestern.edu/component/attachments/download/168

Two's Compliment

- Invert and add **one**

Suppose we're working with 8 bit quantities and suppose we want to find how **-28** would be expressed in two's complement notation.

- First we write out 28 in **binary form.**

  00011100
- Then we **invert the digits**. 0 becomes 1, 1 becomes 0.

  11100011
- Then we **add 1.**

  11100100

That is how one would write -28 in 8 bit binary.

https://www.cs.cornell.edu/~tomf/notes/cps104/twoscomp.html

Numeric Ranges

- **Unsigned Values**
  - *UMin* = 0
    000…0
  - *UMax* = $2^w - 1$
    111…1

- **Two's Complement Values**
  - *TMin* = -2^{w-1}
    100…0
  - *TMax* = $2^{w-1} - 1$
    011…1
- **Other Values**
  - Minus 1
    111…1

Values for *W* = 16

	Decimal	Hex	Binary
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

Signed vs. Unsigned in C

- Constants
  - By default are considered to be signed integers
  - Unsigned if have “U” as suffix

    0U, 4294967259U
- Casting
  - Explicit casting between signed & unsigned same as U2T and T2U

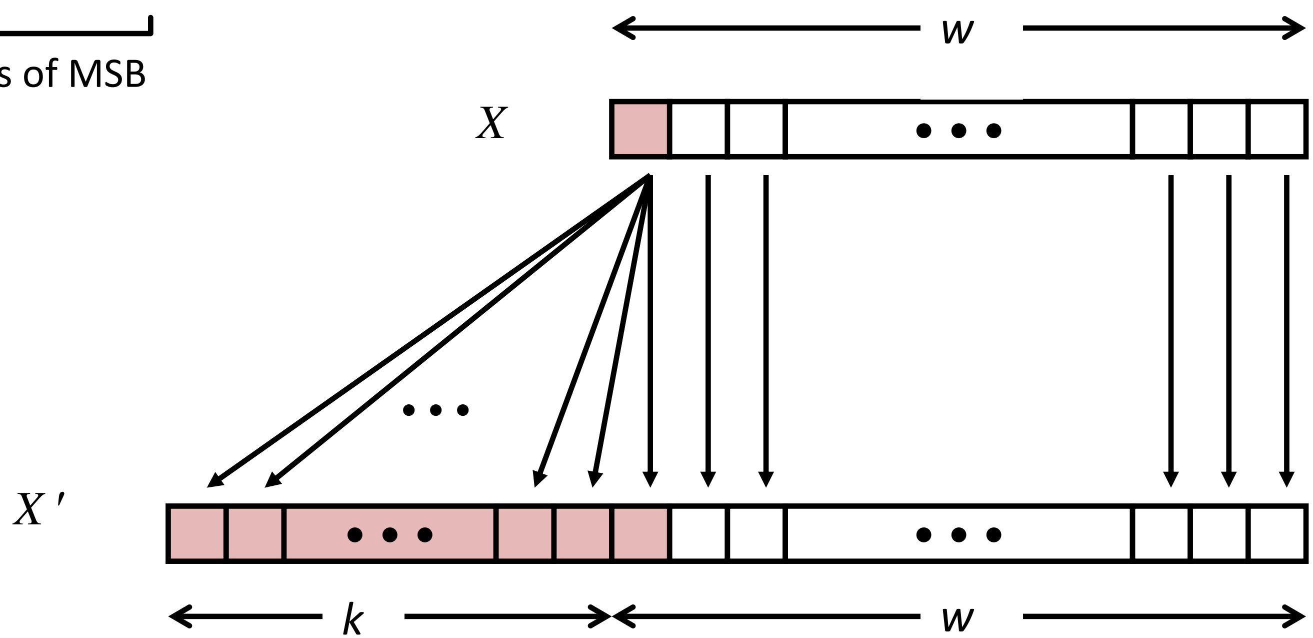
```
int tx, ty;
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```

  - Implicit casting also occurs via assignments and procedure calls

```
tx = ux;
uy = ty;
```

Sign Extension

- **Task:**
  - Given w-bit signed integer x
  - Convert it to w+k-bit integer with same value
- **Rule:**
  - Make k copies of sign bit:
  - $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$



Truncating Numbers

- Reduce the number of bits representing the number
- Truncating *w-bit* number to a *k* bit number, we drop the high order *w-k* bits
  - Can alter its value
    - A form of overflow

Summary: Expanding, Truncating: Basic Rules

- **Expanding (e.g., short int to int)**
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result

- **Truncating (e.g., unsigned to unsigned short)**
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
  - For small numbers yields expected behavior

OVERFLOW RULE:

- If two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the result has the opposite sign.

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 Binary Multiplication

```
    1011      Multiplicand (11)
   ×1101      Multiplier (13)
    1011
   0000       Partial products
  1011
 1011
10001111      Product (143)
```

Figure 10.7 Multiplication of Unsigned Binary Integers

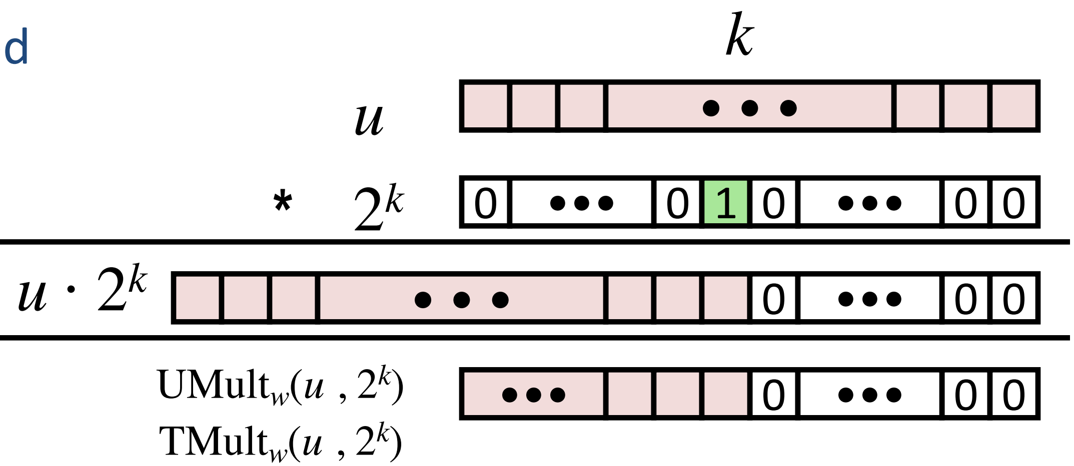
Power-of-2 Multiply with Shift

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

Operands: *w* bits

True Product: *w+k* bits

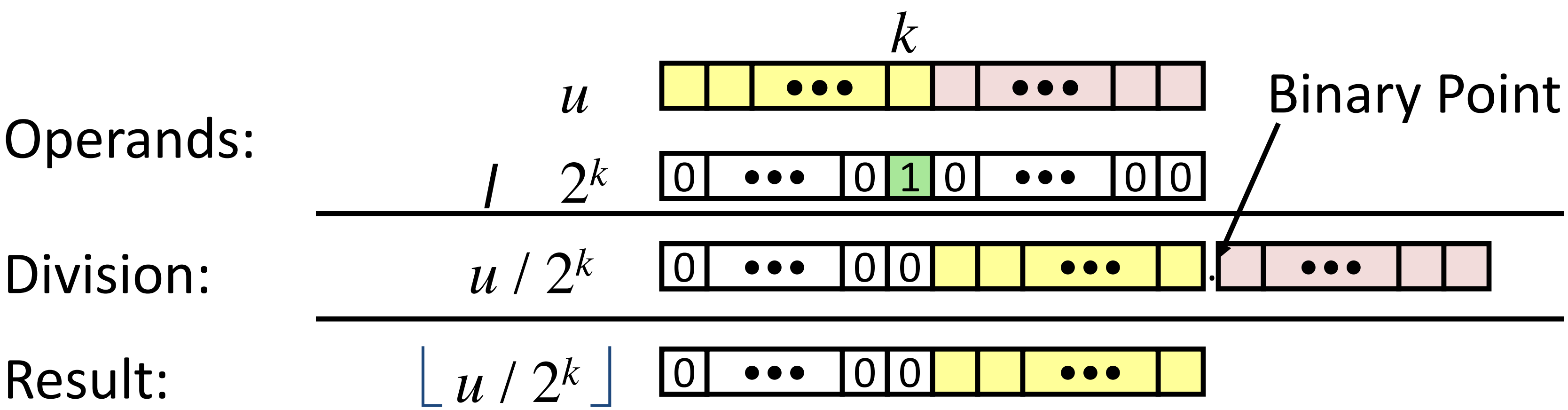
Discard *k* bits: *w* bits



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

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`

Arithmetic: Basic Rules

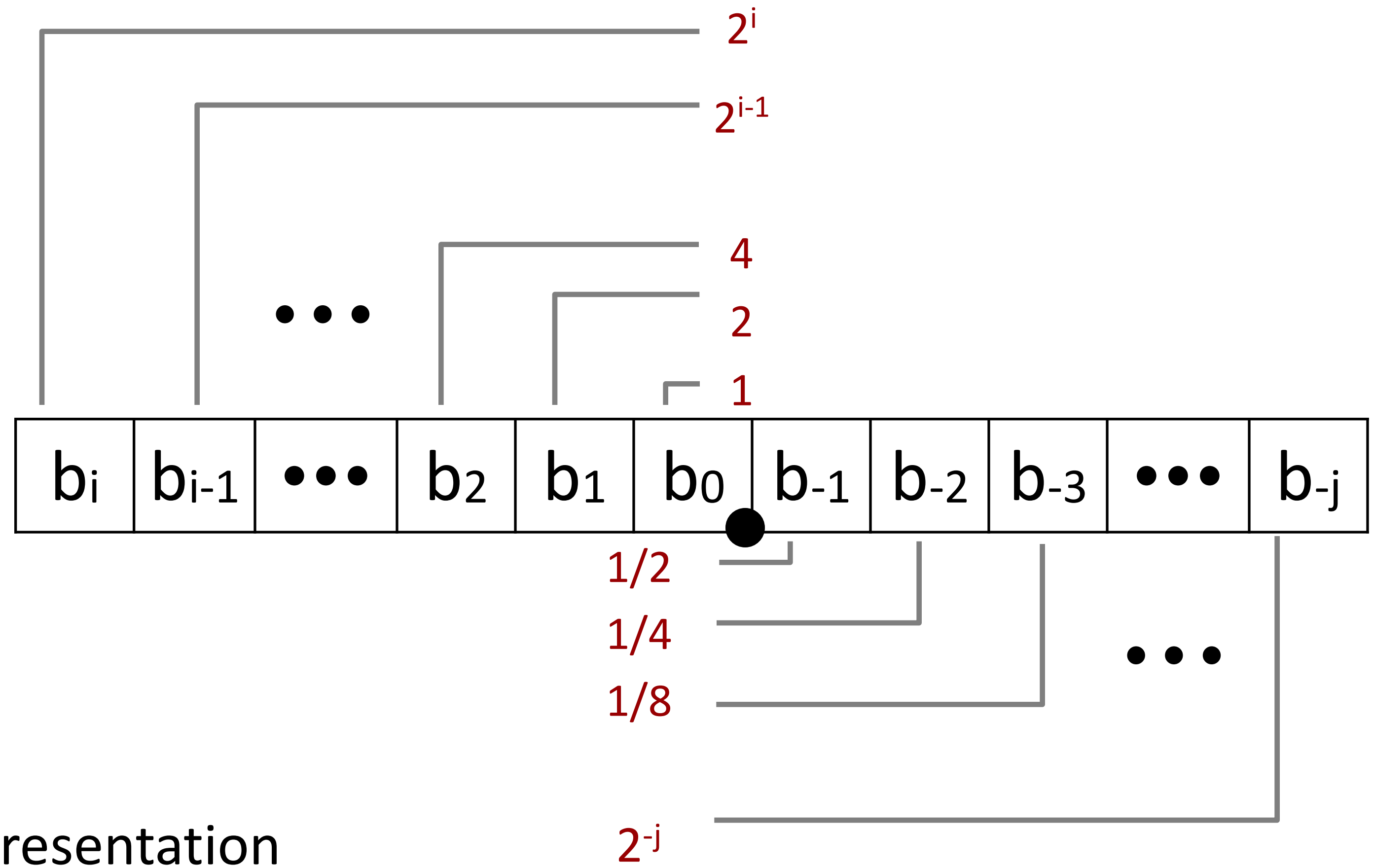
- **Addition:**
  - Unsigned/signed: Normal addition followed by truncate, same operation on bit level
  - Unsigned: addition mod 2^w
    - Mathematical addition + possible subtraction of 2^w
  - Signed: modified addition mod 2^w (result in proper range)
    - Mathematical addition + possible addition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

Fractional binary numbers

- What is 1011.101_2?

Fractional Binary Numbers



- Representation
  - Bits to right of "binary point" represent fractional powers of 2
  - Represents rational number:

$$\sum_{k=-j}^{i} b_k \times 2^k$$

Fractional Binary Numbers: Examples

- Value | Representation

Value	Representation
5 3/4	`101.11`$_2$
2 7/8	`10.111`$_2$
1 7/16	`1.0111`$_2$

- Observations
  - Divide by 2 by shifting right (unsigned)
  - Multiply by 2 by shifting left
  - Numbers of form $0.111111\ldots_2$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - Use notation $1.0 - \varepsilon$

Representable Numbers

- **Limitation #1**
  - Can only exactly represent numbers of the form $x/2^k$
    - Other rational numbers have repeating bit representations
  - **Value** **Representation**
    - 1/3 `0.0101010101[01]`…2
    - 1/5 `0.001100110011[0011]`…2
    - 1/10 `0.0001100110011[0011]`…2

- **Limitation #2**
  - Just one setting of binary point within the *w* bits
    - Limited range of numbers (very small values? very large?)

IEEE Floating Point

- **IEEE Standard 754**
  - Established in 1985 as uniform standard for floating point arithmetic
    - Before that, many idiosyncratic formats
  - Supported by all major CPUs

- **Driven by numerical concerns**
  - Nice standards for rounding, overflow, underflow
  - Hard to make fast in hardware
    - Numerical analysts predominated over hardware designers in defining standard

Floating Point Representation

- Numerical Form:

  $$(-1)^s \, M \;\; 2^E$$

  – Sign bit s determines whether number is negative or positive
  – Significand M normally a fractional value in range [1.0,2.0).
  – Exponent E weights value by power of two

- Encoding
  – MSB s is sign bit s
  – exp field encodes E (but is not equal to E)
  – frac field encodes M (but is not equal to M)

s	exp	frac

Precision options

- Single precision: 32 bits

s	exp	frac
1	8-bits	23-bits

- Double precision: 64 bits

s	exp	frac
1	11-bits	52-bits

- The value encoded by a given bit representation can be divided into three different cases, depending on the value of **exp**.
- Case 1: Normalized Values
- Case 2: Denormalized Values
- Case 3: Special Values

Case 1: "Normalized" Values

$\boxed{v = (-1)^s M\ 2^E}$

- *When: exp ≠ 000…0 and exp ≠ 111…1*

- Exponent coded as a biased value: $E = Exp - Bias$
  - **Exp:** unsigned value of exp field
  - **Bias =** $\mathbf{2^{k-1} - 1}$, where k is number of exponent bits
    - Single precision: 127 (Exp: 1…254, E: -126…127)
    - Double precision: 1023 (Exp: 1…2046, E: -1022…1023)

- Significand coded with implied leading **1:** $\mathbf{M = 1.xxx\ldots x_2}$
  - xxx…x: bits of frac field
  - Minimum when frac=000…0 (M = 1.0)
  - Maximum when frac=111…1 (M = 2.0 – ε)
  - Get extra leading bit for "free"

Case 2: Denormalized Values

$v = (-1)^s M\ 2^E$
$E = 1 - \text{Bias}$

- **Condition: exp = 000…0**

- Exponent value: $E = 1 - \text{Bias}$ (instead of $E = 0 - \text{Bias}$)
- Significand coded with implied leading 0: $M = 0.xxx\ldots x_2$
  - `xxx…x`: bits of `frac`
- Cases
  - `exp` = 000…0, `frac` = 000…0
    - Represents zero value
    - Note distinct values: +0 and –0 (why?)
  - `exp` = 000…0, `frac` ≠ 000…0
    - Numbers closest to 0.0
    - Equispaced

Case 3: Special Values

- Condition: **`exp`** = **`111`**…**`1`**

- Case: **`exp`** = **`111`**…**`1`**, **`frac`** = **`000`**…**`0`**
  - Represents value ∞ (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$

- Case: **`exp`** = **`111`**…**`1`**, **`frac`** ≠ **`000`**…**`0`**
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), $\infty - \infty$, $\infty \times 0$

Special Properties of the IEEE Encoding

- **FP Zero Same as Integer Zero**
  – All bits = 0

- **Can (Almost) Use Unsigned Integer Comparison**
  – Must first compare sign bits
  – Must consider −0 = 0
  – NaNs problematic
    - Will be greater than any other values
    - What should comparison yield?
  – Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity

Floating Point Operations: Basic Idea

- $\mathtt{x} +_{\mathtt{f}} \mathtt{y} = \mathtt{Round}(\mathtt{x} + \mathtt{y})$

- $\mathtt{x} \times_{\mathtt{f}} \mathtt{y} = \mathtt{Round}(\mathtt{x} \times \mathtt{y})$

- **Basic idea**
  - First compute exact result
  - Make it fit into desired precision
    - Possibly overflow if exponent too large
    - Possibly round to fit into **`frac`**

Rounding

- **Rounding Modes (illustrate with $ rounding)**

	$1.40	**$1.60**	**$1.50**	**$2.50**	**–$1.50**
– Towards zero	$1	$1	$1	$2	–$1
– Round down ($-\infty$)	$1	$1	$1	$2	–$2
– Round up ($+\infty$)	$2	$2	$2	$3	–$1
– Nearest Even (default)	$1	$2	$2	$2	–$2

FP Multiplication

- $(-1)^{s1}\ M1\ \ 2^{E1}\ \ \ x\ \ \ (-1)^{s2}\ M2\ \ 2^{E2}$
- **Exact Result:** $(-1)^{s}\ M\ \ 2^{E}$
  - Sign s: s1 ^ s2
  - Significand M: M1 x M2
  - Exponent E: E1 + E2

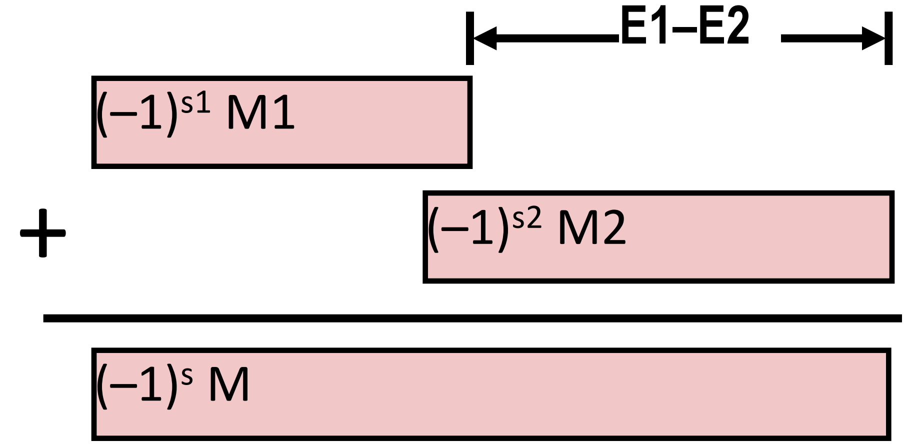
- **Fixing**
  - If M ≥ 2, shift M right, increment E
  - If E out of range, overflow
  - Round M to fit `frac` precision

- **Implementation**
  - Biggest chore is multiplying significands

Floating Point Addition

- $(-1)^{s1}\ M1\ \ 2^{E1}\ \ +\ \ (-1)^{s2}\ M2\ \ 2^{E2}$
  - –Assume E1 > E2

Get binary points lined up



- **Exact Result:** $(-1)^{s}\ M\ \ 2^{E}$
  - –Sign s, significand M:
    - Result of signed align & add
  - –Exponent E: E1

- **Fixing**
  - –If M ≥ 2, shift M right, increment E
  - –if M < 1, shift M left k positions, decrement E by k
  - –Overflow if E out of range
  - –Round M to fit `frac` precision

Mathematical Properties of FP Add

- Compare to those of Abelian Group
  - Closed under addition? Yes
    - But may generate infinity or NaN
  - Commutative? Yes
  - Associative? No
    - Overflow and inexactness of rounding
    - `(3.14+1e10)-1e10 = 0, 3.14+(1e10-1e10) = 3.14`
  - 0 is additive identity? Yes
  - Every element has additive inverse? Almost
    - Yes, except for infinities & NaNs
- Monotonicity
  - $a \geq b \Rightarrow a+c \geq b+c$? Almost
    - Except for infinities & NaNs

Mathematical Properties of FP Mult

- **Compare to Commutative Ring**
  - Closed under multiplication? Yes
    - But may generate infinity or NaN
  - Multiplication Commutative? Yes
  - Multiplication is Associative? No
    - Possibility of overflow, inexactness of rounding
    - Ex: `(1e20*1e20)*1e-20`= `inf`, `1e20*(1e20*1e-20)`= `1e20`
  - 1 is multiplicative identity? Yes
  - Multiplication distributes over addition? No
    - Possibility of overflow, inexactness of rounding
    - `1e20*(1e20-1e20)=0.0`, `1e20*1e20 - 1e20*1e20 = NaN`
- **Monotonicity**
  - $a \geq b$ & $c \geq 0 \Rightarrow a * c \geq b * c$? Almost
    - Except for infinities & NaNs

Floating Point in C

- **C Guarantees Two Levels**
  - **`float`** single precision
  - **`double`** double precision
- **Conversions/Casting**
  - Casting between **`int`**, **`float`**, and **`double`** changes bit representation
  - **`double`**/**`float`** → **`int`**
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range or NaN: Generally sets to TMin
  - **`int`** → **`double`**
    - Exact conversion, as long as **`int`** has ≤ 53 bit word size
  - **`int`** → **`float`**
    - Will round according to rounding mode

Floating Point Puzzles

- For each of the following C expressions, either:
  - Argue that it is true for all argument values
  - Explain why not true

```
int x = …;
float f = …;
double d = …;
```

Assume neither **`d`** nor **`f`** is NaN

- `x == (int)(float) x`
- `x == (int)(double) x`
- `f == (float)(double) f`
- `d == (double)(float) d`
- `f == -(-f);`
- `1.0/2 == 1/2.0`
- `d * d >= 0.0`
- `(d+f)-d == f`

Intel x86 Processors

- Dominate laptop/desktop/server market
- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on
- Complex instruction set computer (CISC)
  - Many different instructions with many different formats
    - But, only small subset encountered with Linux programs
  - Hard to match performance of Reduced Instruction Set Computers (RISC)
  - But, Intel has done just that!
    - In terms of speed. Less so for low power.

Our Coverage

- IA32
  - The traditional x86
  - For ???: RIP, Fall 2018

- x86-64
  - The standard
  - `moat.cis.uab.edu`
  - `gcc hello.c`
  - `gcc –m64 hello.c`

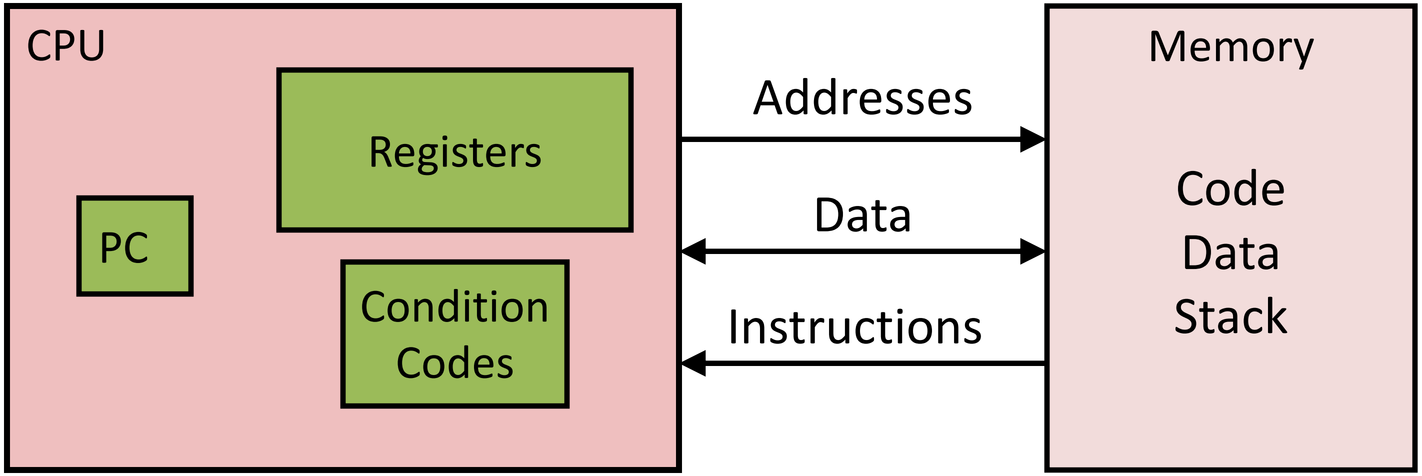
- Presentation
  - Book covers x86-64
  - So, we will cover x86-64

- **C, assembly, machine code**

Definitions

- Architecture: (also ISA: instruction set architecture) The parts of a processor design that one needs to understand or write assembly/machine code.
  - Examples: instruction set specification, registers.
- Microarchitecture: Implementation of the architecture.
  - Examples: cache sizes and core frequency.
- Code Forms:
  - Machine Code: The byte-level programs that a processor executes
  - Assembly Code: A text representation of machine code
- Example ISAs:
  - Intel: x86, IA32, Itanium, x86-64
  - ARM: Used in almost all mobile phones

Assembly/Machine Code View

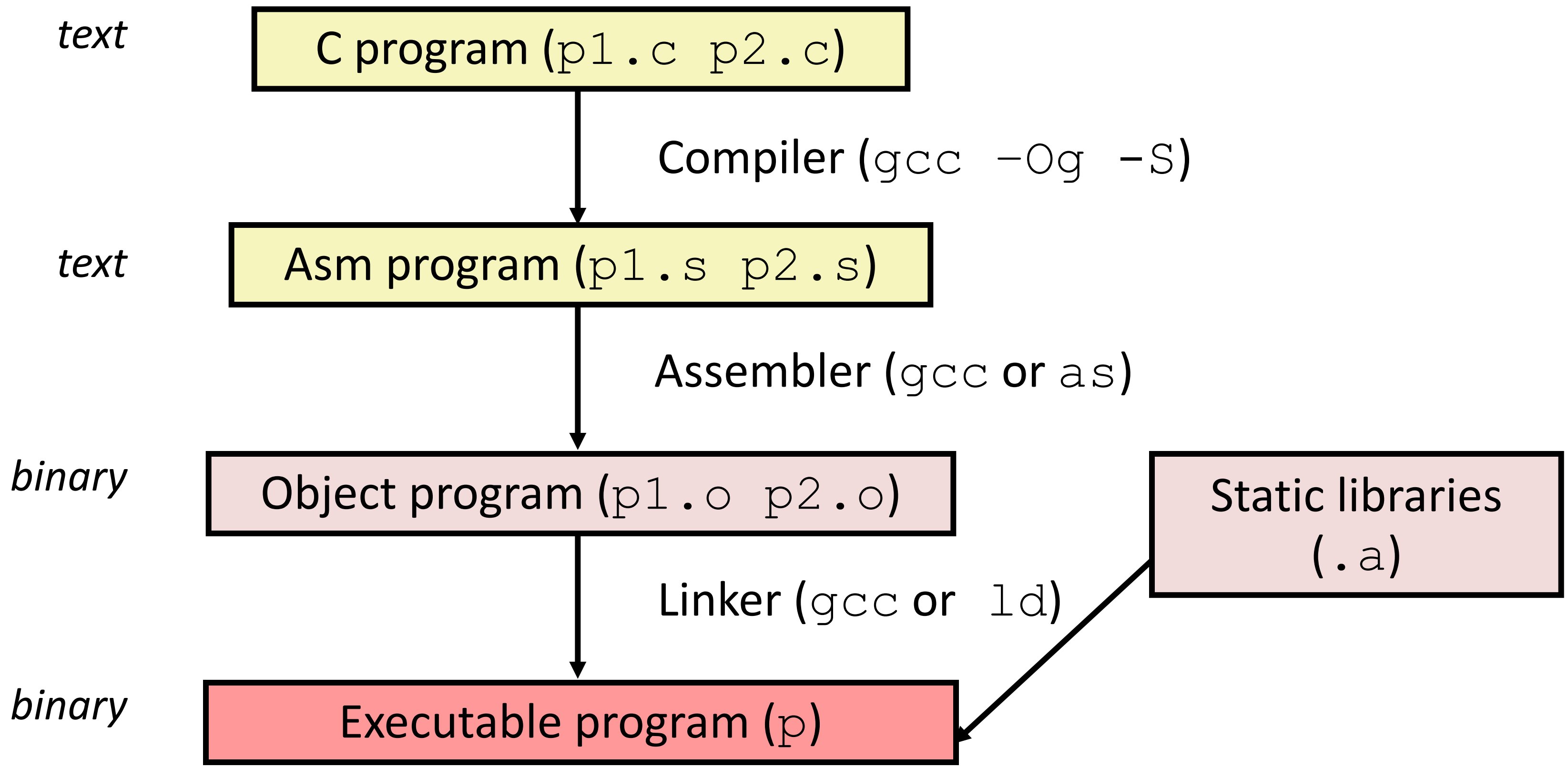


Programmer-Visible State

- **PC: Program counter**
  - Indicates the address of next instruction
  - Called "%rip" (x86-64)
- **Register file**
  - Heavily used program data
  - 16 named locations storing 64bit values
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching
- **Memory**
  - Byte addressable array
  - Code and user data
  - Stack to support procedures

Turning C into Object Code

- Code in files **`p1.c p2.c`**
- Compile with command: **`gcc -Og p1.c p2.c -o p`**
  - Use basic optimizations (**`-Og`**) [New to recent versions of GCC]
  - Put resulting binary in file **`p`**



Compiling Into Assembly

C Code (sum.c)

```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

Generated x86-64 Assembly

```
sumstore:
   pushq   %rbx
   movq    %rdx, %rbx
   call    plus
   movq    %rax, (%rbx)
   popq    %rbx
   ret
```

Obtain (on shark machine) with command

```
gcc -Og -S sum.c
```

Produces file `sum.s`

Warning: This is the output of the textbook. We will get very different results on our machines due to different versions of gcc and different compiler settings.

Vulcan server output

```
        .file   "longplus.c"
        .text
        .globl  sumstore
        .type   sumstore, @function
sumstore:
.LFB0:
        .cfi_startproc
        pushq   %rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16
        movq    %rdx, %rbx
        call    plus
        movq    %rax, (%rbx)
        popq    %rbx
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
.LFE0:
        .size   sumstore, .-sumstore
        .ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-28)"
        .section        .note.GNU-stack,"",@progbits
```