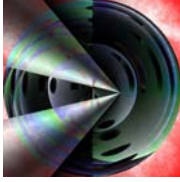


inst.eecs.berkeley.edu/~cs61c/su06  
**CS61C : Machine Structures**  
 Lecture #6: Intro to MIPS

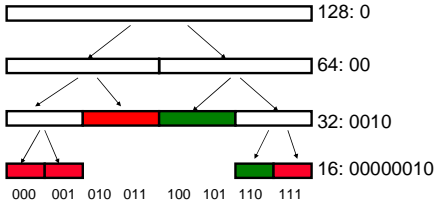


2006-07-06  
 Andy Carle

CS 61C L07 MIPS Intro (1) A Carle, Summer 2006 © UCB

**Buddy System Review**

• Legend: **FREE** **ALLOCATED** **SPLIT**



128: 0  
 64: 00  
 32: 0010  
 16: 00000010  
 000 001 010 011 100 101 110 111

Initial State → Free(001) → Free(000) → Free(111) → Malloc(16)

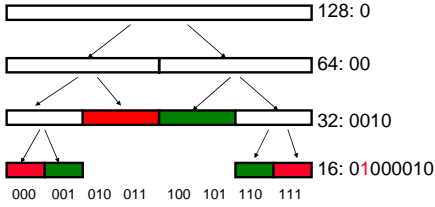
↑  
 1

Kudos to Kurt Meinz for these fine slides

CS 61C L07 MIPS Intro (2) A Carle, Summer 2006 © UCB

**Buddy System**

• Legend: **FREE** **ALLOCATED** **SPLIT**



128: 0  
 64: 00  
 32: 0010  
 16: 01000010  
 000 001 010 011 100 101 110 111

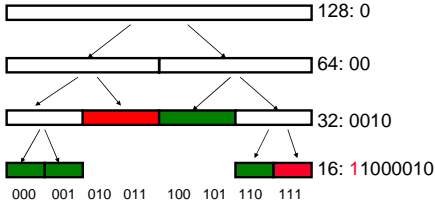
Initial State → Free(001) → Free(000) → Free(111) → Malloc(16)

↑  
 1

CS 61C L07 MIPS Intro (3) A Carle, Summer 2006 © UCB

**Buddy System**

• Legend: **FREE** **ALLOCATED** **SPLIT**



128: 0  
 64: 00  
 32: 0010  
 16: 11000010  
 000 001 010 011 100 101 110 111

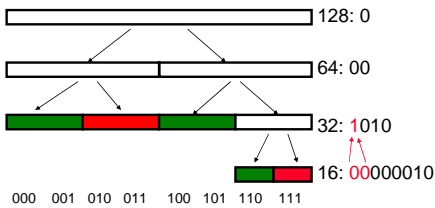
Initial State → Free(001) → Free(000) → Free(111) → Malloc(16)

↑  
 1

CS 61C L07 MIPS Intro (4) A Carle, Summer 2006 © UCB

**Buddy System**

• Legend: **FREE** **ALLOCATED** **SPLIT**



128: 0  
 64: 00  
 32: 1010  
 16: 00000010  
 000 001 010 011 100 101 110 111

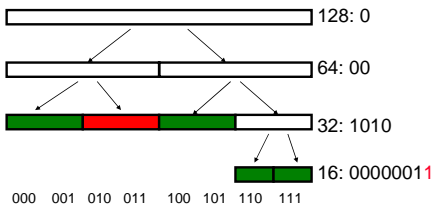
Initial State → Free(001) → Free(000) → Free(111) → Malloc(16)

↑  
 2

CS 61C L07 MIPS Intro (5) A Carle, Summer 2006 © UCB

**Buddy System**

• Legend: **FREE** **ALLOCATED** **SPLIT**

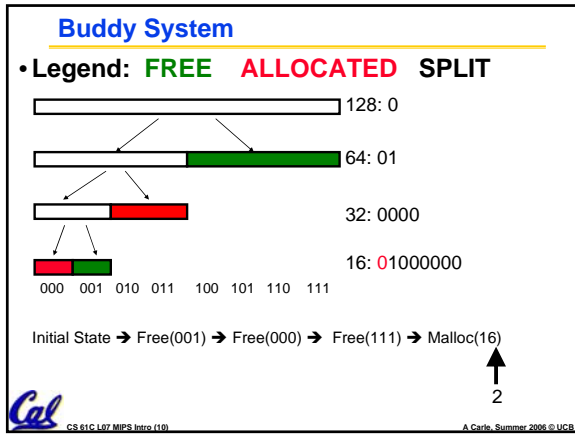
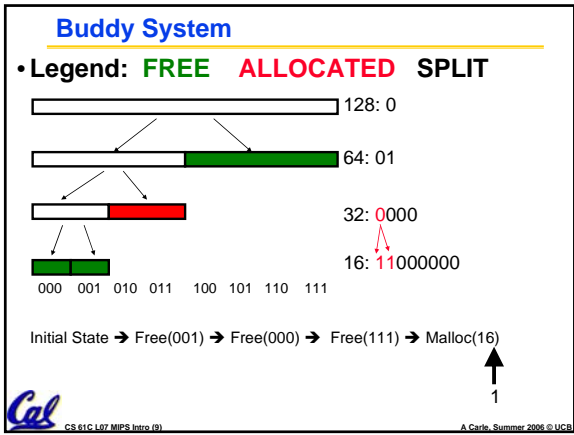
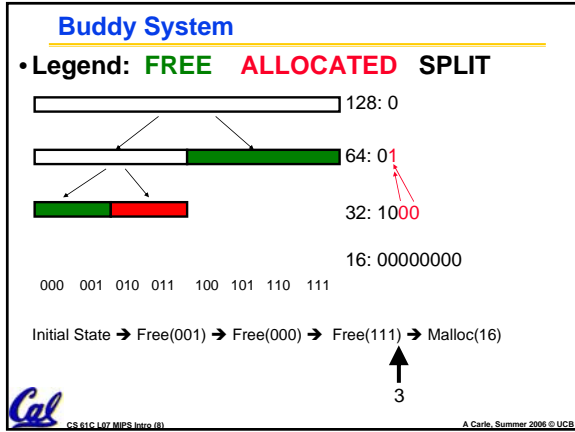
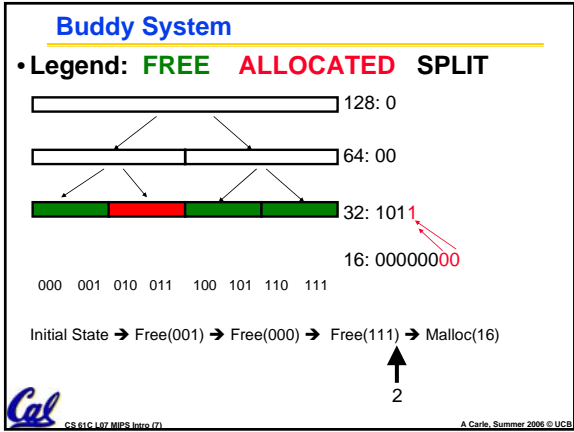


128: 0  
 64: 00  
 32: 1010  
 16: 00000011  
 000 001 010 011 100 101 110 111

Initial State → Free(001) → Free(000) → Free(111) → Malloc(16)

↑  
 1

CS 61C L07 MIPS Intro (6) A Carle, Summer 2006 © UCB



### Tracking Memory Usage

- Depends heavily on the programming language and compiler.
- Could have only a single type of dynamically allocated object in memory
  - E.g., simple Lisp/Scheme system with only cons cells (61A's Scheme not "simple")
- Could use a *strongly typed* language (e.g., Java)
  - Don't allow conversion (casting) between arbitrary types.
  - C/C++ are not strongly typed.

• Here are 3 schemes to collect garbage

CS 61C L07 MIPS Intro (11) A. Carls, Summer 2006 © UCB

### Scheme 1: Reference Counting

- For every chunk of dynamically allocated memory, keep a count of number of pointers that point to it.
- When the count reaches 0, reclaim.
- Simple assignment statements can result in a lot of work, since may update reference counts of many items

CS 61C L07 MIPS Intro (12) A. Carls, Summer 2006 © UCB

## Reference Counting Example

- For every chunk of dynamically allocated memory, keep a count of number of pointers that point to it.
- When the count reaches 0, reclaim.

```
int *p1, *p2;
p1 = malloc(sizeof(int));
p2 = malloc(sizeof(int));
*p1 = 10; *p2 = 20;
```

Reference count = 1 (green) for 20, Reference count = 1 (red) for 10



CS 61C L07 MIPS Intro (13)

A. Carls, Summer 2006 © UC Berkeley

## Reference Counting Example

- For every chunk of dynamically allocated memory, keep a count of number of pointers that point to it.
- When the count reaches 0, reclaim.

```
int *p1, *p2;
p1 = malloc(sizeof(int));
p2 = malloc(sizeof(int));
*p1 = 10; *p2 = 20;
p1 = p2;
```

Reference count = 2 (green) for 20, Reference count = 0 (red) for 10



CS 61C L07 MIPS Intro (14)

A. Carls, Summer 2006 © UC Berkeley

## Reference Counting (p1, p2 are pointers)

```
p1 = p2;
```

- Increment reference count for p2
- If p1 held a valid value, decrement its reference count
- If the reference count for p1 is now 0, reclaim the storage it points to.
  - If the storage pointed to by p1 held other pointers, decrement all of their reference counts, and so on...
- Must also decrement reference count when local variables cease to exist.



CS 61C L07 MIPS Intro (15)

A. Carls, Summer 2006 © UC Berkeley

## Reference Counting Flaws

- Extra overhead added to assignments, as well as ending a block of code.
- Does not work for circular structures!
  - E.g., doubly linked list:



CS 61C L07 MIPS Intro (16)

A. Carls, Summer 2006 © UC Berkeley

## Scheme 2: Mark and Sweep Garbage Col.

- Keep allocating new memory until memory is exhausted, then try to find unused memory.
- Consider objects in heap a graph, chunks of memory (objects) are graph nodes, pointers to memory are graph edges.
  - Edge from A to B => A stores pointer to B
- Can start with the root set, perform a graph traversal, find all usable memory!
- 2 Phases: (1) Mark used nodes; (2) Sweep free ones, returning list of free nodes



CS 61C L07 MIPS Intro (17)

A. Carls, Summer 2006 © UC Berkeley

## Mark and Sweep

- Graph traversal is relatively easy to implement recursively
 

```
void traverse(struct graph_node *node) {
    /* visit this node */
    foreach child in node->children {
        traverse(child);
    }
}
```
- But with recursion, state is stored on the execution stack.
  - Garbage collection is invoked when not much memory left
- As before, we could traverse in constant space (by reversing pointers)



CS 61C L07 MIPS Intro (18)

A. Carls, Summer 2006 © UC Berkeley

### Scheme 3: Copying Garbage Collection

- Divide memory into two spaces, only one in use at any time.
- When active space is exhausted, traverse the active space, copying all objects to the other space, then make the new space active and continue.
  - Only reachable objects are copied!
- Use “forwarding pointers” to keep consistency
  - Simple solution to avoiding having to have a table of old and new addresses, and to mark objects already copied (see bonus slides)



CS 61C L07 MIPS Intro (19)

A. Carls, Summer 2006 © UCB

### PRS Round 1

- Of {K&R, Slab, Buddy}, there is no best (it depends on the problem).
- Since automatic garbage collection can occur any time, it is **more difficult to measure the execution time** of a Java program vs. a C program.
- We don't have automatic garbage collection in C because of **efficiency**.



CS 61C L07 MIPS Intro (20)

A. Carls, Summer 2006 © UCB

### Review

- Several techniques for managing heap w/ malloc/free: best-, first-, next-fit, **slab, buddy**
- 2 types of memory fragmentation: **internal & external**; all suffer from some kind of frag.
- Each technique has strengths and weaknesses, **none is definitively best**
- Automatic memory management relieves programmer from managing memory.
  - All require help from language and compiler
  - **Reference Count**: not for circular structures
  - **Mark and Sweep**: complicated and slow, works
  - **Copying**: move active objects back and forth



CS 61C L07 MIPS Intro (21)

A. Carls, Summer 2006 © UCB

### New Topic!

#### MIPS Assembly Language



CS 61C L07 MIPS Intro (22)

A. Carls, Summer 2006 © UCB

### Assembly Language

- Basic job of a CPU: execute lots of **instructions**.
- Instructions are the primitive operations that the CPU may execute.
- Different CPUs implement different sets of instructions. The set of instructions a particular CPU implements is an **Instruction Set Architecture (ISA)**.
  - Examples: Intel 80x86 (Pentium 4), IBM/Motorola PowerPC (Macintosh), MIPS, Intel IA64, ...



CS 61C L07 MIPS Intro (23)

A. Carls, Summer 2006 © UCB

### 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.



CS 61C L07 MIPS Intro (24)

A. Carls, Summer 2006 © UCB

## ISA Design

- Must Run Fast In Hardware → Eliminate sources of complexity.

### Software

- Symbolic Lookup → fixed var names/#
- Strong typing → No Typing
- Nested expressions → Fixed format Inst
- Many operators → small set of insts

### Hardware

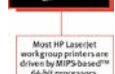


CS 61C L07 MIPS Intro (26)

A. Carls, Summer 2006 © UCB

## MIPS Architecture

- MIPS – semiconductor company that built one of the first commercial RISC architectures
- We will study the MIPS architecture in some detail in this class (also used in upper division courses CS 152, 162, 164)
- Why MIPS instead of Intel 80x86?
  - MIPS is simple, elegant. Don't want to get bogged down in gritty details.
  - MIPS widely used in embedded apps, x86 little used in embedded, and more embedded computers than PCs



CS 61C L07 MIPS Intro (26)

A. Carls, Summer 2006 © UCB

## Assembly Variables: Registers (1/4)

- Unlike HLL like C or Java, assembly cannot use variables
  - Why not? Keep Hardware Simple
- Assembly Operands are **registers**
  - limited number of special locations built directly into the hardware
  - operations can only be performed on these!
- Benefit: Since registers are directly in hardware, they are very fast (faster than 1 billionth of a second)



CS 61C L07 MIPS Intro (27)

A. Carls, Summer 2006 © UCB

## Assembly Variables: Registers (2/4)

- Drawback: Since registers are in hardware, there are a predetermined number of them
  - Solution: MIPS code must be very carefully put together to efficiently use registers
- 32 registers in MIPS
  - Why just 32? **Smaller is faster**
- Each MIPS register is 32 bits wide
  - Groups of 32 bits called a **word** in MIPS



CS 61C L07 MIPS Intro (28)

A. Carls, Summer 2006 © UCB

## Assembly Variables: Registers (3/4)

- Registers are numbered from 0 to 31
- Each register can be referred to by number or name
- Number references:  
\$0, \$1, \$2, ... \$30, \$31



CS 61C L07 MIPS Intro (29)

A. Carls, Summer 2006 © UCB

## Assembly Variables: Registers (4/4)

- By convention, each register also has a name to make it easier to code
- For now:
  - \$16 - \$23 → \$s0 - \$s7  
(correspond to C variables)
  - \$8 - \$15 → \$t0 - \$t7  
(correspond to temporary variables)
  - Later will explain other 16 register names
- In general, use names to make your code more readable



CS 61C L07 MIPS Intro (30)

A. Carls, Summer 2006 © UCB

## C, Java variables vs. registers

- In C (and most High Level Languages) variables declared first and given a type
  - Example:

```
int fahr, celsius;
char a, b, c, d, e;
```
- Each variable can ONLY represent a value of the type it was declared as (cannot mix and match int and char variables).
- In Assembly Language, the registers have no type; operation determines how register contents are treated



CS 61C L07 MIPS Intro (31)

A. Carls, Summer 2006 © UC Berkeley

## Comments in Assembly

- Another way to make your code more readable: comments!
- Hash (#) is used for MIPS comments
  - anything from hash mark to end of line is a comment and will be ignored
- Note: Different from C.
  - C comments have format

```
/* comment */
```

so they can span many lines



CS 61C L07 MIPS Intro (32)

A. Carls, Summer 2006 © UC Berkeley

## Assembly Instructions

- In assembly language, each statement (called an **instruction**), executes exactly one of a short list of simple commands
- Unlike in C (and most other High Level Languages), each line of assembly code contains at most 1 instruction
- Instructions are related to operations (=, +, -, \*, /) in C or Java



CS 61C L07 MIPS Intro (33)

A. Carls, Summer 2006 © UC Berkeley

## Administrivia

- Office Hours:
- HW3 Due Monday
- Proj1 Due 7-16
- Midterm 1:
  - Friday, 7/14
  - Probably 11 – 2
  - Room TBD



CS 61C L07 MIPS Intro (34)

A. Carls, Summer 2006 © UC Berkeley

## MIPS Addition and Subtraction (1/4)

- Syntax of Instructions:

```
<op> <dest> <src1> <src2>
```

where:
  - op) operation by name
  - dest) operand getting result (“destination”)
  - src1) 1st operand for operation (“source1”)
  - src2) 2nd operand for operation (“source2”)
- Syntax is rigid:
  - 1 operator, 3 operands
  - Why? **Keep Hardware simple via regularity**



CS 61C L07 MIPS Intro (35)

A. Carls, Summer 2006 © UC Berkeley

## Addition and Subtraction of Integers (2/4)

- Addition in Assembly
  - Example: `add $s0, $s1, $s2` (in MIPS)  
Equivalent to: `s0 = s1 + s2` (in C)  
where MIPS registers `$s0, $s1, $s2` are associated with C variables `s0, s1, s2`
- Subtraction in Assembly
  - Example: `sub $s3, $s4, $s5` (in MIPS)  
Equivalent to: `d = e - f` (in C)  
where MIPS registers `$s3, $s4, $s5` are associated with C variables `d, e, f`



CS 61C L07 MIPS Intro (36)

A. Carls, Summer 2006 © UC Berkeley

### Addition and Subtraction of Integers (3/4)

- How does the following C statement?

```
a = b + c + d - e;
```

- Break into multiple instructions

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

- Notice: A single line of C may break up into several lines of MIPS.

- Notice: Everything after the hash mark on each line is ignored (comments)



CS 61C L07 MIPS Intro (07)

A. Carls, Summer 2006 © UCB

### Addition and Subtraction of Integers (4/4)

- How do we do this?

```
f = (g + h) - (i + j);
```

- Use intermediate temporary register

```
add $t0, $s1, $s2 # temp = g + h
add $t1, $s3, $s4 # temp = i + j
sub $s0, $t0, $t1 # f=(g+h)-(i+j)
```



CS 61C L07 MIPS Intro (08)

A. Carls, Summer 2006 © UCB

### Immediates

- Immediates are numerical constants.
- They appear often in code, so there are special instructions for them.
- Add Immediate:

```
addi $s0, $s1, 10 (in MIPS)
f = g + 10 (in C)
```

where MIPS registers  $\$s0, \$s1$  are associated with C variables  $f, g$

- Syntax similar to add instruction, except that last argument is a number instead of a register.



CS 61C L07 MIPS Intro (09)

A. Carls, Summer 2006 © UCB

### Immediates

- There is no Subtract Immediate in MIPS: Why?
- Limit types of operations that can be done to absolute minimum
  - if an operation can be decomposed into a simpler operation, don't include it
  - `addi ..., -X = subi ..., X =>` so no `subi`
- `addi $s0, $s1, -10 (in MIPS)`  
`f = g - 10 (in C)`  
where MIPS registers  $\$s0, \$s1$  are associated with C variables  $f, g$



CS 61C L07 MIPS Intro (49)

A. Carls, Summer 2006 © UCB

### Register Zero

- One particular immediate, the number zero (0), appears very often in code.
- So we define register zero ( $\$0$  or  $\$zero$ ) to always have the value 0; eg  
`add $s0, $s1, $zero (in MIPS)`  
`f = g (in C)`  
where MIPS registers  $\$s0, \$s1$  are associated with C variables  $f, g$
- defined in hardware, so an instruction  
`add $zero, $zero, $s0`



will not do anything!

CS 61C L07 MIPS Intro (11)

A. Carls, Summer 2006 © UCB

### Peer Instruction Round 2

- Types are associated with declaration in C (normally), but are associated with instruction (operator) in MIPS.
- Since there are only 8 local ( $\$s$ ) and 8 temp ( $\$t$ ) variables, we can't write MIPS for C exprs that contain > 16 vars.
- If  $p$  (stored in  $\$s0$ ) were a pointer to an array of ints, then `p++` would be `addi $s0 $s0 1`



CS 61C L07 MIPS Intro (42)

A. Carls, Summer 2006 © UCB

## “And in Conclusion...”

- **In MIPS Assembly Language:**
  - Registers replace C variables
  - One Instruction (simple operation) per line
  - Simpler is Better
  - Smaller is Faster
- **New Instructions:**  
add, addi, sub
- **New Registers:**
  - C Variables: \$s0 - \$s7
  - Temporary Variables: \$t0 - \$t9
  - Zero: \$zero



CS 61C:197 MIPS Intro (43)

A. Chien, Summer 2006 ©UCB