The Hardware/Software Interface

CSE351 Winter 2011

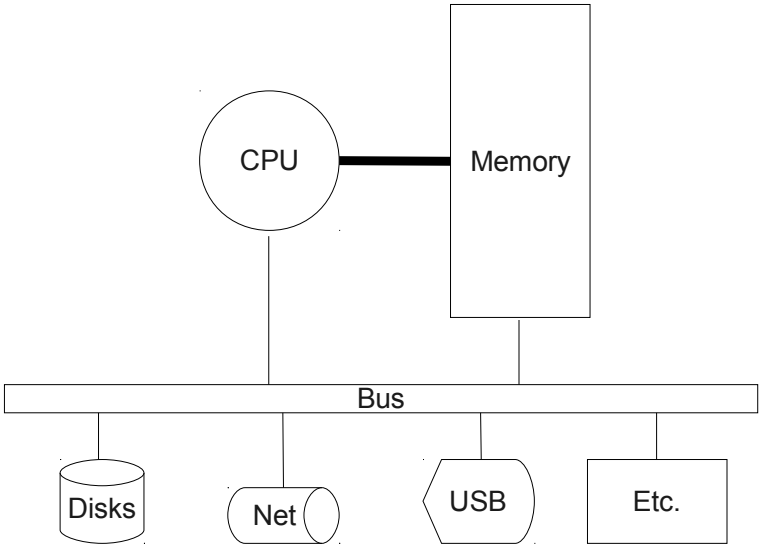
Module 2: Memory

Today's topics

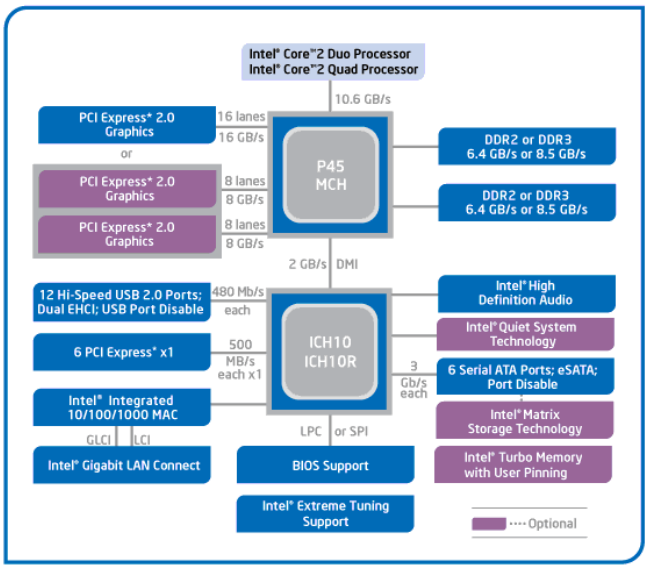
- Part I: Brief hardware overview
- Part II: Memory
  - Memory and its bits, bytes, and integers
  - Representing information as bits
  - Bit-level manipulations
    - Boolean algebra
    - Boolean algebra in C
- Part III: Addresses / Pointers / C Arrays

Part I: Hardware Overview

Hardware: Logical View

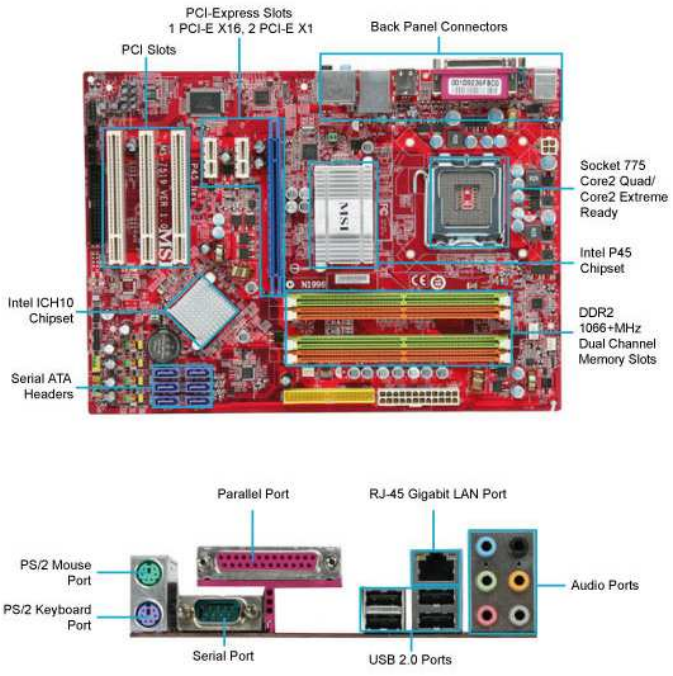


Hardware: Semi-Logical View



Intel® P45 Express Chipset Block Diagram

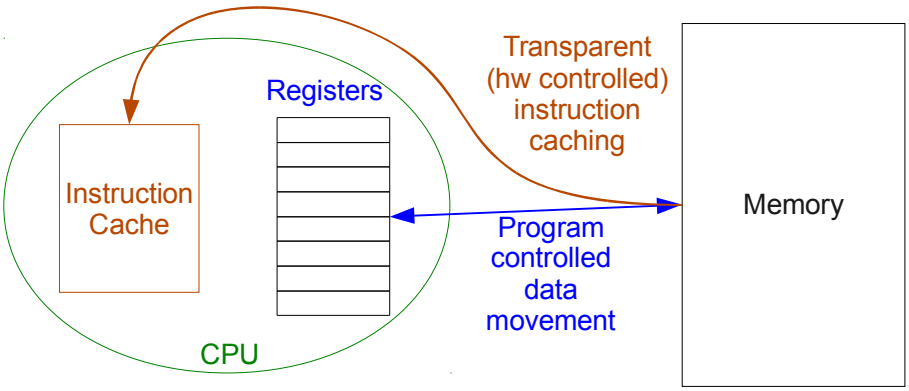
Hardware: Physical View



Performance: It's Not Just CPU Speed

- Data and instructions reside in memory
  - To execute an instruction, it must be fetched onto the CPU
  - Then, the data the instruction operates on must be fetched onto the CPU
- CPU ⇔ Memory bandwidth can be a limiting factor to performance
  - Improving performance 1: hardware improvements to increase memory bandwidth (e.g., DDR → DDR2 → DDR3)
  - Improving performance 2: move less data into/out of the CPU
    - Put some "memory" on the CPU chip
    - *The next slide is just an introduction. We'll see a more full explanation later in the course.*

CPU "Memory": Registers and Instruction Cache



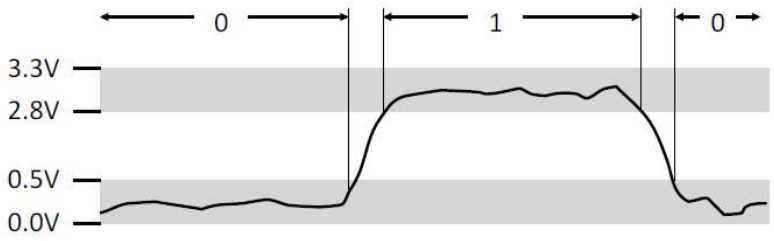
- **There are a fixed number of registers on the CPU**
  - Registers hold data
- **There is an I-cache on the CPU holding recently fetched instructions**
  - If you execute a loop that fits in the cache, the CPU goes to memory for those instructions only once, then executes out of its cache

Part II: Introduction to Memory

Hardware Memory Organization
Memory and C Data Types

Binary, Everywhere

- Why binary?
  - Easy to store with bi-stable elements
  - Reliably transmitted on noisy and inaccurate wires



- Memory contains a very long bit array:
  - 00000001010111110110001010001111010101001000...0_2
- Everything stored in memory is represented by some bit string
  - Numbers, characters, instructions, objects, ...

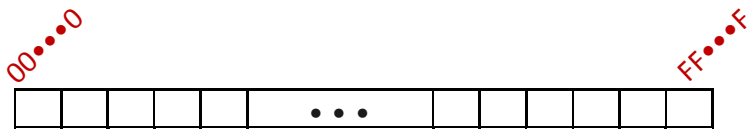
Hexadecimal Notation ("Hex")

- **Humans aren't very good at reading binary strings**
- **A better / more compact notation for a length 4 binary string is a hexadecimal (base 16) digit**
  - Example: $1100_2 \Leftrightarrow C_{16}$
- **Bit strings of length 4N can be written as N hex digits**
  - Example: $0010010011000000_2 \Leftrightarrow 24C0_{16}$
  - In C, you indicate hex by prefixing with '0x'
    - 0x24C0 or 0x24c0

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

Using Memory

- **To fetch a data item from memory, the CPU must specify:**
  - An address: where do the start?
  - A length: how long is the item?
- **There is a minimum length that can be fetched: 8 bits**
  - A byte is an string of 8 consecutive bits
- **Addresses refer to a byte offset, not a bit offset**
  - Address 4 means fetch starting at byte 4 in memory (not bit 4)
- **The amount fetched is restricted to some small number of bytes**
  - Typically: 1, 2, 4, or 8 bytes

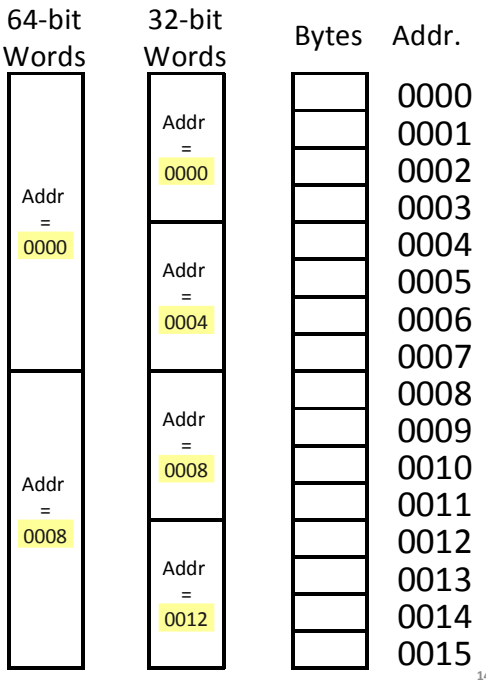


Machine Words

- **CPUs have a "word size"**
  - This is the width of the registers (measured in bits)
    - Nominal size of an integer
    - Also the number of bits in an addresses
- **All reasonably recent Intel/AMD processors are 64-bit hardware**
  - However, some software (including operating systems) written for 32-bit processors won't work when the word size is 64-bits
    - We'll see a bit about why in a moment
    - The processors can run in 32-bit mode...
- **The main benefit of larger word size is bigger addresses → larger physical memories**
  - With a 32 bit word/register, the largest possible address is 4G
    - Limits memory to 4GB
  - With 64 bit word/register, the potential address space ≈ 1.8 X 10^{19} bytes
    - x86-64 hardware supports 48-bit addresses: 256 TB

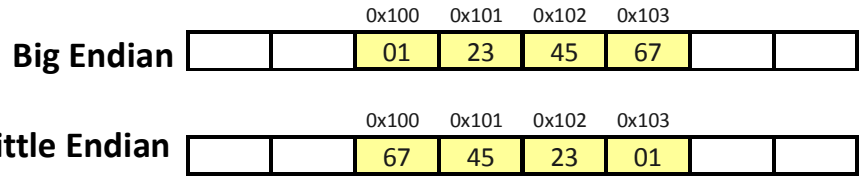
Word-Oriented Memory Organization

- **Addresses always specify <u>locations of bytes</u> in memory**
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)
- **It's possible to fetch less than a word, or perhaps more than one word**
  - E.g., a byte
  - E.g., a double-word



Byte Ordering

- **How should bytes within multi-byte word be ordered in memory?**
- **Conventions**
  - Big-endian: Sun, PPC Mac, Internet
    - Least significant byte has highest address
  - Little-endian: x86
    - Least significant byte has lowest address
- **Example**
  - Variable has 4-byte representation `0x01234567`
  - Address of variable is `0x100`



Data Representations

Data Types / Sizes (in bytes)

Java Data Type	C Data Type	Typical 32-bit	x86-64
boolean		1	1
byte	unsigned char	1	1
char	char	1	1
short	Short int	2	2
int	int	4	4
float	float	4	4
	long int	4	8
double	double	8	8
long	long long	8	8

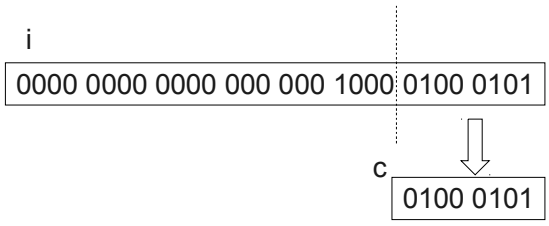
Example C program

```
int main(int argc, char* argv[]) {
    int i;      // give me 4 bytes of memory, call it 'i'
    char c; // give me 1 byte of memory, call it 'c'
    float f;    // give me 4 bytes of memory, call it 'f'

    // type checking happens in the compiler, not the hardware.
    // C is very "generous" about type conversions

    f = i;      // okay, just like in Java
    i = f;      // sort of okay in both C and Java

    i = c;      // totally okay in C; not okay in java
                // (means set i to the bit string formed
                // by appending the 8-bits of c to 24
                // leading bits of zeroes

    c = i;      // Also okay in C!
                // Set the 8 bits of c to the low order
                // 8 bits of i

    i = 'A';    // Also okay in C...
}
```


"Live" Example

```
#include <stdio.h>

int main() {
  int i = 256*8 + 'A';
  printf("i = %d\n", i);

  char c = i;
  printf("c = %c\n", c);

  return 0;
}
```

```
$ ./a.out
i = 2113
c = A
```



Boolean Operations on Bits

- **All data is bits (no matter what types have been declared)**
- **Sometimes it's useful to operate on bits, using boolean operators:**
  - Think of a 1 bit as true, and a 0 as false
  - AND: A&B = 1 when both A is 1 and B is 1
  - OR: A|B = 1 when either A is 1 or B is 1
  - XOR: A^B = 1 when either A is 1 or B is 1, but not both
  - NOT: ~A = 1 when A is 0 and vice-versa
  - DeMorgan's Law: ~(A | B) = ~A & ~B
    ~(A&B) = ~A | ~B

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

~	
0	1
1	0

General Boolean Algebras

- **Operate 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**

```
  01010101
^ 01010101
  --------
  00000000
```

Representing & Manipulating Sets

- **Representation**
  - Width *w* bit vector represents subsets of {0, …, *w*–1}
  - $a_j = 1$ if $j \in A$

    01101001 { 0, 3, 5, 6 }

    76543210

    01010101 { 0, 2, 4, 6 }

    76543210
- **Operations**

&	Intersection	01000001	{ 0, 6 }
|	Union	01111101	{ 0, 2, 3, 4, 5, 6 }
^	Symmetric difference	00111100	{ 2, 3, 4, 5 }
~	Complement	10101010	{ 1, 3, 5, 7 }

Bit-Level Operations in C

- **Operations &, |, ^, ~ are 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`

    $\sim 01000001_2 \dashrightarrow 10111110_2$
  - `~0x00 --> 0xFF`

    $\sim 00000000_2 \dashrightarrow 11111111_2$
  - `0x69 & 0x55 --> 0x41`

    $01101001_2 \;\&\; 01010101_2 \dashrightarrow 01000001_2$
  - `0x69 | 0x55 --> 0x7D`

    $01101001_2 \;|\; 01010101_2 \dashrightarrow 01111101_2$

Contrast: Logic Operations in C

- **Contrast to logical operators**
  - `&&, ||, !`
    - View 0 as "False"
    - Anything nonzero as "True"
    - Always return 0 or 1
    - Early termination (aka short-circuit evaluation)
- **Examples (char data type)**

```
!0x41   -->  0x00
!0x00   -->  0x01
!!0x41  -->  0x01

0x69 && 0x55   -->  0x01
0x69 || 0x55   -->  0x01
p && *p++      (avoids null pointer access, null pointer = 0x00000000 )
```


Other (non-Boolean) Bit Operations: Shifting

- **The bits in a word can be shifted**
  - When shifting left, zero bits are shifted in from the right, bits "shifted off" the left end are lost
    - 01011101_2 << 2 is 01110100_2
  - When shifting right, there are two possibilities:
    - Logical shift: shift in zeros from the left
    - Arithmetic shift: repeat the high-order bit (We'll see why later)
      - C uses arithmetic right shift
    - Examples:
    - 01011101_2 >> 2 is 00010111_2 *(for both arithmetic and logical)*
      11011010_2 >> 2 is 00110110_2 *(logical)*
      11110110_2 *(arithmetic)*

Using Shifts and Masks

- **Extract 2nd most significant byte of a 32-bit integer**
  - First shift: x >> (2 * 8)
  - Then mask: (x >> 16) & 0xFF

x	01100001 01100010 01100011 01100100
x >> 16	00000000 00000000 01100001 01100010
(x >> 16) & 0xFF	*00000000 00000000 00000000 11111111* 00000000 00000000 00000000 01100010

HW1 Sample Question

```
/*
 * replaceByte(x,n,c) - Replace byte n in x with c
 *   Bytes numbered from 0 (LSB) to 3 (MSB)
 *   Examples: replaceByte(0x12345678,1,0xab) = 0x1234ab78
 *   You can assume 0 <= n <= 3 and 0 <= c <= 255
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 10
 *   Rating: 3
 */
int replaceByte(int x, int n, int c) {
  return 2;
}
```

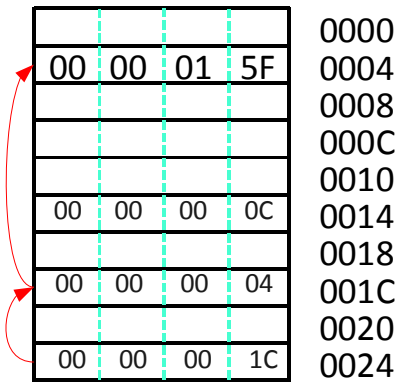
HW1 Sample Question Answer

```
/*
 * replaceByte(x,n,c) - Replace byte n in x with c
 *   Bytes numbered from 0 (LSB) to 3 (MSB)
 *   Examples: replaceByte(0x12345678,1,0xab) = 0x1234ab78
 *   You can assume 0 <= n <= 3 and 0 <= c <= 255
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 10
 *   Rating: 3
 */
int replaceByte(int x, int n, int c) {
  /* Mask out current byte value and OR in replacement */
  int n8 = n << 3;
  int mask = 0xff << n8;
  int cshift = c << n8;
  return (x & ~mask) | cshift;
}
```


Part III: Addresses / Pointers / C Arrays

A new data type: Addresses / Pointers

- An address names a location in memory
- A pointer is a data object that contains an address
- the value 351 (0x15F) is stored at address 0004
- Pointer to address 0004 stored at address 001C
- Pointer to a pointer at 0024
- The value 12 is stored at address 0014
  - Is it a pointer?



Why Have a Pointer Data Type?

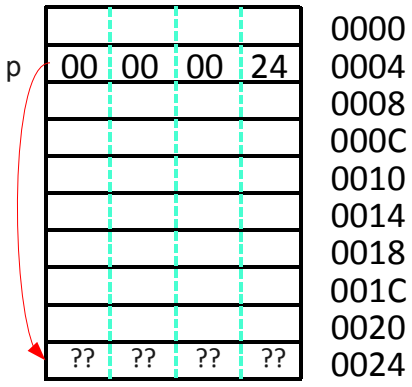
- When you say '`int x;`' in a program, you're both asking for 4-bytes of storage and giving those bytes a name: `x`
- Suppose you allocate memory at run time:

```
p = new int; // this isn't how you write this in C
```

  - What does 'p' name?
  - How do you name the storage just allocated?

C Pointers

- `int* p;`
  means "give me 4 bytes of storage; call it p. I'm going to use it to hold the address of int's (i.e., as an address of a 4-byte integer)"
- `p = new int;  // not how you write this in C!`
  allocates 4 bytes when executed, and assigns the address of those 4 bytes to p.
- The 4 bytes for p were reserved at compile time
- The 4 bytes for the new int is found at run time
- Note: the new space isn't initialized
- Note: if you now do 'p=0;',
  there is no name for the new int
  - "memory leak"

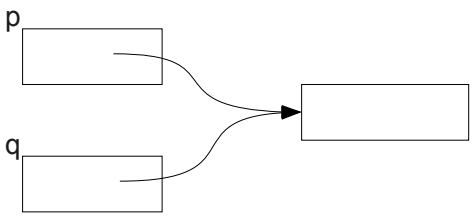


Using Pointers

- **The name 'p' means 'the 4-bytes allocated for the pointer'**
  - int i =p;
    would assign the pointer (address) to the integer i
- **The operator '*' de-references the pointer**
  - int i = *p;
    assigns the value pointed at by p to i
- **The operator '&' takes the address of something**
  - p = &i;
    assigns the address of the memory reserved for i to p
  - p = &*p; *// This isn't useful. It's just an example.*
    takes the address of the memory pointed at by p and assigns it to p – i.e., assigns p to p.

Pointer Assignment

- **int* p;**
  int* q;
  p = new int; // not actual C...
  q = p;



- **p and q are aliases for the new'ed memory**

Arrays

¢ **Arrays represent adjacent locations in memory storing the same type of data object**
Example: int big_array[128];
_ allocates 128*4 = 512 adjacent bytes in memory (e.g., starting at 0x00ff0000)

¢ **You can't point to an array, only to an element, but... consecutive elements are in contiguous memory**

```
int * p;
p = &big_array[0];      0x00ff0000
p = big_array;          0x00ff0000
p = &big_array[3];      0x00ff000c
p = big_array + 3;      0x00ff000c (adds 3 * size of int)
```

¢ **[] is an operator**

```
p = big_array;
p[3] = 4;     // same as big_array[3] = 4;
```

¢ **Array names are like pointers.**
Pointers are just addresses.
→ There is no array bound checking
→ In fact, there's no general way to determine the length of an array!

big_array[130] = 1;
is legal, executes, but has undetermined result

Representing Strings

char S[6] = "12345";

- **Strings in C**
  - Represented by array of characters
  - Each character encoded in ASCII format
  - Standard 7-bit encoding of character set
    - Fits into 8 bits with a leading 0`
  - Character "0" has code `0x30`
    - Digit *i* has code `0x30`+ *i*
  - String must be null-terminated
    - Final character = 0x00

S
31
32
33
34
35
00

- **Unicode characters – up to 4 bytes/character**
  - ASCII codes still work (leading 0 bit) but can support the many characters in all languages in the world
  - Java and C have libraries for Unicode (Java commonly uses 2 bytes/char)

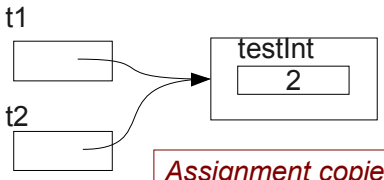
Pointers and Java

```
class test {

  public int testInt = 0;

  public static void main(String args[]) {
    int x =0;
    int y;

    test t1 = new test();
    test t2;

    y = x;
    x = 2;
    System.out.println("y = " + y);

    t2 = t1;
    t1.testInt = 2;
    System.out.println("t2.testInt = " + t2.testInt);
  }
}
```

- What does this print?
- Why?

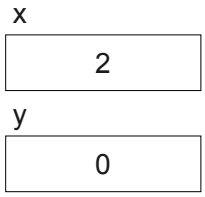
Java References

- **In Java, "almost all" variables are in fact pointers**
  - Java terminology is reference
- **Assignment is pointer assignment**
  - You're simply creating an alias



Assignment copies the reference.

- **On the other hand, for efficiency reasons, variables of primitive types are not references**



Assignment copies the value.

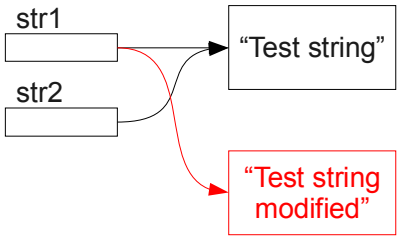
Java Strings

```
class testStr {

  public static void main(String args[]) {
    String str1 = "Test string";
    String str2 = str1;

    str1 = str1.concat(" modified");

    System.out.println("str2 = " + str2);
    System.out.println("str1 = " + str1);

  }
}
```

- What does this print?
- Why?

Java Strings (Are Special)

```
class testStr {

  public static void main(String args[]) {
    String str1 = "Test string";
    String str2 = str1;

    str1 = str1.concat(" modified");

    System.out.println("str1 = " + str1);
    System.out.println("str2 = " + str2);

  }
}
```



- What does this print?
- Why?