More On Memory (mis)-Management

Berkeley EECS
ELECTRICAL ENGINEERING & COMPUTER SCIENCES

Reminder on Strings...

- Reminder: Strings are just like any other C array...
  - You have a pointer to the start and no way of knowing the length
  - But you have an in-band "end of string" signal with the '\0' (0-byte) character
- Since you can have multiple pointers point to the same thing...
  - `char *a, *b;` ...
    `a = b;` ...
    `b[4] = 'x';` /* This will update a as well, since they are pointing to the same thing */
- So how do you copy a string?
  - Find the length (strlen), allocate a new array, and then call strcpy...
  - `a = malloc(sizeof(char) * (strlen(b) + 1) );`
    /* Forget the +1 at your own peril */
  - `strcpy(a, b)` or `strncpy(a, b, strlen(b) + 1);`
    - `strcpy` doesn't know the length of the destination, so it can be very unsafe
    - `strncpy` copies only n character for safety, but if its too short it ***will not copy the null terminator!***

Pointer Ninjitsu Reminder: Pointers to Functions

- You have a function definition
  - `char *foo(char *a, int b){ … }`
- Can create a pointer of that type…
  - `char *(*f)(char *, int);`
    - Declares f as a function taking a char * and an int and returning a char *
- Can assign to it
  - `f = &foo`
    - Create a reference to function foo
- And can then call it...
  - `printf("%s\n", (*f)("cat", 3))`
- Necessary if you want to write generic code in C:
  E.g. a hashtable that can handle pointers of any type

So You See This In The Project...

- You don't need to modify the hashtable...
  - It can actually store ***any*** pointer types since it uses (void *)
- Instead you just need to provide it a hash function and an equality comparison
  - You see this in **`main()`**: it gets the addresses of the hash function and the equality function
- If you wanted to store different types of data, you could
  - But you notice there is no enforced correctness, so its easy to ***misuse*** and cause things to have a problem
  - And you can only store ***pointers*** to data, not data copied by value:
    You can cast any pointer to **`void *`**, but you can't reliably cast anything else to **`void *`**

Pointer Ninjitsu Reminder: Pointers to arrays of structures


- ```
  typedef struct foo_struct
      { int x;
        char *z;
        char y;} foo;
  ```
  - How big is a foo?
    - assume an aligned architecture, sizeof(int) == sizeof(void *) == 4:
    - 12... It needs to be padded
- Dynamically allocated a single element:
  - `foo *f = (foo *) malloc(sizeof(foo))`
- Dynamically allocate a 10 entry array of foos:
  - `foo *f = (foo *) malloc(sizeof(foo) * 10);`
  ```

Pointer Ninjitsu Continued: Accessing that array...

- Accessing the 5th element's string pointer:
  - `f[4].z = "fubar"`
    - Assigns the z pointer to point to the static string fubar
      - It is undefined behavior to then do `f[4].z[1] = 'X'`
      - If you want to modify the string pointed to by `z` you are going to have to do a string copy
- What does it look like "under the hood"?
  - The address written to in `f[4].z = "fubar"` is `(f + 4 * 12 + 4)`:
    - Note: This math is the 'under the hood' math: if you actually tried this in C it would not work right! But it is what the compiler produces in the assembly language
    - The 5th element of type `foo` is offset (4*12) from `f`
      - Since we want all elements in the array to have the same alignment this is why we had the padding
    - The field z is offset 4 from the start of a foo object

Managing the Heap

- Recall that C supports functions for heap management:
  - `malloc()` allocate a block of uninitialized memory
  - `calloc()` allocate a block of zeroed memory
  - `free()` free previously allocated block of memory
  - `realloc()` change size of previously allocated block
    - careful – it might move!

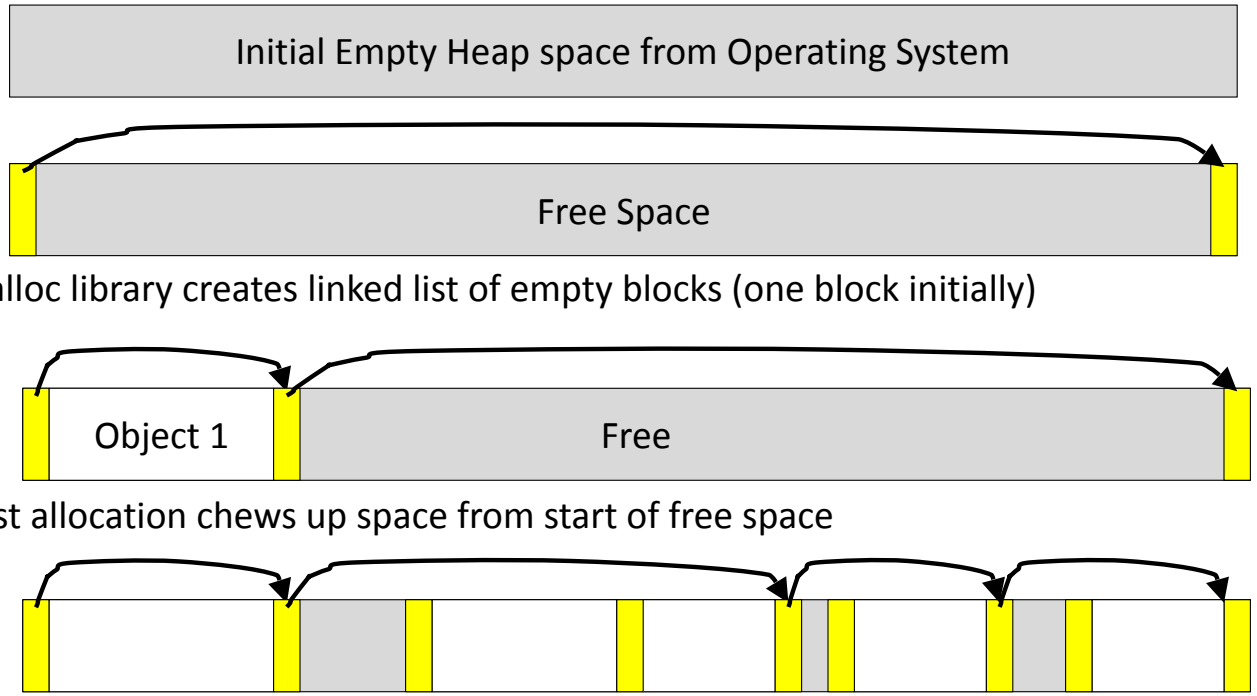
Observations

- Code, Static storage are easy: they never grow or shrink
- Stack space is relatively easy: stack frames are created and destroyed in last-in, first-out (LIFO) order
- ***Managing the heap is tricky***: memory can be allocated / deallocated at any time

How are Malloc/Free implemented?

- Underlying operating system allows malloc library to ask for large blocks of memory to use in heap (e.g., using Unix **`sbrk()`** call)
  - This is one reason why your C code, when compiled, is dependent on a particular operating system
- C standard malloc library creates data structure inside unused portions to track free space
  - This class is about how computers work:
    How they allocate memory is a huge component

Simple Slow Malloc Implementation



After many mallocs and frees, have potentially long linked list of odd-sized blocks
Frees link block back onto linked list – might merge with neighboring free space

Faster malloc implementations

- Keep separate pools of blocks for different sized objects
- "Buddy allocators" always round up to power-of-2 sized chunks to simplify finding correct size and merging neighboring blocks:
  - Then can just use a simple bitmap to know what is free or occupied

Power-of-2 “Buddy Allocator”

Step	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K
1	2^4															
2.1	2^3								2^3							
2.2	2^2				2^2				2^3							
2.3	2^1		2^1		2^2				2^3							
2.4	2^0	2^0	2^1		2^2				2^3							
2.5	A: 2^0	2^0	2^1		2^2				2^3							
3	A: 2^0	2^0	B: 2^1		2^2				2^3							
4	A: 2^0	C: 2^0	B: 2^1		2^2				2^3							
5.1	A: 2^0	C: 2^0	B: 2^1		2^1		2^1		2^3							
5.2	A: 2^0	C: 2^0	B: 2^1		D: 2^1		2^1		2^3							
6	A: 2^0	C: 2^0	2^1		D: 2^1		2^1		2^3							
7.1	A: 2^0	C: 2^0	2^1		2^1		2^1		2^3							
7.2	A: 2^0	C: 2^0	2^1		2^2				2^3							
8	2^0	C: 2^0	2^1		2^2				2^3							
9.1	2^0	2^0	2^1		2^2				2^3							
9.2	2^1		2^1		2^2				2^3							
9.3	2^2				2^2				2^3							
9.4	2^3								2^3							
9.5	2^4															

Malloc Implementations

- All provide the same library interface, but can have radically different implementations
- Uses headers at start of allocated blocks and/or space in unallocated memory to hold **`malloc`**'s internal data structures
- Rely on programmer remembering to **`free`** with same pointer returned by **`malloc`**
  - Alternative is a "conservative garbage collector"
- Rely on programmer ***not messing with internal data structures accidentally***!
  - If you get a crash in **`malloc`**, it means that somewhere else you wrote off the end of an array

Conservative Mark/Sweep Garbage Collectors

- An alternative to **`malloc`** & **`free`**...
  - **`malloc`** works normally, but **`free`** just does nothing
- Instead, it starts with the stack & global variables as the "live" memory
  - But it doesn't know if those variables are pointers, integers, or whatevers...
- So assume that every piece of memory in the starting set is a pointer...
  - If it points to something that was allocated by malloc, that entire allocation is now considered live, and "mark it" as live
  - Iterate until there is no more newly discovered live memory
- Now any block of memory that isn't can be deallocated ("sweep")

The Problems: Fragmentation & Pauses...

- A conservative garbage collector can't move memory around
  - So it gets increasingly fragmented...
    When we get to both caches and virtual memory we will see how this causes problems
- A conservative collector needs to ***stop the program!***
  - What would happen if things changed underneath it? Ruh Roh...
  - So the system needs to pause
- Java and Python don't have this problem
  - Java is designed to understand garbage collection:
    Able to have ***incremental*** collectors that don't require a long halt
  - Python doesn't do real garbage collection:
    Just uses "reference counting". Every python object has a counter for the number of pointers pointing to it. When it gets to 0, free the object

Clicker Question

- What will the following print:

```
int a, b, c, *d;
a = 0;
b = 1;
c = 2;
d = &a;
(*d) += b + c;
d = &b;
(*d) += a + b + c;
printf("a=%i b=%i\n", a, b);
```

- A) a=0, b=3
- B) a=3, b=3
- C) a=3, b=4
- D) a=3, b=7
- E) I ditched class today and had a friend "borrow" my clicker

Administrivia

- DSP students:
  - Make ***sure*** you contact Peijie as well as get your accommodation letters in
- Exam conflicts:
  - Make ***sure*** you have filled out the exam conflict form on Piazza
- Labs/Discussions:
  - If your lab/discussion is extra full, feel free to switch to an emptier one (e.g. the new ones)
  - In lab, if the lab is full, priority is given to the students enrolled in that section for checkoffs
- Enrollment:
  - Are there still problems with CalCentral?
    If so, don't worry, we ***will*** let everyone into the class
- Influenza:
  - Get a flu shot...
  - If you are sick (especially with a fever, muscle aches, etc), please stay home and don't spread the plague...

Common Memory Problems: aka Common "Anti-patterns"

- Using uninitialized values
  - Especially bad to use uninitialized pointers
- Using memory that you don't own
  - Deallocated stack or heap variable
  - Out-of-bounds reference to stack or heap array
  - Using NULL or garbage data as a pointer
  - Writing to static strings
- Improper use of **`free`**/**`realloc`** by messing with the pointer handle returned by **`malloc`**/**`calloc`**
- Memory leaks (you allocated something you forgot to later free)
- Valgrind is designed to catch ***most*** of these
  - It runs the program extra-super-duper-slow in order to add checks for these problems that C doesn't otherwise do

Using Memory You Don't Own

- What is wrong with this code?

```
int *ipr, *ipw;
void ReadMem() {
 int i, j;
 ipr = (int *)
    malloc(4 *
       sizeof(int));
 i = *(ipr - 1000);
 j = *(ipr + 1000);
free(ipr);}
```

Out of bounds reads

```
void WriteMem() {
   ipw = (int *)
      malloc(5 *
         sizeof(int));
   *(ipw - 1000) = 0;
   *(ipw + 1000) = 0;
   free(ipw);
}
```

Out of bounds writes

Faulty Heap Management

- What is wrong with this code?
-

```
int *pi;

void foo() {
  pi = malloc(8*sizeof(int));
  …
  free(pi);
}

void main(){
  pi = malloc(4*sizeof(int));
  foo();
  … }
```

The first `malloc` of pi leaks

Reflection on Memory Leaks

- Memory leaks are not a problem ***if your program terminates quickly***
  - Memory leaks become a much bigger problem when your program keeps running
  - Or when you are running on a small embedded system
- Three solutions:
  - Be very diligent about making sure you **`free`** all memory
    - Use a tool that helps you find leaked memory
    - Perhaps implement your own reference counter
  - Use a "Conservative Garbage Collector" **malloc**
  - Just quit and restart your program a lot ("burn down the frat-house")
    - Design your server to crash!
      But memory leaks will ***slow down your program*** long before it actually crashes

So Why Do Memory Leaks Slow Things Down?

- Remember at the start we saw that pyramid of memory?
  - Small & fast -> cache
    Big & slow -> main memory
- Memory leaks lead to ***fragmentation***
  - As a consequence you use more memory, and its more scattered around
- Computers are designed to access ***contiguous*** memory
  - So things that cause your working memory to be spread out more and in smaller pieces slow things down
- There also may be nonlinearities:
  - Fine... Fine... Fine... Hit-A-Brick-Wall!

Faulty Heap Management

- What is wrong with this code?

```
int *plk = NULL;
void genPLK() {
   plk = malloc(2 * sizeof(int));
   … … …
   plk++;
}
```

This MAY be a memory leak if we don't keep somewhere else a copy of the original malloc'ed pointer

Faulty Heap Management

- How many things are wrong with this code?

- ```
  void FreeMemX() {
    int fnh[3] = 0;
    ...
    free(fnh);
  }
  ```

Can't `free` memory allocated on the stack

- ```
  void FreeMemY() {
    int *fum = malloc(4 * sizeof(int));
    free(fum+1);
    ...
    free(fum);
    ...
    free(fum);
  }
  ```

Can't `free` memory that isn't the pointer from `malloc`

Can't `free` memory twice

Using Memory You Haven't Allocated

- What is wrong with this code?

```
void StringManipulate() {
  const char *name = "Safety Critical";
  char *str = malloc(10);
  strncpy(str, name, 10);
  str[10] = '\0';
  printf("%s\n", str);
}
```

`sizeof(char)` is 1 but should have `sizeof` as a good habit

Write off of the end of the array!

Using Memory You Don't Own

- What's wrong with this code?

```
char *append(const char* s1, const char *s2) {
   const int MAXSIZE = 128;
   char result[128];
   int i=0, j=0;
   for (j=0; i<MAXSIZE-1 && j<strlen(s1); i++,j++) {
    result[i] = s1[j];
   }
   for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++,j++) {
    result[i] = s2[j];
   }
   result[++i] = '\0';
   return result;
}
```

Returning a pointer to stack-allocated memory!

Using Memory You Don't Own

- What is wrong with this code?

```
typedef struct node {
    struct node* next;
    int val;
} Node;

int findLastNodeValue(Node* head) {
    while (head->next != NULL) {
       head = head->next;
    }
    return head->val;
}
```

What if head is null?
Always check arguments.
Your code may be good...
But you make mistakes!
PROGRAM DEFENSIVELY

Using Memory You Don't Own

- What is wrong with this code?

```
void muckString(char *str) {
    str[0] = 'b';
}
void main(void) {
    char *str = "abc";
    doReverse(str);
    puts(str);
}
```

Pointing to a static string...
Ruh Roh...

So Why Was That A Problem...

- When the compiler sees
  - **`char *foo = "abc"`**
  - The compiler interprets it as 'have the constant string "abc" somewhere in static memory, and have foo point to this'
    - If you have the same string "abc" elsewhere, it will point to the same thing...
      If you are lucky, the compiler makes sure that these string constants are set so you can't write
      - "Access violation", "bus error", "segfault"
- There is something safe however...
  - **`char foo[] = "abc"`**
  - The compiler interprets this as 'create a 4 character array on the stack, and initialize it to "abc"'
  - But of course we can't now say **`return foo;`**
    - Because that would be returning a pointer to something on the stack...

Managing the Heap: `realloc(p, size)`

- Resize a previously allocated block at **`p`** to a new size
- If **`p`** is NULL, then **`realloc`** behaves like **`malloc`**
- If size is 0, then **`realloc`** behaves like **`free`**, deallocating the block from the heap
- Returns new address of the memory block; NOTE: it is likely to have moved!
- ```
  int *ip;
  ip = (int *) malloc(10*sizeof(int));
  /* always check for ip == NULL */
  … … …
  ip = (int *) realloc(ip,20*sizeof(int));
  /* always check NULL, contents of first 10 elements retained */
  … … …
  realloc(ip,0); /* identical to free(ip) */
  ```
  ```

Using Memory You Don't Own

- What is wrong with this code?

```
int* init_array(int *ptr, int new_size) {
   ptr = realloc(ptr, new_size*sizeof(int));
   memset(ptr, 0, new_size*sizeof(int));
   return ptr;
}

int* fill_fibonacci(int *fib, int size) {
   int i;
   init_array(fib, size);
   /* fib[0] = 0; */ fib[1] = 1;
   for (i=2; i<size; i++)
    fib[i] = fib[i-1] + fib[i-2];
   return fib;
}
```

Realloc might move the block!

Which means this hasn't updated *fib!

And Now A Bit of Security: Overflow Attacks


- ```
  struct UnitedFlyer{
      ...
      char lastname[16];
      char status[32];
      /* C will almost certainly lay this out in memory
         so they are adjacent */
      ...
  };
  ...
  void updateLastname(char *name, struct UnitedFlyer *f){
      strcpy(f->lastname, name);
  }
  ```
  ```

So what...

- Well, United has my status as:
  - `name = "Weaver", status = "normal-person: hated"`
- So what I need to do is get United to update my name!!!
  - So I provide United with my new name as:
    `Weaver super-elite: actually like`
  - `name = "Weaver super-elite: actually like",`
    `status = "super-elite: actually like"`
- And then update my name ***again*** back to just "Weaver"
  - `name = "Weaver", status = "super-elite: actually like"`
- Basic premise of a ***buffer overflow*** attack:
  - An input that overwrites past the end of the buffer and leaves the resulting memory in a state suitable to the attacker's goals

Hijacking Control Flow With Buffer Overflows...

- Reminder: The stack stores a lot of stuff...
  - Not just local variables, but where to return execution to when a function returns
- So if you write some code with a stack-allocated array
  - `char foo[32];`...
    `gets(foo);` /* gets reads input into the string until a newline */
- A bad dude gets to provide your program a string (e.g. because it is a server)
  - His string is significantly longer than 32 characters...
- Result:
  - His string overwrites other local variables, including where the function is supposed to return!
  - And now your program is ***his*** program!

And In Conclusion, …

- C has three main memory segments in which to allocate data:
  - Static Data: Variables outside functions
  - Stack: Variables local to function
  - Heap: Objects explicitly malloc-ed/free-d.
- Heap data is biggest source of bugs in C code

Levels of Representation/Interpretation

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g., MIPS)

Anything can be represented as a *number*, i.e., data or instructions

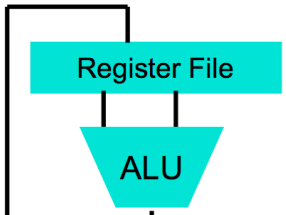
Assembler

Machine Language Program (MIPS)

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

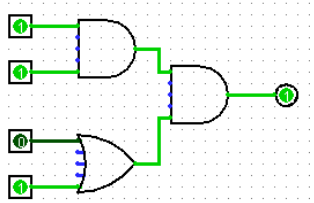
Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



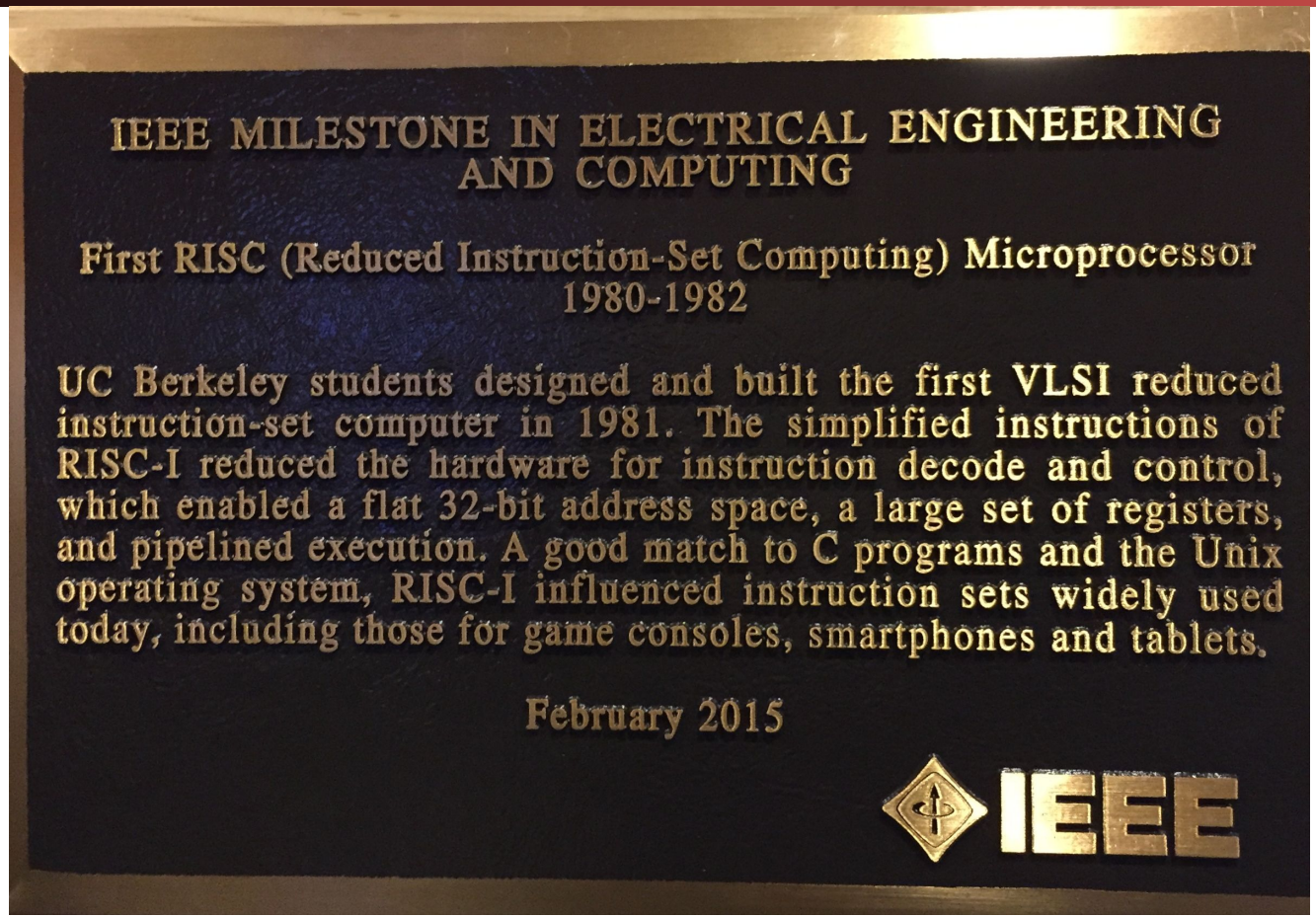
Assembly Language

- Basic job of a CPU: execute lots of ***instructions***.
- Instructions are the primitive operations that the CPU may execute
  - The stored program in "stored program computer"
- Different CPUs implement different sets of instructions. The set of instructions a particular CPU implements is an Instruction Set Architecture (ISA).
  - Examples: ARM, Intel x86, Intel x64 (64b x86), MIPS, RISC-V, IBM/Motorola PowerPC (old Mac), Motorola 68k (really old Mac), Intel IA64 (aka Itanic), ...

Instruction Set Architectures

- Early trend was to add more and more instructions to new CPUs to do elaborate operations
  - VAX architecture had an instruction to multiply polynomials!
- RISC philosophy (Cocke IBM, Patterson, Hennessy, 1980s) Reduced Instruction Set Computing
  - Keep the instruction set small and simple, makes it easier to build fast hardware
  - Let software do complicated operations by composing simpler ones

Berkeley Acknowledge for the first RISC computer



From RISC-I to RISC-V

The RISC-V Instruction Set Architecture

RISC-V (pronounced "risk-five") is a new instruction set architecture (ISA) that was originally designed to support computer architecture research and education, which we now hope will become a standard open architecture for industry implementations. RISC-V was originally developed in the Computer Science Division of the EECS Department at the University of California, Berkeley.

Pretty much everything in Berkeley uses RISC-V

But at the same time... All RISCs are mostly the same except for one or two silly design decisions

- MIPS:
  - "The Branch Delay Slot": The instruction immediately after an if is always executed
- SPARC:
  - "Register Windows"
- ARM:
  - 32b is ugly as sin...
  - But 64b is pretty much generic RISC with condition codes for instructions: Allow you to say "only do this instruction if a previous value was TRUE/FALSE"
- RISC-V:
  - You will hate how immediates are encoded until you understand how muxes work...

And x86s these days are really RISCs in disguise...

- They start by translating the x86/x64 CISC instructions into a series of "micro operations"
  - These instructions are then executed on something that looks very much like a RISC processor
- They just end up costing a fair bit more... Because they first have to turn x86/x64 into a RISC internal structure

So Why Do Some Architectures "Win"?

- The big winners: x86/x64 (desktop) and Arm (phones/ embedded)
  - Neither are the cheapest nor the best architectures available...
- They won because of the legacy software stack...
  - x86 had Windows and then Linux for servers and a history of optimizing for performance ***without breaking old things***.
    - For a decade+ you'd be able to just make everything run faster by throwing some money at the problem...
  - Arm became entrenched with Linux->Android in the phone market
- But since ***our*** focus is understanding how computers work, our software stack is RISC-V