BINARY

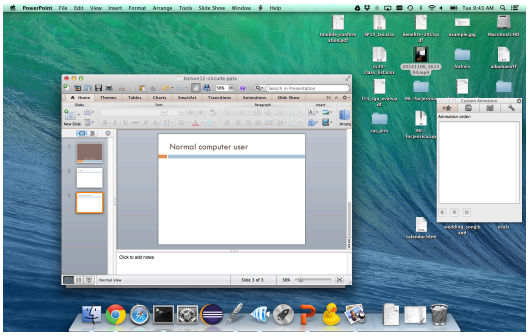
David Kauchak
CS 52 – Spring 2017

Admin

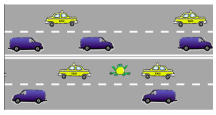
- Assignment 5

Diving into your computer

Normal computer user



After intro CS



```
import objectdraw.*;

public class Frog {
   // Height of the frog image
   private static final double FROG_HEIGHT = 48;

   // This should refer to the image of the frog. Note that it is not
   // initialize by
   // the code we have provided.
   private VisibleImage frogImage;

   public Frog() {

   }

   public boolean overlaps(VisibleImage vehicleImage) {
      return false; // YOU NEED TO CHANGE THIS!
   }

   public void kill() {

   }

   public void reincarnate() {

   }

   public void hopToward(Location point) {

   }

   public boolean isAlive() {
      return false; // YOU NEED TO CHANGE THIS!
   }
}
```

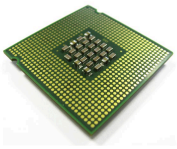
After 5 weeks of cs52

```
import objectdraw.*;

public class Frog {
   // Height of the frog image
   private static final double FROG_HEIGHT = 48;

   // This should refer to the image of the frog. Note that it is not
   // initialize by
   // the code we have provided.
   private VisibleImage frogImage;

   public Frog() {

   }

   public boolean overlaps(VisibleImage vehicleImage) {
      return false; // YOU NEED TO CHANGE THIS!
   }

   public void kill() {

   }

   public void reincarnate() {

   }

   public void hopToward(Location point) {

   }

   public boolean isAlive() {
      return false; // YOU NEED TO CHANGE THIS!
   }
}
```

```
        loa r1 r0 0         ; get a value for a
        loa r2 r0 0         ; get a value for b
        add r3 r0 r0        ; result = 0;

        ble r1 r0 endloop   ; test if a <= 0
loop
        add r3 r3 r2        ; result += b;
        sbc r1 r1 1         ; a--;
        blt r0 r1 loop      ; return for another iteration

endloop
        sto r0 r3 0         ; write the value of product
        hlt                 ; halt
        end
```

What now?

```
        loa r1 r0 0         ; get a value for a
        loa r2 r0 0         ; get a value for b
        add r3 r0 r0        ; result = 0;

        ble r1 r0 endloop   ; test if a <= 0
loop
        add r3 r3 r2        ; result += b;
        sbc r1 r1 1         ; a--;
        blt r0 r1 loop      ; return for another iteration

endloop
        sto r0 r3 0         ; write the value of product
        hlt                 ; halt
        end
```



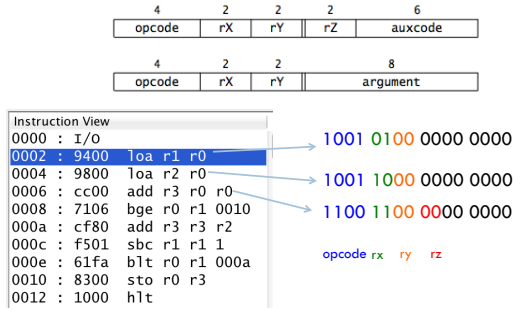
One last note on CS52

```
Instruction View
0000 : I/O
0002 : 9400  loa r1 r0
0004 : 9800  loa r2 r0
0006 : cc00  add r3 r0 r0
0008 : 7106  bge r0 r1 0010
000a : cf80  add r3 r3 r2
000c : f501  sbc r1 r1 1
000e : 61fa  blt r0 r1 000a
0010 : 8300  sto r0 r3
0012 : 1000  hlt
```

memory address | binary representation of code | instructions (assembly code)

How do we get this?

Encoding assembly instructions



Binary numbers revisited

What number does 1001 represent in binary?

Depends!
Is it a signed number or unsigned?
If signed, what convention are we using?

Twos complement

For a number with *n* digits the high order bit represents -2^{n-1}

unsigned	2^3	2^2	2^1	2^0
signed (twos complement)	-2^3	2^2	2^1	2^0

Twos complement

What number is it?

unsigned	1	0	0	1	9
	2^3	2^2	2^1	2^0	
signed (twos complement)	1	0	0	1	-7
	-2^3	2^2	2^1	2^0	

Twos complement

What number is it?

unsigned	1	1	1	1	15
	2^3	2^2	2^1	2^0	
signed (twos complement)	1	1	1	1	-1
	-2^3	2^2	2^1	2^0	

Twos complement

What number is it?

unsigned	1	1	0	0	12
	2^3	2^2	2^1	2^0	
signed (twos complement)	1	1	0	0	-4
	-2^3	2^2	2^1	2^0	

Twos complement

How many numbers can we represent with each approach using 4 bits?

16 (2^4) numbers, 0000, 0001, …., 1111
Doesn't matter the representation!

unsigned				
	2^3	2^2	2^1	2^0
signed (twos complement)				
	-2^3	2^2	2^1	2^0

Twos complement

How many numbers can we represent with each approach using 32 bits?

$2^{32} \approx$ 4 billion numbers

unsigned				
	2^3	2^2	2^1	2^0
signed (twos complement)				
	-2^3	2^2	2^1	2^0

Twos complement

What is the range of numbers that we can represent for each approach with 4 bits?

unsigned: 0, 1, … 15
signed: -8, -7, …, 7

unsigned	2^3	2^2	2^1	2^0
signed (twos complement)	-2^3	2^2	2^1	2^0

binary representation	unsigned	
0000	0	
0001	1	
0010	?	
0011		
0100		
0101		
0110		
0111		
1000		
1001		
1010		
1011		
1100		
1101		
1110		
1111		

binary representation	unsigned	twos complement
0000	0	?
0001	1	
0010	2	
0011	3	
0100	4	
0101	5	
0110	6	
0111	7	
1000	8	
1001	9	
1010	10	
1011	11	
1100	12	
1101	13	
1110	14	
1111	15	

binary representation	unsigned	twos complement
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	?
1001	9	
1010	10	
1011	11	
1100	12	
1101	13	
1110	14	
1111	15	

binary representation	unsigned	twos complement
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	?
1010	10	
1011	11	
1100	12	
1101	13	
1110	14	
1111	15	

binary representation	unsigned	twos complement
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

binary representation	unsigned	twos complement
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

How can you tell if a number is negative?

binary representation	unsigned	twos complement
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

High order bit!

A two's complement trick

You can also calculate the value of a negative number represented as twos complement as follows:

- Flip all of the bits (0 → 1 and 1→ 0)
- Add 1
- The resulting number is the magnitude of the original negative number

flip the bits add 1

1101 → 0010 → 0011 → -3

A two's complement trick

You can also calculate the value of a negative number represented as twos complement as follows:

- Flip all of the bits (0 → 1 and 1→ 0)
- Add 1
- The resulting number is the magnitude of the original negative number

flip the bits add 1

1110 → 0001 → 0010 → -2

Shifting

Shifting shifts the binary representation of the number right or left

Shifting

Shifting shifts the binary representation of the number right or left

37 >> 2 ?

number to be shifted

right shift

number of positions to shift

Shifting

Shifting shifts the binary representation of the number right or left

37 >> 2

37 → 100101 → 1001 → 9

number in binary | shift right two positions (discard bits shifting off) | decimal form

Shifting

Shifting shifts the binary representation of the number right or left

37 >> 3 ?

Shifting

Shifting shifts the binary representation of the number right or left

37 >> 3

37 → 100101 → 100 → 4

number in binary | shift right three positions (discard bits shifting off) | decimal form

Shifting with fixed bit representations

In real computers, we generally have a fixed number of bits we use to represent a number (e.g. 8-bits, 16-bits, 32-bits)

Shifting 8-bit numbers

Shifting shifts the binary representation of the number right or left

37 >> 2

What is 37 as an 8-bit binary number?

37 → 00100101

pad with 0s

number in binary

Shifting 8-bit numbers

Shifting shifts the binary representation of the number right or left

37 >> 2

How do we fill in the leftmost bits?

37 → 00100101 →

number in binary

shift right two positions (discard bits shifting off)

Shifting 8-bit numbers

Shifting shifts the binary representation of the number right or left

37 >> 2

How do we fill in the leftmost bits?

37 → 00100101 → 00001001

number in binary

shift right two positions (discard bits shifting off)

Shifting 8-bit numbers

Shifting shifts the binary representation of the number right or left

37 >> 2

37 → 00100101 → 00001001 → 9

number in binary

shift right two positions (discard away bits shifting off)

decimal form

Shifting 8-bit numbers

Shifting shifts the binary representation of the number right or left

15 << 2

?

Shifting 8-bit numbers

Shifting shifts the binary representation of the number right or left

15 << 2

15 ⇒ ?

number in binary

Shifting 8-bit numbers

Shifting shifts the binary representation of the number right or left

15 << 2

15 ⇒ 00001111

number in binary

Shifting 8-bit numbers

Shifting shifts the binary representation of the number right or left

15 << 2

15 ⇒ 00001111 ⇒ ?

number in binary

shift left two positions
(discard bits shifting off)

Shifting 8-bit numbers

Shifting shifts the binary representation of the number right or left

15 << 2

15 → 00001111 → 001111??

number in binary — shift left two positions (discard bits shifting off)

Shifting 8-bit numbers

Shifting shifts the binary representation of the number right or left

15 << 2

15 → 00001111 → 00111100

number in binary — shift left two positions (discard bits shifting off)

Shifting 8-bit numbers

Shifting shifts the binary representation of the number right or left

15 << 2

15 → 00001111 → 00111100 → 60

number in binary — shift left two positions (discard bits shifting off) — decimal form

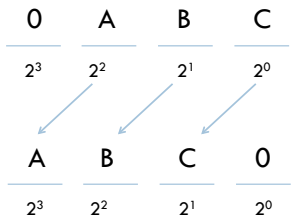
Shifting mathematically

What does **left** shifting by one position do mathematically?

0	A	B	C
2^3	2^2	2^1	2^0

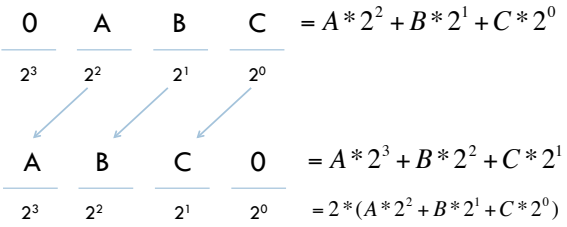
Shifting mathematically

What does **left** shifting by one position do mathematically?



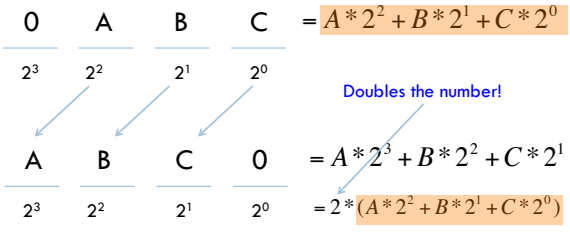
Shifting mathematically

What does **left** shifting by one position do mathematically?



Shifting mathematically

What does **left** shifting by one position do mathematically?



Shifting mathematically

What does **left** shifting by ***n*** positions do mathematically?

Multiply by 2^n (double n times)

Shifting mathematically

What does **right** shifting by one position do mathematically?

0	A	B	C
2^3	2^2	2^1	2^0

Shifting mathematically

What does **right** shifting by one position do mathematically?

0	A	B	C
2^3	2^2	2^1	2^0

0	0	A	B
2^3	2^2	2^1	2^0

Shifting mathematically

What does **right** shifting by one position do mathematically?

0	A	B	C	$= A*2^2 + B*2^1 + C*2^0$
2^3	2^2	2^1	2^0	

0	0	A	B	$= A*2^1 + B*2^0$
2^3	2^2	2^1	2^0	$= (A*2^2 + B*2^1 + C*2^0)$ div 2

Shifting mathematically

What does **right** shifting by one position do mathematically?

0	A	B	C	$= A*2^2 + B*2^1 + C*2^0$
2^3	2^2	2^1	2^0	

Integer divide by 2

0	0	A	B	$= A*2^1 + B*2^0$
2^3	2^2	2^1	2^0	$= (A*2^2 + B*2^1 + C*2^0)$ div 2

Shifting mathematically

What does **right** shifting by ***n*** positions do mathematically?

Integer division by 2^n (halve n times)

Shifting 4-bit numbers

Shifting shifts the binary representation of the number right or left

-4 >> 1

?

Shifting 4-bit numbers

Shifting shifts the binary representation of the number right or left

-4 >> 1

What is -4 as a 4-bit binary number?

-4 ➡ 1100

number in binary

Shifting 4-bit numbers

Shifting shifts the binary representation of the number right or left

-4 >> 1

How do we fill in the leftmost bit?

-4 ➡ 1100 ➡ ?110

number in binary

shift right one position (discard bits shifting off)

Shifting 4-bit numbers

Two types of right shifts:
- logical shift: always shift in 0s
- arithmetic shift: shift in the same as the high-order bit

-4 → 1100 → ?110

number in binary — shift right one position (discard bits shifting off)

Shifting 4-bit numbers

Two types of right shifts:
- logical shift: always shift in 0s
- arithmetic shift: shift in the same as the high-order bit

-4 → 1100 → ?110

number in binary — shift right one position (discard bits shifting off)

Shifting 4-bit numbers

Two types of right shifts:
- logical shift: always shift in 0s
- arithmetic shift: shift in the same as the high-order bit

-4 → 1100 → 1110

number in binary — shift right one position (discard bits shifting off)

Shifting 4-bit numbers

Two types of right shifts:
- logical shift: always shift in 0s
- arithmetic shift: shift in the same as the high-order bit

-4 >> 1

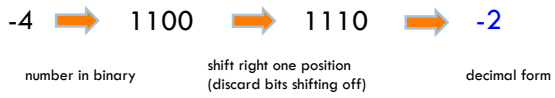
-4 → 1100 → 1110 → ?

number in binary — shift right one position (discard bits shifting off) — decimal form

Shifting 4-bit numbers

Two types of right shifts:
- logical shift: always shift in 0s
- arithmetic shift: shift in the same as the high-order bit

-4 >> 1



number in binary | shift right one position (discard bits shifting off) | decimal form

Shifting 4-bit numbers

Shifting shifts the binary representation of the number right or left

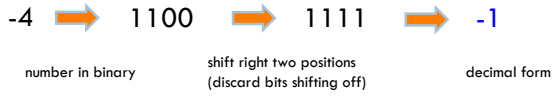
-4 >> 2

?

Shifting 4-bit numbers

Two types of right shifts:
- logical shift: always shift in 0s
- arithmetic shift: shift in the same as the high-order bit

-4 >> 2



number in binary | shift right two positions (discard bits shifting off) | decimal form

Arithmetic shifting mathematically

What does **right** arithmetic shifting by ***n*** positions do mathematically for **signed numbers**?

Integer division by 2^n (halve n times)

Same thing!!

Shifting 4-bit numbers

Two types of right shifts:
- logical shift: always shift in 0s
- arithmetic shift: shift in the same as the high-order bit

-4 → 1100 → ?110

number in binary — shift right one position (discard bits shifting off)

Shifting 4-bit numbers

Two types of right shifts:
- logical shift: always shift in 0s
- arithmetic shift: shift in the same as the high-order bit

-4 → 1100 → 0110

number in binary — shift right one position (discard bits shifting off)

Shifting 4-bit numbers

Two types of right shifts:
- logical shift: always shift in 0s
- arithmetic shift: shift in the same as the high-order bit

-4 >>> 1

-4 → 1100 → 0110 → ?

number in binary — shift right one position (discard bits shifting off) — decimal form

Shifting 4-bit numbers

Two types of right shifts:
- logical shift: always shift in 0s
- arithmetic shift: shift in the same as the high-order bit

-4 >>> 1

-4 → 1100 → 0110 → 6

number in binary — shift right one position (discard bits shifting off) — decimal form

Left shifts

Two types of left shifts?
- logical shift: always shift in 0s
- arithmetic shift: ?

arithmetic
-3 << 1 ?

logical
-3 <<< 1 ?

Left shifts

Two types of left shifts?
- logical shift: always shift in 0s
- arithmetic shift: ?

arithmetic
-3 << 1 1101 → 101? (double the number)

logical
-3 <<< 1 1101 → 1010

Left shifts

Two types of left shifts?
- logical shift: always shift in 0s
- arithmetic shift: ?

arithmetic
-3 << 1 1101 → 1010 (double the number)

logical
-3 <<< 1 1101 → 1010

Only one type of left shift

Shifting summarized

Arithmetic shift:
- Right shift n
  - shift n bits to the right
  - discard right n bits
  - left n bits match high-order bits of original number
  - Effect: Integer division by 2^n (halve n times)
- Left shift
  - shift n bits to the left
  - discard left n bits
  - right n bits are 0s
  - Effect: multiply by 2^n (double n times)

Logical shift right:
- left n bits are 0s (no mathematical guarantees for negative numbers)

Adding numbers base 10

Add: 456 and 735

Adding numbers base 10

$$\begin{array}{r} 456 \\ +\ 735 \\ \hline ? \end{array}$$

Adding numbers base 10

$$\begin{array}{r} {\scriptstyle 1\,0\,1} \\ 456 \\ +\ 735 \\ \hline 1191 \end{array}$$

Adding numbers base 5

Add: 223_5 and 414_5

Adding numbers base 5

$$\begin{array}{r} 223_5 \\ +\ 414_5 \\ \hline ? \end{array}$$

Adding numbers base 5

$$\begin{array}{r} {\scriptstyle 1\,0\,1} \\ 223_5 \\ +\ 414_5 \\ \hline 1142_5 \end{array}$$

Adding numbers base 5

$$\begin{array}{rr} {\scriptstyle 1\,0\,1} & \\ 223_5 & 63_{10} \\ +\ 414_5 & 109_{10} \\ \hline 1142_5 & 172_{10} \end{array}$$

Adding numbers base 2

Add: 0001_2 and 0101_2

Adding numbers base 2

$$\begin{array}{r} 0001_2 \\ +\ 0101_2 \\ \hline ? \end{array}$$

Adding numbers base 2

$$\begin{array}{r} {\scriptstyle 0\ 0\ 1} \\ 0001_2 \\ +\ 0101_2 \\ \hline 0110_2 \end{array}$$

Adding numbers base 2

$$\begin{array}{rr} {\scriptstyle 0\ 0\ 1} & \\ 0001_2 & 1_{10} \\ +\ 0101_2 & 5_{10} \\ \hline 0110_2 & 6_{10} \end{array}$$

Addition with 4-bit *twos complement* numbers

$$\begin{array}{r} 0001_2 \\ +\ 0101_2 \\ \hline ? \end{array}$$

Addition with 4-bit *twos complement* numbers

$$\begin{array}{r} {\scriptstyle 0\ 0\ 1\ \ } \\ 0001_2 \\ +\ 0101_2 \\ \hline 0110_2 \end{array}$$

Addition with 4-bit *twos complement* numbers

$$\begin{array}{r} 0110 \\ +\ 0101 \\ \hline ? \end{array}$$

(Note: I'm going to stop writing the base 2 ☺)

Addition with 4-bit *twos complement* numbers

$$\begin{array}{r} {\scriptstyle 1\ \ \ \ } \\ 0110 \\ +\ 0101 \\ \hline 1011? \end{array}$$

Addition with 4-bit *twos complement* numbers

$$\begin{array}{rl} {\scriptstyle 1\ \ \ \ } & \\ 0110 & 6 \\ +\ 0101 & 5 \\ \hline 1011? & \text{-5? (11 unsigned)} \end{array}$$

Overflow! We cannot represent this number (it's too large)

Addition with 4-bit *twos complement* numbers

```
  0110
+ 1101
------
   ?
```

Addition with 4-bit *twos complement* numbers

```
  1 1
  0110
+ 1101
------
  0011
```

Addition with 4-bit *twos complement* numbers

ignore the last carry

```
  1 1
  0110    6
+ 1101   -3
------
  0011    3
```

Subtraction

Ideas?

Subtraction

Negate the 2nd number (flip the bits and add 1)

Add them!

Midterm

Average:	28 (77%)
Q1:	24.9 (70%)
Median:	28.5 (81%)
Q3:	32 (91%)