Binary Representation

CSE 410 22wi

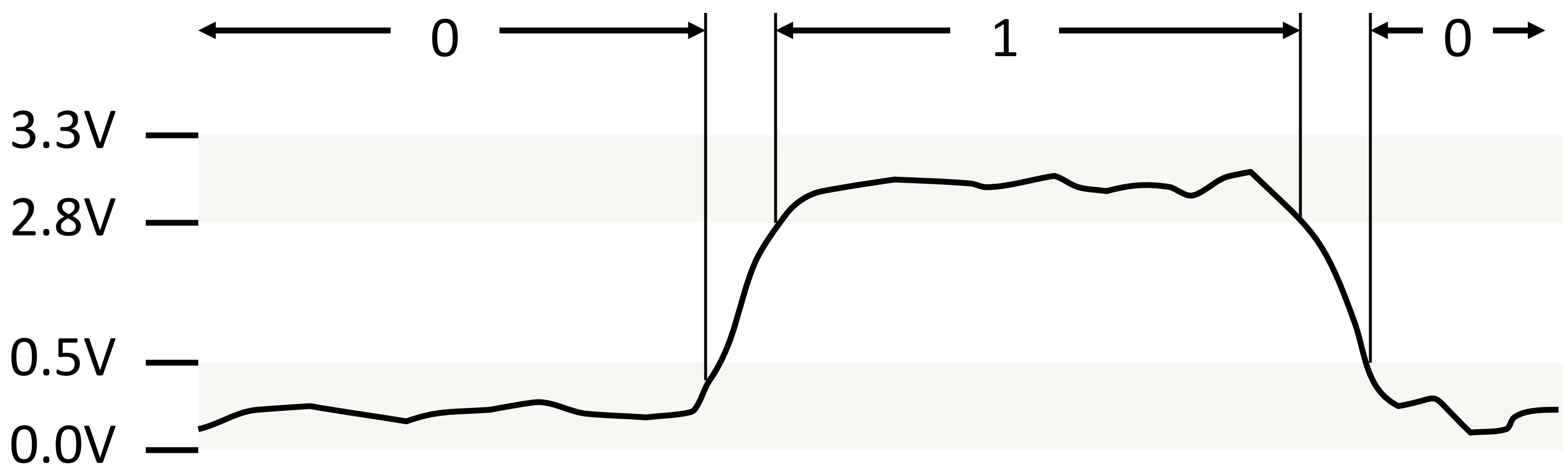
Lecture 03

Lecture Outline

- Binary
- Decimal, Binary, and Hexadecimal Integers
- Why Place Value Representation?
- Floating Point Representation
- Character Representation
- Pointer Representation
- Array Representation
- Structure (Object) Representation

First: Why Binary?

- Electronic implementation
  - Easy to store
  - Reliably transmitted on noisy and inaccurate wires



- Other bases possible:
  - Distinguish more voltage levels
  - DNA data storage (base 4: A, C, G, T)

"binary" vs "digital"

Bit

- A bit is a single binary value
- "Binary" means there are (only) two distinct values
  - in computers, high and low voltage
- We can map the two values to any other pair of values
  - Orange vs Apple; Up vs Down; 8 vs 10; 0 vs 1; true vs false
- Of these, the last two have many attractive properties
  - 0 and 1 → base-2 (binary) integers
  - true and false → Boolean circuits

Bit (Logical) Operations

- Unary operation
  - not
    - ~ 1 == 0
    - ~ 0 == 1
- Binary operations
  - and
    - 0 & 0 == 0
    - 0 & 1 == 0
    - 1 & 0 == 0
    - 1 & 1 == 1

Operators are written as in C (and many other languages)

Note that operator & is different from operator &&

Bit Operations

- Binary Operations
  - or
    - 0 | 0 == 0
    - 0 | 1 == 1
    - 1 | 0 == 1
    - 1 | 1 == 1
  - xor (“exclusive or”)
    - 0 ^ 0 == 0
    - 0 ^ 1 == 1
    - 1 ^ 0 == 1
    - 1 ^ 1 == 0

Bit Strings

- A bit string is a concatentation of bits
  - Example 0 1 0 1 0 1 1 1

- Terminology:

Common Term	Usual #bits
Byte	8
Word	32
Long word	64
Half-word	16
Nibble	4

Bit Strings: Logical Operations

- The bit operators can be applied to bit strings
  - ```
      0 1 0 1 0 1 1 1
    & 1 1 0 0 0 1 1 0
      ---------------
      0 1 0 0 0 1 1 0
    ```
  - Similarly for |, ^, and ~
    ```

Bit Strings: Shift Operations

- Left shift: <<
  - Throw away bits that spill off the string to the left

    0 1 0 1 0 1 0 1 << 1 == ~~[0]~~ 1 0 1 0 1 0 1 0

    0 1 0 1 0 1 0 1 << 3 == ~~[0 1 0]~~ 1 0 1 0 1 0 0 0

- Right shift logical: >>
  - Shifts bits to the right, inserting 0's from the left

    1 1 0 1 0 1 0 1 >> 1 == 0 1 1 0 1 0 1 0 ~~[1]~~

    1 1 0 1 0 1 0 1 >> 3 == 0 0 0 1 1 0 1 0 [1 0 1]

- Right shift arithmetic: >>

  We'll see why in a bit...

  - Right shift arithmetic propagates the high order bit
    - 0 1 0 1 0 1 0 1 >> 3 == 0 0 0 0 1 0 1 0
    - 1 0 1 0 1 0 1 0 >> 3 == 1 1 1 1 0 1 0 1

Bit Masks: “and masks”

- “and masks” turn off bits wherever the mask has a 0 and copies bits wherever the mask has a 1
  - Example mask: 0 0 0 0 0 0 0 1
    - and’ed with another 8 bit string, it copies the low order bit of the other string and sets everything else to zero

```
  1 1 1 1 1 1 1 1
& 0 0 0 0 0 0 0 1
  ---------------
  0 0 0 0 0 0 0 1
```

    - Other masks:
      - 0 0 0 0 0 0 1 1 => copy two low order bits
      - 0 0 0 0 1 1 0 0 => copy bits 2 and 3
      - etc.

Forcing bits on: “or masks”

- “or masks” turn on bits wherever the mask has a 1 and copy bits wherever it has a 0
  - Example mask: 0 0 0 0 1 0 0 1

```
  1 0 1 0 1 0 1 0
| 0 0 0 0 1 0 0 1
  ---------------
  1 0 1 0 1 0 1 1
```

Lecture Outline

- Binary
- Decimal, Binary, and Hexadecimal Integers
- Why Place Value Representation
- Floating Point Representation
- Character Representation
- Pointer Representation
- Array Representation
- Structure Representation

Integers and Integer Representations

- What is 7061?
  - It's a "place value" representation of an integer
  - We could equally write
    $7*10^3 + 0 * 10^2 + 6 * 10^1 + 1*10^0$
    but that's a lot less convenient

- What about 70000000000000000000061?
  - It might be handier to write $7*10^{22} + 61$

- There is no "right representation" there are just ones that are more convenient than others

Place value representation

- We write n consecutive digits, numbering them 0 to n-1 starting from the right. Place j has value b^j for some base b.
- We write in each place a *digit*. There are b digits, representing the numbers 0, 1, 2, …, b-1.

d_3	d_2	d_1	d_0
b^3	b^2	b^1	b^0

- The place value string represents the integer $d_{n-1}b^{n-1} + d_{n-2}b^{n-2} + \ldots + d_0b^0$

Example: 1024_{10}

- b=10 (decimal)
  - Digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - 1024 means $1*10^3 + 0 *10^2 + 2*10^1 + 4*10^0$
- b=2 (binary)
  - Digits are 0, 1
  - 10000000000 means $1*2^{10}$ (plus a lot of "zero times x" terms)
    - Which is 1024_{10}

Simplifying representations

- Which is bigger, $231237943432586732275839_{10}$ or $23123794343584332235839_{10}$?

- We (humans) prefer representations with fewer digits
- We can reduce the number of digits a factor of k by raising the base by a power of k.
  - E.g., instead of base 10, use base 1000
    - Of course, we now need a 1000 different symbols for digits

- 231,237,943,432,586,732,275,839

  versus

  23,123,794,343,584,332,235,839

Simplifying binary

- Start with (32-bit) binary representation:
  00000001001000110100010101100111

- **Octal**: Raise the base by a power of 3 (so, base 8)
  00 000 001 001 000 110 100 010 101 100 111
  0 0 1 1 0 6 4 2 5 4 7

- **Hexadecimal (Hex)**: Raise the base by a power of 4 (base 16)
  0000 0001 0010 0011 0100 0101 0110 0111
  0 1 2 3 4 5 6 7

Hexadecimal

- Grouping by four bits is handy
  - Memories are always a multiple of 8 bits in length
- Hex digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
  - Correspond to values in base 10 of 0, 1, ..., 9, 10, 11, 12, 13, 14, 15
  - Case insensitive
- Often (but not necessarily) written like 0x0FC0138B
  - 0000 1111 1100 0000 0001 0011 1000 1011

Hex ⇔ Binary

Hex Digit	Binary String
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

What is 0xFFFF in binary?

Is 0x237E even or odd?

We should specify what base we're using when writing integers.

In C:

- 123 is a decimal constant
- 0123 is an octal constant
- 0X0123 is a hex constant

Lecture Outline

- Binary
- Decimal, Binary, and Hexadecimal Integers
- Why Place Value Representation
  - And why not
- Floating Point Representation
- Character Representation
- Pointer Representation
- Array Representation
- Structure Representation

Addition with Place Value Representations

- Addition is easy with the standard algorithm (carry ripple)

1	1			
0	0	1	0	2
0	1	1	1	7
1	0	0	1	9

- One problem: what about addition of negative numbers?

  24
  +(-7)

- Another problem: Hey, what about negative numbers at all?

- Third problem: Overflow

Overflow

- A fixed amount of space is allocated for each value on a computer
  - For integers, usually 1, 2, 4, or 8 bytes (8, 16, 32, or 64 bits)

- Q: What if the result is too big to fit in that much space?
  A: Too bad. The highest order bit is thrown away.

- That's called overflow

	1	1	1		
	0	1	0	1	5
	1	0	1	1	11
~~1~~	0	0	0	0	0

Representing Signed Integers: Two's Complements

- "Two's complement" is a representation for positive and negative integers
  - Addition is always addition, even if one or both values are negative
  - About half the bit strings are negative and half are positive

000	001	010	011	100	101	110	111
0	1	2	3	-4	-3	-2	-1

Verify that x + -x == 0

Properties of Two's Complement Integers

	000	001	010	011	100	101	110	111
signed	**0**	**1**	**2**	**3**	**-4**	**-3**	**-2**	**-1**
unsigned	0	1	2	3	4	5	6	7

- If you count up from 0 by 1, you *wrap* from the largest positive integer to the smallest negative integer
- If the high order bit is 0, the number is non-negative.
  If it's 1, the number is negative.
- If the low order bit is 0 the number is even, otherwise it's odd
- -X = ~X + 1
  - Example: -011 = 100 + 1 = 101
- There is one more negative value than positive values
  - -<most negative int> = <most negative int>

Unsigned Integers

	000	001	010	011	100	101	110	111
signed	0	1	2	3	-4	-3	-2	-1
unsigned	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**

- All values are non-negative
  - About twice as many non-negative values can be represented compared with signed
  - Useful (in any case) for things like array indices (since they can't sensibly be negative)
  - If X is an unsigned integer, -X is a mistake
- You get the same bit string result adding bit strings as unsigned values as you do adding them as signed
- If the low order bit is 0 the number is even, otherwise it's odd

Overflow

- Overflow occurs when the result doesn't fit in the limited number of bits you have
  - 0001 + 0111 => 1000
    1 + 7 = -8
  - You can overflow when subtracting or multiplying as well
- Unsigned integers also overflow
  - 0001 + 0111 = 1000
    1 7 8 [no overflow]
  - 0001 + 1111 = 0000
    1 15 0 [overflow]

Lecture Outline

- Binary
- Decimal, Binary, and Hexadecimal Integers
- Why Place Value Representation
  - And why not
- Floating Point Representation
- Character Representation
- Pointer Representation
- Array Representation
- Structure Representation

Floating Point Representation Overview

- We have only 32 bits, so we have only 2^{32} different values we can represent

- We're going to do the binary version of scientific notation: 2.357×10^{14}
  - If I had six decimal digits of space, I might write this as 142357

- Different choices for how to use the digits (bits) have different:
  - range – roughly, how big the exponent can be
  - precision – basically the number of significant digits in the fraction

32-bit Binary Floats

- Called “single precision” floats
- Value is [+/-] [fraction] x $2^{[exponent]}$
- The 32 bits are used as:
  - High order bit is the sign of the value: 1 for negative, 0 for non-negative
  - The next 8 bits are the signed (two’s complement) value for the exponent: 127 to -128
  - The remaining 23 bits are the fraction
- Range: approximately 2.0×10^{38} to 2.0×10^{-38}
- Numbers can overflow: exponent gets too big
- Numbers can underflow: exponent gets too small

Lecture Outline

- Binary
- Decimal, Binary, and Hexadecimal Integers
- Why Place Value Representation
  - And why not
- Floating Point Representation
- Character Representation
- Pointer Representation
- Array Representation
- Structure Representation

Character Representation

- We simply agree on a mapping from bit strings to characters
  - “Everyone” knows what the mapping is
  - The compiler inserts the agreed bit string when you write ‘A’
  - The output system writes A when it sees that bit string

- There is more than one agreed representation
- ASCII
  - Historically the agreed mapping
  - Fixed, 8-bit long strings
- Unicode
  - Variable length encoding: 8, 16, or 32 bits per character
  - Many, many more bit strings, so many, many more characters/alphabets

ASCII

0	<NUL>	32	<SPC>	64	@	96	`	128	Ä	160	†	192	¿	224	‡
1	<SOH>	33	!	65	A	97	a	129	Å	161	°	193	¡	225	·
2	<STX>	34	"	66	B	98	b	130	Ç	162	¢	194	¬	226	‚
3	<ETX>	35	#	67	C	99	c	131	É	163	£	195	√	227	„
4	<EOT>	36	$	68	D	100	d	132	Ñ	164	§	196	ƒ	228	‰
5	<ENQ>	37	%	69	E	101	e	133	Ö	165	•	197	≈	229	Â
6	<ACK>	38	&	70	F	102	f	134	Ü	166	¶	198	∆	230	Ê
7	<BEL>	39	'	71	G	103	g	135	á	167	ß	199	«	231	Á
8	<BS>	40	(	72	H	104	h	136	à	168	®	200	»	232	Ë
9	<TAB>	41	)	73	I	105	i	137	â	169	©	201	…	233	È
10	<LF>	42	*	74	J	106	j	138	ä	170	™	202		234	Í
11	<VT>	43	+	75	K	107	k	139	ã	171	´	203	À	235	Î
12	<FF>	44	,	76	L	108	l	140	å	172	¨	204	Ã	236	Ï
13	<CR>	45	-	77	M	109	m	141	ç	173	≠	205	Õ	237	Ì
14	<SO>	46	.	78	N	110	n	142	é	174	Æ	206	Œ	238	Ó
15	<SI>	47	/	79	O	111	o	143	è	175	Ø	207	œ	239	Ô
16	<DLE>	48	0	80	P	112	p	144	ê	176	∞	208	–	240	
17	<DC1>	49	1	81	Q	113	q	145	ë	177	±	209	—	241	Ò
18	<DC2>	50	2	82	R	114	r	146	í	178	≤	210	“	242	Ú
19	<DC3>	51	3	83	S	115	s	147	ì	179	≥	211	”	243	Û
20	<DC4>	52	4	84	T	116	t	148	î	180	¥	212	‘	244	Ù
21	<NAK>	53	5	85	U	117	u	149	ï	181	µ	213	’	245	ı
22	<SYN	54	6	86	V	118	v	150	ñ	182	∂	214	÷	246	ˆ
23	<ETB>	55	7	87	W	119	w	151	ó	183	∑	215	◊	247	˜
24	<CAN>	56	8	88	X	120	x	152	ò	184	∏	216	ÿ	248	¯
25	<EM>	57	9	89	Y	121	y	153	ô	185	π	217	Ÿ	249	˘
26	<SUB>	58	:	90	Z	122	z	154	ö	186	∫	218	⁄	250	˙
27	<ESC>	59	;	91	[	123	{	155	õ	187	ª	219	€	251	˚
28	<FS>	60	<	92	\	124	|	156	ú	188	º	220	‹	252	¸
29	<GS>	61	=	93	]	125	}	157	ù	189	Ω	221	›	253	˝
30	<RS>	62	>	94	^	126	~	158	û	190	æ	222	ﬁ	254	˛
31	<US>	63	?	95	_	127	<DEL>	159	ü	191	ø	223	ﬂ	255	ˇ

Character Strings

- A string is an array of characters

S	e	a	t	t	l	e

- Suppose memory had this. What is "the string"?

S	e	a	t	t	l	e	W	A

- Two common choices

7	S	e	a	t	t	l	e	2	W	A

S	e	a	t	t	l	e	\0	W	A	\0

Lecture Outline

- Binary
- Decimal, Binary, and Hexadecimal Integers
- Why Place Value Representation
  - And why not
- Floating Point Representation
- Character Representation
- Pointer Representation
- Array Representation
- Structure Representation

Pointers (or Not Pointers?)

- If you write this in some language

```
X = 10;
Y = X;   // Is Y a new name for X, or is Y a clone of X?
X = 20;
```

what is the value of Y at this point?

- If 10, then X and Y name different things
  - Y is not a pointer (reference)
- If 20, then Y is an alias for X (names the same thing)
  - Y is a pointer (reference)

- In Java, object variables are <u>references</u>
- In C, things aren't pointers unless you go out of your way to make them so

Pointers in C

- int   x;   // x names 32-bits that we'll use as an int
- int   *p;   // p names a 32-bit string that can hold a
  // memory address. We'll use the bit string
  // at that address as an int
- p = &x;   // set p's 32 bits to the address of x
- *p = 4;   // sets the word of memory pointed at by p
  // to 4 (i.e., x = 4)

C Language Pointers

```
int    x;
int   *p;
p = &x;
*p = 4;
```

```
      .text
      addi   x2, x0, x      # x2 = &x
      sw      x2, p         # p = &x
      addi    x3, x0 ,4     # 4
      sw       x3, 0(x2)    # *p = 4

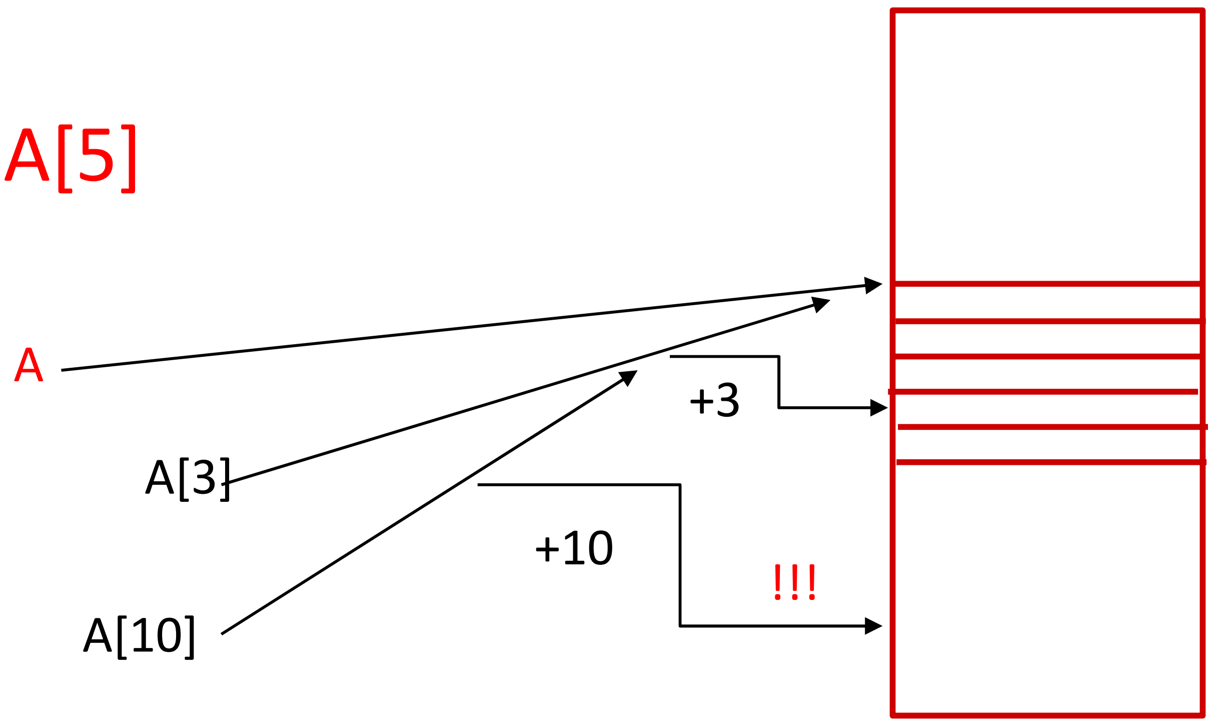
      .data
      ...
x:    .word   0
       ...
p:   .word    0
```

Lecture Outline

- Binary
- Decimal, Binary, and Hexadecimal Integers
- Why Place Value Representation
  - And why not
- Floating Point Representation
- Character Representation
- Pointer Representation
- Array Representation
- Structure Representation

Arrays

- Arrays are just consecutive words of memory
  - The CPU doesn’t know anything about “arrays”
- The array name is the base address of the array
- The index is the offset from that base address
- int A[5]



Arrays

- int A[10];
- A[3] = 4;

```
.text
addi    x1, x0, 4     # 4
addi    x2, x0, A     # base address of A
sw      x1, 3(x2)     # store at A[3]
```

- int *pA = <something>;
- pA[3] = 4;

```
.text
addi    x1, x0, 4
lw      x2, <smthgn> # establish value for pA
sw      x1, 3(x2)     # store at A[3]
```

Lecture Outline

- Binary
- Decimal, Binary, and Hexadecimal Integers
- Why Place Value Representation
  - And why not
- Floating Point Representation
- Character Representation
- Pointer Representation
- Array Representation
- Structure Representation

Structure Representation

- struct person {
  int id;
  int department;
  };
  struct person *p;

This defines a type. It doesn't allocate memory.
"id" and "department" are offsets from the base of a struct person.
They have values 0 and 1 respectively.

p can "point to" memory used as a struct person

- ...
  p->department = 10;

```
addi  x1, x0, 10
lw    x2, p
sw    x1, 1(x2)   # "department" is an offset
```

It's a similar idea for objects. They're hunks of consecutive memory.
Field names are offsets into those hunks.

Lecture Outline

- Binary
- Decimal, Binary, and Hexadecimal Integers
- Why Place Value Representation
  - And why not
- Floating Point Representation
- Character Representation
- Pointer Representation
- Array Representation
- Structure Representation
- Strings

Summary

- 01100001
  - Is its value as an (8 bit) int positive, negative, or zero?
  - Is its value as an int an even number?
  - What is its value as an int expressed in decimal?
  - What is its value as an int expressed in hex?
  - Might it be a float?
  - What is its value as a char?
  - Is it a C string?
  - Could it be the start of a C string?
  - Might it be an array?
  - Might it be a struct?