This test has six problems worth a total value of 100 points. Note that problems have different values. Be sure to read over all of the problems carefully before answering them. A little thought will help you avoid making careless errors.

You may use your class texts, notes, handouts, class newsgroups postings and listings of your assignments. You may not use any kind of computing devices on this examination.

Please Do Not Detach This Page.

Name ____________________________ Key ____________________________

ID Number ____________________________

Please do not write below this line.

1. _____ 10 _____ / 10
2. _____ 20 _____ / 20
3. _____ 20 _____ / 20
4. _____ 10 _____ / 10
5. _____ 20 _____ / 20
6. _____ 20 _____ / 20

Total _____ 100 _____ / 100
1. (10 points) (a) Explain the difference between an assembly language instruction and a directive. (b) Give an example of each.

An assembly language instruction translates directly into a binary machine language instruction for the processor. An example would be a jump instruction. A directive is a command to the assembler to perform some action. This action is something other than a direct translation into a machine instruction. An example of this would be a directive telling the assembler to allocate some number of words of space, or one telling the assembler to start assembling this program at an LC of 0.

2. (20 points) (a) Show the output of the following program. (b) What output would you have gotten if i_u had been unsigned short?

```
#include <stdio.h>

main() {

    short /* Assume shorts are two bytes */
    i_u;

    long /* Assume longs are four bytes */
    l_a,
    l_b;

    i_u = 0xFFFF;
    printf("i_u: %d\n", i_u);
    l_a = (long)i_u;
    printf("l_a: %ld\n", l_a);
    l_b = (long)(unsigned) i_u;
    printf("l_b: %ld\n", l_b);
}
```

(a):

```
i_u: -1
l_a: -1
l_b: 65535
```

(b):

```
i_u: -1
l_c: 65535
l_d: 65535
```
3. (20 points) Write a function with the following prototype:

```c
int strcat_size(char *a, char *b);
```

where the contents of the string `b` are appended to string `a`. The size of the resulting string (not including the ‘\0’) is returned by the function. Assume that storage has already been allocated for the strings. You do not need to worry about space limitations in the string. You may not use the `strlen()`, `strcpy()` or related standard library functions.

```c
int strcat_size(char *a, char *b) {
    int i_count = 0;

    /* Move to the end of the first string. */
    while (*a) {
        /* Only move the pointer until we see the null, which 
           we don’t move past */
        a++;
        i_count++;
    }

    /* Write the second string onto the end of the first */
    while (*a++ = *b++)
        i_count++;

    /* Return the length of the string you’ve created */
    return i_count;
}
```
4. (10 points) A program in a RISC assembly language, such as MIPS, will generally take more instructions than the same program written in a CISC assembly language, such as the VAX. (a) Why is this? (b) Why is this not really considered a disadvantage for RISC computers?

(a) The programs are generally longer because RISC language machine instructions are simpler than their CISC counterparts, and therefore it takes more RISC instructions to perform a given task.

(b) This is generally considered not to be a problem since the RISC computers can more easily designed to be faster, so that they execute at least as quickly, even though they are executing more instructions.

5. (20 points) Give the register and memory locations, and their values, that the following MIPS machine code sequence will change.

24030040
34640008
AC640000

R3: 00000040
R4: 00000048
00000040: 00000048
6. (20 points) Write the functions `push()` and `pop()` corresponding to the prototypes and behavior given in the program below. `push()` must dynamically allocate a new stack element using `malloc()`. Both functions take the pointer to the present stack top as an argument, and return a pointer to the new stack top.

```c
struct stack_elm * push(struct stack_elm *top, int top_val) {
    struct stack_elm *new_elm;

    /* Malloc the new node and put it onto the stack */
    new_elm = (struct stack_elm *)malloc(sizeof(struct stack_elm));

    new_elm->elm_val = top_val;
    new_elm->next = top;

    /* Return this new element as the stack top */
    return new_elm;
}

struct stack_elm *pop(struct stack_elm *top, int *top_val) {
    struct stack_elm *new_top;

    /* Get the value of the old top */
    *top_val = top->elm_val;

    /* Get the new stack top */
    new_top = top->next;

    /* Free up the old element */
    free(top);

    /* Return the top of the new stack */
    return new_top;
}
```