1. Answers to Last Time's Problems

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
      int value; // or char or whatever storage you want
      struct node * next;
};
typedef struct node nodePtr;

int main()
{
      int i=0;
      nodePtr* a = (nodePtr*) malloc (sizeof(nodePtr));
      nodePtr* b = (nodePtr*) malloc (sizeof(nodePtr));
      nodePtr* walker;

      a->value = 0;
      b->value =3;
      a->next = b;
      b->next = NULL;

      for(walker = a; walker != NULL; walker = walker->next)
            printf ("%d\n", walker->value);

      return 0;
}
```

2. SLAB Allocator and Buddy System

To avoid external fragmentation due to small chunks of memory lying free between bigger slices of allocated space, the SLAB allocator preallocates memory into small predefined sizes (16kb, 32kb, etc.).

The size of the slab is also a tweakable parameter which specifies the largest memory request that uses SLAB; all larger cases use the regular best fit/first fit/next fit method we've seen. Whenever a request comes that is small enough to be handled by SLAB, an entire chunk is marked as used and returned, even if it's bigger than what was requested. This causes internal fragmentation, or waste of space within a chunk, but allows for faster and more efficient use and mallocing of small spaces.

The buddy system is similar to SLAB in that small amounts of memory are pre-allocated, except that they are marked so that certain contiguous same-sized blocks can be joined or broken to form larger or smaller chunks depending on demand. If a free chunk just bigger than (to the closest power of 2) the requested amount is found, that chunk is returned. However, if there is nothing available, a larger chunk can be split into smaller ones via the buddy system. This allows even more efficient usage than SLAB but takes longer because checks must be done to join/split buddies.

3. Garbage Collection

C does not support garbage collection because it is weakly typed. This means that though variables have types, they can be interpreted bitwise as something different. For example, int, float, and char * are all 4 bytes wide in a 32 bit system. How does the system know (when blindly trekking through memory) whether the 4 bytes it encounters is a pointer? With no way to distinguish, there can be no garbage collection in C.

In Java and interpreted languages, there are 3 main methods to do garbage collection:

Method	Description	Problems
Reference Counting	Have a few bits attached to each malloc that stores how many things point to it. Increment and decrement those bits accordingly when pointers are changed or set. When the bits reach 0 for an allocated chunk, free it.	Can generate massive waits when removing the head of a linked list. Doesn't deal with doubly linked lists.
Mark and Sweep	Have a tag bit for every section of allocated memory. Go through all the pointers and the nodes they connect. Tag/mark everything that's accessible. Free the rest.	Very slow! Program stalls to clean tagged memory all at once.
Copying GC	Use half of memory. When one side gets fully, pick up the reachable stuff and move to other half. Compact to remove fragmentation.	Can only use half of memory! Has very bad performance in edge cases (e.g. almost full).

Java and C# use a hybrid technique that combines elements of the above with Generational GC.

5. Intro to MIPS

MIPS (Microprocessor without Interlocked Pipeline Stages) is a CPU architecture that was pioneered back in the early 80s @ Stanford. It went against the dogma at the time of complex computer architectures (CISC) that had more and more exotic instructions. x86 is a good example of MIPS's nemesis – it became bloated over time as more and more extensions were added (SSE, SSE2, MMX, 3D Now!, etc.). From the name, you can tell that "interlocked pipelines" was the colossal beast of the time, which MIPS desperately wanted to slay.

MIPS wanted a clean and simple instruction set architecture (ISA) that was easy to implement in hardware, reasoning that the transistors saved in implementing logic can be used on more execution resource and cache. As testament to the degree of its success, CPUs today are predominantly made with RISC cores.

- **MIPS Registers**

The MIPS design has 32 registers, each of which can store 32 bites of information. Registers are like hands for the CPU (and are physically located on-die), which can only work with the data stored in registers. Everything else must be stashed in memory and retrieved as required. The reason for this is that instructions are conveniently also 32 bits wide. A summary of all 32 registers is given below:

Name	Number	Function
$zero	$0	Defines a constant zero
$at	$1	Used by system
$v0-$v1	$2–$3	Function returns and expression evaluations
$a0-$a3	$4–$7	Function arguments
$t0-$t7	$8–$15	Temporary storage (can change after system call)
$s0-$s7	$16–$23	Long-term static storage (guaranteed not to change after a system/function call)
$t8-$t9	$24–$25	More temporary storage
$k0-$k1	$26–$27	Reserved
$gp	$28	Global pointer – reserved
$sp	$29	Stack pointer – don't modify
$fp	$30	Frame pointer – don't modify
$ra	$31	Return address

Most of your work will be dealing with $t0-$t9, $s0-$s7, and $sp.

- **MIPS instructions**

The format of most instructions is:

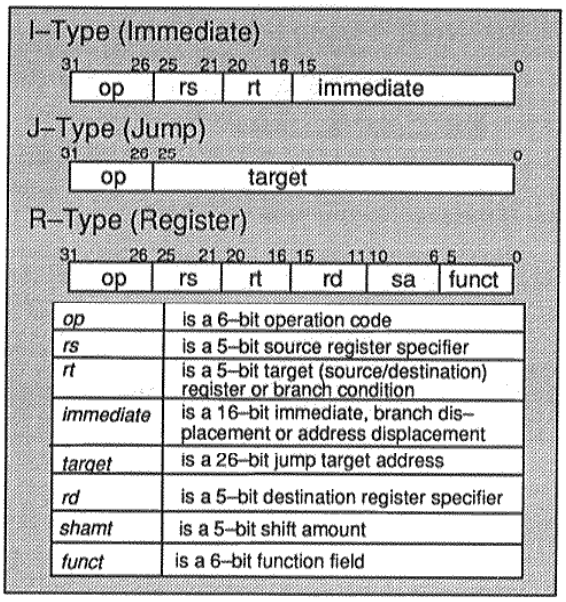
```
name $destination, $source1, $source2
```

Some basic instructions you should have covered in lecture:

Instruction	Type	Function	Notes
add $dest, $source1, $source2	R	$dest = $source1 + $source2	
addi $dest, $source1, CONST	I	$dest = $source1 + CONST	CONST can be negative
sub $dest, $source1, $source2	R	$dest = $source1 - $source2	
subu $dest, $source1, $source2	R	$dest = $source1 - (unsigned int) $source2	$source2 is unsigned
addu $dest, $source1, $source2	R	$dest = $source1 + (unsigned int) $source2	$source2 is unsigned
addiu $dest, $source1, CONST	I	$dest = $source1 + (unsigned int) CONST	CONST is always pos
lw $dest, OFFSET($addr)	I	$dest = (int) *($addr+OFFSET)	Must be word aligned
sw $source, OFFSET($addr)	I	*($addr+OFFSET) = (int)$dest	Must be word aligned
lb $dest, OFFSET($addr)	I	$dest = (char) *($addr+OFFSET)	Sign extended
lbu $dest, OFFSET($addr)	I	$dest = (unsigned char) *($addr+OFFSET)	Not sign extended
sb $source, OFFSET($addr)	I	*($addr+OFFSET) = (char)$dest	Sign extended

beq $reg1, $reg2, LABEL	I	if ($reg1 == $reg2) goto LABEL	
bne $reg1, $reg2, LABEL	I	if ($reg1 != $reg2) goto LABEL	
j LABEL	J	goto LABEL	

There is no subiu instruction. How can you imitate it by using others?



6. Solved Problems

Translate the following C into MIPS:

```
// assume ptr is the address of a0[4]
// num → $s0, num2 → $s1, dest → $s2

int num = *(ptr);
char num2 = (char) (ptr[8] >> 24);
if (num == num2)     dest = num + num2;
else                 dest = num - num2;
```

```
lw $s0, 16($a0)
lbu $s1, 48($a0)
beq $s0, $s1, equal
subu $s2, $s0, $s1
j end
equal: addu $s2, $s0, $s1
end:
```

```
// Nth_Fibonacci(N)
// N → $s0, fib → $s1
// i → $t0, j → $t1
if(N==0) return 0;
else if(N==1) return 1;

N-=2;
int fib=1, i=1, j=1;
while(N!=0)
{
      fib = i+j;
      j = i;
```

```
zero: bne $s0 $0 one
      addi $s1 $0 0
      j done
one:  addi $s1 $0 1
      bne $s0 $s1 init
      j done
init: addi $s0 $s0 -2
      addi $t0 $0 1
      addi $t1 $0 1
loop: beq $s0 $0 done
      add $s1 $t0 $t1
      addi $t1 $t0 0
```

```
      i = fib;
      N--;
}
return fib;
```

```
      addi $t0 $s1 0
      subi $s0 $s0 1
      j loop
done:
```