CS61C: Exam1 Review

CS61C Fall2007 – Exam #1 Review
Greg Gibeling
Select Slides from Prof. Wawrzynek

Rules

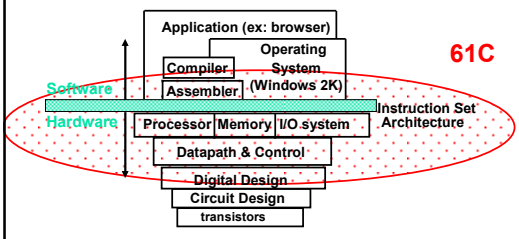
- Participate or Leave
  - Shut up. Dead serious. You talk, we all stop until you're done or you leave.
  - It's not or job to force understanding upon you
    - This is college, not gradeschool
- If you have a question, ask it NOW
  - You aren't the only one with that question
  - Even when you're wrong about something works its always an interesting idea
  - We cannot stress this enough
- Have a question to ask when I call on you
  - … or you have to answer all subsequent questions
  - What are you more afraid of? Appearing not to know everything, or having that proved?
- Speak up, interrupt, don't raise your hand
- I'm often wrong, I'm not being tested, you are, you have to be right

Format

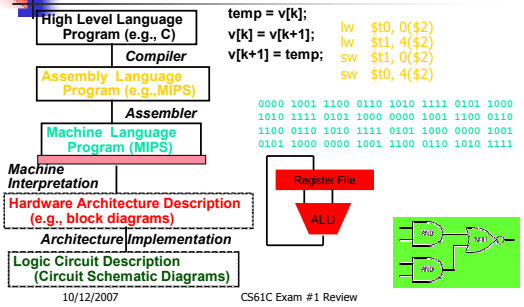
- 4:15-6:00 Lecture-ish
  - Highlight lecture slides from Prof. Wawrzynek
  - Discussion slides
  - If you want us to review a particular quiz/homework/proj you lost points on now is the time!
  - Before presenting an assignment
    - What's your first guess as to the most common problem with this assignment?
    - Who says this was an easy assignment?
  - What to discuss
    - What was the problem, from a detailed technical standpoint?
    - Why did the student make this mistake?
    - Our solution
    - Some wrong student solutions
  - Quizzes, Homeworks, Projects & Labs
- 6:15-7:15 Dinner+
  - Random questions
  - Started old midterm questions
- 7:15-8:00 Wrapup, Questions & Examples
  - Old midterm questions
  - General problem solving, we'll roam and provide help
  - What's still not clear?
  - Revisit any problem you like

Course Overview

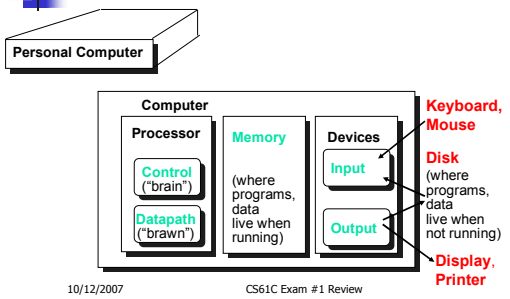
What are "Machine Structures"?



Levels of Abstraction



5 Main Components



Technology Trends

- Processor
  - 2X in speed every 1.5 years (since '85); 100X performance in last decade.
- Memory
  - DRAM capacity: 2x / 2 years (since '96); 64x size improvement in last decade.
- Disk
  - Capacity: 2X / 1 year (since '97)
  - 250X size in last decade.

Stored Program Computer

- Binary Data
  - Numbers: integer & floating point
    - ±2Gi for integers, $2*10^{\pm 38}$ for float (single precision)
    - 4Gi values either way in 32b
  - Characters: ASCII & Unicode
    - An assignment from numbers to symbols
  - Pointers: an integer with meaning
  - Instructions: MIPS, IA32, SBN
    - Perform operations on data
- The meaning of data depends on it's interpretation
  - In C we specify a type for all data
  - In MIPS we don't know the type apriori
    - For example the format specifier tells us the type

Course/Review Outline

- Basics
  - C-Language
  - Pointers
  - Memory management
- Machine Representations
  - Numbers
  - Assembly Programming
  - Floating Point
  - Compilation, Assembly
- Processors & Hardware
  - Logic Circuit Design
  - CPU organization
  - Pipelining
- Memory Organization
  - Caches
  - Virtual Memory
- I/O
  - Interrupts
  - Disks, Networks
- Advanced Topics
  - Performance
  - Virtualization & Simulation
  - Parallel Programming
  - Parallel Architectures

C Overview

Compilation: Overview

- Java compiled to architecture independent "bytecode"
  - Java compiler is written in java/C
- C compiled to architecture specific machine code
  - Part1: compiling .c files to .o files
  - Part2: linking the .o files into executables
- Scheme is interpreted
  - Interpreter is written in C
- Machine Code is interpreted
  - Interpreter is in hardware!

Translation & Interpretation

- Examples
  - Compiler: Translates an HLL into an LLL
  - Assembler: Mechanical translation from assembly to machine code
  - Linker
    - A "translator"
    - Truthfully it doesn't translate anything
  - BASH/TCSH/CMD: Shell interpreters
  - CPU: A machine for interpreter implemented in hardware
- Translation
  - Turn code into a different kind of code
- Interpretation
  - Imperative languages: do what the commands say
    - All languages we've see to date are imperative
    - Programs are a list of commands
  - Declarative languages: ?
    - Verilog, the next language we'll learn is declarative
    - "Programs" are statements of existence, not commands

C vs. Java™ Overview

Java	C
Object-oriented (OOP)	No objects
"Methods"	"Functions"
Class libraries of data structures	C libraries are lower-level
Automatic memory management	Manual memory management
High memory overhead	Pointers
All variables initialized	No overhead
	No variables initialized

C vs. Java™ Syntax

- C to Java: A rough transition
  - "." in Java becomes "->" in C
  - Both turn into `lw/sw` in MIPS
- Pointer vs Memory
  - A pointer is different than the memory it points to
    - E.g. strings are all in your imagination in C
    - Can someone tell me how to free a string?
  - You can store pointers in memory
    - Leads to complex `lw/sw` sequences, lab5ex1

All Kinds of Zeros

- Not my IQ
- Kinds of Zeros
  - `NULL` – for pointers
  - `0` – for integers
  - `0.0` – for floating point
  - `'\0'` – for characters
  - #define FALSE 0
    - Remember everything except 0 is TRUE
    - No boolean types
- Why
  - So that your code is readable
  - `NULL` might not always be zero!
  - **These constants specify value & type**

Common C Error

- There is a difference between assignment and equality
  - `a = b` is assignment
  - `a == b` is an equality test
- This is one of the most common errors for beginning C programmers!

C Syntax : flow control

- Conditionals
  - `if-else`
  - `switch`
- Loops
  - `while` and `for`
  - `do-while`
- All of these are branches in assembly

C Syntax: `main`

- `int main (int argc, char *argv[])`
- What does this mean?
  - `argc`: number of strings on command line
    - the executable counts as one
    - plus one for each argument
    - Example: `unix% sort myFile`
  - `argv` is an array of pointers to strings
    - Can also be written as a `char ** argv`

Assembly

Assembly Language

- Assembly is nearly what a CPU interprets
  - "Instructions" are the primitive operations
    - The set of instructions a particular CPU implements is part of the ***Instruction Set Architecture (ISA)***
    - Intel 80x86 (Pentium 4), IBM/Motorola PowerPC (Old Macintosh), MIPS, Intel IA64, ARM, ...
- Assembly language is a textual representation
  - Intermediate form in a C compiler
  - As a means to directly program the CPU
    - Why would anyone want to do this?

ISAs (1)

- CISC
  - Complex Instruction Set Computing
  - Useful for human assembly programmers
  - Lots of complex instructions
    - Ex: DEC VAX architecture had an instruction to evaluate polynomials!
- RISC (Cocke IBM, Patterson, 1980s)
  - Reduced Instruction Set Computing
  - Keep the instruction set small and simple
    - Makes it easier to build fast hardware.
    - Composing simpler instructions into more complex ones
  - Useful for compilers

ISAs (2)

- Problem: Design A Processor
  - How many and what kind of registers?
    - Native data width? MIPS is 32…
  - How many and what kind of instructions?
    - Will you have branches?
    - Encoding?
  - RISC or CISC?
  - Superscalar? Vector? Dataflow? OOO?
    - Take CS152!
  - What's the minimum ISA?
    - SBN!

ISAs (3)

- Single Instruction
  - sbn: subtract branch if negative
  - Build all operations up from this instruction
  - No need for registers, just use memory
  - "Universal Operator"
    - NAND, Mux, etc…
- Performance
  - CISC: Bad, RISC: Good, SBN: Very Bad
  - Why is the sweet spot in the middle at RISC?
  - So what?

MIPS Architecture

- MIPS
  - An ISA
    - Not a CPU!
  - Semiconductor company
    - Built one of the 1st commercial RISC CPUs
- Why MIPS instead of Intel 80x86?
  - MIPS is simple, elegant
  - CS152, CS162, CS164, Research!
  - MIPS still widely used in embedded appl

Storage in Assembly (1)

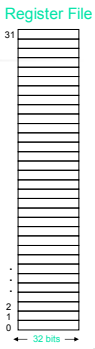
- C & Java: Variables
  - Stored where? Memory?
  - How big are they?
- Assembly: Registers
  - Special storage locations built directly into the CPU
    - Must be fast & cheap to build
    - Limited in number (32 for MIPS)
    - Must be used efficiently
    - Keeps machine code small
  - Small, fast & simple
    - All the same size (32bits for MIPS)
      - Width of registers generally the same as ISA word size
    - Thousands of times faster than main memory (< 1ns)

Storage in Assembly (2)

- C & Java have variables
  - Logical or virtual storage
  - Can be stored in memory or a register
  - Variable is just a name for a value
- Assembly has locations
  - Physical storage
  - Locations can be named (registers & labels)
  - The value in a location can change, the location cannot
- Register Coloring
  - Deciding which variables go in which locations
  - Similar to 4-color theorem. See CS164

		Meaning	
		Known	Unknown
Width	Known	Java	Assembly
	Unknown	C	

Storage in Assembly (2)



- MIPS
  - Registers are numbered from $0 to $31
  - Also named: $zero, $at …
    - `$16 - $23` → `$s0 - $s7` (C Variables)
    - `$8 - $15` → `$t0 - $t7` (Temporaries)
- 0 is a common constant
  - Register zero (`$0` or `$zero`) = 0
  - `$0` is immutable

Assembly Instructions (1)

- Instructions
  - An imperative (command) statement
  - executes exactly one of a short list of simple commands
    - One of (=, +, -, *, /)
  - One per line (no need for ;)
- Syntax of General Instructions:

  1 2,3,4

  1) name of operation

  2) operand getting result ("destination")

  3) First operand for operation ("source1")

  4) Second operand for operation ("source2"), register or immediate
  - Syntax is rigid
    - 1 operator, 3 operands (for MIPS)
    - Why?

Assembly Instructions (2)

- Simple Examples

```
add     $s0,$s1,$s2 # a = b + c
```

  - a stored in `$s0`, b in `$s1`, c in `$s2`

```
sub     $s3,$s4,$s5 # d = e - f
```

  - d stored in `$s3`, e in `$s4`, f in `$s5`
- Composition (Compilation)

```
# a = b + c + d - e;
add $t0, $s1, $s2 # temp = b + c
add $t0, $t0, $s3 # temp = temp + d
sub $s0, $t0, $s4 # a = temp - e
```

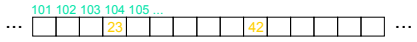
Assembly Instructions (3)

- Immediates are numerical constants
  - Very common
  - How else do we get meaningful values?
  - Add Immediate:
    - `addi $s0,$s1,10 # f = g + 10`
      - `f` stored in `$s0`, `g` in `$s1`
    - Similar to `add` instruction
    - No subi in MIPS. Why? RISC!

Memory Exposed

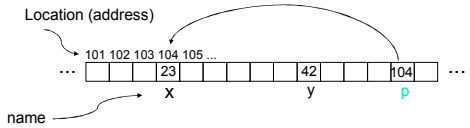
Address vs. Value

- Consider memory to be a single huge array:
  - Each cell of the array has an address associated with it.
  - Each cell also stores some value
  - Do you think they use signed or unsigned numbers? Negative address?!
- Don't confuse the address referring to a memory location with the value stored in that location.



Pointers (1)

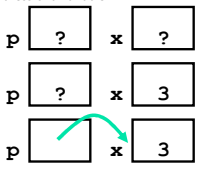
- An address refers to a particular memory location. In other words, it points to a memory location.
- Pointer: A variable that contains an address



Pointers (2)

- How to create a pointer:
  - `&` operator: get address of a variable

```
int *p, x;
  x = 3;
  p =&x;
```

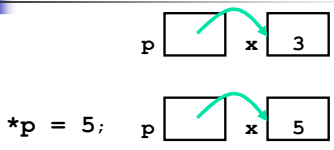


- How get a value pointed to?
  - `*` "dereference operator": get value pointed to

```
printf("p points to %d\n",*p);
```

Note the "*" gets used 2 different ways in this example. In the declaration to indicate that `p` is going to be a pointer, and in the **`printf`** to get the value pointed to by `p`.

Pointers (3)



- How to change a variable pointed to?
  - Use dereference `*` operator on left of `=`

Pointers (4)

- Pointer Types
  - Normally a pointer can only point to one type
    - Can point to different instances of that type!
    - `void *` is a type that can point to anything
  - Function Pointers: Remember JALR?
- Advantages
  - Cheaper than passing a big array or struct
    - What if your struct is >2GB on a 32bit machine?
  - Cleaner, more compact code (K&R allocator)
- Disadvantes
  - Single largest source of bugs in software
    - Segfault & bus errors
    - Security holes
  - Declaring a pointer just allocates space to hold the pointer – it does not allocate something to be pointed to!
  - Local variables in C are not initialized, they may contain anything.

Assembly Labels

- Example:
  `N: .word 0`
  - N: A constant number, the address of the word
  - .word: Allocate a word's worth of storage
  - 0: fill the allocated storage with 0
- Basic Use
  ```
  lw $t0, 25($t1) # $t0 = *($t1 + 25)
  sw $t0, 25($t1) # *($t1 + 25) = $t0
  ```
- Advanced Uses
  `sw $s0, 10+N($a1)`
  - 10+N is a constant computed by the assembler
  `sw $s0, N # sw $s0, N($0)`
  - Simple shorthand (useable only because of $0!)
- PsuedoInstructions
  ```
  la $t0, N # lui + ori as needed
  li $t0, N # should be the same as la…
  ```


Pointers and Parameters (1)

- Java and C pass primitives "by value"
  - Procedure/function gets a copy of the parameter
  - Changing the copy cannot change the original

```
void addOne (int x) {
     x = x + 1;
}
int y = 3;
addOne(y); // y is still == 3
```

  - How would we "fix" this?
- Java passes objects "by reference"
  - *Autoboxing/Unboxing in Java 1.5 complicates this*
- C++, VB, etc… can pass arguments byref or byval

Arrays (1)

- Declaration
  - `int ar[2];`
    - declares a 2-element integer array
    - An array is just a block of memory.
  - `int ar[] = {795, 635};`
    - Declares & initializes a 2-element integer array
    - Where do the initial values come from?
- Declared arrays are only allocated while the scope is valid
  - Lexical scoping (remember CS61A?)
- Accessing elements
  - `ar[num];`
  - returns the `num`th element.

```
char *foo() {
   char string[32]; ...;
   return string;
} // hopefully segfault
```


Arrays (2)

- Arrays are (almost) identical to pointers
  - `char *string` and `char string[]`
  - Nearly identical declarations
  - An array variable is an *immutable* "pointer" to the first element.
- Consequences: `int ar[10];`
  - `ar` is an array variable but looks like a pointer in many respects (though not all)
  - `ar[0]` is the same as `*ar`
  - `ar[2]` is the same as `*(ar+2)`
  - We can use pointer arithmetic
- Differences
  - See right ->

```
void one() {
    char *x = "foo", *y = "foo";
    x[0] = 'm';
}
void two() {
    char x[] = "foo", y[] = "foo";
    x[0] = 'm';
}
```


Arrays (3)

- Array size n, want to access [0 to n-1]
  - Test against element after array…

```
int ar[10], *p = ar, *q = ar + 10, sum = 0;
while (p != q) sum += *p++;
```

    - Is this legal?
  - Wrong style
    `int i, ar[10]; for(i = 0; i < 10; i++) { ... }`
  - Right style

```
#define ARRAY_SIZE 10
int i, a[ARRAY_SIZE];
for(i = 0; i < ARRAY_SIZE; i++){ ... }
```

- Pitfall: An array in C does not know its own length, & bounds not checked!
  - Can accidentally access off the end
  - Must pass the array and its size together

Pointer Arithmetic

- A Pointer is a memory address
  - p+1 returns a ptr to the next array element

```
(*p)+1 vs *p++ vs *(p+1) vs (*p)++ ?
x = *p++          x = *p ; p =  p + 1;
x = (*p)++        x = *p ; *p = *p + 1;
```

- Array of Structs
  - P+1 adds size of array element, not 1 byte!
- Valid Arithmetic
  - Add an integer to a pointer (traverse an array)
  - Subtract 2 pointers (in the same array? anywhere?)
  - Compare pointers (<, <=, ==, !=, >, >=)
  - Compare pointer to `NULL`
    - Indicates that the pointer points to nothing by convention
    - Most ISAs/OSs will turn `*((void*)NULL)` into a segfault for you

C Strings (1)

- There are no string in C
- We approximate a string with an array

```
char string[] = "abc";
```

  - We know this is a string, C doesn't
  - ASCIIZ Format
    - Last character is followed by a '\0'
    - Length =# of characters (excluding '\0' @ end)
- Functions

```
int strlen(char *string);
int strcmp(char *str1, char *str2);
char *strcpy(char *dst, char *src);
```


C Strings (2)

- Buffer Overflows
  - Endless source of viruses & bugs
  - Will get you FIRED

```
void foo(char* string) {
  int length = strlen(string);
  char* buffer = (char*)malloc((length+1)*sizeof(char));
  strncpy(buffer, string, length);
  buffer[length] = '\0';
  // etc…
}
```


Virtual & Physical Storage

Virtual & Physical Storage

- C Variable Declaration/Storage Allocation
  - Virtual Storage
  - Arguments, Locals, Return Values
    - Allocated on the "stack"
    - Remember scoping rules

```
int i; struct Node list; char *string;
```

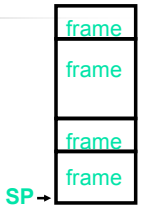
  - Global
    - Similar to above but outside of any block
    - Allocated in "static storage"

```
int myGlobal; main() {}
```

  - Dynamic (Later): Allocated on the "heap"
- Assembly
  - Physical Storage
  - Registers & Raw Memory
- Java?

The Stack

- Stack frame includes
  - Return "instruction" address
    - $ra saved here after jal
  - Parameters
  - Space for local variables
- In memory
  - Stack frames are contiguous
  - Stack pointer tells where top stack frame is
    - Stack grows downwards in memory
  - LIFO Data Structure (CS61B)
- Function Calls
  - On entry a function creates a stack frame
  - Stores all it's variables
  - On exit a function destroys its stack frame



The Heap

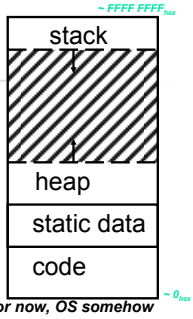
- Large pool of memory
  - Not allocated in contiguous order
  - Back-to-back requests could result in blocks very far apart
  - Where Java `new` command allocates memory
- In C, specify number of bytes of memory to allocate

```
struct int *iptr;
 iptr = (int *) malloc(8*sizeof(int));
 /* malloc returns type (void *),
 so need to cast to right type */
```

- `malloc()`: Allocates raw, uninitialized memory from heap

MIPS Sections

- A program's *address space* contains 4 regions:
  - Stack: local variables, grows downward
  - Heap: space requested for pointers via `malloc()`; resizes dynamically, grows upward
  - Static data: variables declared outside main, does not grow or shrink
  - code: loaded when program starts, does not change



For now, OS somehow prevents accesses between stack and heap (gray hash lines). Wait for virtual memory

Intel 80x86 Sections

- A C program's 80x86 *address space* :
  - heap: space requested for pointers via `malloc()`; resizes dynamically, grows upward
  - static data: variables declared outside main, does not grow or shrink
  - code: loaded when program starts, does not change
  - stack: local variables, grows downward

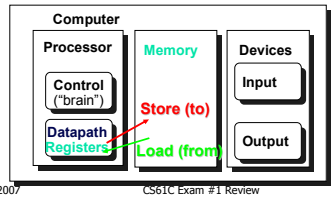
Managing Sections

- Code, Static storage are easy
  - Never change size
  - Filled from object file sections
- Stack space is also easy
  - Stack frames are created and destroyed in (LIFO) order
  - Just need to make sure we don't run out
- Managing the heap is tricky
  - Memory can be allocated/freed at any time
  - Size are unpredictable

Accessing Memory

5 Main Components

Registers are in the datapath of the processor; if operands are in memory, we must transfer them to the processor to operate on them, and then transfer back to memory when done.



Data Transfer (1)

- Move data to/from memory
- Arguments: Register & Memory Address
  - Register: number ($0 - $31) or name ($s0,..., $t0, ...)
  - Memory address: a pointer from a register
    - Maybe an offset as an immediate
    - Location is register contents + offset
    - `8($t0) = 8 + contents_of($t0)`
- Load Instruction Syntax:

  1 2,3(4)

  1) operation name (lw, sw, lbu, sb, etc...)
  2) register that will receive value
  3) numerical offset in bytes
  4) register containing pointer to memory

Data Transfer (2)

- Load Word
  - Example: `lw $t0,12($s0)`
  - Take the pointer in $s0, add 12 to it, and then load the value at that memory address into $t0
  - 12 is called the offset
    - Used in accessing elements of array or structure
    - Base register points to beginning of array or structure
    - Offset must be a constant
  - If you write: `lw $t2,0($t0)`
    - Then `$t0` better contain a pointer
    - Don't mix these up!
- Store Word
  - Store instruction syntax is identical to Load's
  - Example: `sw $t0,12($s0)`
  - This instruction will take the pointer in $s0, add 12 bytes to it, and then store the value from register $t0 into that memory address
- **A register can hold any 32-bit value**
  - That value can be a (signed) `int`, an `unsigned int`, a pointer (memory address), and so on
  - Stored program computing!

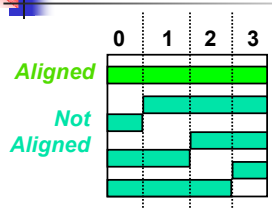
Addressing (1)

- Every word in memory has an address
  - Similar to an index in an array
    - Memory[0], Memory[1], Memory[2], ...
  - Needed to access 8-bit bytes and 32-bit words
  - Today machines address memory as bytes
    - "Byte Addressed"
    - Hence 32-bit (4 byte) word addresses differ by 4
    - Memory[0], Memory[4], Memory[8], ...
- Compile by hand using registers:

```
g = h + A[5];
```

  `g`: `$s1`, `h`: `$s2`, base address of `A`: `$s3`
  - What offset in `lw` to select `A[5]` in C? 4x5=20 to select `A[5]`: byte v. word
  - `lw $t0, 20($s3)`
    - `$t0` gets `A[5]`
    - Add `20` to `$s3` to select `A[5]`, put into `$t0`
  - `add $s1, $s2, $t0`
    - `$s1 = h+A[5]`
    - Next add it to `h` and place in `g`

Addressing (2)



Last hex digit of address is:

- *0, 4, 8, or C_{hex}*
- *1, 5, 9, or D_{hex}*
- *2, 6, A, or E_{hex}*
- *3, 7, B, or F_{hex}*

- MIPS requires that all words addresses are multiples of 4 bytes
  - Lowest 2 bits of PC are always 0!
- Called Alignment: objects fall on address that is multiple of their size.

Registers vs. Memory

- What if more variables than registers?
  - Compiler tries to keep most frequently used variable in registers
  - Less common variables in memory: spilling
- Why not keep all variables in memory?
  - Smaller is faster: registers are faster than memory
  - Registers more versatile:
    - MIPS arithmetic instructions can read 2, operate on them, and write 1 per instruction
    - MIPS data transfer only read or write 1 operand per instruction, and no operation
    - X86/IA32 is different. It was a bad idea.

Dynamic Memory

Dynamic Memory Alloc. (1)

- Allocating Memory in C
  - `malloc()`
  - Why isn't there a pointer involved?
- Example: `ptr = (int *) malloc (sizeof(int));`
  - `ptr` points to memory block of size `(sizeof(int))` bytes
  - `(int *)` tells the type of that block (called a typecast)
- General Use Rules
  - malloc is almost never used for 1 variable
  - `ptr = (int *) malloc (n*sizeof(int));`
    - This allocates an array of `n` integers.
  - `ptr = (struct stupid *) malloc(sizeof(struct stupid));`

Dynamic Memory Alloc. (2)

- Pointer Declaration
  - `int*ptr;`
  - ptr doesn't actually point to anything yet?
  - `ptr = (int*)malloc(sizeof(int));`
  - Can point to something that exists

```
int *ptr, var1, var2;
var1 = 5;
ptr  = &var1;
var2 = *ptr;
```

  - `var1` and `var2` have room implicitly allocated for them.

Dynamic Memory Alloc. (3)

- C has operator `sizeof()`
  - Gives size in bytes
  - Argument is type or type of variable
    - Difference from `strlen()`?
    - Assume size of objects can be misleading, so use sizeof(type)
  - Changing Type Sizes
    - `int` used to be 8b, 16b, 36b, 11b, etc
    - How big are pointers?
  - An operator, not a function!

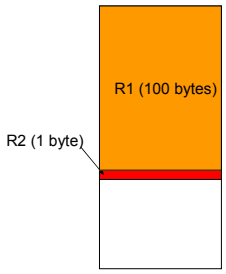
```
char foo[3*sizeof(int)]
char foo[3*myfunction(int)]
char foo[3*myfunction(7)]
```


Dynamic Memory Alloc. (4)

- Dynamic Memory Initialization
  - `malloc()`ed memory contains garbage
    - Causes crashes & security flaws!
    - `memset()` is a handy function
- Freeing memory
  - `free(ptr);`
  - What about freeing twice?
  - What about things not allocated by malloc?
- Requirements
  - Must run fast `malloc()` and `free()`
  - Minimal memory overhead
  - Avoid fragmentation (external & internal fragmentation)

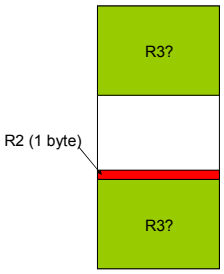
Heap Management

- An example
  - Request R1 for 100 bytes
  - Request R2 for 1 byte
  - Memory from R1 is freed
  - Request R3 for 50 bytes



Heap Management

- An example
  - Request R1 for 100 bytes
  - Request R2 for 1 byte
  - Memory from R1 is freed
  - Request R3 for 50 bytes



K&R Malloc/Free (1)

- Look at Section 8.7 of K&R
  - Code is terse/dense
  - Uses some advanced features (unions)
- Basic Data Structure
  - Block of Memory with Header
    - size of the block
    - pointer to the next block (unused in an allocated block)
    - All free blocks are kept in a linked list
- `malloc()`
  - Searches the free linked list for a block that is big enough.
  - If none is found, more memory is requested from OS
  - Otherwise it fails
- `free()`
  - Checks if the blocks adjacent to the freed block are also free
  - If so, adjacent free blocks are merged (coalesced)
  - Otherwise, the freed block is just added to the free list

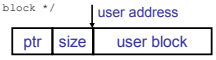
K&R Malloc/Free (2)

- Choosing among eligible (large enough) blocks
  - best-fit: choose the smallest block that is big enough
    - Tries to limit fragmentation but at the cost of time
    - Leaves lots of small blocks (why?)
  - first-fit: choose the first block we see that is big enough
    - Quicker than best-fit (why?) but potentially more fragmentation
    - Concentrate small blocks at the beginning of the free list
  - next-fit: like first-fit but remember where we finished searching and resume searching from there
    - Does not concentrate small blocks at front like first-fit, should be faster as a result.

K&R Malloc/Free (3)

- Each block is preceded by a header

```
typedef struct header {
  struct header * ptr; /* next free block */
  unsigned size;     /* size of this block */
} Header;
```



- Free list:
  - Circularly linked
  - Partially traversed by malloc and free
  - Block appear on list in increasing memory position
- Next fit algorithm for allocation
- New block taken from tail of next sufficiently large block
- Free merges blocks existing free block(s)
- Globals:

```
static Header base; /* empty list to get started */
static Header *freep = NULL;  /* start of free list */
```

- K&R uses a "union" type to force alignment
  - Complicates field extraction: `p->size` becomes `p->s.size`

K&R Malloc/Free (4)

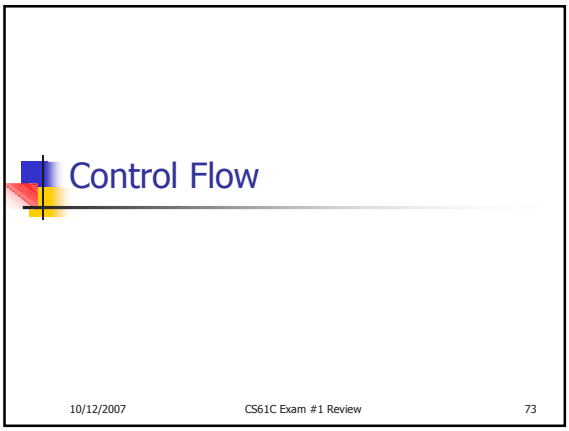
- IMS Design & Organization
- Run through K&R 8.7 Code

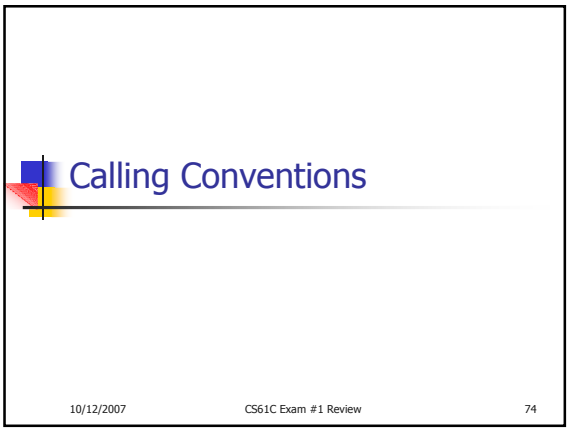
Garbage Collection

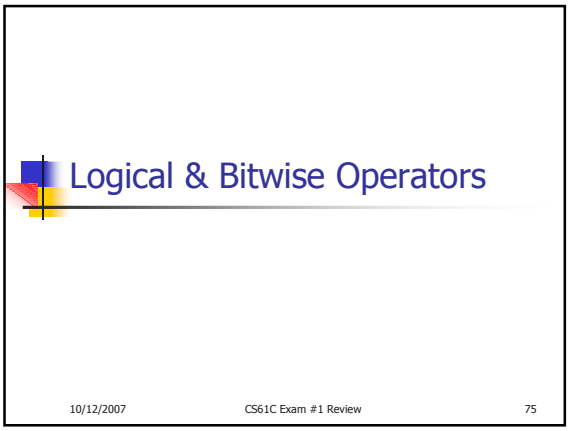
- Dynamically allocated memory is difficult
  - Why not track it automatically?
    - We need to know what's reachable
    - In other words what we have a pointer to
  - Unreachable memory is called *garbage*
    - Reclaiming it is called *garbage collection*
  - So how do we track what is in use?
    - Techniques depend on the language
    - Must rely on help from the compiler & OS
    - Is this possible in C?

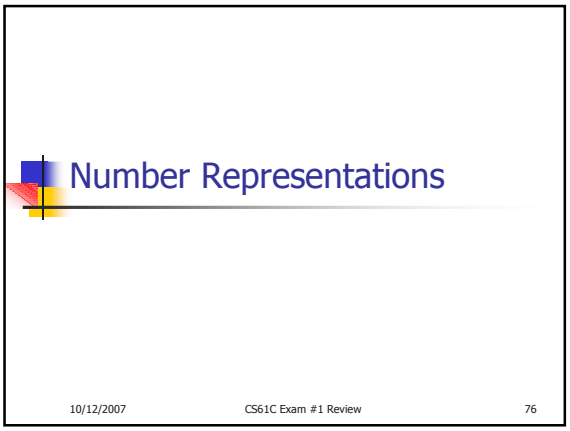
End of Prepared Slides

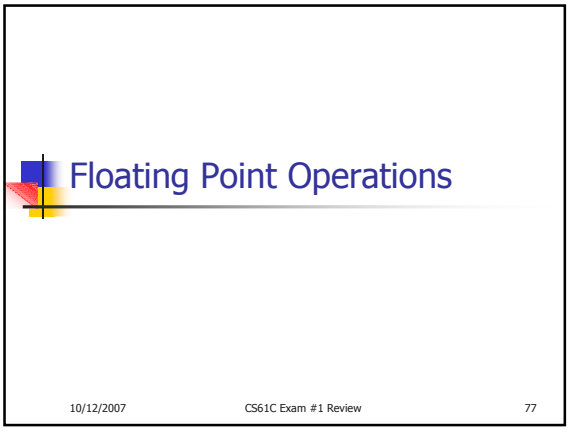
- Remaining Time
  - Dinner
  - Sp06 Problems
  - Basic review of floating point
- Suggestions for Studying
  - Go back over all your solutions/assignments
  - Double check EVERY wrong answer
  - Autograder logs should provide a starting place

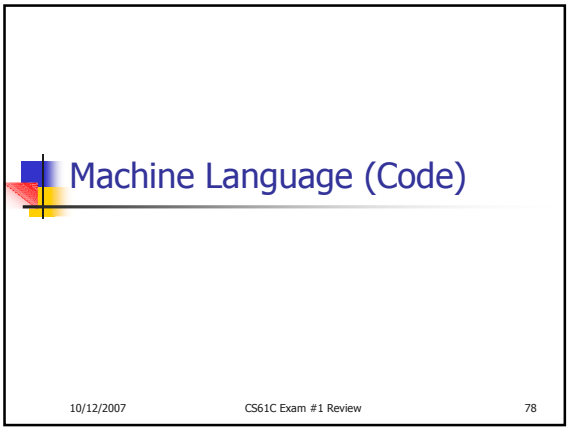

Control Flow
10/12/2007 CS61C Exam #1 Review 73


Calling Conventions
10/12/2007 CS61C Exam #1 Review 74


Logical & Bitwise Operators
10/12/2007 CS61C Exam #1 Review 75


Number Representations
10/12/2007 CS61C Exam #1 Review 76


Floating Point Operations
10/12/2007 CS61C Exam #1 Review 77


Machine Language (Code)
10/12/2007 CS61C Exam #1 Review 78

Machine vs. Assembly Language

Unsigned in MIPS

- Overloaded Term
  - It meaning is context dependant
    - What else in this class is like this?
  - A bad design practice (cf C static & extern)
- Meanings
  - Unsigned = Unsigned Integer
    - Multiplication, Division & Comparison
    - These operations must actually interpret bits differently
  - Unsigned = No overflow check
    - Addition, Subtraction
    - These operations result in identical bit patterns whether signed or unsigned
  - Unsigned = No sign extension
    - lb/lbu
- Unrelated
  - Add/subtract immediate **always** sign extend
  - Logical immediate operations **never** sign extend

MIPS Instruction Design

- Performance
  - CISC: Bad, RISC: Good, SBN: Very Bad
  - Why is the sweet spot in the middle at RISC?
  - So what?
- Load `0xDEADBEEF` into `$s0`
  - `lui $s0, 0xDEAD; ori $s0, $s0, 0xBEEF`
    - Any other ways to do this?
  - Why in two halves? Why not in one instruction?
    - Important point of design for RISC!
    - Relationship to stored program computer?
- Single Instruction: SBN (subtract branch if negative)
  - No need for registers, just use memory
  - "Universal Operator": NAND, Mux, etc…
- IA32/x86
  - 1-18 byte instructions (e.g. strcpy!)
  - But internally, executed as RISC

Belief & Debugging

- Belief: You believe you know what your program does
  - You think you understand it
  - You think you know what the library calls do
- Fact: You can read what it actually does
  - Computers are as close to perfect as possible
  - A computer error or fault is very unlikely
- Consequence
  - A mismatch means your beliefs are wrong
  - Always assume that you are **dead wrong**
    - It's possible the bug is a typo

Assignments

All assignment are interrelated!
New parts of the class build on old ones (unlike most classes)

Lab3 Vector Alternatives

- What we did
  - Explicitly store the size
  - NewSize = max(2*Size, loc+1)
  - Memset to zero out the new storage
  - Memcpy to copy other elements
- Other Options
  - Use EOF to mark end of the array
  - Ropes
  - Gapped circular array buffer

Quiz3

```
1 )  /*
2 )      Return the result of appending the characters in s2 to s1.
3 )      Assumption: enough space has been allocated for s1 to store
4 )      the extra characters.
5 ) */
6 ) char* append (char s1[ ], char s2[ ]) {
7 )      int s1len = strlen (s1);
8 )      int s2len = strlen (s2);
9 )      int k;
10)      for (k=0; k<=s2len; k++) {
11)          s1[k+s1len] = s2[k];
12)      }
13)      return s1;
14) }
```


Quiz4

```
0 ) #include <stdio.h>
1 ) struct point {
2 )      int x;
3 )      int y;
4 ) };
5 )
6 ) struct point* scanpoint() {
7 )      struct point *temp = new point;
8 )      scanf("%d %d", &(temp->x), &(temp->y));
9 )      return temp;
10) }
11)
12) void main() {
13)      struct point p = scanpoint();
14)      printf("%d %d", p->x, p->y);
15) }
```


Quiz5

- For each of the following kinds of data
  - List all possible storage locations
    - The Stack
    - The Heap
    - Static Storage
    - None of the above
  - Temporary variables
  - Function arguments
  - A global variable
  - A linked list
- What will `foo()` return?

```
char bar(int *p) { int b; return (&b < p) ? 't' : 'f'; }
char foo() { int a; return bar(&a); }
```


Quiz6

- Max & Min `int` on nova.cs.berkeley.edu
  - 2147483647, -2147483648
    - (2^31)-1 and -(2^31)
  - Nova is a 32bit machine
    - C int and unsigned int datatypes will be 32 bits
    - `printf("%u\n", sizeof(int));`
- Convert the unsigned binary value 10110010 to decimal
  - 10110010 = 178
- Convert the signed (twos complement) binary value 10110010 to decimal
  - 10110010 = -78
  - Signed will always means "twos complement" unless otherwise specified
  - Other answers
    - 50: sign magnitude
    - -178: who knows?
    - 178: thought for some reason it was a 32bit value (Why? I don't know…)

Quiz7

- Lines of code
  - Order the three languages from most to least
    - Java: very dense, thanks to extensive libs & language support
    - C: has libraries, but little language support
    - MIPS Assembly: generally no libraries, certainly no language support, simple commands
  - Question hints at the power of abstraction
- Assemble `k++`, where `k` is in `$s1`
  - `addiu?$s1.?$s1.?[0-9]*[1-9][0-9]*`
  - What's the constant added to k?

Quiz8

- Two instructions for "branch on less than"
  - `slti?,b(ne|eq)`
    - These are the four ways I know of to do this
    - Anyone know any more?
  - Why isn't it in MIPS?
- Load `0xDEADBEEF` into `$s0`
  - `lui $s0, 0xDEAD; ori $s0, $s0, 0xBEEF`
    - Any other ways to do this?
  - Why in two halves? Why not in one instruction?

Quiz10 Optimized

```
gcd: bne $a1,$0,recursive_case
                # if b == 0 …
     add $v0,$a0,$0 # gcd = a
     jr $ra
recursive_case:
     div $a0,$a1     # hi=a%b, lo=a/b
     mfhi $t0        # $t0 = hi
     add $a0,$a1,$0 # gcd = gcd (b, a%b)
     add $a1,$t0,$0
     j gcd
```


HW4Q2 Optimized

- C Code

```
int compare (int a, int b) {
  if (sub (a, b) >= 0) return 1;
  else return 0;
}

int sub (int a, int b) {
  return a-b;
}
```

- MIPS Code

```
slt $v0, $a1, $a0
jr $ra
```