1 Overview

Your assignment is to write a Mips disassembler. This is a program that takes a 32-bit integer as input, decodes it as a Mips instruction, and prints out the resulting assembly code.

2 Assignment

/home/cs340/prog4/dis.c contains the following template:

```c
int mask[] = {MASK};
int value[] = {VALUE};
int class[] = {CLASS};
char* instr[] = {NAME};
char* regs[] = {REGS};
int test[] = {0x210aff}; // add $10, $8, -1

int signExtend(int x, int size) {
    int s = x >> (size-1);
    if (s)
        x |= (-1)<<((sizeof(int)*8-size);
    return x;
}

    // ADD YOUR OWN FUNCTIONS HERE!

void disassemble(int x) {
    // ADD YOUR OWN CODE HERE!
}

int dis (int code[], int length) {
    int i;
    for(i=0; i<length; i++)
        disassemble(code[i]);
}

int main () {
    dis(test,sizeof(test)/sizeof(int));
}
```
You should add code to

```c
void disassemble(int x) {
}
```

that decodes `x` as a Mips instruction and prints out the resulting assembly code. For example, the call

```c
disassemble(0x0000000c)
```

should print out "syscall" and

```c
disassemble(0x210affff)
```

should print out "addi $t2,$t0,-1".

### 3 Algorithm

To disassemble an instruction-word `X` we need to do three things:

1. We must **classify** `X`, i.e. determine if `X` represents an add-instruction, a beq-instruction, etc.
2. We must **extract** the arguments from `X`. For example, if `X` represents an add-instruction, we must extract the three registers in the instruction.
3. We must **format** the disassembled instruction and print it out. For example, if we have classified `X` as a **sw** instruction and have extracted the register to be stored as `$a0`, the offset as `-24` and the indirect-register as `$sp` then we should format the instruction as

```c
sw $a0,-24($sp).
```

### 3.1 Classification

The header-file `mips.h` contains enough information about the Mips ISA to be able to classify every instruction. For example, the entry for **syscall** looks like this:

```c
// 000000 00000 00000 00000 00000 001100 syscall
#define syscall_MASK 0xffffffff
#define syscall_VALUE 0xc
#define syscall_CLASS 0
#define syscall_NAME "syscall"
```

The values in `mips.h` are collected together into four arrays in `dis.c`: `mask[]`, `value[]`, `class[]`, and `instr[]`. The `i`th entry in these arrays (`mask[i]`, `value[i]`, `class[i]`, and `instr[i]`) contains all the information necessary to disassemble the `i`th-instruction.
For example, the 0:th instruction ("syscall") have the entries

\[
\begin{align*}
\text{mask}[0] &= 0xffffffff \\
\text{value}[0] &= 0xc \\
\text{class}[0] &= 0 \\
\text{instr}[0] &= "syscall"
\end{align*}
\]

mask[i] contains a "1" in every place where the instruction encoding for the instruction has a "0" or a "1". In other words, mask[i] points out to us which bits in an instruction can be used to identify it. In the case of syscall, mask is 0xffffffff (which is 111111111111111111111111111111112 in binary) since all the bits in the encoding of syscall should be used for classification.

value[i] is the hex value of those identifying bits, in the case of syscall 0xc (binary 1100). This means that doing a bitwise-and with mask[i] and an instruction word X and then comparing the result to value[i] is enough to tell us if X is an i-instruction.

If it is, the opcode of the instruction can be found in instr[i].

class[i] divides the instruction set into groups, such that all instructions with the same class-number are decoded the same. There are 13 groups.

### 3.2 Argument Extraction

The entry for mfhi is:

\[
\begin{align*}
\text{mask}[1] &= 0xffffff7ff \\
\text{value}[1] &= 0x10 \\
\text{class}[1] &= 1 \\
\text{instr}[1] &= "mfhi"
\end{align*}
\]

The mask is '1111 1111 1111 0000 0111 1111 1111' (binary for 0xffffff7ff). This tells us that bits 11-15 should not be used to classify mfhi, since they contain a register number:

\[
000000 00000 00000 rd 00000 010000 mfhi
\]

Instead we should extract bits 11-15 from the instruction word and format them as a register name. The name of register number \( r \) is in \( \text{regs}[r] \).

### 3.3 Bit-Manipulation in C

C has several useful operators that manipulate bits in a word. \( \text{word} \ll s \) shifts \( \text{word} \) (which should be an integer) \( s \) steps to the left. \( \text{word} \gg s \) shifts \( \text{word} \) \( s \) steps to the right. \( \text{word1} \& \text{word2} \) performs the bitwise-and on two words, and \( \text{word1} | \text{word2} \) performs the bitwise-or. "\( \text{word} \) complements the \( \text{word} \), i.e. turns all 0's to 1's and vice versa."
4 Development Strategy

As always, it’s a good idea to develop the program in stages. For example, you could start by writing a function class0(x) which formats and prints an instruction belonging to class 0. This is easy, since there’s only one such instruction (syscall). Next write a function class1(x) that handles mfhi and mflo. When these work satisfactorily go back and code disassemble(x) which uses mask and value to classify an instruction. When disassemble has classified an instruction it can call on class0 and class1 to format and print it out. When you’ve gotten this far your program is almost done: all that is left to do is to write additional classi functions to decode the remaining classes of instructions.

5 Extra Credit

For extra credit, create symbolic labels for branch instructions. That is, instead of generating the following code

```
beq   $zero,$at,12
add  $t0,$a1,$a2
srl $t0,$t0,1
addi $a1,$t0,1
lui  $at,$zero,4097
```

you should generate

```
beq   $zero,$at,Lab1
add  $t0,$a1,$a2
srl $t0,$t0,1
addi $a1,$t0,1
Labi:
lui  $at,$zero,4097
```

You should be able to handle both forward and backwards branches.

6 Honors Section

You should do the extension for extra credit.

7 Turnin

When you have completed the program, submit your dis.c file by typing make turnin. You may turn in as many times as you want; turnin will always replace the previously turned-in version with the new version.